

# Bilag O: Kommunikations-system

Kristian Lund

1. juni 2023

# Indhold

<b>1</b>	<b>Indledning</b>	<b>1</b>
<b>2</b>	<b>Design</b>	<b>1</b>
2.1	Valg af Wifi Modul . . . . .	1
2.2	Hardware . . . . .	1
2.3	Software . . . . .	3
2.3.1	Overordnet . . . . .	3
2.3.2	ESP modul . . . . .	4
2.3.3	iPotWifi . . . . .	5
<b>3</b>	<b>Implementering</b>	<b>7</b>
3.1	Software . . . . .	7
3.1.1	ESP . . . . .	7
3.1.2	iPotWifi . . . . .	8
<b>4</b>	<b>Modultest</b>	<b>10</b>
4.1	ESP . . . . .	10
4.2	iPotWifi . . . . .	11

# 1 Indledning

I dette dokument er udviklingsarbejdet for kommunikations-systemet mellem ESP modul og RPI beskrevet. For at undgå overlap i rapporten, inkluderer systemet beskrevet herunder ikke PSoC'en.

Der redegøres først for hardware valg af Wifi-modul, og efterfølgende gennemgås de designvalg der er taget - først for ESP modulets software, dernæst for iPotWifi klassen. Til sidst beskrives implementeringen i samme rækkefølge.

## 2 Design

### 2.1 Valg af Wifi Modul

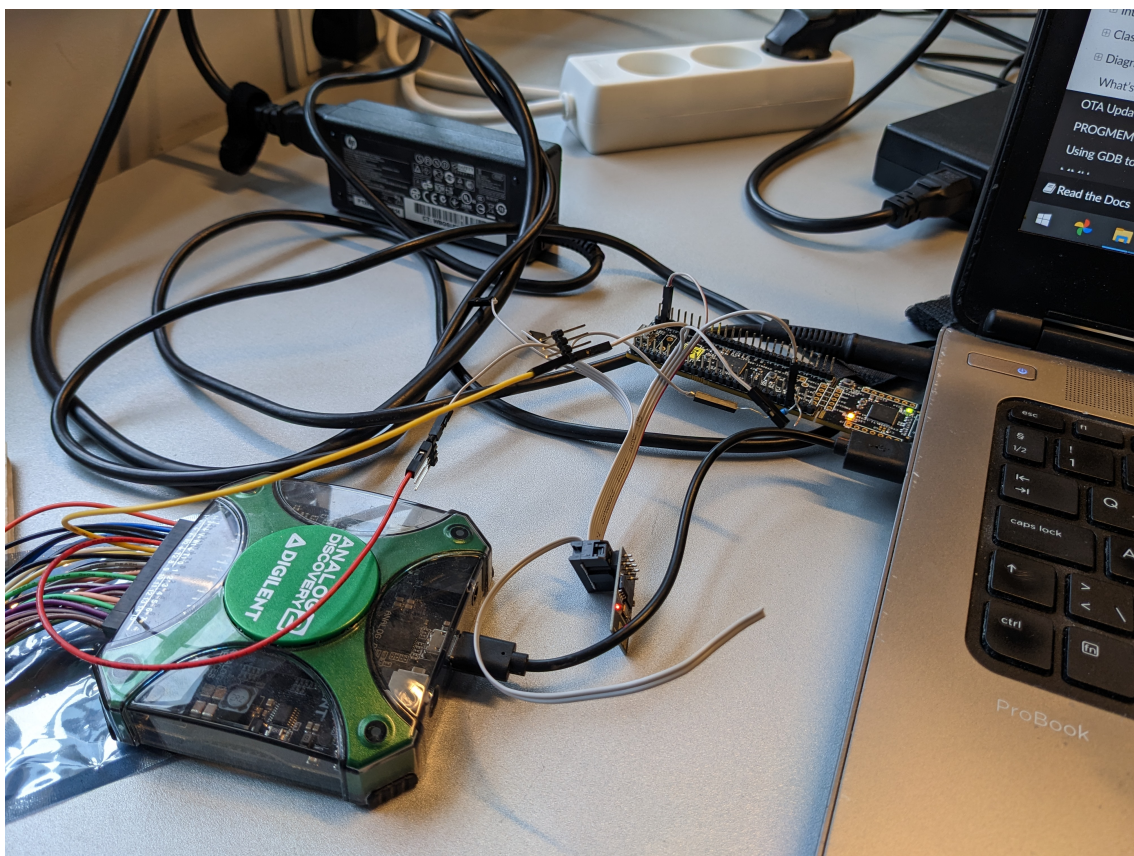
WiFi kommunikation kræver et modul der sluttet til PSoC'en. Denne har indbyggede hardware- og software skabeloner til SPI, I2C og UART, som vi har arbejdet med før - en af disse var klart at foretrække.

Vi valgte derfor ESP8266 modulet. Dels fordi det var tilgængelig i embedded stock, og dels fordi det fra fabrikken understøtter både simpel kommunikation igennem såkaldte AT-kommandoer over UART. Modullet kan desuden programmeres gennem Arduino platformen. Det gav en frihed tidligt i processen, samt noget at falde tilbage på, hvis én ikke virkede.

Sidstnævnte blev nødvendigt undervejs, da den medfølgende firmware tilsyneladende ikke inkluderede fungerende AT kommandoer. Vi skiftede derfor til Arduino IDE, hvilket krævede indkøb af en loader der tillod denne at kommunikere med ESP-modulet (CH340).

### 2.2 Hardware

For at kunne teste ESP8266 med PSoC'en, lavede vi et fladkabel, og koblede den til PSoC og Analog Discovery (AD):



Figur 1: Billede af testopsætning af ESP modul

Forbindelserne er lavet efter følgende tabel:

Kabel pin	ESP Pin nr	ESP-01 Pin navn	Forbundet til
0	1	GND	PSoC GND
1	8	TXD	PSoC 1.6 (RX_ESP)
2	2	GPIO-2	-
3	7	CH_PD	AD W1 (3,3 V)
4	3	GPIO-0	-
5	6	RST	-
6	4	RXD	PSoC 1.7 (TX_ESP)
7	5	VCC	AD V+ supply (3,3 V)

Tabel 1: Tabel over ESP forbindelser i test

Ovenstående kredsløb er senere erstattet af et veroboard, der samler PSoC'en med alle dens eksterne moduler. Der er derfor heller ikke nogen beskrivelse af hardware implementationen herunder, da den - for så vidt den er anderledes end ovenstående - er beskrevet under de tilsvarende hardware afsnit og bilag.

## 2.3 Software

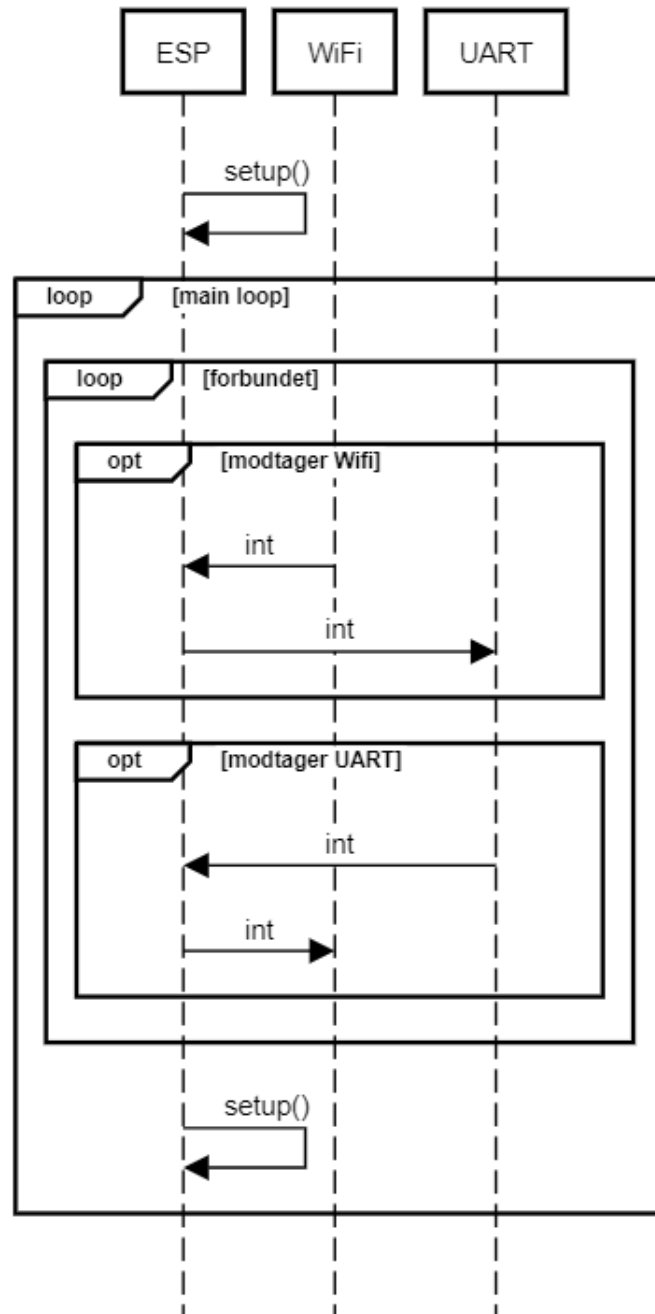
### 2.3.1 Overordnet

Før ESP og RPI klasse kan designes, skal det besluttes hvilken enhed der er server og klient, i hvilke sammenhænge. Vi har taget disse valg ud fra ud fra en betragtning om at RPI'en er vores primære enhed, og de andre dele skal holdes så simple som muligt. Ikke mindst fordi fejlsøgning, gen-kompilering, osv. er langt nemmere fra RPI end fra de to andre enheder vi skal skrive software til.

Vi har valgt at RPI'en skaber Access Point'et, mens ESP'en agerer TCP server. For det første gør det logikken simplere på ESP'en; i det øjeblik denne logger på AP'et som klient, kan den omgående starte en TCP server. Det gør også fejlsøgning simplere: Det er på RPI'en vi kan køre shell-kommandoer og undersøge hvad der er galt, og med denne arkitektur får vi her bekræftigelse i at alt er som det skal være, når det er det. Den vil kun melde korrekt TCP-forbindelse, hvis der er en server klar, på en WiFi-forbindelse til et AP. Det er let fra RPI'ens shell, med en enkelt kommando, at se om dens AP er oppe (og teste med en tredje enhed om denne logges på), om ESP'en er logget på, og om der er svar fra en TCP server på denne.

### 2.3.2 ESP modul

ESP modulet skal egentlig kun fungere som "bro" mellem PSoC og RPI. Den behøver ikke forholde sig til hvad der kommer ind, blot videresende det. Til gengæld skal den gerne forsøge at genskabe forbindelse og TCP-server, hvis den taber forbindelse. Designet kan derfor beskrives med et simpelt sekvensdiagram:

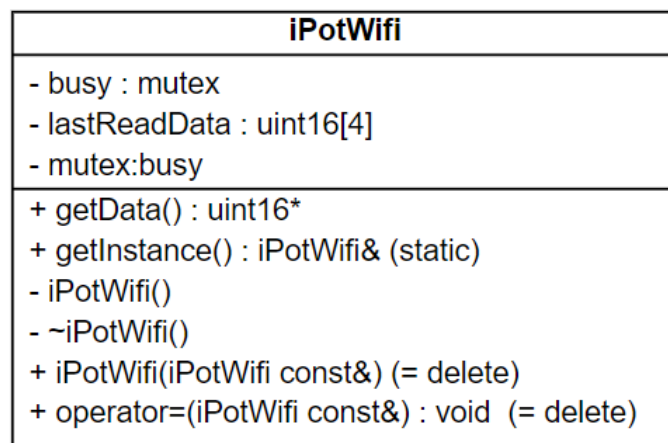


Figur 2: Sekvensdiagram over ESP modul software

`setup()` skal naturligvis stå for at skabe forbindelse til WiFi, og i hvert af de to `opt` segmenter skal den fulde besked modtaget også sendes inden loopet forlades.

### 2.3.3 iPotWifi

Vi vælger at designe iPotWifi funktionaliteten som singleton klasse, der benytter en mutex for at sikre ensartet, korrekt opførsel hvis den blev brugt i flere tråde. Klassen leverer adgang til en række hardware komponenter som vi ikke kan garantere ensartet opførsel fra, hvis der kommer flere getData() anmodninger samtidig. Tilsigtet brug er naturligvis at der kun initialiseres ét objekt, og at getData() ikke bruges concurrent, så man kan sige at det ikke er klassens problem hvis den bruges forkert. Men det er til gengæld klassens ansvar at sikre sig selv - og ikke mindst underliggende hardware - mod forkert brug. Disse krav har ført til følgende klassediagram:



Figur 3: Klassediagram for iPotWifi

Udover boilerplate kode til at sikre singleton status (privat constructor, fjernet copy- og assignment-operator), er der to vigtige funktioner:

**static iPotWifi& getInstance()** Returnerer altid reference til det samme objekt af klassen. Bruges istedet for en normal constructor.

**uint16\* getData()** Låser mutex så det ikke er muligt at to forespørgsler er i gang samtidig, hvis metoden skulle blive brugt i flere tråde. Sender en enkelte byte ('R') som forespørgsel i en TCP pakke til ESP'en og venter på svar. Tager herefter payload modtaget via. TCP, omformer den modtagne data til rette enheder og gemmer dem i lastReadData. Slipper lås af mutex, og returnerer pointer til lastReadData - eller nullptr hvis der ikke kunne skabes forbindelse.

Mutex låsen rundt om resten af getData() metoden sikrer at denne kritiske del af koden ikke kører concurrent. Singleton idiomet sikrer mod at flere objekter oprettes, som så hver især kunne starte hver deres getData() metode med hver deres mutex og dermed omgå formålet med låsen. Vi har valgt at holde implementation af dette simpel, uden brug af conditional variable. Det kan koste ressourcer hvis tilstrækkeligt mange tråde i gang, men det er ikke klassens opgave at sikre mod ressourcspild ved forkert brug - blot at garantere ensartet funktionalitet og især sikre de moduler som den selv kalder, mod samtidige getData kald.

Hvis `getData` ikke kan få forbindelse til serveren, returnerer den en null pointer, som brugere af klassen kan teste efter før de aflæser data.



## 3 Implementering

### 3.1 Software

Herunder gennemgås de vigtigste dele af softwaren til ESP-modulet og iPotWifi klassen, med begrundelser af de valg der er taget. Kildekode præsenteres i udklip, en eller flere udeladte linier markeret med "...". Kommentarer ofte klippet fra. Se mapperne ESP og iPotWifi i fil-bilagene for den komplette kode.

#### 3.1.1 ESP

Vi har implementeret designet af ESP's kode ved hjælp af ESP8266 library'et - det giver en række simple funktioner til at sende og modtage via TCP, og sammen med Arduino frameworket's indbyggede funktioner til UART er der næsten kun løkkerne tilbage at skrive.

Ved opstart køres `setup()`, som vi også benytter som fri funktion senere. Den er ikke medtaget her, da den består næsten udelukkende af boilerplate kode. Logikken fra fra Figur ?? ses til gengæld i følgende:

```
void loop()
{
    WiFiClient client = server.available();

    while(client.connected()){
        //Modtag
        while(client.available()>0) {
            Serial.write(client.read());
        }
        //Send
        while(Serial.available()>0) {
            client.write(Serial.read());
        }
    }
    Serial.print("Disconnected");
    setup();
}
```

Der etableres et `WiFiClient` objekt, som kan bruges til at læse/skrive til den forbundne klient over TCP. Funktionen med dette navn kører Arduino frameworket som et uendeligt loop i main, så `setup()` køres hvis den næst-yderste while-loop undslippes. Inde i dette køres to inderste loops, som hver især sender evt. input fra UART til TCP, og vice versa.

### 3.1.2 iPotWifi

iPotWifi er implementeret som en "Meyers singleton", og koden hertil er ikke medtaget her. Vi har benyttet boost/asio library'et til at håndtere forbindelse og håndtering af TCP. Den interessante del er getData metoden, gennemgået her:

```
uint16_t const * iPotWifi::getData() {  
    busy.lock();  
    try {  
        ...
```

Først låses vores mutex "busy". Den er naturligvis en attribute ved klassen. Hvis et forbindelsesforsøg med boost/asio fejler, kastes en exception; derfor er størstedelen af koden i en try-blok.

Efter lidt boost/asio standardkode til at skabe forbindelse, og socket (s, herunder), køres:

```
    // Skriver R til ESP  
    boost::asio::write(s, boost::asio::buffer(request, request_length));  
  
    // Modtager REPLY_LENGTH bytes  
    uint8_t reply[REPLY_LENGTH];  
    size_t reply_length = boost::asio::read(s, boost::asio::buffer(reply, (size_t)R
```

Først bruges boost/asio's write, så read. Bemærk at read læser præcis REPLY\_LENGTH bytes, her 8. Vi kan ikke bruge eksempelvis null-terminering da der ingen validering er af de sendte tegn, og vores sensorer kunne give et hvilket som helst resultat. De 8 modtagne bytes svarer selvfølgelig til de 4 uint16 som sensorerne har læst, og skal nu omsættes hertil:

```
    uint16_t replyIn16b[4];  
    for (uint8_t valueIt = 0 ; valueIt != 4 ; valueIt++) {  
        uint16_t tempValue = 0;  
        tempValue += reply[2*valueIt] << 8;  
        tempValue += reply[2*valueIt + 1];  
        replyIn16b[valueIt] = tempValue;  
    }
```

For hver plads i et array af uint16, hives to på hinanden følgende værdier ind fra vores modtagne data (reply), "oveni" en tom uint16. Den ene bitshiftet så den udgør de 8 most significant bits på dens plads, den anden ikke.

Da vi under integrationstesten opdagede at det ikke var blevet aftalt hvor de læste sensor værdier skal omsættes til brugbar data, valgte vi at gøre det her i iPotWifi klassen. Det gav mening ift. hvem der var under tidspres, men design-mæssigt havde det nok passet bedre med en separat klasse eller funktion til at lave omregningen, nok på PSoC'en.

To af resultaterne er procent-intervaller, fundet alt efter hvilke cutoff værdier det

modtagne data befinder sig over - et eksempel er givet her:

```
const uint16_t waterLevelCutoffs[6] = { 20200, 18700, 18030, 17740, 17230, 0};
for (uint8_t i = 0; i<6 ; i++) {
    if (replyIn16b[0] > waterLevelCutoffs[i]) {
        lastReadData[1] = i*20;
        break;
    };
}
```

Cutoff værdierne indlæses i et const array, hvorfra de let kan ændres. Herfra kan de også hentes via vores loop variabel. Netop ved at benytte et loop, kan vi også bruge "break" til at stoppe med at checke værdier, så snart vi har fundet et cutoff vores data er højere end. Det fungerer selvfølgelig kun fordi vi checker fra høje til lave cutoff værdier - rækkefølgen i arrayet er vigtig!

De to andre værdier kan udregnes ud fra simple formler, som her:

```
lastReadData[2] = floor( (replyIn16b[2] - 596) / 278.58 );
```

Cutoff-værdierne og formlerne er udledt i bilag M. Til sidst afsluttes med:

```
    }
    catch (std::exception& e) {
        busy.unlock();
        return nullptr;
    }

    busy.unlock();
    return lastReadData;
}
```

Vi fanger vores mulige exceptions, og returnerer her en null pointer - og husker også her at frigive mutexen busy. Uden exception returneres istedet den hentede og omformede data.

## 4 Modultest

I de følgende testes koden til ESP og iPotWifi klassen. Vi har desuden testet det samlede system ved at sætte en ”frisk”RPI og dennes ejer til at følge guiden for hvordan wifi og program startes automatisk, som en slags test af bilag D - WiFi og GUI opstart.

## 4.1 ESP

Modultest af ESP koden skete tidligt i forløbet, sammen med test af PSoC forbindelse og WiFi, som det ses på Figur 1. Testen er kørt med et fundet program til PSoC, og et simpelt program på RPI. Detaljer og opførsel er ikke videre relevant her; pointen med testen er at ESP'en videresendte de fra PSoC'en over UART sendte uint8 til ESP-modulet, som videresendte disse som TCP pakker, modtog svar og sendte tilbage over UART. Dermed er grund-funktionaliteten for ESP opfyldt:

The screenshot shows a terminal window titled "Termite 3.4 (by CompuPhase)". The status bar at the top indicates "COM10 115200 bps, 8N1, no handshake". There are buttons for "Settings", "Clear", "About", and "Close".

```

~!çicoÿiÉZiiI IÊ¹ [JA;ýçzúKie{([Ioia*siFO`S·o(ûEkâ$Siýço÷û9Ú$¿yzÜ¿uí[Biyûç-8ýþ?
yûýýýýûýþýýý·ý/yýýýýýýý~£·yüþyûþýþçý*ýýýýýþýýýýáýý-çý?iýýþ¿iýýiýý)y~SâçûUZ~iy
(iýûýýéý- ^ýž( ^Kýnc¹i_ýë¿ýA~ýç·î(iüý=IOBâýýüiaâ9þKi¥iaAi·woJ?ipİçý~ûüizI y*yYâ¿iñû·yi)
ý-¿?û(ûð³·û$¿ëioiUííiyyOêO(yýüiôóýç(Ikþýþ-oííE~ý(=[19]_-{é)
8ýýþðýEKSyþýýýýýýýþþiçýiýý·oýýiýùEyois<y[00>Hello!
g
Sent command: g
h
Sent command: h
H: This help
devices: get number of connected devices
2:
3:
4:
4: hdev
Sent command: dev
Number of devices: 0
Number of devices: 0devdev
Sent command: dev
Number of devices: 0
Number of devices: 0devhelp
Sent command: help
H: This help
devices: get number of connected devices
2:
3:
4:
4: helpdevices
Sent command: devices
Number of devices: 0
Number of devices: 0devicesdrfgdfg
Sent command: drfgdfg
devices
Sent command: devices
Number of devices: 1
Number of devices: 1devices

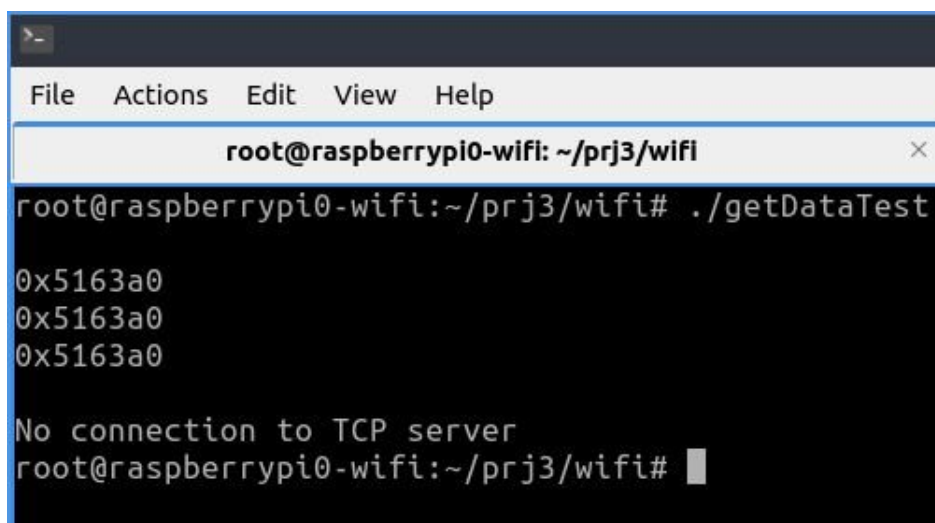
```

Figur 4: Test af ESP

## 4.2 iPotWifi

Til test af iPotWifi er skrevet getDataTest programmet, som er vedlagt som filbilag sammen med klassen. Da vi på dette tidspunkt havde en fungerende PSoC med tilsluttet ESP og sensorer, brugte vi programmet sammen med denne.

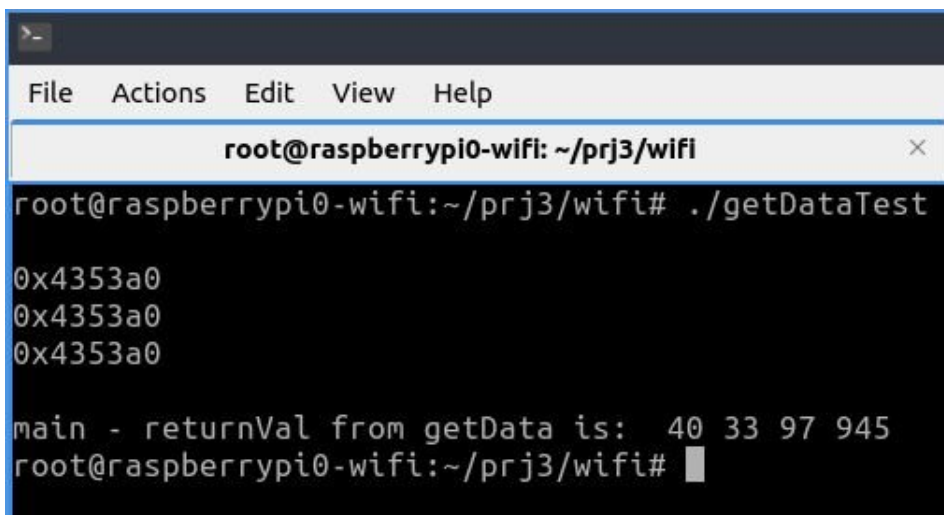
Programmet tester singleton funktionaliteten ved at kalde getInstance() flere gange, og printe den returnerede pointer hver gang - disse skulle selvfølgelig gerne være ens. Desuden checkes om der er forbindelse, og printer en fejlbesked hvis ikke. Hvis der er forbindelse, printes de modtagne værdier. Til nærværende test har vi kørt programmet to gange, med og uden forbindelse:



```
File  Actions  Edit  View  Help
root@raspberrypi0-wifi: ~/prj3/wifi
root@raspberrypi0-wifi:~/prj3/wifi# ./getDataTest
0x5163a0
0x5163a0
0x5163a0

No connection to TCP server
root@raspberrypi0-wifi:~/prj3/wifi#
```

Figur 5: Shell output ved getDataTest uden forbindelse



```
File  Actions  Edit  View  Help
root@raspberrypi0-wifi: ~/prj3/wifi
root@raspberrypi0-wifi:~/prj3/wifi# ./getDataTest
0x4353a0
0x4353a0
0x4353a0

main - returnVal from getData is:  40 33 97 945
root@raspberrypi0-wifi:~/prj3/wifi#
```

Figur 6: Shell output ved getDataTest med forbindelse

Som det ses printes det forventede, og modulet må siges at være i orden.