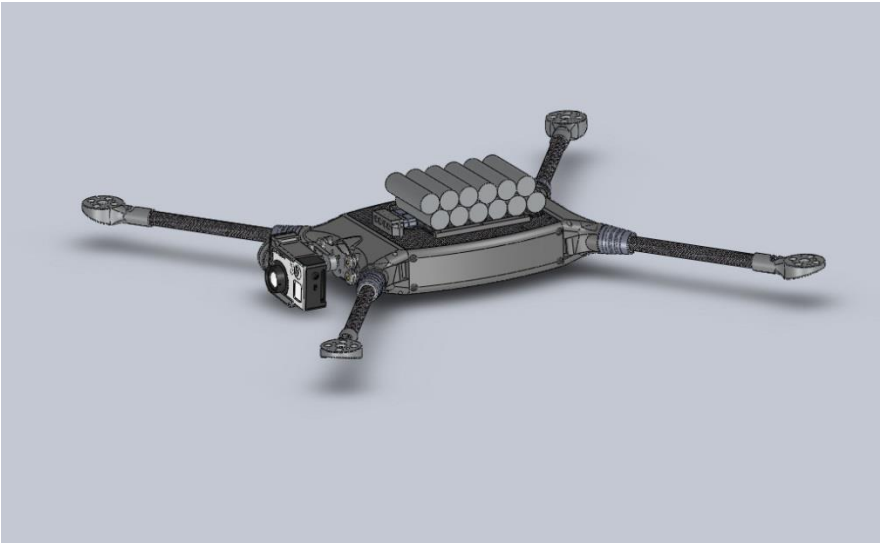


TDD til indlejret softwareudvikling

Jan Sørensen, 19862770

Nikolaj Rahbek, 19972246

Masters Thesis
September 2014

Vejleder: René Rydhof Hansen

AALBORG UNIVERSITET

Aalborg Universitet, Cassiopeia, Institut for Datalogi
Selma Lagerløfs Vej 300
9220 Aalborg Ø

1 Abstract

English:
In this thesis we are analyzing the possibility to use TDD as a design paradigm for small embedded systems. To demonstrate the workflow we implement a controller for a remote controlled quadrocopter. The controller is implemented in C++ for an AVR ATmega32 using Eclipse and google test frame. We end up with a fully functional quadrotorcopter implemented sorely via TDD. We compare TDD for embedded and traditional PC development, and we show how software development can be started even before any hardware exists. Furthermore, we also look briefly on simulation of the embedded code and automated schedulability analysis. We conclude that TDD is as effective for embedded development, as is for traditional PC applications.

Dansk:
I denne afhandling analyserer vi muligheden for at bruge TDD som et design paradigme for små indlejrede systemer. For at demonstrere arbejdsgangen implementerer vi en controller til en fjernstyret quadrokopter. Styringen er implementeret i C++ til en AVR ATmega32 ved hjælp af Eclipse og google test framework. Vi ender med en fuldt funktionel quadrokopter implementeret udelukkende via TDD. Vi sammenligner TDD for indlejret og traditionel PC udvikling og vi viser, hvordan softwareudvikling kan startes, selv før enhver hardware eksisterer. Ydermere, ser vi kort på implementering af en simulator af det indlejrede kode og automatiseret skedulerbarheds analyse. Vi konkluderer, at TDD er lige så effektiv for indlejret udvikling, som er til traditionelle pc-programmer.

2 Indholdsfortegnelse

3 Indledning

Vi har begge en lang erfaring som softwareudviklere til indlejrede systemer. Et af de problemer vi har set igen og igen er, at man har noget kode der virker, men ”man kunne da også bare lige rette en lille detalje”. Man tester det, der er blevet rettet, og alt ser rigtigt ud… indtil man skal bruge en helt anden del af koden, og pludselig opdager man, at der var en afhængighed til det, man tidligere havde rettet, og som man nu har glemt alt om. Jo længere tid der går fra man har introduceret fejlen og til man skal rette den, jo længere tid tager det også at finde den[1].

Indlejret software anvendes ofte med henblik på at simplificere hardware design, da softwaren ofte kan erstatte store elektroniske systemer. Dette kan også betyde, at det ikke er ualmindeligt, at det vil være personer, der er trænet i elektronikudvikling frem for softwareudvikling, der kommer til at skrive koden til disse systemer [1], hvilket muligvis kan have indvirkning på kvaliteten af den skrevne kode [2][2].

Vi har også begge en lang erfaring som software udviklere til PC applikationer. Her har vi set at mange af de problemstillinger omkring regressioner, altså afhængige eller tilbagevendende problemer, i nogen grad kan løses via automatiserede software tests. Disse test kan være fremkommet på flere forskellige måder, men den mest omfattende metoder vi kender til, er testdrevet udvikling (TDD), hvor der skrivers unittest til alt kode, inden selve koden skrives. Dette leder frem til at alle veje igennem koden bliver testet, og dermed, at alle ændringer der har afhængigheder til andre dele i koden vil bliver fundet med det samme fejlen introduceres.

Vi har snakket med flere personer der har eller stadig arbejder inden for udvikling af indlejret software, og som har nævnt at de har kørt automatiserede test af det indlejrede kode på en udviklings PC. Men vi har ikke fundet nogen der kunne fortælle os om hvordan vi kunne lave noget, der minder om den arbejdsgang, vi har prøvet i forbindelse med TDD til udvikling af PC applikationer.

Vores ønske med denne thesis er derfor at finde ud af, om TDD kan bruges i forbindelse med udvikling af software til et indlejret system, om det giver mening, samt hvor meget man egentligt kan teste, da store dele af koden vil være meget hardware-nært.

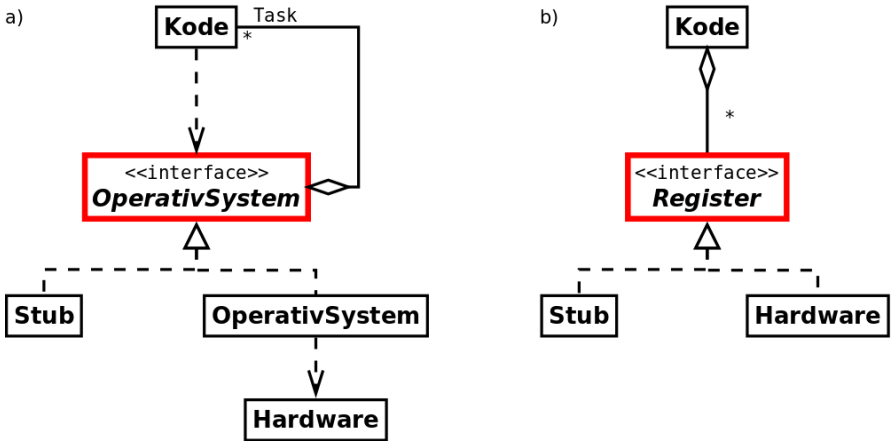
3.1 Problemformulering

Vores forståelse for hvad et indlejret system er, og den definition vi vil bruge i denne rapport er, at et indlejret system vil være et hvilket som helst computersystem, der ikke er en traditionel PC, og udviklingen af et indlejret system kan derfor både inkludere mekanik-, hardware-, softwaredesign, kommunikationsmodeller og meget mere. Ofte vil de indlejrede

[1] “The longer a bug (or vulnerability) goes unfixed, the more it costs to fix it later." - John C. Checco

[2] “There are surprisingly few engineers, though, who have received significant instruction on the fundamental topic of coding style.” – Matt Gordon

systemer, vi har kendskab til have en mindre processor med begrænsede ressourcer og direkte tilgang til hardwaren samt at systemet skal bruge hardware på måder der er så unikke, at der ikke findes generelle drivere til formålet. Eksempelvis at blinke med en lysdiode afhængig af batteri spændingen, målt via en analog til digital konvertering (ADC) målt over et modstandsnetværk. For at sikre sig at alle dele af koden rent faktisk opfører sig korrekt når man retter et sted, ønsker vi at udføre automatiske tests af softwaren, der eksercerer hver enkelt funktion i koden, kaldet unittests. Og hvordan kan man så skrive en unittest, der kan vise at ens kode rent faktisk tænder og slukker dioden på de rigtige tidspunkter?



Figur 1 a) viser arkitekturen af en applikation der kører på en OS, hvor operativsystemet skal stubbes af i test øjemed. b) viser et bare-metal system, hvor registrene skal stubbes af.

Hvis man ser på funktionalitet, der ikke benytter sig af hardware tilgang såsom en matematisk beregning, vil der ikke være nogen forskel på at teste dette for en PC eller en indlejret implantation, da man vil kunne antage, at 2+3 skal give 5 uanset om udregningen udføres på en PC eller et indlejret system. Derfor antager vi, at der hvor TDD for indlejrede systemer adskiller sig mest fra PC applikationer, vil være når vi ser på det "hardwarenære kode". Hvis der anvendes et operativ system (OS) på et indlejret system, vil dette bl.a. kunne definere et interface til hardwaren, som kan sammenlignes med implementering på almindelige PC systemer. Indlejrede systemer uden OS kaldes bare-metal systemer. På disse systemer bliver skedulering og kommunikationen med hardware implementeret dedikeret til systemets formål. Hvor OS giver et abstraktionslag ned imod hardwaren, vil hardware registrene være det nedre abstraktionslag for bare-metal systemer. Dette interface er resultat af processor arkitekturen. For kode, der skal tilgå hardwaren, vil dette kunne testes via teststubbe, der "afspejler" hardware registrene, og dermed vil også hardwarenær kode kunne udvikles via TDD processen. Figur 1 viser denne afkobling generelt - alt tilgang fra koden til hardware, sker via registre, og hvis disse registre kan stubbes af, kan de også testes via unittest.

Dette leder frem til det initierende spørgsmål:

> *Er det muligt at anvende TDD som design paradigme, til udvikling, dokumentation og test, af et indlejret system?*

3.2 Problemafgrænsning

For at efterprøve om det er muligt at bruge TDD til udvikling af software til indlejrede systemer, har vi valgt at anvende implementering af kode til en quadrokopter, da vi mener at denne vil indeholde mange af de problemstillinger, der generelt kan forekomme ved indlejret programmering, såsom realtids krav og direkte hardware tilgang, og derfor vil den være repræsentativ for en klasse af indlejrede systemer.

Styringen skal modtage styresignaler fra en radio kontroller (RC), samt input fra gyro og regulere effekten på 4 motorer ved hjælp af pulsbrede modulation (PWM). Styringen indeholder tidskritisk håndtering af input og har tidskritisk output, samt der er krav til maksimal intern beregningstid. Styringen vil blive bygget op omkring mekanik fra John Jensen[3], kombineret med billige moduler fra dx.com.

Vi har valgt at bruge en Atmel AVR ATmega32 til vores styring. Valget er truffet ud fra at denne processor har de fornødne ind og udgange samt timere. Da AVR familien har mange fællestræk, tillader dette valg også, at portere koden til en større eller mindre processor, hvis dette skulle vise sig nødvendigt eller muligt.

Generelt vil der være både fordele og ulemper uanset om man vælger at implementere sit indlejrede system med et OS eller bare-metal. I dette projekt ønsker vi ikke at argumentere for eller imod nogle af de to. Set ud fra et TDD synspunkt, vil den største forskel mellem traditionel TDD på en PC og TDD på en indlejret platform, ligge i de systemer der ikke har et OS som på en PC. Derfor vælger vi udelukkende at se på bare-metal, da dette bringer TDD problemet tættere på hardwaren.

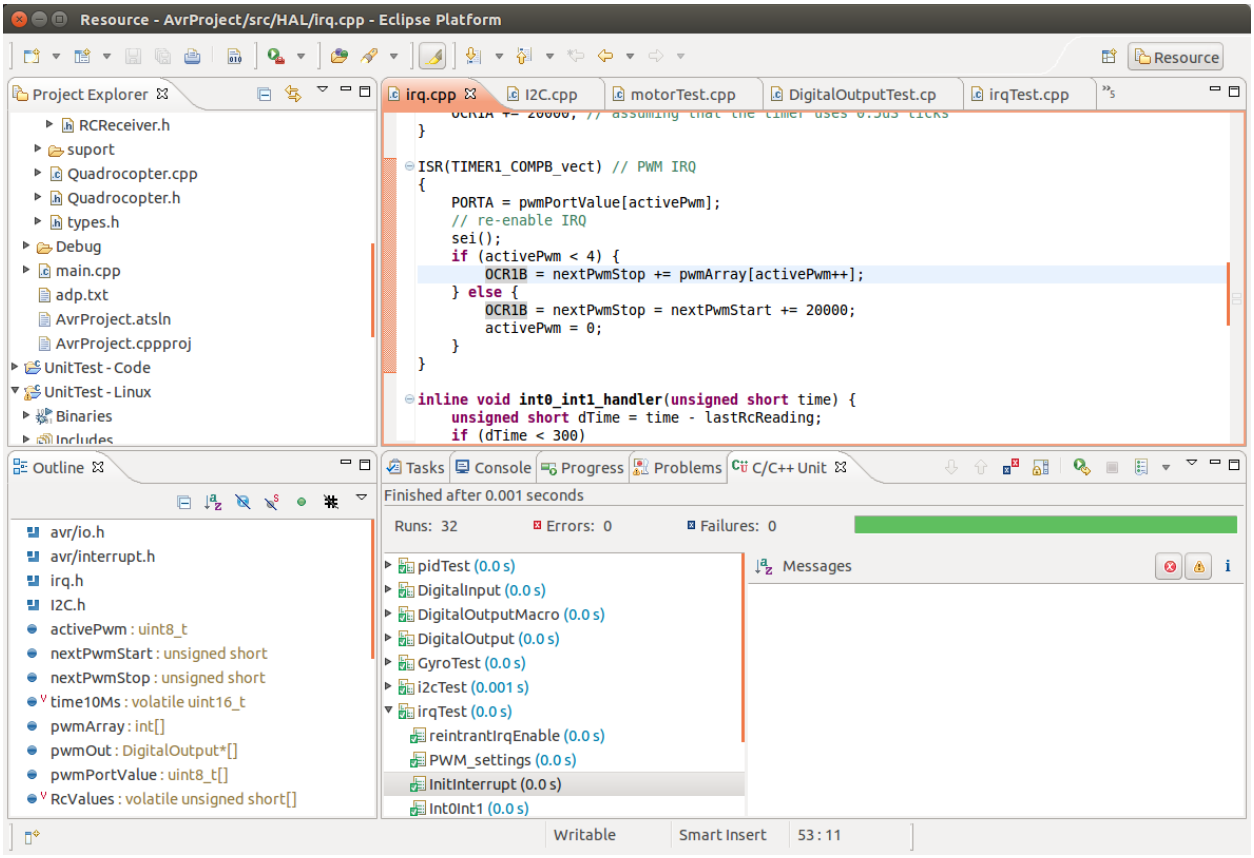
Endelige vil vi undersøge hvilke konsekvenser TDD processen for har for kvaliteten af det implementerede kode, og efterprøve hvordan TDD kan bruges som dokumentation for overfor eksisterende kode, ved at implementere test til koden, efter denne er skrevet.

4 Grundlæggende TDD vs Indlejret

Dette afsnit giver en kort beskrivelse af nogle af de grundlæggende ideer i TDD samt viser et simpelt eksempel og forklarer hvordan TDD skal bruges for at være et effektivt udviklingsværktøj til indlejret udvikling.

[3] En bekendt af forfatterene, erfaren RC modelbygger og pilot.

Det måske vigtigste formål med at anvende TDD er, at man altid har en aktuel test rapport, der viser om der findes fejl i koden. Figur 2 viser et skærmbillede fra udviklingsmiljøet Eclipse, der er blevet sat op til at køre unittest på vores indlejrede quadrokopter projekt.



Figur 2 Udviklings miljøet Eclipse og en testrapport (nederst til højre) der viser at der ikke er fejl i koden.

4.1 Generelt om agile metoder

Agile udviklingsmetoder er en række af udviklings metoder, der baserer sig på udsagnene nævnt i agile manifest [3]. I de senere år er agile udviklingsmetoder blevet meget populære (såsom scrum [4]) og udbredte. Test dreven udvikling (TDD) er måske blevet mere berygtet en populær, og er langt fra så udbredt som andre agile metoder. Årsagerne til dette kan være mange, og ligger uden for emnet for denne rapport. Vi vil blot konstatere, at der er meget delte holdninger til TDD. Der findes adskillige diskussioner på nettet omkring for og imod TDD.

Agile udviklings metoder bliver tit forbundet som værende modsætningen til system development life cycle modellem (SDLC) (også kaldet vandfalds modellen) [5]. Metoderne kan spores tilbage til midt 1970'erne [6], men begrebet agile udviklingsmetoder opstod først i 1990'erne [7] sammen med udviklings metoder som "scrum", "extreme programming" med flere, hvor kritikkere af vandfaldsmodellen ønskede at skabe en let og smidig udviklingsmodel, der kunne tilpasses efterhånden som udviklere i en projektgruppe

fik bedre kendskab til projektet, og ønskede at undgå micromanagement som SDLC blev beskyldt for at være [8], samt øgenavnet ”vandfalds modellen” opstod.

Kritikkere af agile udviklingsmetoder går ofte på, at agile metoder er en dårlig undskyldning for ikke at skrive dokumentation, og at ”iterative processer” bare er en serie af små vandfald [5].

En kort opsummering og sammenligning af udviklings modeller kan findes i [9].

4.2 Generelt omkring TDD

Som alle andre agile udviklingsmetoder [10], er TDD baseret på iterative forløb, hvor koden gradvis bliver udviklet sideløbende med at dennes arkitektur opstår. I TDD bliver koden testet ved at skrive tests til enkelte dele (units) af koden, altså funktioner, metoder eller macroer, allerede inden selve koden er skrevet. TDD er på denne måde en white-box [11] testmetode, altså hvor test koden bliver skrevet til at eksercere konkrete spidsfindigheder i implementationen af koden under test (UUT), i modsætning til black-box [11] testing, hvor forfatteren af test koden ikke kender til selve implementationen, og dermed kun kan teste funktionaliteten af UUT. Andre testmetoder inkluderer modul test, som tester samspillet mellem blokke af kode (eksempelvis klasser) [11] kapitel 8.1.5 og integrations test, der tester systemets overordnede funktionalitet [11] kapitel 8.1.5.

Grundsætningerne i TDD er meget enkle [12]

1. Skriv eller ret kun kode, hvis der er en test der fejler
2. Eliminer alle gentagelser

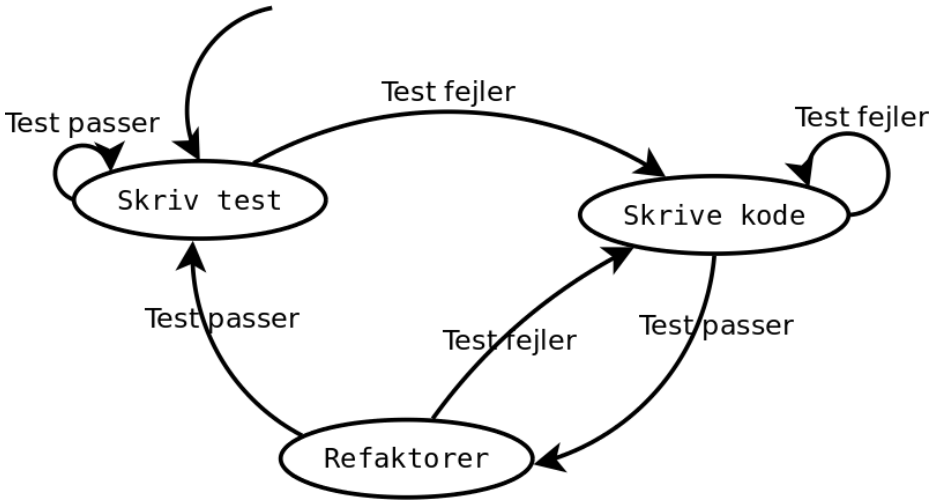
Traditionelt dikterer TDD, i modsætning til eksempelvis SDLC, at man ikke analyserer problemet til bunds, inden implementeringen af produktionskoden begynder [13]. På denne måde sikres det, at der ikke bruges tid på at diskutere eller implementere funktionalitet, der viser sig ikke at være nødvendige for projektet. På den anden side vil man ikke kunne implementere hvad som helt uden at have kendskab til, hvad kunden ønsker [14]. Derfor skal problemet først brydes ned i nogle mindre dele. Dette gøres ud fra en overordnet arkitektur nedbrydning (evt. via 4+1 se (Appendiks 2) eller en simpel sproglig analyse se afsnit 5.1). Når projektet er brudt ned i nogle dele, der er så detaljerede at alle blokke har beskrevet interfaces (ikke nødvendigvis i detaljer) til hinanden, kan udviklere uafhængige af hinanden begynde at udvikle på de enkelte dele, og hvis det viser sig nødvendigt, kan de overordnede dele brydes yderligere ned.

Filosofien er, at hvis man lader arkitekturen udvikle sig via evolution, får man over tid den bedst egnede arkitektur (overlevelse af den stærkeste). Når dette er sagt, er det dog ikke

ensbetydende med, at man ikke skal lægge en overordnet plan for arkitektur, interfaces eller deployering [11][4].

TDD arbejdsgangen er vist i Figur 3 og er som følger:

1. **Skriv test**. I TDD er det vigtigt, at der altid er 100% testdækning. Derfor tilføjer man en ny test, der kan vise, at der er et behov for at skrive eller ændre noget i koden, inden man skriver noget kode. Dette betyder også, at hvis den nye test man har tilføjet ikke fejler, er der enten ikke fejl i koden, eller også er der fejl i den nye test. De enkelte test der skrives kaldes test cases, og en samling af test cases der tester et modul kaldes en test suite.
2. **Skriv kode**. Eftersom der på dette tidspunkt er en test der fejler, skal der skrives eller rettes kode, så testen kan bestå. TDD dikterer, at man skriver mindst muligt kode for at ikke fejler testen. Så længe der er tests der fejler, skal man skrive/rette kode. Kode der bliver ekserceret af en test case kaldes unit under test (UUT).
3. **Refaktorer**. I denne sammenhæng betyder refaktorering generelt bare "oprydning". Når man har skrevet noget kode, og ingen test fejler, skal man rydde op i koden. Efter som man har 100% test dækning af sin kode, kan man roligt rydde grundigt op, da eventuelle fejl man introducerer op oprydningen, vil blive fundet af testen med det samme.



Figur 3 TDD arbejdsgangen opstillet som et tilstandsdiagram. Den enkelte udvikler kan være i en af tre tilstande, være ved at skrive test, skrive kode eller rydde op.

I denne rapport forudsætter vi, at læseren har kendskab til TDD som en generel proces. Hvis ikke, findes der i (Appendiks 1) et beskrevet eksempel, hvor der implementeres et simpelt program til at udskrive om et tal er et primtal. Appendiks 1 beskriver trin for trin i hvilken rækkefølge hvad gøres. For en detaljeret beskrivelse henvises til [11] og [12].

[4] ③ i Design Patterns: Elements of Reusable Object-Oriented Software "overvej hvad der kan variere inden du skriver kode"

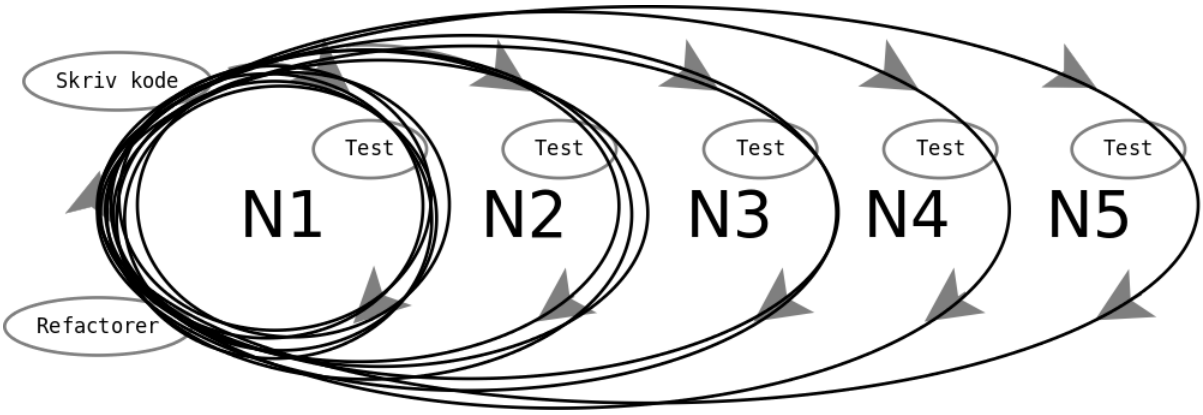
Når man bevæger sig mellem de enkelte faser af TDD arbejdsgangen vist i Figur 3, dikterer [12], at man skal tage små skridt. Med dette mener han, at man ikke skal skrive en komplet test specifikation til hele sit indlejrede system, inden man begynder at implementere noget kode. Spørgsmålet er bare hvor små "små skridt" så er. Umiddelbart skal det være så små stykker kode, at man let kan overskue hvad der er gået galt, i tilfælde af at testen fejler. Robert Martin siger at "alle fejl er fejl" og at "bygge fejl og warnings er også fejl". Med det mener han, at kompilerfejl også er fejl på lige fod med en test der fejler, og at man derfor først inkluderer en headerfil for den klasse man har tænkt sig at implementere. Nu vil kompileringen fejle, da testprojektet ikke kan finde den fil, man prøver at inkludere. Så kan man oprette filen, og se at kompileren nu kan kompilere. Så skal man oprette en instans af en klasse, man har tænkt sig at implementere, men som stadig ikke er implementeret. Eftersom test projektet nu ikke længere kan bygge, pga. reference til et ukendt objekt, skal man så derpå oprette denne klasse. Og på denne måde vil det være muligt at sige, at man har en test, der påviser, at man har behov for at tilføje et nyt objekt til projektet.

Hvilken størrelse skridt man ønsker at tage, vil være noget man finder frem til efterhånden, som man bliver vandt til arbejdsgangen.

4.3 TDD for indlejret kode

Al software der anvendes, vil blive testet på et tidspunkt. Hvis der ikke er defineret nogen test af koden, vil brugen af softwaren være en test. Hvis brugeren ikke har mulighed for at rapportere tilbage til udvikleren, kan man enten vælge at acceptere det eller finde en alternativ løsning. Uanset hvad brugeren gør, vil fejl kunne lede til uoprettelig skade, samt at det er ikke klart for brugeren, hvilke fejl der er i en given software. Udviklingstiden kunne afkortes ved at stoppe fejl hurtigst muligt samt finde en metode til at sikre, at de ikke opstår igen.

Ved TDD skriver man testen først, og koden bagefter, hvilket betyder, at der altid er 100% dækkende test af alt kode, hvilket igen vil bevirke, at man vil blive gjort opmærksom på det, hvis man på et tidspunkt introducerer en fejl i noget kode, der allerede er testet og har virket. Vi vælger at opdele test efter omfang / tid til afvikling, hvor den simple test er at koden kan kompileres til test afvikling (kompilerings fejl er også fejl), og en mere kompliceret test er på et samlet system i naturlige omgivelser.



Figur 4 illustrerer forskellige testniveauer, ud fra [15] Figur 5.1

Figur 4 er en udvidelse af Figur 3, og illustrerer forskellige testniveauer og hvor ofte der bliver testet på de forskellige niveauer. Test på Niveau N1 er automatiske og kan udføres ofte med minimale omkostninger, derfor bliver de udført oftere. En gang imellem udfører man så en test på N2, som er lidt mere tidskrævende og uanset udfaldet af N2 testen, går man så tilbage til N1. Tests på N5 er de dyreste og mest tidskrævende, og udføres derfor kun sjældent. Ud fra [15] Figur 5.1 er der defineret 5 stages, som vi fortolker som testniveauer:

N1: TDD iterationer, automatisk test som køres hver gang der er foretaget ændringer. Dette sker for at sikre at ny kode opfører sig som forventet, og at rettelser i koden ikke ødelægger allerede skrevet kode.

N2: Target kompilering, koden bygges med target kompiler. Sikrer eksempelvis at der ikke er brugt typer der ikke er defineret i target.

N3: Simulatortest. Testcases der baserer sig på simuleret hardware. Dette kunne eksempelvis teste samtidighed mellem tasks, eller påvirkninger af/fra det fysiske miljø.

N4: HW test, programmet afvikles på target (smoke test), her testes hver enkelt hardware driver for sig selv. Udvikleren kan afprøve enkeltdele af koden.

N5: Integrationstest, test af det samlede system i et naturligt miljø.

Listen af testniveauer kunne fortsættes efter behov, eksempelvis:

N6: Kunde acceptance test.

N7: Myndighedsgodkendelse.

Ovenstående præsenterer også en valid udviklingsmodel for almindelig PC software, men her vil indholdet at de enkelte punkter være anderledes. Eksempelvis kunne N2 være, at der

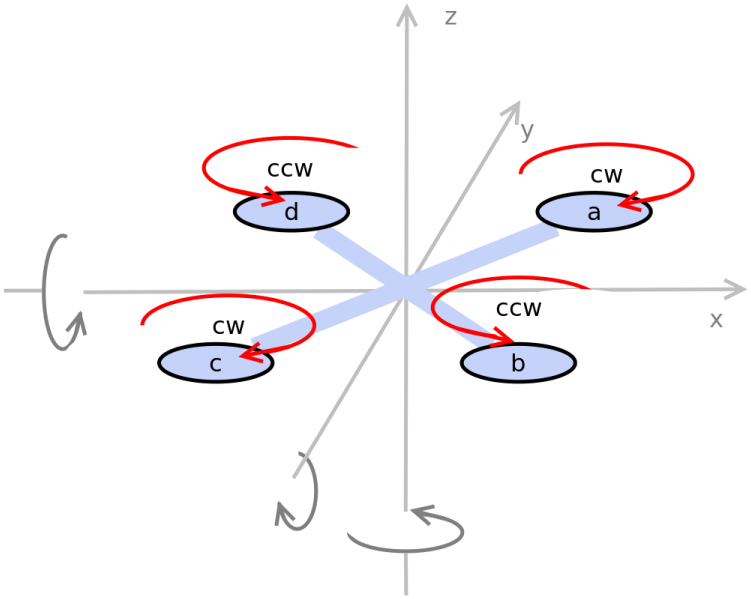
kan krydskompileres til andre platforme. N3 Selv PC programmer kan have relationer til hardware, der kan simuleres i denne test. N4 de enkelte delmoduler testes på PC. N5 alle moduler testes sammen. N6 og N7 vil ikke være anderledes.

5 Analyse

Inden vi kan gå i gang med at implementere selve quadrokopteren, er der nogle detaljer og begreber, der er nødvendige at få på plads først. I dette afsnit viser vi hvordan projektet nedbrydes til mindre dele, så vi kan begynde at arbejde på det, samt hvordan projektet blev sat op i vores udviklingsmiljø.

5.1 Anatomien for en quadrokopter

En quadrokopter har, som navnet antyder, fire roterer. Alle rotorerne er orienteret vandret, så de alle medvirker til at give opdrift, i modsætning til en normal helikopter, der har en hovedrotor, der giver opdriften, og en mindre halerotor der "kun" bidrager til at styre helikopteren. På en helikopter styres retningen ved at give mere eller mindre gas på halerotoren, og fremdrift genereres ved at justere vinklen (pitch) på hovedrotoren.



Figur 5 forestiller en skitse af en quadrokopter i en 3 dimensionalt koordinat system, hvor positiv X flyveretningen. De fire rotorer er navngivet a, b, c og d og det ses at de parvis roterer samme vej for at udligne kræfter i Yaw planet.

På en quadrokopter er propellerne meget simple. Ligesom på en flyvemaskine kan man ikke justere pitch på disse, og opdrift bliver udelukkende et resultat af at give mere eller mindre gas. Propellerne er orienteret således, at to rotorer diagonalt over for hinanden roterer med uret, og de to andre imod uret. Forklaring på dette følger om lidt. Figur 5 viser en quadrokopter i en 3 dimensionalt koordinatsystem.

Hvis en quadrokopter står stille i luften, og man ønsker, at den skal blive stående der, skal alle fire rotorer kører lige hurtigt rundt. Da der ikke er noget, der bringer den ud af balance, vil den kun kunne bevæge sig lodret op og ned.

Hvis man ønsker at quadrokopteren skal bevæge sig fremad, altså positiv x retning på Figur 5, skal quadrokopteren først roteres i negativ retning (med uret) om y aksen (pitch). Dette gøres ved at sænke hastigheden på de to forreste propeller, a og b, og øge hastigheden på de to bagerste, c og d. På denne måde er summen af propellernes hastighed ens, og dermed er den samlede opdrift også cirka ens, bortset fra at opdriften er set i forhold til quadrokopteren. Dette betyder, at da den før var udsat for 1 G lodret nedaf, vil quadrokopteren nu stadig kun modarbejde 1 G, og da den accelererer fremad, betyder det også, at den accelererer nedad og taber højde, hvis ikke der kompenseres ved at give mere gas på alle propellerne. Quadrokopterens acceleration vil altså være afhængig af dens vinkel, og hastighed opnås som et integral af vinklen over tid.

For at stoppe quadrokopteren igen, skal man så rotere den positivt omkring y aksen ved at hæve hastigheden på rotor a og b og sænke hastigheden på c og d. Dette vil bringe snuden op på quadrokopteren, og standse den igen.

At få quadrokopteren til at panorere til højre og venstre gøres efter samme princip som at styre den frem og tilbage, blot at den skal roteres omkring x aksen (roll).

Hvis man vil have quadrokopteren til at ændre kompas retning, altså rotation omkring z aksen (yaw), gøres dette ved at ændre hastigheden for propellerne diagonalt. Hvis man ønsker, at den skal rotere med uret eller fra nord mod øst, gøres dette ved at sænke hastigheden for de propeller, der roterer med uret, a og c, og samtidig øge hastigheden for de der roterer mod uret, b og d, for ikke at ændre opdriften. Dette ændrer vindmodstanden for propellerne, hvilket resulterer i, at de propeller, der drejer mod uret, vil yde større modstand væk fra centrum af quadrokopteren, hvilket igen vil resultere i en kraft, der peger med uret for quadrotokopteren.

5.2 Problem nedbrydning og arkitektur

En væsentlig forskel på indlejrede programmer og almindelige PC programmer er, at den indlejrede processor skal kontrollere forskellige perifere hardware enheder, som kan være integreret eller tilsluttet eksternt. Ved kode til PC vil hardwareabstraktionen være håndteret af et OS. Denne abstraktion giver et naturligt interface, som kan stubbes af i TDD. Ved bare-metal (indlejret system uden OS) systemer findes dette interface ikke, da OS ikke forefindes. Ved indlejrede systemer vil det også være muligt at anvende OS til at kontrollere hardware, herved bliver forskellen ikke så stor på de to systemer. Vi har her valgt at undersøge systemer uden OS, hvor programmøren skal kode alt fra bunden. Indlejrede systemer er ofte anvendt til styring og regulering. Til dette formål bruger vi quadrokopter som et styrings eksempel, da dette system stiller realtids krav.

Arkitekturen i software kan opstå på flere måder. En simpel metode til at komme i gang med er den sprogligt orienterede metode [11] kapitel 15.2. Alternativt kan der laves en 4 + 1 arkitekturanalyse, eller man kan lade arkitekturen opstå via evolution. Et af

grundprincipperne i TDD er, at man ikke implementere en funktionalitet, før man har brug for denne. Dette gælder også for arkitekturen. Med andre ord, hvis ikke man har brug for en service orienteret arkitektur (SOA) [16] fra starten, er der ikke nogen grund til at gå i gang med at implementere en sådan. På den anden side, hvis man ved, at der skal laves flere forskellige produktvarianter med samme software, så er der en god ide at tænke SOA ind fra starten.

Af samme årsag ønskede vi kun at gøre os minimale overvejelser omkring arkitekturen for quadrokopteren, og herefter lade arkitekturen udvikle sig efterhånden som projektet blev implementeret og testet. Målet var så, at vi til sidst skulle kunne sammenligne med, hvad vi ville være kommet frem til, hvis vi havde lavet en grundig arkitekturanalyse fra starten.

For at starte dette, lavede vi derfor en hurtig 4+1 analyse af vores behov se (Appendiks 2), og lagde denne analyse til side. Herefter lavede vi en sproglig analyse af problemstillingen (se afsnittet 5.3), og begyndte at implementere quadrokopteren ud fra hvad denne analyse havde givet.

Efter quadrokopteren var blevet færdigimplementeret, har vi så haft mulighed for at sammenligne den arkitektur, der er opstået med den, vi analyserede os frem til. Og der var forskelle om end dog måske flere fællestræk end forventet.

De dele, der var blevet implementeret på samme måde, som vores analyse havde fundet frem til, var alle de store dele. Der var en hovedklasse, en regulerings klasse, et driver lag og et hardware abstraktions lag.

Der var tre store forskelle imellem det, der blev implementeret og det vi havde designet os frem til. Oprindeligt, i 4+1 analysen, havde vi tænkt, på at vi ville undersøge noget med distribuerede enheder, og at vi derfor skulle have et interface til en anden computer, som på simpel vis skulle kunne styre quadrokopteren. Dette interface er vi aldrig nået til at skulle implementere. Til gengæld viste det sig (måske ikke overraskende), at det var en god ide at kunne logge data fra quadrokopteren. Vi har derfor implementeret et log interface i stedet for. Men den største forskel er nok i selve quadrokopterklassen, er den klasse, der skal samle alle de enkelte komponenter og gøre dem i stand til at styre en quadrokopter. Her havde vi oprindelige tænkt et meget skarpt adskilt interface ind, hvor alle klasser kunne aflevere et resultat i et format som den næste klasse kunne arbejde videre med. I praksis viste det sig, at disse interfaces ikke blev helt så skarpt adskilte, og hvis vi engang i fremtiden vil genbruge noget kode, eksempelvis til at implementere styringen til en flyvemaskine, så bliver der lidt tilretning.

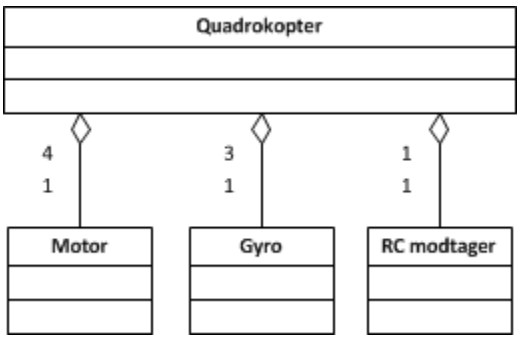
Om resultatet er som forventet, eller om det er overraskende, er svært at sige. Eftersom vi selv havde lavet 4+1 analysen, er der mulighed for at vi har prøvet at tilstræbe om et design, der mindede om det. På den anden side er der i princippet heller ikke noget i vejen for, at vi kunne refaktorere og rydde op i vores kode og på den måde måske få afkoblet interfacene endnu mere.

5.3 Implementering

For at komme i gang med en initiel arkitektur valgte vi at anvende en sproglig orienteret model [11] kapitel 15.2. Metoden går i korte træk ud på at lave en tekstuel beskrivelse af produktet, og navneordne i den beskrivelse vil så være de klasser der skal implementeres. Beskrivelsen er som følger:

> *"Implementer styringen til en radiostyret **quadrokopter**. En **quadrokopter** har fire **motorer** som sørger for at holde den flyvende, og modtager **styresignaler** fra brugeren via en **fjernbetjening**. For at hjælpe med at holde den balancerende, bruger den input fra en **gyro**. Signalet fra **fjernbetjeningen** og **gyroen** bliver født ind i en motor **regulator**, hvorfra kræften til hver enkelt **motor** beregnes."*

Ud fra teksten at projektet består af en quadrokopterklasse, som har en instans af en fjernbetjening og en gyro. En datastruktur af typen "styresignal" skal fødes ind i en motor regulator og så videre. Og så har vi brugt en del af vores erfaring på området, og vi tror på at styresignalerne bliver simple tal-værdier fra modtager og gyro, og dermed er de ikke tegnet med i klassediagrammet. Vi vælger også, at lade motor og motorregulering smelte sammen, da motor stort ser kun vil være en direkte spejling af et PWM register i hardwaren. Dette fører til klassediagrammet i Figur 6



Figur 6 Simpelt klasse diagram af den overordnede arkitektur for styringen til quadrokopteren.

Implementering af TDD til selve quadrokopterklassen kan udføres efter samme metode som anvendes ved større systemer på en PC, det vil sige, ved at lave en stub til hver af de klasser, den afhænger af, da quadrokopter klassen ikke har direkte hardwaretilgang. Motor, Gyro og RC modtagerklasserne adskiller sig derfra, da de alle skal interagere med ekstern hardware, skal TDD test funktioner kunne tilgå / teste på hardware registre i systemet.

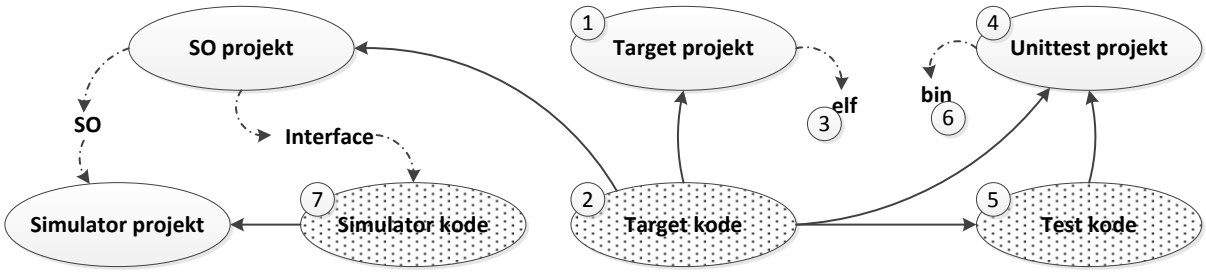
I Appendiks 2 er der vist en 4+1 analyse, der giver et sammenligneligt resultat.

5.3.1 Opsætning af udviklingsmiljø

Mange moderne udviklingsmiljøer enten har, eller har mulighed for at integrere med et eller flere unittest frameworks. Eksempelvis kommer Eclipse EE ide som standard med et vindue til grafisk visning af unittest reporter (se Figur 2), og for at skrive en ny unittest trykker man bare "New test case" i menuen, på samme måde som man ville have valgt at ny klasse eller file. Vi har desværre ikke fundet noget, der integrerer unittest helt så enkelt til

indlejret udvikling. I dette afsnit fortæller vi ganske kort om de build targets man skal sætte op, for at kunne lave et projekt, der kan teste ens indlejrede kode. Dette afsnit vil ikke give en detaljeret beskrivelse af opsætningen, da dette kan variere meget alt efter hvilket udviklingsmiljø man bruger og hvilken microcontroller man bruger. Dog har vi også skrevet en udførlig guide med skærmbilleder, der er frit tilgængelig på [17], som viser hvordan vi har sat vores testmiljø op for dette projekt.

Figur 7 viser de projekter/build targets (hvide bobler), kode (prikkede bobler) og outputs fra projektere (3 og 6) vi har sat op i vores udviklingsmiljø for implementationen af styringen til quadrokopteren.



Figur 7 Viser de projekter der skal sættes op for at kunne teste koden til et indlejret system fra udviklings PCen. Inderst er der tre kodebaser (target-, test,- og simulator kode) der bliver brugt i fire projekter (target, unittest, shared object og simulator).

For at bygge et kodeprojekt til et indlejret projekt (Figur 7, 1), er det nødvendigt at have nogle filer med kildekode (Figur 7, 2), eksempelvis c filer, samt en beskrivelse af hvordan disse skal bygges, eksempelvis en makefil, og resultatet af dette bliver en binær fil (Figur 7, 3), eksempelvis en elf fil, der skal uploades i target. Når denne binære fil bliver afviklet på target, starter afviklingen fra en main funktion. Denne binære fil vil ikke kunne afvikles på den PC, man har udviklet den på, så der skal gøre noget andet for at få det indlejrede projekt gjort klar til TDD.

For at kunne køre unittests på koden skal der derfor laves endnu et projekt (Figur 7, 4), der kan afvikles på udviklings PC'en. Dette projekt skal inkludere kildekoden fra det indlejrede projekt (Figur 7, 2), samt alle de kildekode filer (Figur 7, 5) der beskriver hvordan testene skal udføres. Unittest projektet skal også implementere sin egen main funktion. På denne måde vil der ikke være noget fra det indlejrede projekt, der bliver startet op fra starten, når unittest projektet startes. Når unittesten afvikles (Figur 7, 6) på udviklings PC'en, vil det være muligt at udføre funktionerne fra de inkluderede indlejrede kilde filer under kendte omstændigheder, og måle på hvordan de kaldte funktioner påvirker test miljøet. Man skal dog være opmærksom på at forskelle i arkitekturen på den indlejrede processor og test platformen kan medføre markante afvigelser i testene. Eksempelvis vil Listing 1 give forskellige resultater alt efter endianess for den platform det afvikles på.

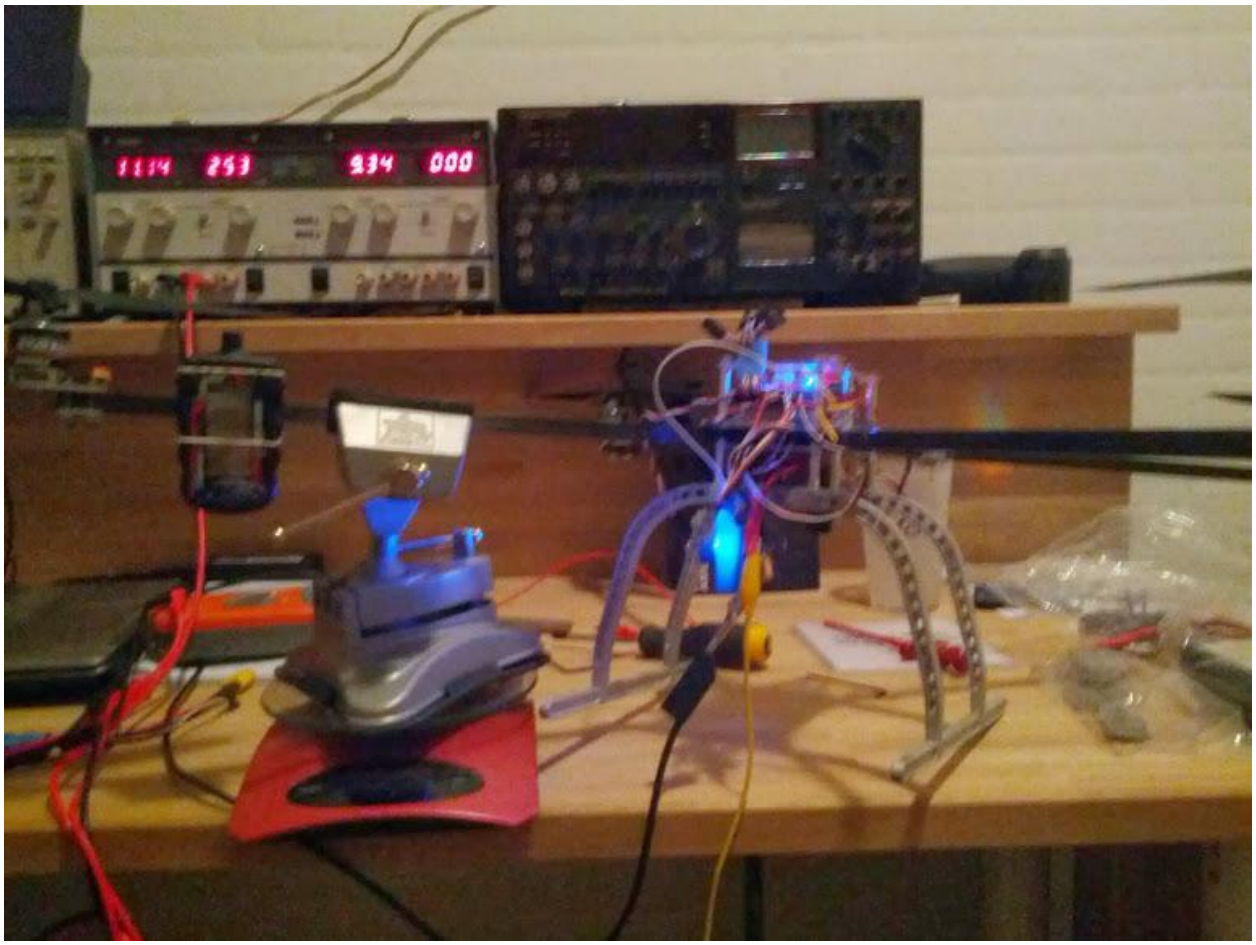
```
uint8_t data[] = {0x12, 0x34};
```

```
uint16_t encodedData = *(uint16_t*)data;
```

Listing 1 Viser en let konvertering fra data i et array, til en dataværdi der kan regnes videre med. Problemet er blot, at encodedData vil have forskellig værdi, om koden bliver afviklet på en platform med big eller small endian.

I nogle tilfælde kan det være relevant at lave en simulering af det indlejrede kode, eksempelvis at lave en grafisk visning af, hvordan en motorregulering vil virke. Det kan gøres ved at lave en simulator (Figur 7, 7) som simulerer dele af en fysisk verden, eksempelvis belastning af motoren i regulatoren. Simulatoren skal kunne aflæse nogle påvirkninger fra det indlejrede kode og skal kunne give stimuli, så den indlejrede kode eksempelvis prøver at aflæse motorhastigheden, samt at simulatoren skal kunne give sin beregnede værdi til det indlejrede projekt. Her er det vigtigt at nævne, at en simulation aldrig bliver bedre end den model simulatoren har af det miljø, som den skal prøve at simulere. Hvis man oplever at ens indlejrede produkt opfører sig markant anderledes i den virkelige verden, end var simulationen vidste, kan det derfor være nødvendig/ønskeligt at gå tilbage og rette modellen i simulatoren.

For at prøve at lave en simulator prøvede vi til quadrokopteren, at bygge et shared object (.SO) af det indlejrede kode. Via Java Native Interface (JNI) kunne vi så lave en simulator i Java, der kunne "måle" en opdrift fra en af motorerne, og kunne føde påvirkningen fra propellen tilbage i det indlejrede system, ved at "give" en vinkel acceleration til gyroen. Figur 8 viser testopstillingen, hvor en motor og propel blev spændt op i en bordskruestik (der var tungere end hvad propellen kunne løfte), og stillet på en køkkenvægt. På denne måde kunne vi måle forholdet mellem motor PWM og hvilken opdrift denne resulterede i. Da dette emne kun relaterer svagt til dette thesis egentlige formål, blev der ikke gjort videre ud af at gøre simulatoren 100% korrekt eller at implementere en fuld simulation i denne.



Figur 8 Testopstilling for at måle opdrift som funktion af den kraft der blev født til motoren.

5.3.2 Lag Opdeling HAL

Hvis et indlejret system betragtes i software lag, hvor hardwaren er de nederste lag og overordnet beskrivelse af interaktion / funktionalitet er øverst, vil man kunne lave opdelingen vist i Figur 9. Figuren viser et programmeringsinterface (API) eller brugergrænseflade øverst (UI). Herfra kan en bruger eller et kontrollerende system bede om at få udført en operation, som eksempelvis kunne kræve en forespørgsel til noget hardware. Denne forespørgsel skal så divageres ned gennem alle lagene i Figur 9. Under API/UI findes et lag med forretningslogik (BL). Dette lag er ansvarlig for beregninger og modelleringer af systemet, eksempelvis udregningerne i vores PID regulering. Herunder igen findes et driverlag (DDL) hvorfra tilgang til hardware er afkoblet. I vores quadrokopter har vi bl.a. en gyro driver.

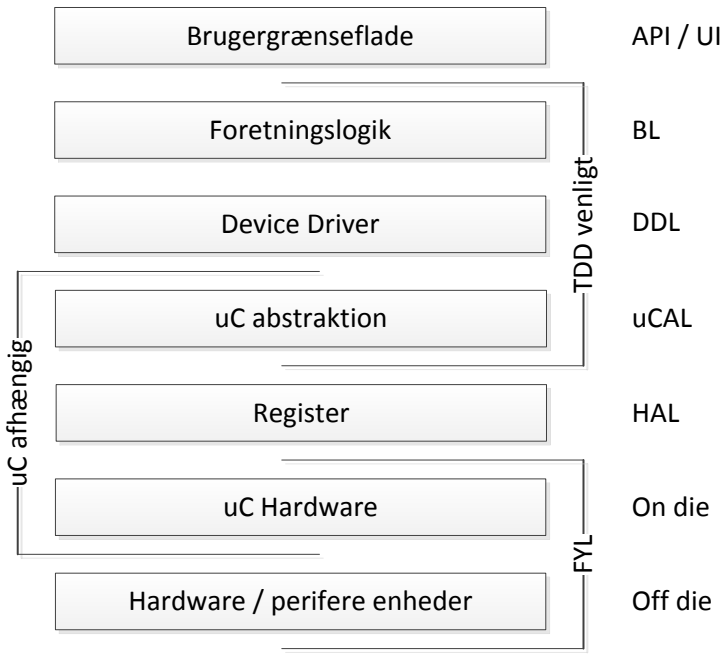
Fra starten af vores projekt havde vi overvejet at se på flere forskellige microcontrollere, så for at få en let måde at afkoble koden fra den processor det skulle køre på, havde vi valgt at designe et microcontroller abstraktionslag (uCal). Tanken med dette lag skulle være, at køre de højereliggende lag uafhængige af den hardware, de skulle afvikles på. Laget blev dog aldrig implementeret, da det vidste sig at være uinteressant (og muligvis også fordi vi indså hvor meget arbejde, der skal ligges i at vedligeholde et sådant lag), da dette ville betyde at

alt over dette lag ville kunne testes og implementeres på samme måde som beskrevet i afsnit 8.1, og alt under dette lag ville kun implementeres som, hvordan vi nu alligevel skulle gøre det (beskrevet i afsnit 0).

Lagene over uCAL anser vi derfor som værende simple at teste, da de i princippet ikke har noget med hardwaren at gøre, og derfor kommer til at minde meget om, hvordan TDD skal udføres for udvikling af en PC applikation. Lagene fra uCAL og herunder er derimod der hvor TDD for indlejret udvikling adskiller sig fra udvikling af PC applikationer, da disse lag er afhængige af, hvilken hardware de skal afvikles på og muligvis kræver direkte adgang til specifikke dele af hardwaren.

Under uCAL laget findes et hardware abstraktionslag (HAL). Dette vil i mange tilfælde være leveret af producenten af microcontrolleren, da dette lag skal gøre det muligt at tilgå de perifere enheder, der er bygget med på samme silicium som processoren (On die). I quadrokopterstyringen bruger vores implementation af gyro klassen en I2C driver, som bruger en hardware I2C driver, der er implementeret on-die. Denne on-die I2C driver bruges til at kommunikere med vores gyrokreds, som ikke findes på samme stykke silicium som processoren, og derfor betegnes som værende off-die.

Med hensyn til TDD er det interessant at konstatere hvor meget af koden, der kan omfattes af TDD. Ved brug af en opdeling som vist i Figur 9, vil det det være muligt at teste de enkelte lag hver for sig eller kombinere flere. Hvor man vil lægge testgrænser, vil afhænge af de enkelte projekters opbygning, kompleksitet og formålet med testen.



Figur 9 Lagopdeling af et indlejret projekt, øverst ses en grænseflade for brugen af systemet, nederst er noget perifer hardware som systemet gør brug af.

Da det ikke er muligt at teste, hvordan den rigtige hardware opfører sig via en simulering i en automatiseret softwaretest, ønskes det at grænsefladen mellem koden og hardwaren (HAL) bliver så tynd som mulig. I vores tilfælde bliver hardwaren tilgået via nogle registre, der er defineret i en header fil (io.h) leveret af chipproducenten. Ved at stubbe denne fil af (forklaring følger i afsnit 0), får vi et tyndt HAL. Vi vælger, at det ikke er nødvendigt at teste denne meget tynde HAL afstubning, da den rigtige HAL (io.h) som er leveret af chipproducenten må formodes at være blevet testet af adskillige brugere før os, og dermed ville eventuelle fejl være blevet fundet (denne problemstilling beskrives også, og der gives argumenter i afsnit 0).

Det valgte test framework tager ikke højde for at man i C og C++ kan skrive direkte til hukommelsen / register ved hjælp af en pointer. Hvis denne form for skrivninger er nødvendige i et projekt, vil det kræve ændringer i test opsætningen. Som f. eks. Hvis en funktion vil nulstille alle hardware registre i en samlet blok, ved at tage en pointer til første register og iterere til det sidste register.

5.3.3 Task's og skedulering

Som beskrevet i afsnit 5.3 skal vores quadrokopter styring have 3 ansvarsområder. Disse ansvarsområder skal også kunne afvikles "på samme tid". Så for at opnå parallelitet skal disse ansvar implementeres i tasks.

1. Modtage data fra fjernbetjening (RC)
2. Beregne effekten til hver motor (Quadrokopter)
3. Generering af PWM til motorstyringen

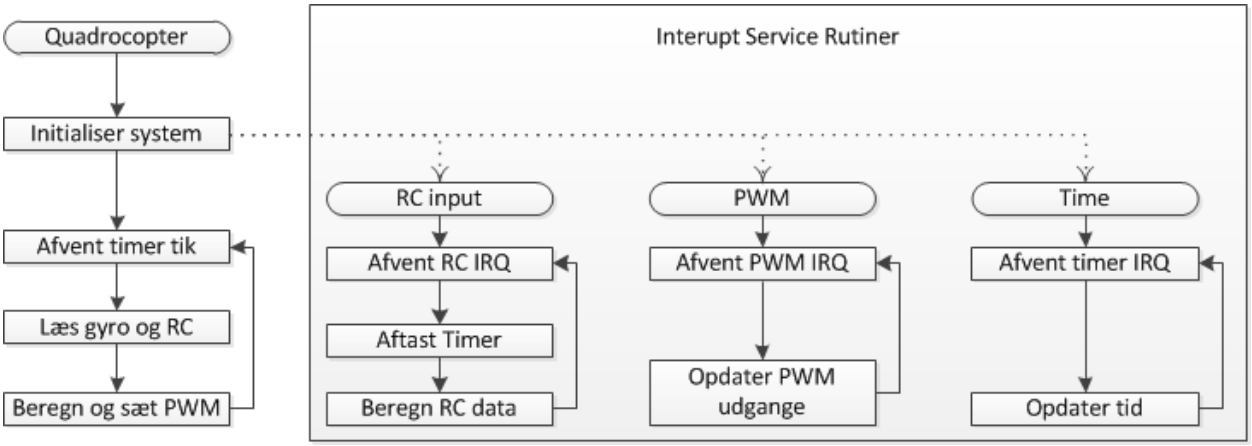
For at bestemme hvordan skeduleringen af disse task skal foregå, måtte vi se på hvilke krav, de skulle overholde udad til (motorstyringen) og hvor hårdt det blev beslastet ude fra (RC). Der findes ikke én rigtig måde at kommunikere med motorer og servoer til RC produkter, men de fleste producenter er enige om, at en form for PWM styring, hvor en puls med en varighed på 1-2.5 ms skal angive minimal til maximal udslag/gas for en motor. Disse pulser skal så gentages ofte, og her er de fleste producenter også enige om, at man skal kunne opdatere den med 10 ms intervaller. Nogle producenter kan også håndtere helt med til 3 ms, men med 10 ms burde vi være kombatible med de fleste.

Når alligevel vi skal opdatere motorerne med 10 ms intervaller, giver det ikke mening at prøve at beregne, hvilken værdi de skal opdateres med oftere (det kunne fint give mening at beregne oftere, men dette projekt handler ikke om reguleringsteknikker, og vi har valgt en simpel regulerings algoritme). Det betyder, at kommunikation med perifere enheder og beregninger af motor reguleringen skal være klar, inden motorerne skal opdateres – altså hver 10 ms.

Efter at have rådført os med erfarne RC quadrokopterpiloter fandt vi frem til, at vi skulle kunne opdatere motorerne med en opløsning på 10 bit, svarende til ca 1 us nøjagtighed. Den valgte microcontroller har godt nok en 16 bit hardware PWM kontroller, men dette

giver kun en opløsning på ca 150 us. I stedet vælges det at implementere en PWM kontroller i software, og dermed får vi et selvstændig task til dette formål.

Da der er valgt bare-metal system, er der ikke noget OS, der kan tilbyde tidsinformation, og der er ikke drivere til on die perifere enheder. Dette medfører, at det er nødvendigt at implementere en timer til at holde et internt ur opdateret, samt implementering af drivere til RC og PWM.



Figur 10 Simplificeret Quadrocopter Program flow.

Figur 10 viser et konkret eksempel på et system, hvor der udefra kommer et input (RC interrupt), intern behandling af input (Quadrokopter), digitale udgange (PWM) og et tidstagnings system (Time). Figur 10 indikerer også, at der er et task (Quadrokopter), samt drivere til håndtering af "RC input", generering af "PWM" samt et "Time" interrupt, der holder styr på tiden.

6 TDD for indlejrede systemer

På en PC er det meget let at komme i gang med en unittest og grundprincipperne kan let forklares, se (Appendiks 1). På en bare-metal indlejrede platform er problemet, at man ikke har et operativsystem med driver, der afkobler hardware, og at udviklingsmiljøet ikke nødvendigvis kan simulere hardwaren perfekt. Det betyder, at man som udvikler selv skal skrive sine driver til det hardware, man ønsker at bruge - og dermed skal man også selv skrive unittests til det kode, der skal tilgå dette hardware.

I dette afsnit vil vi give eksempler på hvordan man kan gøre hardwarenært kode testbart, og vise, at man ved hjælp af abstraktionslag og test stubbe kan teste alt det kode man selv skriver. Afsnittet er delt ind i nogle konkrete cases som hver især repræsenterer en klasse af problemstillinger. Følgende klasser af problemer kan identificeres i quadrokopter projektet:

- Test stub til repræsentation af hardware
- Test af interrupt rutiner
- Test af kommunikationsveje og API
- Test af business logik
- Unitest til tidligere og tredjeparts kode
- Unittest som dokumentation af tidligere og tredjeparts kode

6.1 Digital udgang

Et klassisk "hello world" projekt for indlejret programmering er at få en lysdiode til at tænde og slukke. Programmeringsmæssigt er det en meget simpel opgave, problemet er at skrive testkode, der kan se om lysdioden faktisk tænder og slukker som forventet.

For at vise dette, må vi starte lidt længere tilbage med et af grundprincipperne i TDD; at man ikke tester andres kode [11]. Dette er der flere årsager til.

- Hvis man bruger tredjeparts kode, hvad enten det er direkte som kildekode eller via et bibliotek, må man gå ud fra at det er testet.
- Hvis ikke det er udviklet via TDD, vil det alligevel blive svært at opnå alle de fordele TDD giver (evolutionært design) (Se afsnittet 8.1)
- Hvis tredjepartskoden kommer fra et bibliotek uden kildekode, vil man kun kunne lave blackboxtest.

For et PC program betyder det eksempelvis, at hvis man bruger `printf()` i sin kode, behøver man ikke teste at funktionen rent faktisk virker, da man må gå ud fra at leverandøren af dette kode har testet det inden, det er blevet frigivet. På den anden side, hvis man vil teste, at ens egen kode kalder `printf()` med de rigtige parametre, bliver man nødt til at lave en test stub af `printf()`. Ligeledes vil processorproducenterne også tilbyde en form for abstraktionslag som gør det muligt at skrive kode, der tilgår hardware i deres processorer.

Et simpelt hardware abstraktionslag (HAL) og testspion kunne implementeres via macro i Listing 2. I linie 1 bruges en define TEST, som kun er defineret i unittest projektet, og altså

ikke defineret i produktionskoden. I produktionskoden (linie 6 og 7) bliver der defineret et simpelt HAL (`LED_ON()` og `LED_OFF()`) til at tænde og slukke for et output på processoren. Og i unittest koden defineres der en testspion (i linie 2 til 4) der kan "sladre" om api'et til at styre lysdioden er blevet kaldt.

```
1: #ifdef TEST
2: bool ledStatus;
3: #  define LED_ON ( ledStatus = true );
4: #  define LED_OFF ( ledStatus = false );
5: #else
6: #  define LED_ON ( PORTA |= 1 )
7: #  define LED_OFF ( PORTA &= ~1 )
8: #end
```

Listing 2 Viser et eksempel på et simpelt HAL implementeret via macroer, samt en afstubning af dette HAL for at gøre dette testbart.

Eksemplet i Listing 2 kan dog ikke teste for alt

1. Er led monteret på pin 0? Eller er den overhovedet monteret?
2. Er led aktiv høj eller aktiv lav?
3. Er der andre der bruger port a.0?

Om led er monteret korrekt kan ikke eftervises fra software testkoden, uanset hvor smart man laver sit HAL (man vil kunne med selvtest i target, men det er uden for scope for dette projekt), men vil kunne ses når koden bliver uploadet til target. Dette svarer til en N4 test (se afsnit 4.3). En sådan test vil godt nok kunne eftervise fejl i softwaren, men det vigtigste disse test på N4 vil kunne bidrage med, er at vise om softwareudviklerne har forstået, hvordan hardwaren er lagt ud.

Spørgsmålet om den port lysdioden er monteret på bliver brugt andre steder i koden er mere konkret for softwaren. Reelt set vil det være muligt at implementere et nyt interface til eksempelvis en buzzer, der bruger samme port pin som lysdioden, og at ingen tests vil kunne eftervise at dette er sket.

En mulig løsning på ovenstående problem kunne være at bruge en teststub til at spionere på hardwaren. På samme måde som det blev beskrevet omkring `printf()` problematikken, vil det også være muligt at lave en teststub omkring `PORTA` som vist i Listing 3. I unittestkoden bliver definitionen af `PORTA` ændret til at pege på en global variabel (linje 2) i stedet for at bruge producentens definition af `PORTA` (som peger på en bestemt registeradresse). På denne måde vil produktionskode og unittestkoden kunne bruge det samme hardware abstraktionslag til at tilgå og ændre tilstanden af lysdioden (linje 7 og 8).

```
1: #ifdef TEST
2:    uint8_t PORTA;
3: #else
4:   /* Inkluder producentens implementation... */
5: #endif
6:
7: #define LED_ON ( PORTA |= 1 )
8: #define LED_OFF ( PORTA &= ~1 )
```

Listing 3 En teststub implementeret på register niveau

For at lave disse stubbe, er det vigtigt at teststubben overholder samme interface som det, den skal stubbe af [18]. For stubben i Listing 3 betyder det, at variablen `PORTA` (i linie 2) skal være af samme type (eller kompatibel med) som det register, den repræsenterer. Da vi bruger en 8 bit microcontroller vil alle registre, med få undtagelser, kunne repræsenteres som 8 bits datastørrelser. Havde der ikke fandtes en passende datastørrelse, der kunne repræsentere hardwareregistret i testmiljøet, ville det i dette tilfælde ikke have været noget problem at lade repræsentationen i testmiljøet bruge en større datatype, eksempelvis 32 bits int, da de brugte operationer vil give samme resultater for de to datastørrelser. Når man vælger datatype til abstraktionslaget er det derfor vigtigt at overveje:

- Endianness for testsystem og target system
- Datastørrelser for registre/variabler
- Radix repræsentation

Vi har valgt at bruge googletest framework (gtest) til vores projekt. Vi har ikke undersøgt, hvordan dette er at bruge i forhold til alternativerne. Hovedargumenterne for at vælge gtest er, at det understøtter det sprog, vi har valgt at udvikle vores styring i (C++), og det integrerer godt med det udviklingsmiljø vi har valgt (Eclipse). Listing 4 viser vores unittests der tester om en digital udgang bliver sat rigtigt.

```
TEST(DigitalInput, isOnOff) {
     DigitalInput *di = new DigitalInput(PINA, DDRA, 0);

     PINA = 0;
     EXPECT_EQ(false, di->isOn());
     EXPECT_EQ(true, di->isOff());

     PINA = 1;
     EXPECT_EQ(true, di->isOn());
     EXPECT_EQ(false, di->isOff());

     PINA = 0xfe;
     EXPECT_EQ(false, di->isOn());
     EXPECT_EQ(true, di->isOff());
```

```
    delete di;
}
```

Listing 4 En test af vores implementationen af en driver til et digital input.

6.2 Interrupts i indlejrede systemer

I indlejrede systemer bruger man ofte interrupts til at afbryde den aktuelle kontekst, pause denne og skifte til en håndtering af det, der forårsagede afbrydelsen. Når processoren er færdig med at håndtere afbrydelsen, fortsætter den med det, den var i gang med inden afbrydelsen. Interrupts kan forekomme som følger af flere forskellige typer hændelser, niveau skift for en digital indgang, en timer der løber ud eller en datapakke, der bliver modtaget. Ydermere kan hyppigheden og forekomsten af interupts deles ind i tre grupper [19]:

- **Periodisk**, interrupt udløses i faste veldefinerede intervaller.
- **Aperiodisk**, interrupt kommer på helt tilfældige tidspunkter (disse er ikke beskrevet ydderligere i dette thesis).
- **Sporadisk**, interrupt udløses i varierende intervaller, hvor den minimale og den maksimale afstand mellem interrupt er kendt.

TDD N1 tester funktioner enkeltvis, det vil sige ved N1 vil der ikke være forskel på, hvordan de tre forskellige typer af interrupts bliver testet. Funktionel test af interrupt funktioner, også kaldet interrupt service routine (ISR), kan gøres på samme måde som enhver anden funktion. Den eneste forskel på interrupt funktioner og almindelige funktioner er, at en interrupt funktion skal defineres på en måde, så kompileren kan oversætte den til noget der kan blive kaldt ved et interrupt. Hvordan dette er implementeret i hardwaren, kan variere alt efter processorarkitekturen.

```
ISR(TIMER1_COMPA_vect) {
    ...
}
```

Listing 5 viser hvordan en interrupt servise rutine for en timer 1 compare match skal implementeres via det HAL chipproducenter har leveret.

I api'et leveret af Atmel til vores microcontroller, skal en ISR erklæres som vist i Listing 5 hvor `ISR()` er defineret som vist i Listing 6. Dette vil erklære en funktion med et par ekstra attributter, så kompileren kan genkende at funktionen er en ISR, og dermed skal håndteres specielt i target build.

```
#define ISR(vector, ...)            \
    extern "C" void vector (void) __attribute__
((signal,__INTR_ATTRS)) __VA_ARGS__; \
    void vector (void)
```

Listing 6 viser den macro der bruges for at oversætte ISR til noget kompileren kan oversætte.

For at kunne kompilere en ISR til sit unittest framework, opretter vi en ny macro definition i Listing 7, som implementerer samme interface som givet i Listing 6.

```
#define ISR(vector, ...) void vector(void)
```

Listing 7 Viser hvordan vil "kaprer" chipproducentens HAL på en måde der gør det muligt for os at test ISR.

Herved bliver det muligt at teste funktionalitet i en vilkårlig interrupt funktion via almindeligt test, som vist i Listing 8. Et konkret eksempel på brugen af dette følger i næste afsnit.

6.2.1 Periodisk interrupt (ur)

Periodiske interrupts er interrupts, der kommer med et helt fast interval. Da klok frekvensen for systemer vil drive over tid, vil det også sige, at periodiske interrupts er noget der forekommer synkroniceret med microcontrollerens egen clock frequense, eksempelvis et interrupt fra en timer. Mange indlejrede systemer har brug for en form for internt ur. Quadrokopteren bruger det interne ur til at starte kontrol algoritmen, samt at PID regulatoren bruger tidsinformation til integral og differential beregninger. Et system kan have flere af denne type interrupts med forskellige faste intervaller.

Figur 11 Eksempel på periodisk interrupt

Da tidsintervallet for hvert interrupt er velkendte, vil det være muligt at lave et Gantt diagram som vist i Figur 11, der viser hvornår et interrupt er aktiv, i forhold til andre periodiske interrupts. Ved kodeanalyse vil det være muligt at fastlægge eksekveringstiden (se afsnittet 7.3) for hvert enkelt interrupt, og det bliver synligt hvor meget tid, der bliver brugt.

Mange systemer bruger en HW timer til at gener et periodisk interrupt. I eksemplet Quadrokopter skal PID regulatoren bruge en tidsangivelse for at kunne beregne ΔT, til integral- og differential delen. Da det ikke er nødvendigt med en absolut tid startes uret, når der tændes for systemet og tælles op for hver 10 ms, samt at reguleringsalgoritmen skal aktiveres for hver 10ms, vist i Listing 8.

```
ISR(TIMER1_COMPA_vect) { // 10mS interrupt
     ++time10Ms;
     sei(); // re-enable IRQ
     OCR1A += 20000; // Timer uses 500 ns ticks
}
```

Listing 8 Timer 1 ISR genarmerer sig selv til at blive kaldt 10 ms senere.

Da den valgte microcontroller kun har ét niveau af interrupt, vil et igangværende interrupt blokere for alle andre interrupts, der måtte forekomme. For at minimere den tid der er blokeret for andre interrupts, er det valgt at bruge re-entrant interrupt. Ved normal afvikling vil microcontrolleren automatisk aktivere interrupt efter endt interrupt afvikling. Ved at lave Listing 9 macro til testkode, bliver det muligt at teste om interrupt aktiveres under afvikling af funktionen.

```
#define sei() irqEnable=true
```

Listing 9 Aktiver IRQ test

Testen til ovenstående er implementeret ses i Listing 10

```
TEST(Timer, Interrupt) {
     irqEnable = false;
     OCR1A = 0;
     time10Ms =0;

     TIMER1_COMPA_vect();
     EXPECT_EQ(1, time10Ms);
     EXPECT_EQ(true, irqEnable);
     EXPECT_EQ(20000, OCR1A);

     TIMER1_COMPA_vect();
     EXPECT_EQ(2, time10Ms);
     EXPECT_EQ(40000, OCR1A);
}
```

Listing 10 Timer 1 ISR test

Ved at konfigurere microcontrolleren til ikke automatisk at genaktivere interrupts, og ved at anvende re-entrante interrupts, opnås der tre prioriteter i kodeafvikling:

- Højeste prioritet: Når et interrupt bliver hævet, vil dets kode blive afviklet, og samtidig vil ingen andre interrupts kunne afbryde det.
- Mellemste prioritet: Når interrupt service rutinen har udført sin kritiske del af koden, genaktiverer det igen interrupts, hvilke vil tillade det at bliver afbrudt at et andet interrupt, men det fortsætter selv med at udføre sin egen kode. Kontekst bliver først givet tilbage til det forrige afviklede kode, når interrupt service rutine er færdig.
- Laveste prioritet: Alt kode, der ikke er interrupt service rutiner eller bliver kaldt fra en, har i princippet ingen prioritet. Dette kode vil kun køre når alle interrupt service rutiner er færdige med at afvikle deres kode.

Ved unittest vil der kun blive afviklet et modul af gangen, altså vil der ikke blive testet for påvirkninger fra funktioner, der kører med højere prioritet. I quadrokopter tilfældet, blev der fundet en overskrivning, der var forårsaget af at to uafhængige funktioner tilgik den samme hardware, hvor der er krav om, at softwaren aflæser to 8 bit registre uden

afbrydelse. Denne fejl blev først fundet ved integrationstesten i endelig hardware. Denne fejl blev først opdaget i en N5 test (se afsnit 8.3).

6.2.2 Sporadisk interrupt (RC / PWM)

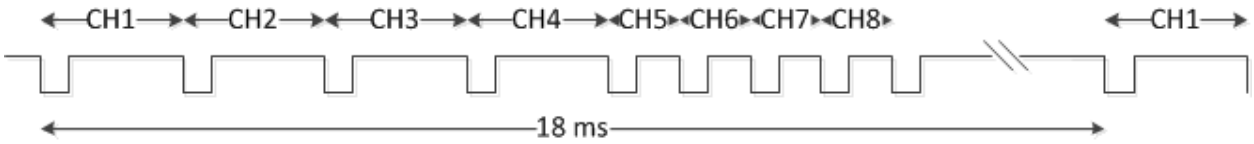
Sporadiske interrupts illustreret på Figur 12 er defineret som havende en minimum og maksimal frekvens, de kan forekomme med. Når man skal regne på deres indflydelse, er det derfor nødvendigt at anvende det værste tænkelige af de to. For beregning af udnyttelsesgraden af microcontrolleren, anvendes maks. frekvensen for de enkelte interrupts.

Figur 12 Sporadisk interrupt der afbryder main tråden med variabel interval.

Ved tidsanalyse skal der tages højde for at disse interrupts kan komme på et vilkårligt tidspunkt i forhold til den definerede maksimale / minimale frekvens. Dette betyder, at det ikke er muligt at sikre at to interrupts ikke kan komme på samme tid, samt at man skal kunne håndtere at et interrupt enten venter til et andet er kørt færdigt, eller til et interrupt tillader at andre interrupts kan afbryde (re-entrant interrupt).

Med hensyn til test, kan der laves testcases på samme måde som til periodiske interrupts (afsnit 6.2.1), da testen udelukkende fokuserer på funktionaliteten af de enkelte moduler.

Quadrokopteren har bl.a. et aperiodisk interrupt fra fjernbetjeningen.



Figur 13 RC modtager udgangs signal

Quadrokopteren skal bruge 4 kanaler fra RC modtageren. Udslag for hver kanal bliver repræsenteret som en tid, mellem 1 og 2 ms svarende til 0 til 100% udslag. Figur 13 viser et eksempel på, hvordan signalet fra RC modtageren kan se ud. Data for kanalerne bliver sendt som CPPM (Combined Pulse Position Modulation), hvilket vil sige adskilt af en nedadgående flanke. Således vil tiden for kanal 1 være tiden mellem den første og den anden nedadgående flanke. Tiden for kanal 2 vil være tiden mellem den anden og tredje nedadgående flanke. Hele dette pulstog bliver gentaget for hver 18 ms. For at vide værdien for det, der bliver repræsenteret for hver kanal, er det derfor nødvendigt at måle tiderne mellem de nedadgående flanker.

Ved implementering af RC funktionen, er det tiden fra sidste interrupt, der er interessant. Derfor kan dette testes ved at kalde interruptfunktionen to gange, og stille uret mellem de to gange. Listing 11 viser en implementering af testen.

```
TEST(irqTest, Int0Int1) {
     TCNT1 = 0;
     INT0_vect();
     TCNT1 = 2000;
     INT0_vect();
     EXPECT_EQ(2000, RcValues->GetChannel(0));
}
```

Listing 11 implementationen af testen til periodisk ISR

I interruptet beregnes tiden som vist i Listing 12

```
uint16_t dTime = time - lastRcReading;
```

Listing 12 Delta tids beregning uden at tage højde for om lastRcReading er større end den sktueller tidd (time).

Vi bruger en 16 bits timer, og hver timer tick repræsenterer 0.5µs. Det vil sige, at timeren løber over hvert 32,768ms. For at sikre at vores kode kan håndtere tider der løber hen over timerens maks, laves der en testcase Listing 13:

```
     TCNT1 = 0xFFFF;
     INT1_vect();
     TCNT1 = 2299;
     INT1_vect();
     EXPECT_EQ(2300, RcValues->GetChannel(0));
```

Listing 13 Timer overløbs test

Dette er et eksempel på, hvordan TDD tilbyder, at man laver test af specielle konditioner, der kan opstå i sofware - i dette eksempel at teste om en bestemt udregning giver det ventede resultat. Her viste testen, at vores første simple implementering giver PASS, så der er ikke grund til at lave speciel håndtering af tilfældet, hvor timeren passerer 0xFFFF og begynder ved 0, og derved blev tiden for forrige tidsmåling et større tal end for den sidste.

6.3 On Die perifer funktionalitet (I²C)

Da microcontrollere ofte er et helt computersystem integreret på ét stykke silicium, betyder det også, at meget af det perifere hardware en normal PC har, vil være integreret på det samme stykke silicium som selve processoren. Dette kaldes on-die hardware. I quadrokopteren bruges eksempelvis en I^2C bus[5] [20] (som er on die) til at kommunikere med vores sensorer (som er off die). Og meget af dette on die hardware vil være i stand til at afvikle deres funktionalitet fuldstændig parallelt med at processoren udfører sine opgaver og uden at bruge nogen form for ressourcer fra processoren.

[5] atmega32 har et on-die two wire interface, som kan konfigureres til at køre som I^2C.

Et digital output er meget simpelt i sin virkemåde; man sætter et bit i et register, hvorefter en pin på microcontrolleren skifter tilstand. Set ud fra testens synsvinkel er det ikke vigtigt, hvor lang tid der går fra bittet er sat til pinden skifter, så det er fint nok, at disse to operationer sker lige så hurtigt efter hinanden, som testen kan afvikles.

For perifer hardware som kommunikation interfaces, heriblandt I^2C, forholder det sig anderledes; i koden sætter man noget data op, man ønsker at få transmitteret. Herefter forventer man, at processoren kan bruge sine ressourcer på noget andet, indtil der indløber et svar på den forespørgsel man tidligere sendte. Fra processoren beder om at få data sendt og indtil svar er modtaget kan der ske mange ting: kommunikationen kan forløbe planmæssigt, der kan gå data tabt eller blive misforstået, der kan opstå fejl i protokol og timing, og meget mere. Fra koden skal man kunne spørge på status for kommunikationsinterfacet, som vil være forskellig alt efter hvor langt i send/modtage processen hardwaren er nået.

NOTE: Vi har valgt kun delvis at implementere I^2C driveren event drevet. Dette skyldes kombinationen af frekvensen af interrupts samt tiden for at foretage selve kontekstskiftet.

Det egentlige problem opstår i og med at selve I^2C kommunikationen ikke er tidskritisk, og dermed ikke skal have videre høj prioritet. Da den valgte microcontroller ikke har interrupt prioritering, betyder det at I^2C interfacet vil få høj prioritet (afsnit 6.2.1), hvis dette implementeres interrupt drevet. Dette vil dermed betyde at det I^2C driveren vil kunne blokere andre interrupts. Dette fænomen/problem er beskrevet nærmere i afsnit 8.3.

Hele dette kommunikationsforløb kan ses som en tilstandsmaskine, som bliver afviklet parallelt og uden indvirkning af processoren, men som kan give forskellige interrupts alt efter hvilken tilstand, den befinder sig i. For at kunne teste denne samtidighed, kræves det også at testen kan udføre flere ting "samtidig".

Listing 14 viser et eksempel på, hvordan man kan teste denne samtidighed. UUT er metoden `busyWait()` (linie 1). `busyWait()` har til ansvar at holde program afviklingen tilbage så længe microcontrolleren er i gang med at sende. Mens microcontrolleren sender, vil status flaget `command.status` være `IIC_BUSY`. I linierne 2-8 vil microcontrolleren stå og vente indtil `command.status` skifter tilstand, eller timeren (timeout i linie 6) render ud. Det letteste stimuli at generere til busyWait testen, er at lade timeren rende ud. I det tilfælde skal tilstanden for I^2C modulet være det samme som efter at have kalde metoden `stop()` (linie 10 og 11).

Men hvis man skal give et stimuli der indikerer at I^2C modulet har modtaget data inden timeren render ud, skal `command.status` skifte tilstand mens `busyWait()` busy-waiter (linierne 2-6). For at teste dette starter vi med at emulere at I^2C modulet sender data (linie 23) i én tråd. Dernæst starter vi én tråd mere, der opdaterer status registret med

samme værdi, som microcontrolleren ville have gjort, hvis den havde modtaget et ack fra slaven. Efter denne test/simulering burde I^2C modulet være i `IIC_IDLE` state (linie 29).

```
1:  void I2C::busyWait() {
2:      uint16_t timeout = 1000;
3:      while ((command.status != IIC_IDLE)
4:              && (command.status != IIC_ERROR)
5:              && (timeout > 0)){
6:          --timeout;
7:          isr();
8:      }
9:
10:     if (command.status != IIC_IDLE)
11:         stop();
12: }
13:
14: void sendThread() {
15:     i2c.writeRegister(0x10, 0x12, 0x13);
16: }
17:
18: void receiveThread() {
19:     TWSR = 0x28; // Emulate slave signaling ACK
20: }
21:
22: TEST(i2cTest, BusyWaitNormal) {
23:     boost::thread send(sendThread);
24:     boost::thread receive(receiveThread);
25:
26:     receive.Join();
27:     send.Join();
28:
29:     EXPECT_EQ(IIC_IDLE, i2cTest.getCommand()->status);
30: }
```

Listing 14 Samtidigheds test

Listing 14 viser hvordan opførelsen af I^2C klassen kan testes, hvor det skal simuleres, at driveren har modtaget et svar.

6.4 Off Die perifer funktionalitet (Gyro)

Nu hvor vi har implementeret en I^2C driver til vores projekt, kan vi begynde at implemente, at processoren skal kommunikerer med en enhed, der ikke er integreret på samme die. For at kunne regulere vores quadrokopter er det nødvendig for reguleringen at vide noget om, hvordan quadrokopteren bevæger sig i luften. Til dette formål vælger vi at anvende en gyro, der kan kommunikere via I^2C bussen.

Helt kort fortalt skal gyromodulet være i stand til at kommunikere med en hardware gyro via I^2C bussen. Modulet skal kunne udføre opsætning af hardwaren, starte nye målinger og konvertere de rå målinger fra hardwaren til noget resten af softwaren skal kunne regne videre med.

Set fra en testsynsvinkel er det trivielt at teste, om data bliver omregnet rigtigt (se afsnit 6.1). Det er derimod mindre indlysende, hvordan det skal kunne testes, om gyromodulet kommunikerer rigtigt med hardwaren via I^2C bussen. For at kunne teste dette får vi brug for en test stub/spion af I^2C interfacet.

Som tidligere nævnt (i digital output eksemplet) når man laver test stubbe, er det vigtigt at de overholder samme interface som det de skal stubbe af. Derfor startes I^2C test stubben med at arve fra den faktiske I^2C driver (Listing 15). På denne måde vil det være muligt, via dependency injection [11] at instantiere gyromodulet med en reference til teststubben, og gyromodulet vil ikke kunne se, at det kommunikerer med en spion i stedet for den rigtige I^2C driver.

```
class I2CStub: public I2C {
    …
}
```

Listing 15 Viser en test stub til I2C klassen, som arver fra den oprindelige klasse som den skal stubbe af.

Her efter skal teststubben tilsidesætte de offentlige dele af det oprindelige interface. I dette tilfælde betyder det, at vi skal skrive nye implementationer af funktionaliteten til at læse og skrive fra I^2C bussen, vist i Listing 16. For at kunne overvåge hvilke data gyromodulet prøver at sende til hardwaren vælger vi at gemme det "sendte" data. Endelig må gyromodulet heller ikke kunne kommunikere med andre enheder på I^2C bussen end selve gyroen. Derfor implementeres der også en sladrehank, der kan fortælle om gyromodulet prøver at snakke med andre enheder.

```
uint8_t count;
data_t data[100];
void writeRegister(
        uint8_t deviceAddress,
        uint8_t reg,
        uint8_t val) {
    if(deviceAddress != GYRO_IIC_ADRESS)
        illegalDeviceAddress = 1;
    this->data[count].reg = reg;
    this->data[count].count = 1;
    this->data[count].data[0] = val;
    ++count;
}
```

Listing 16 en spion der gemme alt data der prøves sendt via dens test stub. Data kan på denne måde sene inspiceres af en test case.

Og for at kunne se hvilket data der er blevet sendt implementeres en funktion, der kan fortælle at dataarrayet indeholder en skrivning til et specifikt register. Denne funktion er ikke en del af den oprindelige I2C klasse, og tjener ikke noget formål i produktionskoden.

```
uint8_t getRegister(uint8_t needle, data_t *data) {
    for(int i=0; i<count; ++i)
    if(data[i].reg == needle) {
        *data = &data[i];
        return true;
    }
    return false;
}
```

Listing 17 Hjælpe funktion til at finde data der tidligere er prøvet sendt via I2C test stubben.

Under initialiseringen af gyromodulet skal der sendes en række parametre til hardwaren, som er beskrevet i databladet for den valgte gyro. I unittesten af initialiseringen af gyro modulet, kan det nu testes om modulet sender det forventede data. Listing 18 viser implementationen af testen af initialiseringen af gyromodulet. Først oprettes der en instans af en I^2C spion (linie 2), som bliver født ind i gyroklassen (kaldet dependency injection), hvor denne instantieres (linie 3). På nuværende tidspunkt burde hele gyromodulet være initialiceret, og dermed bør hele nævnte række af initialiseringsparametre være sendt. I alt burde der være foretaget 5, hverken flere eller færre skrivninger (linje 5) til gyroens I^2C adresse (linje 6). Herefter tester vi, at de 5 registre, vi forventede skrivninger til, også er blevet tilgået, og at de indeholder det forventede data (linje 9 til 19). Af hensyn til den videre kommunikation med gyroen, viste det sig at være vigtigt, at der bliver skrevet til register 2 som det første (linje 22).

```
1:  TEST(GyroTest, Initialization) {
2:      I2CStub i2c;
3:      Gyro g(&i2c);
4:
5:      EXPECT_EQ(5, i2c.count);
6:      EXPECT_EQ(0, i2c.illegalDeviceAddress);
7:
8:      // All registers are accessed
9:      data_t data;
10:     EXPECT_EQ(true, i2c.getRegister(CTRL_REG1, &data));
11:     EXPECT_EG(0x0f, data.data[0]);
12:     EXPECT_EQ(true, i2c.getRegister(CTRL_REG2, &data));
13:     EXPECT_EG(0x09, data.data[0]);
14:     EXPECT_EQ(true, i2c.getRegister(CTRL_REG3, &data));
15:     EXPECT_EG(0x88, data.data[0]);
16:     EXPECT_EQ(true, i2c.getRegister(CTRL_REG4, &data));
17:     EXPECT_EG(0x80, data.data[0]);
18:     EXPECT_EQ(true, i2c.getRegister(CTRL_REG5, &data));
```

```
19:     EXPECT_EG(0x12, data.data[0]);
20:
21:     // Register 2 must be the first register written
22:     EXPECT_EQ(CTRL_REG2, i2c.data[0].reg);
23: }
```

Listing 18 Gyro test

7 Ting der ikke kan testes via TDD, men…

Indtil nu har vi beskrevet hvordan vi har brugt TDD arbejdsgangen til at drive udviklingen af indlejrede systemer. Men TDD i sig selv kan ikke dække alle af de situationer, man skal igennem i et indlejret udviklingsprojekt. I dette afsnit vil vi se på nogle problemstillinger, der generelt kunne være relevante for indlejret udvikling, hvor TDD i sig selv ikke kan hjælpe. Det vil sige, at selv om man har udviklet sit indlejrede projekt via en traditionel udviklingsmetode, vil man kunne indføre et unittest framework, og udføre de testscenarier vi beskriver i dette afsnit. Da vi har brugt TDD, har vi allerede været tvunget igennem processen med at få et sådant unittest framework sat op, og vi vil derfor fortsætte med at bruge samme framework.

Med dette mener vi, at hvis man ikke har og bruger et unittest framework til at teste sin kode, vil det være en anselig ekstra omkostning (tidsmæssigt) at sætte dette op bare for at kunne få én af de egenskaber, vi nævner i dette afsnit. På den anden side, hvis man allerede har valgt TDD, har man være tvunget til at bruge tiden på at sætte dette op, og det må formodes, at alle udviklere bruger dette framework, hvorved prisen/tiden for at implementere en af de egenskaber, vi nævner i dette afsnit, vil blive mindre.

7.1 Er det seneste kodereview stadig gyldigt?

Nogle ting kan ikke testes via normal TDD og unittests. Eksempelvis viste det sig, at der i quadrokopterkoden kunne opstå en konflikt mellem to interrupt, hvis de blev hævet med en hel specifik tid i forhold til hinanden (se afsnit 8.3). Ved at lave en manuel inspektion af koden var det muligt at finde den pågældende fejl, men hvordan sikrer vi, at det ikke sker igen?

Ifølge functional safety metoder [21], kan et review udført af en ”kompetent person” [21] også ses som en gyldig test af koden. Det vil sige, man kan lave en manuel inspektion af koden, for derefter at afgøre om denne er OK eller ej. Hvis koden kan godkendes ved en manuel review process, indføres en unittest der sikrer, at koden ikke er ændret siden review. På denne måde fungerer unittesten som “vagthund” over kode fragmenter, der ikke umiddelbart kan testes, men som er kritiske for systemet.

En simpel metode til at sikre at der ikke er ændret i filer, vil være at lade unittest casen lave en checksumsberegning på den eller de filer, der er berørt af den manuelle inspektion og sammenligne med værdien fundet efter review. Ved brug af denne metode, vil det kræve at

der er et veldefineret interface til koden der ligger i de berørte filer, for at sikre størst mulig afkobling fra den resterende kode.

For at teste om ovenstående test kan udføres, har vi valgt at implementere en test, der kan overvåge revisionen for en fil. I quadrokopterprojektet har vi set et problem omkring "samtidig" tilgang til 16 bit timer registre (se afsnit 8.3). Vi ønsker derfor at lave en test, for at overvåge om revisionen af vores interrupt service rutiner for timeren er blevet ændret.

Først skal der implementeres et script, der kaldes som "pre build" step, der autogenerer en header fil, der indeholder en checksum på de enkelte filer vist i Listing 19.

```
// checksums.h
...
#define MD5_irq_cpp 0e0c2180dfd784dd84423b00af86e2fc
...
```

Listing 19 Et udklip fra en automatisk genereret header fil der indeholder en checksum for alle filer i projektet.

Alle funktioner der tilgår de omtalte registre findes i files irq.cpp. I Listing 20 ses implementationen af en test, der overvåger om denne fil er blevet ændret.

```
#include "checksums.h

TEST(irq_cpp, revisions) {
    expect(0e0c2180dfd784dd84423b00af86e2fc, MD5_irq_cpp);
}
```

Listing 20 Implementation af en test der tester om en enkelt fil har en bestemt og velkendt checksum (hardcoded for at gøre det MIDRE attraktiv at rette)

Denne procedure vil kræve at testcasen rettes hver gang, der er lavet en ændring, som er godkendt i review. Om testcasen så er valid i forhold til det tidligere udførte review er en helt anden problem stilling (evt. ved at reviewe testcases eller ikke give udviklere skrive adgang til denne fil) dette er udenfor emnet i denne afhandling.

7.2 Simulering af konsekvenser af fejl

I afsnit 8.3 beskriver vi en fejl, der blev fundet i N5 og giver også nogle løsningsforslag til det konkrete problem. I vores situation er følgen af problemet, at den næste handling der skal udføres ikke bliver sat, hvilket medfører at hardwaren først aktiveres interrupt når timeren har talt hele vejen rundt. Det er ikke noget problem at skrive en test, der simulerer denne hardware fejl/begrænsning.

1. Udfør koden, der skal sætte timer registret op
2. Når det kode er udført ændres test repræsentationen af timer registret

Som koden er implementeret, er der ikke noget i softwaren, der kan registrere, at det er gået galt. Det betyder også, at selv om vi implementerer ovennævnte test, vil den ikke fejle, til trods for at det tydeligvis er en ikke ønsket opførsel ved koden, hvis ikke registre bliver opdateret, som vi forventer det.

På den ene side kunne det nu føles valid at begynde at implementere kode, så denne "falske positive" test vil kunne bliver "sand negativ". Men det er ikke i henhold til TDD ånden (skriv kun ny kode, hvis der er en test der fejler). Problemet ved at begynde at skrive kode nu ville være, at hvis man blev distraheret fra sit arbejde (evt. en telefon der ringede), ville kunne glemme, hvad man var i gang med, og dermed ville man risikere at efterlade koden i en ikke virkende tilstand, med en falsk positiv test og noget kode der ikke er testet. Løsningen vil derfor i stedet være at tilføre en test mere, indtil testen fejler. Som vi beskriver i afsnit 8.3, kommer vi med to løsningsforslag (at skrive noget kode, der kan overvåge timerregistret eller at undgå situationen ved at slå interrupts fra), og alt efter hvilken løsning man vælger at implementere, vil man derfor først skulle implementere en test, der vil kunne hjælpe koden i den retning.

En af grundene til at der ikke er noget i koden, der kan registrere dette, kunne skyldes at koden er implementeret via TDD, og dermed har vi kun implementeret funktionalitet, som vi har set har været nødvendig for projektet. På den anden side er der heller ikke noget der garanterer, at vi ville have tænkt over denne begrænsning i hardwaren, hvis vi have valgt ikke at implementere via TDD. Når man skriver de initielle testcases, vil det være ud fra de kendte krav til systemet, hvilket betyder at testen ikke er bedre end designeren er til at stille krav. Hvis de initielle krav ikke er fyldestgørende, vil testen ligeledes ikke være tilstrækkelig.

7.3 Test af skedulering

Når man snakker indlejret udvikling betyder dette ofte også en mindre microcontroller med begrænsede ressourcer og regnekraft. Derfor er det vigtigt at vide, om der er plads (ROM) nok i target til at gemme hele ens program, om der kan forekomme stackoverflow (RAM), eller om programmet overhovedet kan skeduleres (timing). Det vil derfor også være en fordel, hvis det kunne være muligt at teste skedulerbarhed på samme måde, som man kan teste om en interrupt rutiner udfører de tiltænkte operationer.

Med hensyn til skedulerbarhed findes der forskellige værktøjer til at beregne længste eksekveringstid (WCET) [22] for kode til forskellige processorer. Hvis man opstiller en skedulerigsbudget, vil det med disse værktøjer være muligt at undersøge om koden overordnet set overholder de opstillede krav til eksekveringstider. Disse typer værktøjer udfører en statisk analyse på selve maskinkoden, altså det kompilerede kode, og vil dermed også være afhængig af, om projektet bliver bygget med debug symboler, eller hvilke optimeringsgrad der er valgt.

Det er dog også vigtigt at bemærke, at en undersøgelse af eksekveringstider i sig selv ikke er det samme som at hele systemet altid vil kunne skeduleres, hvis ikke foranalysen og skedulerings budgettet er fyldestgørende.

For at undersøge om en automatisk WCET analyse og check vil kunne bruges i praksis og i samspil med TDD, har vi prøvet at implementere en testcase, der overvåger WCET for nogle funktioner. I dette eksempel ønsker vi at overvåge `I2C::isr()`.

Først har vi skrevet et bashscript der bliver kaldt som en pre-compile handling i vores udviklingsmiljø. Scriptet bruger analyseværktøjet Bound-T [23] til at analysere alle funktion og metoder i projektet og bruger resultaterne fra Bound-T til at generere en headerfil, vist i Listing 21, hvor hver WCET er givet som en define. WCET bliver målt i antal klok cycler.

```
#ifndef __AUTO_GEMERATED_WCET_ANALYSIS_H__
#define __AUTO_GEMERATED_WCET_ANALYSIS_H__

…
#define C_I2C__isr 238
#define C_TIMER1_COMPA_vect 82
#define C_TIMER1_COMPB_vect 118
…

#endif // __AUTO_GEMERATED_WCET_ANALYSIS_H__
```

Listing 21 Et udklip af en automatisk genereret header fil, der indeholder det faktiske WCET for samtlige funktioner i projektet.

Resultatet fra Boud-T er WCET for de enkelte funktioner, men dog uden prisen for at funktionen kan blive afbrudt af andre interrupts. Det vil sige, at før WCET analysen kan bruges til noget rigtigt, er vi nødt til at analysere hvor mange gange og af hvilke interrupts hver enkelt funktion kan risikere at blive afbrudt (se afsnit 6.2). Da dette emne allerede er ganske fjernt beslægtet med TDD, har vi valgt ikke at udføre en komplet analyse, men blot ganske arbitrært valgt at `timer1a` og `timer1b` i værste fald hver ville kunne afbryde `I2C::isr()` to gange inden `I2C::isr()` når at afslutte. Disse reelle priser skriver vi i en header fil, som vist Listing 22.

```
#define M_I2C_ISR (C_I2C__isr                          \
                   + 2*C_TIMER1_COMPA_vect     \
                   + 2*C_TIMER1_COMPB_vect)
```

Listing 22 Et eksempel på hvordan vi har defineret den maksimale cost (i klokcycler) for en funktion. Her antages det, at I2C isr i værste fald kan blive afbrudt to gang af timer 1 compare match a og to gange af timer 1 compare match b.

Den sidste del af skeduleringsbudgettet er at bestemme, hvordan ressourcerne skal fordeles på de enkelte funktioner og metoder. Lige så arbitrært som før vælges det at `I2C_ISR` ikke må tage mere end 350 clock cycler, som vist i Listing 23.

```
#define D_I2C_ISR 350
```

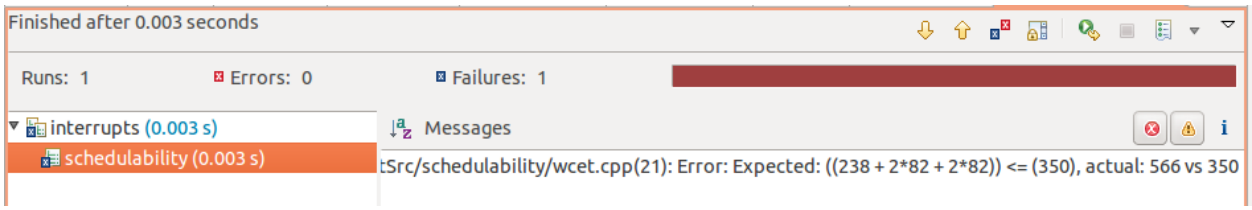
Listing 23 Viser vores skedulerings budget.

De ovenstående forberedelser leder frem til, at vi nu kan skrive en testcase, vist i Listing 24, som køres på lige fod med de unittest, der opstår som følge af TDD processen.

```
TEST(schedulability, interrupts) {
     EXPECT_LE(M_I2C_ISR, D_I2C_ISR);
}
```

Listing 24 WECT testcase

Endelig kan resultatet af skeduleringsanalysen vises som et testresultat, vist i Figur 14, på lige fod med de andre unittests.



Figur 14 Skeduleringstest hvor det ses, at vores projekt muligvis ikke kan skeduleres i øjeblikket, da I2C::isr kan tage op til 566 klokcycler, men der er kun afsat 350 klokcycler til denne funktion.

8 Sideeffekter ved TDD

Om TDD i sig selv er brugbart er en ting. Men at have 100% test af sin kode giver en række sideeffekter, som kan være fordelagtigt. I dette afsnit vil vi nævne nogle af de situationer vi har oplevet, som ikke direkte stammer fra TDD, men som er enten afledt deraf eller relateret dertil.

8.1 Legacy kode (PID)

TDD omhandler kun, hvordan man skriver ny kode (se afsnit 4), ikke hvordan man integrerer med eksisterende kode. Men når man starter et nyt projekt, vil man ofte have noget eksisterende kode med dele af den funktionalitet, man skal bruge, eller det er muligt, at man skal bruge noget tredjeparts kode, som oprindeligt ikke vil være udviklet via TDD. Dette betyder, at hvis man skulle følge en ren TDD tankegang/arbejdsgang, skulle man først forstå detaljerne omkring, hvordan de dele man skal bruge af denne kode virker, og derefter kan man begynde at implementere det igen. Alternativt skal man se dette kode som værende noget blackbox tredjepart kode, og derfor bruge koden uden at teste eller forstå det (se Appendiks 1).

På den anden side kunne man også have noget kode, som man forventer at kunne bruge, men som man skal have skabt sig en forståelse for først. Her vil man med fordel kunne

bruge unittests til at skabe sig overblik over funktionalitet af det kopierede kode og til at sikre sig, at koden opfører sig som forventet.

I quadrokopter projektet skulle vi bruge en PID regulator. For at afprøve hvordan vi kunne integrere legacy kode i vores nyligt udviklede projekt, valgte vi at finde en PID regulering på nettet. En hurtig søgning på Google gav mange gode bud på en løsning. En løsning blev valgt og koden blev kopieret ind i projektet. Herefter startede vi med at læse koden og der blev skrevet en unittest, der kunne eftervise, om vi havde forstået det stykke kode, vi lige havde læst. Denne proces gav en dyb forståelse af den kopierede kode samt sikrede, at vi i quadrokopterprojektet bruger koden korrekt.

8.2 Eksperimenter i det ukendte - Trimning af parametre

Efter implementering af de basale funktioner via TDD har der opstået en stabil platform, hvor TDD garanterer for at funktionaliteten er som forventet for alle implementerede moduler. Vi havde forventet, at den samlede integration ville afsløre graverende fejl, da vi ikke havde testet koden løbende i target, men dette var ikke tilfældet. Den initielle implementering brugte en ældre analog fjernbetjening, hvorved det analoge støjbidrag langt oversteg de afvigelser vi forventede som resultat af samtidighed mellem interrupts. Der var altså ingen "synlige" fejl ved første integration, hvilket resulterede i en problemfri første flyvning med quadrokopteren. Efter et par ture hvor PID konstanter blev trimmet var systemet stabilt.

Efterfølgende er RC sender og modtager udskiftet til nyere digital model, hvorved det blev muligt at se en fejl i prioriteter i koden. I^2C brugte interrupt, hvilket blokerede for aftastning af RC samt opdatering af PWM. Denne fejl var dog ikke stor nok til at give en væsentlig påvirkning af det samlede systems opførsel, hvorfor den først blev opdaget ved eksperimenter med RC af tastning.

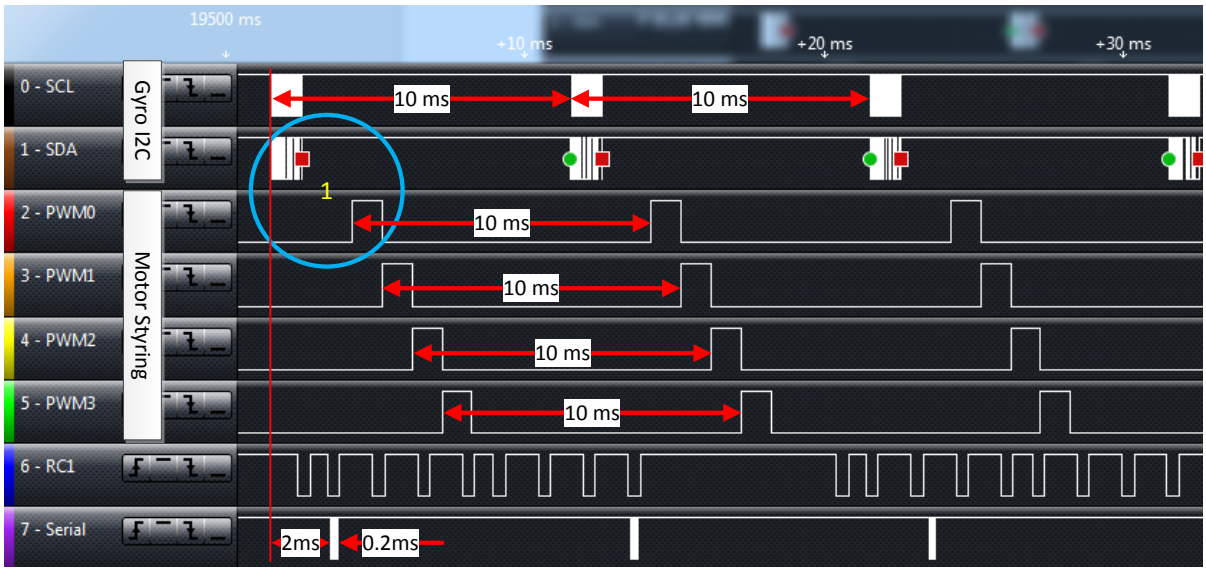
8.3 Integrationstest (N5)

Integrationstesten udføres i en opstilling, der set fra programmets synspunkt er identisk med det endelige miljø programmet skal operere i. Dette kan udføres som en test, hvor der er lavet stubbe til hardware (se afsnit 6.4), eller testen udføres på endelig hardware, hvis denne er tilgængelig.

Det er her, det bliver synligt om skedulerbarhedsberegningerne er korrekte, og hvorvidt de enkelte moduler kan fungere sammen i samme hardware samt at deres interface er korrekt forstået.

For quadrokopteren er det her alle moduler fra Figur 6 skal bringes til at fungere på samme tid, samt lave funktionstest af det samlede system. Afprøvning af det samlede system resulterede i en flyvende quadrokopter.

Vi ser et flyvende system, men hvad er vores ressourceforbrug i processoren? Ud fra programmet ved vi, at reguleringsalgoritmen starter med at læse fra gyro. Faktisk kan gyro kommunikation bruges til at indikere starten på reguleringen. For at have mulighed for at inspicere koden, er det valgt at relevante parametre fra reguleringen sendes ud på en seriel port efter hvert gennemløb af reguleringsalgoritmen. Ved at måle på denne kommunikation kan vi få en indikation af, hvornår beregningerne er færdige, samt at det er muligt at måle hvor lang tid det tager at sende vores debug information. Dette vil kunne give en indikation om belastning af processoren.



Figur 15 Plot I/O pinde

Figur 15 viser input / output aktiviteten til henholdsvis gyro, motorstyring, input fra RC samt test output fra reguleringsalgoritmen. Designet er lavet til, at der skal reguleres for hver 10 ms. Figur 15 viser, at gyroen aftastes med 10 ms interval, hvilket kan betyde at forståelsen af hardwaretimeren er korrekt i den test der sikre, at der generes et tick for hvert 10 ms. Ligeledes ses, at intervallet på de 4 PWM udgange er 10 ms som forventet. Af hensyn til fejlfinding har vi instrumenteret reguleringsalgoritmen, således at den som sidste handling sender en seriel datapakke, der indeholder en række nøgleparametre, som kan plottes på en opsamlede PC (bit 7 - Serial). Ved at måle tiden fra starten af gyroaflastningen og til starten af den serielle datapakke får vi en indikering af den tid, der bruges på beregninger, som er 2 ms; her medregnet tiden til afbrydende interrupts og ved at måle længden af det serielle burst ca. 200 µs, kan vi se den belastning vi har påført systemet ved at indføre test kode.

På Figur 15 (cirkel 1) kan vi se, at der er en forskydning mellem PWM og gyroaftastning, men ifølge koden skulle der ikke være nogen forsinkelse? For at undersøge dette fænomen nærmere laves der en 400 sekunders dataopsamling af IO pindenes opførsel. En hurtig skimning af plottet viser, at efter 19,44 sekunder sker der noget "anderledes", se Figur 16. PWM0 bliver sat som forventet af PWM interrupt Figur 16 (1), ca. 2,33 µs senere bliver

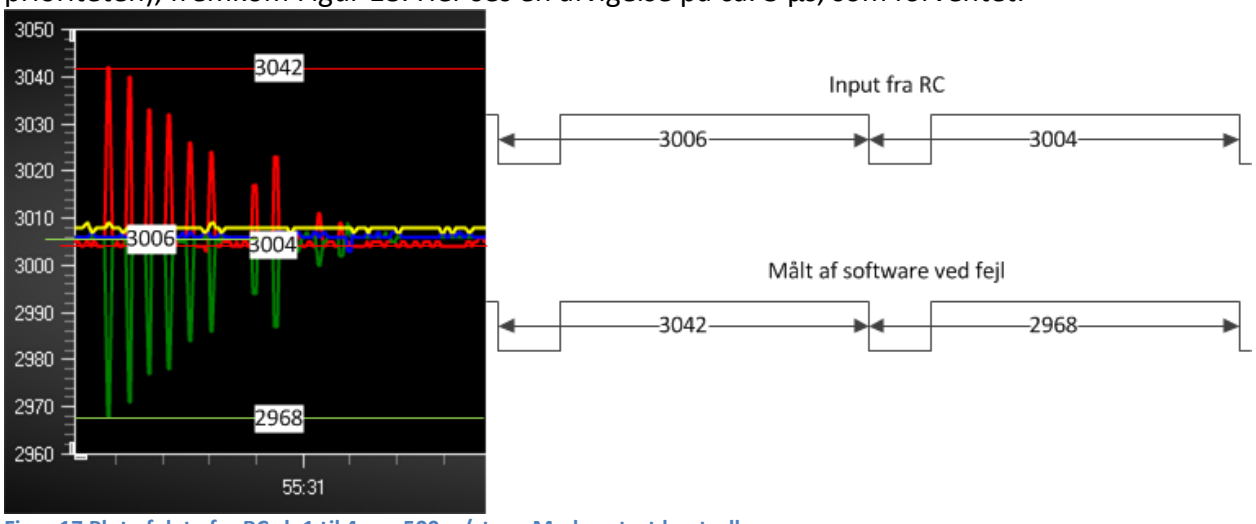
PWM0 interrupt afbrudt af RC interrupt (Figur 15, (2)), dette falder sammen med opdatering af timer registret "OCR1B", et 16 bit register, der anvendes som comparematch til PWM interrupt. Ifølge databladet bruger processoren et 8 bit skyggeregister for at kunne skrive 16 bit på engang. Da RC interrupt aflæser timerens værdi, som ligeledes er et 16 bit register, hvor samme skyggeregister anvendes, vil den efterfølgende skrivning af de sidste 8 bit ikke virke, da skyggeregistret ikke længere har den korrekte værdi. Da "OCR1B" ikke bliver opdateret udløses PWM interrupt ikke før timeren har passeret 0 talt op til den værdi "OCR1B" sidst var sat til. på Figur 15 (3) kommer PWM interrupt igen, men her efter at timeren har talt til 2^{16}. Timer perioden er 500 ns per step, hvilket giver 32,768 ms. Måling af denne tid på opstillingen giver 32,750 en afvigelse på 549 ppm, målenøjagtigheden er opgivet til 0,1% (1000 ppm). Efter dette ses på Figur 15 (4), at vi har fået et offset mellem PWM0 og gyro aflæsningen.

Figur 16 Plot af I/O pinde ved fejl

Samme hændelse gør sig gældende for tidsinterrupt, som kan blive afbrudt, hvorved tiden i systemet forskydes 22,768 ms. Med de valgte signaler i ovenstående diagram er der ikke mulighed for at bevise denne sammenhæng.

Denne analyse har vist, at det ikke altid forholder sig, som man forventer, da TDD ikke tester samtidighed, vil det ikke være muligt at lave test, der fanger denne type fejl. For at undersøge om vores antagelser angående interrupt forsinkelser forårsaget på grund af samtidige interrupts, skal vi kunne "se" de tider, der måles af RC interrupt. Vi valgte at instrumentere koden på en sådan måde, at det er muligt at få udlæst data fra driverlagene. Den valgte hardware har en seriel port, som ikke anvendes. For at kunne teste DDL, sendes de 'rå' data fra RC og gyro ud i et format, der er tilpasset Atmel Data Visualizer. For at påvirke systemet mindst muligt, sendes data først efter, de er blevet behandlet. Herved bliver det muligt at sende data uden at påvirke det resterende system. På Figur 17 ses et plot af de målte tider for de første 4 RC kanaler. For at få alle kanaler ind i billedet er alle kontroller centreret til "middel" udslag forventet 1,5 ms.

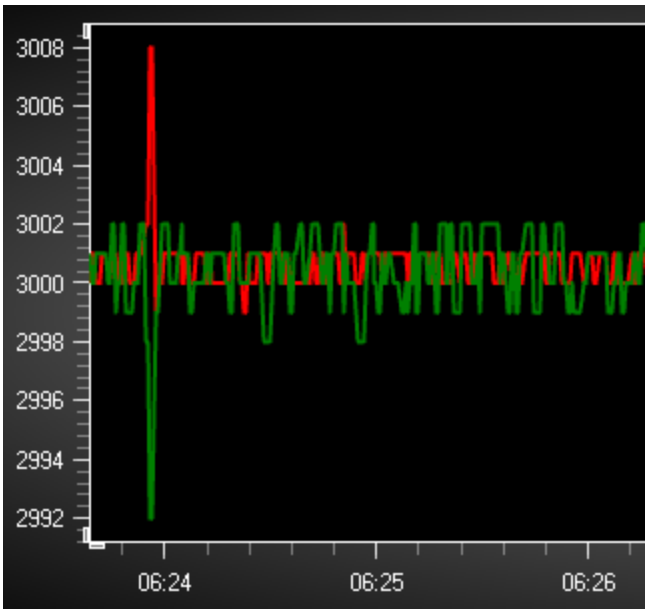
Ud fra analysen havde vi forventet op til 2,9 µs jitter på vores RC signal, men målingen viser ca. 20 µs. En analyse af koden viste, at det var I^2C kommunikationens interruptfunktion der gav anledning til forstyrrelserne. Ved at ændre I^2C til ikke at anvende interrupt (sænke prioriteten), fremkom Figur 18. Her ses en afvigelse på ca. 3 µs, som forventet.



Figur 17 Plot af data fra RC ch 1 til 4, x = 500ns/step. Med centret kontroller.

Figur 17 viser de 4 første RC kanaler fortolket af vores microcontroller ved middel udstyring på fjernbetjeningen. Enheden på y aksen er 500ns. X aksen er tiden, det viste er et udklip, der viser problemet ved I^2C konflikt.

Forkert aftastning af RC som følge af forsinket interrupt (I^2C blokere i her i ca. 20 µs hvilket giver et offset på ca. 40 steps a´ 500 ns) fejlen er tydelig ved at det der lægges til en kanal trækkes fra den næste. Denne ved at skifte prioriteten for I^2C fra høj til lav, blev problemet løst, se Figur 18.



Figur 18 Plot af data fra RC ch 1 til 4, x = 500 ns/step.

Ud fra en analyse af den genererede kode, vil det være forventet med op til 2.9µs forsinkelse, med en timer kørende ved 2MHz, giver det 6 step. Hertil kommer støj / jitter fra modtageren.

8.4 Fejl opdaget i N4/N5

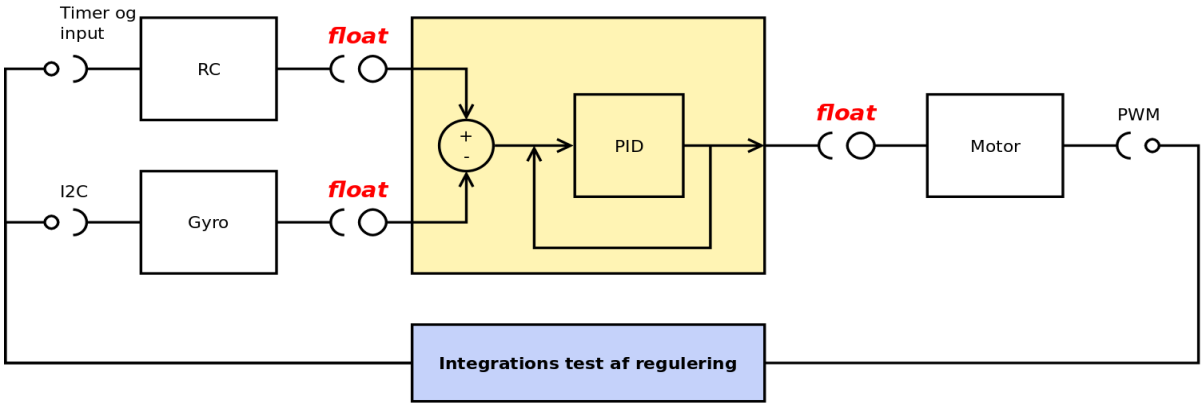
I denne fase testes det samlede system. For quadrokopteren er den vigtigste test at sikre at alle fortegn og akser passer, således at når næsen går ned skal der gives mere gas på de to forreste motorer, og så fremdeles for alle 3 akser. Denne test betinger også, at den fysiske hardware er monteret således, at man får de forventede signaler fra gyro, samt at ledningerne er forbundet til de rigtige motorer.

Dette viser også et klassisk problem ved indlejrede systemer, nemlig at det er nødvendigt at medregne hardwarefejl og uhensigtsmæssigheder, når der skal laves en endelig integrationstest. Ved almindelige PC systemer vil fokus oftest være rettet mod softwaren, da selve PC hardwaren er kendt, og kan testes uafhængigt af eventuelle applikationer, der udvikles hertil. Ved indlejrede systemer udvikles hardware ofte til et dedikeret formål, og der indsættes en processor til at udføre de nødvendige beregninger.

I quadrokopter eksemplet kan det diskuteres, hvorvidt gyroen er monteret forkert eller om der er en fejl i softwaren, hvis der er byttet om på to akser. Vi valgte at rette fejlen i softwaren ved først at rette de testcases, der stimulerer det relevante kode, så vores N1 test kunne eftervise den fejl vi fandt i N5. Herefter blev koden opdateret, så testcasen resulterede i at alle test består.

8.5 Skift fra float til int

Under implantationen af PID regulatoren valgte vi at foretage alle beregninger med floating points. Hovedargumentet for dette valg var, at vi ikke regnede med, at vores forholdsvis lille processor ville være i stand til at kunne lave alle de nødvendige beregninger til tiden. Tanken bag valget var, at vi skulle kunne skrive en test, der kunne give os sikkerhed for,at vi kunne rette noget så vitalt i et modul, som at skifte fra floating point til fixed point beregninger. Denne integrationstest skulle kunne stimulere input for Gyro og RC modulerne, spænde over PID regulatoren og måle på output for motor modulet, vist på Figur 19.



Figur 19 Konceptuel tegning der viser hvordan motor regulatoren interfacet til resten af systemet.

Dette forsøg blev imidlertid ikke udført grundet tidspres og at emnet kun relaterer svagt til dette thesis egentlige formål. Vores forventninger er dog stadig, at dette ville kunne lade sig gøre, men muligvis ikke uden mindre problemer. Eksempelvis ville vi blive nødt til at lave afrunding i nogle mellemregninger, som potentielt ville kunne medføre, at det vil være umuligt at få samme resultat i alle situationer. Derfor skulle denne test formentlig heller ikke teste om PWM var ens, om det blev regnet med floating eller fixed point, men snarere give et vindue på nogle procent til hver side.

En anden problemstilling, der ville kunne opstå i denne test, kommer fra integralet i reguleringen. Vi vil kunne lave en test, der viser hvor meget de to implementationer vil være forskellige for hver iteration. Men eftersom vi har implementeret PID regulatoren med anti-windup, vil denne kunne blive aktiveret på forskellige tidspunkter alt efter hvilke inputs, vi stimulerer med.

8.6 Optimering af kode

Den resulterende kode fra designet er objektorienteret. Dette giver mulighed for indkapsling af funktionalitet. Vores design til styring af IO pinde er baseret på, at der er et objekt til hver IO pind, som sikrer at brugen af denne er korrekt. Ved at bruge denne metode kommer der et ekstra funktionskald med hver gang en port tilgås. Det er nødvendigt, da en pind er repræsenteret ved en bit i en byte, og den valgte microcontroller ikke har support for bit manipulation direkte i enkelt bit, skal værdien først læses, modificeres med den ønskede funktion, og skrives tilbage. Alt samme ting der koster clok

cykler i processoren. For at sikre at opdateringen af PWM udgangen sker til den korrekte tid, er dette lagt i et interrupt hvilket betyder, at det kører med højeste prioritet. Da hele processen med at opdatere pinden er implementeret uden brug af branches, vil den tid der går være den samme hver gang, hvilket betyder at vores PWM blot bliver forskudt nogle clock cykler. Dette vil ikke betyde noget for PWM. Men da PWM opdateringen foregår med højeste prioritet, vil det blokere for at RC aflæsningen kan finde sted. For at minimere påvirkningen af opgaver der kører med højeste prioritet, skal prioritet 1 opgaver være afviklet hurtigst muligt. Det endelige PWM interrupt kan ses i Listing 25.

```
uint8_t pwmPortValue[5] = { 0x01, 0x02, 0x04, 0x08, 0x00 };
ISR(TIMER1_COMPB_vect) // PWM IRQ
{
      PORTA = pwmPortValue[activePwm];
      // re-enable IRQ
      sei();
      if (activePwm < 4) {
            OCR1B = nextPwmStop += pwmArray[activePwm];
            ++activePwm;
      } else {
            OCR1B = nextPwmStop = nextPwmStart += 20000;
            activePwm = 0;
      }
}
```

Listing 25 PWM IRQ

Af denne grund bliver interrupt rutinen til PWM omskrevet fra at bruge objektorienteret kode til en global funktion (ikke en del af en klasse), der skriver direkte i de nødvendige hardware registre. Da vores test abstraktions lag er på registerniveau, kan koden refaktoreres og testes med den samme test som før. Vi har 4 PMW udgange placeret på samme port hvilket betyder, at det er muligt at opdatere alle 4 udgange i en 8 bits skrivning. De 4 mest betydende bit (de resterende 4 udgange på porten) bliver ikke brugt, så vores kode skal ikke tage højde for dette. Dette er afspejlet i vores testcases, der ikke tester tilstanden på de IO pinde, der ikke bruges. På denne måde sikrer TDD, at man ikke implementerer mere kode end nødvendigt som i dette tilfælde, hvor den objektorienterede model tager højde for, at alle IO pinde kan bruges uafhængigt, men det er ikke krævet af testen, hvilket giver udvikleren frihed til at lave en optimal løsning.

Der er mange måder, man kan optimere kode, alt efter hvad man ønsker at opnå; skal det afvikles hurtigere? Skal det bruge mindre RAM eller flash? Dette er en meget interessant problemstilling, hvad enten man snakker udvikling til indlejrede systemer eller PC. Emnet relaterer ikke specifikt til TDD, og der findes masser af andre værktøjer, der kan eftervise udfaldet af de optimeringer, man har udført; output fra kompileren kan vise noget om ram og flash forbrug, WCET (se afsnit 7.3) kan fortælle noget om afviklingstiden.

Ifølge TDD reglerne må man ikke tilføje eller rette noget kode, med mindre man kan vise, at der er et behov. På den anden side dikterer TDD også, at man skal rydde op i sin kode hver gang, man har tilføjet noget kode[6]. På samme måde som man hele tiden laver disse mindre oprydninger, hver gang man har tilføjet ny kode (se afsnit 4.2), vil en kode optimering også "bare" være en oprydning og dermed være helt i tråd med TDD ånden.

Ved at have brugt TDD til at udvikle sin kode vil man have et rigtig godt fundament for at lave disse kode optimeringer, da man har unittest, der kan eftervise om optimeringen, man har lavet, har ført til ændringer i funktionalitet af softwaren.

8.7 Kodekvalitet og udviklingstid

Uden udvikling af styringen til quadrokopteren og andre tidlige projekter udviklet via TDD har vi set, at koden af ukendte årsager har haft en tendens til at have en højere kvalitet end kode, der ikke er udviklet via TDD. I sig selv er det svært at måle på kodekvalitet, og det bliver ikke lettere, hvis man prøver at sammenligne, hvad forskellige udviklere har lavet i helt forskellige projekter.

I [24] skriver Hakan Erdogmus, at blandt studerende der bruger TDD er der en tendens til at være mere produktive og levere en højere kodekvalitet, og at minimums kvaliteten blev højnet liniært med antallet af relevante tests, der blev skrevet. Ligeledes findes der masser af artikler der konkluderer, at TDD ikke giver nogle fordele. I [25] skriver Jacob Proffitt at TDD (beskrevet som test first) performer dårligere end "test last", da det med test last er lettere at skrive en præcis test. Dette betyder dog ikke, at de to forfattere er uenige, da modsætningen til TDD både kan være test-last og ingen test.

Dog kan der ikke være meget tvivl om, at man hurtigere kan finde fejl og regressioner i koden, hvis man på en eller anden måde får testet alle hjørner af koden, hver gang der foretages ændringer (hvad enten denne test er automatiseret unittest skrevet før eller efter koden, eller om det er en komplet manuel test).

Ved at kunne køre en 100 % dækkende unittest hver gang der er blevet foretaget ændringer, er påstanden derfor, at der bliver brugt mere tid på udvikling og mindre på at fejlfinde trivielle problemer. Som beskrevet i afsnit 8.3 beskriver vi et problem, som vi ikke havde forudset, mens vi skrev koden. Vi har ikke noget belæg for at påstå, at vi hurtigere kunne finde årsagen til det her problem, fordi vi havde 100 % test dækning, men vores test kan heller ikke have gjort det sværere at finde denne fejl.

Alt i alt konkluderer vi på dette område, at 100 % test dækning ikke betyder at vi ikke laver fejl, men det er hurtigere at finde de "dumme" fejl og få dem rettet hurtigt. Endvidere betyder 100 % test dækning, at vi uden større bekymringer kunne refaktorere kode ofte,

[6] "Make it work. Make it right. Make it fast." - Kent Beck

når vi har fået en ide til, hvordan det ville være bedre at strukturere den, og på den måde er kodekvaliteten blevet højnet.

9 Relateret arbejde

James Grenning, som er en af underskriverne til the agile manifesto [3] og den eneste underskriver med en baggrund i indlejret software, har skrevet en bog [15] samt nogle artikler [26] [27] om emnet. I sin bog [15] skriver han, at man skal lave et hardware abstraktions lag (HAL) mellem microcontrollerens registre og ens kode for på denne måde at kunne afkoble microcontrolleren og gøre koden testbar. I princippet er der ikke meget forskelligt fra det vi har gjort i dette projekt bortset fra, at vi har lavet en stub af det API microcontrollerproducenter har leveret, og dermed er vi kommet et skridt nærmere på registrene, og dermed hardwaren. På denne måde er vi ikke tvunget til at lave et HAL, og funktionaliteten af vores kode er dermed heller ikke afhængig af, at dette HAL er implementeret korrekt. På den anden side giver Grennings model en smuk afkobling ned til processoren, og man vil reelt set kunne begynde at skrive koden til et projekt, inden man overhovedet ved, hvilken processor projektet skal køre på.

De metoder vi har brugt for at lave stubbe af microcontrolleren producentens API er velkendte test teknikker og der findes mange referencer til dette, bl.a. [11] [12]. Endvidere bruger vi også teknikker til at åbne et lukket interface (6.4). Dette er også beskrevet i [11], hvor han beskriver hvordan et adapter pattern [28] kan bruges til netop det formål. Endvidere beskriver Bærbak også, hvordan dependency injection kan bruges i testsammenhæng, hvor inversion of control forekommer.

Der findes adskillige unittest frameworks, der ville kunne bruges i forbindelse med TDD og indlejret software udvikling. I [15] bruger Grenning Unity og CppUnit. Vi har kun set kort på disse to, men de ville helt sikkert kunne bruges som alternativ til gtest, som vi valgte at bruge.

En mere traditionel test model der bruges på indlejret platforme er Built-in Self-Test (BIST) [29]. En måde at implementere en BIST kunne være at køre en lille stump test kode, der kunne overskrive registerværdier og på den måde simulere ekstern stimulering ved hvert kontekst skifte. En sådan test vil have flere fordele og ulemper. En fordel vil være, at man rent faktisk ville kunne gøre brug af nogle af hardware modulerne i microcontrolleren. Men sammenlignet med TDD, giver BIST ikke den hurtige og enkelte mulighed for at teste alt sin kode hver gang, man har lavet selv små ændringer, da man er afhængig af at kunne uploade sin kode til et target. Endvidere vil BIST også være afhængig af, at der er hardware til stedet, så det vi så med at udvikle næsten alt software inden vi prøvede det i et target, vil heller ikke komme til at virke.

10 Reflektion

Ud fra vores erfaring mener vi, at implantationen af styringen til quadrokopteren kan anses som værende repræsentativ for en større klasse af indlejrede systemer. Da styringen har flere elementer fælles med implementeringer som vi også har kendskab til fra andre og helt anderledes projekter.

Følgende liste er eksempler på forskellige systemer, vi tidligere arbejdet med og som er implementeret bare-metal systemer, og som i en eller anden form har nogle tidskritiske elementer.

- **Dotline**: Synkronisering af flere enheder i et lysshow via en DMX bus [30]. Et system der modtager DMX512 og omsætter DMX kommandoer til forskellige sekvenser på 150 LED's. I Dottline-projektet er der tidskritisk input (læsning af seriel port), som varetages af hardware, hvilket gør denne del mindre interessant i software sammenhæng, samt tidskritisk output, men ikke noget egentlig krav til interne beregninger ud over, at de skal blive færdige på et tidspunkt.
- **PSU**: Implementation af en strømforsyning. Brugeren kan indstille spænding og max. strøm via nogle knapper, og indstillingerne samt aktuelle målingen vises på et display. Bag brugerfladen styrer det indlejrede system et effekttrin, via pulsbredemodulering (PWM) og analog til digital konvertering (ADC), så output fra strømforsyningen stemmer overens med, hvad brugeren har indtastet.
- **Powerpc flashloader**, et system til at programmere FLASH i en powerpc ud fra dual port ram. Powerpc flash loaderen er baseret på en protokol, hvor der ikke er nogen form for tidskrav, hvilket vi ikke mener, man kan udelade i denne sammenhæng, samt det er ikke muligt at få adgang til hardware.

Alle de ovenstående systemer overholder Figur 1, hvor input kan være stimulering fra en bruger eller andre enheder, herunder følere / transducere. Forretningslogikken laver en fortolkning af indkommende data og sætter udgangene efter dette. Dette inkluderer også ting, der skal ske til en given tid i fremtiden. Et andet fællestræk for ovenstående er også, at deres interne tid ikke nødvendigvis skal relatere til et "rigtigt" ur men kun give information om tidsforskel eller starte en handling relativt til en anden handling / input, der er håndteret. Skulle den korrekte tid være kendt, vil det kræve, at der kommer en information udefra der indikerer, hvad denne er (input), og herefter vil sådanne systemer stadig kunne beskrives efter Figur 20 Simplificeret indlejret system.

Figur 20 Simplificeret indlejret system.

11 Konklusion

Ja, det er muligt at bruge TDD på små indlejrede systemer. Vores implementering af quadrokopteren har vist, at det var muligt at lave testcases til de enkelte delmoduler, hvorved funktionen af disse kunne sikres via automatiserede teste fra udviklingsplatformen. Vi har haft adskillige overraskende oplevelser i løbet af projektet. Den velsagtens største og nærmest relaterede til vores initierende spørgsmål var, at se hvor tæt på hardwaren, vi kunne teste. Vi havde fra starten udtænkt planer med at lave et HAL, som skulle testes manuelt på target. Det vidste sig dog, at dette ikke var nødvendigt, da vi kunne stubbe selve registerskrivninger af og dermed teste helt ned til det HAL, der blev leveret fra chip producenten.

En af de største og mest positive overraskelser var også, at vi faktisk var i stand til at implementere stort set alt koden, inden vi havde noget mekanik at teste på. Og det med ret overbevisende resultat i og med, at quadrokopteren faktisk fløj med den første version af koden, der blev uploadet til den.

Det har også vist sig, at koden der kom ud af projektet var af ganske høj kvalitet (let læselig, godt afkoblet, osv.). Fra starten var det ikke noget mål i sig selv, men det faktum at ny introducerede fejl blev fundet med det samme og uden regressioner gjorde, at vi kunne rydde op og refactorere koden ofte og med meget hård hånd, og på denne måde skulle vi ikke bekymre os om konsekvenserne af vores oprydninger.

Vi har ikke i dette projekt fundet en endelig automatisk løsning til at teste, hvorvidt koden er skedulerbar, men ved brug af Bound-T har vi vist, at det er muligt at få nogen grad af automatisering af kontrol af tidsforbrug i de enkelte kodemoduler. Dog kræver denne type værktøjer stadig, at man laver et skeduleringsbudget, og det eneste testen kan holde øje med er, om man overholder det budget, man har stillet op med de regler og afhængigheder, man selv har defineret. Om budgettet er realistisk eller ej, vil der ikke blive testet for.

Men alt dette har jo også haft en pris. Det har taget tid at komme i gang, og at få den forståelse, der var nødvendig for at kunne gå i gang med at udvikle. Vi var heldige fra starten, at vi kunne bruge en stor mængde tid på at få sat udviklingsmiljøet op til at arbejde sammen med vores microcontroller. Dette tog vel lige over en uge med fuldtids arbejde på, og kostede masser af frustrationer over afhængigheder i alle retninger. Efterfølgende har vi dog startet flere nye projekter fra bunden, hvor vi har fulgt den guide, vi lavede oprindeligt [17], og nu hvor vi ved hvordan, tager det mindre end en time at starte et nyt indlejret projekt op med TDD. Taget i betragtning hvor mange udtalelser vi har fundet om folk der ikke kan lide TDD, selv når de udvikler til Java eller C#, hvor den første testcase kan implementeres på mindre en et minut, er der ikke noget at sige til, der ikke findes mange, der bruger TDD til indlejrede systemer. Dog er opsætning af udviklingsmiljøer bare en hurdle, man skal over, og når projektet først kører, er det ikke mere besværligt at anvende TDD til indlejrede systemer end det er til PC applikationer.

Vi havde helt fra starten også mange ambitioner om relaterede problemstillinger vi ville undersøge. Bl.a. ville vi gerne have prøvet at udvikle styringen, så den var kompatibel med forskellige processortyper. Vi valgte dog at droppe den ide, inden vi næsten overhovedet var gået i gang, da vi ikke kunne finde et konkret eksempel fra virkeligheden, hvor denne egenskab kunne være nødvendig. Når man designer sit indlejrede system, har man ofte en mindre processor, så ressourcerne kan være knappe. Hvis vi skulle lave et ekstra lag i form af uCAL (se Figur 9), ville dette kræve ekstra ressourcer. Og hvis dette uCAL så skulle være så generisk, at man kunne bruge de helt samme drivere (hvor det tilbudte interface ikke er identisk med det tilbudte interface fra HAL), ville der skulle bruges rigtig mange ressourcer på at vedligeholde både driver og uCAL. Vores erfaringer fra den professionelle verden viser, at man vælger at udvikle til én familie af microcontrollere.

Den microcontroller vi valgte har ingen DMA kanaler, og dermed fik vi ikke mulighed for at afprøve dette. Det kunne ellers have været interessant at undersøge, men vi antager, at der ikke vil være væsentlig forskel på, hvordan DMA skal testes i forholde til andre on-die enheder.

12 Litteratur

[1] E. Jung, »A Test Process Improvement Model for Embedded Software Developments,« IEEE, 2009.

[2] M. Gordon, »Six Rules for Writing Clean Code,« 2009. [Online]. Available: http://www.embedded.com/design/prototyping-and-development/4008302/Six-Rules-for-Writing-Clean-Code. [Senest hentet eller vist den 24 08 2014].

[3] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland og D. Thomas, 2001. [Online]. Available: http://agilemanifesto.org/. [Senest hentet eller vist den 24 08 2014].

[4] »Scrum,« [Online]. Available: https://www.scrum.org/. [Senest hentet eller vist den 1 09 2014].

[5] F. Sabir, A. Ullah, M. A. Khan, M. Q. Rafique og M. Ali, »Improved Scrum Model with SDLC«.

[6] V. Günal, »Agile Software Development Approaches and Their History,« *Enterprise Software Engineering,* 2012.

[7] M. Lindvall, V. Basili, B. Boehm, P. Costa, K. Dangle, F. Shull, R. Tesoriero, L. Williams og M. Zelkowitz, Empirical findings in Agile methods, Springer, 2002, pp. 197-198.

[8] S. S. Alam og S. Chandra, »Agile Software Development: Novel Approaches For Software Engineering,« *The International Journal Of Engineering And Science (IJES),* årg. 3, nr. 01, pp. 36-40, 2014.

[9] CMS, 27 03 2008. [Online]. Available: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf. [Senest hentet eller vist den 27 08 2014].

[10] »Agile software development,« [Online]. Available: http://en.wikipedia.org/wiki/Agile_software_development. [Senest hentet eller vist den 27 08 2014].

[11] H. B. Christensen, Flexible, Reliable Software Explained: Patterns and Frameworks - Tools and Processes, Chapmann Hall / CRC Press, 2009.

[12] K. Beck, Test Driven Development: By Example, Addison-Wesley Professional, 2002.

[13] M. Fowler, 05 2004. [Online]. Available: http://martinfowler.com/articles/designDead.html. [Senest hentet eller vist den 24 08 2014].

[14] J. Oakland, »Total Quality Management,« i *Total Quality Management*, 1989, p. Tree Swing Figure.

[15] J. W. Grenning, Test-Driven Development for embedded C, Pragmatic Programmers, 2011.

[16] H. He, »What Is Service-Oriented Architecture,« *XML.com,* pp. 1-5, 30 09 2003.

[17] N. Rahbek og J. Sørensen, »TDD for embedded development with eclipse, avr and gtest,« 2014. [Online]. Available: https://sites.google.com/a/b2vn.org/tdd/. [Senest hentet eller vist den 28 08 20141].

[18] T. Mackinnon, S. Freeman og P. Craig, »Endo-Testing: Unit Testing with Mock Objects,« 2000.

[19] D. Isovic og G. Fohler, »Effcient Scheduling o fSporadic, Aperiodic, and Perodic Tasks with Complex Constrians,« IEEE, 2000.

[20] P. Semiconductors, »The I2C-bus specification«.

[21] DS/EN, »Automatic electronik control for household and similar use,« p. kap. H.11.12.1, 08 02 2012.

[22] L. Tan, »The Worst Case Execution Time, Tool Challenge 2006: The External Test,« 2006. [Online]. Available: http://rw4.cs.uni-sb.de/~lili/papers/WCETToolChallengep_Extl.pdf. [Senest hentet eller vist den 24 08 2014].

[23] Tidorum, Bound-T time and stack analyzer, FI-00200 Helsinki: Tidorum Ltd, 2013.

[24] H. Erdogmus, »On the Effectiveness of Test-first Approach to Programming,« *IEEE Transactions on Software Engineering,* 2005.

[25] J. Proffitt, »TDD Proven Effective! Or is it?,« 2008.

[26] J. Grenning, »Extreme Programming and Embedded Systems development,« 2002.

[27] J. Grenning, »Progress Before Hardware,« Januar 2008.

[28] A. pattern, »wikipedia,« [Online]. Available: http://en.wikipedia.org/wiki/Adapter_pattern.

[29] V. D. Agrawal, C. R. Kime og K. K. Saluja, »A Tutorial on Built-in Self-Test. I. Principles,« *IEEE Design & Test,* pp. 73-82, 1 1993.

[30] DMX512, »wikipedia,« [Online]. Available: http://en.wikipedia.org/wiki/DMX512.

[31] J. Ronkainen og P. Abrahamsson , »Software Development under Stringent Hardware Constraints: Do Agile Methods Have a Chance?,« [Online]. Available: http://link.springer.com/chapter/10.1007/3-540-44870-5_10. [Senest hentet eller vist den 24 8 2014].

13 Forkortelser og ord forklaring

RC Radio Controlled, Radio Kontrol

PMW Pulse With Modulation, Puls brede modulation, anvendes i denne rapport også til beskrivelse styrer signal til RC motor, som har en periode tid på 10 ms, samt en udstyring på 10 ti 20 %.

Build Anvendes om software kompilering til for skellige platforme.

Skedulerbarhed En fordanskning af det engelske ord scheduability.

Target

Interrupt Afbrydelse, i denne rapport brugt om afbrydelser af software, som følge af en hardware hændelse.

14 Appendiks:

Appendiks 1 TDD arbejdsgang for isPrime()

Eksemplet i dette afsnit er ikke skrevet i noget konkret eksisterende programmeringssprog eller med et konkret eksisterende testframe. Dog er programmeringssproget meget inspireret af C, og implementationen af testene meget inspireret af Google Test Framework.

Der skal implementeres en funktion der kan teste, om et tal er et primtal. Funktionen isPrime() skal have følgende funktionalitet:

> Funktionen printIsPrime() skal tage et positiv heltal, n, som argument, og bruge printf() til at skrive en besked på skærmen, om n er et primtal eller ej.

Først skal der skrives

```
Test(printMessage) {
    printIsPrine(2);
    expect_eq("2 is prime", lastPrintedMessage);
}
```

Listing 26

Denne test vil fejle (compile fejl), da lastPrintedMessage ikke er defineret. For at få testen til at passe, skal vi her gøre det absolut mindste stykke arbejde nødvendig for at få testet til at passe. I stilen "fake it till you make it" implementers følgende:

```
string lastPrintedMessage;
void printIsPrime(int n) {
    lastPrintedMessage = "2 is prime";
}
```

Listing 27

Nu vil testen ikke fejle længere, så der tilføjes en ny test.

```
Test(printMessage) {
    printIsPrine(2);
    expect_eq("2 is prime", lastPrintedMessage);
    printIsPrine(3);
    expect_eq("3 is prime", lastPrintedMessage);
}
```

Listing 28

Denne test fil fejle, da test casen forventer at lastPrintedMessage skal være "3 is prime" i linie 3, men lastPrintedMessage vil stadig være "2 is prime". For at løse dette må vi tilføje noget kode. Funktionen printIsPrime resset til:

```
string lastPrintedMessage;
void printIsPrime(int n) {
    lastPrintedMessage = n + " is prime";
}
```

Listing 29

Nu passer testen igen, så der tilføjes en ny test

```
Test(printMessage) {
    printIsPrine(2);
    expect_eq("2 is prime", lastPrintedMessage);
    printIsPrine(3);
    expect_eq("3 is prime", lastPrintedMessage);

    printIsPrine(4);
    expect_eq("4 is not prime", lastPrintedMessage);
}
```

Listing 30

Testen vil n fejle, da testcasen forventer at lastMessagePrinted er "4 is not prime", men lastMessagePrinted vil være "4 is prime".

```
string lastPrintedMessage;
void printIsPrime(int n) {
    if(n!=4)
        lastPrintedMessage = n + " is prime";
    else
        lastPrintedMessage = n + " is not prime";
}
```

Listing 31

Nu passer testen, og da udvikleren ikke bryder sig om udtrykket "n!=4" i linie 3, vælges der at rydde lidt op i koden. Der indføres derfor en ny funktion.

```
string lastPrintedMessage;
void printIsPrime(int n) {
    if(isPrime(n))
        lastPrintedMessage = n + " is prime";
    else
        lastPrintedMessage = n + " is not prime";
}
```

Listing 32

Dette kan ikke kompileres, så funktionen isPrime() tilføjes:

```
bool isPrime(int n) {
    return n!=4;
}
```
Listing 33

Nu vil koden igen fejle, men udvikleren har en mistanke om, at den nylige tilføjede funktion isPrime() ikke opfører sig som forventet, så der tilføjes en test.

```
Test(isPrime) {
    expect_eq(true, isPrime(2));
}
```
Listing 34

Testen passer, så der tilføjes endnu en test:

```
Test(isPrime) {
    expect_eq(true, isPrime(2));
    expect_eq(true, isPrime(5));
}
```
Listing 35

Nu fejler testen igen, og som udvikler skal vi begynde og overveje om vi gider at fake det længere, eller om det ikke er lettere at begynde at implementere noget kode, der rent faktisk kan fortælle om et tal er et primtal.

```
bool isPrime(int n) {
    for(int i=2; i<n; ++i)
        if(n%i != 0)
            return false;
    return true;
}
```
Listing 36

Testen vil nu passe igen, men der er fejl i implementationen: 0 og 1 vil med ovenstående blive kategoriseret som primtal (hvilket de per definition ikke er).

Nu ønsker vi at lave et simpelt kommandolinie interface. Programmet skal bede brugeren om at indtaste et tal, og programmet skal så skrive, om tallet er et primtal. Vi har allerede lavet funktionen printIsPrime() og skrevet test til denne, så vi ved, at den virker efter hensigten… lige bortset fra, at det gør den ikke. printIsPrime() skriver ikke noget ud på skærmen. For at forstå problemet starter vi lidt bagfra med at skrive funktionen som den skal se ud.

```
void printIsPrime(int n) {
    if(isPrime(n)
        printf("%d is prime", n);
    else
        printf("%d is not prime", n);
}
```

Listing 37

Problemet er her at det kan blive et større arbejde at få lavet noget kode, der kan undersøge om printf() nu rent faktisk gør, det som vi forventer. Men for sig vidt er vi heller ikke intereserede i at undersøge printf(). Denne funktion er blevet leveret til os, og der er mange andre, der bruger den, så vi formoder at printf() fungerer som forventet. Det betyder så, at vi kun skal undersøge, om vi får kaldt printf() med det rigtige parametre fra vores funktion. Dette kan gøres ved at lave en testspion af printf().

```
// test stub af stdio.h

string lastPrintedMessage;
void printf(string format, va_list[]) {
    sprintf(lastPrintedMessage, format, va_list);
}
```

Listing 38

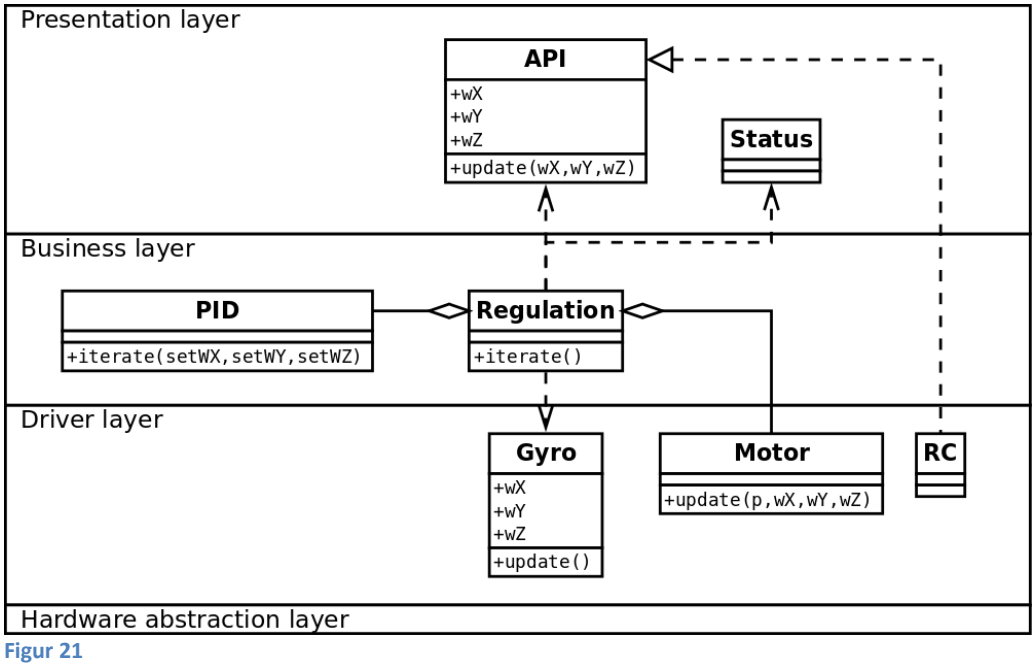
Vores kode under test skal nu bare inkludere stdio.h og bruge primtf(), som vi ellers ville. For at bygge testkoden skal vi så bare sørge for, at vores spionversion af stdio.h bliver inkluderet i stedet for den "rigtige". På denne måde vil den test vi allerede har skrevet til printIsPrime() stadig være valid.

Appendiks 2 4+1 analyse

Ud fra det initierende spørgsmål, kan følgende krav til arkitekturen opstilles, som de drivende krav til den valgte arkitektur.

Da der skal udvikles til en microcontroller, er ressourcerne begrænsede, så koden skal afvikles effektivt.	Performance	Udfør beregning mens der ventes på RC pwm og Gyro
	Performance	Tyndt HAL
Da der ønskes at undersøge mulighed for distribution, skal arkitekturen være forberedt på at styringen skal kunne udføres af en onboard computer	Usability	Kontrol implementeres i eget interface
Skal kunne fungere stabilt i mere end 30 minutter (mere end rigeligt for en batteriopladning)	Availability	Det her lyder mere som et HW krav… Det er heller ikke noget der direkte kan udledes af kravsspecifikationen.

Logisk view



Figur 21

Regulation.iterate() bliver kørt periodisk. Denne kalder PID.iterate() med aktuelle vinkelhastigheder fra hhv Gyro og API. Retur værdien fra PID.iterate() bliver brugt som argument til Motor.update().

Development view

Projektets størrelse gør, at vi forventer, at alt bør kunne implementeres i en pakke.

Process view

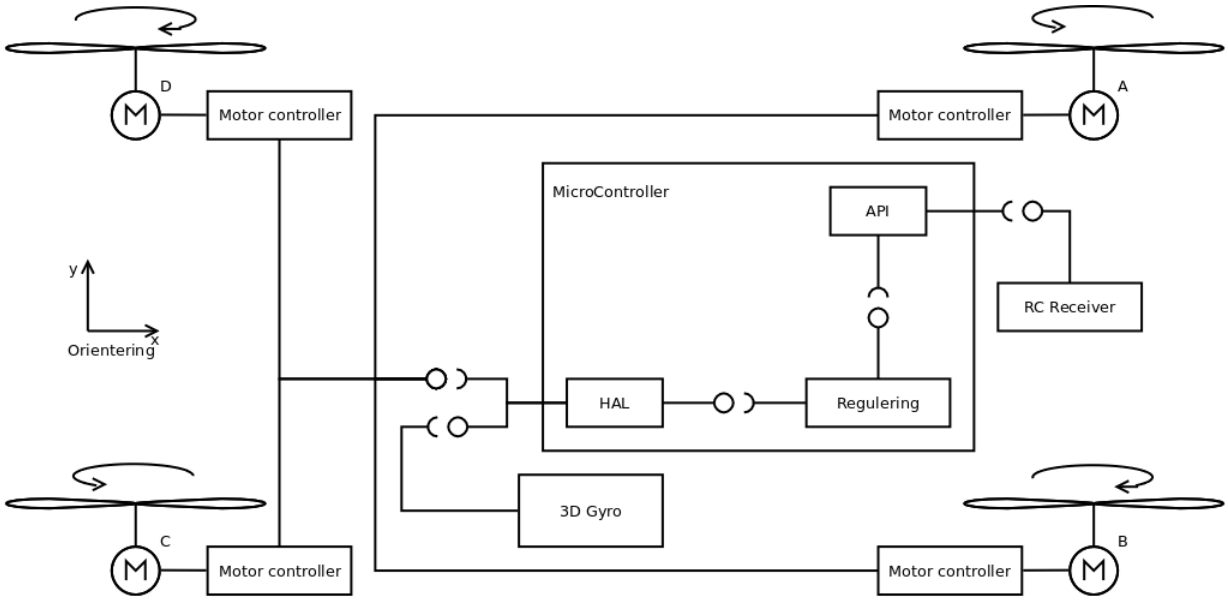
Beskrivelse af tråde og timing af tasks.

1. RC pwm er aperiodisk og bliver triggered af RC modtageren.
    a. Hvis API bliver implementeret med et distribueret interface, skal API være aperiodisk og triggers af "master" computeren.
2. Reguleringen er periodisk og skal køres hvert 10ms
    a. Reguleringen kunne trigges gyroen, når den er færdig.
3. Status skal vise lidt information til brugeren på et display. Skal være periodisk, have lav priotet og køres relativt sjældent.

Physical view

Som udviklingsplatform vælges følgende hardware:

- Quadrocopter, John Jensen design
    - 4 stk Turnigy Park450 børsteløse motorer
    - 4 stk Turnigy Plush 25A motor controllere
    - 1 stk Gyro
- Controller board
    - xmega explained



Figur 22