NTNU
Kunnskap for en bedre verden

Institutt for teknisk kybernetikk

TTK4235 - Tilpassede datasystemer

Heisprosjekt

Kristian Hope
Jon Torgeir Grini

Mars, 2021

Innhold

Liste over figurer

Liste over tabeller

1 Introduksjon

Denne rapporten vil ta for seg heisprosjektet gjort i forbindelse med faget TTK4235 Tilpassede datasystemer våren 2021. Heisprosjektet går ut på å designe og implementere en heis som et resultat av en liste med spesifikasjonskrav beskrevet i [referanse til oppgaveark]. Rapporten vil først belyse hvordan heissystemet er designet, implementert og videre om stegene vi har tatt for å teste at systemet oppfyller kravene. Til slutt vil vi diskutere implementasjonsvalg, og mulige forbedringer av systemet. Vi har tatt utgangspunkt i V-modellen i vårt arbeid med heisprosjektet. Ut i fra kravspesifikasjonene designet vi en arkitektur, og fant ut hvilke moduler som var viktige for prosjektet. Deretter implementerte vi modulene og testet enhetene hver for seg. Påfølgende hadde vi integrasjonstester, før heisen ble evaluert ved en FAT.

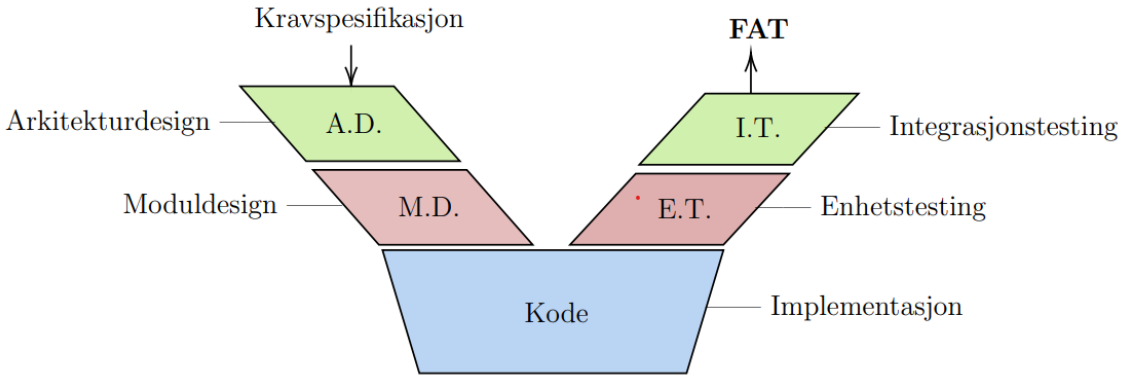


Figure 3: Illustrasjon av V-modellen.

Figure 1: V-modellen

2 Overordnet arkitektur

En oversikt over denoverordnete arkitekturen for et program er viktig for å få et overblikk over hvordan systemet er bygget opp. Det er hjelpsomt både for utviklere som skriver programmet, men også viktig for de som senere trenger å forstå systemet, for eksempel for å vedlikeholde det. Et nyttig verktøy for å skape oversikt over arkitekturen er UML (Unified Modeling Language). I denne seksjonen presenteres tre UML-diagrammer som beskriver den overordnede arkitekturen til heissystemet: klassediagram, sekvensdiagram og tilstandsdiagram.

2.1 Klassediagram

Et klassediagram presenterer modulene som inngår i systemet, og forbindelsene mellom disse modulene. Figur 2 viser et klassediagram for heissystemet denne rapporten betrakter. Merk at moduler omtales med fet skrift gjennom hele rapporten.

Modulen **elevator state machine** inneholder systemets `main()`-funksjon. Tilstandsmaskinen for systemet er implementert direkte i `main()`, og det er her modulene i hovedsak kobles sammen til et fungerende system. **elevator state machine** oppretter også en instans av typen `Elevator`, en `struct` som er definert i modulen **elevator**. Modulen **elevator** innholder også noen operasjoner som anvender en `struct Elevator`. Videre har vi modulen **timer** som inneholder funksjonalitet for en fungerende stoppeklokke. Modulen **queue handler** er modulen som tar seg av funksjonaliteten til køsystemet. Her oppdateres køsystemet med nye ordrer. Modulen inneholder primært funksjonalitet som avgjør hvilke ordre som skal ha prioritet. Til slutt er **hardware** en modul som inneholder en rekke nyttige funksjoner for å kommunisere med hardware, i tilegg til `enum`-ene `HardwareMovement` og `HardwareOrder`.

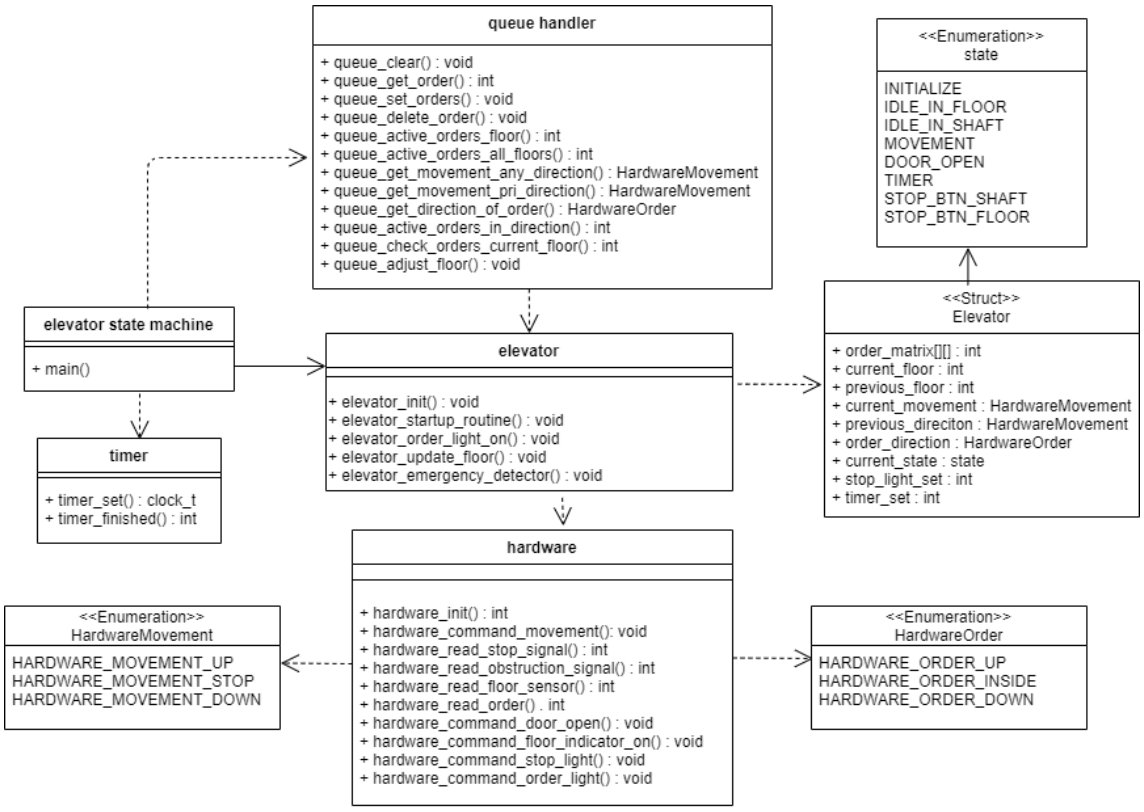


Figure 2: Klassediagram

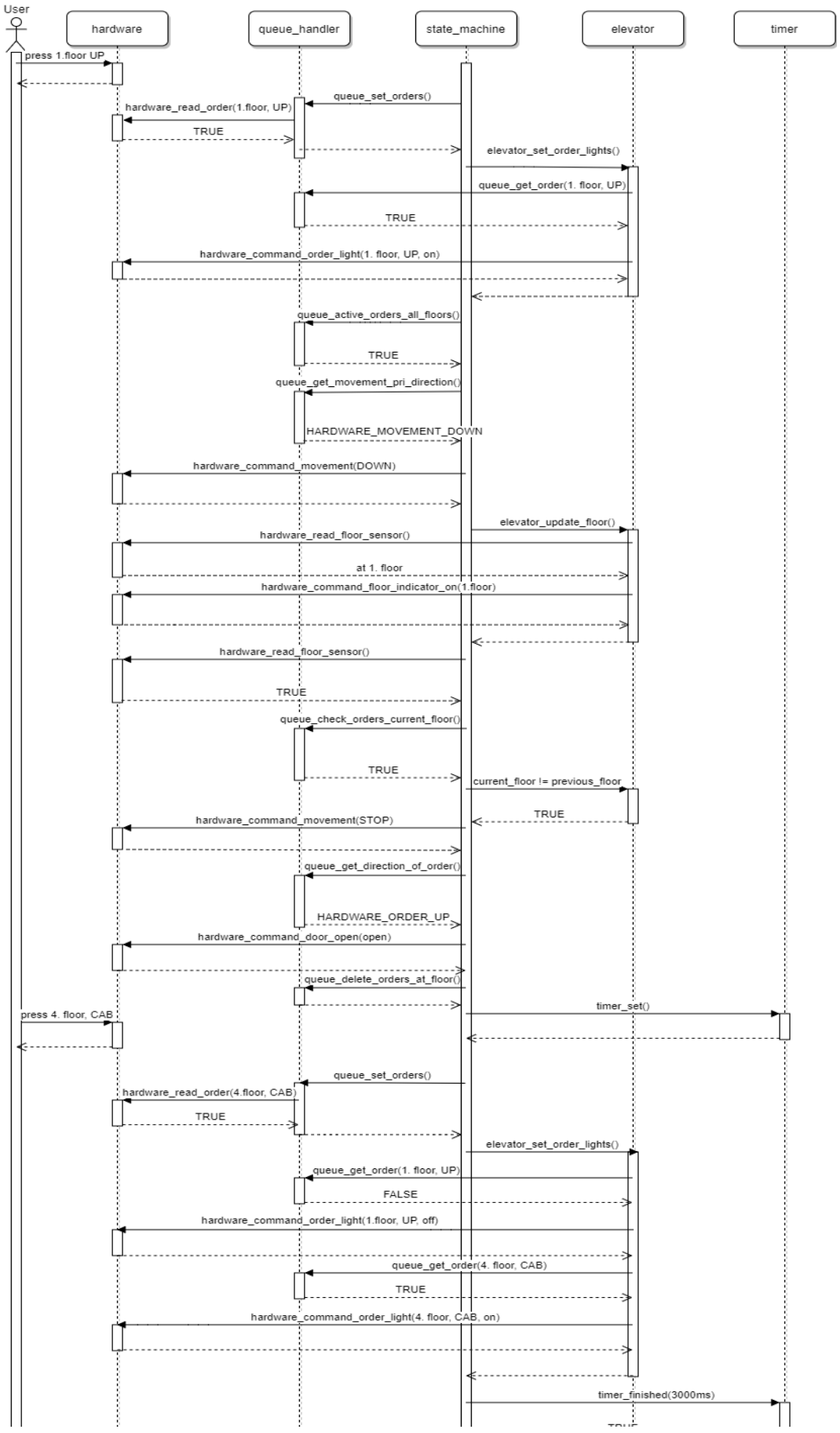
Av diagrammet i Figur 2 er sammenhengene mellom modulene i systemet vist. Man kan se at `struct Elevator` er avhengig av `enum state`, og at modulen **elevator** er avhengig av `struct Elevator`. I selve implementasjon av koden er `struct Elevator` definert i modulen **elevator**, men for å gjøre diagrammet mest mulig oversiktlig er dette separert i klassediagrammet. **elevator** har en avhengighet til **hardware**, fordi en del av funksjonaliteten til **hardware** brukes i **elevator**. Det samme gjelder **queue handler** sin avhengighet til **elevator** og **elevator state machine** sine avhengigheter. I diagrammet forstås det implisitt at **elevator state machine** har en avhengighet til **hardware**, ettersom **elevator** avhenger av denne modulen. Merk at funksjonene i klassediagrammet ikke inneholder sine argumenter. Dette vil tatt stor plass og dratt fokuset noe vekk fra det klassediagrammet primært skal formidle - en oversikt over modulene i systemet.

2.2 Sekvensdiagram

Et sekvensdiagram viser interaksjoner mellom moduler over tid. Sekvensdiagrammet som presenteres i Figur 3 gjelder for sekvensen:

1. Heisen står stille i andre etasje med døren lukket.
2. En person bestiller heisen fra første etasje utenfor heisrommet.
3. Når heisen ankommer etasjen går personen inn i heisen og bestiller fjerde etasje
4. Heisen ankommer fjerde etasje, og personen går av.
5. Etter tre sekunder lukker dørene til heisen seg.

Sekvensdiagrammet i Figur 3 viser i stor grad hvordan modulene i systemet interagerer gjennom sekvensen diagrammet betrakter. I sekvensdiagrammet fokuseres det på de delene av systemet som under sekvensen er aktive. Eksempelvis er funksjonskall som sjekker om stopp-knappen er


User
hardware
queue_handler
state_machine
elevator
timer
press 1.floor UP
queue_set_orders()
hardware_read_order(1.floor, UP)
TRUE
elevator_set_order_lights()
queue_get_order(1. floor, UP)
TRUE
hardware_command_order_light(1. floor, UP, on)
queue_active_orders_all_floors()
TRUE
queue_get_movement_pri_direction()
HARDWARE_MOVEMENT_DOWN
hardware_command_movement(DOWN)
elevator_update_floor()
hardware_read_floor_sensor()
at 1. floor
hardware_command_floor_indicator_on(1.floor)
hardware_read_floor_sensor()
TRUE
queue_check_orders_current_floor()
TRUE
current_floor != previous_floor
TRUE
hardware_command_movement(STOP)
queue_get_direction_of_order()
HARDWARE_ORDER_UP
hardware_command_door_open(open)
queue_delete_orders_at_floor()
timer_set()
press 4. floor, CAB
queue_set_orders()
hardware_read_order(4.floor, CAB)
TRUE
elevator_set_order_lights()
queue_get_order(1. floor, UP)
FALSE
hardware_command_order_light(1.floor, UP, off)
queue_get_order(4. floor, CAB)
TRUE
hardware_command_order_light(4. floor, CAB, on)
timer_finished(3000ms)

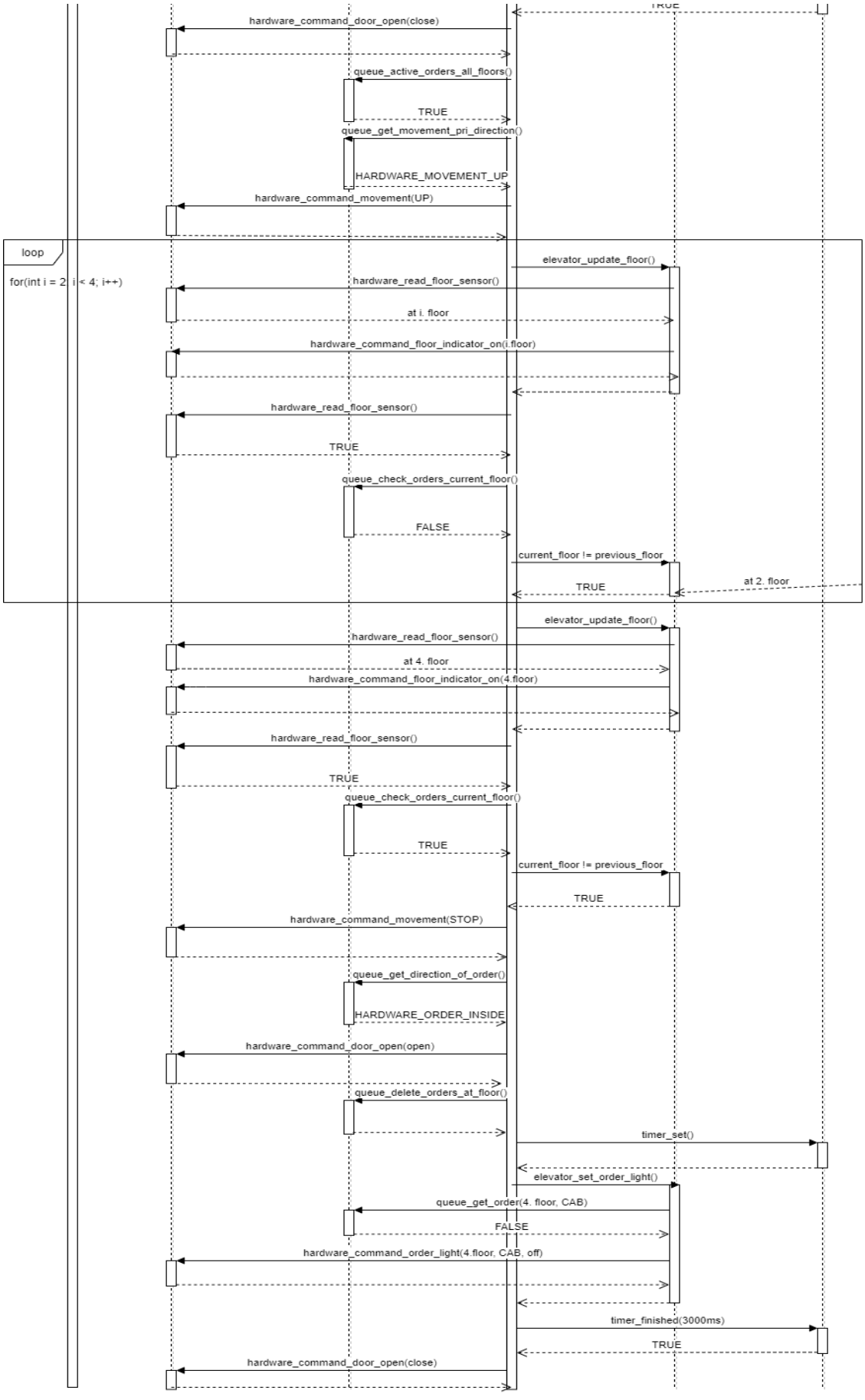


Figure 3: Sekvensdiagram

presset ned ikke en del av sekvensdiagrammet, selv om dette gjøres for hver iterasjon i `main()`. Under sekvensen er det en del funksjonalitet som er av liten betydning, og derfor neglisjeres dette i sekvensdiagrammet. I sekvensdiagrammet ses det også bort i fra oppdatering av variabler. Skulle man tatt hensyn til dette aspektet hadde sekvensdiagrammet blitt uforholdsmessig stort, og ikke minst uoversiktlig.

2.3 Tilstandsdiagram

Et tilstandsdiagram er et diagram som brukes når et system består av et endelig antall tilstander. Det viser gjerne transisjoner mellom tilstandene og hva som trigger en transisjon. I tillegg inneholder diagrammet handlinger heisen gjør mens den er i tilstanden, eller handlinger den gjør ved inngang og utgang av den. Dette betegnes henholdsvis av `do`, `enter` og `exit`.

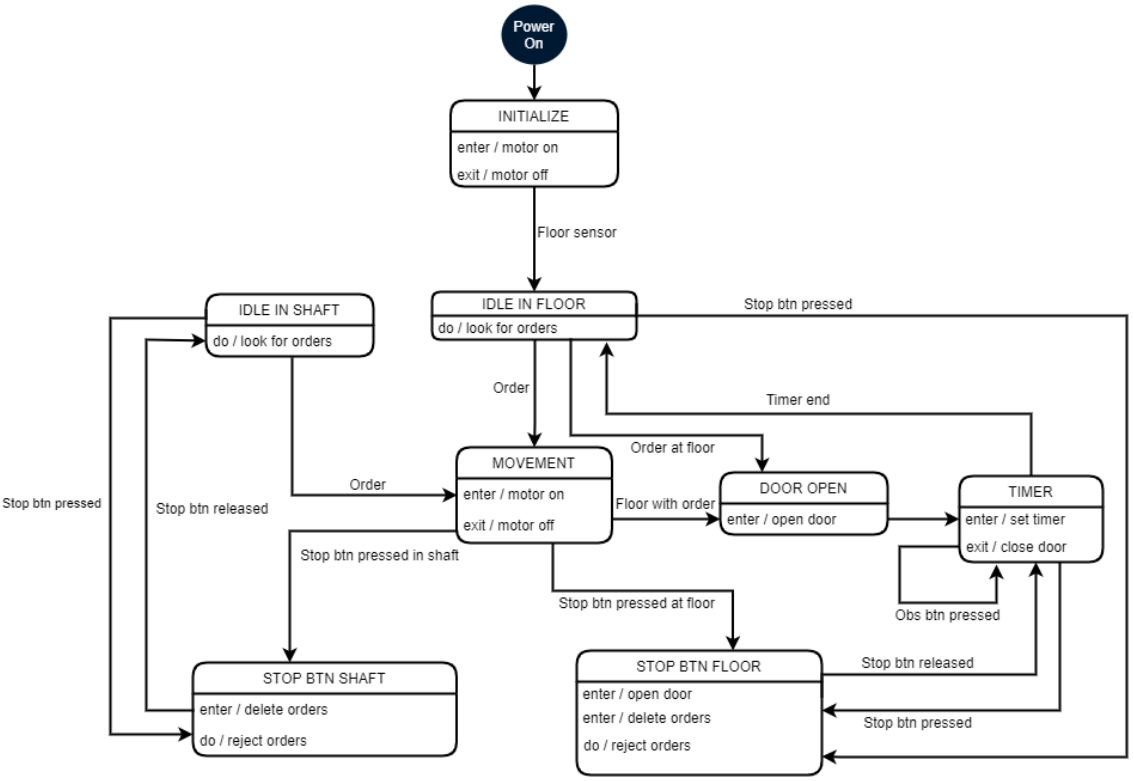


Figure 4: Tilstandsdiagram

Tilstandsdiagrammet i Figur 4 viser at heissystemet består av totalt åtte tilstander. For enkelt- og ryddighetens skyld er handlingene i hver tilstand skrevet med pseudokode. Vi kan finne igjen de samme tilstandene i Figur 2 i `enum state`. I det programmet kjøres går heisen inn i tilstanden `INITIALIZE`. Her initialiseres blant annet en instans av `struct Elevator`. I tillegg kjører heisen oppstartssekvensen slik at den kommer i en definert tilstand. Heisen kommer aldri tilbake til `INITIALIZE` i samme kjøring av programmet, og den fører kun til én tilstand, `IDLE IN FLOOR`. Når heisen er ankommet tilstanden `IDLE IN FLOOR` vil et vanlig heisforløp føre til tilstanden `MOVEMENT` dersom heisen har en aktiv ordre utenfor nåværende etasje. Dersom heisen har en aktiv ordre i nåværende etasje endres tilstanden direkte til `DOOR OPEN`. `MOVEMENT` er den eneste tilstanden der heisen er i bevegelse etter initialisering. Når heisen ankommer en etasje med en ordre den skal behandle, endres tilstanden til `DOOR OPEN`. `DOOR OPEN` er en tilstand som kun er aktiv i én iterasjon av `main()`-løkken av gangen, da tilstanden direkte endres til `Timer`. I `TIMER` startes en stoppeklokke. Denne klokken restartes dersom obstruksjonsbryteren er aktiv. Når klokken når den tiden den er satt til, endres tilstanden tilbake til `IDLE IN FLOOR`. Det er i tillegg to tilstander for når stopp-knappen blir trykket inn: `STOP BTN FLOOR` for når stopp-knappen presses inn mens heisen er i en etasje, og `STOP BTN SHAFT` for når stopp-knappen presses inn mens er heisen er utenfor en etasje. Bare i tilstanden `STOP BTN FLOOR` vil heisdøren åpne seg. Når stopp-knappen

da eventuelt slippes vil tilstanden endres til `TIMER`. Dersom heisen er i tilstanden `STOP BTN SHAFT` og stopp-knappen slippes, går den til tilstanden `IDLE IN SHAFT`. `IDLE IN SHAFT` er en tilsvarende tilstand som `IDLE IN FLOOR`, bortsett fra at heisen, i denne tilstanden, ikke står i en etasje. En av tilstandene for nødstopp nås fra enhver tilstand, utenom fra tilstandene `INITIALIZE` og `DOOR OPEN`, på bakgrunn av implementasjonsvalg vi har gjort.

Arkitekturen vi her har presentert fungerer godt til sitt formål. Den gir et tydelig skille mellom modulenes ansvarsområder. I tillegg gir bruken av `struct` en tydelig samling av nøkkelvariabler, noe som gjør det enkelt for andre moduler å aksessere variabler de har behov for til enhver tid. Implementasjonen av tilstandsmaskinen i `main.c`-filen gir et naturlig knutepunkt for modulene. Arkitekturen har for øvrig en naturlig avhengighetskjede, og dermed unngår man problemer med sirkulære avhengigheter. Alt i alt er dette et godt utgangspunkt for implementasjon av koden.

3 Moduldesign

Denne delen tar for seg en mer omfattende redegjørelse av de ulike modulene, og samhandlingen mellom dem. En modul regnes i dette prosjektet som en samling av operasjoner som er nyttig for resten av programmet. Heissystemet ble separert inn i moduler basert på hva som er de viktigste byggeblokkene i en heis. Hver av modulene har en spesifikk funksjon, og er viktig for helheten i programmet.

3.1 Elevator

elevator modulen består av en `struct Elevator`, samt fem funksjoner som gjør operasjoner på en `Elevator` instans. Modulen er deklarert og definert i filene `elevator.h` og `elevator.c`. `struct Elevator` er en sentral del av programmet da den inneholder variabler med informasjon om heisens nåværende og tidligere kjøremønster og posisjon. Den inneholder også heisens bestillingsmatrise, `order_matrix`, og kontrollvariabler for stoppeklokken og stoppknapplys. Disse vil oppdateres kontinuerlig i tilstandsmaskinen under hele systemets kjøretid. Brorparten av logikken i programmet gjøres på bakgrunn av variablene i `Elevator`.

```
typedef struct Elevator {

    int order_matrix[ELEVATOR_NUMBER_OF_ORDERS][HARDWARE_NUMBER_OF_FLOORS];

    int current_floor;
    state current_state;
    HardwareMovement current_movement;
    HardwareOrder order_direction;

    int previous_floor;
    HardwareMovement previous_direction;

    int stop_light_set;
    int timer_set;

} Elevator;
```

Figure 5: `Elevator` struct

Denne modulen inneholder også oppstartssekvensen til heisen `elevator_startup_routine()`. Oppstartsekvensen sørger for at heisen kommer i en definert tilstand ved å kjøre den til en etasje. I tillegg inneholder **elevator** modulen ordrelys- og stoppknappfunksjonalitet. `elevator_set_order_light()`

setter ordrelysene basert på hvilke verdier i `order_matrix` som er høy og lav. `elevator_emergency_detector()` sørger for at heisen kommer i rett tilstand når stoppknappen blir trykket.

3.2 Queue handler

Modulen `queue handler` omhandler køsystemet til heisen. Den sørger for at heisens køsystem oppdateres med riktige bestillinger, og har funksjonalitet for sletting av ordrer. I tillegg sørger den for at heisen besvarer riktige bestillinger til enhver tid. Køsystemet tar utgangspunkt i en 3x4 bestillingsmatrise med boolske verdier, implementert gjennom et todimensjonalt array. Hver rad i matrisen er en bestillingstype, og hver kolonne er en etasje. Den boolske verdien er `1` for en aktiv ordre, og `0` for en ikke-aktiv ordre for ordretypen på aktuell etasje. Systemet er bygd opp slik at det enkelt skal fungere for et annet antall etasjer også.

$$\begin{bmatrix} 1U & 2U & 3U & - \\ 1I & 2I & 3I & 4I \\ - & 2D & 3D & 4D \end{bmatrix}$$

U - Order up
I - Order inside
D - Order down

Figure 6: Bestillingsmatrise

Modulen består i hovedsak av 13 funksjoner deklarert og definert i filene `queue_handler.h` og `queue_handler.c`. Fellesnevneren ved disse er at alle tar inn en peker til en instans av `struct Elevator` som argument. Dette kommer av at selve bestillingsmatrisen til heissytemet er en variabel i en instans av `Elevator`. Grunnen til at `Elevator` sendes inn som en peker er at funksjonene da har mulighet til å endre variabler i `struct`-en direkte. Fordelen med å ta inn hele `Elevator struct`-en som parameter, kontra kun selve bestillingsmatrisen, er at funksjonene også får enkel tilgang til andre nøkkelvariabler i `Elevator`. Variablene brukes i kombinasjon med bestillingsmatrisen til å blant annet bestemme hvilken retning heisen skal bevege seg i, og hvilke(n) ordre(r) som skal besvares først.

Det er i denne modulen man også finner den mest kompliserte funksjonaliteten til heisen. Funksjonen `queue_get_movement_pri_direction()` returnerer retningen heisen skal bevege seg i ved aktive ordrer. Den vurderer randbetingelser, ordretypen til den sist behandlede ordren og heisens nåværende etasje for å komme frem til riktig bevegelsesretning for heisen. I tillegg har vi `queue_active_orders_in_direction()` som sjekker om heisen har aktive ordrer i retningen spesifisert av argumentet. Disse funksjonene er viktige for at heisen skal ha et logisk prioriteringssystem for bestillinger.

Det er flere grunner til at det å implementere bestillingssystemet gjennom en todimensjonal array er fornuftig. Siden det ikke er nødvendig at en ordretype fra en etasje akkumuleres for flere knappetrykk, gjør dette at man slipper å ta hensyn til antall personer som venter ved en etasje. Det er heller ikke nødvendig at heisen behøver å vite nøyaktig hvilken ordre den skal behandle som neste. Den trenger kun å vite hvilken retning den skal bevege seg i. Da kan den sjekke aktive ordrer for hver etasje den passerer, og avgjøre om det er en bestilling på etasjen den passerer som den skal behandle nå. Siden prioriteringssystemet er nokså enkelt, fungerer et todimensjonal array godt. Som tidligere nevnt, har vi definert funksjoner som sjekker prioritering. Disse kaller heisen når den ankommer en etasje, og dette er også prioriteringssystemet til heisen. Med det todimensjonale arrayet unngår man i tillegg problematikk som minnelekkasje.

3.3 Timer

`Timer`-modulen inneholder funksjonalitet for en stoppeklokke. Denne brukes hovedsakelig for å holde heisdøren åpen i tre sekunder. Modulen ligger i filene `timer.h` og `timer.c`. Stoppeklokken er implementert ved hjelp av C-biblioteket `time`. Modulen er enkel i bruk, da den kun består av to funksjoner. `timer_set()` returner en verdi av typen `clock_t` som inneholder antall klokkesykluser siden starten av programmet. Den andre funksjonen `timer_finished()` sammenligner en tidligere

clock_t-verdi med nåværende clock_t-verdi. Antall sekunder som har gått mellom de to clock_t-verdiene beregnes ved hjelp av en konstant CLOCKS_PER_SEC som er definert i time.

3.4 Hardware

Modulen **hardware** er brukergrensesnittet til maskinvaren. Den inneholder primært funksjoner for å lese og skrive til hardware. I **hardware** er heisetasjene nullindeksert, altså det vi i rapporten omtaler som 1. etasje, den nederste etasjen, er den 0. etasjen i koden. Derfor er også heisetasjene i den selvimplementerte koden nullindeksert.

3.5 Elevator state machine

elevator state machine er heisens endelige tilstandsmaskin. Den er en essensiell del av programmet som sørger for at heisen utfører riktig prosedyre avhengig av tilstanden den er i. Modulen er et knutepunkt for de andre modulene, hvilket er en av grunnene til at den er implementert i main() i main.c. Den har ingen egne funksjoner, men sørger for å koble de andre modulene sammen til en fungerende enhet. En annen grunn til at tilstandsmaskinen er implementert i main() er at denne koden skal bare kjøres én gang og det å legge koden inn i funksjoner har derfor liten til ingen hensikt. I tillegg vil det å legge koden inn i funksjoner være å "gjemme unna" kompleksitet, og det er ikke noe argument i seg selv. Tilstandsmaskinen baserer seg i stor grad på et "switch-statement", der current_state i en instans av Elevator blir evaluert. De ulike tilstandene tilstandsmaskinen kan oppta er definert i enum state i states.h.

```
typedef enum state {
    INITIALIZE,
    IDLE_IN_FLOOR,
    IDLE_IN_SHAFT,
    MOVEMENT,
    DOOR_OPEN,
    TIMER,
    STOP_BTN_SHAFT,
    STOP_BTN_FLOOR
} state;
```

Figure 7: Enum med tilstander

Som man kan se av Figur 7, finnes det to ulike tilstander for stoppknappen. STOP_BTN_SHAFT er tilstanden hvor stoppknappen er trykket inn mens heisen er i sjakten, og STOP_BTN_FLOOR er tilstanden hvor stoppknappen trykkes inn når heisen er i en etasje. Bakgrunnen for at det skilles mellom disse tilstandene, er at de har ulike innganger og utganger. I STOP_BTN_FLOOR skal heisen åpne døren og endre tilstand til Timer, men i STOP_BTN_SHAFT skal heisen holde døren lukket og endre tilstand til IDLE_IN_SHAFT. En annen bemerkning er at det skilles mellom IDLE_IN_SHAFT og IDLE_IN_FLOOR, selv om forskjellen mellom disse er på grensen til marginal. I IDLE_IN_FLOOR skal heisen søke etter aktive ordre i nåværende etasje. Dette skal imidlertid ikke skje i IDLE_IN_SHAFT ettersom heisen står i sjakten. I DOOR_OPEN endres tilstanden direkte til TIMER. Vi vurderte det som hensiktsmessig å skille DOOR_OPEN og TIMER, ettersom det å åpne døren på inngangen til TIMER ville skapt problematikk med tilstanden STOP_BTN_FLOOR. I STOP_BTN_FLOOR er døren allerede åpen i det tilstanden endres til TIMER. MOVEMENT er tilstanden der heisen er i bevegelse for å behandle aktive ordre. Denne sjekker hver etasje for aktive ordrer, og dersom den finner en ordre den skal behandle endres tilstanden til DOOR_OPEN.

4 Testing

Det å teste et system både underveis og avslutningsvis er avgjørende for et godt resultat. I dette prosjektet har vi testet modulene hver for seg, *enhetstesting*, og modulene samlet, *integrasjonstesting*. Enhetstesting og integrasjonstesting er steg som inngår i V-modellen, Figur 1, og har derfor vært nyttig å bruke gjennom de ulike fasene av dette prosjektet.

4.1 Enhetstesting

Enhetstesting har blitt gjort på modulene for å teste at de virker som ønsket. Det var mer eller mindre ukomplisert å enhetsteste modulene som har ingen eller få avhengigheter. Eksempelvis ble modulen **timer** testet ved å starte en klokke, og deretter skrive ut en melding til terminalen når tiden gikk ut. Da kunne vi bekrefte at klokken fungerte i et ønsket antall sekunder. Videre ble **elevator**-modulen testet ved hjelp av GNU-debuggeren GDB. Dette viste seg å være et nyttig verktøy for blant annet å bekrefte at initialisering av en `struct Elevator` oppførte seg som forventet. Deretter ble det lagt inn litt enkel kode som fikk heisen til å kjøre, og det kunne bekreftes ved hjelp av GDB at funksjonene i **elevator**-modulen virket som ønsket. For eksempel at `elevator_update_floor()` oppdaterer `current_floor` og `previous_floor` fra `struct Elevator` i Figur 5 til forventede verdier. I tillegg ble det observert at rett etasjeindikator lyste fra etasjepanelet. Modulen **queue handler** var i dette prosjektet mest omfattende å teste, fordi den inneholder flest funksjoner. Testing av denne modulen krevde i tillegg at modulen **elevator** var mer eller mindre ferdigstilt ettersom **queue handler** avhenger av denne. For å teste **queue handler** ble `order_matrix` overvåket i GDB for å se at matrisen ga ønsket oppførsel når nye bestillinger ble opprettet og slettet. For å teste prioriteringssystemet implementert i **queue handler** ble det opprettet en rekke kall på prioriteringsfunksjonene fra `main()`. Siden disse funksjonene typisk returnerer `1` og `0` kunne man enkelt se om funksjonene ble aktivert ved å skrive ut en melding til terminal.

4.2 Integrasjonstesting

Integrasjonstestingen og enhetstesting hadde i dette prosjektet en mer eller mindre glidende overgang. Siden **queue handler** krevde en ferdigstilt **elevator**-modul kan man si at integrasjonstestingen startet allerede her. Uansett, var det her det var hensiktsmessig å implementere tilstandsmaskinen. Denne delen av testingen ble ikke så utfordrende, da modulene allerede fungerte som tilsiktet etter enhetstestingen. Likevel dukket det opp noen utfordringer knyttet til tilstandsmaskinen, der den gjorde noen uforventede transisjoner. Dette ble løst ved å følge variabelen `current_state` fra `struct Elevator` i GDB.

4.3 Egen testprosedyre

I tillegg til enhetstesting og integrasjonstesting ble det også gjennomført en egen testprosedyre for systemet i forkant av FAT. Den består i hovedsak av fire tester, som samlet sjekker om systemet oppfyller kravspesifikasjonene. Hvilke kravspesifikasjoner hver test sjekker kan man se i Tabell 1.

Tabell 1 gir en oversikt over hvordan testene oppfyller heisspesifikasjonene.

Test 1 - oppstart:
Skru på heisen i sjakten, og trykk på vilkårlige bestillinger.

Respons:
Heisen ignorerte bestillinger, kjørte oppover, og stoppet i nærmeste etasje der etasjelyset ble satt.

Test 2 - lys, bestilling og kjørebane:
Heisen skal stå i 1. etasje. Trykk på bestilling 2. etasje opp og ned, 3. etasje ned og 4. etasje

Krav	Test 1	Test 2	Test 3	Test 4
O1	X			
O2	X			
H1		X	X	X
H2		X		
H3		X		
H4		X	X	
L1		X		
L2		X		
L3	X	X		
L4		X		
L5		X		
L6			X	
D1			X	
D2		X	X	
D3			X	X
D4			X	
S1		X	X	
S2				X
S3		X		X
S4				X
S5				X
S6				X
S7			X	X
R1			X	
R2	X	X	X	X
R3	X	X		

X = heisspesifikasjon bekreftet oppfylt av test

Table 1: Heisspesifikasjonskrav

ned. Når heisen ankommer 4. etasje, trykk 1. etasje opp. Observer om bestillingslys og etasjelys responderer.

Respons:
Heisen kjørte til 4. etasje, og stoppet i 2. etasje på veien for å ta bestillingene der. På vei til 4. etasje kjørte heisen forbi 3. etasje. Etter at bestillingen i 4. etasje ble behandlet stoppet heisen i 3. etasje på vei ned til 1. etasje, der den siste bestillingen ble behandlet og heisen stoppet. Etasjelysene ble oppdatert ettersom heisen flyttet seg mellom etasjene. Bestillingslysene ble skrudd på da bestillingsknappene ble trykket inn, og slo seg av i det heisen ankom og stoppet i aktuell etasje. Heisen holdt seg innenfor den definerte kjørebanen og heisdøren holdt seg lukket så lenge heisen var i bevegelse.

Test 3 - heisdør:
Heisen skal stå i 1. etasje uten bestillinger. Vent et par sekunder med å trykke på bestillingsknapper. Trykk ned obstruksjonsbryteren i 3s. Bestill så 2. etasje opp. Etter døren har åpnet og lukket seg, trykk på 2. etasje på heispanelet, og deretter skru på obstruksjonsbryteren i 5s. Når heisen har gjort seg ferdig i etasjen, holdes stoppknappen inne i 5s.

Respons:
Heisdøren var lukket mens heisen var i 1. etasje. Heisen ble ikke påvirket av obstruksjonsbryteren når døren var lukket. Heisen kjørte så opp til 2. etasje, og døren åpnet seg i 3s. Etter ny bestilling i 2.etasje åpnet døren seg igjen, og forble åpen så lenge obstruksjonsbryteren var skrudd på, og ytterligere 3s etter den var skrudd av. Heisdøren var åpen alle 5s stoppknappen var trykket, og lukket seg 3s etter den ble sluppet. Stoppknappen lyste så lenge den var trykket inn. Heisen forble i etasjen.

Test 4 - sikkerhet:
Heisen skal stå i 3. etasje. Trykk på 3. etasje på heispanelet, slik at døren åpner seg. Trykk

deretter ned obstruksjonsbryteren før døren lukker seg. Gjør så en rekke vilkårlige bestillinger på enten heispanelet eller etasjepanelet. Skru av obstruksjonsbryteren, og trykk deretter hurtig på stoppknappen før heisen får flyttet på seg. Hold den i 5s, og gjør vilkårlige bestillinger. Slipp stoppknappen, og trykk på 2. etasje opp. Mens heisen er i sjakten, trykk på stoppknappen, og hold den i 5s. Igjen gjør vilkårlige bestillinger.

Respons:
Både i 3. etasje og i sjakten ble alle bestillinger fjernet da stoppknappen ble trykket inn. Heisdøren var åpen i 3 sekunder etter at stopp-knappen ble sluppet mens heisen sto i 3. etasje. Heisen ignorerte alle nye forsøk på å gjøre bestillinger mens stoppknappen var trykket inn. Heisen forble mellom 2. og 3. etasje etter knappen ble sluppet i sjakten.

5 Diskusjon

Denne seksjonen vil inneholde en diskusjon av heissystemet. Vi vil diskutere ulike implementasjonsdetaljer, samt argumentere for valg vi har tatt, og bringe frem mulige forbedringer til systemet.

Systemet består, som tidligere diskutert, av fem moduler. Vi kan se av tilstandsdiagrammet i Figur 2 at disse modulene har et ulikt antall avhengigheter. Modulen **queue handler** avhenger av både **elevator** og **hardware**. Ideelt sett bør en modul ha så få avhengigheter som mulig, slik at modulen enkelt kan erstattes ved et senere tidspunkt. Vi mener at modulene i systemet har et hensiktsmessig antall avhengigheter for å ikke gjøre systemet for komplekst. Disse avhengighetene muliggjør at **queue handler** har enklere funksjoner som i større grad samler funksjonalitet. For eksempel kan `queue_get_order()` både ta inn en peker til en instans av `Elevator` som argument og kalle `hardware_read_order()` fra **hardware** for å oppdatere bestillingsmatrisen med nye ordrer. Funksjonens virkning kunne imidlertid også vært muliggjort med færre avhengigheter ved at funksjonen hadde tatt inn en todimensjonal array. Denne metoden ble prøvd ut tidlig i utviklingsfasen, men det viste seg å være en noe kaotisk løsning, siden noen av funksjonene da måtte ta inn en rekke andre argumenter med informasjon som allerede var lagret i `Elevator`. Derfor landet vi på at det var fornuftig å bruke en peker til `Elevator` som argument i **queue handler**-funksjonene.

De ulike modulene ble designet for å separere funksjonalitet. Utover i designfasen oppsto det noen funksjoner som ikke hadde en tydelig tilhørlighet. Et eksempel på dette er funksjonen `queue_adjust_floor()` i **queue handler** modulen. Dette er i grunn kun en "workaround" for å få heisen til å alltid bevege seg i riktig retning etter stoppknappen har blitt trykket inn. Konklusjonen ble at siden funksjonen benytter seg av andre funksjoner i kømodulen, var det naturlig at den også ble plassert der. På en annen side hadde koden blitt mer oversiktlig om kømodulen kun inneholdt funksjonalitet for selve køsystemet.

Valget vårt om å implementere tilstandsmaskinen i `main()` har både fordeler og ulemper. Som tidligere nevnt i rapporten er valget basert på at det ikke har noe for seg å "gjemme unna" kompleksitet, spesielt når dette er kode som kun kjøres én gang. Likevel ville det å legge tilstandsdiagrammet i en egen fil ført til en ryddigere `main()`-funksjon og en mer veldefinert **elevator state machine** modul. Dette kunne potensielt gjort det enklere å bytte ut tilstandsmaskinen i fremtiden, samt enklere for en annen utvikler å få oversikt over modulene i programmet.

I løpet av utviklingsprosessen oppdaget vi at heisdøren ble stående åpen i en etasje så lenge en bestillingsknapp i denne etasjen holdes nede. Heisen blir med andre ord ute av stand til å bevege seg videre fram til knappen blir sluppet. Hvorvidt dette er normal heisoppførsel er diskuterbart. De fleste heiser vil nok lese signalet fra etasjepanelet kun i det knappen trykkes inn. Det hjelper dermed ikke å holde inne knappen kontinuerlig. Det er likevel uproblematisk å holde en virkelig heis fast i en etasje. Dette kan for eksempel gjøres ved å trykke på etasjepanelet hver gang heisdøren er i ferd med å lukke seg. Det må nevnes at denne detaljen har liten betydning for heissystemet i sin helhet.

En av tilstandene i **elevator state machine** er `DOOR_OPEN`. Denne tilstanden har ingen annen funksjon enn å åpne heisdøren, og deretter endre tilstand til `TIMER`. `DOOR OPEN` kan derfor knapt

kalles en tilstand, og bør helst ses på som en transisjon. Dette implementasjonsvalget ble gjort, fordi `STOP BTN FLOOR` også endrer tilstand til `TIMER`. Dersom `TIMER` tilstanded skulle håndtert åpning av heisdøren ved inngang, hadde det vært forvirrende siden døren allerede er åpen når tilstanden endres fra `STOP BTN FLOOR`. Dersom man derimot ønsker å redusere antall tilstander som er implementert, kan dette være et godt sted å begynne. Et annet poeng er at vi skiller mellom stopknappscenarioer i sjakten og i etasjer. Dette ble gjort for å gjøre koden mer oversiktlig og intuitiv, men kan enkelt erstattes med en enkelt tilstand og noen ekstra "`if`-statements".

Man kan se av tilstandsdiagrammet i Figur 4 at tilstanden `DOOR OPEN` ikke har en mulig transisjon til noen av nødstopp-tilstandene. Dette kan virke lite intuitivt, men dette er et bevisst implementasjonsvalg vi har gjort. For det første er tilstanden `DOOR OPEN` aldri aktiv i mer enn en iterasjon i `main()` av gangen, og siden C er et svært effektivt språk vil det ikke merkes forskjell på om systemet bruker en ekstra iterasjon på å nå en nødstopp-tilstand. I tillegg er heisen allerede i ro og døren åpnet i `DOOR OPEN`-tilstanden. Dette er akkurat samme funksjonalitet som `STOP BTN FLOOR` gjør ved en inngang, da den kun stopper heisen, åpner døren og endrer tilstanden til `TIMER` når stopp-knappen slippes. Dermed vil det ha en ubetydelig virkning å implementere en transisjon fra `DOOR OPEN` til `STOP BTN FLOOR`.

6 Avslutning

Heissystemet som er implementert i dette prosjektet har både sine styrker og svakheter. Det kan argumenteres for at noen av modulene som inngår i systemet ideelt sett burde hatt færre avhengigheter. Heisen har også flere tilstander som tilsynetalende ser ganske like ut, men som er viktige for å skille heisens prosedyre i ulike situasjoner fra hverandre. Dette gjør programmet mer oversiktlig og lesbart. Gjennom arbeidet med prosjektet er det funnet enkle og robuste løsninger til utfordringer som angår tidtaking og køsystem. Med gjennomgående bruk av V-modellen svarer prosjektet til alle heisspesifikasjonskravene.