Teknisk dokumentation

Redaktör: Hanna Müller
Robin Christensen
Måns Gezelius
Mattias Hammar
Henrik Henriksson
Elin Petersén

Version 1.1

Status

Granskad		
Godkänd		

PROJEKTIDENTITET

2015/HT, Grupp 13
Linköpings Tekniska Högskola, MAI

Gruppdeltagare

Namn	**Ansvar**	**Telefon**	**E-post**
Henrik Henriksson	Projektledare (PL)		
Hanna Müller	Dokumentansvarig (DOK)		
Mattias Hammar			
Elin Petersén			
Måns Gezelius	Tidsansvarig (TID)		
Robin Christensen			

E-postlista för hela gruppen: kmm2015@lists.lysator.liu.se
Hemsida: https://bitbucket.org/henrixh/tsea29

Kursansvarig: Tomas Svensson
Handledare: Anders Nilsson

Innehåll

Figurer

Tabeller

1 Inledning

Denna dokumentation syftar till att fungera som konstruktionsunderlag samt som utgångspunkt för underhåll och felsökning av både hård- och mjukvara till den robot som utvecklats av projektgrupp 13 som en del i kursen *TSEA29 - Konstruktion med mikrodatorer* vid Linköpings universitet. Kursen ska ge erfarenhet av praktisk elektronikkonstruktion både på det tekniska och administrativa planet. Roboten ska klara av att autonomt navigera genom en labyrint samt delta i tävlingar i labyrintnavigering, slalom och dragrace.

1.1 Parter

Beställare av projektet var Tomas Svensson, ISY LiTH. Anders Nilsson har under projektets gång agerat handledare. Projektet utfördes av en projektgrupp bestående av sex studenter vid Linköpings universitet, se projektidentitet.

1.2 Disposition

Inledningsvis presenteras den generella strukturen på det integrerade systemet där läsaren kan få en översikt av de olika moduler som systemet inkorporerar och hur de interagerar med varandra. Detta följs av en mer ingående beskrivning av designen för vardera modul. I appendix A visas kopplingsscheman över enheten. I appendix B listas de program som systemet använder sig av. Detta är tänkt att fungerar som komplement till den källkod som finns dokumenterad elektroniskt. I appendix C finns de kommunikationsprotokoll som används vid överföring av data mellan mikrokontroller och BeagleBoard bekrivna.

2 Produkten

Roboten levereras med alla komponenter färdigmonterade på chassit *Trossenrobotics Hexapod*[7], se figur 1. Modulerna är programmerade med respektive mjukvara enligt appendix B. Utöver roboten leveras även ett tillhörande datorprogram som kan köras på valfri dator med operativsystemet Linux. Programmet kommunicerar med roboten via bluetooth och kan skicka kommandon till roboten samt ta emot data från roboten som sedan visas i grafer. Vidare levereras roboten med användarhandledning och teknisk dokumentation.

Figur 1: Bild på produkten

3 Teori

3.1 AI

Robotens autonoma beslutsdel har i uppgift att ta in sensordata, analysera hur dess omgivning ser ut, fatta beslut om vilken rörelse som ska utföras, och sist skicka resultatet till regleralgoritmen som finputsar kommandot. Det finns fem typer av svängar och korsningar som kan stötas på i en labyrint, beskrivna i figur 2.

Till en början evalueras alla avståndssensorer för vilket avstånd de befinner sig från en vägg. Roboten vet hur långt det borde vara till en vägg nära, medelnära, samt långt bort under optimala förhållanden, och utefter dessa värden avrundas sensorvärdena till kort/medel/lång. Detta förenklar beslutshanteringen markant.

Programmet befinner sig i olika tillstånd för att kunna avgöra vilken typ av rörelse som är aktiv, samt vilka nya beslut som kan fattas utifrån tillståndet. Nedan är alla tillstånd beskrivna.

Evaluera korsning Här fattas beslut om vilken typ av korsning eller läge roboten befinner sig i, samt vilken rörelsetyp som krävs. Här hanteras även vinst, återvändsgränd, eller okänd korsning vid sensorfel.

Gå framåt Låt roboten gå rakt fram tills det finns en vägg framåt, eller någon av sidorna är öppna.

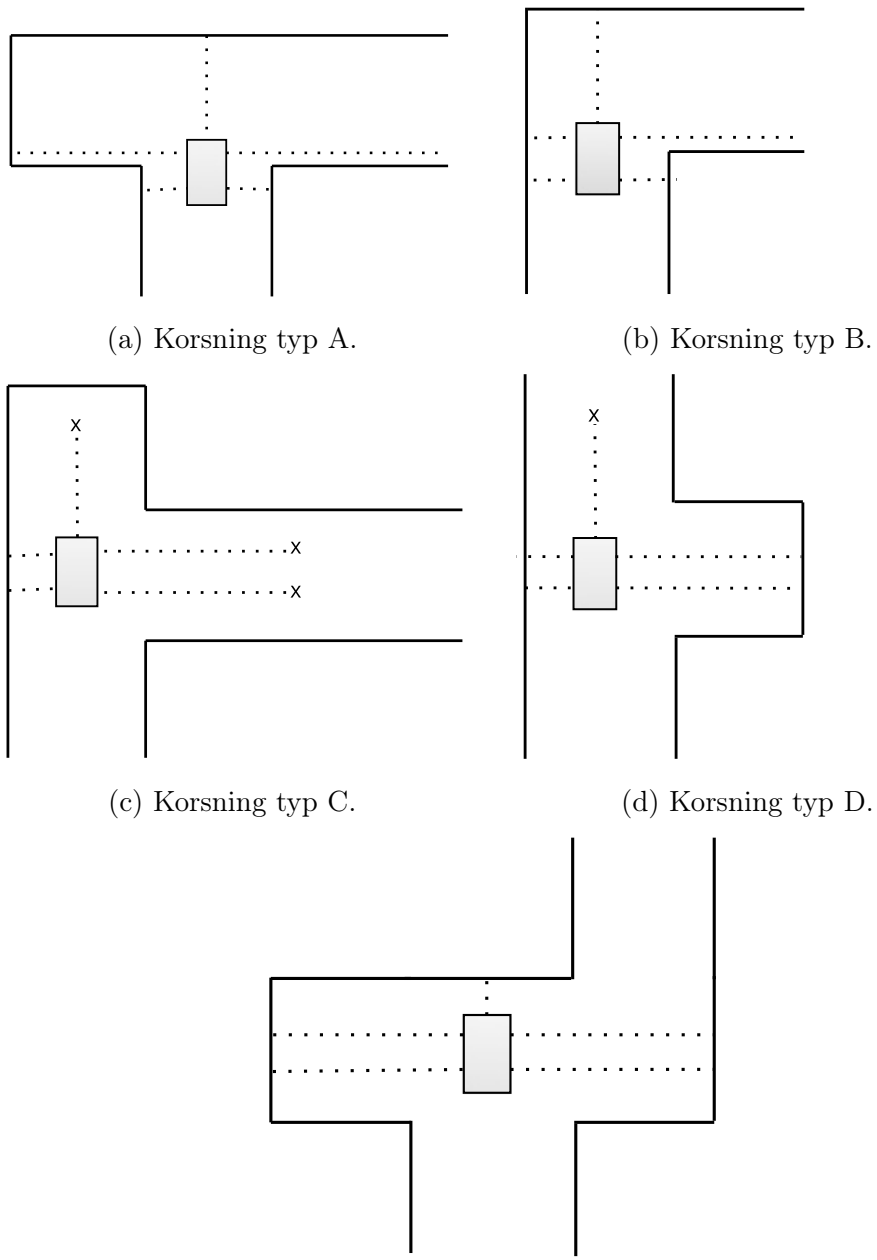
Gå i sidled Sidogång används enbart i korsning E enligt figur 2. Roboten byter håll då det befinner sig en vägg åt sidan. Om det inte finns en vägg framåt är det dit roboten ska gå.

Steg-rotera Rotera roboten en bestämd vinkel. Efter rotation ska roboten stega framåt.

Stega framåt Gå framåt en bestämd längd utan att titta åt sidorna. Om fram-sensorn är kort avbryts gången. Detta läge hjälper roboten att stega till mitten av en cell, samt undvika gå in i en gång den kom ifrån.

Stega i sidled Gå i sidled en bestämd längd utan att titta framåt. Samma teknik som stegning framåt, men stegning framåt forceras efter detta tillstånd.

Vinst I detta tillstånd utför roboten en dans, men utöver det fastnar den i tillståndet tills det att autonoma läget startas om.



(a) Korsning typ A.

(b) Korsning typ B.

(c) Korsning typ C.

(d) Korsning typ D.

(e) Korsning typ E.

Figur 2: Skiss över de olika svängar och korsningar som kan förekomma. Notera de kryss som representerar den maximala gränsen för hur långt en sensor kan läsa. Varje cellblock är 80x80 cm.

3.2 Regleralgoritm

Roboten använder sig av tre separata regleringar för att hålla sig rätt i labyrinterna. Robotens position i sidled regleras för att hålla roboten mitt i labyrinten och robotens rotation regleras för att hålla roboten rak i labyrinten. En sorts korsning roboten kan stöta på i labyrinten löses enklast genom att gå i sidled, därmed måste även avståndet mellan robotens front och väggen framför regleras. Alla regleringar är inte aktiva samtidigt, utan de aktiveras och avaktiveras beroende på vad AI-modulen hittar på.

Generell beräkning av reglerutslag Roboten använder PID-reglering, det vill säga att man tar hänsyn till felet, integralen av felet samt derivatan av felet. Dessa skalas av parametrarna K_p, K_i, K_d, vilket utifrån felet $e[x]$ som samplats med sampeltiden Δt ger uttrycket

$$u[x] = K_p e[x] + K_i \sum_{0}^{x} [e[x]\Delta t] + \frac{K_d}{\Delta t}[e[x] - e[x-1]] \tag{1}$$

$$x \in [0, \infty[, e[-1] = 0 \tag{2}$$

för reglerutslaget. Det antas att $e[x] \sim\in [-1, 1]$.

Beräkning av felet De tre olika felen beräknas separat med relativt enkel geometri och trigonometri.

Applicering av reglerutslaget De tre regleringarna skapar tre separata reglerutslag som appliceras på gånghastighet framåt, gånghastighet i sidled, respektive rotationshastighet. För enkelhetens skull reglerar man inte i den riktning som roboten rör sig. Det vill säga, när roboten rör sig framåt reglerar man sidorörelse och rotation, när roboten roterar reglerar man ingenting och när roboten rör sig sidlänges reglerar man frontavstånd och rotation. Utifrån detta kan man ta fram en önskad rörelse $(\Delta x, \Delta y, \Delta\omega)$ som skalats så att $|\binom{\Delta x}{\Delta y}| \leq \Delta v_{max}$ där Δv_{max} är önskad maxhastighet.

3.3 Gångalgoritm

Robotens gångalgoritm styr roboten enligt styrbeslut från andra. Indata består av vektorn

$$\Delta\vec{v}_t = \begin{pmatrix} \Delta x \\ \Delta y \\ \Delta\omega \end{pmatrix}, \tag{3}$$

alltså önskad rörelsehastighet framåt, åt sidan och önskad rotationshastighet. Beräkningar genomförs i ett kartesiskt högersystem.

Gångalgoritmen bygger på att tre ben är i luften och tre ben på marken. Ytterligare definierar man en normalposition $\vec{b}_n$ och ett maximalt tillåtet avstånd r_b från $\vec{b}_n$ för respektive ben. Låt v_b vara positionen för ben. Vidare låter man tiden mellan rörelseberäkningarna, en "tickrate", vara Δt, samt den absoluta rörelsen för detta "tick" vara

$$\vec{v} = \begin{pmatrix} x \\ y \\ \omega \end{pmatrix} = \Delta\vec{v}_t \cdot \Delta t. \tag{4}$$

Definiera även en rotationsmatris R_ω som roterar ω enheter runt z-axeln enligt

$$R_\omega = \begin{pmatrix} \cos\omega & \sin\omega & 0 \\ -\sin\omega & \cos\omega & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{5}$$

De tre benens nya position på marken blir då

$$\vec{v}_b^+ = R_\omega^T(\vec{v}_b - \vec{v}). \tag{6}$$

De ben som är i luften kräver dock en mer komplicerad rörelse. För att roboten ska ta så stora steg som möljigt vill man att ett ben i luften ska landa i den punkt som är i den riktning som roboten ska röra sig från benets normalposition $\vec{b}_n$ med avstånd r_b från denna. För att ta hänsyn till rotationen kan man även beräkna en vektor

$$\vec{v}_r = \vec{b}_n \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}. \tag{7}$$

En optimal landningspunkt

$$\vec{v}_0 = \frac{|\Delta\vec{v}_t|\frac{\vec{v}}{|\vec{v}|} + \Delta\omega\frac{\vec{v}_r}{|\vec{v}_r|}}{\left||\Delta\vec{v}_t|\frac{\vec{v}}{|\vec{v}|} + \Delta\omega\frac{\vec{v}_r}{|\vec{v}_r|}\right|} \cdot r_b + \vec{v}_b \tag{8}$$

kan då beräknas, varvid rörelsen för de ben som är i luften beskrivs av

$$\vec{v}_b^+ = k|\vec{v}|\frac{\vec{v}_b - \vec{v}_0}{|\vec{v}_b - \vec{v}_0|}, \tag{9}$$

där k är en skalningsvariabel för att få benen att röra sig snabbare i luften än på marken.

Koordinaterna kan sedan med hjälp av kinematiken konverteras till servopositioner och därmed kan roboten gå.

Låt a_b vara den sträcka som ett ben på marken har kvar till gränsen av det område det har tillåtelse att vistas i. Detta beräknas internt som en stråL/sfär skärning. Man kan då beskriva lyfthöjden hos benen i luften som en funktion $z = f_z(a_{bmin})$. Man har valt att använda ett andragradspolynom för en mjuk rörelse.

Utökningar Utöver den gångalgoritm som beskrivs ovan har man lagt till ett antal förbättringar och optimeringar.

- Benen återgår till sina normalpositioner om $\Delta\vec{v}_t = \vec{0}$,
- r_b och b_n beräknas dynamiskt under körning som en funktion av robotens höjd,
- Robotens höjd kan ställas dynamiskt $\vec{v}_t = \vec{0}$, varvid roboten justerar benen till nya normalpositioner.
- Robotens globala koordinatsystem kan förflyttas och roteras då $\Delta\vec{v}_t = \vec{0}$, genom några enkla matrismultiplikationer, något som möljliggör enkel implementation av till exempel dans.

Möjliga förbättringar Ett antal förbättringar är möjliga för att göra gångalgoritmen bättre.

- Prestandan skulle kunna förbättras genom att skriva om gångalgoritmen till optimerad C. Nuvarande tickrate ligger på $\sim$ 30Hz, vilket är tillräckligt, men aningen ryckigt. Optimal tickrate tycks ligga på minimalt $\sim$ 120Hz.
- För tillfället beräknas r_b och b_n endast en gång för alla ben, och då endast vid förändring av robotens höjd vid stillastående, av prestandaskäl. Givet en mer optimerad implementation skulle man kunna beräkna dessa värden separat för varje ben och därmed enkelt kunna manipulera robotens höjd och koordinatsystem under gång.
- Steglängden skulle kunna ökas genom att ersätta det enkla dataparet r_b, b_n med en mer komplex representation. Denna representation skulle till exempel kunna vara en polygonmodell av roboten och robotens tillåtna rörelseområde, tillsammans med enkla *animerade* modeller av alla ben. Detta skulle inte kräva särskillt mycket mer prestanda[1], men skulle utgöra en betydligt mer komplex beräkning.

3.4 Kinematik

För att kunna behandla benens positioner i ett enkelt 3D-koordinatsystem krävs att man kan omvandla fritt fram och tillbaka mellan servovinklar och positioner i ett kartesiskt koordinatsystem.

Låt längderna på benets olika leder anges av l_0, l_1, l_2, samt de tre servovinklarna vara $\theta_0, \theta_1, \theta_2$, inifrån robotchassit och ut. Låt ytterligare R_b vara en rotationsmatris anpassad för berört ben och $\vec{b}_p$ vara vektorn från robotens origo till benets fästpunkt.

Servovinklar $\rightarrow$ Kartesiska koordinater För att skapa en kartesisk koordinat ifrån servovinklarna på ett ben börjar man med att beräkna benets position i cylindriska koordinater enligt

$$\vec{v}_c = \begin{pmatrix} r \\ \omega \\ z \end{pmatrix} = \begin{pmatrix} l_0 + l_1\cos(\theta_1) + l_2\cos(\theta_1+\theta_2) \\ \theta_0 \\ l_1\sin(\theta_1) + l_2\sin(\theta_1+\theta_2) \end{pmatrix}, \tag{10}$$

varefter man omvandlar detta till en vektor i kartesisk rymd genom

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} r\cos(\omega) \\ r\sin(\omega) \\ z \end{pmatrix}, \tag{11}$$

för att sedan beskriva vektorn i ett globalt koordinatsystem med

$$\vec{v}_g = \vec{b}_p + R_b\vec{v}. \tag{12}$$

[1]Detta skulle utgöra en s.k. Ray-Mesh intersection, en $O(\log n)$ operation tillsammans med mycket väloptimerade bibliotek.

Kartesiska koordinater → Servovinklar Genom att bryta ut $\vec{v}$ ur ekvation 12 får man

$$\vec{v} = R_b^{-1}(\vec{v}_g - \vec{b}_p). \tag{13}$$

Vidare omvandlar man till cylindriska koordinater enligt

$$\vec{v}_c = \begin{pmatrix} r \\ \omega \\ z \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \text{atan2}(y, x) \\ z \end{pmatrix} \tag{14}$$

får man en cylindrisk koordinat relativ benets fästpunkt. Man kan då ta fram att $\theta_0 = \omega$. Låt sedan $r_0 = r - l_0$ och $d = \sqrt{r_0^2 + z^2}$, det vill säga avståndet från andra servot till foten. Man kan nu ställa upp en triangel med sidlängderna d, l_1, l_2, samt låta $\omega_{l_1 d}$ vara vinkeln mellan andra servot och foten, ω_{d0} vara vinkeln från foten till andra servot och $\omega_{l_1 l_2}$ vara vinkeln mellan sidorna med längderna l_1 respektive l_2. Det gäller då att

$$\omega_{l_1 d} = \cos^{-1} \frac{l_1^2 + d^2 - l_2^2}{2 l_1 d} \tag{15}$$

$$\omega_{l_1 l_2} = \cos^{-1} \frac{l_1^2 + l_2^2 - d^2}{2 l_1 l_2} \tag{16}$$

$$\omega_{d_0} = \text{atan2}(z, r_0), \tag{17}$$

vilket medger att man beräknar

$$\begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \end{pmatrix} = \begin{pmatrix} \omega \\ \omega_{l_1 d} + \omega_{d_0} \\ \omega_{l_1 l_2} - \pi \end{pmatrix}. \tag{18}$$

Därmed har man vinklar som efter diverse skalningar kan användas för att sätta servopositionerna. Observera att man i dessa beräkningar bortser från en möjlig lösning då man utgår från att det yttersta servot vinklas nedåt. Detta är alltid optimalt såvida man inte vill gå upp och ner.

3.5 RS-232

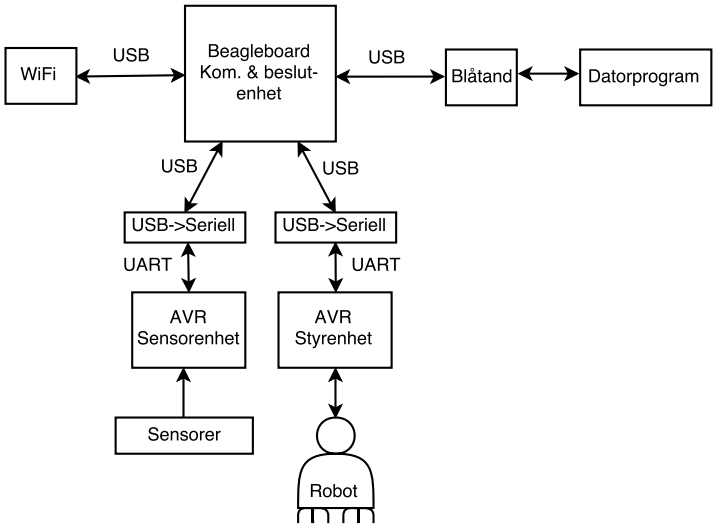
Kommunikationen till och från kommunikations- och beslutsenheten till systemets microprocessorer använder sig av standarden för seriell kommunikation, RS-232 [8]. Kommunikationen sker genom att båda enheterna som ska kommunicera har en så kallad `UART`. En `UART` är en hårdvarumodul som har två portar av intresse, en mottagare `RX` och en sändare `TX`. Den ena enhetens mottagare kopplas in i den andras sändare och vice versa. Detta möjligör full duplex kommunikation, det vill säga att enheterna kan skicka och ta emot data samtidigt. Undantaget till detta är kommunikatonen med servomotorerna vilket är half duplex. RS-232 stödjer synkron samt asynkron kommunikation men i detta projekt används det endast asynkront. Detta innebär att innan enhenterna kan kommunicera måste båda vara inställda på samma överföringshastighet.

4 Systemet

Systemets huvudsakliga uppgift är att samla in data om sin omgivning för att sedan utföra de beräkningar som krävs för att roboten autonomt ska kunna navigera genom en labyrint, alternativt koppla bort den autonoma delen och navigera via en dator. För att göra systemet lättare att vidareutveckla och för att underlätta arbetsprocessen är systemet indelat i tre delsystem.

4.1 Ingående delsystem

Systemet består på hög nivå av ett antal distinkta enheter. Roboten har en sensorenhet som möjliggör detektektion av den närmsta omgivningen, en styrenhet som ansvarar för motorstyrning, samt en kombinerad Kommunikations- och beslutsenhet som hyser gångalgoritmer, regleralgoritmer, labyrintnavigering, ai, samt kommunikation med en dator. På datorn körs ett datorprogram som kan användas för att styra och övervaka diverse parametrar på roboten och direkt fjärrstyra roboten. Figur **??** visar en översiktlig bild av de olika modulerna, samt hur dessa kommunicerar med varandra.



Figur 3: Översikt av systemet

Till Kommunikations- och beslutsenheten används enkortsdatorn BeagleBoard, medan styrenheten och sensorenheten använder sig av varsin mikrokontroller(ATMega1284p). Eftersom BeagleBoarden inte använder 5V-logik används USB till serieadaptrer för kommunikationen över spänningsgränsen.

5 Kommunikations- och Beslutsenhet

Kommunikations- och Beslutsenheten ansvarar dels för kommunikation med både dator och systemets AVR-processorer, och dels för AI-beslut, regleralgoritm och gångalgoritm. Denna modul är väldigt mjukvarubaserad och kan utan problem ersättas med till exempel en vanlig laptop.

5.1 Hårdvara

Kommunikations- och Beslutsenheten är implementerad på en Beagleboard-xM [1], något som gör att man kan skriva mjukvaran snabbt och enkelt i Python samt använda den fulla kraften hos det Linuxsystem som körs på den.

Man använder en WiFi-dongel för att få tillgång till internet och en Bluetooth-dongel för att kommunicera med dator. För kommunikation med de två AVR-processorerna används två USB-serielladaptrar. Linux hanterar all hårdvara utan att någon konfiguration är nödvändig. Det finns dock en liten möjlighet att Linux under uppstart råkar "byta plats" på USB-serielladaptrarna.

5.2 Operativsystem

Man har använt Ubuntu [3], då detta system var enkelt att installera och konfigurera. Ovanpå detta har man installerat diverse mjukvara både för utveckling av programvaran och nödvändiga programvarubibliotek. För att ansluta till internet används `NetworkManager`.

5.3 Programmeringsspråk

För att snabbt och smidigt kunna utveckla mjukvaran har man valt att använda Python. För bästa stöd gentemot bibliotek har man även valt att arbeta med version 2 av språket.

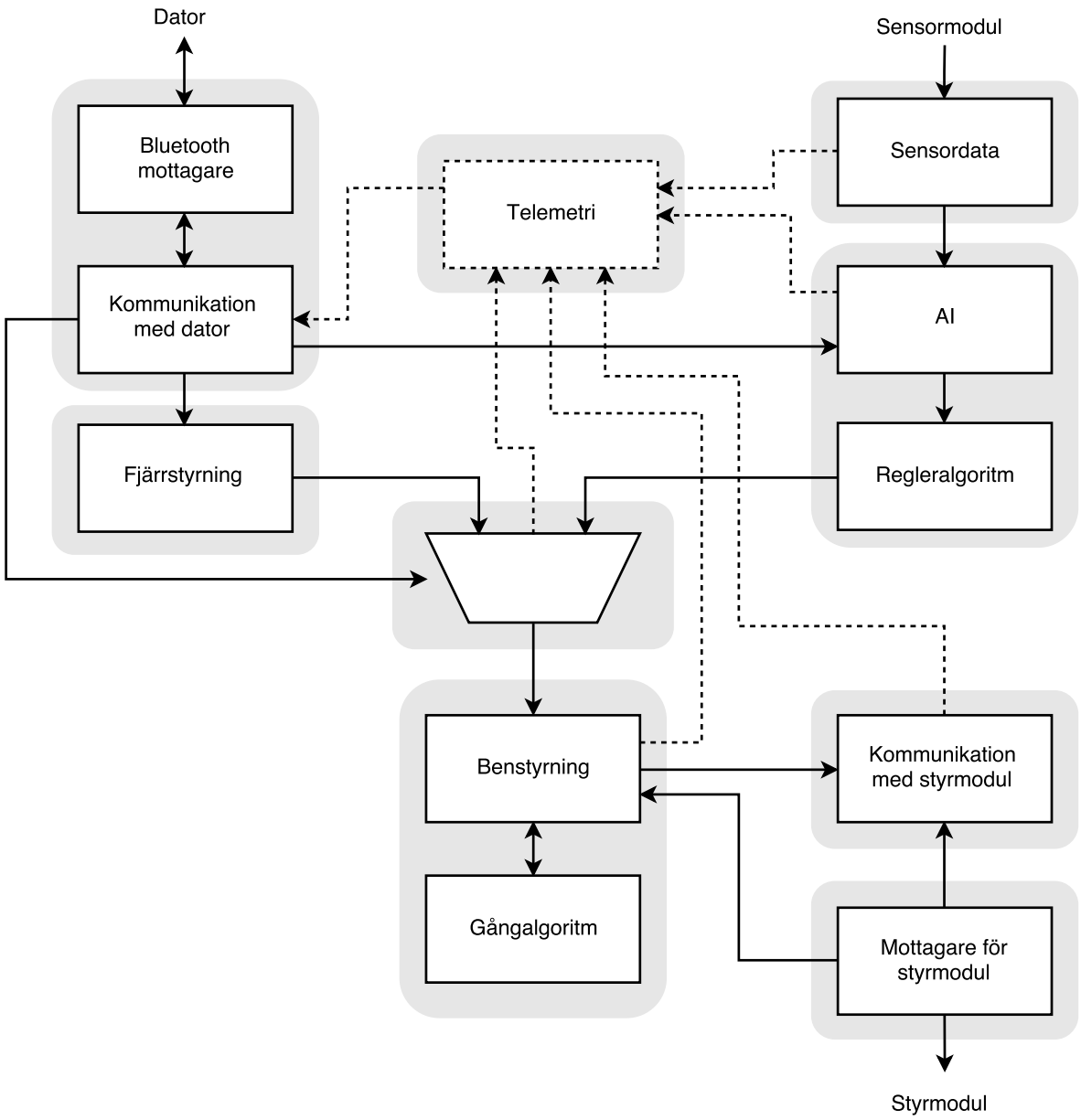
Programmvarubibliotek Man har valt att använda ett antal programvarubibliotek för att förenkla utvecklingsarbetet.

- PySerial [4], används för seriell kommunikation över UART,
- NumPy [5], används för diverse beräkningar för kinematik och gångalgoritmer,
- PyBluez [6], används för bluetoothkommunikation med dator.

5.4 Mjukvaruarkitektur

Enhetens mjukvara bygger på en flertrådad arkitektur där varje tråd ansvarar för en begränsad uppgift.

Main Huvudprogrammet ansvarar för att initiera alla hanterar klasser, trådar och bussar sinsemellan. Vid uppstart skapas först alla bussar, varav alla klasser instansieras med dess angivna bussar. Sist startas alla trådar samtidigt och programmet är i gång.



Figur 4: Schematisk bild över mjukvaruarkitekturen. De ljusgrå rutorna visar hur programmet delas upp i flera trådar. Pilarna representerar de enkelriktade meddelandeköer som sätts upp mellan de olika modulerna. Streckade pilar är meddelandeköer specifikt för telemetridata.

Message passing Mjukvaran bygger på en message-passing arkitektur. Detta leder till en enkel och ren modell för flertrådning av mjukvaran utan state mellan trådarna. Nackdelen är dock en viss intern latens och en något ökad minnesåtgång. Denna latens påverkar dock inte systemet märkbart, och det finns gott om minne på kortet. Message passing bör inte ha någon större effekt på prestandan, men beroende på schemaläggning kan tiden för att utföra en uppgift variera ganska mycket. Detta har dock inte varit något problem i praktiken. Vår message-passing implementation är en tunn wrapper kring Pythons synkrona `Queue` som ingår i standardbiblioteket.

Systemets message-passing-implementation kan användas både blockerande och icke-blockerande. Ytterligare sätts en maxstorlek för meddelandekön. Denna maxstorlek får aldrig uppnås under normal användning. Implementationen kallar kön för *pipe*, mottagarens del för *recieve* och avsändarens del för *send*. Man bildar alltså ett par av *sends* och *recieves*.

Vid programstart skapas par av sends och recieves, varefter de ges som argument till de trådar som startas upp, vilket bildar ett nätverk, en riktad graf.

För att systemet ska fungera krävs att följande krav uppfylls:

- Ingen del av systemet får använda sig av busy waiting. Detta skulle stjäla processortid från övriga trådar samt öka latensen i systemet. Därmed måste man antingen be schemaläggaren vänta en viss tid med `time.sleep(x)` eller använda blockerande message-passing.
- Om man väljer att använda `time.sleep(x)` måste tråden läsa sin recieve icke-blockerande, samt se till så att den tömmer kön helt varje gång den läses. Att inte göra detta kan leda till att man fyller kön.
- Om man använder blockerande message passing får man inte använda `time.sleep(x)` i samma tråd. Dessutom får man då bara ha endast en recieve för att undvika att fylla upp de underliggande köerna.
- Man får inte skicka meddelanden snabbare än de kan behandlas på mottagarsidan annat än högst tillfälligt.

Telemetri Telemetriklassen hanterar inkommande telemetridata från andra enheter i syfte av att logga status, debug-meddelanden eller samlingar av data. Klassen kan ta emot dictionaries eller strängar, oavsett från var det kommer ifrån. Om inkommande data är en sträng så läggs meddelandet in i en log-lista. Om det istället är en dictionary så läggs det till i en uppsamlings dictionary som ska skickas vidare, tillsammans med log-meddelanden, till kommunikationsmodulen.

På grund av att all inkommande data hanteras på samma villkor, medger uppsamlings dictionary:n ett globalt namespace där data kan övreskrivas av andra delar av programmet.

Sensordata Sensordataklassen tar hand om mottagandet av data från sensorenheten, vilket sker seriellt med hjälp av bilioteket Pyserial. Klassen väntar på att *CTS* blir aktiv vilket betyder att sensorenheten är redo att börja sända data till BeagleBoarden. Synkronisering av paketen sker genom att leta efter en header som finns i varje paket. För att försäkra att överföringen gått korrekt till sker en kontroll med hjälp av den checksumma

som medföljer varje paket. Innan datan skickas vidare till relevanta mottagare konverteras rådatan från sensorenheten till mer hanterbara enheter som centimeter, spänning eller vinkelhastighet.

AI I denna klass hanteras all beslutstagning för autonoma styrkommandon. Klassen väntar på inkommande sensordata och slussar sedan igenom det via dess separata ai-modul och regleralgoritm-modul för att få fram ett färdigt styrbeslut, se 3.1 och 3.2. Sist vidarebefodras kommandot till MUX. Klassen tar även hand om att informera MUX om knappen för byte mellan manuellt och autonomt läge, vidarebefodran av sensordata och autonoma beslut till telemetri, samt intern hantering av inkommande styr- och reglerparametrar, alla vilka skickas till de separata modulerna.

När sensordata kommer in i hanteraren skickas detta direkt in i ai-modulen, vilket returnerar ett generellt styrbeslut om hur roboten vill röra sig i helhet. Resultatet tillsammans med all sensordata skickas direkt vidare till regleralgoritmen för finjustering av styrningen. Till sist returneras det autonoma beslutet vilket konverteras till ett färdigt styrkommando.

Bluetooth mottagare Denna klass hanterar all direkt dataöverföring via bluetooth mellan datorprogrammet och BeagelBoarden. Koden är avsedd att ligga i vilo-läge och lyssna på ett datorprogram som vill parakoppla sig, varav en blåtandslänk initieras. Vid denna punkt kan data, enligt protokollspecifikation i appendix, tas emot från datorprogrammet tills dess att den kopplar ifrån.

I denna klass hanteras även data överföringen till datorprogrammet då en blåtandslänk inkluderar både skicka som ta emot. Däremot hanteras all skickning i kommunikation med dator modulen som ligger i en separat tråd. Detta tillåter telemetri data att skickas kontinuerligt.

Kommunikation med dator Detta är den centrala enheten för kommunikation mellan robotens interna system och datorprogrammet. Klassen har i uppgift att delegera bluetooth kommandon till fjärrstyrning, AI samt MUX. Dessa kommandon innefattar manuella styrbeslut, styrparametrar samt byte mellan autonomt och manuellt läge. Klassen har även i uppgift att vidarebeforda alla inkommande telemetri data paket via bluetooth till datorprogrammet.

Fjärrstyrning Här passerar alla manuella styrkommandon och styrparametrar relevanta till robotens benstyrning. Klassens huvuduppgift är att översätta datorprogrammets relativt spridda kommandon till funktionerande kommandon för benstyrningen. Denna abstraktion åtgärdar även eventuella fel eller ofullständigheter av kommandon skickade från datorprogrammet, samt tillåter ändringar i benstyrningen utan att datorprogrammet behöver skrivas om.

MUX Denna klass har i uppgift att välja vilka styrkommandon som ska skickas vidare till benstyrningen beroende på om roboten befinner sig i manuellt eller autonomt läge. MUX tar kontinuerligt emot eventuella styrkommandon från fjärrstyrning och AI, och bestämmer vad som ska vidarebefodras.

MUX vidarebefodrar även styrparametrar från datorprogrammet som är skilda från styrning. Dessa inkluderar ändringar i höjd, hastighet samt servo-styrka.

Benstyrning Denna klass har i uppgift att läsa inkommande styrkommandon och kontinuerligt beräkna vinkelbeslut för alla unika servomotorer, vilket slutligen skickas till klassen för kommunikation med styrmodulen.

Benstyrningen använder sig av den separata gångalgoritm-modulen för att utföra den grundliga servoberäkning, se 3.3. Gångalgoritmens beräkningar klockas av benstyrningen till att utföra beräkningar 35 gånger i sekunden, allt för att hålla uppdateringen kontinuerligt. Efter beräkningar konverteras alla utmatade servo vinklar i radianer till absoluta servo positioner.

För att hålla beräkningshastigheten konstant läser klassen in eventuella styrkommandon icke-blockerat från dess buss. Vid inläsning töms bussen fullständigt för att försäkra benstyrningen om att den hanterar senaste data.

Kommunikation med styrmodul Kommunikationen med styrmodulen sker seriellt med hjälp av biblioteket Pyserial. Den kollar först om AVR är redo att skicka och ta emot data genom att kolla om signalen CTS är aktiv. Den synkar paketen den får från styrmodulen genom att leta efter paketets header. Den har ett antal olika funktioner för att skicka alla kommandon till styrmodulen. Den kan skicka servo positioner, maximala vridmomenten, maximala hastigheten och slå på/av vridmomentet.

Mottagare för styrmodul Här hanteras all seriell läsning av data skickat från styrenheten. Denna data innefattar aktuella servo vinklar, vilket skickas direkt vidare till telemetri enheten.

6 Styrenhet

Styrenhetens uppgifter är att abstrahera och hantera kontrollen av servomotorerna. Enheten är ganska primitiv och agerar som ett abstraktionslager mellan beslutsenheten och hårdvaran. Den kommer att ta emot data från kommunikations- och beslutsenheten för att sedan anpassa datan och vidarebefodra det till motorerna. Styrenheten kommer även att läsa telemetridata från motorerna och vidarebefodra det till kommunikations- och beslutsenheten. Se appendix A för enhetens kretsschema.

6.1 Hårdvara

Mikrokontroller Mikrokontrollern som används i styrenheten är Atmega1284P. Den valdes eftersom det är den enda tillgängliga mikrokontrollen med 2 st UART bussar, vilket styrenheten behöver. En UART-buss används för servostyrning och en för kommunikation med kommunikations- och beslutsenheten.

Motorstyrning Servomotorerna kommunicerar seriellt i half-duplex medans mikrokontrollerns UART-buss använder full-duplex. Det innebär att TX och RX måste kortslutas på kontrollerns UART vilket görs genom att använda två st tri-state buffers enligt figur 10. På detta vis så får man inte ett eko där kontrollern tar emot datan den själv skickar. Istället väljs riktningen på kommunikationen med en tredje signal, `servo_dir`.

Nödstoppsknapp Som säkerhetsåtgärd så finns det en nödstoppsknapp på styrenheten som stoppar all servostyrning. Knappen skickar ett avbrott till mikrokontrollern som sedan stänger av vridmomentet i motorerna så fort den kan. En avstudsad tryckknappsmodul används för att säkerhetsställa att bara ett interrupt skickas. För att enheten ska börja fungera igen efter en knapptryckning måste den återställas med hjälp av reset knappen. På projektgruppens konstruktion är nödstoppsknappen den knapp som har rosa tejp med en dödskalle på.

Kristalloscillator Kristalloscillator EXO-3 16 Mhz används som klocka till både styr- och sensorenheten.

6.2 Mjukvara

Mjukvaran i styrenheten är skrivet helt och hållet i programmeringsspråket C. Mjukvaran använder många avbrott eftersom att det ger programmet möljighet att göra flera uppgifter samtidigt och uppfölja realtids begränsningar.

Buffrar Styrenheten använder sig av ett antal olika buffrar för att spara data. Buffrarna är implementerade som en kö i en cirkulär array innehållande pekare till minnet.

UART avbrott Programmet använder tre olika avbrott för varje UART, `RX complete`, `TX complete` och `UDRE empty`. `RX complete` används för att ta emot data från antingen motorerna eller kommunikations- och beslutsenheten. Detta görs genom att först allokera minne för hela paket, sedan spara undan alla inkommande bytes och sist så läggs en pekare till paketet in i en buffer.

`UDRE empty` och `TX complete` används för att skicka data på UART-bussarna. `UDRE empty` används för att skicka varje enskild byte, först så hämtas ett helt paket ut från en buffer. Sedan så skickas en byte vid varje avbrott och räknare används för att veta när hela paketet har skickats. `TX complete` kallas efter ett paket har skickats färdigt och används bland annat till att rensa upp allokerat minne.

Timeravbrott Styrenheten använder sig av ett antal olika timeravbrott. Två av dem används som timeouts för UART-bussarna. Detta är nödvändigt eftersom man annars riskerar att hela bussen låser sig, om enheten till exempel förväntar sig få ett svar från ett servo men aldrig får ett. Då sätts signalen `servo_dir` hög vilket blockerar all utgående trafik och låser bussen. Timeravbrottet förhindrar då detta genom att sätta signalen låg igen efter några millisekunder har gått utan att den fått ett paket.

Två andra timeravbrott används för att läsa nuvarande positionen från servon och att skicka alla positioner till kommunikations- och beslutsenheten med jämna mellanrum. För nuvarande så skickas ett paket till kommunikations- och beslutsenheten varje 192ms och data läses från servon varje 32ms.

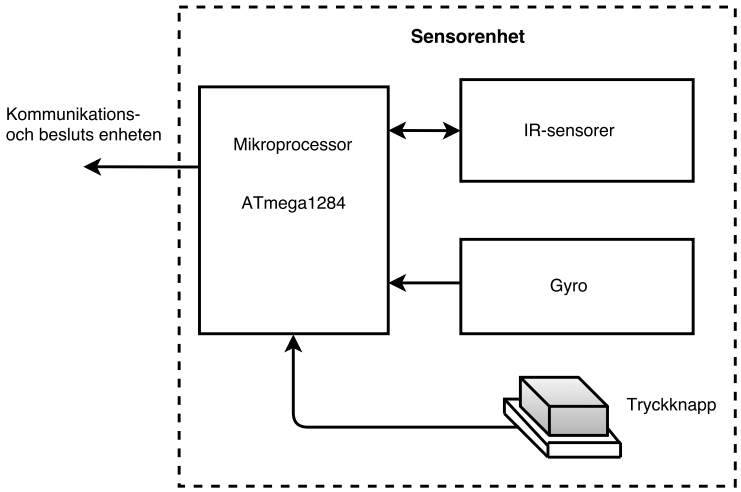
Externtavbrott Modulen använder sig av ett externtavbrott kopplat till nödstopps-knappen. När knappen trycks ned skickas ett broadcast till alla servon som stänger av dess vridmoment och så sätts en flagga vilket blockerar att fler kommandon skickas till motorerna.

Huvudloop Huvudloopen är det som håller ihop allting. Den kollar om vi har fått in någon data genom att kolla i buffrarna. Har vi fått in ett paket så hanterar den det på direkten. Det är även huvudloopen som ser till att skicka paket till motorerna och kommunikations- och besluts enheten varje gång vi har fått ett timeravbrott som säger åt enheten att göra det.

7 Sensorenhet

Mikrokontrollern i sensorenheten läser data från sensorerna och knapparna som används för att återställa mikrokontrollern och för att slå av och på det autonoma läget. Den mottagna datan behandlas samt filtreras av enheten och skickas sedan vidare till kommunikations- och beslutsenheten seriellt i ett väl definerat protokoll.

En översikt av sensorenheten visas i figur 5. I figur 11 visas ett kopplingsschema över sensorenheten.



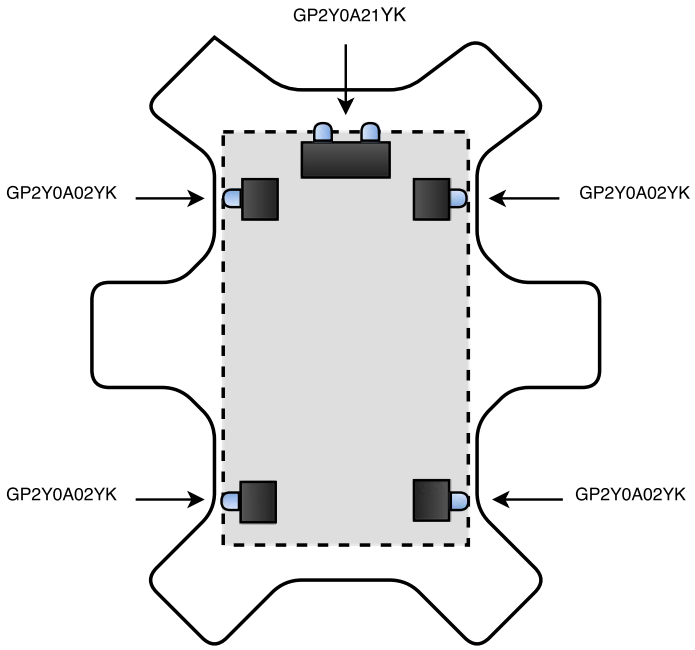
Figur 5: Översikt av sensorenheten

7.1 Hårdvara

Mikrokontroller Mikrokontrollen som används till sensorenheten är en *ATmega1284P*. Den har åtta stycken analoga insignaler kopplade till A/D-omvandlaren, där sensorerna är anslutna. Mikrokontrollern har 128 KB minne, vilket är tillräckligt med utrymme för sensorenheten.

Optiska avståndsmätare För att läsa av robotens omgivning är fem stycken optiska avståndsmätare placerade på robotens chassi. Sensorernas placering kan ses i figur 6. Utsignalerna från de optiska avståndsmätarna är analoga. Längst fram på roboten är avståndsmätaren som mäter inom intervallet 10-80 cm (GP2Y0A21YK) placerad. På bägge sidorna av roboten är två sensorer som mäter mellan 20-150 cm (GP2Y0A02YK) fastsatta så långt ifrån varandra som möjligt för att undvika att störa varandras mätvärden. Mellan de optiska avståndsmätarna och AVR:en är ett LP-filter med resistansen 18kohm och kapacitansen 100nF fastvirade, i syfte att dämpa störningar av mätvärdena.

Gyro För att beräkna vinkelhastigheten då roboten roterar används vinkelhastighetssensorn *MLX90609*. Gyrot är placerad ovanpå chassits mitt. Utsignalen från gyrot kan avläsas både digitalt och analogt och till sensorenheten valdes den analoga utsignalen. Signalen är linjär och inget filter behöver användas.



Figur 6: Sensorplacering över chassit sett från ovan.

Knapp för autonomt läge För att aktivera robotens autonoma/manuella läge används en avstudsad tryckknappsmodul 2x1. Den är placerad på chassins främre kant, för att lätt komma åt den. Tryckknappsmodulen har en grå och en blå knapp, där det är den blå som används för att ändra till autonomt/manuellt läge. Då roboten startas kommer den att vara i manuellt läge. Den grå knappen tillhör och hanteras i styrenheten.

Kristalloscillator En kristalloscillator *EXO-3* används som klocka både till styr- och sensorenheten. Den har hastigheten 16 MHz.

JTAG En JTAG används för att programmera mikrokontrollen direkt från Atmel Studio samt emulera programmet. JTAG:en är seriekopplad mellan styr- och sensorenheten.

RESET En röd enkel tryckknapp är placerad längst fram på robotens chassi. När denna knapp trycks ned återställs alla inställningar på mikrokontrollen.

UART Mellan AVR:en och beagleboarden är en uart kopplad. Den används för att skicka paket innehållandes sensordata från sensorenheten till kommunikations- och beslutsenheten.

7.2 Mjukvara

Mjukvaran i sensorenheten är skrivet helt och hållet i programmeringsspråket C. I centrum för mjukvaran står de olika avbrott som sköter det mesta av programmets olika

funktioner. Dessa gör det möjligt för sensorenheten att ta hand om sensormätningar och dataöverföring till kommunikations- och beslutsenheten efter specifika tidsintervall.

Huvudloop Huvudloopen gör inte mycket annat än att initiera och möjliggöra användning av de register och funktioner som behövs. Den väntar hela tiden på olika avbrott.

Timeravbrott Programmet för mikrokontrollern styrs av två tidsbundna avbrott, vilka triggas var 8:e och 40:e millisekund. I den snabbare tidsbaserade avbrottsrutinen hämtas data från den senaste gyromätningen. I detta fall valdes tidsintervallet 8 ms för att få med alla gyrots förädndringar med lagom noggrannhet. I den andra avbrottsrutinen hämtas data från alla IR-sensorer. Därefter beräknas medelvärdet av det nya värdet plus de fyra tidigare mätvärdena, för att filtrera bort feldata. Detta görs för varje IR-sensor. Vi satte timern på 40 ms eftersom att IR-sensorerna mäts i samma takt. Var 40:e ms sänds även ett paket till besluts- och kommunikationsenheten, med de senaste uppdateringarna av värdena från sensorerna och tryckknappen. Paketet innehåller 5 gyro-mätningsvärde och en mätning för varje IR-sensorn, eftersom gyrot läses av 5 gånger snabbare än IR-sensorerna. Paketet sänds med hjälp av en buffer, som är implementerad som en kö i en cirkulär array innehållande pekare till minnet.

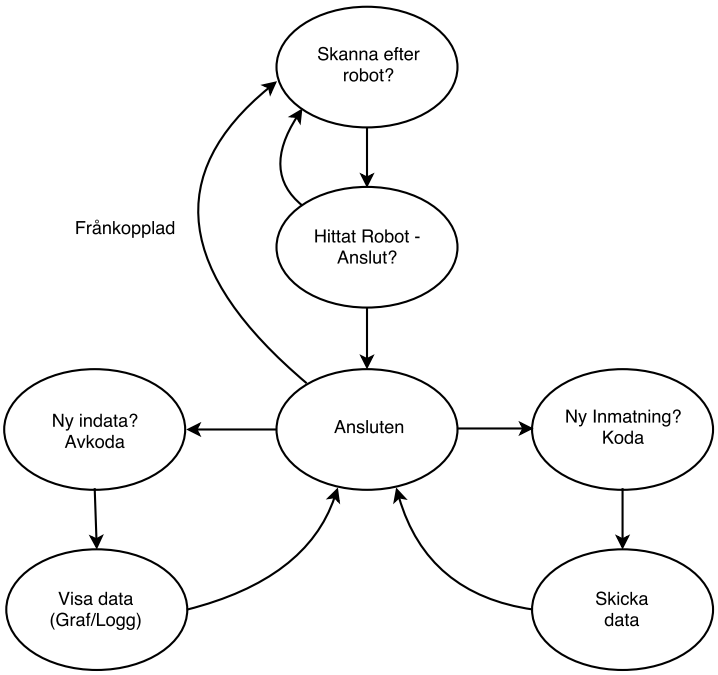
UART-avbrott Det tidsbundna avbrottet, på 40 ms, triggar igång uartavbrott i samband med att paketet sätts. I uartavbrottet hanteras paketet och innehållet delas upp så att all data skickas seriellt via uarten. Programmet använder två olika avbrott för uarten, `TX complete` och `UDRE empty`. De används för att skicka data på UART-bussarna. `UDRE empty` används för att skicka varje enskild byte. Först så hämtas ett helt paket ut från en buffer. Sedan så skickas en byte vid varje avbrott. En räknare används för att veta när hela paketet har skickats. `TX complete` kallas efter att ett paket har skickats färdigt och används bland annat till att rensa upp allokerat minne.

Externa avbrott Programmet för mikrokontrollern innehåller även ett externt avbrott, som triggas då den autonoma/manuella knappen tryckts ner. Knappdatan innehåller värdet 0 förutom då knappen tryckts ned. Då blir den 1, och så fort det första paketet med detta knappvärde har skickats iväg så får den värdet 0 igen.

Protokoll Sensorenhetens kommunikationsprotokoll beskrivs i appendix C.1.

8 Datorprogrammet

Datorprogrammet är nödvändigt för att kunna styra roboten. För att styra roboten måste datorn som programmet körs på vara ansluten till roboten via bluetooth. Programmet kommer till en början endast ge användaren alternativet att söka efter roboten. När roboten är hittad, kan användaren ansluta.



Figur 7: Flödesschema över datorprogrammet

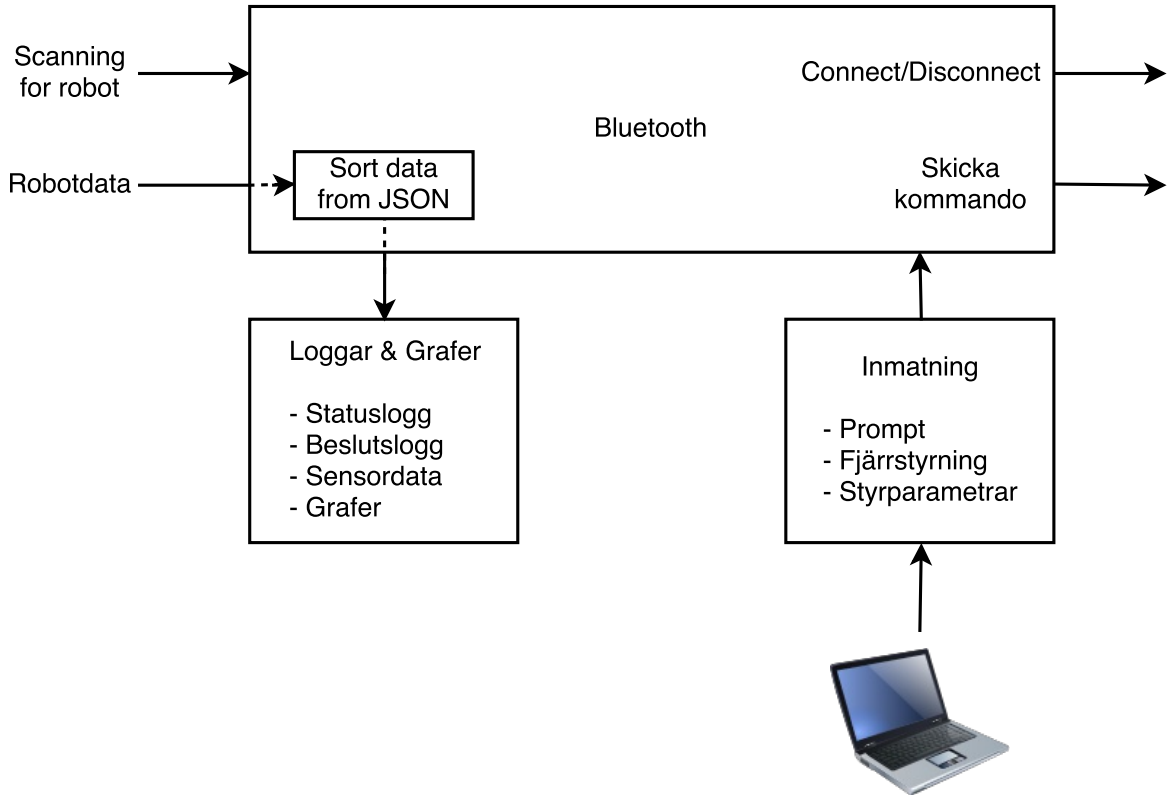
Vid lyckad anslutning ändrar datorprogrammet beteende. In- och utdata är nu den data som tas emot från roboten respektive det som användaren matar in. Vid ny indata kommer programmet avkoda den nya datan, och uppdatera fönstret så att det visas korrekt. Vid ny inmatning kollar programmet först om det är en giltig inmatning. Är inmatningen ogiltig ignoreras den. Är den däremot giltig, kodas kommandot och överförs till roboten via bluetooth.

8.1 Programkod

Datorprogrammet är skrivet i C++ med hjälp av utvecklingsmiljön QtCreator. Programmet är uppbyggt med hjälp av Qt Creators egna verktyg för design av användargränssnitt, Qts Json-klasser, Qt Bluetooth och det externa biblioteket QCustomPlot. Json-klasserna används för att avkoda, samt koda data. Qt Bluetooth används till att skicka och ta emot data, och ansluta till roboten. QCustomPlot fungerar som ett tillägg i designverktyget som utvecklaren med hjälp av kan lägga till grafer i gränssnittet.

8.2 Kommunikation

Datorprogrammet kommunicerar med robotens kommunikations- och beslutsenhet. Data sänds i JSON format, enligt specifikation i appendix.



Figur 8: Blockschema för datorprogrammet

8.3 Json

Data som överförs via Bluetooth är formaterad som JSON i UTF-8. När ny data ska skickas, skapar programmet ett QJsonObject. Utifrån det skapas sedan ett QJsonDocument, vilket alltså skickas via bluetooth som JSON. Processen blir omvänd vid mottagning av data. Utifrån JSON-data skapar programmet ett QJsonDocument, som i sin tur blir nyckeln i skapandet av ett QJsonObject. När det är gjort kan programmet sortera ut data ur objekten utifrån existerande variabelnamn från JSON-dokumentet. Data kan sedan visas i användargränssnittet.

8.4 Loggar och grafer

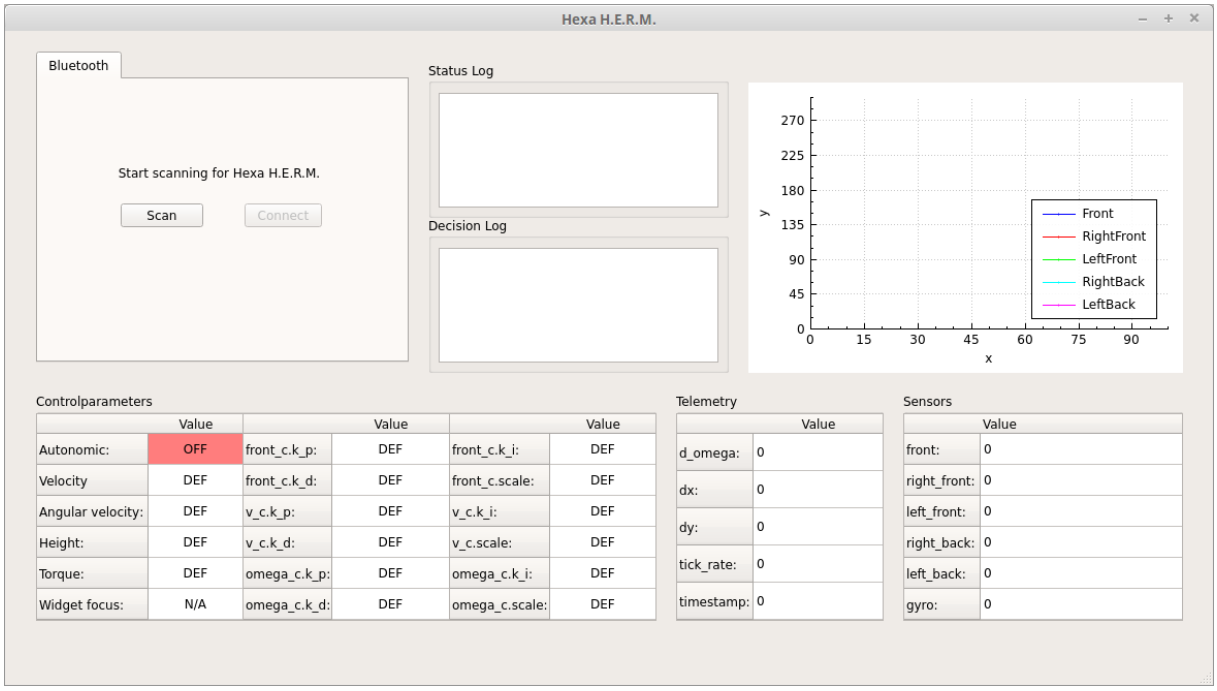
Loggarna och graferna visar data som programmet tagit emot via bluetooth. Sensorvärden visas i textformat, och ritas ut i grafer. Loggarna visar följande:

- Autonoma beslut
- Sensor- och telemetridata
- Robotens nuvarande tillstånd

Sensordata och information i statusloggen och beslutsloggen som skickats från roboten, loggas i en textfil på hårddisken.

8.5 Användarinmatning

Det finns totalt 22 tillåtna kommandon för användaren att mata in, varav 16 kommandon är för att sätta styrparametrar. När en styrparameter sätts, uppdateras gränssnittet under controlparameters, för att visa vilket värde som är satt. Se figur 9.



Figur 9: Översikt av datorprogrammet

För att flytta roboten genom piltangentstyrning krävs av användaren att först mata in kommandot `move` i prompten, för att då kunna styra med piltangenterna. Alternativ kan styrning ske genom knapparna WASD. Det finns totalt 8 möjliga riktningar att flytta sig med piltangenterna. Rotering vänster och höger görs med knapparna Q och E. Vid piltangentstyrning skickas kommandon med styrdata direkt till roboten. För mer information hur kommandon fungerar, se användarmanualen.

9 Slutsatser

Projektet har gått väldigt bra och hela gruppen är väldigt nöjda med resultatet. Men det finns som alltid många små saker som skulle kunna förbättras och även några större. Detta beror mest på att vi har väldigt liten erfarenhet med liknande projekt och det var lätt hänt vi glömde tänka på vissa saker.

Val av komponenter och programspråk har i allmänhet lyckats bra. Särskilt möjligheten att använda ett högnivåspråk som Python har underlättat avsevärt, om än med något degraderad prestanda. Val och placering av sensorer visade sig vara i allmänhet bra, dock hade man kunnat ta likadana avståndssensorer överallt. Gyrosensorn verkade fungera bra, men används inte i den färdiga koden, då precisionen på gångalgoritmens uppfattning av sin egen rotationshastighet visade sig vara bättre.

Placeringen av knappar skulle kunna förbättras märkvärt. Just nu är det ganska pilligt att komma åt knapparna och svårt att se vilken knapp som är vilken. Detta på grund av att vi har en platta med alla sensorer precis över dem. Det är även lätt hänt att man stör den främre sensor när man startar autonoma läget vilket får roboten att börja rotera.

Annan enkortsdator är en av de större sakerna man kan förbättra. Den Beagleboard som används under projektet har inte samma tillgänglighet på dokumentation och framför allt modern mjukvara som önskats.[2] Projektet hade gått smidigare med en nyare och mer popluär enkortsdator, till exempel en Raspberry Pi. Den har även lättillgängliga pins på rätt spänningsnivå och skulle ha kunnat styra en del hårdvara själv istället för att alltid gå via en USB-serielladapter till en AVR.

Tillgång till bättre Wifi hade gjort allting mycket smidigare. Vi har använt en Wifi-dongel på beagleboarden bland annat för att uppdatera kod via git och felsöka programmet. Men problemet har varit att den tappar uppkopplingen till eduroam hela tiden, ibland efter bara någon minut. Detta har gjort att vi blev tvunga att koppla in en seriell kabel och köra en terminal över den, vilket inte fungerar alls lika bra som SSH. Det hade sparat oss en tid om vi hade haft en stabil uppkoppling till internet. Problemet med uppkopplingen hade att göra med hur eduroam och vår raspberry inte tyckte om varandra, övriga nät verkar fungera bra.

Större försiktighet med servona underlättar för alla. Vi har bränt ett servo och slitit ut kuggarna på ett antal. Servona verkar vara svåra att bränna, men enkla att slita ut. Några av robotens servon var även utslitna redan när vi fick den.

Bättre tassar/golv skulle underlätta både för projektgrupp och servon. Det hala golvet får benen att glida isär, vilket medför både en slirig gångstil och mer tryck på servon.

[2] Ubuntu 14.04 börjar bli riktigt gammalt.

Referenser

[1] Beagleboard.org. Beagleboard-xm, http://beagleboard.org/beagleboard-xm, 2015.

[2] Doxygen. Doxygen, http://www.stack.nl/~dimitri/doxygen/, 2015.

[3] elinux.org. Beagleboardubuntu, http://elinux.org/BeagleBoardUbuntu, 2015.

[4] Chris Liechti. Pyserial, https://github.com/pyserial/pyserial, 2015.

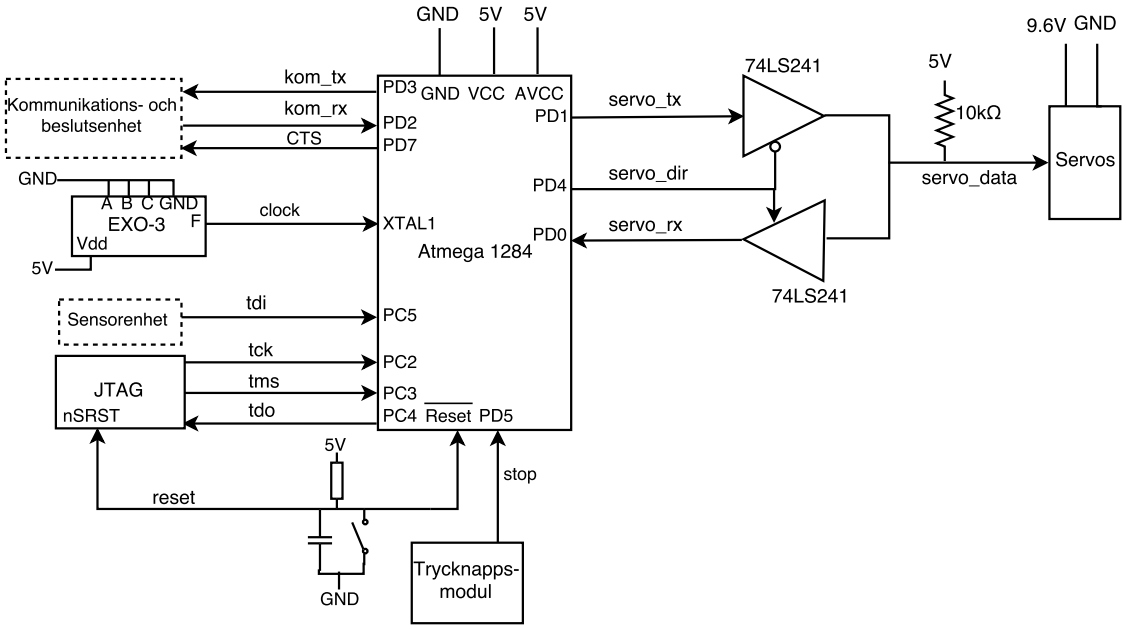
[5] NumPy.org. Numpy.org, http://numpy.org, 2015.

[6] `karulis`. Pybluez, https://github.com/karulis/pybluez, 2015.

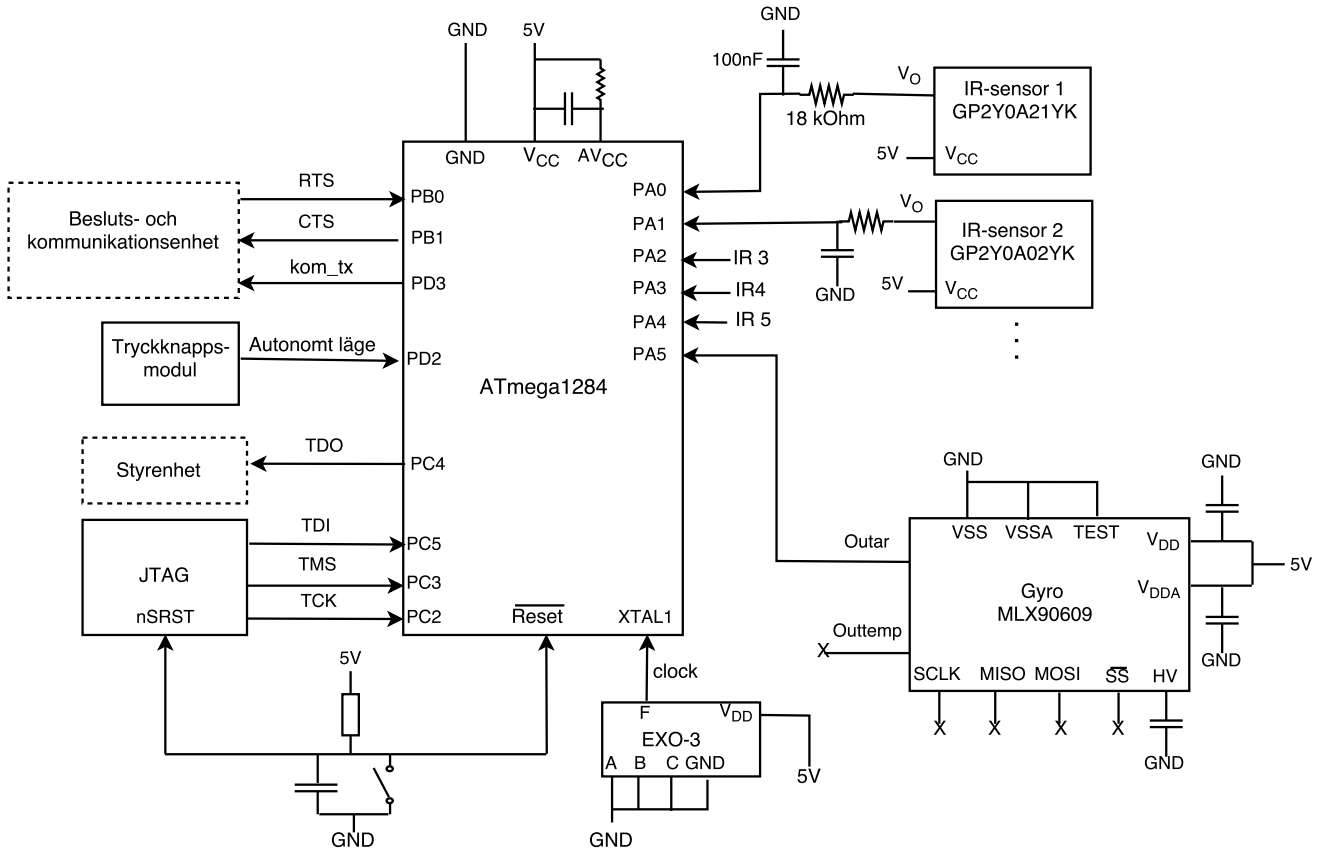
[7] `TrossenRobotics`. Phantom ax metal hexapod mark iii, http://www.trossenrobotics.com/phantomx-ax-hexapod.aspx, 2015.

[8] Wikipedia. Rs-232, https://en.wikipedia.org/wiki/RS-232, 2015.

A Kopplingsschema



Figur 10: Styrenhet



Figur 11: Kopplingsschema för sensorenheten

B Programlistning

Den källkod som skapats tillhandahålls på tre sätt,

- En zip-fil innehållandes all källkod och dokumentation som skrivits,
- En git-repo som även den innehåller all källkod och dokumentation,
- Separata PDF-filer genererade av Doxygen [2], innehållandes både koddokumentation och kodlistning.

B.1 Dokumentation

Funktioner, klasser och filer är i de flesta fall dokumenterade enligt det system som Doxygen rekomenderar. Man har alltså för varje funktion, klass och fil dokumenterat dess funktion, parametrar och returvärde. Dessa kommentarer återfinnes i koden samt blir använda i den genererade dokumentationen.

B.2 Git

Eftersom man använt git finns det en utförlig och komplett historik över projektets utveckling, från första version av Kravspecifikation till slutgiltig kodinlämning. När man står

i projektmappen finns några praktiska kommandon för att ta reda på saker om koden.

- `git blame <filnamn>` Ta reda på vem som *senast* ändrade på saker i en fil. Observera att detta kan vara missvisande om någon t.ex. har ändrat ett mellanrum.
- `git log <filnamn>` Ta fram en komplett historik över vem som har ändrat en fil tillsammans med deras commit-meddelande.
- `git log -p <filnamn>` Ta fram en komplett historik över vem som har ändrat en fil, deras commit-meddelande, samt en komplett diff över förändringarna.

Man kommer enklast åt git-repon genom att antingen packa upp den inlämnade zip-filen eller köra kommandot

```
git clone https://bitbucket.org/henrixh/tsea29.git
```

C Övrigt

C.1 Specifikation av kommunikationsprotokoll

I detta appendix specificeras de kommunikationsprotokoll som används i projektet.

Kommunikations- och beslutsenhet ↔ Stryrenhet För kommunikationen till och från styrenheten används ett binärt protokoll enligt tabell 1 och 2. Kommunikationen går i båda riktningar. Hastigheten på kommunikationen är 76800 bps.

Tabell 1: Format på de datapaket som sänds från styrenheten till kommunikations- och beslutsenheten.

Byte #	Namn	Beskrivning
0	`MAGIC`	Magisk konstant med värdet `0xBB`
1	`CHKSUM`	Checksumma enligt `MAGIC` + $\sum_{i=2}^{38}$ `byte`$[i]$
$2(n+1)+0, n \in [0, 17]$	`SERVO_POS_H_`$[n]$	Position för servo n, bit 8-15
$2(n+1)+1, n \in [0, 17]$	`SERVO_POS_L_`$[n]$	Position för servo n, bit 0-7

Tabell 2: Format på de datapaket som sänds från kommunikations- och beslutsenheten.

Byte #	Namn	Beskrivning
0	`INSTR_ID`	Innehåller ett instruktions-id ifrån tabell 3
1	`CHKSUM`	Checksumma som beräknas enligt `INSTR_ID` + $\sum_{i=2}^{n}$ `byte`$[i]$
$i \in [2, n]$	`PAYLOAD`	Ett antal datapaket, vars antal n och betydelse definieras i tabell 3

Tabell 3: De instruktions-ID som används i kommunikationspaketen från Kommunikations- och beslutsenheten till styrenheten. Varje `uint16` sänds som 2 bytes, den första representerar bit 8-15, den andra representerar bit 0-7.

ID	n	Beskrivning av data
`0`	36	18st `uint16`, representerar absoluta servopositioner
`1`	2	En `uint16`, sätter maximal belastning för alla servon
`2`	2	En `uint16`, sätter maximal hastighet för alla servon
`3`	1	En byte som sätter på och stänger av all servostyrning

Sensorenhet → Kommunikations- och beslutsenhet Kommunikationen är enkelriktad i och med att endast sensorenheten sänder data. Data sänds enligt protokollet beskrivet i tabell 4 med hastigheten 76800 bps.

Datorprogram ↔ Kommunikations- och beslutsenhet Den data som sänds mellan datorprogrammet och roboten är formaterad som JSON-data enligt tabell 5 och 6 . Som programmerare ska man inte behöva röra rå JSON-data, utan detta abstraheras bort med bibliotek för respektive programmeringsspråk.

Tabell 4: Format på de datapaket som sänds från sensorenheten

Byte #	Namn	Beskrivning
0	`MAGIC`	Magisk konstant med värdet `0xAC`
1	`CHKSUM_L`	Checksumma, bit 0-7, enligt `MAGIC` + $\sum_{i=2}^{14}$ `byte`$[i]$
2	`CHKSUM_H`	Checksumma, bit 8-15, enligt `MAGIC` + $\sum_{i=2}^{14}$ `byte`$[i]$
3	`BUTTON`	Knappdata för autonomt/manuellt läge
$2n+4, n \in [0,4]$	`GYRO_L_`$[n]$	Gyrodata nr n, bit 0-7
$2n+5, n \in [0,4]$	`GYRO_H_`$[n]$	Gyrodata nr n, bit 8-15
$2n+14, n \in [0,4]$	`IR_L_`$[n]$	Data från IR-sensor n, bit 0-7
$2n+15, n \in [0,4]$	`IR_H_`$[n]$	Data från IR-sensor n, bit 8-15

Tabell 5: Format på de JSON-paket som skickas från roboten till datorprogrammet.

Nyckel	Typ	Beskrivning
`timestamp`	string	Tidpunkt då paketet skapades
`d_omega`	float	Rotationshastighet
`dx`	float	Rörelsehastighet i x-led
`dy`	float	Rörelsehastighet i y-led
`tick_rate`	float	Den hastighet med vilken man uppdaterar benpositioner
`tick_rate`	float	Den hastighet med vilken man uppdaterar benpositioner
`front`	float	Värde på frontsensorn
`gyro`	float	Värde på gyro
`sides`	[float]	Lista med värden på sidosensorer
`log`	[string]	Lista med de loghändelser som skett sedan förra paketet

Tabell 6: Format på de JSON-paket som skickas från datorprogrammet till roboten.

Nyckel	Typ	Beskrivning
`autonomic`	bool	Välj mellan autonomt och manuellt läge
`velocity`	float	Robotens hastighet i millimeter per sekund
`angular_velocity`	float	Rotationshastighet i millimeter per sekund
`movement_x`	float	Rörelsehastighet i x-led
`movement_y`	float	Rörelsehastighet i y-led
`rotation`	float	Rotationshastighet i radianer per sekund
`set_height`	float	Sätter robotens höjd
`set_torque`	float	Sätter servomotorernas torque
`set_v_kp`	float	Sätt reglerparameter K_p för sidoreglering
`set_v_ki`	float	Sätt reglerparameter K_i för sidoreglering
`set_v_kd`	float	Sätt reglerparameter K_d för sidoreglering
`set_v_sc`	float	Skala alla reglerparametrar för sidoreglering
`set_omega_kp`	float	Sätt reglerparameter K_p för rotationsreglering
`set_omega_ki`	float	Sätt reglerparameter K_i för rotationsreglering
`set_omega_kd`	float	Sätt reglerparameter K_d för rotationsreglering
`set_omega_sc`	float	Skala alla reglerparametrar för rotationsreglering
`set_front_kp`	float	Sätt reglerparameter K_p för frontreglering
`set_front_ki`	float	Sätt reglerparameter K_i för frontreglering
`set_front_kd`	float	Sätt reglerparameter K_d för frontreglering
`set_front_sc`	float	Skala alla reglerparametrar för frontreglering