Aarhus University School of Engineering

Autonom forfølgelsesdrone med smartphone applikation

Bachelorprojekt
Elektroingeniør

Modul- & Integrationstest

Projekt nr: 16114
Jonas Risager Nielsen - 201270337
Benedikt Wiese - 201310362
Mathias Poulsen - 201370945

Vejleder
Torben Gregersen
Aarhus University School of Engineering

15. december 2016

Versionshistorie for Modultest & Integrationstest

Version	**Dato**	**Beskrivelse**
0.0	31.08.2016	Opsætning af dokumentet.
0.1.0	07.10.2016	Tilføjet modultest for serveren.
0.1.1	08.10.2016	Tilføjet modultest for dronens software.
0.1.2	08.10.2016	Tilføjet modultest for dronens hardware.
0.2.0	15.11.2016	Modificeret modultest for serveren efter ændringer.
0.2.1	22.11.2016	Modificeret modultest for dronens software.
0.2.2	23.11.2016	Modificeret modultest for dronens hardware.
0.3	23.11.2016	Tilføjet modultest af dronens matematiske funktioner.
0.4	24.11.2016	Tilføjet modultest for applikationen.
0.5	07.12.2016	Ordforklaring tilføjet.
0.6	15.12.2016	Revideret
1.0	15.12.2016	Aflevering

Ordforklaring

Forkortelse	Forklaring
3G	3. Generations mobilnetværk.
API	Application Programming Interface.
GPS	Global Positioning System.
HTTP	Hypertext Transfer Protocol.
JSON	JavaScript Object Notation.
LED	light-emitting diode.
Maple 2016	Matematikprogram.
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor.
RGB	Red Green Blue.
Shadow BAC-X1	Navnet på dronen, der udvikles i dette projekt.
UART	universal asynchronous receiver/transmitter.
URL	Uniform Resource Locator.
USB	Universal serial bus.
UTC	Coordinated Universal Time.
V	Volt.

Indholdsfortegnelse

Modultest - Shadow BAC-X1

1

1.1 Modultest - Server

I dette afsnit af modultesten er serverens tests beskrevet. Disse er brugt for at sikre, at serveren fungerer optimalt, når denne er isoleret fra det samlede system.

Da serveren indeholder 6 primære funktionaliteter er disse blevet testet gennem modultesten. Dette omfatter følgende funktionalitet:

- Test 1 - At kunne returnere den seneste besked fra en drone.
- Test 2 - At kunne returnere den seneste besked fra en applikation.
- Test 3 - At kunne returnere den seneste besked med en given kommando fra en drone.
- Test 4 - At kunne returnere serverens tid og dato.
- Test 5 - At kunne modtage og gemme en besked fra en drone.
- Test 6 - At kunne modtage og gemme en besked fra en applikation.

Til at udføre disse tests benyttes programmet, Postman [1], som også er brugt tidligere i projektet. Når der skal sendes HTTP beskeder i testene, skal dette gøres til specifikke URL's. Disse er listet herunder og er baseret på projektets server.

1. `http://shadowbac-x1.gear.host/api/shadowbac?time=gettime`
2. `http://shadowbac-x1.gear.host/api/shadowbac?identifier=appXXXX&command=KOMMANDO`
3. `http://shadowbac-x1.gear.host/api/shadowbac?identifier=droneXXXX&command=KOMMANDO`
4. `http://shadowbac-x1.gear.host/api/shadowbac`

1.1.1 Test 1 - Seneste besked fra en drone

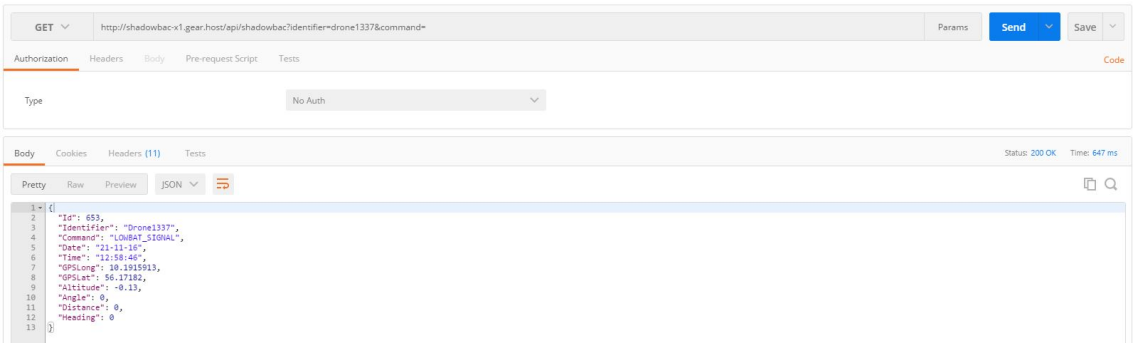
Formål
Formålet med denne test er at verificere, at serveren kan returnere den seneste besked fra en drone med en given identifier.

Test forløb
For at sikre en ordenlig test, skal serveren være oppe og køre, og minimum en besked fra dronen skal ligge i databasen.

Gennem programmet, Postman, sendes en HTTP GET-besked til URL nr. 3. URL'en skal modificeres med den korrekte identifier og en tom string som command. URL'en brugt i testen ses i Listing 1.1.

```
http://shadowbac-x1.gear.host/api/shadowbac?identifier=drone1337&command=
```

Listing 1.1: URL - Test 1



Figur 1.1: Postman - Test 1

Resultatet viser, at serveren kan returnerer, en korrekt formateret besked med en statuskode: 200 OK.

1.1.2 Test 2 - Seneste besked fra en applikation

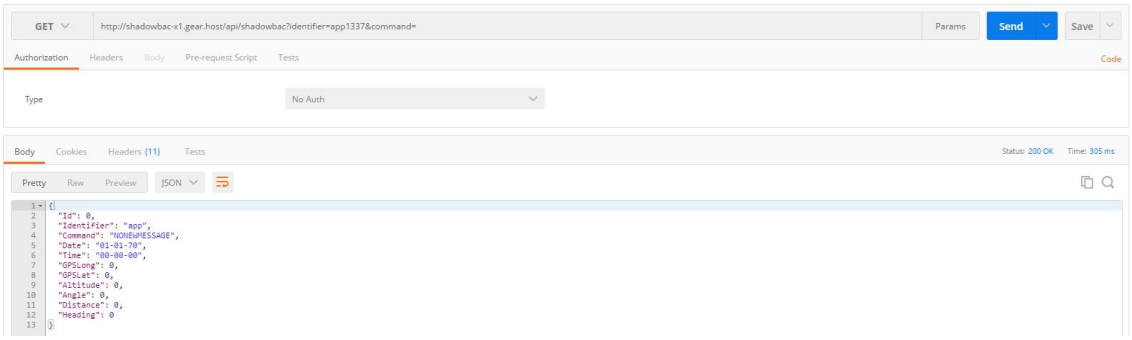
Formål
Formålet med denne test er at verificere, at serveren kan returnere den seneste besked fra en applikation med en given identifier.

Test forløb
For at sikre en ordenlig test skal serveren være oppe og køre, og minimum en besked fra applikationen skal ligge i databasen.

Gennem programmet, Postman, sendes en HTTP GET-besked til URL nr. 2. URL'en skal modificeres med den korrekte identifier og en tom string som command. URL'en brugt i testen ses i Listing 1.2.

```
http://shadowbac-x1.gear.host/api/shadowbac?identifier=app1337&command=
```

Listing 1.2: URL - Test 2



Figur 1.2: Postman - Test 2

Resultatet viser, at serveren returnerer, en korrekt formateret besked med en statuskode: 200 OK. Hvis beskeden, der er returneret, indeholder en command med teksten "NONEWMESSAGE" er dette, fordi serveren ikke har modtaget en besked fra applikationen fornylig. En besked returneres kun, hvis denne ikke er ældre end 20 sekunder.

1.1.3 Test 3 - Seneste besked fra en drone med en specifik kommando

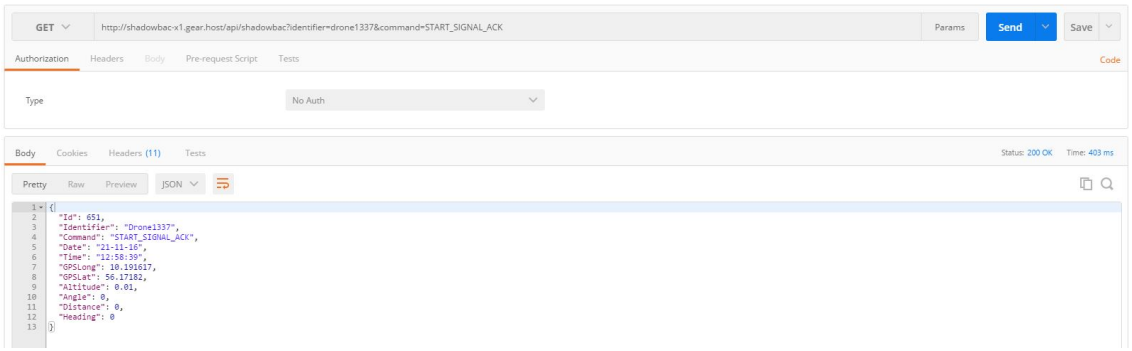
Formål
Formålet med denne test er at verificere, at serveren kan returnere den seneste besked fra en drone med en given identifier og en given kommando.

Test forløb
For at sikre en ordenlig test, skal serveren være aktiv, og minimum en besked fra dronen skal ligge i databasen med en kommando "START_SIGNAL_ACK".

Gennem programmet, Postman, sendes en HTTP GET-besked til URL nr. 3. URL'en skal modificeres med den korrekte identifier og en korrekt command. URL'en brugt i testen ses i Listing 1.3.

```
http://shadowbac-x1.gear.host/api/shadowbac?identifier=drone1337&command=START_SIGNAL_ACK
```

Listing 1.3: URL - Test 3



Figur 1.3: Postman - Test 3

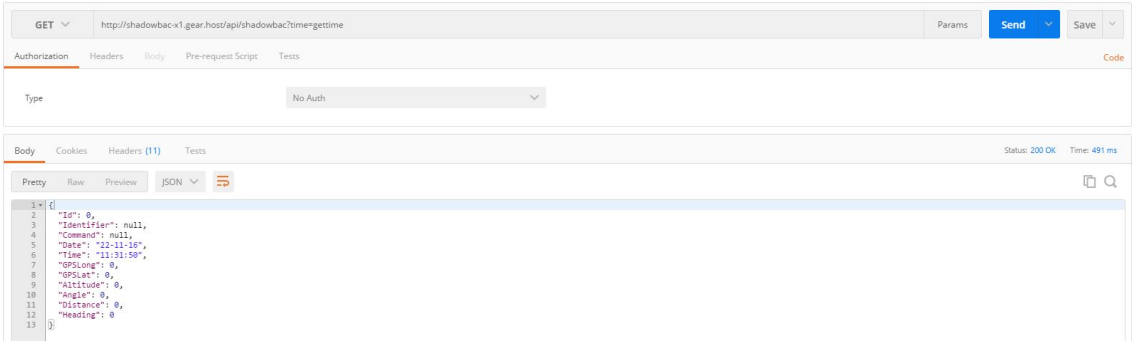
Resultatet viser, at serveren returnerer, en korrekt formateret besked med den korrekte kommando og en responskode: 200 OK.

1.1.4 Test 4 - Serverens tid og data

Formål
Formålet med denne test er, at verificere at serveren kan returnere en besked indeholdende tid og data.

Test forløb
For at sikre en ordenlig test, skal serveren være oppe og køre.

Gennem programmet, Postman, sendes en HTTP GET-besked til URL nr. 1.



Figur 1.4: Postman - Test 4

Resultatet viser, at serveren returnerer, en korrekt formateret besked med UTC tid og data angivet og en responskode: 200 OK.

1.1.5 Test 5 - Modtage en besked fra en drone

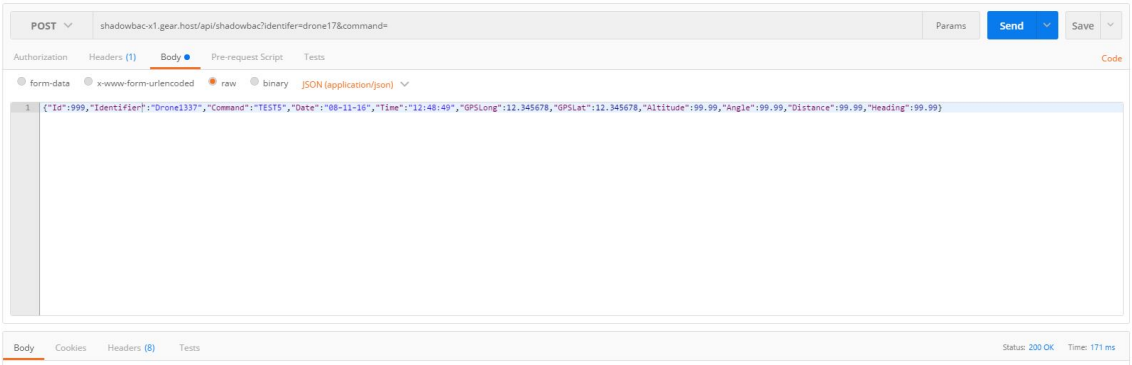
Formål
Formålet med denne test er, at verificere at serveren kan modtage en besked fra dronen og gemme denne i databasen.

Test forløb
For at sikre en ordenlig test, skal serveren være oppe og køre.

Gennem programmet, Postman, sendes en HTTP POST-besked til URL nr. 4. Body'en skal indholde følgende JSON-string, som ses i Listing 1.4. Under fanen Body vælges "raw" og i dropdown menuen vælges "JSON(application/json)".

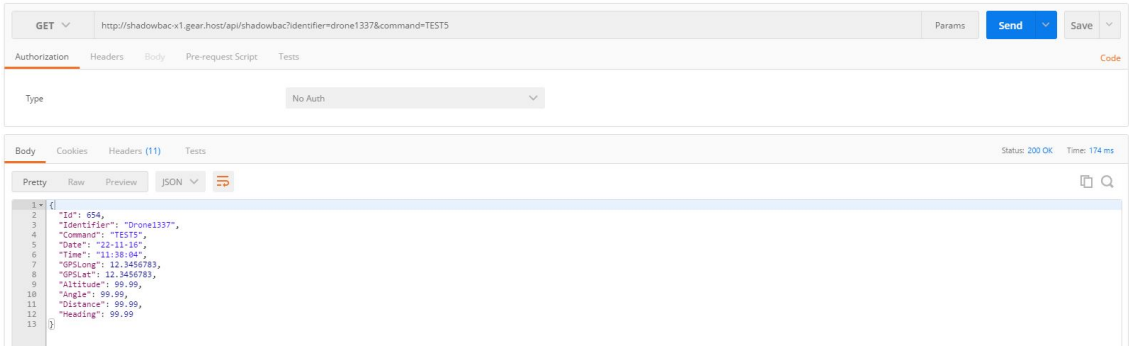
```
{"Id":999," Identifier ":"Drone1337","Command":"TEST5","Date":"08−11−16",
"Time":"12:48:49","GPSLong":12.345678,"GPSLat":12.345678,"Altitude":99.99,"Angle":99.99,
"Distance":99.99,"Heading":99.99}
```

Listing 1.4: URL - Test 5



Figur 1.5: Postman - Test 5

Resultatet viser en responskode: 200 OK. Dette kan verificeres ved at køre test 3 igen, men med kommando "TEST5" istedet for "START_SIGNAL_ACK". Dette ses på figur 1.6.



Figur 1.6: Postman - Test 5

Resultatet viser, at serveren har gemt beskeden i databasen og kan hente den frem igen.

1.1.6 Test 6 - Modtage en besked fra en applikation

Formål

Formålet med denne test er, at verificere at serveren kan modtage en besked fra applikationen og gemme denne i databasen.

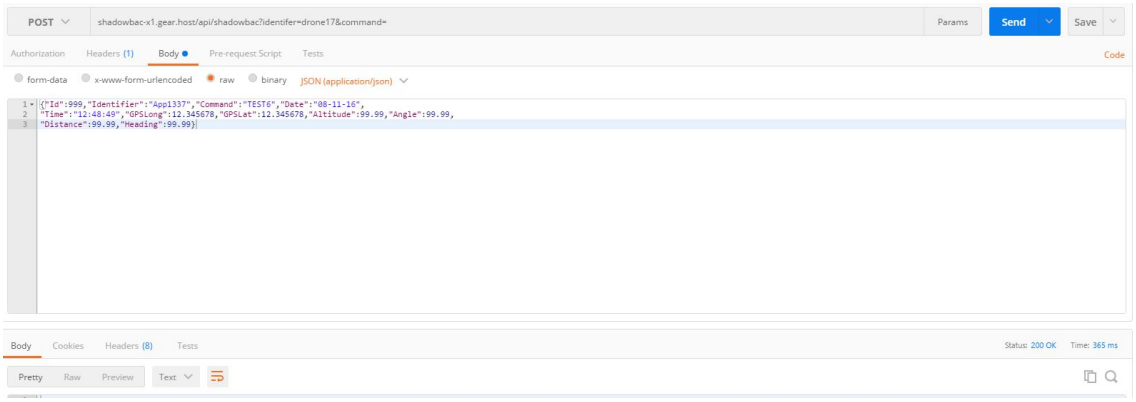
Test forløb

For at sikre en ordenlig test, skal serveren være oppe og køre.

Gennem programmet, Postman, sendes en HTTP POST besked til URL nr. 4. Body'en skal indholde følgende JSON-string som ses i Listing 1.5. Under fanen Body vælges "raw" og i dropdown menuen vælges "JSON(application/json)".

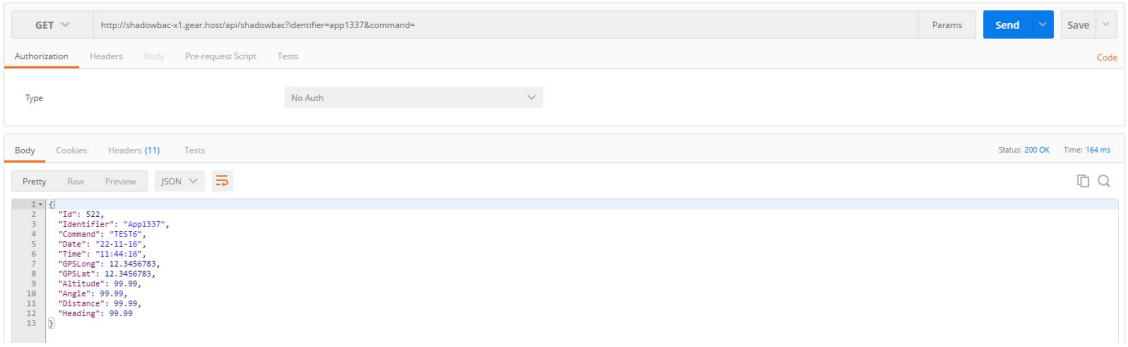
```
{"Id":999," Identifier ":"App1337","Command":"TEST6","Date":"08−11−16",
"Time":"12:48:49","GPSLong":12.345678,"GPSLat":12.345678,"Altitude":99.99,"Angle":99.99,
"Distance":99.99,"Heading":99.99}
```

Listing 1.5: URL - Test 6



Figur 1.7: Postman - Test 6

Resultatet viser en responskode: 200 OK. Dette kan verificeres ved at køre test 2 igen, men dette skal ske inden for 20 sekunder af denne test. Dette ses på figur 1.8.



Figur 1.8: Postman - Test 6

Resultatet viser, at serveren har gemt beskeden i databasen og kan hente denne frem igen.

1.2 Modultest - Drone

I dette afsnit vil modultesten af dronen være beskrevet. Modultesten sikrer, at dronen virker optimalt. Dronens modultest er opdelt i hardware og software tests.

1.2.1 Hardware

I dette afsnit vil modultesten af dronens hardware fremgå.

1.2.1.1 Interface print

Dette print er testet, ved at tilslutte en stelforbindelse til printets ground-pin. Herefter er alle LED'erne testet med en 3.3V-forbindelse, ved at tilslutte hver enkel LED-pin til 3.3V-forbindelse. RGB-dioden kan testes ved at forbinde de 3 RGB-pins til en eller flere 3.3V-forbindelser. Derved kan flere forskellige farvemuligheder også testes.

1.2.1.2 Stepmotor print

Dette print er testet ved, at tilslutte printet til dronens Raspberry Pi. Gennem testkode er Raspberry Pi'ens pins sat høj en af gangen og udgangen på den tilhørende MOSFET tjekket med et multimeter.

Når MOSFET'ens gate udsættes for et 3.3V signal fra Raspberry Pi'en, åbner den op mellem source og drain. Spændingen på drain falder fra 5V til 0V, da strømmen nu kan løbe gennem stepmotoren, til drain-benet og ud igennem source-benet til stel. På denne måde er alle printets udgange testet, ved at tjekke, at disse falder til 0V, når den tilhørende pin aktiveres.

1.2.2 Software

I dette afsnit vil modultesten af dronens software fremgå. Denne er testet gennem klassen *DroneClassTester*, som indeholder test kode til dronens forskellige klasser.

1.2.2.1 Communicator

Communicator-klassen er en klasse, som håndterer kommunikationen på dronen. Denne benytter blot underklasser til at håndtere kommunikationen. Når underklasserne er testet, er *Communicator*-klassen også testet. *Communicator*'en benytter følgende underklasser:

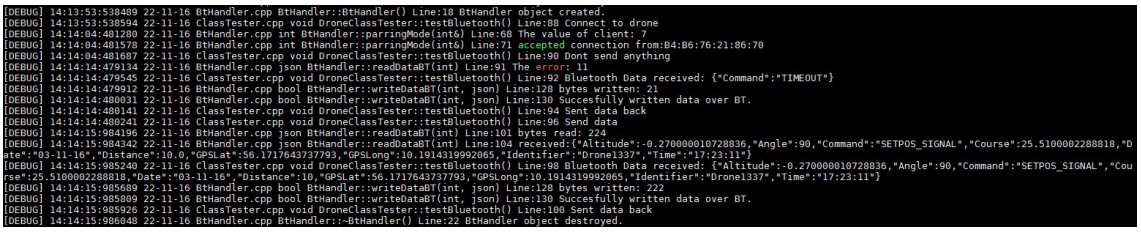
- *BtHandler*
- *ISPHandler*
- *GPSHandler*
- *Interface*

Testen af de forskellige klasser er forklaret i de efterfølgende afsnit.

1.2.2.2 BtHandler

Formål
Formålet med denne test er, at verificere funktionaliteten af klassen *BtHandler*.

Test forløb
For at teste denne klasse oprettes en instans af *DroneClassTester*, og funktionen *testBluetooth* kaldes. For at kunne interagere med dronen skal et Bluetooth-terminal program være installeret på en computer eller en telefon. Herefter følges anvisningerne på skærmen.



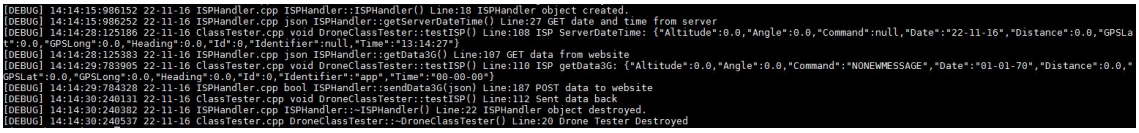
Figur 1.9: BtHandler - Test

Det ses i debug loggen på figur 1.9, at dronen accepterer en forbindelse fra Bluetooth-terminal programmet. Herefter ventes der på at dronen ikke modtager data i 10 sekunder for at teste, at denne kan aktivere timeouten på det blokerende funktionskald. Dette ses, da der kommer en "error 11". Herefter sendes data til dronen og denne svarer tilbage med den samme data.

1.2.2.3 ISPHandler

Formål
Formålet med denne test er, at verificere funktionaliteten af klassen *ISPHandler*.

Test forløb
For at teste denne klasse oprettes en instans af *DroneClassTester*, og funktionen *testISP* kaldes. For at dronen skal kunne nå serveren, må denne ikke have et ethernet kabel i, da dette tvinger Raspberry Pi'en til at benytte denne forbindelse og derved ikke benytter dronens 3G dongle.



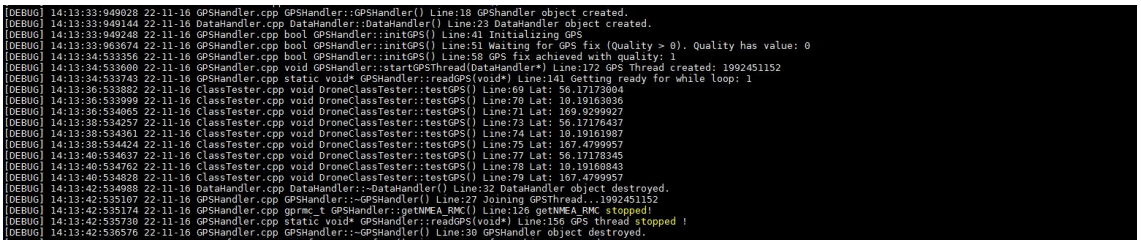
Figur 1.10: ISPHandler - Test

Det ses i debugloggen på figur 1.10, at dronen først spørger serveren om tiden. Svaret modtager denne i en JSON-string. Herefter henter dronen den seneste besked fra applikationen og sender derefter en besked tilbage til serveren.

1.2.2.4 GPSHandler

Formål
Formålet med denne test er at verificere funktionaliteten af klassen *GPSHandler*.

Test forløb
For at teste denne klasse oprettes en instans af *DroneClassTester*, og funktionen *testGPS* kaldes. Raspberry Pi'en skal have GPS-modulet forbundet gennem USB-UART konverteren således, at denne kan nåes gennem den serielle kommunikation.



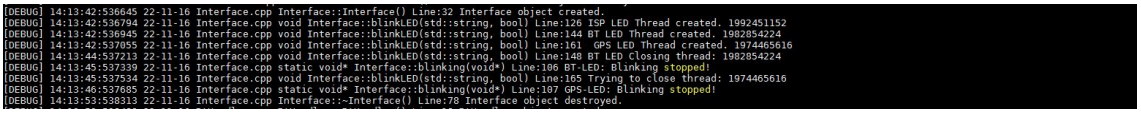
Figur 1.11: GPSHandler - Test

Det ses i debugloggen på figur 1.11, at dronen først initialiserer *GPSHandler*-klassen og derefter begynder at lede efter et GPS-fix. Når dette er modtaget, udskrives værdierne fra GPS'en 3 gange med 2 sekunders mellemrum. Lokationen kan verificeres gennem Google Maps ved at indtaste koordinatet.

1.2.2.5 Interface

Formål
Formålet med denne test er at verificere funktionaliteten af klassen *Interface*.

Test forløb
For at teste denne klasse oprettes en instans af *DroneClassTester*, og funktionen *testInterface* kaldes. Raspberry Pi'en skal være forbundet til interface-printet.



Figur 1.12: Interface - Test

Det ses i debugloggen på figur 1.12, at dronen først opretter de tre tråde til at blinke med LED'erne. Herefter stopper den trådene igen. Denne test ses bedst visuelt på dronens interface. Først begynder GPS-, 3G- og Bluetooth-LED'erne at blinke. Efter 2 sekunder stoppes de igen. Herefter tændes alle LED'erne og batteridioden skifter mellem 3 forskellige farver. Til sidst slukkes alle LED'erne igen.

1.2.2.6 DroneController

DroneController'en benytter underklasser ligesom *Communicator*'en, dog indeholder denne også selv en stor del af dronens samlede funktionalitet. Denne klasse testes gennem systemets integrationstest, da denne indeholder dronens sammensatte funktionalitet. *DroneController*'en indeholder de matematiske funktioner benyttet i dronens regulering. Test af disse er forklaret i de efterfølgende afsnit. DroneControlleren benytter følgende underklasser;

- *DataHandler*
- *StepMotorHandler*
- *SonarHandler*

- *SerialArduinoHandler*

Testen af de forskellige klasser er forklaret i de efterfølgende afsnit.

1.2.2.7 DataHandler

Dronens *DataHandler* står for at gemme alt nuværende data. Alle klasser, som har et behov for at gemme noget data har en reference til *DataHandler*'en. *DataHandler*'en bliver testet implicit i dronens klasser, når de, som benytter en *DataHandler*, testes.

1.2.2.8 StepMotorHandler

Formål
Formålet med denne test er at verificere funktionaliteten af klassen *StepMotorHandler*.

Test forløb
For at teste denne klasse oprettes en instans af *DroneClassTester*, og funktionen *testStepMotor* kaldes. Raspberry Pi'en skal være forbundet til stepmotor printet og stepmotoren ligeledes.

```
Starting JonasExecute.sh[DEBUG] 09:52:49:144539 23-11-16 main.cpp int main(int, char**) Line:36 Starting Program
[DEBUG] 09:52:49:145289 23-11-16 DroneClassTester.cpp DroneClassTester::DroneClassTester() Line:16 Drone Tester Created
[DEBUG] 09:52:49:145701 23-11-16 DroneClassTester.cpp DroneClassTester::DroneClassTester() Line:17 Starting tests
[DEBUG] 09:52:49:146025 23-11-16 DroneClassTester.cpp void DroneClassTester::testStepMotor() Line:122 Starting StepMotorHandler test
[DEBUG] 09:52:49:146368 23-11-16 StepMotorHandler.cpp StepMotorHandler::StepMotorHandler() Line:23 A new stepmotor object has been created.
[DEBUG] 09:52:49:146948 23-11-16 StepMotorHandler.cpp void StepMotorHandler::setCameraAngle(float, float) Line:46 Thread created. 1992397904
[DEBUG] 09:52:49:147388 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:53 Camera turning
[DEBUG] 09:52:49:147461 23-11-16 StepMotorHandler.cpp void StepMotorHandler::turnCameraClockwise() Line:71 in the function C
[DEBUG] 09:53:06:589432 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:65 Camera thread stopped !
[DEBUG] 09:53:09:147704 23-11-16 StepMotorHandler.cpp void StepMotorHandler::setCameraAngle(float, float) Line:46 Thread created. 1982854224
[DEBUG] 09:53:09:147900 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:53 Camera turning
[DEBUG] 09:53:09:148023 23-11-16 StepMotorHandler.cpp void StepMotorHandler::turnCameraCounterClockwise() Line:126 In the function CC
[DEBUG] 09:53:26:632204 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:65 Camera thread stopped !
[DEBUG] 09:53:29:148180 23-11-16 StepMotorHandler.cpp void StepMotorHandler::setCameraAngle(float, float) Line:46 Thread created. 1974465616
[DEBUG] 09:53:29:148368 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:53 Camera turning
[DEBUG] 09:53:29:148492 23-11-16 StepMotorHandler.cpp void StepMotorHandler::turnCameraCounterClockwise() Line:126 In the function CC
[DEBUG] 09:53:37:829823 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:65 Camera thread stopped !
[DEBUG] 09:53:44:148700 23-11-16 StepMotorHandler.cpp void StepMotorHandler::setCameraAngle(float, float) Line:46 Thread created. 1966077008
[DEBUG] 09:53:44:148878 23-11-16 StepMotorHandler.cpp StepMotorHandler::~StepMotorHandler() Line:181 Joining stepmotor thread...
[DEBUG] 09:53:44:148988 23-11-16 StepMotorHandler.cpp StepMotorHandler::~StepMotorHandler() Line:182 Waiting for the stepmotor to reset the position of the camera.
[DEBUG] 09:53:44:149113 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:53 Camera turning
[DEBUG] 09:53:44:149248 23-11-16 StepMotorHandler.cpp void StepMotorHandler::turnCameraClockwise() Line:71 in the function C
[DEBUG] 09:53:52:911654 23-11-16 StepMotorHandler.cpp static void* StepMotorHandler::turnCamera(void*) Line:65 Camera thread stopped !
[DEBUG] 09:53:52:912202 23-11-16 StepMotorHandler.cpp StepMotorHandler::~StepMotorHandler() Line:184 Cameraposition resetted.
[DEBUG] 09:53:52:912340 23-11-16 StepMotorHandler.cpp StepMotorHandler::~StepMotorHandler() Line:185 StepMotor object destroyed.
[DEBUG] 09:53:52:912440 23-11-16 DroneClassTester.cpp DroneClassTester::~DroneClassTester() Line:21 All test has completed successfully
[DEBUG] 09:53:52:912548 23-11-16 DroneClassTester.cpp DroneClassTester::~DroneClassTester() Line:22 Drone Tester Destroyed
```

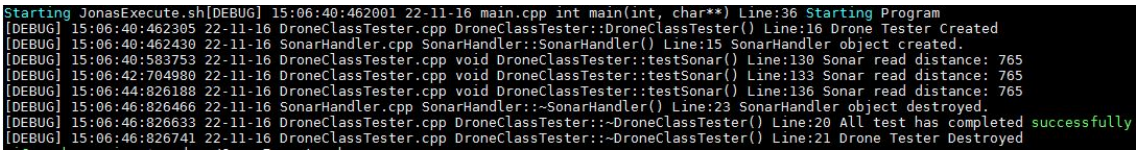
Figur 1.13: StepMotorHandler - Test

Det ses i debugloggen på figur 1.13, at kameraet først kører med uret, da denne i testkoden skal dreje kameraet til vinkel 180°. Herefter drejes kameraet tilbage til vinkel 0°og herefter ud i -90°. Inden instansen nedlægges nulstilles kameravinklen. Denne test skal også verificeres visuelt.

1.2.2.9 SonarHandler

Formål
Formålet med denne test er at verificere funktionaliteten af klassen *SonarHandler*.

Test forløb
For at teste denne klasse oprettes en instans af *DroneClassTester*, og funktionen *testSonar* kaldes. Raspberry Pi'en skal være forbundet til sonarsensoren. Sonarsensoren må ikke være blokeret, da den mindste afstand der kan måles er 20 cm.



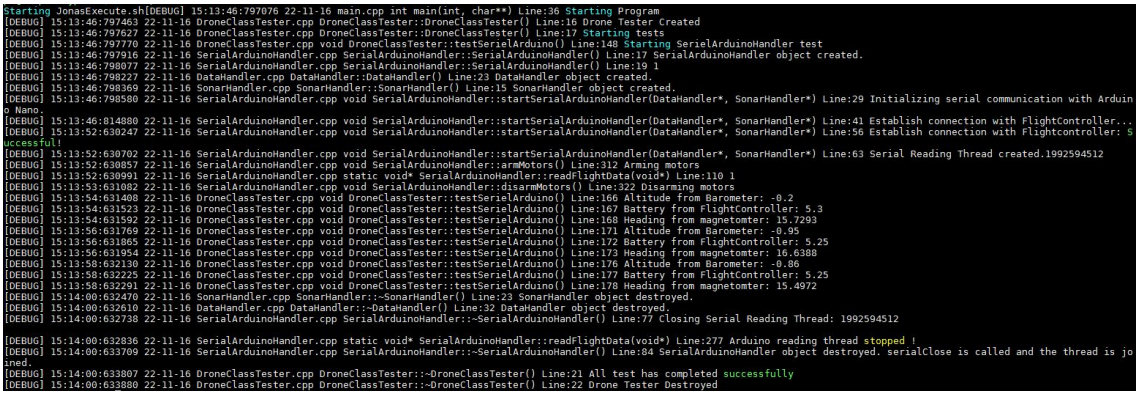
Figur 1.14: SonarHandler - Test

Det ses i debugloggen på figur 1.14, at dronen læser en værdi fra sonarsensoren. Dette gør testen 3 gange med et interval på 2 sekunder.

1.2.2.10 SerialArduinoHandler

Formål
Formålet med denne test er at verificere funktionaliteten af klassen *SerialArduinoHandler*.

Test forløb
For at teste denne klasse oprettes en instans af *DroneClassTester*, og funktionen *testSerialArduino* kaldes. Raspberry Pi'en skal være forbundet til FlightController'en og til Arduino Nano'en. Begge gennem USB kabler.



Figur 1.15: SerialArduinoHandler - Test

Det ses i debugloggen på figur 1.15, at dronen opretter instanserne, som benyttes i testen. Herefter oprettes der en forbindelse til FlightController'en, og tråden til at læse data oprettes. Motorerne bliver herefter tændt og slukket. Herefter udlæses data fra *DataHandler*'en, som *SerialArduinoHandleren* har lagt i denne. Til slut nedlægges alle instanser igen og tråden stoppes.

1.2.3 Matematiske funktioner

Her vil de forskellige matematiske funktioner, der er brugt i koden, blive testet igennem. De matematiske funktioner er udledt fra eksempler på internettet, men testes for at sikre at de virker efter hensigten.

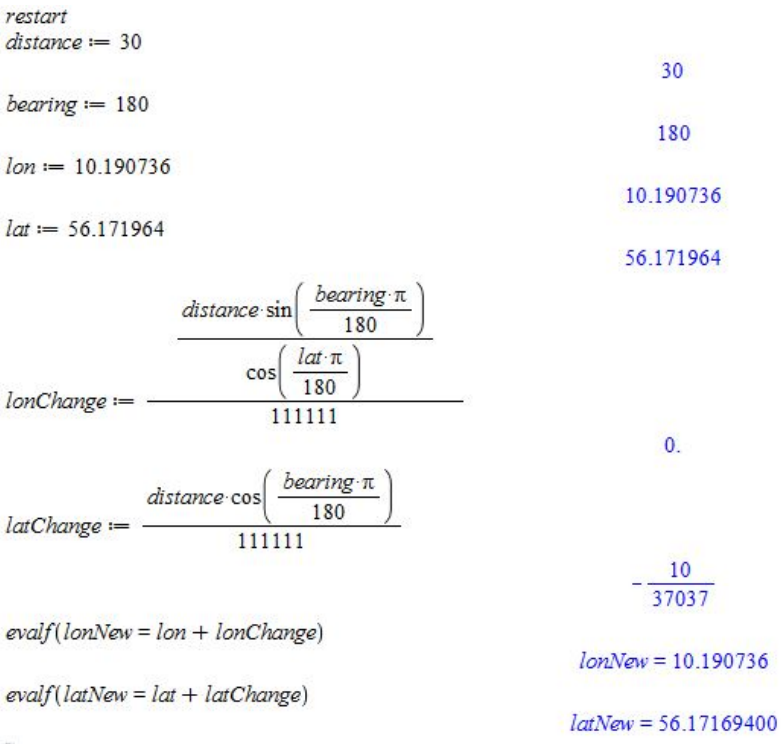
1.2.3.1 DisplacementLocation

Funktionen *DisplacementLocation* er implementeret i koden således:

```
1  void DroneController::displacementLocation(float &longitude, float &latitude, float distance, float bearing) {
2      CLOG(DEBUG, "drone") << "DisplacementLocation";
3      CLOG(DEBUG, "drone") << setprecision(10) << "Lat: " << latitude;
4      CLOG(DEBUG, "drone") << " Long: " << longitude;
5      CLOG(DEBUG, "drone") << " Distance: " << distance;
6      CLOG(DEBUG, "drone") << " Heading: " << bearing;
7
8      longitude = longitude + ((distance * sin(bearing * PI / 180) / cos(latitude * PI / 180)) / 111111);
9      latitude = latitude + (distance * cos(bearing * PI / 180) / 111111);
10     CLOG(DEBUG, "drone") << setprecision(10) << "New Lat: " << latitude << " New Long: " <<
           longitude;
11 }
```

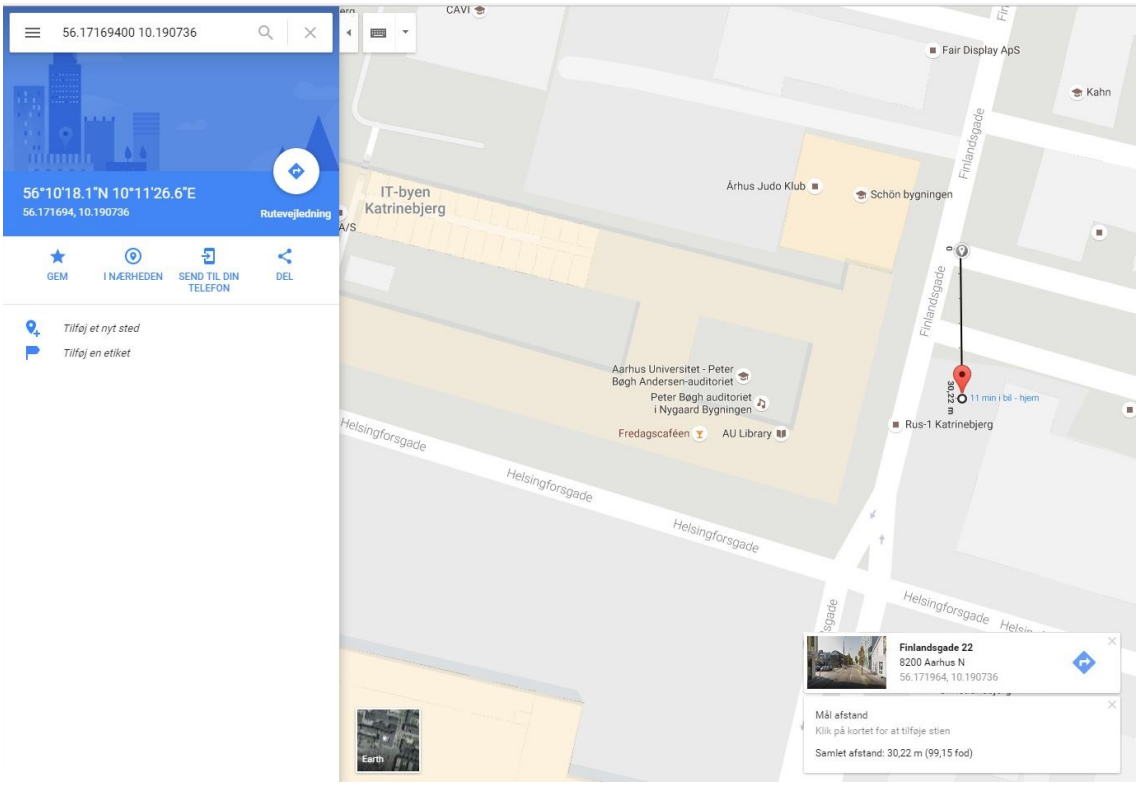
Listing 1.6: DisplacementLocation

For at verificere at denne returnerer et acceptabelt resultat, er funktionen blevet testet. For at teste funktionen, er denne skrevet over i det matematiske værktøj, Maple 2016.



Figur 1.16: DisplacementLocation i Maple 2016

På figur 1.16 er funktionen benyttet til at beregne et nyt punkt der ligger 30 meter fra GPS-koordinatet; 56.171964°, 10.190736° i en kurs af 180° i forhold til nord.
Dette giver et nyt GPS-koordinat; 56.17169400°, 10.190736°. For at tjekke om dette punkt er rigtigt, er Google Maps benyttet. Google Maps benyttes til at afmåle afstanden imellem de to punkter. Derved kan det verificeres at funktionen virker og giver et acceptabelt resultat. Det antages, at Google Maps er præcist og viser en tilnærmelsesvis præcis afstand. Det oprindelige GPS-koordinat; 56.171964°, 10.190736° og det nye GPS-koordinat; 56.17169400°, 10.190736° indsættes i Google Maps, og afstanden måles.



Figur 1.17: Måling af afstand i Google Maps

På figur 1.17 ses det, at afstanden imellem det oprindelige GPS-koordinat; 56.171964°, 10.190736° og det nye GPS-koordinat; 56.17169400°, 10.190736° er 30.22 meter. Det ses, at kursen imellem det oprindelige GPS-koordinat og det nye GPS-koordinat er 180°. Det kan derfor konkluderes, at funktionen virker efter hensigten.

For at verificere at funktionen er implementeret rigtigt i koden, og at koden giver det samme output som Maple 2016, køres koden med de samme parametre, som er brugt i Maple 2016.

Dette giver følgende output:

```
Starting JonasExecute.sh[DEBUG] 10:52:58:216766 23-11-16 main.cpp int main(int, char**) Line:36 Starting Program
[DEBUG] 10:52:58:217077 23-11-16 DroneClassTester.cpp DroneClassTester::DroneClassTester() Line:17 Drone Tester Created
[DEBUG] 10:52:58:217204 23-11-16 DroneClassTester.cpp DroneClassTester::DroneClassTester() Line:18 Starting tests
[DEBUG] 10:52:58:217307 23-11-16 DroneClassTester.cpp void DroneClassTester::testDisplacementLocation(float, float, float, float) Line:184 Starting DisplacementLocation test
[DEBUG] 10:52:58:217586 23-11-16 DroneClassTester.cpp void DroneClassTester::testDisplacementLocation(float, float, float, float) Line:185 Lat: 56.17196274 Long: 10.19073582 Distance: 30 Headin
g: 180
[DEBUG] 10:52:58:217917 23-11-16 DroneClassTester.cpp void DroneClassTester::testDisplacementLocation(float, float, float, float) Line:188 New Lat: 56.17169189 New Long: 10.19073582
[DEBUG] 10:52:58:218031 23-11-16 DroneClassTester.cpp DroneClassTester::~DroneClassTester() Line:22 All test has completed successfully
[DEBUG] 10:52:58:218138 23-11-16 DroneClassTester.cpp DroneClassTester::~DroneClassTester() Line:23 Drone Tester Destroyed
```

Figur 1.18: Output fra test af DisplacementLocation

Det ses på figur 1.18 at det nye GPS-koordinat bliver beregnet til; 56.17169189°, 10.19073582°. Dette betyder, at der er en afvigelse på det nye GPS-koordinat, som Maple 2016 beregner, og det som koden beregner. For at finde ud af hvor stor betydning denne afvigelse har, beregnes afvigelsen.

$$56.17169400° - 56.17169189° = 0.00000211°$$

$$10.190736° - 10.19073582° = 0.00000018°$$

I henhold til hjemmesiden [2] kan der beregnes en forskel på henholdsvis 137.15mm og 11.7mm.
Forskellen i længder er beregnet med værdier fra [2], hvor det antages at ændringen pr. 0.000001 decimal grader ligger på 65mm og at ændringen ved 0.0000001 decimal grader ligger på 6.5mm.

$$2.11 \cdot 65mm = 137.15mm$$

$$1.8 \cdot 6.5mm = 11.7mm$$

Dette er en meget lille ændring, som højst sandsynligt skyldes forskellen i floating values i Maple 2016 og C++. Denne udregning er derfor acceptabel.

1.2.3.2 DistanceBetweenGPSCoordinates

Funktionen *DistanceBetweenGPSCoordinates* er implementeret i koden således:

```
1  float DroneController::distanceBetweenGPSCoordinates(float lonUser, float latUser, float lonDrone, float
       latDrone) {
2      CLOG(DEBUG, "drone") << "DistanceBetweenGPSCoordinates";
3      CLOG(DEBUG, "drone") << setprecision(10) << "Lat User: " << latUser;
4      CLOG(DEBUG, "drone") << " Long User: " << lonUser;
5      CLOG(DEBUG, "drone") << "Lat Drone: " << latDrone;
6      CLOG(DEBUG, "drone") << " Long Drone: " << lonDrone;
7      float dz, dx, dy, result;
8      lonUser = lonUser - lonDrone;
9      lonUser = lonUser * PI / 180;
10     latUser = latUser * PI / 180;
11     latDrone = latDrone * PI / 180;
12
13     dz = sin(latUser) - sin(latDrone);
14     dx = cos(lonUser) * cos(latUser) - cos(latDrone);
15     dy = sin(lonUser) * cos(latUser);
16     //Returns the distance in meters
17     result = asin(sqrt(dx * dx + dy * dy + dz * dz) / 2) * 2 * EARTHRADIUS * 1000;
18
19     CLOG(DEBUG, "drone") << "Distance is: " << result << "m";
20     return result;
21 }
```

Listing 1.7: DistanceBetweenGPSCoordinates

For at verificere at denne returnerer et acceptabelt resultat, er funktionen blevet testet. For at teste funktionen, er denne skrevet over i det matematiske værktøj, Maple 2016.

$$
\begin{aligned}
&\textit{restart}\\
&\textit{lonUser} := 10.190736:\\
&\textit{latUser} := 56.171964:\\
&\textit{lonDrone} := 10.190736:\\
&\textit{latDrone} := 56.171694:\\
&\textit{EARTHRADIUS} := 6372:\\
&\textit{lonUser} := \textit{lonUser} - \textit{lonDrone}:\\
&\textit{lonUser} := \frac{\textit{lonUser}\cdot\pi}{180}:\\
&\textit{latUser} := \frac{\textit{latUser}\cdot\pi}{180}:\\
&\textit{latDrone} := \frac{\textit{latDrone}\cdot\pi}{180}:\\
&dz := \sin(\textit{latUser}) - \sin(\textit{latDrone}):\\
&dx := \cos(\textit{lonUser})\cdot\cos(\textit{latUser}) - \cos(\textit{latDrone}):\\
&dy := \sin(\textit{lonUser})\cdot\cos(\textit{latUser}):\\
&\textit{distance} := \arcsin\left(\frac{\sqrt{(dx\cdot dx + dy\cdot dy + dz\cdot dz)}}{2}\right)\cdot 2\cdot \textit{EARTHRADIUS}\cdot 1000\\
&\qquad\qquad 30.02944452
\end{aligned}
$$

Figur 1.19: DistanceBetweenTwoCoordinates i Maple 2016

På figur 1.19 er funktionen brugt til at beregne afstanden imellem to GPS-koordinater. GPS-koordinaterne der er brugt er de samme som er brugt i modultesten af DisplacementLocation-funktionen.
DistanceBetweenTwoCoordinates giver en afstand på 30.03 meter. Dette kan verificeres på figur 1.17. Her er den samme afstand målt op til 30.22 meter. Der er altså en lille afvigelse imellem den målte værdi og den beregnede værdi. Dette skyldes formentlig, at målingen i Google Maps, er sat ind i fri hånd.
Det viser dog, at udregningen er præcis nok.

For at verificere, at udregningen i koden også beregner afstanden korrekt, testes koden med de to samme GPS-koordinater. Dette ses på figur 1.20.

```
Starting JonasExecute.sh[DEBUG] 11:47:03:527239 23-11-16 main.cpp int main(int, char**) Line:36 Starting Program
[DEBUG] 11:47:03:527984 23-11-16 DroneClassTester.cpp DroneClassTester::DroneClassTester() Line:17 Drone Tester Created
[DEBUG] 11:47:03:528359 23-11-16 DroneClassTester.cpp DroneClassTester::DroneClassTester() Line:18 Starting tests
[DEBUG] 11:47:03:528679 23-11-16 DroneClassTester.cpp void DroneClassTester::testdistanceBetweenGPSCoordinates(float, float, float, float) Line:193 Starting DistanceBetweenGPSCoordinates test
[DEBUG] 11:47:03:529245 23-11-16 DroneClassTester.cpp void DroneClassTester::testdistanceBetweenGPSCoordinates(float, float, float, float) Line:194 Lat User: 56.17196274 Long User: 10.19073582L
at Drone: 56.17169571 Long Drone: 10.19073582
[DEBUG] 11:47:03:529694 23-11-16 DroneClassTester.cpp void DroneClassTester::testdistanceBetweenGPSCoordinates(float, float, float, float) Line:207 Distance is: 29.91759872m
[DEBUG] 11:47:03:530043 23-11-16 DroneClassTester.cpp DroneClassTester::~DroneClassTester() Line:22 All test has completed successfully
[DEBUG] 11:47:03:530360 23-11-16 DroneClassTester.cpp DroneClassTester::~DroneClassTester() Line:23 Drone Tester Destroyed
```

Figur 1.20: Output fra test af DistanceBetweenGPSCoordinates

Det ses på figur 1.20 at koden beregner afstanden til 29.92 meter. Der er altså igen en lille afvigelse i forhold til både udregningen i Maple 2016, som gav 30.03 meter og målingen i Google Maps, som gav 30.22 meter.

$$30.22m - 29.92m = 0.30m$$

Da fejlen kun er på 0.30m, altså 30 cm, accepteres denne fejl og funktionen benyttes.

1.2.3.3 RadiansBetweenHeadingAndVector

Funktionen *RadiansBetweenHeadingAndVector* er implementeret i koden således:

```
1  float DroneController::radiansBetweenHeadingAndVector(float heading, float lonUser, float latUser, float
       lonDrone, float latDrone) {
2      CLOG(DEBUG, "drone") << "RadiansBetweenHeadingAndVector";
3      CLOG(DEBUG, "drone") << setprecision(10) << "Lat User: " << latUser;
4      CLOG(DEBUG, "drone") << " Long User: " << lonUser;
5      CLOG(DEBUG, "drone") << "Lat Drone: " << latDrone;
6      CLOG(DEBUG, "drone") << " Long Drone: " << lonDrone;
7      CLOG(DEBUG, "drone") << " Heading: " << heading;
8
9      float angle;
10     double a = latDrone * PI / 180;
11     double b = lonDrone * PI / 180;
12     double c = latUser * PI / 180;
13     double d = lonUser * PI / 180;
14
15     double angleCritical = cos(c) * sin(d - b);
16     if (closeToZero(angleCritical))
17         if (c > a)
18             angle = 0;
19         else
20             angle = 180;
21     else {
22         angle = atan2(cos(c) * sin(d - b), sin(c) * cos(a) - sin(a) * cos(c) * cos(d - b));
23         angle = (int) round((angle * 180 / PI + 360)) % 360;
24     }
25
26     angle = angle - heading;
27     if (angle < 0) {
28         angle = angle + 360;
29     }
30     float angleRad = angle * PI / 180;
31
32     CLOG(DEBUG, "drone") << "The angle between drone and point: " << angle;
33     CLOG(DEBUG, "drone") << "The angle between drone and point in radians: " << angleRad;
34
35     return angleRad;
36 }
```

Listing 1.8: RadiansBetweenHeadingAndVector

For at verificere at denne returnerer et acceptabelt resultat, er funktionen blevet testet. For at teste funktionen, er denne skrevet over i det matematiske værktøj, Maple 2016.

$$
\begin{aligned}
&\textit{restart}\\
&\textit{heading} := 60:\\
&\textit{lonUser} := 10.190736:\\
&\textit{latUser} := 56.171964:\\
&\textit{lonDrone} := 10.190736:\\
&\textit{latDrone} := 56.171694:\\
&a := \frac{\textit{latDrone}\cdot\pi}{180}:\\
&b := \frac{\textit{lonDrone}\cdot\pi}{180}:\\
&c := \frac{\textit{latUser}\cdot\pi}{180}:\\
&d := \frac{\textit{lonUser}\cdot\pi}{180}:\\
&\textbf{if } \cos(c)\cdot\sin(d-b)=0 \textbf{ then if } c>a \textbf{ then } \textit{angle} := 0 \textbf{ elif } c\le a \textbf{ then } \textit{angle} := 180 \textbf{ end if else } \textit{angle} :=\\
&\quad\left(\frac{(\arctanh(\cos(c)\cdot\sin(d-b),\sin(c)\cdot\cos(a)-\sin(a)\cdot\cos(c)\cdot\cos(d-b)))\cdot 180}{\pi}+360\right) \textbf{ mod } 360 \textbf{ end if}\\
&\textit{angle} := \textit{angle}-\textit{heading}:\\
&\textbf{if } \textit{angle}<0 \textbf{ then } \textit{angle} := \textit{angle}+360 \textbf{ end if}:\\
&\textit{angleRad} := \frac{\textit{angle}\cdot\pi}{180}:\\
&\textit{evalf}(\textit{angleRad})\\
&\qquad\qquad 5.235987758
\end{aligned}
$$

Figur 1.21: RadiansBetweenHeadingAndVector i Maple 2016

På figur 1.21 er funktionen brugt til at beregne antallet af radianer imellem dronens kurs og den vector der er fra dronens punkt til brugerens ønskede punkt for dronen, med urets retning. Funktionen giver 5.235987758, altså 300°.
Det er de samme GPS-koordinater, der er brugt, som i *DisplacementLocation* og *DistanceBetweenTwoCoordinates*. Derudover er der valgt en kurs på 60°. Da dronens kurs er 60° og kursen for vectoren imellem punkterne er 0°, stemmer det overens med at udregningen, giver 300°, da der er 360° i en cirkel.

For at verificere, at udregningen i koden også beregner vinklen korrekt, testes koden med de to samme GPS-koordinater og den samme kurs. Dette ses på figur 1.22.

```
[DEBUG] 12:38:59:546038 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:219 Starting RadiansBetweenHeadingAndVect
or test
[DEBUG] 12:38:59:546170 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:220 Lat User: 56.17196274
[DEBUG] 12:38:59:546316 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:221  Long User: 10.19073582
[DEBUG] 12:38:59:546444 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:222  Lat Drone: 56.17169571
[DEBUG] 12:38:59:546574 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:223  Long Drone: 10.19073582
[DEBUG] 12:38:59:546696 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:224  Heading: 60
[DEBUG] 12:38:59:546825 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:246 The angle between drone and point: 30
0
[DEBUG] 12:38:59:546950 23-11-16 DroneClassTester.cpp void DroneClassTester::testradiansBetweenHeadingAndVector(float, float, float, float, float) Line:248 The angle between drone and point in
radians: 5.235987663
```

Figur 1.22: Output fra test af RadiansBetweenHeadingAndVector

Det ses på figur 1.22 at koden beregner vinklen til 5.235987663 radianer. Ved at beregne forskellen på disse to, kan afvigelsen bestemmes.

$$5.235987758 - 5.235987663 = 0.000000095$$

Omregnes denne til en vinkel i grader, giver dette

$$\frac{0.000000095 \cdot 180}{\pi} = 0.00000544^{\circ}$$

Afvigelsen er så lille, at den ikke har nogen reel betydning. Derfor kan afvigelsen accepteres i koden. Afvigelsen skyldes formentlig forskellen i floating values i Maple 2016 og C++. Denne udregning er derfor acceptabel.

1.3 Modultest - Applikation

I dette afsnit vil applikationens modultests blive gennemgået. Testene blev udført på en reel Android enhed.
For at få response fra telefonen, blev klassen *Log* [3] brugt, som kan bruges til at sende log beskeder fra mobiltelefonen til computeren. Det blev valgt, at lave debug-logs, da disse bliver kompileret, men ikke kørt runtime, når telefonen ikke er forbundet til en computer. Der er lavet en *TestActivity*, som består af en række knapper, der kan starte de forskellige modultests.

1.3.1 MainActivity

MainActivity er forbindelsen mellem brugergrænsefladen og *CommunicatorService*. For at teste klassen udkommenteres funktionaliteten i *CommunicatorService*. I stedet sendes der med det samme den forventede besked, der ville blive sendt efter et udført arbejde i *CommunicatorService*-klassen. Et eksempel kan ses i listing 1.9.

```
1 public void land(){
2 //        DroneMessage msg = new DroneMessage("LAND_SIGNAL");
3 //        sendMsgToDroneTask = new SendMsgToDroneTask();
4 //        sendMsgToDroneTask.execute(msg);
5     writeToActivity(ErrorCode.DRONE_MSG_SENT);
6 }
```

Listing 1.9: Testkode eksempel

Brugerinterfacet benyttes og det ses, at applikationen ikke crasher, når mobiltelefonen drejes i de forskellige skærme. Interfacet må heller ikke ændre udseende efter en skærm er blevet vendt. Ved at kalde funktionerne i *CommunicatorService*, ses det derudover, at denne startes rigtigt op af Activitien og at der bindes rigtigt til denne.

Testens resultat stemmer overens med forventningerne og det kan konkluderes, at *MainActivity* fungerer som ønsket.

1.3.2 CommunicatorService

Efter det blev verificeret, at *MainActivity* fungerede som forventet, blev *CommunicatorService* testet i forbindelse med denne. For at teste dette, bliver funktionaliteten i *IspHandler* og *BtHandler* udkommenteret og erstattet med testkode. I *readMessage()* oprettes en *ArrayList* [4], som fyldes med *DroneMessage* objekter. Der udlæses fra listen, for at simulere status updates, der i realiteten ville være sendt af dronen. Der verificeres følgende punkter:

- Servicen starter i forgrunden, når *takeoff()* kaldes.

- Servicen stopper med at køre i forgrunden, når *land()* kaldes.
- Verificere med Log beskeder, at Servicen ikke genstarter, når den er i forgrunden og applikationen lukkes ned.
- Verificere med Log beskeder, at Servicen ikke genstarter, når mobiltelefonens skærm låses.
- Status opdateringer sendes korrekte til *MainActivity*.

Alle tests kunne verificere, at *CommunicatorService* klassen fungerede som forventet.

1.3.3 IspHandler

Da *IspHandler*-klassens funktionalitet i store dele består af at lægge data op på serveren, testes klassen vha. af systemets server. Ydermere bliver programmet Postman [1] benyttet, for at simulere svar fra dronen.
For at starte testen, benyttes *TestActivity*. Testen startes flere gange. Første gang testes funktionaliteten, når der ikke er forbindelse til nettet. Dernæst testes funktionaliteten, når applikationen har forbindelse, men hvor der ikke kommer svar fra dronen. Til sidst testes klassen ved at simulere en besked fra dronen. Både funktionerne *writeMessage()* og *readMessage()* bliver brugt i funktionen *connect()*. Hvis der altså kan forbindes til den simulerede drone, virker alle *IspHandler*'s funktioner.
For at verificere om applikationen har sendt en besked, benyttes hjemmesiden `http://shadowbac-x1.gear.host/api/shadowbac?identifier=app1337&command=`.

I den første test forventes det, at *checkConnection()* funktionen returnerer, at der ikke er forbindelse til internettet, og at *connect()* funktionen ikke kan få svar fra serveren.

```
D/DRONE_LOG_UNIT_TEST: Error Code from checkConnection(): NO_INTERNET
D/DRONE_LOG_UNIT_TEST: Error Code from connect(): NO_SERVER
```

Figur 1.23: ISP - Ingen internetforbindelse

Det ses på figur 1.23, at resultatet er som det forventede. I den næste test, tændes mobiltelefonens internet og testen udføres igen. På figur 1.25 ses, at applikationen nu lægger en besked op på serveren.

{"Id":576,"Identifier":"app1337","Command":"START_SIGNAL","Date":"05-12-16","Time":"12:33:09","GPSLong":0.0,"GPSLat":0.0,"Altitude":0.0,"Angle":0.0,"Distance":0.0,"Heading":0.0}

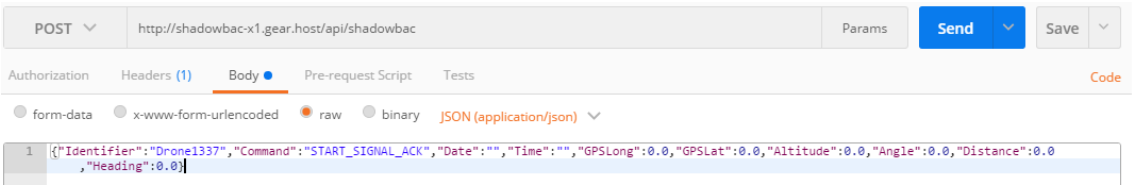
Figur 1.24: ISP - Server output, når der sendes en besked fra applikatioenen

Da der ikke simuleres et svar fra dronen, returnerer *IspHandler*-klassen efter ca 30 sekunder, at den ikke kunne oprette forbindelsen til dronen. Dette kan ses på figur 1.25.

```
13:30:27.868 19385-22089/com.benediktwiese.shadowdrone D/DRONE_LOG_UNIT_TEST: No error occured in checkConnection()
13:31:03.493 19385-22089/com.benediktwiese.shadowdrone D/DRONE_LOG_UNIT_TEST: Error Code from connect(): DRONE_FAILED
```

Figur 1.25: ISP - Log, når der ikke modtages en besked fra dronen

For at udføre den sidste test, simuleres en besked fra dronen ved hjælp af Postman programmet. Opsætningen kan ses på figur 1.26.



Figur 1.26: ISP - Opsætningen af Postman programmet

Denne besked sendes indenfor 30 sekunder efter testen er startet. Det ses nu i loggen, at der blev modtaget en besked. Dette kan ses på figur 1.27.

```
13:35:46.104 19385-19620/com.benediktwiese.shadowdrone D/DRONE_LOG_UNIT_TEST: No error occured in checkConnection()
13:35:49.768 19385-19620/com.benediktwiese.shadowdrone D/DRONE_LOG_UNIT_TEST: Error Code from connect(): DRONE_CONNECTED
```

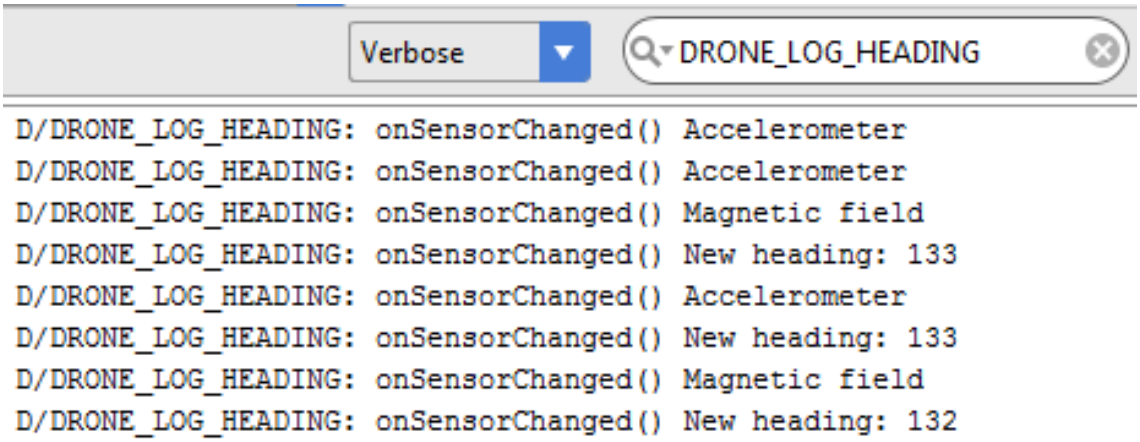
Figur 1.27: ISP - Simuleret forbindelse til dronen

1.3.4 BtHandler

Da klassen ikke fungerer uden en tilhørende drone, testes den i integrationstesten og er ikke uafhængig af dronen og andre klasser.

1.3.5 HeadingSensorHandler

Klassen blev testet gennem *TestActivity*. Det blev testet, om der kan modtages sensordata fra de to sensorer, og om beregningen af brugerens retning giver et acceptabelt resultat. Resultatet af udregningen sammenlignes med retningen i applikationen Compass 360 Pro [5].

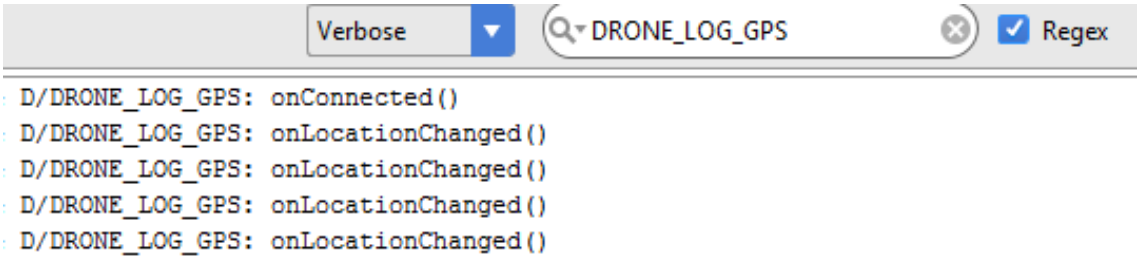


Figur 1.28: Heading - Forbindelsestest

På figur 1.28 ses det, at applikationen får input fra både accelerometer og magnetometeret. Der kan derudover beregnes brugerens retning i forhold til nord, som er tilsvarende til denne der bliver vist i Compass 360 Pro applikationen.

1.3.6 GpsHandler

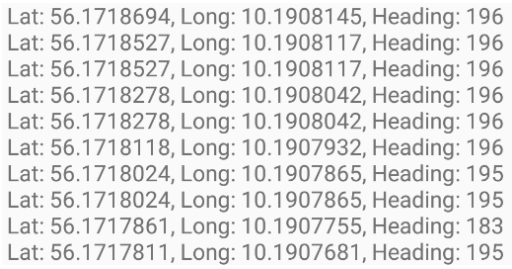
Den første test, der er blevet udført, er hvorvidt applikationen kan forbinde til Google Play Service API'et og om der modtages et GPS koordinat. For at teste dette blev Android Log funktionen benyttet, for at udskrive en besked i callback funktionerne når disse bliver kaldt. For at udføre testen startes *TestActivity* og der trykkes på GPS HANDLER.
Det forventede resultat er, at *onConnect()* bliver kaldt, hvorefter der løbende modtages nye GPS koordinater ved at funktionen *onLocationChanged()* kaldes.



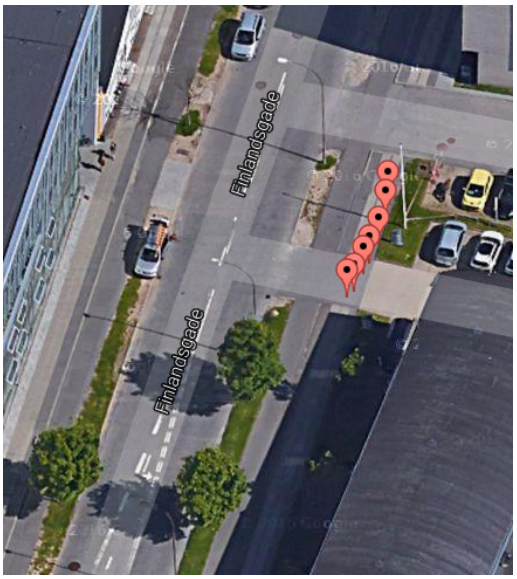
Figur 1.29: GPS - Forbindelsestest

På figur 1.29 ses det, at testens resultat er som forventet, da funktionen *onLocationChanged()* kaldes..

For at teste modulets præcision, blev det valgt at starte GPS testen i *TestActivity*, imens testeren er i bevægelse. De modtagede koordinater bliver løbende vist på skærmen. Efterfølgende plottes koordinaterne i Google Maps på en computer og det ses, om der er afvigelse fra testerens position.



(a) Udskrift på mobiltelefonens skærm



(b) GPS koordinater i Google Maps

Figur 1.30: GPS - Præcisions test

Testen viser, at mobiltelefonens GPS er meget præcis, da de fundne GPS koordinater stemmer overens, med ruten, som testeren bevægede sig i.

1.3.7 DataHandler

For at teste denne klasse, trykkes på knappen "DATA HANDLER" i *TestActivity*.

Testens udførsel
Hver gemt variabel bliver først sat, ved hjælp af set-funktionerne og dernæst udlæst med get-funktionerne. Dernæst testes *resetAll()* funktionen, som sletter alle gemte værdier. Efter funktionen blev kaldt, kaldes get-funktionerne, for at teste default værdierne.
På figur 1.31 ses resultatet, der blev udskrevet i loggen.

```
Drone ID                                Expected value: 1234,          Actual value: 1234
Drone Status                            Expected value: HOVERING,      Actual value: HOVERING
Position (Altitude, angle, distance)    Expected value: 200, 50, 39,   Actual value: [200, 50, 39]
Is position set                         Expected value: true,          Actual value: true
Position (Altitude, angle, distance)    Expected value: 100, 50, 39,   Actual value: [100, 50, 39]
Connection Status                       Expected value: FAILED,        Actual value: FAILED
Reset all values
Drone ID                                Expected value: 0,             Actual value: 0
Drone Status                            Expected value: ON_GROUND,     Actual value: ON_GROUND
Position (Altitude, angle, distance)    Expected value: 1, 0, 1,       Actual value: [1, 0, 1]
Is position set                         Expected value: false,         Actual value: false
Connection Status                       Expected value: NO_CONNECTION, Actual value: NO_CONNECTION
```

Figur 1.31: DataHandler - Test

Det ses, at testresultatet var som det forventede. Klassens funktionalitet er dermed godkendt, da den "Expected value" værdi matcher den "Actual" værdi i figur 1.31.

1.3.8 DroneMessage

Klassen bliver testet for at verificere om objekter kan oprettes med dens forskellige constructor, derunder oprettelsen ud fra en JSON streng. Derudover verificeres det, at et oprettet objekt konverteres korrekt til en JSON streng. For at starte testen trykkes i *TestActivity* på DRONEMESSAGE.


```
Expected: {"Identifier":"","Command":"START_SIGNAL","Date":"","Time":"","GPSLong":56.123456,"GPSLat":10.123424,"Altitude":15.0,"Angle":90.0,"Distance":23.2,"Heading":19.4}
Actual:   {"Identifier":"","Command":"START_SIGNAL","Date":"","Time":"","GPSLong":56.123456,"GPSLat":10.123424,"Altitude":15.0,"Angle":90.0,"Distance":23.2,"Heading":19.4}
Expected: {"Identifier":"app1234","Command":"EXAMPLE_SIGNAL","Date":"","Time":"","GPSLong":0.0,"GPSLat":0.0,"Altitude":0.0,"Angle":0.0,"Distance":0.0,"Heading":0.0}
Actual:   {"Identifier":"app1234","Command":"EXAMPLE_SIGNAL","Date":"","Time":"","GPSLong":0.0,"GPSLat":0.0,"Altitude":0.0,"Angle":0.0,"Distance":0.0,"Heading":0.0}
Expected: {"Identifier":"","Command":"","Date":"","Time":"","GPSLong":0.0,"GPSLat":0.0,"Altitude":0.0,"Angle":0.0,"Distance":0.0,"Heading":0.0}
Actual:   {"Identifier":"","Command":"","Date":"","Time":"","GPSLong":0.0,"GPSLat":0.0,"Altitude":0.0,"Angle":0.0,"Distance":0.0,"Heading":0.0}
Expected: {"Identifier":"","Command":"TAKEOFF_SIGNAL","Date":"","Time":"","GPSLong":0.0,"GPSLat":0.0,"Altitude":22.0,"Angle":0.0,"Distance":0.0,"Heading":0.0}
Actual:   {"Identifier":"","Command":"TAKEOFF_SIGNAL","Date":"","Time":"","GPSLong":0.0,"GPSLat":0.0,"Altitude":22.0,"Angle":0.0,"Distance":0.0,"Heading":0.0}
Expected: {"Identifier":"app89","Command":"START_SIGNAL","Date":"24.12.2016","Time":"20:15:32","GPSLong":52.2829,"GPSLat":13.2344,"Altitude":0.5,"Angle":90.2,"Distance":301293.0,"Heading":51.2}
Actual:   {"Identifier":"app89","Command":"START_SIGNAL","Date":"24.12.2016","Time":"20:15:32","GPSLong":52.2829,"GPSLat":13.2344,"Altitude":0.5,"Angle":90.2,"Distance":301293.0,"Heading":51.2}
Expected: app1, LAND_SIGNAL, 29.12.2016, 23:27:32, 23.2829, 77.2344, 23.5, 92.4, 1.0, 0.0
Actual:   app1, LAND_SIGNAL, 29.12.2016, 23:27:32, 23.2829, 77.2344, 23.5, 92.4, 1.0, 0.0
```


Figur 1.32: DroneMessage - Test

Det ses på figur 1.32 at resultaterne stemmer overens med de forventede værdier da den "Expected value" værdi matcher den "Actual" værdi i figur 1.32.

Integrationstest - Shadow BAC-X1

2

2.1 Integrationstest - Samlet system

I dette afsnit er integrationstesten beskrevet for det samlede system. Disse test er udført for at sikre et korrekt sammenspil mellem systemets blokke.

Projektets udvikling har været opdelt i 3 faser. Hver fase med et hovedansvar baseret på Use Case's.

- Fase 1:
  - Kommunikations opstart både 3G og Bluetooth.
  - Use Case's: 1, 2, 7 og 8.
  - Test fokuspunkter:
    - Kommunikation gennem serveren.
    - Kommunikation gennem Bluetooth.
    - Korrekt kommunikation.
- Fase 2:
  - Automatisk TakeOff og Land.
  - Use Case's: 3, 6, 9 og 11.
  - Test fokuspunkter:
    - Korrekt formateret beskeder.
    - Korrekt opførelse af dronen og applikationen ved TakeOff-signal.
    - Korrekt opførelse af dronen og applikationen ved Land-signal.
- Fase 3:
  - SetPosition og Follow
  - Use Case's: 4, 5 og 10.
  - Test fokuspunkter:
    - Korrekt opførelse af dronen og applikationen ved SetPostion-signal.
    - Korrekt opførelse af dronen og applikationen ved Follow-signal.

2.1.1 Fase 1

Igennem denne fase har fokus været lagt på at kunne etablere et kommunikationslink imellem dronen og applikationen. Dette indebærer både 3G og Bluetooth. Følgende fokuspunkter blev testet gennem udviklingen i denne fase:

Test af kommunikation gennem serveren

Formålet med denne test er at sikre applikationen og dronens samspil igennem serveren. Den eneste funktionalitet fra hardware, dronen behøver for denne test, er adgang til 3G netværket igennem 3G donglen.

Testen af dette fungerer på den måde at applikationen sender en besked til serveren indeholdende et Start-signal. Dette kan verificeres gennem serverens hjemmeside og applikationens logsystem. Herefter er det forventet, at dronen henter den respektive besked ned fra serveren hvilket kan verificeres gennem dronens debug log. Herefter sender dronen et acknowledgement tilbage til serveren, hvilket igen kan verificeres gennem serverens hjemmeside. Applikationen henter beskeden ned og informerer om at forbindelsen er oprettet. Beskeden kan igen verificeres gennem applikationens logsystem. Dronen sender herefter en status besked til serveren hvert 5. sekund. Dette kan verificeres i dronens debuglog. Applikationen henter disse status beskeder, hvilket kan tjekkes i applikationens log.

Test af kommunikation gennem Bluetooth

Denne test skal verificere det samme som testen af kommunikationen gennem serveren. I denne test benyttes blot Bluetooth frem for 3G. Testen fungerer ved at dronen tvinges til at benytte Bluetooth ved at indsætte et ethernet kabel i Raspberry Pi'en. Dette gør, at dronen forsøger at hente netværksdata gennem et kabel uden internetforbindelse. Dette tester, at dronen kan skifte til Bluetooth.

Når dronen befinder sig i Bluetooth mode opretter applikationen forbindelse til dronen. At applikationen forbinder til dronen er verificeret igennem applikationens log system. Når dronen accepterer at applikationen forbinder til den, kan det ses i debugloggen. Det verificeres gennem applikationens log at denne modtager status beskeder fra dronen hvert 5 sekund, ligeledes kan det tjekkes i dronens log at denne sender disse.

Test af korrekt kommunikation

Denne test ligger implicit i de to tidligere afsnit. I denne fase er det vigtigt at JSON pakkerne er formateret korrekt. Kommandoerne i beskederne skal overholder formateringen aftalt i projektet. Begge enheder skal opføre sig korret ved både en succes, men også når eventuelle fejl opstår skal dette håndteres korrekt.

Det testes at dronen automatisk fortsætter til Bluetooth såfremt denne ikke modtager et Start-signal inden for 60 sekunder. Dette testes ved at sende et dummy signal til serveren således at dronen ikke har mulighed for at hente et Start-signal, men kun dummy signalet.

Dronens LED'er skal ligeledes opfører sig korrekt i forhold til 3G og Bluetooth.

2.1.2 Fase 2

Denne fase har haft fokus omkring at kunne sende en TakeOff og Land besked til dronen. Dronen har skulle forsøge på at lette og lande, ved de respektive kommandoer. I denne fase har reguleringen af dronen TakeOff og Land ligeledes haft stort fokus. Testen af fase 1 er igen gennemført på dette tidspunkt for at sikre funktionalitet stadig fungerer.

Korrekt formateret beskeder

Efter denne fase skal det verificeres at beskederne er korrekt formateret og indeholder alle de nødvendige informationer. Dette kan verificeres ved at kigge på beskeder på serveren, i dronens debug log og applikationens log.

Korrekt opførelse af dronen og applikationen ved Takeoff-signal

Præcis som tidligere tests skal det verificeres at dronen kan modtage et TakeOff-signal gennem både serveren og Bluetooth. Dette kan verificeres i de respektive logs. Det verificeres yderligere i dronens debug log at denne modtager beskeden og registerer den højde applikationen ønsker dronen i. Herefter starter dronen sin regulering algoritme for at lette gennem TakeOff tråden. På applikationen tjekkes det at interface'et er opdateret således at brugeren har muligheden for at lande dronen når den er lettet.

For at teste dronens TakeOff algoritme benyttes et stativ hvori dronen kan fast monteres således at den kun kan bevæge sig i y-aksen. Herefter gennemgåes proceduren fra fase 1 og et TakeOff-signal sendes til dronen. Visuelt ses det at dronen regulere motorerne op indtil denne begynder at løfte sig selv i stativet. Det testes at dronen nedjusteret motorkraften såfremt denne kommer over den ønskede højde.

Korrekt opførelse af dronen og applikationen ved Land-signal

Denne test ligner testen for TakeOff-signalet. Denne skal blot have den modsatrettede effekt på dronen. Kommunikationen kan igen verifieres i de respektive logs. Når dronen modtager Land-signalet fra applikationen skal denne lukke TakeOff tråden og starte Land tråden. Dette kan tjekkes i dronens debug log. Herefter nedjusterer dronen motorkraften således at denne begynder et fald i stativet. Når denne er inden for 30cm af gulvet slukker dronen sine motorer. Når dronen er landet skal brugeren have mulighed for at kunne trykke TakeOff på applikationens interface igen.

2.1.3 Fase 3

Denne fase har haft fokus på at kunne sende en SetPosition og Follow besked til dronen. Dronen har skulle forsøge at forblive på et given GPS koordinat eller følge applikationens GPS position. Reguleringsalgoritmerne i projektet har haft stor fokus. Testen af både fase 1 og 2 er igen gennemført for at sikre funktionaliteten stadig fungerer.

Korrekt opførelse af dronen og applikationen ved SetPosition-signal

Denne test skal verificerer at dronen og applikationen har den korrekte opførelse når en SetPosition-signal benyttes i systemet. Ligesom tidligere tests verificeres det at beskederne

er korrekt formateret og at disse stadig kan sendes gennem både 3G og Bluetooth. Når dronen modtager et SetPosition-signal skal det tjekkes at den lukker den givne tråd, som kører og starter SetPosition tråden. På applikationen testes det at de værdier brugeren har valgt i applikationens interface også er de, som er sendt i beskeden. Ligeledes tjekkes værdierne også på dronen i debug loggen for at se om det er de korrekte værdier, som benyttes i SetPosition tråden.

Dronen placeres i et stativ hvor den kan bevæge sig frit dog ikke i y-aksen. På denne måde er det muligt at tjekke i hvilken retning dronen prøver at bevæge sig. Dette bruges til at teste at dronen flyver i den rigtige retning. Dette kan også tjekkes i dronen debug log. Dette testes både gennem 3G og Bluetooth.

Korrekt opførelse af dronen og applikationen ved Follow-signal

Follow-signalet minder utrolig meget om SetPosition funktionaliteten. I dette tilfælde skal dronen blot opdateret GPS koordinaterne fra applikationen således at den hele tiden har et nyt punkt at bevæge sig imod. Værdierne valideres på både dronen og applikationen. Applikationens interface skal igen give brugeren mulighed for at stoppe dronen i at følge. Når dronen modtager et Follow-signal lukker den tråden som kører og starter Follow tråden. Dette verificeres i dronens debug log.

Igen placeres dronen i stativet, som begrænser denne i y-aksen. Det valideres at dronen flyver i forskellige retninger baseret på GPS koordinatet denne modtager fra applikationen. Dette testes ved at sende en række prædefineret GPS koordinater til dronen. Visuelt ses det at dronen flyver i den korrekt retning. Update beskederne fra applikationen kan verificeres på serveren eller i debug logs såfremt testen benytte Bluetooth.

References

[1] Postman. *Postman*. Sep. 2016. URL: https://www.getpostman.com/.

[2] Wikipedia. *Decimal Degrees*. Nov. 2016. URL: https://en.wikipedia.org/wiki/Decimal_degrees.

[3] Android. *Android Log*. Nov. 2016. URL: https://developer.android.com/reference/android/util/Log.html.

[4] Android. *Android ArrayList*. Nov. 2016. URL: https://developer.android.com/reference/java/util/ArrayList.html.

[5] DavidJanuzai. *Compass 360 Pro*. Dec. 2014. URL: https://play.google.com/store/apps/details?id=com.pro.app.compass&hl=da.