CHALMERS

Kandidatarbetesrapport


# Självspelande bordshockeyspel

Emil Andersson, Björn Berntsson,
Johan Lindqvist, Teodor Olsson
& Gustav Vidhög

*Automation*
*Institutionen för signaler och system*
Chalmers tekniska högskola
Göteborg, 2012

SSYX02-12-02

# Förord

Årets kandidatgrupp har arbetat med att vidareutveckla det numera anrika självspelande bordshockeyspelet. Projektet kan ses som en intressant automationstillämpning gjord på en vardaglig produkt och innehåller många av de arbetsmoment som en ingenjör kan ställas inför. Vi vill tacka alla som tidigare har arbetat med spelet och därmed varit med att bygga den grund vilket vi arbetat vidare på. Vi vill dessutom rikta ett extra stort tack till vår handledare Martin Fabian för hans hjälp under vårt arbete inklusive korrekturläsning av rapporten.

**Sammanfattning**

Bordshockeyspel har funnits i Sverige sedan slutet av 1930-talet och har idag utvecklats till mer än bara ett sällskapsspel. Årligen spelas numera turneringar runt om i hela världen. Samtidigt har automation blivit viktig, både i industrin och i hemmen, varför det är intressant att designa ett helt automatiserat bordshockeyspel. I denna rapport beskrivs ett sådant spel.

Arbetet har bestått i att ansluta sensorer och motorer till spelet samt skriva olika typer av programvara som tolkar sensorernas signaler och genererar lämpliga signaler till motorerna.

Projektet har resulterat i ett bordshockeyspel som i stort sett kan genomföra matcher på egen hand sånär som på att pucken manuellt måste placeras på spelplanen vid varje tekning.

**Abstract**

Table hockey games were first introduced in Sweden in the late 1930's and have today evolved into something more then just a leisure game. Every year tournaments are played all around the world. As automation has become important in industrial processes as well as in households it would be interesting to design a fully automated table hockey game. This report describes such a game.

The work has consisted of connecting sensors and motors to the game and writing different kind of software that interpret the sensor signals and generates suitable signals for the motors.

The project has resulted in a table hockey game that can carry through games on its own, besides the fact that the puck has to be manually placed on the pitch at every face off.

# Innehåll

# 1 Inledning

År 1938 lanserades det första bordshockeyspelet i Sverige. Det var Aristospel som låg bakom spelet som kallades PUCK. Spelet var konstruerat i trä och masonit och spelarna var gjorda i plåt [1]. Nästan 20 år senare, strax efter att Sverige vunnit ishockey-VM i Moskva 1957, lanserade STIGA sitt första bordshockeyspel. Sedan dess har otaliga versioner av spelet lanserats och STIGA har dominerat marknaden. De har idag en årlig produktion på ungefär 100 000 spel [2].

Idag är spelet och spelarna huvudsakligen gjorda av plast. Varje lag har fem utespelare och en målvakt. Spelare flyttas genom att en styrpinne förs framåt eller bakåt, vrider man styrpinnen roteras spelaren. I Figur 1 visas ett typiskt STIGA bordshockeyspel.

Bordshockeyn, som ursprungligen endast var tänkt som en leksak, har med åren även utvecklats till en sport. Runt om i Europa, och även i Nordamerika, hålls turneringar av olika slag och vartannat år anordnas världsmästerskap [3]. I och med det ökande intresset kring bordshockey föds också innovativa idéer kring hur man kan utveckla spelet. Arbetet med det självspelande bordshockeyspelet har som mål att utveckla ett bordshockeyspel helt styrt av datorer.

Självstyrning och automation är något som blir allt vanligare i dagens samhälle, både inom industrin och i hemmen [4]. Inom industrin kan det exempelvis handla om att göra tillverkningsprocesser billigare och effektivare. I hemmen drivs automatiseringen vanligtvis av människors strävan efter att lägga mindre tid på sådant som egentligen hade kunna skötas automatiskt, så som exempelvis dammsugning och gräsklippning.

## 1.1 Bakgrund

Arbetet med att utveckla det det självspelande bordshockeyspelet påbörjades 2008, och har sedan dess behandlats av tre olika kandidatgrupper. Då projektet övertogs var spelet utrustat med motorer, givare och kamera och den mjukvara som ligger till grund för styrning av spelet hade en god struktur. Arbetet har sedan dess varit inriktat på att finjustera den mekaniska och elektriska uppställningen samt vidareutveckla olika delar av mjukvaran för att göra spelet självspelande.


Figur 1: STIGA PlayOff bordshockeyspel [5].

## 1.2 Syfte och mål

Huvudmålet med projektet är att vidareutveckla och arbeta mot att färdigställa det självspelande bordshockeyspelet. För att nå huvudmålet har flera delmål satts upp:

- Anpassa den befintliga mekaniska uppställningen för att minimera friktion och risk för kollisioner mellan systemets rörliga delar.
- Designa individanpassade regulatorer där varje spelare har sina egna parametrar beroende på rörelsebana och spelstil. Regulatorerna ska kunna styra spelarna till en önskad position med olika hastigheter, för såväl translation som rotation.
- Vidareutveckla serverprogramvaran. Den ska kunna registrera mål, hålla ordning på matchresultat och se till så att spelprogrammen håller sig till spelets regler.
- Utveckla en variant av ett spelprogram som kan köras på hockeyspelet. Spelprogrammet ska ha förmågan att planera och utföra olika handlingar i olika situationer, såsom anfall, försvar m.m. Det spelprogram som utvecklas ska kunna spela en konstruktiv hockey, vilket innebär att det skall försöka vinna matchen samtidigt som det följer reglerna.

För att underlätta för framtida utveckling av det självspelande bordshockeyspelet bör också systemet modifieras så att det är mekaniskt och elektriskt lättöverskådligt. En rapport ska skrivas, vilken ska vara informativ nog för att fungera som en manual för den eller de personer som vill vidareutveckla systemet.

## 1.3 Avgränsningar

Trots att den långsiktiga målsättningen med projektet är att skapa ett helt självspelande bordshockeyspel har kandidatgruppen valt att begränsa sig till att skapa ett system som klarar av att genomföra en match på egen hand sånär som på att sätta pucken i spel vid matchstart, efter mål och då pucken skjutits av spelplanen.

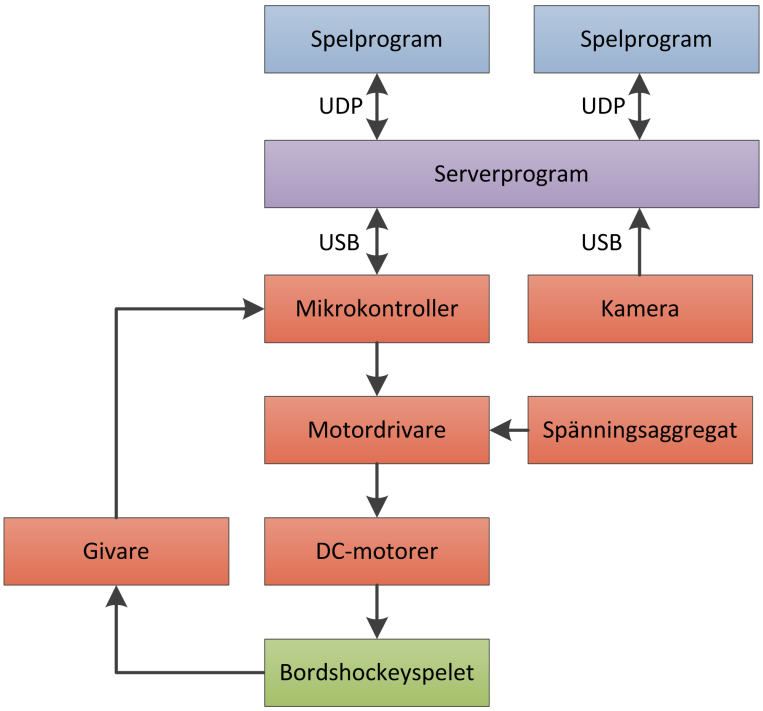
En annan avgränsning som har gjorts under projektets gång är att endast ett av lagen på spelet har automatiserats, detta på grund av att en kortslutning skadade en del elektroniska komponenter under projektets gång och leveransen av reservdelar har uteblivit. Målet är fortfarande att spelet i slutändan ska ha två självspelande lag.

## 1.4 Rapportens disposition

Rapporten inleds med kapitlet *Systemets uppbyggnad* vilket syftar till att ge en överblick av den mekaniska och elektriska uppställningen samt mjukvarustrukturen. Detta följs upp av *Systemegenskaper* där en mer ingående beskrivning av hur systemets olika delar fungerar ges. Kapitlet har också som funktion att spegla det väsentliga utvecklingsarbete som har gjorts på systemet under projektet. I *Utvärdering av systemet* analyseras hur väl spelet uppfyller de uppsatta delmålen och i *Diskussion och slutsats* presenteras våra tankar kring detta. I diskussionen kommenteras också aspekter av systemet vi anser kan förbättras.

# 2 Systemets uppbyggnad

Detta kapitel syftar till att ge en överblick över systemets uppbyggnad, såväl mekaniskt som mjukvarumässigt. I Figur 2 ges en schematisk bild av systemet och dess olika delar, samt hur dessa hänger ihop och kommunicerar.

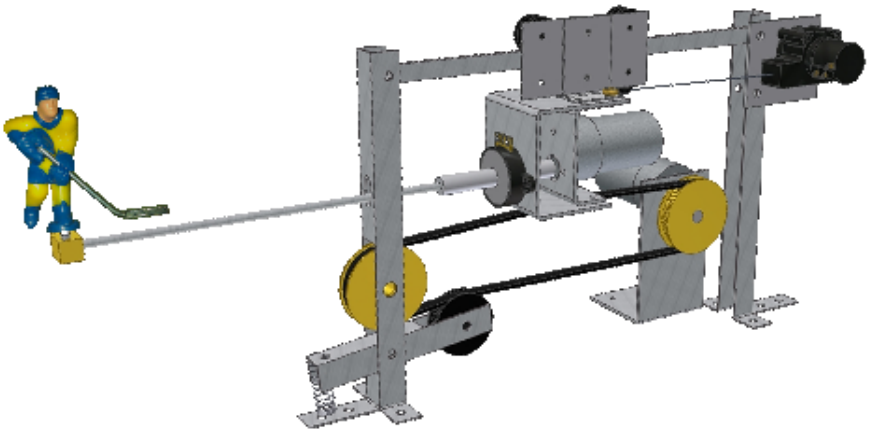



Figur 2: Schematisk bild av systemet. Schemat har olika färg för olika delar av systemet. Grönt och rött indikerar hårdvara, rödmarkerad hårdvara finns beskriven i underavsnittet Komponenter (2.2, sid. 6). Blått och lila representerar programkod.

## 2.1 Mekanisk och elektrisk uppställning

Systemet består i grunden av ett bordshockeyspel (Figur 3) där varje spelares styrpinne är ansluten till två likströmsmotorer. Den ena motorn för spelaren fram och tillbaka och den andra roterar spelaren. Till varje styrpinne är en dragvajergivare och en vinkelgivare ansluten. Motorer och givare sitter monterade på specialkonstruerade motorstativ (Figur 4) för att kunna röra sig fram och tillbaka tillsammans med styrpinnen.


Figur 3: Översikt av den mekaniska uppställningen.



Figur 4: Motorstativ med dragvajer- och vinkelgivare. [6]

Motorer och givare är kopplade till tolv motordrivare respektive två mikrokontrollerkort. Kortens uppgift är att sköta hanteringen av in- och utsignaler till spelet samt regleringen av spelarnas translationer och rotationer, för ett lag vardera.

Motordrivarnas uppgift är att leverera erforderlig ström till motorerna. Drivarna matas med ström från två nätaggregat. Varje motordrivare styr två motorer; translation och rotation för en spelare.

För att detektera spelarnas längsgående position används de potentiometriska dragvajergivarna. Vajergivarna har varierande längd för olika spelare eftersom spelarnas rörelsebanor skiljer sig åt. Exempelvis har målvakten en mycket kortare rörelsebana än en utespelare. Spelarnas vinkel detekteras med hjälp av de potentiometriska vinkelgivarna. En kamera är monterad på ett stativ rakt ovanför spelplanen, dess uppgift är att registrera puckens läge.

Pucken är målad grön för att enkelt kunna filtreras ut i bilden från kameran. Kameran och mikrokontrollerkorten är anslutna till en serverdator.

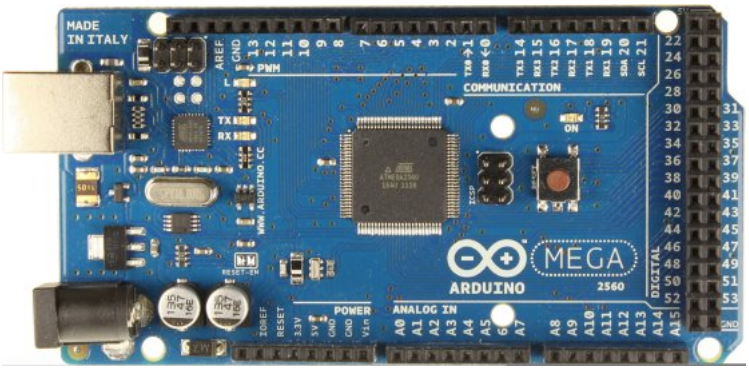
## 2.2 Komponenter

I detta avsnitt ges en utförlig beskrivning av de komponenter som används i systemet, samt deras tekniska specifikationer.

### 2.2.1 Mikrokontrollerkort

De två mikrokontrollerkorten som används i systemet är av typen Arduino Mega 2560 (Figur 5) Korten är baserade på mikrokontrollern ATmega2560 som arbetar med en klockfrekvens på 16 MHz [7]. Korten har 54 digitala anslutningar som kan användas som antingen in- eller utgångar. 14 av de digitala anslutningarna har dessutom stöd för PWM[1]-styrning så att de exempelvis kan imitera analoga utgångar. Det finns utöver de digitala anslutningarna även 16 analoga ingångar. De analoga ingångarna tolkas via en 10-bitars A/D-omvandlare från 0-5 V till värden mellan 0 och 1023. Korten har totalt 4 seriella anslutningar. En av dessa är en USB-anslutning så att det kan kopplas till en dator för kommunikation och programmering.

På korten finns också olika sorters minnen. Det finns ett 256 kB flashminne för att lagra programkod, 8 kB SRAM som används som arbetsminne och 4 kB EEPROM. För att programmera arduinokort används en något modifierad version av C/C++. Färdiga funktioner finns tillgängliga för att förenkla in- och utsignalshantering.




Figur 5: Arduino Mega 2560 mikrokontrollerkort [8].

[1]Se Appendix B, för korta beskrivningar av tekniska begrepp och förkortningar som förekommer i rapporten.

### 2.2.2 Elmotorer

De 24 elmotorer som används har (Figur 6) modellnummer 1.61.046.332 och är tillverkade av Buhler Motor. De är designade för att drivas med en 24 V likspänning [9] och har en växellåda vilken växlar ner med en faktor 9,9. Motorernas märkvärden innefattar bland annat 0,3 A i drivström och ett moment på 70 Ncm vid ett varvtal på 335 rpm.

Figur 6: Borstad elmotor modell 1.61.046.332 [10].

### 2.2.3 Motordrivare

Motordrivarna kommer från Pololu och är totalt 12 till antalet. Modellnumret är Dual MC33887 Motor Driver Carrier (Figur 7). De är designade för att driva två likströmsmotorer parallellt med en drivspänning på mellan 5 V och 28 V [11]. Motordrivarna har inbyggda H-bryggor och möjlighet till att PWM-styra motorerna i upp till 10 kHz. Drivströmmen som motordrivarna kan leverera till de olika motorerna kontinuerligt är 2,5 A men kan under korta perioder gå upp till 5,0 A.

### 2.2.4 Sensorer

I konstruktionen används två olika typer av sensorer. Dragvajergivare används för att detektera vilken position spelarna har och vinkelgivare för att bestämma deras vinkel.

#### Dragvajergivare

De 12 positionsgivarna som används är av modell WPS-MK30, som är en dragvajergivare av potentiometrisk typ från Micro-Epsilon. WPS-MK30 finns att köpa i längder mellan 100 mm och 1250 mm och är bland de minsta inom sitt verksamhetsområde [13]. Givarens hus monteras fast och änden på vajern

Figur 7: Dual MC33887 Motor Driver Carrier motordrivare [12].

fästs i objektet till vilket man vill mäta avståndet. När vajern sedan dras ut ändras resistansen i givaren. Genom att lägga en spänning över denna och läsa av hur utspänningen varierar med den ändrade resistansen kan man enkelt räkna ut hur långt vajern har dragits ut.

Figur 8: WPS-MK30 dragvajergivare [13].

### Vinkelgivare

De 12 vinkelgivarna är av modell WAL 305 5K (Figur 9) och är tillverkade av Contelec. WAL 305 5K är en liten och lätt vinkelgivare av potentiometrisk typ. Vinkelgivaren har två delar, en som monteras fast och en roterande del i mitten vilken träs över en roterande axel. Komponenten har olika resistans beroende på vilken vinkel axeln står i. Genom att lägga en spänning över

givaren och läsa av utspänningen från denna kan man då räkna ut i vilken vinkel axeln står. För ett litet intervall av vinklar tar dock det resistiva materialet i givarna slut och lämnar en lucka, vilken gör att kretsen bryts och ingen utsignal ges för dessa positioner [14].


Figur 9: WAL 305 5K vinkelgivare [15].

### 2.2.5 Kamera

För att detektera pucken används industrikameran DFK 21AUC03 (Figur 10) från The Imaging Source. Kameran filmar i färg med en högsta upplösning av 744x480 bildpunkter och en uppdateringsfrekvens på 60 Hz [16]. Kameran har en 1/3 tums CMOS-sensor och använder sig av ett USB-gränsnitt för kommunikation med en dator.

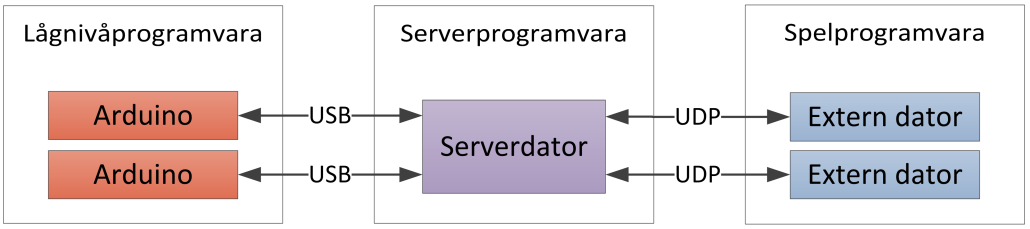

Figur 10: DFK 21AUC03 industrikamera [17].

### 2.2.6 Nätaggregat

För att förse hockeyspelet med ström används två nätaggregat av typen SP-500-15 från Mean Well. Nätaggregaten omvandlar växelström i intervallet 88 V till 264 V till likspänning i intervallet 13,5 V till 18 V men används normalt för att leverera 15 V [18]. Maxströmmen anges som 32 A och därmed kan nätaggregaten leverera upp till 480 W i effekt. För att skydda mot elfel finns finns flera inbyggda skydd, bland annat för kortslutning, överlast, överspänning och överhettning.

## 2.3 Mjukvara

Mjukvarumässigt består systemet av tre nivåer (Figur 11). Lågnivåprogramvara körs på arduinokorten, dessa kommunicerar över USB med en serverdator som i sin tur kommunicerar med två spelprogram över nätverksprotokollet UDP. Detta gör att man kan lyfta ut spelprogram till en extern dator på valfri plats i världen. Dessutom möjliggör det att spelprogram kan implementeras plattforms- och språkoberoende. Varje lag körs av ett eget spelprogram oberoende av det andra. Kommunikationen mellan de olika nivåerna har samma uppbyggnad vilket ger en enkelhet och låter all avancerad beräkning ske i spelprogrammen.




Figur 11: Systemets olika mjukvarunivåer och deras kommunikationsvägar.

### 2.3.1 Lågnivåprogramvara

Lågnivåprogramvaran som körs på de två mikrokontrollerna är skriven i en modifierad version av C/C++ och har som uppgift att sköta insamling och tolkning av sensordata samt reglering av spelarnas positioner, vinklar och hastigheter. Koden på de båda mikrokontrollerkorten är identisk, och det är bara vilket fysiskt kort som koden körs på som avgör om det hanterar hemma- eller bortalag.

Regleringen sköts av PI-regulatorer vilka har designats speciellt för att passa in i tillämpningen av bordshockeyspelet. I mikrokontrollerna finns också en funktion för att kalibrera spelet. Under kalibreringen sparas data om de minimala motorspänningarna som behövs för att flytta varje spelare. Det kontrolleras också var de olika spelarnas rörelsebanor börjar och slutar sett till givarnas värden. All kalibreringsdata sparas i mikrokontrollernas fasta minne och informationen är nödvändig för att spelarna ska gå att styra korrekt.

### 2.3.2 Serverprogramvara

Serverprogramvarans uppgift är att sköta kommunikationen mellan mikrokontrollerna, kameran och spelprogrammen. Dessutom ansvarar den för att sköta bildbehandlingen och puckidentifieringen. I serverprogramvaran finns även algoritmer för att hålla ordning på måldetektion, resultat samt begränsningar för vad som är tillåtet för spelprogrammen. I detta avseendet fungerar serverprogramvaran som en domare.

En vanlig persondator används för att köra serverprogramvaran, vilken är skriven i C++ till viss del mot Windows API. De hjälpbibliotek som används är OpenCV för bildhanteringsfunktioner, IC Imaging Control C++ för att fånga bilder från kameran samt några filer för seriell kommunikation med Arduino och för att enklare hantera nätverkskommunikation.

### 2.3.3 Spelprogramvara

Spelprogramvaran tar emot information om spelets status från serverprogramvaran. Spelprogramvarans uppgift är sedan att med hjälp av denna information bestämma och utföra den bästa spelstrategin för att vinna matchen. Programvaran är skriven i Java och är objektorienterad. Det finns ett antal olika varianter av mer eller mindre ”intelligent” spelprogramvara som bygger på en gemensam bas.

# 3 Systemegenskaper

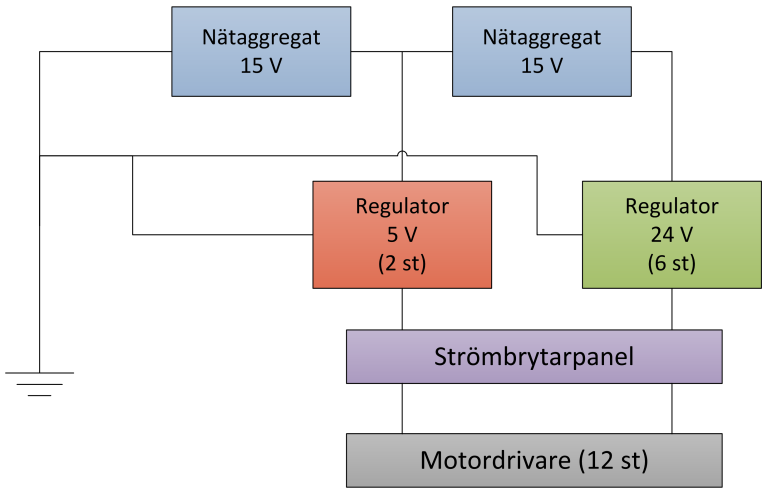
I detta avsnitt ges en mer ingående beskrivning av hur systemet fungerar. I Appendix A presenteras en guide för hur man startar upp det självspelande bordshockeyspelet.

## 3.1 Mekanik och elektronik

För att systemet ska få önskad funktion har det krävts att olika typer av hårdvarumässiga problem har lösts. I detta avsnitt presenteras de lösningar som är av teknisk relevans för projektet.

### 3.1.1 Strömförsörjning

För att förse motordrivarna med erforderlig ström krävs att spänningen från de två nätaggregaten regleras ned. Det behövs en drivspänning på 24V och en spänning på 5V för att sköta kretsarnas logik. 24V-spänningen åstadkoms genom att de två 15Vs nätaggregaten seriekopplas och ansluts till en krets med sex spänningsregulatorer. För att leverera 5V används endast ett av spänningsaggregaten och ytterligare två spänningsregulatorer. Regulatorerna är anslutna mellan aggregatet och en strömbrytarpanel, för att sedan kopplas till motordrivarna. Se Figur 12 för en schematisk bild över spänningsregleringen.




Figur 12: Schematisk bild över strömförsörjningen.

Tabell 1: Mikrokontrollerkortens spelarnumrering.

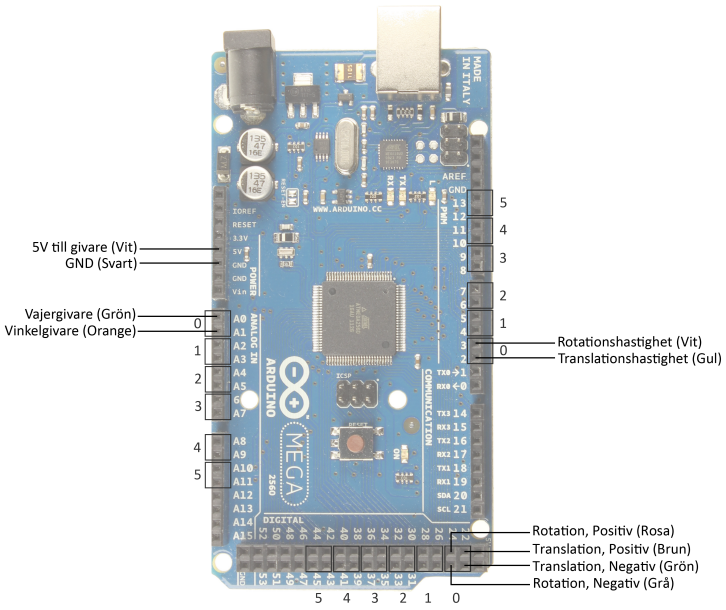
| Nummer | Spelare |
|---|---|
| 0 | Målvakt |
| 1 | Höger back |
| 2 | Vänster back |
| 3 | Höger forward |
| 4 | Center |
| 5 | Vänster forward |

### 3.1.2 Modifiering av vinkelgivare

Som tidigare nämnts (2.2.4, sid. 8) kan vinkelgivarna inte detektera vinklar inom ett visst, litet intervall. För att lösa detta problem har högresistiva motstånd monterats parallellt med givarnas inre resistanser så att givaren ger en utsignal även inom intervallet. Alla vinklar inom intervallet tolkas därmed som samma vinkel, men intervallet är så smalt att detta inte påverkar spelet märkbart.

### 3.1.3 Pinnkarta mikrokontrollerkort

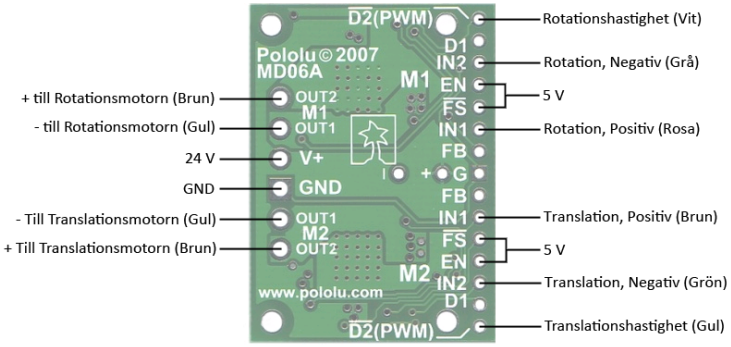
I avsnittet om den mekaniska och elektriska uppställningen (2.1, sid. 5) nämns att vardera lag använder var sitt mikrokontrollerkort. I Figur 13 visas en pinnkarta för ett av mikrokontrollerkorten. Lagets spelare är numrerade 0 till 5 och i Tabell 1 förklaras vilket nummer som hör till vilken spelare. Inom parentes står de anslutna kablarnas färg. Kortet förser lagets tolv givare med en 5V drivspänning och tar emot deras signaler på de analoga ingångarna. Via de analoga utgångarna PWM-styrs lagets tolv motorer, vilka är anslutna till respektive motordrivare. Notera att samma analoga signal används för fram- respektive bakåtdrivning. För att bestämma riktning används två digitala utgångar per motor.




Figur 13: Pinnkarta för ett mikrokontrollerkort. För tydlighets skull har endast kopplingar för målvakten ritats ut.

### 3.1.4 Pinnkarta motordrivare

Pinnkarta för en av målvakternas motordrivare visas i Figur 14, inom parentes står de anslutna kablarnas färg. Motordrivarens $EN$- (enable) och $\overline{FS}$- (fault status) pinnar matas hela tiden med 5V från en av 5V-spänningsregulatorerna. De två $\overline{D2}$-ingångarna är kopplade till de analoga utgångarna på arduinokortet, vilka förser motordrivarna med spänning mellan 0 och 5 volt. Denna signal förstärks med hjälp av en 24V matningsspänning till en spänning mellan 0 och 24 volt vilket är utsignalen från motordrivaren till motorerna. Utgångarna $OUT1$ och $OUT2$ är kopplade till plus- och minuspol på respektive motor. $IN1$ och $IN2$ är ansluta till arduinokortets digitala utgångar vilka förmedlar riktning för motorn.




Figur 14: Pinnkarta för en motordrivare. Målvaktens anslutningar är markerade.

## 3.2 Mjukvara

I detta avsnitt ges en mer utförlig beskrivning av den programvara (Figur 11) som skrivits på respektive programvarunivå. All programvara är uppbyggd av moduler. Dessa moduler samt hur de hänger samman förklaras. Vidare beskrivs de algoritmer som ligger till grund för att det självspelande bordshockeyspelet ska fungera.

### 3.2.1 Lågnivåprogramvara

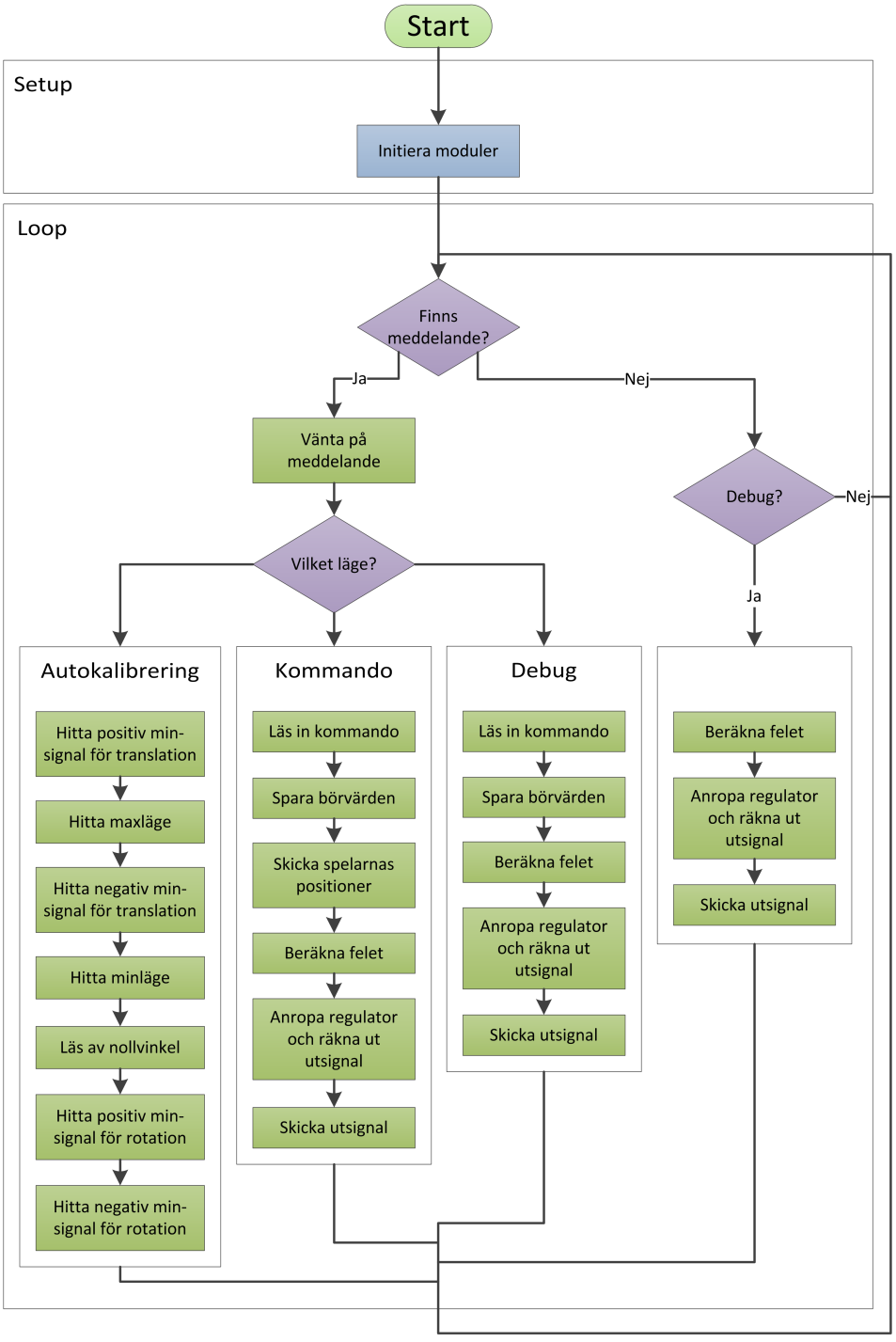
Lågnivåprogramvarans uppgift är som bekant att tolka och styra hockeyspelets in- och utsignaler på maskinnära nivå. De sensorsignaler som arduinons A/D-omvandlare uppmätt som ett värde mellan 0–1023 mappas om till ett värde mellan 0–255 för att informationen ska rymmas i en byte, vilket är praktiskt då denna information ska skickas mellan de olika programmeringsnivåerna. Då vänsterforwarden har en mycket böjd rörelsebana har även speciella villkor lagts in för att korrigera den vinkelvridning som spelaren får, utan att styrpinnen vrids, då den rör sig genom den mest böjda delen.

Programvaran är strukturerad enligt en klassisk struktur för ett Arduinoprogram (Figur 15). Programmet bygger på ett antal filer uppdelade så att varje fil innehåller en modul för programmet. De olika modulerna presenteras i Tabell 2. De två standardfunktionerna Setup och Loop ligger i en separat huvudfil varifrån resten av objekten till programmet initieras.

Det första som görs i Loop är att vänta på att det ska tas emot ett meddelande på den seriella anslutningen. När ett meddelande finns tillgängligt läses den första byten av detta in. Byten förväntas innehålla en bokstav som ska

Tabell 2: Tabell över lågnivåprogramvarans moduler.

| Modul | Beskrivning/användning |
|---|---|
| Player | Klass som har hand om allt som rör en enskild spelare. Har variabler som ärvärden, börvärden, vinkelfel m.m. Player innehåller funktioner som beräknar felen mellan en spelares ärvärde och börvärde, se avsnittet Felberäkning (3.2.1, sid. 20). Klassen innehåller två Driverklasser och två Controller-klasser. |
| Driver | Klass som har hand om att kommunicera med motordrivarna. Ser till att rätt pinnar sätts så att motorn roterar korrekt håll och att rätt spänning skickas till motordrivarna. Driverklassen har även hand om autokalibreringen av spelaren. se avsnittet Autokalibreringsläge (3.2.1, sid. 18) |
| Controller | Klass som är en implementering av en digital PI-regulator, med några små förändringar. Regulatordesignen presenteras i avsnittet Reglerteknik (3.2.1, sid. 22). När klassen initieras sätts regulatorns P- och I-parameter samt samplingstid, sedan används felet och hastigheten som invariabler. |




Figur 15: Arduinoprogramvarans struktur.

berätta i vilket läge programmet ska köras. Det finns tre olika lägen: autokalibreringsläge $a$, kommandoläge $c$ och debugläge $d$. Om det inlästa värdet inte motsvarar någon av dessa lägen går programmet bara vidare. När Loop är slut startar denna om och programvaran återgår till att lyssna efter meddelanden.

### Autokalibreringsläge

I autokalibreringsläget genomför programvaran en kontroll så att dess bild av spelets fysiska tillstånd överensstämmer med verkligheten. Kalibreringen av varje spelare sker genom att det skickas olika signaler till spelet samtidigt som programvaran läser av sensorerna och drar slutsatser. Kalibrering av spelet kan antingen göras från serverprogramvaran eller via konsolen i utvecklingsmiljön för Arduino. För att kalibrera en spelare skickas kommandon på formen

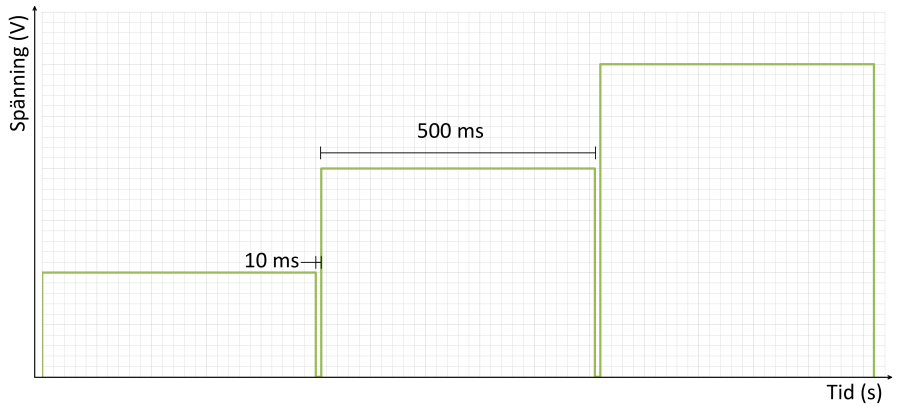
$a$ $[p]$

$p$ betecknar numret på den spelare som ska kalibreras (se Tabell 1 för de olika spelarnas nummer). Om $p$ inte fylls i utan endast ett $a$ skickas kommer spelet att kalibrera alla spelare i tur och ordning med början från spelare 0.

Först kalibreras translationsdrivaren. En kalibrerad translationsdrivare känner till *min-* och *maxlägen* för spelarens bana. Dessutom vet den hur stark signal som krävs för att få motorn att precis börja röra på sig, i såväl positiv som negativ riktning. Dessa kallas *min-signaler*.

För att finna den positiva min-signalen höjs spänningen till motorn stegvis tills dess att den börjar flytta spelaren. Den stegvisa höjningen görs genom att en spänning läggs ut över motorn under 500 ms och därefter tas bort under 10 ms. Spelarens läge kontrolleras och jämförs med läget före spänningen applicerades över motorn. Om spelaren inte har flyttat sig upprepas proceduren fast med en något högre spänning. Denna spänningsstegring illustreras i Figur 16. När spelaren flyttat sig sparas informationen om vilken spänning som krävdes som spelarens positiva min-signal.

När min-signalen för rörelse framåt är funnen körs spelaren framåt på banan. Spelarens läge kontrolleras hela tiden och när detta värde inte längre ökar har spelaren nått sitt max-läge och givarens signal för detta läge sparas. Motsvarande procedur upprepas för bakåtrörelse och den negativa min-signalen och min-läget sparas.




Figur 16: Schematisk bild över spänningshöjningen vid autokalibrering.

När translationsgivaren är färdigkalibrerad påbörjas kalibreringen av rotationsdrivaren. En kalibrerad rotationsdrivare har precis som translationsdrivaren min-signaler för respektive riktning på motorn, men saknar min- och maxlägen då rotationen saknar ändstopp. Rotationsdrivaren har däremot en *nollvinkel*. Nollvinkeln är definierad som den vinkel vid vilken spelaren står med ansiktet vänt rakt mot motståndarlaget. Spelaren ska vara manuellt placerad i denna vinkel innan kalibreringen påbörjats och vinkelgivarens utslag sparas. För att finna min-signalerna för rotation genomförs samma procedur som för translationsdrivaren.

### Kommandoläge

Kommandoläget används av servern för att skicka instruktioner till spelet. Upp till sex kommandon kan skickas samtidigt, och de skrivs som en följd av bytes på formen

$$c\ [p_1\ tv_1\ tp_1\ rv_1\ rp_1\ [p_2\ tv_2\ tp_2\ rv_2\ rp_2\ [...]]]$$

där: $p_i$ betecknar spelarnumret på den spelare som ska styras (se Tabell 1 för de olika spelarnas nummer), $tv_i$ är translationshastigheten för spelaren, $tp_i$ är den önskade translationspositionen för spelaren, $rv_i$ är rotationshastigheten och $rp_i$ är den önskade rotationspositionen. Önskade positioner samt translationshastighet anges som ett tal mellan 0–255, medan rotationshastigheten anges som ett tal mellan -128–127. Rotationshastigheterna 127 respektive -127 särbehandlas och får spelaren att rotera med maximal hastighet i 500

Tabell 3: Format för lågnivåprogramvarans svar.

| Index | Värde |
|---|---|
| 0 | translationsvärde för spelare 0 |
| 1 | rotationsvärde för spelare 0 |
| 2 | translationsvärde för spelare 1 |
| 3 | rotationsvärde för spelare 1 |
| ⋮ | ⋮ |
| 10 | translationsvärde för spelare 5 |
| 11 | rotationsvärde för spelare 5 |

ms i respektive riktning varpå den stannar. Rotationsdestinationen ignoreras således. Detta används för att få spelaren att skjuta utan att behöva definiera vilken vinkel den ska hamna i.

Efter mottaget kommando lagras de mottagna värdena i aktuell spelarklass som nya börvärden. När värdena är satta anropas funktionen update som har hand om att räkna ut felet, skicka detta till regulatorerna och uppdatera styrsignalerna efter given utsignal. Detta förklaras vidare i avsnitten Felberäkning (3.2.1, sid. 20) och Reglering (3.2.1, sid. 22). När ett kommando mottagits svarar programmet med att skicka information om spelets nuvarande status i form av tolv bytes enligt Tabell 3.

### Debugläge

Debugläget möjliggör att enskilda styrkommandon kan skickas till programmet. Kommandon skickas via den seriella konsolen i utvecklingsmiljön för Arduino. Kommandona som skickas har samma struktur som dem som skickas från servern, skillnaden är att bokstaven $c$ byts ut mot $d$, att endast ett kommando kan skickas och att värdena skickas som textsträngar separerade med mellanslag istället för som bytes. I debugläget ges heller inget svar med spelets nuvarande status. En typisk debugrad skickas på formen:

$$dp_1\ tv_1\ tp_1\ rv_1\ rp_1$$

### Felberäkning

Skillnaden mellan en spelares faktiska position vid en tidpunkt $t_i$ och den position spelaren önskas placeras i definierar spelarens fel $e(t_i)$. Felet används

Tabell 4: Felberäkning för rotationsvinklar under de olika möjliga förutsättningarna. Se Figur 17 för en schematisk bild över vinklarna $a$ och $b$.

| Ärvärde | $a$ | $b$ | $a$ |
|---|---|---|---|
| Börvärde | $b$ | $a$ | $a$ |
| Kontroll | börv. $>$ ärv. | börv. $<$ ärv. | börv. $=$ ärv. |
| Positivt fel | $a - b + 255$ | $b - a$ | 0 |
| Negativt fel | $-(b - a)$ | $-(a - b + 255)$ | 0 |

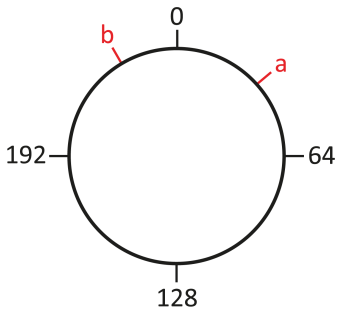
som insignal till systemets regulatorer för att dessa ska kunna korrigera respektive spelares translations- och rotationsläge. Beräkningen av felet sker på olika sätt för translation och rotation.

För translationsrörelse beräknas felet genom att subtrahera värdet för den nuvarande positionen, *ärvärdet*, från värdet för den önskade positionen, *börvärdet*. Ett positivt värde på felet innebär att spelaren ska drivas framåt och ett negativt fel innebär att spelaren ska drivas bakåt.

Rotationsberäkningen är mer komplicerad då spelaren kan nå den önskade positionen genom rotation antingen med- eller moturs. Felet för rotationen beräknas därför på två sätt, ett positivt vinkelfel och ett negativt vinkelfel. Beräkningarna kompliceras vidare av att givaren kan roteras över sin nollvinkel vilket gör att dess utsignal snabbt växlar från 0 till 255 eller tvärt om. I Tabell 4 presenteras hur de båda felberäkningarna utförs genom att kontrollera om givarens nollvinkel kommer passeras under rörelsen. I Figur 17 visas en schematisk bild av en rotationsgivare där två godtyckliga vinklar $a$ och $b$ placerats på vardera sida om givarens nollvinkel.

Så länge spelaren inte har nått den önskade positionen kommer det fel som matchar den önskade rotationsriktningen att skickas till regulatorn, denna riktning definieras av tecknet på rotationshastigheten som skickats till programvaran. Det vill säga om man vill rotera moturs kommer felet som skickas till regulatorn hela tiden att vara det fel som räknas moturs för spelaren.

Om spelaren roteras för långt och alltså passerar det önskade läget, skickas istället det fel vars absolutbelopp är minst till regulatorn. Detta innebär att då spelaren roterar för långt kommer felet att byta tecken och spelaren byter rotationsriktning.




Figur 17: Schematisk bild av en rotationsgivare med nollvinkel samt två godtyckliga vinklar $a$ och $b$.

Tabell 5: Parameterförklaring för ekvation (1)

| Parameter | Beskrivning | Värde |
|---|---|---|
| $u(t_i)$ | Utsignal | 0 – 255 |
| $e(t_i)$ | Fel | -255 – 255 |
| $t_i$ | Diskret tidsparameter | $\mathbb{N}$ |
| $K_p$ | Förstärkning | Konstant |
| $K_i$ | Integrerande del | Konstant |
| $spw$ | Börvärdesvikt | 0 – 1 |
| $spi$ | Integrationsvikt | 0 – 1 |

### Reglerteknik

För styrning av spelarnas translation och rotation används en viktad PI-regulator. Regulatorns uppgift är att hålla felet mellan aktuellt läge och önskat läge så litet som möjligt. Dessutom ska regulatorn vara konstruerad så att felet kan närma sig noll med olika hastighet.

Matematiskt kan regulatorn beskrivas enligt ekvation (1) där parametrarna beskrivs i Tabell 5.

$$u(t_i) = spw \cdot K_p \cdot e(t_i) + spi \cdot K_i \cdot \sum_{\forall i}(e(t_i)) \tag{1}$$

Regulatorn erhåller felet $e(t_i)$ som insignal. Detta används för att beräkna ett utsignalvärde som indikerar hur stor spänning som ska appliceras på aktuell motor under tidssteget $t_i$, som varar tills nästa gång regulatorn anropas. Vikterna $spw$ och $spi$ beräknas enligt ekvation (2) och (3) där $v$ är den

Tabell 6: Tabell över de olika spelarnas individuella regulatorparametrar.

| Spelare | Translation $K_p$ | Translation $K_i$ | Rotation $K_p$ | Rotation $K_i$ |
|---|---|---|---|---|
| 0 | 20 | 1 | 2 | 1 |
| 1 | 30 | 1 | 2 | 1 |
| 2 | 30 | 1 | 2 | 1 |
| 3 | 30 | 1 | 2 | 1 |
| 4 | 50 | 1 | 2 | 1 |
| 5 | 30 | 5 | 3 | 2 |

hastighet som skickats från spelprogramvaran, vilken har värden mellan 0 – $v_{max}$. För translation är $v_{max} = 255$ och för rotation är $v_{max} = 127$.

$$spw = \frac{v}{v_{max}} \tag{2}$$

$$spi = \frac{1}{v + 1} \tag{3}$$

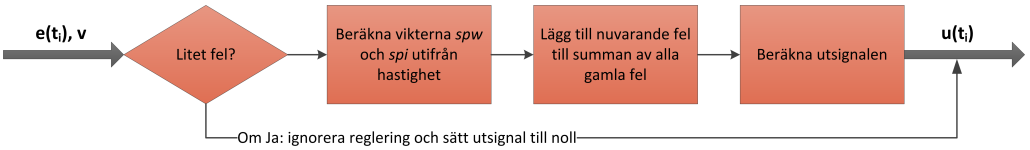
Konstanterna $K_p$ och $K_i$ för såväl rotation som translation är individuellt anpassade för varje spelare och lagrade i en matris i arduinons minne. Dessa har framtagits manuellt genom en trial and error-process där önskade egenskaper för varje spelare testats fram.

För anfallare har aggressivitet prioriterats över risk för översläng. Aggressivitet innebär att utsignalen för translation snabbt ska komma upp i sitt maximala värde så att spelaren snabbt kommer upp i sin högsta hastighet. Detta åstadkoms med högre $K_p$-värde, men kommer på bekostnad av större översläng.

För kantspelare, vilka har en rörelsebana med varierande tröghet, har precision prioriterats. Detta innebär att felminimeringen för translation ska vara effektiv vilket uppnås med högre värde på $K_i$. Ett högre värde på $K_i$ innebär dock att stora fel snabbt summeras upp, vilket ger ett högt värde på utsignalen. Önskas spelaren köras med en låg hastighet leder det höga $K_i$-värdet alltså till en oönskad acceleration. De övriga spelarnas parametrar testades fram med liknande prioriteringar och de satta parametrarna presenteras i Tabell 6.

Ett flödesdiagram för den implementerade PI-regulatorns programvara visas i Figur 18. Notera att i det första steget i varje reglercykel görs en kontroll

om ärvärdet ligger tillräckligt nära börvärdet, i så fall behövs ingen reglering och utsignalen sätts till noll. Här finns möjlighet att definiera hur stora fel som accepteras för translations- respektive rotationspositionering. Fördelen med att inte behandla ett litet fel är att brussignaler från givarna ej aktiverar regulatorn.




Figur 18: Flödesschema för den implementerade PI-regulatorn.

### 3.2.2 Serverprogramvara

Serverprogramvaran har många uppgifter, bland annat kommunikation, bildbehandling och implementering av spelbegränsningar. Hur dessa uppgifter behandlas av programvaran beskrivs i detta avsnitt.

De moduler som bygger upp serverprogramvaran beskrivs i Tabell 7 och en schematisk bild av hur de hänger samman visas i Figur 19. De bibliotek som används i serverprogramvaran beskrivs i Tabell 8.

Tabell 7: Tabell över serverprogramvarans moduler och en kortfattad beskrivning av deras uppgifter.

| Modul | Beskrivning |
|---|---|
| HockeyManager | Innehåller programmets main-funktion. Ansvarar för kommandoradsgränssnittet samt initierar HockeyGame. |
| HockeyGame | Hanterar hockeymatcherna, initierar övriga moduler och definierar kopplingarna mellan vissa av dessa. |
| GUI | Ansvarar för det grafiska användargränssnittet. |
| CameraCalibration | Ett grafiskt gränssnitt för att enkelt kalibrera kameran för puckdetektion. |
| Team | Innehåller en klass som representerar ett lag av spelare. |
| Player | Innehåller en klass som representerar en spelare, dvs. dess position samt rotation. *Translation/Position* är translationsgivarens värde. *Location* är spelarens position på planen i koordinater. *Rotation* är rotationsgivarens värde. |
| Puck | Innehåller funktioner för kamerahantering, puckidentifiering, identifiering av målburarna och detektion av gjorda mål. Modulen innehåller också funktioner för att hämta puckens koordinater. |
| Gametime | Ansvarar för tiden i spelet. |
| Limits | Kontrollerar att kommandot från spelprogramvaran är godkänt, att det spelas konstruktiv hockey samt att ett mål är godkänt. |
| TeamConnection | Ansvarar för UDP-kopplingen till de två spelprogrammen. |
| MicroControllers | Hanterar kommunikationen med de två mikrokontrollerkorten. |
| ObjectTracker | Hjälpklass som hittar positionen av ett objekt i en bild från kameran. |
| CamCapture | Hjälpklass som initierar kameran och hämtar bilder från den. |
| SerialConnection | Implementerar protokollet för kommunikation med mikrokontrollerkorten. |
| Mask | Hjälpklass som används för att hantera bildmasker. |
| SerialClass [19] | Hjälpklass för seriell kommunikation med Arduino. |
| PracticalSocket [20] | Hjälpklass för nätverkskommunikation. |

Tabell 8: Tabell över de bibliotek som används av serverprogramvaran.

| Bibliotek | Användning |
|---|---|
| IC Imaging Control C++ [21] | För att ta foto med kameran. |
| OpenCV [22] | Används för all bildhantering, samt för det grafiska gränssnittet. |

### Kommunikation

Serverprogramvaran kommunicerar med mikrokontrollerkorten på det sätt som står beskrivet i kommandoläget i lågnivåprogrammeringen (3.2.1, sid. 19). Detta sker 60 gånger per sekund, även om inte spelprogramvaran skickat något nytt kommando, för att hela tiden få tillbaka spelets nuvarande status. Vilken arduino som tillhör hemma- respektive bortalaget bestäms av vilken COM-port den är ansluten till.

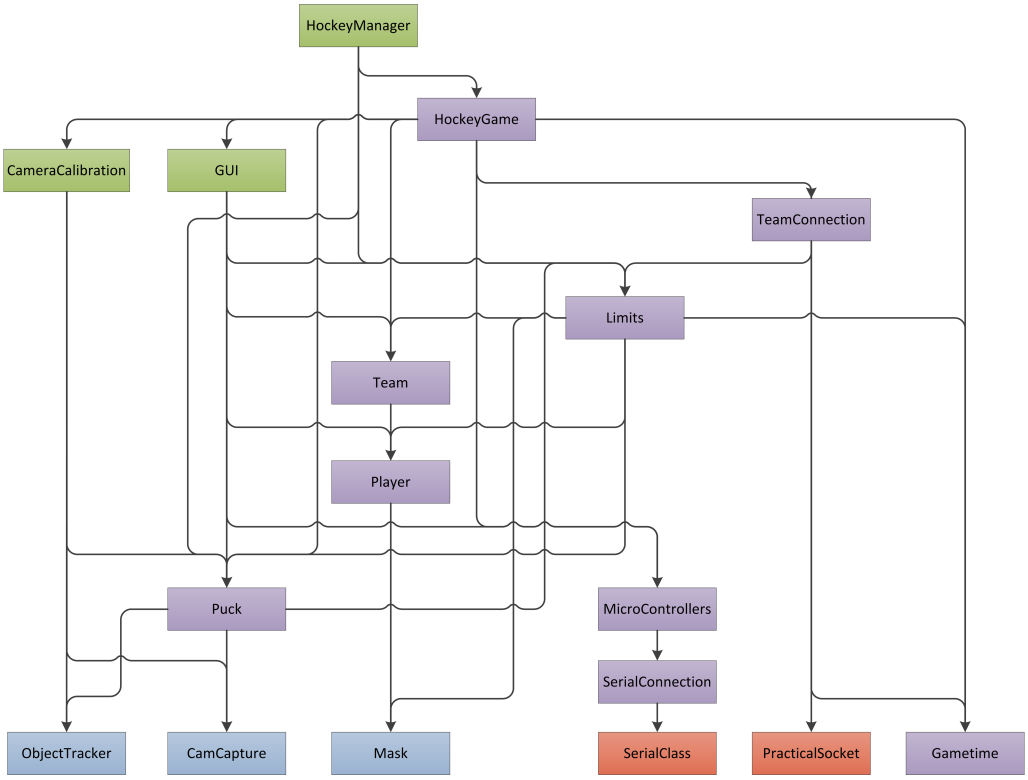
Kommunikationen med anslutna spelprogramvaror sker över UDP. Servern har en tråd som lyssnar på inkommande anslutningar på port 60040. Först väntar den på att hemmalaget ansluter och sedan på bortalaget. Dessa identifieras genom att de ansluter från olika adresser (IP-adress och port) och spelprogramvarorna får reda på vilket lag de är genom en handskakning. Denna handskakning går till genom att servern skickar heltalet 1 till hemmalaget respektiva heltalet 2 till bortalaget. När bägge lagen har anslutit börjar servern att lyssna på kommandon från, samt skicka ut spelets status till, spelprogramvarorna när den fått uppdateringar från mikrokontrollerkorten.

Kommandona skickas till port 60040 på samma sätt som i kommandoläget i lågnivåprogrammeringen. Dock skickas de som heltal (integers) istället för bytes och lägesboktstaven *c* utesluts. Spelets status skickas som 29 heltal till vardera lag i det format som presenteras i Tabell 9.

UDP är förbindelselöst vilket innebär att sändare och mottagare är omedvetna om varandra. För att komma runt problem som kan uppstå som följd av detta finns i serverprogramvaran ett så kallat *heartbeat-protokoll* implementerat som var tionde sekund skickar en förfrågan till spelprogramvarorna om de lever i form av tecknet 'D'. Om svaret 'N' inte fås inom en sekund skickas förfrågan igen upp till totalt fem gånger varpå matchen avslutas om kontakt ej uppnås.

Tabell 9: Formatet för kommunikationen av spelets status.

| Index | Värde |
|---|---|
| 0 | egna lagets antal mål |
| 1 | andra lagets antal mål |
| 2 | puckens x-koordinat |
| 3 | puckens y-koordinat |
| 4 | speltiden i millisekunder sedan matchstart |
| 5 | egna lagets translationsvärde för spelare 0 |
| 6 | egna lagets rotationsvärde för spelare 0 |
| 7 | egna lagets translationsvärde för spelare 1 |
| 8 | egna lagets rotationsvärde för spelare 1 |
| ⋮ | ⋮ |
| 15 | egna lagets translationsvärde för spelare 5 |
| 16 | egna lagets rotationsvärde för spelare 5 |
| 17 | andra lagets translationsvärde för spelare 0 |
| 18 | andra lagets rotationsvärde för spelare 0 |
| ⋮ | ⋮ |
| 27 | andra lagets translationsvärde för spelare 5 |
| 28 | andra lagets rotationsvärde för spelare 5 |



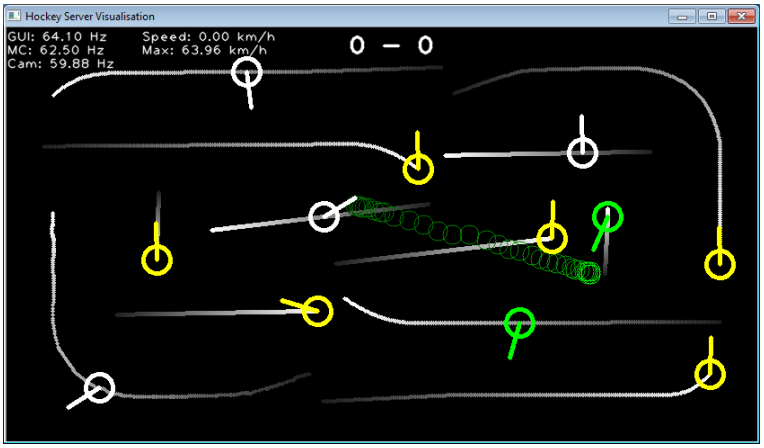

Figur 19: Schematisk bild av serverprogramvarans moduler och hur dessa hänger samman. Gröna moduler innehåller kod för användargränssnitt. Lila moduler innehåller kod specifikt konstruerad för hockeyspelet. Blå moduler är generella hjälpklasser som används av den övriga koden. Röda moduler är generella hjälpklasser som är skrivna av externa parter.

### Grafiskt gränssnitt

För att övervaka spelet finns ett grafiskt gränssnitt (Figur 20) som ger en överblick över programvarans tolkning av alla spelares translations- och rotationslägen, puckens position samt puckens tidigare positioner. Dessutom skrivs matchresultatet, puckens hastighet samt information om olika uppdateringsfrekvenser ut. Det grafiska gränssnittet är användbart för felsökning av sensorer och kamera, samt de algoritmer som tolkar om signalerna från dessa till digitala värden för spelar- och puckpositioner.

### Begränsningar för spelprogram

För att anpassa systemet efter de regler som finns för bordshockey [23] har ett antal spelbegränsningar implementerats i serverprogramvaran. För att




Figur 20: Bild av det grafiskt gränssnitt. Sveriges spelare är gula, finlands vita. Gröna spelare är sådana som har pucken inom räckhåll. Puckens läge visas som gröna cirklar.

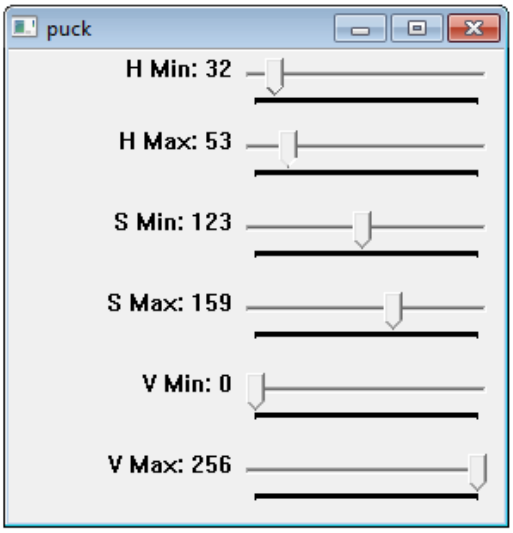
göra matcherna mer lika de matcher människor spelar har begränsningen att maximalt två spelare per lag kan vara i rörelse samtidigt införts. Denna begränsning kan enkelt justeras så att fler spelare kan röra sig samtidigt om så önskas. De kommandon som rejekteras returneras i sin helhet till spelprogramvaran.

För att förhindra att något av spelprogrammen maskar, det vill säga bryter mot femsekundersregeln [23], har en tidsräkningsmetod implementerats. Denna kontrollerar om pucken befunnit sig inom ett lags spelares zoner i längre tid än fem sekunder utan att någon gång varit inom räckhåll för någon motståndare. Om några spelare bryter mot femsekundersregeln vidtas i dagsläget ingen mer åtgärd än att dessa spelare markeras röda i det grafiska gränssnittet.

Det bedöms även om målen är godkända enligt svenska bordshockeyförbundets regler: Mål som görs inom tre sekunder efter matchstart eller tekning är ogiltiga. Har pucken hela tiden befunnit sig inom räckhåll för centern, tills att den kommer in i målgården, för att sedan gå i mål, är mål ogiltigt även efter tre sekunder. Detta gäller även efter målvakts- eller målbursretur [23].

### Kamerakalibrering

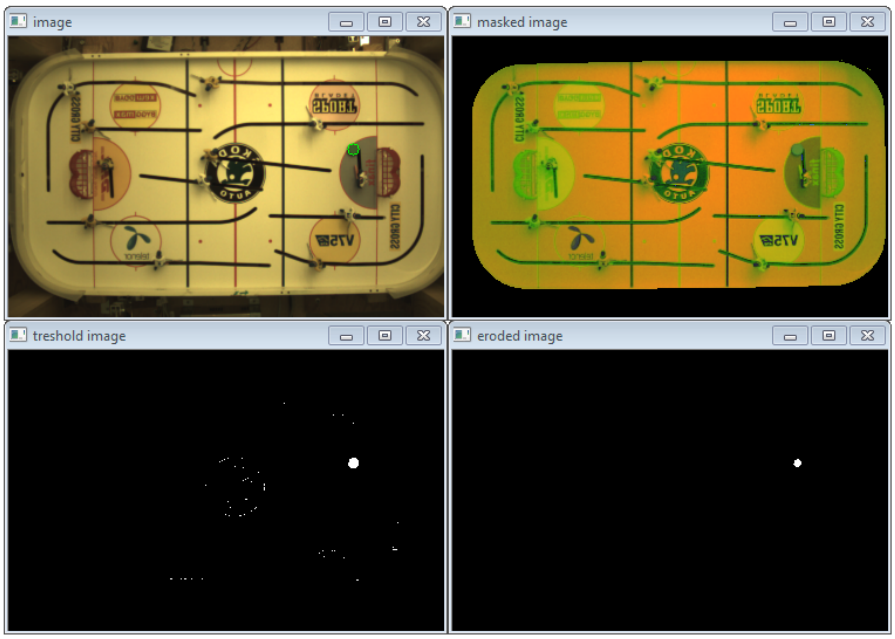
Kameran kalibreras för identifiering av puck respektive de bägge målen med hjälp av ett grafiskt gränssnitt (Figur 21) där man ställer in minsta och största värden för H, S och V i HSV-färgsystemet.




Figur 21: Kalibreringsgränssnitt för kameran med minsta respektivt största värden för H, S och V.

### Bildbehandling och puckidentifiering

För att identifiera pucken respektive de bägga målburarna ur bilden från kameran används ett bildbehandlingsprogram som bearbetrar bilden i flera steg med funktioner ur OpenCV. Först görs bilden om från att vara representerad i RGB till att vara representerad i HSV. Sedan används en bildmask för att klippa bort de delar i bilden som är utanför spelplanen och således ej är intressanta. Därefter anropas funktionen *cvInRangeS* som tar bort de pixlar som inte uppfyller kraven från kamerakalibreringen och de pixlar som ej tas bort görs vita. Efter detta anropas funktionen *cvErode* som så gott som eliminerar bruset i bilden genom att sätta varje pixel till minimivärdet av sig själv samt dess intilliggande pixlar. För att finna puckens position körs funktionerna *cvMoments* följt av *cvGetSpatialMoment* och *cvGetCentralMoment*. Kortfattat beräknar dessa medelvärdet av alla vita pixlars positioner i bilden och kan således användas för att plocka fram koordinater för puckens mittpunkt. I Figur 22 visas bildbehandlingens olika steg.




Figur 22: Bildbehandlingens fyra steg.

### Måldetektion

Om pucken ej längre går att finna i bilden från kameran aktiveras algoritmen för detektion av gjorda mål. Denna använder puckens senaste två kända punkter till att ta fram en ekvation för dess rörelse på räta linjens form. Om denna rörelse är riktad mot ett av målen samt skär dess virtuella mållinje bedöms det som ett mål. Algoritmen är mycket enkel och går att lura, men har visat sig fungera väl.

### Koordinater

Koordinatsystemet som används är relativt planens mitt i pixlar som sett av kameran. Detta motsvarar nästan en pixel per millimeter. Dock inte överallt på planen, då bilden från kameran är något förvrängd. För att lösa detta används bilden från kameran som referens, istället för att omvandla puckens detekterade position till verkliga millimeter. Detta gör att det inte går att mäta saker på hockeybordet för att beräkna olika positioner, utan man får istället mäta i en bild från kameran.

#### Filformat spelarbanor

Spelarnas translationsvärden översätts till koordinater med hjälp av filerna *player00.txt-player05.txt* för hemmalaget (Sverige) respektive *player10.txt-player15.txt* för bortalaget (Finland). Filerna består av ett godtyckligt antal rader av nedanstående format som beskriver spelarens rörelsebana uppdelad i punkter.

$x$ $y$ [*translationsvärde*]

Translationsvärdet är valfritt och om det inte anges kommer de olika translationsvärdena att delas upp jämnt längs banan. De flesta av banorna på bordshockeyspelet är tillräckligt linjära för att translationsvärdet ej ska behöva anges, utan det är främst till för spelare fem i respektive lag (anfallaren som går bakom motståndarens mål) då den spelaren har en led som gör hans rörelse olinjär i samband med kurvan.

### 3.2.3 Spelprogramvara

Spelprogramvarans uppgift är att skicka kommandon till servern för att styra spelarna på önskat sätt. Kommandona skickas via UDP (Figur 11) som en sekvens av heltal[2] på det sätt som beskrivs i avsnittet om kommunikation (3.2.2, sid. 26). Det finns tre olika spelprogramvaror som kan anslutas till servern, dessa är skrivna i Java.

Det finns ett terminalbaserat styrgänssnitt (ConsoleAI) där man skickar ett kommando till servern åt gången, samt ett enkelt självspelande spelprogram (SimpleAI) där spelarna tar sig till pucken och slår till den utan att sikta eller ta några ”avancerade beslut”. Slutligen finns ett mer avancerat spelprogram (PlanningAI) som planerar nästa handling beroende på hur spelare och puck är placerad. Detta spelprogram har möjlighet att få spelarna att föra pucken och lägga den till rätta, passa och skjuta. De tre spelprogrammen beskrivs mer utförligt i varsitt underavsnitt.

Gemensamt för alla spelprogram är att de använder sig av ett paket som heter *Core*. Givet adress och port från ett implementerat spelprogram försöker programmet ansluta till servern. Om anslutningen lyckas fortsätter programmet att ta hand om kommunikationen genom att svara på heartbeatförfrågningar och spara undan den spelinformation servern sänder. I samband med att spelinformationen uppdateras anropas den abstrakta metoden *update* som måste

[2]Heltal representeras, i all kommunikation, av 32-bitars little endian integers.

vara implementerad i spelprogrammet. För att skapa ett spelprogram med Core-paketet ska klassen *AIBase* ärvas. I den ärvda klassens konstruktor ska konstruktorn från AIBase anropas med adress till servern och det portnummer man vill att spelprogrammet ska lyssna på. Den nya klassen bör också implementera den abstrakta metoden onNewState som anropas varje gång AIBase uppdaterar sin spelinformation.

### Terminalbaserat styrgränssnitt (ConsoleAI)

Det terminalbaserade styrgränssnittet låter användaren manuellt styra spelare genom att skicka kommandon innehållande de fem värdena beskrivna i Kommandoläge (3.2.1, sid. 19) separerade med mellanslag. Detta testar hela kommunikationsvägen från spelprogramvarans basklasser via serverprogramvara till mikrokontrollerna. Detta spelprogram är väldigt användbart vid testning och felsökning av systemet.

### Enkelt självspelande spelprogram (SimpleAI)

Spelprogramvaran beräknar utifrån tillhandahållen information om puck och spelares lägen ut var varje spelare ska placera sig för att vara så nära pucken som möjligt. Om pucken är inom räckhåll för en spelare roteras denna spelare medurs med maximal hastighet.

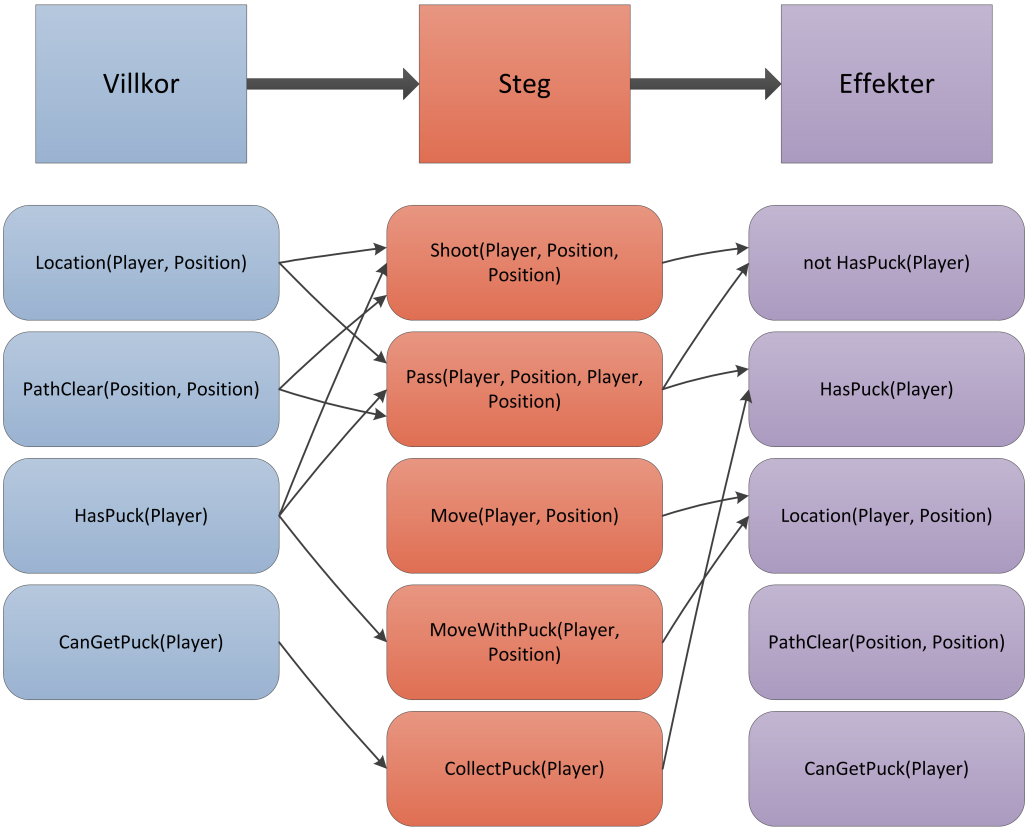
### Planeringsspelprogram (PlanningAI)

Planering är en speciell gren inom AI som handlar om att planera och exekvera planer [24]. I planeringsspelprogramet har en så kallad *continuous planning* implementerats. Detta innebär att agenten, som programvaran kallas, hela tiden är mitt i en plan [25]. Agenten sätter kontinuerligt upp nya mål, förfinar planen som redan finns och exekverar steg ur planen när den hittar ett sådant som kan utföras. När planen exekverats och målet uppfyllts börjar programvaran på en ny plan med ett nytt mål.

För att representera spelets fysiska status används logiska tillstånd, så kallade *conditions*. Dessa representerar agentens tolkning av det nuvarande spelläget, det kommande spelläget eller det läge agenten vill uppnå. Exempel på tillstånd kan vara ”spelare fyra har pucken” eller ”spelare två har position 100”.

För att representera steg i planen, *actions*, behövs tre saker: en lista som beskriver vilka villkor (på tillståndet) som måste vara uppfyllda för att det ska gå att utföra steget, en lista av effekter som beskriver hur tillståndet

kommer att ändras som följd av steget, samt eventuella variabler som beskriver själva steget. Exempel på steg kan vara ”skjut” eller ”passa”. För vardera sådant steg behövs alltså variabler som beskriver steget i form av vilken spelare som berörs och vilka positioner som gäller. I Figur 23 presenteras de steg som finns implementerade i spelprogrammet, vilka villkor som ställs för dessa och vilka effekter de har.




Figur 23: Schematisk bild över planeringsspelprogrammets steg, med tillhörande villkor och effekter.

I planen finns hela tiden ett steg utan effekter som representerar målet som ska uppnås. Det finns också ett *nu-steg* utan förhandsvillkor där effekterna hela tiden uppdateras med de tillstånd som observerats av systemet och förmedlats spelprogrammet via servern. Agentens uppgift är således att stänga gapet mellan startstegets effekter och målstegets förhandsvillkor genom att placera lämpliga steg i planen.

En ordning måste upprätthållas mellan stegen i planen. Inget steg får ske före nu-steget och inget steg får ske efter målsteget. Steg vars effekter uppfyller villkor hos andra steg måste ske före dessa. Steg som negerar ett villkor får ej ske mellan två steg som är kausalt sammankopplade av detta villkor. Ibland spelar ordningen ingen roll och då lämnas den odefinierad. Denna typ av planrepresentation kallas *delvis ordnad plan* [26].

En annan del i uppbyggandet av planen är att förkasta steg vars effekter av någon anledning inte längre är nyttiga. Detta gör att slumpartade, fördelaktiga händelser kan utnyttjas.

För många situationer kan det vara svårt att avgöra vilket steg som lämpar sig bäst för att uppnå ett villkor. Som exempel kan tas när en spelare ska skjuta. Ett av villkoren för att kunna göra detta är att spelaren ska ha pucken. Det finns två sätt att uppfylla villkoret. Om pucken finns nära spelarens bana kan spelaren åka till den, men annars måste någon passa honom. Det är svårt att ta ett bra beslut direkt, varför vissa variabler lämnas öppna och binds så sent som möjligt. Valet kommer då göras med så mycket information som möjligt.

Stegen exekveras genom att en serie kommandon skickas till servern. Mellan varje skickat kommando väntar programvaran tills att spelet har utfört den önskade rörelsen. Efter att steget är utfört återgår programmet till att planera nästa handling.

Om pucken någon gång hamnar utanför lagets spelares räckvidd hamnar agenten i tillståndet *defence* där målvakt och utespelare ska försvara tills någon av dem kan ta pucken igen.

# 4 Utvärdering av systemet

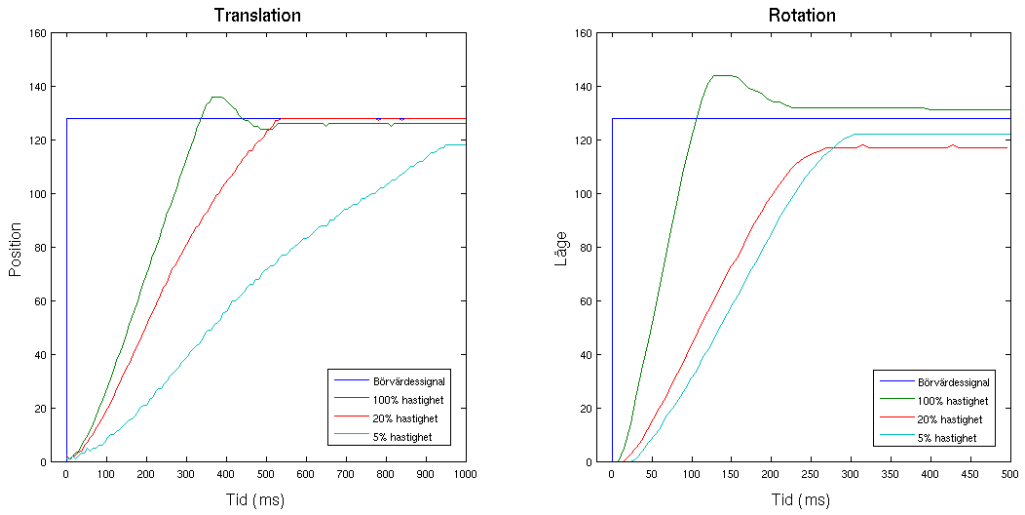
Den mekaniska uppbyggnaden av systemet, med givare och motorer monterade på motorstativ, fungerar relativt väl och risk för kollision mellan systemets rörliga delar är i princip eliminerad. Spelarnas rörelse är i många fall fortfarande mycket trög, och vissa av spelarnas styrpinnar är sneda vilket gör att trögheten varierar beroende på vilka positioner spelarna flyttas mellan. Denna varierande tröghet hanteras dock tillfredsställande av regulatorerna, och är därför inget stort problem.

De komponenter som används i systemet lämpar sig väl för sina respektive uppgifter, även om motorerna är något lågt växlade, vilket innebär att de hastigheter spelarna kan drivas fram med är något lägre än vad som är önskvärt för att göra optimala passningar och avslut. På grund av att endast den ena sidan av hockeyspelet har automatiserats kan inte två spelprogram spela mot varandra, utan den ena sidan måste styras manuellt.

Den viktade PI-regulator som implementerats för att styra spelarnas translation och rotation fungerar bra. Spelarna kan, trots den varierande trögheten, styras med tillräckligt god precision för att en spelprogramvara ska kunna användas för att styra spelet. I Figur 24 visas stegsvar för en spelare då den körs från ett läge till ett annat med olika hastigheter. Att notera är att de hastigheter som anges för att styra spelaren ej svarar linjärt mot den hastighet spelaren faktiskt rör sig med. Detta beror delvis på att parametrarna $K_p$ och $K_i$ för spelarna ej valts på bästa möjliga sätt. Om en rörelse med låg hastighet önskas nås inte alltid den önskade positionen. För högre hastigheter fungerar regleringen mycket bättre.

Serverprogramvaran fungerar bra och kommunikationen sköts på ett effektivt sätt. Implementation av perioder och övrig matchstruktur är inte genomförd vilket innebär att en fullständig match ännu inte kan spelas.

Spelprogrammens grundpaket Core fungerar väl och gör det lätt att utveckla spelprogram. Spelprogrammet PlanningAI har en i stort sett väl fungerande planering, där planer som leder till målchanser exekveras. Den typ av planering som spelprogrammet PlanningAI implementerar fungerar bra med det övriga systemet, då bra information om hockeyspelets tillstånd kan erhållas men spelare och puck inte kan garanteras bete sig som väntat. När verkligheten avviker från planen kan agenten modifiera planen när det går eller göra en helt ny plan om det är bättre. Spelprogrammet saknar än så länge förmågan att passa pucken inom laget, men grunderna för att utveckla sådan funktio-




Figur 24: Stegsvar för translation respektive rotation med olika hastigheter för bortalagets vänsterback.

nalitet finns. Spelprogrammet tar heller inte hänsyn till alla regler som är implementerade i serverprogramvaran, exempelvis att mål inom tre sekunder från tekning är ogiltiga, varför spelprogrammets planering möjligen inte är så bra på att hantera matchens omedelbara början. I exekveringen av steg finns flera buggar. Exempelvis kan spelare i vissa fall rotera åt fel håll eller till fel position. Detta beror på att de order som skickas ibland har fel syntax. Det händer också att programmet låser sig i väntan på att steg utförs. Detta kan bero på att spelprogrammet skickar ett kommando med en önskad hastighet som inte är tillräckligt hög för att få spelaren att nå ända fram till pucken, att kommandopaketet som skickas via UDP inte kommer fram till serverprogramvaran eller på buggar i spelprogramvaran. Spelprogrammet är endast testat på bortalaget. För att det ska fungera för hemmalaget kan vissa modifieringar behövas, främst i valen av vinklar. Överlag får detta spelprogram dock anses spela en konstruktiv hockey och vara kapabelt att genomföra en match.

# 5 Diskussion och slutsats

Det har konstaterats att det fungerar bra att konstruera ett självspelande bordshockeyspel enligt modellen som beskrivs i denna rapport, och vi är nöjda med vårt resultat. Dock finns det många saker att vidareutveckla med systemet.

Högre växlade motorer skulle kunna ge högre maxhastigheter, vilket skulle lösa tidigare nämnda problem (4, sid. 36).

I dagsläget används PI-regulatorer för att styra spelarnas positioner och rotationsvinklar. För att kunna styra spelarna med olika hastigheter har olika vikter införts i regulatorerna vilka skalar ner utsignalens storlek beroende på hastighetsvariabeln. Även om de bidrar till att spelaren når börvärdet långsammare kommer hastigheten inte att vara konstant genom hela rörelsen. För att i framtiden få denna funktionalitet bör regleringen byggas om så att den baseras på en typ av kaskadreglering där det ingår både en hastighetsregulator och en positionsregulator.

Att använda UDP för nätverkskommunikationen var ingen självklarhet utan valet stod mellan TCP och UDP. UDP valdes ändå av flera anledningar. För det första är det är snabbare än TCP då det saknar många av TCPs kontrollmekanismer. För det andra är det i fallet med kommunikationen av spelstatus från serverprogramvaran till spelprogramvaran ointressant om ett paket försvinner med tanke på att det kommer ett nytt paket med uppdaterad information 1/60 sekund senare. När det gäller kommunikationen av kommandon från spelprogramvaran till serverprogramvaran beror det på hur spelprogramvaran är konstruerad, vilket av protokollen som är bäst. Är det en spelprogramvara som hela tiden skickar nya kommandon gör det enligt samma resonemang som ovan inte så mycket om ett paket försvinner men om spelprogramvaran skickar ett kommando i taget och väntar på att detta fullföljs kan det vara ett problem och TCP skulle vara en bättre lösning.

Något som skulle kunna förändras i planeringen är att som mål för programmet försöka göra mål snarare än att försöka göra skott på mål, eftersom det skulle kunna leda till bättre planering av varifrån man ska skjuta. Dessutom borde skottposition, och kanske val av skytt, lämnas öppna och bindas så sent som möjligt precis som görs vid passningar. Planeringen bör även ges ansvar för försvaret, genom att implementera defensiva villkor, eller på något sätt använda sig av redan befintliga metoder. Om man också implementerar delmål skulle ett delmål kunna vara att ha pucken. Delmål gör att man inte

behöver färdigställa hela planen innan man börjar agera, utan istället kan sträva mot delmål oberoende av slutmålet.

Ett självspelande bordshockeyspel är ganska svårt att analysera ur ett hållbarhetsperspektiv. Steg har dock vidtagits för att göra bordshockeyspelet så energieffektivt som möjligt. Att automatisera spelet i sig har kanske inte någon större effekt i strävan mot en hållbarare framtid, men om man börjar använda spelets kamera till att strömma matcherna i realtid på nätet kommer människor som annars hade behövt resa till varandra för att spela mot varandra i stället kunna spela mot varandra på distans. Då spelet med sin nuvarande struktur kan styras av en dator från var på jorden som helst kan det även ses som en tillämpning i ur man styr saker på distans så att man slipper resa dit och medverka fysiskt.

# Referenser


[1] Ahlborn K. et al. (1995). *Svenska Ishockeyspelet.* Stockholm: Strömbergs bokförlag

[2] Svenska Bordshockeyförbundet. *Historik.* `http://www.bordshockey.net` (2012-03-01)

[3] International Table Hockey Federation. *Tournaments* [elektronisk] tillgänglig på: `http://ithf.info/stiga/ithf/index.aspx` (2012-03-01)

[4] Nye D. (2006) *Work: More, or less? Better, or worse?* I Technology Matters. ss. 121-131. Cambridge: The MIT Press.

[5] Etrendstore.se. `http://www.etrendstore.se/pub_images/original/stiga-bordshockey-PlayOff_2357.jpg` (2012-03-09)

[6] Fernandez L. et al. (2011) *Chalmers International Automated Table Hockey Championship* Göteborg: Chalmers tekniska högskola

[7] Arduino. *Arduino Board Mega 2560.* `http://arduino.cc` (2012-03-06)

[8] Arduino. `http://arduino.cc/en/uploads/Main/ArduinoMega2560_R3_Front.jpg` (2012-03-09)

[9] Bühler Motor. *DC Gear Motor 1.61.046.XXX.* `http://www.buehlermotor.de` (2012-03-06)

[10] TraceParts. `http://www.tracepartsonline.net/PartsDefs//Production/BUHLER/10-15022011-099929/pictures/10-15022011-099929L.gif` (2012-03-09)

[11] Pololu Robotics and Electronics. *Dual MC33887 Motor Driver Carrier.* `http://www.pololu.com` (2012-03-06)

[12] Pololu Robotics and Electroics. `http://b.pololu-files.com/picture/0J431.600.jpg?96b009962c6037e2aba0ad6cc491520f` (2012-03-09)

[13] Micro-Epsilon. *wireSENSOR WPS-MK30 analogue* `http://www.micro-epsilon.co.uk` (2012-03-06)

[14] Contelec. *WAL305 Potentiometer conductive plastic rotative* `http://www.contelec.ch` (2012-03-06)

[15] Elfa Distrelec. https://www1.elfa.se/data1/wwwroot/assets/large/241897f.jpg (2012-03-09)

[16] Pyramid Imaging. *DFK 21AUC03.* http://www.pyramidimaging.com (2012-03-06)

[17] The Imaging Source. http://s2.www.theimagingsource.com/img/pph/xl/dfk21auc03.png (2012-03-09)

[18] Mean Well. *SP-500 series* http://www.meanwell.com (2012-03-06)

[19] Arduino Playground. *Arduino and C++ (for Windows)* http://http://arduino.cc/playground/ (2012-05-16)

[20] Jeff Donahoo. *Practical C++ Sockets* http://cs.baylor.edu/~donahoo/practical/CSockets/practical/ (2012-05-16)

[21] IC Imaging Control. *IC Imaging Control Class Library User's Guide* http://www.meanwell.com (2012-05-16)

[22] OpenCV. *OpenCV* http://opencv.willowgarage.com/wiki/ (2012-05-16)

[23] Svenska Bordshockeyförbundet. *SBHFs regler för bordshockey.* http://www.bordshockey.net (2012-03-05)

[24] Department of Computer Science & Engineering, IIT Kharangpur (2008-04-30). *Lecture - 17 Introduction to Planning.* [YouTube] http://www.youtube.com (2012-05-16)

[25] Russel Stuart, Norvig Peter. *Artificial Intelligence A Modern Approach. 2nd ed.* Pearson Education, Inc. New Jersey. 2003. p. 445- 449.

[26] Department of Computer Science & Engineering, IIT Kharangpur (2008-04-30). *Lecture - 17 Partial Order Planning.* [YouTube] http://www.youtube.com (2012-05-16)

# A Uppstart

Nedan följer en guide för hur du startar upp det självspelande bordshockeyspelet.

1. Kontrollera att nätaggregaten är anslutna till nätspänningen.
2. Kontrollera att kameran är ansluten till serverdatorn (USB).
3. Kontrollera att de två mikrokontrollerna är anslutna till serverdatorn (USB).
4. Starta serverdatorn.
5. Slå på strömbrytaren till nätaggregaten.
6. Slå på strömbrytarna på strömförsörjningspanelen.
7. Starta programmet Hockey Manager genom Visual Studio. Projektet återfinns i mappen Hockey2012.
8. Kontrollera att Hockey Managers UI uppdateras genom att röra på någon spelare. (Om det inte händer något kan en omstart av arduinokorten ibland lösa problemet)
9. Starta två spelprogram genom Eclipse. Projektet återfinns i mappen Hockey2012. Se till att två olika portar anges vid start.
10. Placera pucken på spelplanen.
11. Skriv F (face off) i serverns terminalgränssnitt för att teka.
12. Efter mål eller om pucken skjutits av banan, placera pucken på spelplanen och skriv F i serverns terminalgränssnitt.

# B Ordlista

Ordlistan ger korta beskrivningar av tekniska begrepp och förkortningar som förekommer i rapporten.

**Bibliotek** Extern samling av moduler som ska kunna användas av andra utvecklare.

**Bildmask** En bild som används för att klippa ut delar av en annan bild. Pixlar i originalbilden som har färgade motsvarande pixlar i masken kommer med medan pixlar i originalbilden som har svarta pixlar i masken blir svarta i den resulterande bilden.

**Borstad elmotor** En typ av likströmsmotor där strömmen ska ledas in till motorns roterande axel. Detta görs genom att så kallade borstar ligger an som släpkontakter mot kontaktpunkter på axeln. Motortypen har fått sitt namn från just dessa borstar.

**CMOS** Förkortning av Complementary Metal Oxide Semiconductor. Det är en teknik för att tillverka integrerade kretsar. Tekniken förekommer bland annat för att tillverka ljuskänsliga bildsensorer för digitalkameror m.m.

**EEPROM** Förkortning av Electrically Erasable Programmable Read-Only Memory. Är en icke volatil minnestyp, dvs. data finns kvar även om strömmen bryts, som ofta används för att spara små mängder data som ska vara konstanta över en längre tid.

**Flashminne** En icke volatil minnestyp, dvs. data finns kvar även om strömmen bryts. Är egentligen en sorts EEPROM-minne, men benämns som en egen minnestyp då egenskaperna hos flashminnen skiljer sig lite i från de klassiska EEPROM-minnena. Flashminnen är idag mycket populära i bland annat mobiltelefoner och mp3-spelare.

**H-brygga** Elektronisk krets för att kunna leda ström på olika sätt igenom en last exempelvis en likströmsmotor. Anledningen till att kretsen kallas för en H-brygga är att man brukar rita upp kretsschemat som ett H där lasten sitter i centrum. Ut från lasten går då 4 “armar” där man sätter 4 strömbrytare och genom att styra strömmens väg genom lasten kan man styra den till på olika vis. Om det är en likströmsmotor kan man exempelvis få den att rotera medurs, moturs, bromsa och frikopplas.

**HSV-färgsystemet** Hue, saturation & value. På svenska: NMI. Nyans, mättnad och intensitet.

**Little endian** Byteordning för heltal där den lägsta byten kommer först och den största sist.

**Modul** Kod som utför en begränsad uppgift.

**PWM** Förkortning av Pulse-Width Modulation dvs. pulsbreddsmodulering. PWM är en teknik för att få fram en önskad spänning genom att slå på och av en spänningskälla snabbare än vad apparaten i andra änden kan urskilja och på så sätt erhålla den sökta spänningen som ett medelvärde av de skickade pulserna. Exempelvis kan man "lura" en motor att den får 3 V matningsspänning genom att pulsbreddsmodulera den med en 5 V spänningskälla.

**SRAM** Förkortning av Static Random-Access Memory. Är en volatilt minnestyp där det som lagras i minnet försvinner när strömmen bryts. Behöver till skillnad från dynamiska RAM-minnnen, DRAM ej kontinuerligt skrivas om för att spara sin information.