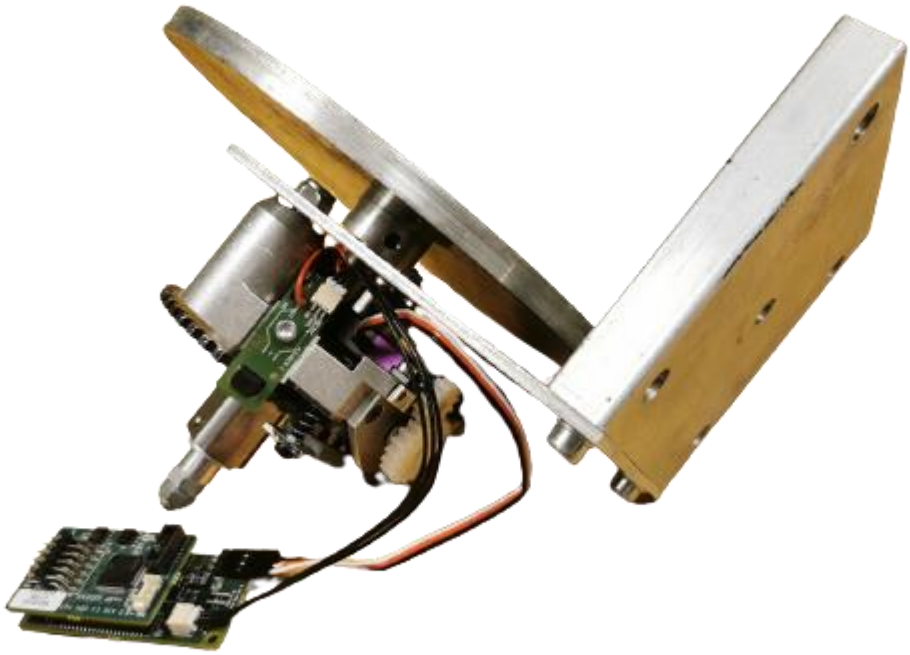
CHALMERS



Utveckling av servostyrd broms

Ett projekt baserat på Kvasers Devkit för framtida implementation i mikrobil

Examensarbete inom Data- och Informationsteknik

Anton Söfting
Sebastian Fonell

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2021

EXAMENSARBETE

Utveckling av servostyrd broms

Ett projekt baserat på Kvasers Devkit för framtida implementation i mikrobil

Anton Söfting
Sebastian Fonell

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg 2021

Utveckling av servostyrd broms
Ett projekt baserat på Kvasers Devkit för framtida implementation i mikrobil

Anton Söfting
Sebastian Fonell

Examinator:
Peter Lundin, Institutionen för Data- och informationsteknik

Handledare:
Sakib Sistek, Institutionen för Data- och informationsteknik
Lars-Berno Fredriksson, Kvaser, VD

Institutionen för Data- och Informationsteknik
Chalmers Tekniska Högskola / Göteborgs Universitet
412 96 Göteborg
Telefon: 031-772 1000

Omslagsbild: Den färdigbyggda bromsriggen med servo kopplad till ett mikrokort

Institutionen för Data- och Informationsteknik
Göteborg, Sverige 2021

Sammanfattning

Dagens bromssystem på bilar är baserad på hydraulik. För att komma bort från hydrauliken vill företaget Kvaser ersätta den hydrauliska delen med elektrisk styrd broms i form av servostyrning. Kvaser är specialister inom kommunikation enligt CAN-protokollet (Controller Area Network). Kommunikationen inom det aktuella systemet är baserad på detta protokoll. Syftet med detta projekt är att utveckla en servostyrd broms till Kvaser. Flera utvecklingsmiljöer har använts i skapandet av olika varianter av systemet så som Kvasers egna CANDev och det utomstående STM32CubeIDE som är skapat av STMicroelectronics. Rapporten fungerar som en snabbguide för att förstå sig på bromsen och hur det är uppbyggt inför vidareutveckling i framtiden. En testrigg konstruerades för att genomföra olika tester med den servostyrda bromsen. Resultaten visade att servot lyckades bromsa de vikter som testades samt den processorkonfigurering som användes för att styra servot. I framtiden förväntas det att dessa bromsar implementeras i Kvasers mikrobil för att användas synkront med regulatorer och ABS-styrning.

Nyckelord: Servo, CAN, CANDev, STM32CubeIDE

Abstract

Today’s brake systems in cars are based on hydraulics. The company Kvaser aims to move away from the hydraulics and replace it with an electrical brake system controlled with a servo. Kvaser are specialists in communication based on the CAN-protocol (Controller Area Network). The communication within the current system is also based on this protocol. The purpose of this project is to develop a servo driven brake system for Kvaser. Multiple development tools have been used in this project such as Kvasers own CANDev and STM32CubeIDE created by STMicroelectronics. This paper is intended to be used as a quick guide to get an understanding of how the brake is built and how to modify it for future development. A test rig was constructed to measure the braking capabilities of the servo. The results showed that the servo was strong enough to withstand the weights used during testing and the configuration of the processor unit used to control the servo.
Future plans for the car include an implementation of multiple synchronous servos with regulators and an anti-lock braking system.

Keywords: Servo, CAN, CANDev, STM32CubeIDE

Förord

Examensarbetet har utförts på Chalmers Tekniska Högskola på dataingenjörsprogrammet. Examensarbetet omfattar 15 hp och har genomförts som ett uppdrag av Kvaser under vårterminen 2021.

Vi skulle vilja tacka Lars-Berno Fredriksson, VD för Kvaser som välkomnade oss varmt, gav goda råd och vägledning samt Sakib Sistek som varit vår handledare på Chalmers och har varit till hjälp under hela projektets gång. Vi vill även tacka Hashem Hashem, David Kvist och André Idoffsson från Kvaser som alltid ställt upp för oss när vi behövt hjälp.

Ordlista

Prescaler	Nedskalning av klockfrekvenser, det betyder att en högre klockfrekvens justeras ner till en lägre frekvens.
PWM	Pulse Width Modulation, delar upp den konstanta spänningen i delar.
CAN	Controller Area Network, en databuss främst avsett för personbilar som låter flera styrenheter kommunicera med varandra på ett säkert sätt.
Duty cycle	Beskriver tiden i antal % då spänningen är på.
CPU-nodskort	Ett mikrodatorkort med kopplingar för ström och debugging.
Bärarkort	Ett mikrokort som har kopplingar för PWM. Detta mikrokort är kopplat med CPU-nodskortet i projektet.

Innehållsförteckning

1 Introduktion

I detta kapitel introduceras rapporten med lite bakgrundsfakta samt mål, syfte, avgränsningar och frågeställningar.

1.1 Bakgrund

Detta examensarbete utfördes hos Kvaser som är ett svenskt företag med huvudkontor i Mölndal men har även kontor i USA, Shanghai, China och Hong Kong. Kvaser specialiserar sig inom utveckling av CAN-baserade system (Controller Arena Network) där de tar fram både hårdvara och mjukvara som ger lösningar till fordon, militär, järnväg, medicin och så vidare [1].

Kvaser har på senare år tagit fram ett utvecklingskit, Kvaser DevKit. Kittet består av en bil i skala ⅕ och bär ett CAN system. Samtidigt är bilen tillräckligt liten för att kunna köras inomhus. Dagens bromssystem på bilar är baserad på hydraulik. För att komma bort från hydrauliken vill Kvaser ersätta den hydrauliska delen med elektrisk styrd broms i form av servostyrning.

Denna rapport kommer beskriva utvecklingen av mjukvaran och konstruktionen av en servostyrd broms till bilen som uppdrag från Kvaser. För utvecklingen av bromsen har Kvaser tagit fram en testrigg för bromsen som används som testbänk vilket vi huvudsakligen kommer att arbeta med under projektets gång. Kvaser har bestämt att bromsen ska styras med ett servo som ska monteras på testriggen.

Utvecklingsmiljöerna som kommer användas under projektets gång är mestadels Kvasers egna: CANDev men också STM32CubeIDE som inte ägs av Kvaser men vi kommer att prata mer om dessa senare i rapporten.

1.2 Syfte

Syftet med arbetet är att experimentera med en servostyrd broms som i ett senare skede ska kunna användas som en referens för att tillsammans med rapporten förstå sig på utvecklingsmiljöerna, systemet och hur det är uppbyggt inför vidareutvecklingen av ett ABS-system. Syftet är också att sätta ihop en bromsrigg där den servostyrda bromsen kommer sitta för de olika testerna.

1.3 Mål

Målet med denna rapport är att utveckla mjukvara för styrning av den servostyrda bromsen som medger genomförande av olika bromstester. Mer specifikt ska vi utföra följande.

- Reglera servot med hjälp av PWM-signaler.
- Konstruera en bromsrigg för att styra bromsen i en kontrollerad miljö.
- Utveckla mjukvaran till servostyrningen med stöd av utvecklingsmiljön CANDev.
- Utveckla mjukvaran till servostyrningen med stöd av utvecklingsmiljön STM32CubeIDE.
- Få en bromsande effekt på bromsskivan med hjälp av servot.

1.4 Avgränsningar

Flera avgränsningar var tvungna att göras för att projektet skulle kunna utföras inom den angivna tiden.

- Det bestämdes strax efter start att arbetet enbart skulle omfatta testriggen, därför kommer det varken att arbetas med bilen eller synkronisering av fler bromsar.
- Arbetet kommer inte att kommunicera via CAN mellan olika noder då endast ett servo används.
- Arbetet kommer inte att gå in på utvecklingen av ett ABS-system.

1.5 Frågeställningar

Är Kvasers mjukvara CANDev lättare att använda än STM32CubeIDE för att utveckla mjukvaran till styning av bromsserevot.
Är servot tillräckligt starkt för att fungera i en broms?

2 Teknisk Bakgrund

I detta kapitel beskrivs de tekniska delarna samt lite kort info om de delar man behöver ha koll på för att hänga med i rapporten.

2.1 CAN

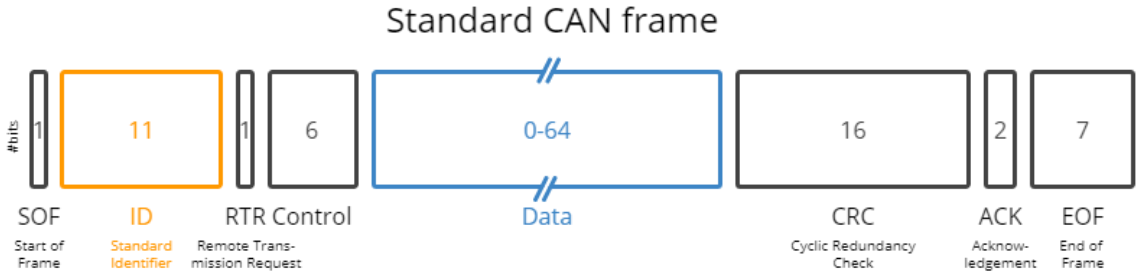
CAN står för Controller Area Network, det är en databuss som används som standard inom fordon. Den låter mikrokort och styrenheter att kommunicera med varandra över databussen. Enheterna som kommunicerar med varandra kallas för noder. En modern bil kan ha upp till 70 noder där alla noder kan kommunicera information till varandra. Anledningen till att CAN används i bilar är därför att det är billigt och simpelt men samtidigt också väldigt robust, stresståligt och effektivt.

Skillnaden mellan CAN och till exempel Ethernet är att CAN är en addressfri, masterfri, flernodig buss med kollisionshantering i realtid medan Ethernet är punkt till punkt-kopplad. Ethernet är tillskillnad från CAN också bättre anpassad för att skicka större datamängder över längre sträckor. CAN används i stället i distribuerat inbyggda system med krav på hög pålitlighet.

2.1.1 Kommunikation

Noderna är kopplade med ett och samma tvinnade kabelpar. Alla noder kan kommunicera med varenda nod som är kopplad på bussen. Man skickar alltså inte data direkt till en specifik nod, alla noder kommer lyssna på all data som skickas. Därefter kan hårdvaran hos noderna filtrera datan för att reagera på relevant meddelande.
Ett exempel är en sensornod som mäter vattentemperatur. Noden själv behöver inte veta varför den mäter temperaturen, den vet bara att den ska skicka en temperatur med ett specifikt ID som systemkonstruktören har bestämt.
Det bör då finnas en annan nod som lyssnar efter vattentemperaturen från systemet. Meddelandet för vattentemperaturen kan exempelvis ha ID 17, då bör sensornoden skicka meddelandet med ID 17 och mottagarnoden bör lyssna efter samma ID för att ta emot meddelandet med vattentemperaturen. Om mottagarnoden hade lyssnat efter ett annat ID hade ID 17 ignorerats.
Detta ID-nummer används också för att lösa konflikter vid kollisioner av meddelanden. Det meddelande med lägst ID kommer att skickas medan det andra får vänta i händelse av samtidig sändning från flera noder. Av denna anledning får de viktigaste funktionerna hos bilen lägst nummer, till exempel bör bromsen ha ett lägre ID än radion.

2.1.2 Meddelanden



Figur 1: Formatet på ett CAN-meddelande

Det finns fyra olika typer av meddelanden på CAN bussen.

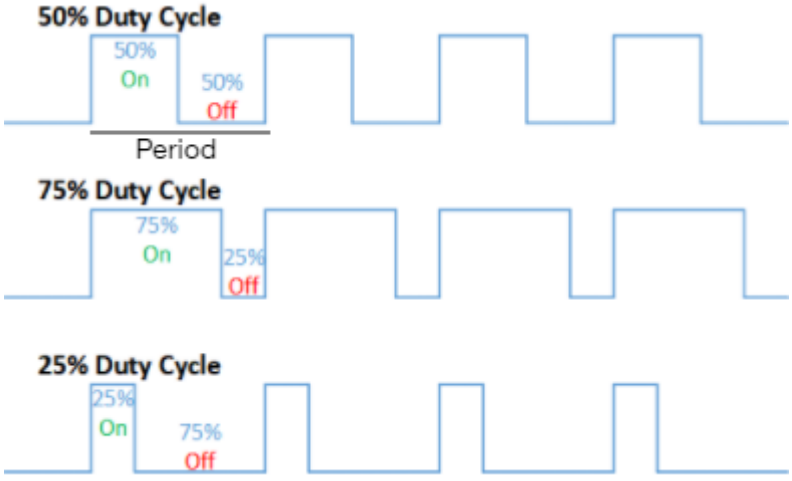
Data frame är den vanligaste typen och innehåller åtta fält(fields).
Arbitration field i meddelandet beskriver prioriteten på meddelandet och används när två noder skickar samtidigt. Data field innehåller upp till åtta bytes med data, här paketeras informationen i meddelandet. CRC Field används för att detektera fel i meddelandet och står för cyclic redundancy check. En acknowledgement-bit används för att meddela sändaren att meddelandet tagits emot korrekt, om inget acknowledgement returneras skickas meddelandet igen.
Remote frame liknar data frame men den saknar ett datafält. Dess syfte är i stället att be om data från en annan nod, till exempel en sensor. I praktiken används den sällan.
Error frame skickas när en nod upptäcker ett fel och får resterande noder att detektera ett fel. Noderna kommer sedan i sin tur också skicka error frames.
En räknare ser till att noderna inte förstör bussen genom att skicka för många error frames.
Overload frame liknar error frame i strukturen. Den används när en nod är för upptagen för att ta emot data. Dagens CAN controllers är smarta nog att inte behöva använda denna frame. [2]

2.2 Pulse Width Modulation

Pulse Width Modulation (PWM) används i vårat fall för att styra servomotorer men kan också användas till att kontrollera hastigheten på en fläkt eller ljusstyrkan i en LED-lampa som två exempel av många. I kontrast till en A/D-omvandlare som gör en analog signal till en digital används PWM för att omvandla digitala signaler till analoga. Mer exakt så är PWM signaler en fyrkantsvåg som periodiskt växlar mellan två bestämda spänningsnivåer.
Tiden då signalen är hög kallas för “on time” respektive “off time” då den är låg. [3]

2.2.1 Duty Cycle

Duty Cycle bestämmer hur länge signalen är påslagen under en period. En duty cycle på 50% kommer alltså generera lika långa höga signaler som låga. Vid 25% kommer signalen vara hög en fjärdedel av periodtiden. [3] Olika duty cycles motsvarar olika positioner hos servot i vårat projekt.



Figur 2: Exempel på olika duty cycles med höga och låga signaler

2.3 Hårdvara

Här förklarar vi vilken hårdvara som används samt hur vi använder den.

2.3.1 Processor och mikrokort

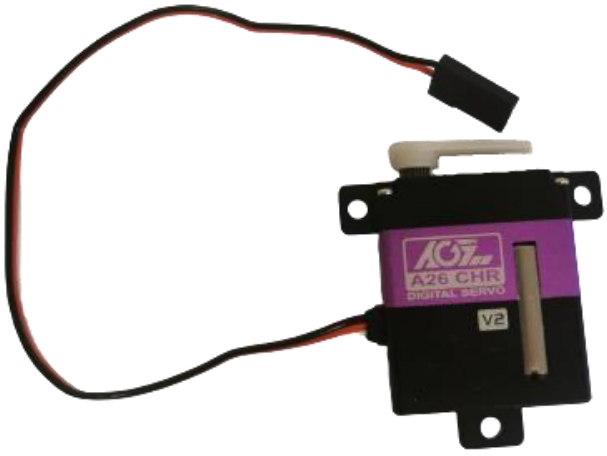
Processorn som vi fick tilldelade oss bygger på en 32-bitars ARM® Cortex®-M4 STM32F3 MCU med fullständiga namnet STM32F302RET6. För mer information se referens [4]

Processorn sitter på ett CPU-nodskort som i sig sitter på ett bärarkort. På bärarkortet finns utgångar för PWM-signaler och på CPU-nodskortet finns anslutningar för CAN-kommunikation. Ytterligare anslutningar finns på korten men det nämnda var endast det vi använde.

Figur 3: CPU-nodskortet som sitter på bärarkortet

2.3.2 Servo

Servot som används är mikroservot ‘AGFRC A26CHR Full Aluminum Case 11KG Coreless Digital Wing Servo’. Servot är ett digitalt servo med en uppdateringsfrekvens på 333Hz och en driftspänning mellan 4.8-8.4V som ger ett stallmoment från 7.5-11kg-cm. Servots vridvinkel är 120° och med hjälp av duty cycles går det att rotera servot till den vinkel som önskas. Detta servo tål mycket högre spänning än de tidigare servona som använts vilket är positivt då de tidigare har brunnit upp vid relativt låga spänningar. [5]



Figur 4: Servo med kabel och arm

2.3.3 T-Cannector

T-Cannector är en CAN-bus hub med totalt fyra D-SUB9 kopplingar där det är två hon- och han-kopplingar, denna användes för att möjliggöra ett skapande av en lokal CAN-buss. [5]

Figur 5: T-cannector med fyra D-SUB9 kopplingar

2.3.4 USBcan Pro

USBcan Pro är en USB till ett dual-channel CAN gränssnitt med D-SUB9 som koppling. Denna användes i projektet för att sätta upp en CAN-buss mellan bärarkortet och datorn för att kunna programmera CPU-nodskortet och skicka PWM-styrsignaler till servot. [6]

Figur 6: USBcan Pro med USB-kontakt och två D-SUB9 kontakter

2.3.5 ST-LINK/V2

Vi använde en ST-LINK/V2 för att ladda ned en ny bootloader till CPU-kortet med namnet CANDev Bootloader som stödjer nedladdning av mjukvara över CAN-bussen. Vi använde även ST-LINKen som koppling mellan dator och processorn för att konfigurera processorn och skriva till portarna i CubeIDE som nämns senare. [7]

Figur 7: ST-LINK/V2 med USB-kontakt och koppling till CPU-nodskortet

2.3.6 Debuggsladd

Vid debugging av vår kod användes en debuggsladd med koppling från datorn till bärarkortet och programmet Tera Term. Den behövdes för att se olika värden och meddelanden från kortet i en Tera Term-terminal. Vi använde Tera Term terminalen i projektet för att se utskriften av vårat program när det kördes.

Figur 8: Debuggsladd för Tera Term med USB-kontakt

2.4 Mjukvara
Här beskrivs den mjukvara som användes samt på vilket sätt den användes.

2.4.1 Kvaser CANDev
CANDev är ett ramverk för design och hantering av CAN system. Den del vi använde oss av var ‘Workbench CANDev’ som är ett Eclipse IDE baserat program inom CANDev och koden skrivs i programspråket C. CANDev har även stöd för realtidshantering med hjälp av rt-collab toolbox från rt-labs.

2.4.2 Tera Term
Tera Term används i vårat fall vid debugging av CPU:n och är ett terminalemuleringsprogram som stödjer SSH 1 & 2 och seriella portanslutningar. Resulten vi får ut från att debugga med Tera Term skrivs ut i Tera Terms terminalfönster som sedan får tolkas för att avgöra var problemen finns. [8]. Se **bilaga 6** för ett exempel på hur terminalen kan se ut när ett program körs.

2.4.3 STM32CubeIDE
STM32CubeIDE, kommer fortsättningsvis kallas för CubeIDE i rapporten. CubeIDE är en utvecklingsmiljö för mikroprocessorer och mikrokontroller som hjälper till med kodgenerering i språket C, debugmöjligheter, kompilering och processorkonfigurering. Det är baserat på ramverket Eclipse och verktygskedjan GCC för utveckling och GDB för debugging. [9]

3 Metod

I detta kapitel redovisas de metoder som använts för att samla information om projektet samt vilka val som gjordes.

Det första som behövdes göra var att förstå problemet. Det fanns tillgång till tidigare rapporter inom företaget på försök till utveckling av bromsen som kunde användas som referens och få en djupare inblick i problemet. Testriggen för bromsen var inte helt färdig än så inga tester kunde genomföras tills den var färdig. Det dök snabbt upp problem då verktyg och hårdvara inte fanns på plats. Mycket tid spenderades då åt att planera och tänka över problemet.

När delarna till bromsriggen fanns på arbetsbordet fick logiskt tänkande användas vid ihopsättning av de olika komponenterna då det inte fanns någon manual. Till hjälp under projektets gång hade vi kontakt med kollegor på företaget om eventuella frågor angående hård- eller mjukvara dök upp. Eftersom de hade bättre kunskaper om mekatronik och elektronik var de väldigt hjälpsamma vid det mer praktiska arbetet.

Vid ett möte redovisade en kollega Kvasers utvecklingsmiljö CANDev. Detta program skulle användas för att kommunicera med servot. Programmeringsspråket som användes var C.
Därefter började experimenteringen med servot och dess egenskaper. Mycket av den informationen som användes hittades på olika sidor och videos på nätet. Enligt handledningen fanns det också tillgång till ett spänningsaggregat för att testa olika spänningar till servot.

Vid programmeringen av processorn kom kunskaperna i Maskinorienterad programmering till användning då det gällde pin-konfigurering samt att signaler skulle skickas på rätt port. För att få fram databladen för processorn kontaktades ett externt företag då de datablad Kvaser hade var felaktiga. Samma företag rekommenderade att använda CubeIDE i stället för CANDev för att konfigurera processorn. Anledningen till detta var för att det finns mycket information om CubeIDE på internet om hur programmet fungerar medans informationen om CANDev inte är lika lättillgänglig. CubeIDE kommer dessutom med en get-started guide vid installation som hjälpte med att förstå grunderna.

För att till slut genomföra testerna på bromsen användes ett drop-test där det släpptes olika vikter för att se hur långt de hann falla innan de kom till ett stop.

4 Genomförande

I denna del beskrivs djupare hur de olika delarna i projektet genomfördes.

4.1 CANDev

Det fanns ingen information om hur det skulle kommuniceras med servot gällande mjuk- och hårdvara, därför redovisade en anställd på Kvaser deras utvecklingsmiljö CANDev. Detta skulle användas som utvecklingsmiljö vid utvecklingen av styrprogram till bromssystemets servo.

Hårdvaran på Kvaser introducerades även. T-Cannector, USBCan Pro, ‘Debug sladd’ och en utomstående komponent ST-LINK/V2. Nästa steg var att bekanta sig med all hårdvara, förstå hur hårdvaran skulle kopplas samman, användas och slutligen sätta upp en CAN-buss. Det började med att skapa en lokal CAN-buss med hjälp av CANDev där det endast skickades meddelanden över CAN-bussen för att se att en koppling hade blivit skapad. Detta gjordes för att se att CAN-bussen hade satts upp på rätt sätt.

Därefter började arbetet med CANDev och programmeringen i samma utvecklingsmiljö. Detta gjordes för att det är enkelt i CANDev att starta om bootloadern och skicka program till servo-kortet. Vid nedladdning av CANDev till datorn för programutveckling så följer några exempelprogram med samt flera .h och .c filer som innehåller färdiga funktioner som används i kombination med Kvasers hårdvara. Efter många timmars letande och tolkande av kod i diverse filer hittades funktionerna för att kontrollera PWM-signalerna varefter koden för styrning av en enkel “handbroms” kunde skrivas. Se bilaga 3 för koden.

4.2 Servo

Vid tidigare försök av detta projekt har ett äldre servo använts som inte klarat den spänning som behövs för att kunna bromsa i högre hastigheter. Nu används ett nyare servo som klarar en högre spänning med maxvärden på 8.4V och en ström på 2A vilket ger ett högre vridmoment från servot. Det valdes att inte överstiga någon av gränserna för att minimera risken att bränna servot. Det nya servot har dessutom en högre frekvens, 333Hz istället för 50Hz vilket innebar att en del av värderna för styrningen av PWM-signalen behövde räknas om då de tidigare värderna avsåg styrfrekvensen 50Hz till servot. Något positivt för framtiden med ett servo med högre frekvens är att upplösningen blir större vilket passar bättre när programmering av ABS kommer in i bilden då en högre frekvens medger en snabbare uppdatering av servots läge.

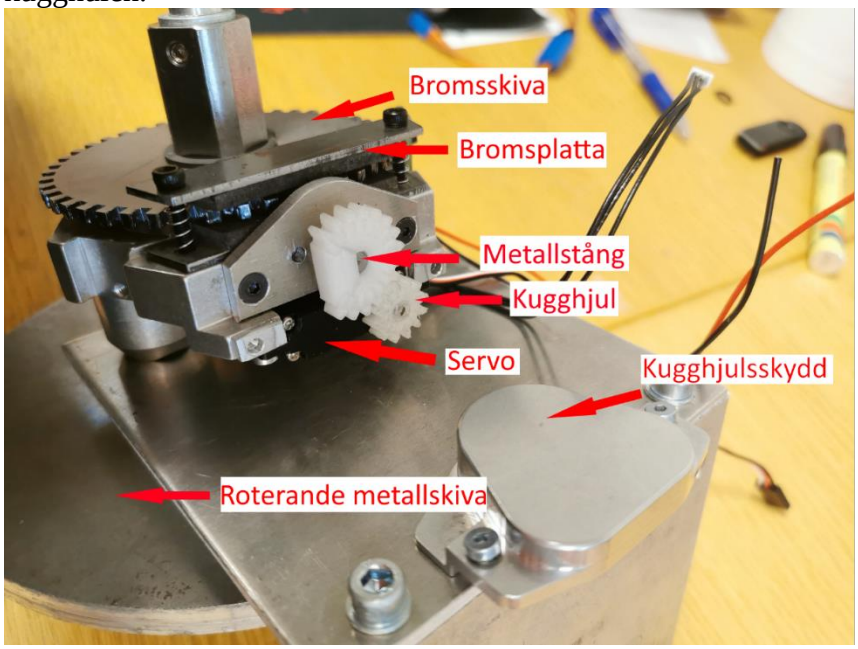
Därefter började experimenteringen med själva servot. Genom att koppla servot till mikrodatorkortet kunde det styras med PWM-signaler. När servot väl kunde styras räknades duty min och duty max ut eftersom de inte såg ut att stämma överens med talen i specifikationen. Med duty min och max menas de positioner som servot maximalt kan uppnå.

Duty min och max uttrycks i ett tal vardera som styr pulsens längd av PWM-perioden. Dessa tal är en procent av duty cyclen och sätts som en variabel i CanDEV, det vill säga att till exempel 4000 motsvarar 40% vilket betyder att signalen som vi skickar är hög 40% av perioden. Den duty cycle som skickas motsvarar också servots position.
Efter en tids testande räknades det ut till att de låg på ca 3500 och 6500. Därefter togs det reda på vilka duty cycles som behövdes för att klämma respektive släppa på bromsen då det är fler mekanismer som är i rörelse än bara servot.

4.3 Bygge av broms

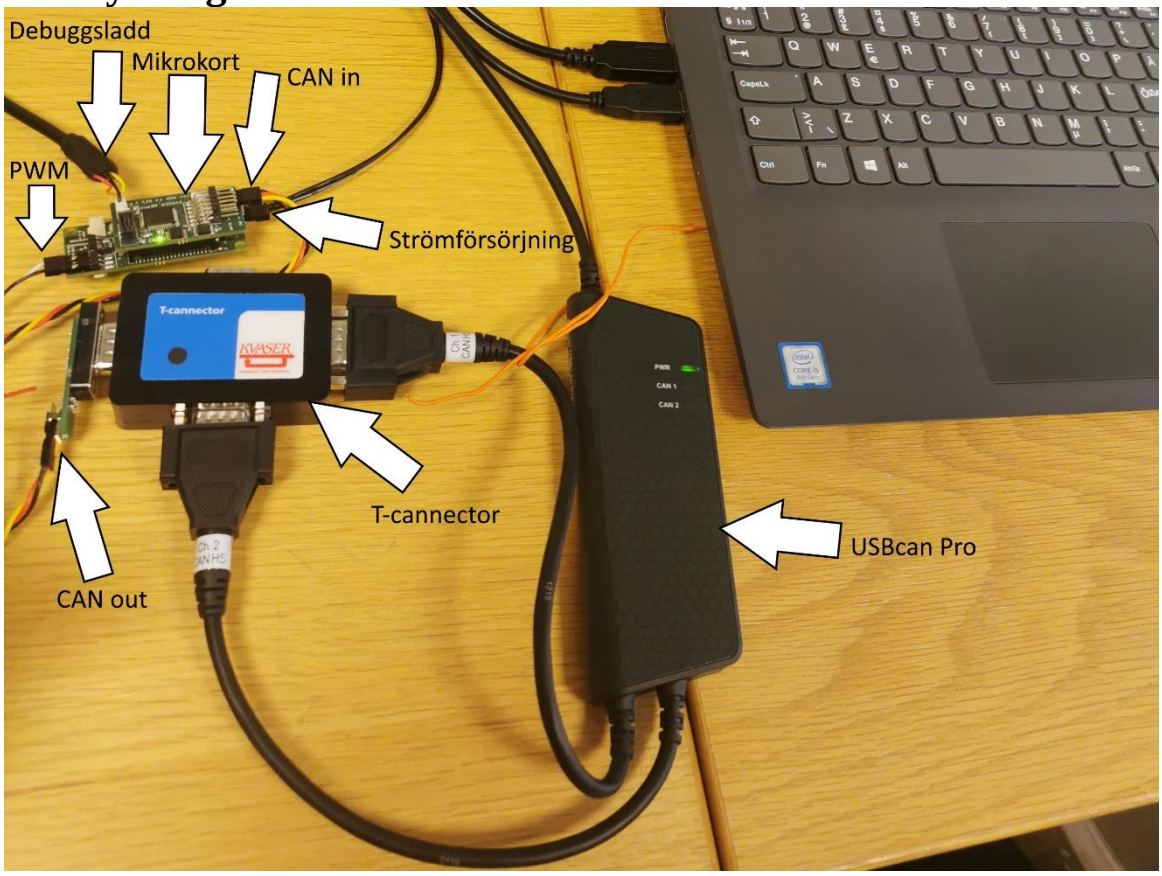
Bromsriggen var inte komplett vid projektets start, det saknades bromsmekanism, kugghjul och fel servo satt i. Den första uppgiften blev då att montera det nya servot på testriggen. Då kugghjulen till bromsen 3D-printades på Kvaser och behövde vara väldigt tåliga för att inte gå sönder under bromsning så tog det ett par dagar för de att bli klara samt att ett extra par printades för säkerhets skull. Efter ett par veckor hade alla komponenter till testriggen levererats så att de kunde monteras med servot.

Figur 9 visar undersidan på den kompletta testriggen. När servot snurrar så kommer de två kugghjulen snurra som i sin tur kommer vrida på en liten metallstång som går att se i mitten av det större kugghjulet. När stången vrider på sig kommer den dra in de svarta skruvarna som i sin tur drar in den yttre bromsplattan och då finns det en bromsande kraft på bromsskivan. Denna bromsade kraft kommer få den stora roterande metallskivan som simulerar ett hjul att bromsas och vid tillräcklig kraft att stannas. Servot styrs med PWM-signaler från mikrokortet. Kugghjulsskyddet som på figur 9 är avmonterat kommer monteras inför testningen av bromsen för att smuts eller skräp inte ska komma emellan kugghulen.



Figur 9: Undersidan av testriggen där skyddet till kuggarna ligger avmonterad vid sidan.

4.4 Styrning av servo med elektronik



Figur 10: Systembeskrivning

Mjukvaran utvecklas med stöd av utvecklingsmiljön CANdev och laddas ned till CPU:n med stöd av CANdev Firmware Downloader. Den inbyggda bootloadern som finns på mikrodatorkortet kollar efter dessa meddelanden vid start och om en nedladdningsförfrågan finns så kommer den mottas och laddas ned innan mikrodatorkortet startar om med den nya mjukvaran.

Gången från datorn till mikrodatorkortet är följande: Koden skickas via USB till en USBcan Pro som är kopplad till en T-Cannector där en lokal CAN-buss automatiskt skapas. Därefter skickas koden via CAN från T-Cannectorn till CPU-nodskortet som i sin tur kör koden och meddelar bärarkortet att skicka PWM-signalerna till servot.

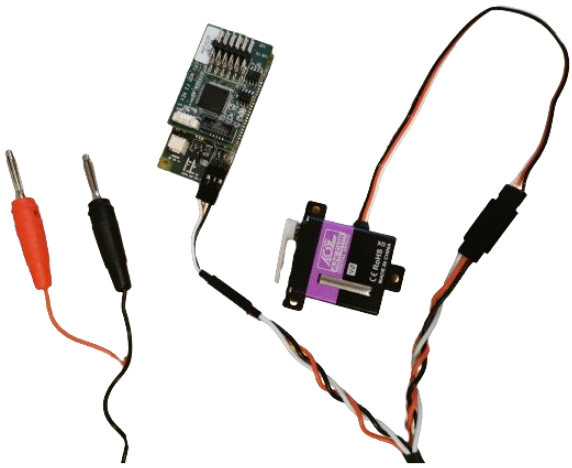
Vid experimentering kopplades borstmotorn på servot till ett spänningsaggregat för att mata mer ström och spänning än vad bärarkortet kunde göra. Detta gjordes med hjälp av banankontakter, spänningsaggregat, förlängningssladd och krokodilklämmor.
I detta stadie experimenterades det endast på servot och det fanns då ingen koppling till bärarkortet. Eftersom detta endast var ett snabbt test så använde vi krokodilklämmorna för att klämma fast sladdarna till motorn, det är inte hållbart i längden men fungerar vid ett snabbt test.

Figur 11: spänningskablar ihopkopplade med servots motor med hjälp av krokodilklämmor

För att få en bättre förståelse för hur servot reagerade på olika spänningar ökades spänningen successivt från 2V upp till 8.4V. Då flyttade sig servot snabbare och snabbare och med starkare moment vilket gav en större bromskraft. När servot tidigare styrdes från kortet låg spänningen enbart på 5V som ej gick att justera.

Vid detta stadie noterades att servoarmen roterade fritt i stället för att hålla sig till sin specificerade vridvinkel på 120 grader. Detta var på grund av att när vi kopplade direkt till motorn så användes inte PWM-signaler för att flytta armen. I stället användes spänningen från spänningsaggregatet för att styra armen direkt via motorn.
För att förhindra att någon krets skulle brännas så användes Over voltage protection(OVP) och Over current protection(OCP). Kortfattat så betyder det att ström och spänningen inte kan gå över den satta gränsen. OVP och OCP sattes enligt specifikationen för servots arbetsspänning och ström vilket slutade på 8.1V och 2.5A.

I nästa stadie var det dags att koppla in bärarkortet för att kunna ansluta servot till en extern spänningskälla samtidigt som man kan styra servot via PWN-signaler.
Tillvägagångssättet för detta blev då följande: Kontakten till servot är en hona så vi tog bort honkontakten på förlängningssladden och skarvar ‘plus’ och ‘minus’ sladdarna för att sätta på banankontakter för att sedan kunna koppla in i spänningsaggregatet.



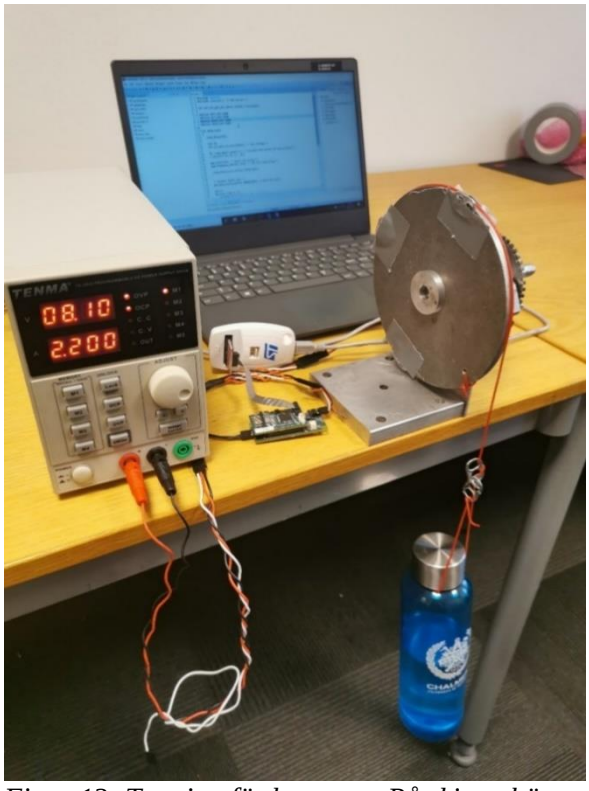
Figur 12: Spänningskontakter hopskarvade med PWM-sladd och jord från mikrokort

Vid första testkörningen skickades PWM-signaler för att flytta servot till en viss punkt men vid start så ‘hackade’ servot fram och tillbaka riktigt snabbt mellan två punkter. Anledningen till detta var att uppkopplingen inte hade en gemensam jord för spänningskällan och servot vilket betyder att det inte fanns en referens för den digitala konverteringen i bärarkortet när spänningskällan var inkopplad. När detta fel var justerat så förflyttade servot sig till den tänkta punkten. Skillnaden från tidigare är att en extern spänningskälla är inkopplad så momentet på servot kan nu justeras.

4.5 Testning

Innan testningen kunde börja så behövdes ett sätt att kunna säkra vikten till metallplattan som simulerar hjulet. Detta gjordes genom att tvinna ett par sladdar som sedan knöts fast i hjulet. Därefter behövdes en sorts krok för att säkra vikten som i detta fall är en vattenflaska och detta gjordes genom att ta ett par delar från en aluminiumburk och hakade fast de i varandra så en krok bildades där vikten sedan hängde.

Det första testet som genomfördes var att se hur väl servots momentkraft tålde motvikter. Det gjordes genom att hänga en vikt i form av en vattenflaska på den klämda bromsskivan vid en viss spänning för att sedan öka vikten. Vikten på flaskan ökades genom att kontinuerligt fylla på den med vatten tills bromsen inte kunde hålla emot längre. Därefter vägdes flaskan för att notera brytpunkten. Vid det här testet användes CANDev som IDE för utvecklingen av programmet som styr servot samt ett spänningsaggregat för test av olika drivspänningar till servot.

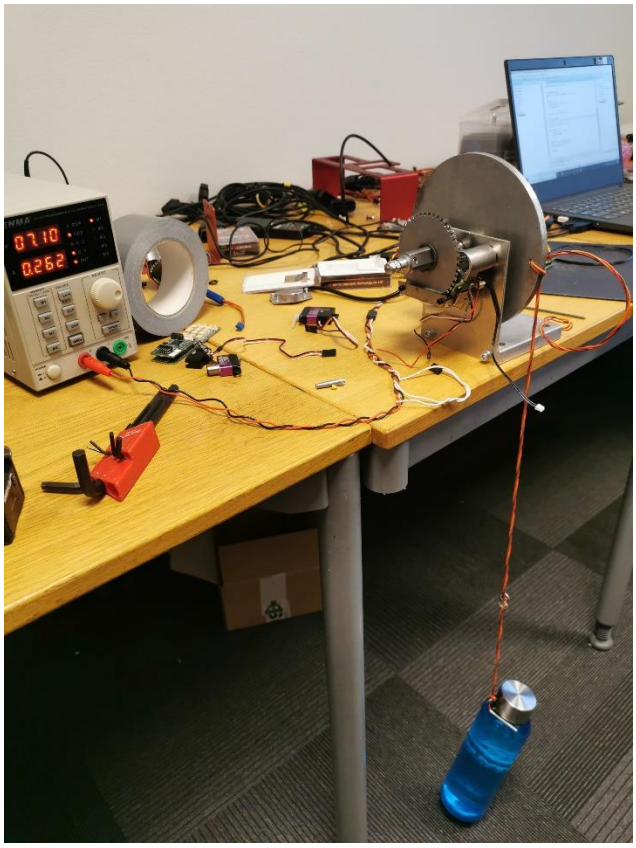


Figur 13: Testrigg för bromsen. På skivan hänger en vikt i form av en vattenflaska. Ett spänningsaggregat är kopplat till servot på bromsen och programmet körs från datorn.

Detta gjordes ett flertal gånger för att se att resultatet inte var en engångsföreteelse. För resultatet se tabell 1 i kapitel 5 Resultat.

Vid det andra testet var det bromssträckan som var i fokus. Ett kort program skrevs i CANDev som skulle låta servot släppa bromsen i 200ms för att sedan klämma åt igen för att mäta bromssträckan på skivan. Testet gick till enligt följande: Ett snöre snurrades runt den stora skivan så att vattenflaskan kom i toppläget och kunde falla utan att snöret fastnade. Sedan startades programmet och servot drog åt bromsen så att vikten stod helt stilla. Efter en kort stund släppte servot i 200ms för att sedan dra åt igen. När programmet hade körts så mättes den totala sträckan som vattenflaskan hade fallit. För resultatet se tabell 2 i kapitel 5 Resultat.

Bilden nedan visar resultatet av ett bromstest då spänningen ligger på 7.1V och flaskan väger 0.75kg. Efter körning av programmet stannade flaskan precis innan den slog i marken.

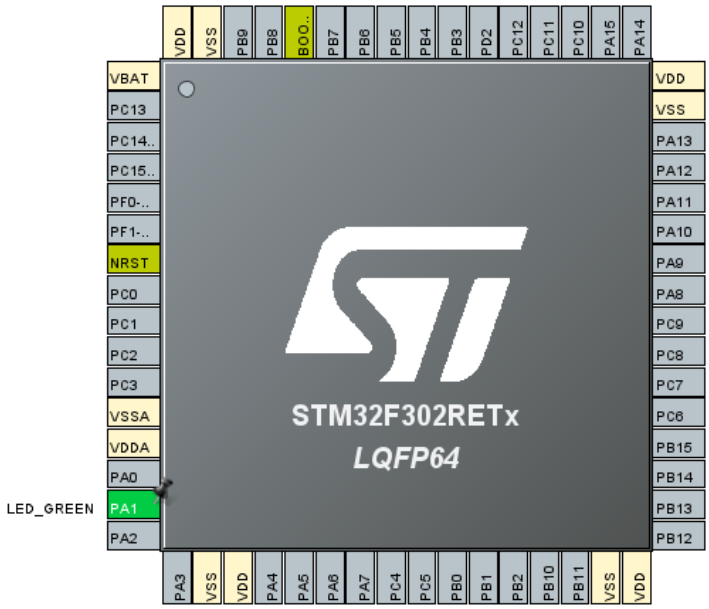


Figur 14: Testrigg för bromsen. På skivan hänger en vikt i form av en vattenflaska. Ett spänningsaggregat är kopplat till servot på bromsen och programmet körs från datorn.

4.6 STM32CubeIDE

STM32CubeIDE är en utvecklingsmiljö som skulle användas i projektet för att konfigurera processorn då det är mycket enklare att göra i CubeIDE än i CANDev. Processorn är en STM32F302RET6 och konfigureras med hjälp av CubeIDE, kopplingen mellan datorn och processorn gjordes med hjälp av en ST-LINK/V2.

För att förstå utvecklingsmiljön så skapades först ett enkelt program som gjorde att debug-lampan på bärarkortet blinkade. Detta krävde mycket mer tid än planerat då databladen från Kvaser var utdaterade och stämde inte överens med den nuvarande versionen av processorn så kopplingarna på pinnarna stämde inte. När tolkandet av databladet var gjort så återstod det att konfigurera PA1 porten som gjordes i main filen i CubeIDE. Detta genomfördes som ett test för att se att programmeringen av processorn funkade som den skulle.

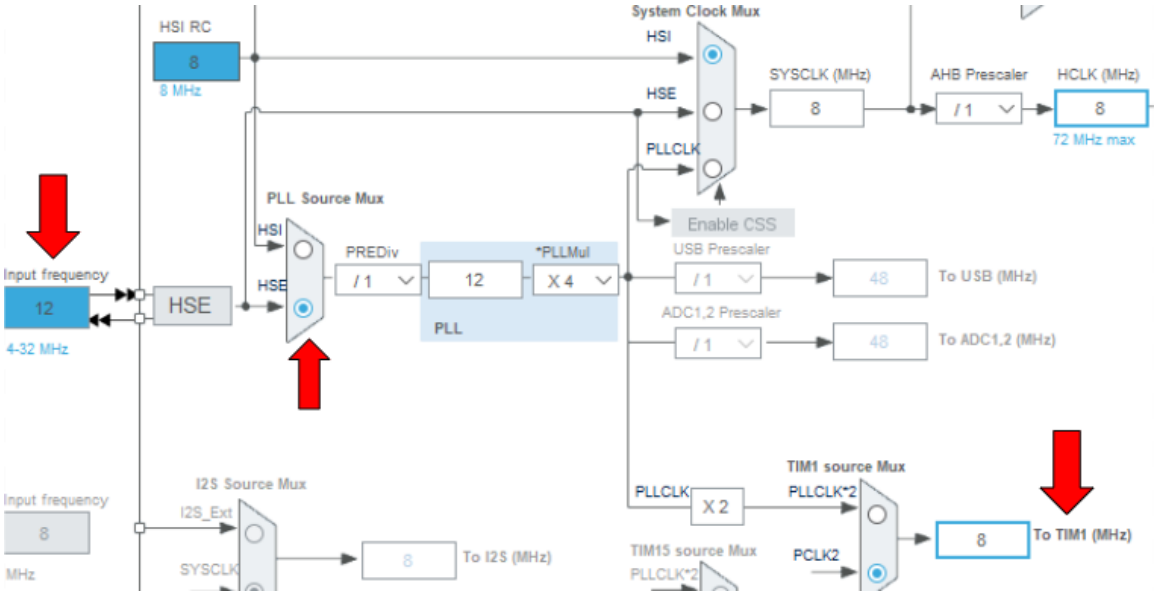


Figur 15: En överblick av konfigureringen av processorns pinnar. Den markerade gröna pinnen är kopplad till en LED lampa.

Efter detta var det dags att återgå till databladet för att ta reda på hur PWM-signalerna skulle skickas för att kunna justera momentet mer än enbart maximal och minimal bromskraft.

Servot som används har en uppdateringsfrekvens på 333Hz medan CodeIDE är förinställt för ett servo med 50Hz så då behövdes en omkonfigurering av prescalern göras för kretsen för att få den korrekta uppdateringsfrekvensen.

Som bilden nedan visar så är Input frequency satt till 12MHz och en High-Speed External clock(HSE) som används med crystal resonator. Crystal används för att det är den enda som tillåter att frekvensen är 12MHz i CodeIDE [10]. Input frequency är nödvändigt då 'TIM1' klockan ska ha en frekvens på 8MHz för att underlätta uträkningen av perioden.



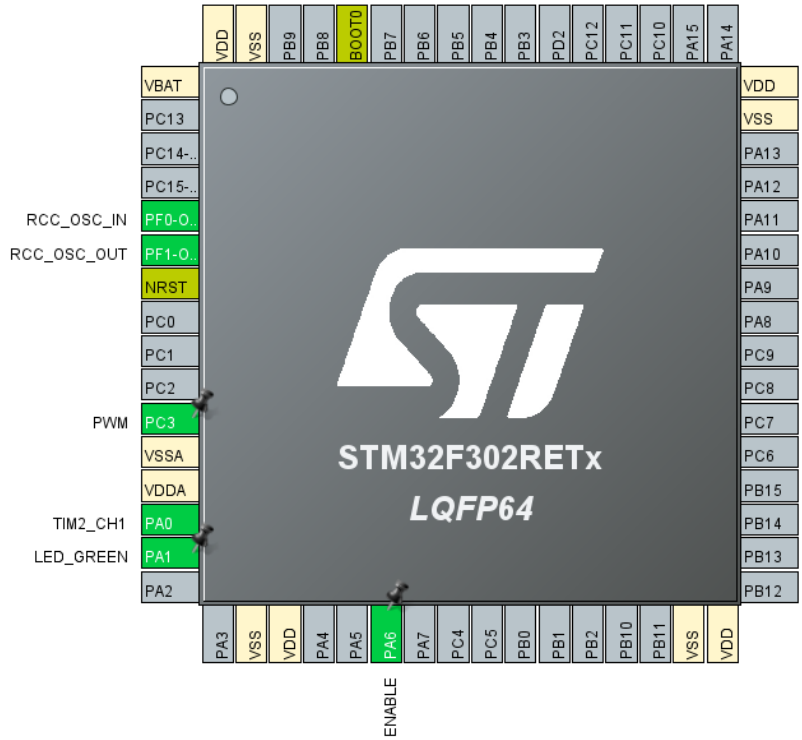
Figur 16: Klockkonfigurationen för processorn i CubeIDE. Pilarna pekar på Input frequency, High-Speed External clock valet och Timer 1 (TIM1).

Servots period låg på 3ms, därför söks en likadan period på pulsen i systemet för att uträkningarna skulle vara smidiga. Klockan dividerades för att få en högre systemperiod. En högre systemperiod gav en högre upplösning på servot. Därför delades klockan på fyra genom att sätta prescale till tre och vi fick en ny klocka på 2MHz. För att få en periodtid på 3ms för styrning av PWM-signalen behövde systemperioden ligga på 6000 eftersom periodtiden är kvoten av systemperioden dividerat med klockans frekvens. För aktuell bild på processorn se figur 17 som finns i kapitel 4.7.

Därefter granskades servots specifikationer och min/max för armens positioner lästes av. De låg på 0.9ms respektive 2.1ms, vilket är PWM-pulsens längd. För att uppnå dessa pulstider laddar man ett register med talen 1800 respektive 4200. Det neutrala läget låg på 3000. Med det gjort hade de pulser som skulle användas för att styra servot hittats och de laddades i motsvarande register hos processorn.

4.7 Reglera servot i STM31CubeIDE

Här är funktionen ‘PWM generation output’ kallat “PWM” på bilden kopplat enligt databladet i bilaga 1 till PC3 som i sin tur är kopplat till mikroservot. Denna koppling med processorn kan nu användas med en spak eller pedal för att få liknande bromskänsla som i en bil men koden testades aldrig i praktiken. I bilaga 4 är kod för en mjukare inbromsning där det går från min broms -> max broms -> min broms. Koden i bilaga 4 är inte en del av “handbromsen”, denna koden är till för att testa en mjukare inbromsning snarare än en maxbromsning. Figur 17 visar konfigurationen av processorn som används.



Figur 17: Processorn i CubeIDE med några konfigurerade portar och tillhörande namn

5 Resultat

I detta kapitel redovisas resultaten gällande servot och utvecklingen.

5.2 Bromstest

Så här såg resultatet ut från det första testet av den statiska bromsen. Kolumnerna visar den spänning vi matade, brytpunkten där bromsen började släppa samt vikten uttryckt i newtonmeter.

Spänning(V)	Brytpunkt(kg)	Newtonmeter(Nm)
2.5	2.1	1.64
4.8	2.3	1.8
6	2.6	2.04
7	2.8	2.2
8.1	3	2.35
8.4	3	2.35

Tabell 1: Resultat från statiskt bromstest

Resultatet vid test av bromssträcka där olika vikter lät fallas i 200ms från toppläget för att sedan bromsas till stopp såg ut så här. Kolumnerna visar den spänning vi matade, vikten som hängdes på bromsen samt sträckan som vikten föll inklusive det fria fallet. Tabellen är sorterad efter i första hand vikt samt i andra hand spänning.

Spänning(V)	Vikt(kg)	Sträcka(cm)
3	0.3	38
4	0.3	34
5	0.3	30
6	0.3	26
7	0.3	22
8	0.3	22
9	0.3	22
5	0.75	48
6	0.75	40.5
7	0.75	37
8	0.75	36
8	1.35	56

Tabell 2: Resultat från dynamiskts bromstest

I bilaga 3 finns den kod som användes i CanDEV för de dynamiska testerna av bromsen. pwm_duty_cycle_set är den metod som används för att skicka PWM signaler till servot. fd är porten till servot. Vid det statiska testet så flyttades servot till en position som bromsar vikten maximalt som kallas BRAKE_MAX i bilaga 3.

6 Diskussion

I detta avsnitt tar vi upp en generell diskussion om hur vi känt kring projektet, vad som skulle kunna förbättras och vidareutveckling av systemet.

6.1 Handbroms

Vid det första testet såg det ut att stämma med servots specifikationer, då momentet planade ut strax efter sju volt. Likadant såg det ut vid det andra testet där vi kollade bromssträckan. Efter sju volt minskade knappt bromssträckan.

Vid våra tester av handbromsen skulle resultatet kunna förbättras med andra bromsklossar som har högre friktion mot bromsskivan så att den inte glider lika mycket. Fler tester med fler variabler kunde också utföras, till exempel fler positioner på servoarmen eller olika hastigheter på bromsskivan.

Är servot tillräckligt starkt för att fungera i en broms?

I den kontrollerade miljö bromsen testades i och med tanke på att vi endast arbetade på testriggen så är servot tillräckligt starkt för att fungera i en enklare broms. Vid maximal spänning så lyckades bromsen hålla emot krafter upp till 2.35Nm. Vid implementering i bilen kommer fler faktorer in som friktion mot vägen, väder samt hur synkroniserade bromsarna är. Detta är dock ett ämne för vidareutveckling av bilen.

6.2 Utveckling

Projektet har varit lärorikt inom många områden men också oerhört rörigt gällande vilken programvara, hårdvara och kunskaper som krävdes för att kunna slutföra uppgiften. Väldigt mycket tid gick åt till att lista ut vad som skulle vara nästa steg då vi först trodde att vi skulle jobba 100% med CAN-kommunikation men i slutändan blev det mer testning i olika utvecklingsmiljöer samt en hel del mekatronik och elektronik.
Vår målinriktning ändrades ofta under projekts gång när vi fick tillgång till ny information från bland annat handledare.

Är Kvasers mjukvara lättare att använda för att kontrollera bromsen?
Det var väldigt mycket lättare att hitta information på nätet om CubeIDE då det inte är Kvasers egna programvara så i vårat fall var det lättare med CubeIDE. Vid inkoppling av en hastighetssensor så kunde vi se i databladet var vi skulle koppla den medan i CANDev så kunde vi aldrig hitta funktionen för det. Kommer man som utomstående så är CubeIDE lättare att använda men om man jobbat mycket med CANDev tidigare och kan systemet så är nog CANDev ett bättre alternativ då funktionerna i CANDev är skapade för att kunna kommunicera med andra komponenter från Kvaser.

Hur hämtade vi och bearbetade kunskap?
Gällande inhämtning och bearbetning av kunskap så har vi främst hållit oss till CubeIDEs och Kvasers egna dokument där vi läst och gjort ett flertal väldigt simpla program för att få en förståelse av systemen och hur de olika funktionerna fungerar. Utöver detta har vi fått information genom anställda på Kvaser och videos på Youtube och Google.

6.3 Miljö

Alla metall och plastdelar i projektet är delar från hobbyvärlden vilket gör så att alla delar redan är konstruerade och inga nya specifika delar behöver tillverkas. Detta gör att konstruktionen av produkterna kan göras på plats så allt material behövs inte skickas från Sverige utan mycket kan köpas på den lokala marknaden för att spara på miljön.

7 Slutsats och vidareutveckling

Hela rapporten fyller funktionen som en snabbguide där man kan se vilken hårdvara och mjukvara från Kvaser eller STMicroelectronics som behövs för att konfigurera processorn, skicka PWM-signaler, förstå hur bromsriggen är uppbyggd samt hur den fungerar för att sedan kunna vidareutveckla systemet. Bromsen fungerar bra i den kontrollerade miljö vi utvecklade den i där spänningen slutade ge mer bromseffekt vid 8.1V så det blir inte en större bromskraft genom att öka spänningen ytterligare. För att få en större bromskraft så hade andra bromsskivor och bromsklossar kunnat användas som har mer friktion mellan varandra.

När det gäller personlig utveckling känner vi att vi har lärt oss en hel del om saker som är viktigt för en ingenjör att veta så som vikten av bra kommunikation, informationshämtning och bearbetning av information. Mer praktiska egenskaper är exempelvis hantering av spänningsaggregat, skarvning av kablar och hur man angriper problem som man aldrig stött på innan. Vi fick också lära oss hur väsentligt det är med bra kommunikation inom ett företag och att det ibland inte är så lätt.

7.1 Vidareutveckling

Det finns mycket som går att vidareutveckla på projektet då mycket har varit testning och utveckling av diverse program och funktion. Eventuell vidareutveckling kan vara så att bromsriggen är fastmonterad någonstans så man slipper stå och hålla fast den när den körs så den inte ramlar över.

Ett naturligt steg i dagens samhälle gällande bromsar och säkerhet är ett att vidareutveckla till ett ABS-system. Om ett ABS-system ska implementeras så behövs en regulator som kontrollerar att det inte blir totallåsning i bromsarna samt att bromsarna kommunicerar så bilen inte dras ur kurs på grund av bromsningen.

En intelligent koppling direkt till motorn på servo hade tillåtit smidigare moment. Det kan behövas vid utvecklingen av ett ABS-system.

8 Referenser

[1] Kvaser, ”Kvaser: World Leading CAN Development,” Kvaser, [Online]. Available: https://www.kvaser.com/about-us/. [Använd 19 05 2021].

[2] Kvaser, ”About CAN: A Flexible and Powerful Protocol,” 2021. [Online]. Available: https://www.kvaser.com/about-can/. [Använd 19 05 2021].

[3] ”PWM: Pulse Width Modulation: What is it and how does it work?,” Analogic IC Tips, [Online]. Available: https://www.analogictips.com/pulse-width-modulation-pwm/. [Använd 19 05 2021].

[4] Digikey, ”STM32F302RET6,” [Online]. Available: https://www.digikey.se/product-detail/en/stmicroelectronics/STM32F302RET6/497-17418-ND/6166906. [Använd 19 05 2021].

[5] Kvaser, ”Kvaser T-Cannector v2,” [Online]. Available: https://www.kvaser.com/product/kvaser-t-cannector-v2/. [Använd 19 05 2021].

[6] Kvaser, ”Kvaser USBcan Pro 2xHS v2,” [Online]. Available: https://www.kvaser.com/product/kvaser-usbcan-pro-2xhs/. [Använd 19 05 2021].

[7] ST, ”ST-LINK/V2 in-circuit debugger/programmer for STM8 and STM32,” [Online]. Available: https://www.st.com/en/development-tools/st-link-v2.html. [Använd 19 05 2021].

[8] ttssh2, ”Tera Term Home Page,” [Online]. Available: https://ttssh2.osdn.jp/index.html.en. [Använd 19 05 2021].

[9] ST, ”Integrated Development Environment for STM32,” [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html. [Använd 19 05 2021].

[10] Programming VIP, ”STM32 Development - Explanation of Clock System,” [Online]. Available: https://programming.vip/docs/stm32-development-explanation-of-clock-system.html#:~:text=(1)%20HSI%20is%20a%20high,is%20from%204MHz%20to%2016MHz. [Använd 19 05 2021].

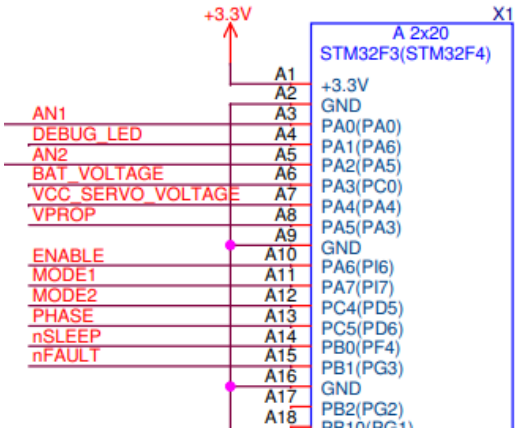
[11] AGFRC, ”www.agf-rc.com,” [Online]. Available: https://www.agf-rc.com/agfrc-coreless-digital-servo-a26chr-p1671195.html. [Använd 19 05 2021].

9 Bilagor

Denna rapport kan vara till stor hjälp för kommande studenter och sommarjobbare som kommer fortsätta arbeta på bromsen och vidareutveckla.

Bilaga 1

Datablad för processorn. Här ser man att PA1 går till DEBUG_LED.



Bilaga 2

Datablad för bärarkortet (SERVO NOD). Här ser man att PC3 går till SERVO_PWM.

Bilaga 3
Kod i CANDev för dynamiska bromstest.

```
#include <kern.h>
#include <pwm/pwm.h> /* PWM driver */

int set_i2c_pot_vcc_servo (uint8_t wra_value);

#define DUTY_MIN 1600
#define DUTY_MAX 8400
#define BRAKE_MIN 3500
#define BRAKE_MAX 6500

int main(void)
{
   task_delay(50);

   int fd;
   set_i2c_pot_vcc_servo(0x57); /* Set voltage */

   fd = pwm_open("/pwm0/3"); /* Activate PWM output for servo pulse */
   //rprintf("fd: %d \n", fd);

   pwm_start(fd); /* Start */
   pwm_frequency_set(fd, 333); /* Set the frequency */

   //pwm_duty_cycle_set(fd, BRAKE_MAX);


   /* DYNAMIC BRAKE TEST */
   pwm_duty_cycle_set(fd, BRAKE_MAX); // Move the servo

   int i;
   for(i=5; i>0; i--){
      rprintf("Brake test in: %d...\n", i);
      task_delay(1000);
   }

   pwm_duty_cycle_set(fd, BRAKE_MIN); /* Release brake */

   task_delay(200);

   rprintf("\n_____STOP_____\n\n", i);
   pwm_duty_cycle_set(fd, BRAKE_MAX); /* Activate brake */

   for(i=10; i>0; i--){
      rprintf("Release in: %d...\n", i);
      task_delay(1000);
   }

   rprintf("\n_____RELEASE_____\n", i);
   pwm_duty_cycle_set(fd, BRAKE_MIN); /* Release brake */
   /* DYNAMIC BRAKE TEST END */

   return(0);
}
```

Bilaga 4
Kod i CubeIDE för bromsning.

```
/*min->max->min code */
int main(void){
    /* Initialise HAL */
    HAL_Init();
    /* Configure the system clock */
    SystemClock_Config();
    MX_TIM1_Init();
    HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_4);

    int32_t Pulse = 1800;

    while (1)
    {
        while(Pulse < 4200){
                TIM1->CCR4 = Pulse;
                Pulse += 10;
                HAL_Delay(10);
            }
        while(Pulse > 1800){
                TIM1->CCR4 = Pulse;
                Pulse -= 10;
                HAL_Delay(10);
            }
    }
}

static void MX_TIM1_Init(void){
htim1.Instance = TIM1;
  htim1.Init.Prescaler = 3;
  htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim1.Init.Period = 6006;
  htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim1.Init.RepetitionCounter = 0;
  htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim1) != HAL_OK)
  {
    Error_Handler();
  }
sConfigOC.Pulse = 3002;
```

Bilaga 5
Kod för blinkande led-lampa i CodeIDE.

```
int main(void){
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();

  while (1)
  {
      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, 1);
      HAL_Delay(100);
      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, 0);
      HAL_Delay(100);
  }
}

static void MX_GPIO_Init(void){
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : LED_GREEN_Pin */
  GPIO_InitStruct.Pin = LED_GREEN_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LED_GREEN_GPIO_Port, &GPIO_InitStruct);
}
```

Bilaga 6

Bild från Tera Term terminalen när handbromsprogrammet i CANDev körs.