CHALMERS

Buss till buss kommunikation med WiFi för CAN-nätverk

Trådlös kommunikationslösning för konfigurering och övervakning av litium-jon batterikontrollsystem

Examensarbete inom Data- och Informationsteknik

FREDRIK COGNELL
DAVID WÜSTENHAGEN

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTERBORGS UNIVERSITET
Göteborg, Sverige 2020

EXAMENSARBETE

Buss till buss kommunikation med WiFi för CAN-nätverk

Trådlös kommunikationslösning för konfigurering och övervakning av litium-jon batterikontrollsystem

Fredrik Cognell
David Wüstenhagen

Institutionen för Data- och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET

Göteborg 2020

Buss till buss kommunikation med WiFi för CAN-nätverk

Trådlös kommunikationslösning för konfigurering och övervakning av litium-jon batterikontrollsystem

Fredrik Cognell
David Wüstenhagen

Examinator: Jonas Almström Duregård

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

Omslag:
Den färdiga prototypen vilkens utveckling var syftet med projektet.

Institutionen för Data- och Informationsteknik
Göteborg 2020

Sammanfattning

Micropower Lionova tillverkar batterisystem, en del av systemet är kontrollenheten BMS som kommunicerar med CAN protokollet. För att få möjligheten att i framtiden kommunicera med deras BMSer från ett och samma ställe, en webbserver, oberoende av hur batterisystemen används har de ett behov att göra kommunikationen trådlös. Lösningen som presenteras här utgår från utvecklingskortet Pyboard och inkluderar även kretskortsdesign, kapsling av kretskortet samt mjukvaruutvecklingen som skrivits i MicroPython. Rapporten innehåller också en kort beskrivning av de olika verktyg och protokoll som används samt en förklaring av varför CAN-buss används för kommunikationen med Micropowers BMSer. Den färdiga prototypen har ett 3D-printat skal och en 9-polig D-sub kontakt för inkoppling av CAN-buss, den har tre LEDar för utläsning av status på kommunikation och batterinivå på det interna laddningsbara batteriet som driver hela kretsen. Vid tester har BMSer med framgång konfigurerats, status så som spänning och temperaturer har utläst från modulerna. Men det finns begräsningar i vissa fall där många meddelanden ska behandlas på kort tid.

Nyckelord: CAN, BMS, Micropython, WiFi, litiumjonbatterier

Abstract

Micropower Lionova produce battery systems, a part of that systems is the BMS that communicates with a CAN protocol. In the future Micropower want to have the ability to communicate with theirs BMSs from one location, through a webserver, independently of the battery systems application. But to do this the communication has to be wireless. The solution that is presented here is based from the development card Pyboard and it includes a PCB design, protective covering for the PCB and software development written in MicroPython. The rapport also includes a short description of the different tools and protocols that have been used together with an explanation on why micropower uses CAN communication for theirs BMSs. The completed prototype has a 3D-printed shell and inside the PCB, CAN-bus is connected through a 9-pin D-sub connecter. It has three LEDs to read status, one for communication status and the rest is for the battery status on the rechargeable internal battery. In testing BMSs have successfully been configured, status like voltage and temperature was read from modules. But some problems occur when many messages get processed in a short time.

Keywords: CAN, BMS, Micropython, WiFi

Förord

Denna rapport är ett examensarbete på mekatronikprogrammet för Institutionen för Data- och Informationsteknik på Chalmers Tekniska Högskola. Examensarbetet utfördes på Micropower Lionova AB och beskriver utvecklingen av en prototyp till en hård och mjukvarulösning för CAN-buskommunikation mellan BMS och dator över WiFi.

Särskilt tack riktas till:

Sakib Sistek, handledare på CTH, för vägledning.

Jon Klaesson, handledare på Micropower AB.

Ivan Majdandzic, mjukvaruutvecklare på Micropower AB.

Dag Lundström, teknisk chef på Micropower AB.

INNEHÅLL

TERMINOLOGI

BMS – Battery **M**anagment **S**ystem, ett system som övervakar olika aspekter av ett batteris användning.

CAD – Computer-**A**ided **D**esign

CAN / CAN bus – **C**ontroller **A**rea **N**etwork, är en kommunikationsstandard som används främst I bilar men även inom andra områden.

CRC – **C**yclic **R**edundancy **C**heck

DFU – Device **F**irmware **U**pdate

DLP - **D**igital **L**ight **P**rocessing

GPIO – General **P**urpose **I**nput **O**utput

IC- Integrated **C**ircuit

KiCad - Elektronisk designautomation program, design av kopplingsschema och mönsterkort (PCB).

MicroPython – Micropython är en implementation av python för körning på microcontrollers.

PCB- Printed **C**ircuit **B**oard

Python – Python är ett programeringsspråk.

Ram/Frame – Sekvens av bitar där bitarnas position i ramen definierar dess innebörd.

REPL – Read **E**val **P**rint **L**oop, ett interaktivt användargränssnitt för exekvering av kod.

RTR – **R**emote **F**rame **R**equest, en typ av ram i CAN-kommunikation tänkt att användas för att efterfråga data från andra noder i en CAN-buss.

SMD - Surface-**M**ounted **D**evice

Socket – Socket är en abstraktion för en nätverksanslutning, en socket är bunden till en IP-adress (extern address) och en port (intern address).

SolidWorks – CAD program från Dassault Systemes.

SSID – Service **S**et **I**dentifier, teknisk term för namn på nätverk.

TCP – **T**ransfer **C**ontrol **P**rotocol

UDP – **U**ser **D**atagram **P**rotocol

UMS – USB **M**ass **S**torage

1 INTRODUKTION

I detta avsnitt presenteras projektets syfte, mål och avgränsningar. En kort bakgrund ges till företaget och dess produkt projektet behandlar.

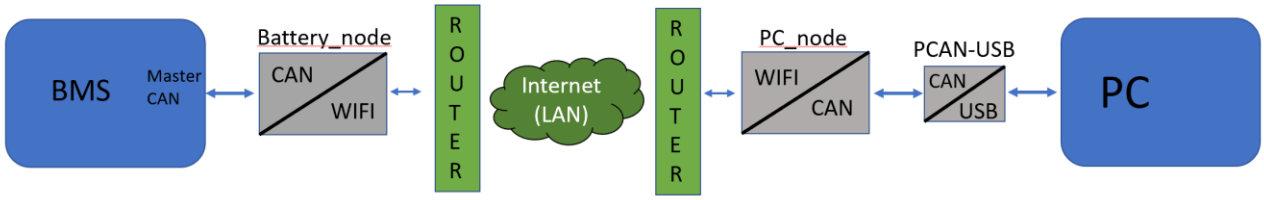
1.1 BAKGRUND

Micropower Lionova AB är ett dotterbolag till Micropower Group AB ett företag som producerar och levererar energilagringslösningar. Micropower Lionova AB utvecklar *LIONBRIX*: ett modulärt batterisystem bestående av litiumjonmoduler på 3.65V som kan kombineras till system med spänningar upp till 1000V [1]. Systemet styrs centralt av en BMS som övervakar och styr modulerna. Genom kommunikation med BMSen kan batterimodulernas status, till exempel spänning avläsas. Inställningar för BMSen angående laddning, urladdning et cetera kan även konfigureras. Kommunikation med BMS sker över CAN-bus. BMSen konfigureras idag på plats med företagets mjukvara ”LIONBRIX CONFIGURATOR TOOL”. För att konfigurera en BMS kopplas en CAN till USB adapter mellan BMSens CAN-buss och en PC.

Vid support och felsökning av LIONBRIX enheter, eller vid konfigurering av BMSen krävs att någon kopplar upp sig på BMSens CAN-bus med en dator som har företagets mjukvara installerad. En kommunikationslösning som inte är kabelbunden till exempel en lösning över ett lokalt trådlöst nätverk, något som ofta redan finns tillgängligt hos kunder, skulle underlätta anslutningen till BMSen. Detta skulle även öka möjligheterna för övervakning av LIONBRIX system vid drift, till exempel på en truck i rörelse. Loggning av data under drift kan även ge företaget insikt om hårdvaran som kan leda till förbättringar av produkten.

1.2 SYFTE

För att enklare ansluta till *LIONBRIX* för konfigurering av BMS, loggning samt övervakning av driftinformation från batterisystem och moduler. Ska en lösning tas fram för att skicka och ta emot CAN-meddelande över lokala trådlösa nätverk. Lösningen skall göra det möjligt att med samma verktyg som redan används av företaget, kunna kommunicera med BMSens CAN-buss på avstånd genom att koppla två moduler på vardera CAN-buss, en till datorn och den andra på BMSens, så som visas i [Figur 1].



Figur 1. Flödesschema över önskad kommunikationslösning.

1.3 MÅL

Över den trådlösa kommunikationen konfigureras och läses data från Micropowers BMSer. Detta genom spegling av CAN-meddelanden på de två fysiskt skilda bussarna. CAN-meddelanden från BMSen är identiska med de ursprungliga meddelandena då dem anländer till PCn och vice versa. När lösningen väl är påslagen och uppkopplad fungerar kommunikation mellan BMS och PC på samma sätt som om en fysisk koppling mellan PC och BMS fanns. lösningen är smidig och enkel att använda. Lösningen fungerar stabilt vid CAN-bus hastigheten 125kbit/sek.

1.4 AVGRÄNSNINGAR

Lösningen används tillsammans med Micropowers konfigueringsverktyg för att kommunicera med BMSen. Lösningen stödjer därför inte nödvändigtvis alla möjliga CAN nätverk, till exempel nätverk med annan baudrate än 125 kBits/s stöds inte. Lösningens kostnad har låg prioritet men får inte anses orimlig, en bedömning som görs av Micropower.

Ingen hänsyns tas till att säkerställa datasäkerheten på kommunikationen, skydd mot dataintrång och försök till att ändra dataströmmen kommer inte implementeras.

1.5 EXISTERANDE TEKNIK

Både CAN och WiFi är inarbetade tekniker och det finns lösningar som sickar CAN-meddelanden över WiFi. Två företag som har produkter för detta är Grid Connect Inc [2] och Kvaser [3], de båda säljer produkter som man kopplar in på en CAN-buss och sedan kan läsa samt skriva data till CAN. Något som skiljer lösningen som presenteras här är att denna lösningen drivs av ett internt batteri, vilket var ett av kraven från Micropower.

2 TEKNISK BAKGRUND

Detta kapitel innehåller förklaringar av tekniska koncept som krävs för att förstå vissa aspekter av projektet och de val som gjordes i implementationen av prototypen.

2.1 TCP/UDP/IP

Vid dataöverföring på Ethernet nätverk paketeras datan i ramar. Ramarna innehåller information som vilken IP-adress och port datan ska till. Internet protocol, IP, innehåller bland annat avsändande och mottagande IP-adress, samt information om vilket transportprotokoll som används. Vid val av transportprotokoll finns det främst två kandidater: Transmission Control Protocol, TCP, och User Datagram Protocol, UDP. Allmänt kan sägas att TCP ger säkrare dataöverföring genom en mängd inbyggda säkerhetsmekanismer som garanterar att datan kommer fram i rätt ordning och att eventuella tappade bytes skickas om. UDP ger generellt högre överföringshastighet genom att utelämna TCPs säkerhetsmekanismer. Det innebär dock att det inte är osannolikt att UDP-meddelanden kommer fram i fel ordning eller inte alls.

2.2 CRC

Vid överföring med UDP kan det hända att data anländer till mottagande socket i fel ordning, flera gånger eller inte alls. Om detta sker blir eventuella ramar datan tillhör förvrängda, och behöver kasseras. För att kontrollera om datan är oskadd kan en kontrollsumma användas. En kontrollsumma är ett värde som tas fram baserat på datan som ska kontrolleras, det kan till exempel vara summan av alla bitar i datan.

Innan datan skickas beräknas kontrollsumman, den skickas sedan tillsammans med datan till mottagande socket. När datan anlänt beräknas kontrollsumman åter och jämförs med den bifogade kontrollsumman. Om summorna skiljer sig vet man säkert att meddelandet är felaktigt och bör kasseras.[1]

Cyclic Redundancy Check, CRC, är en vanlig metod för att beräkna kontrollsummor, i prototypen används CRC-16.

2.3 HÅRDVARA

Viktigast i lösningen är utvecklingskortet pyboard d-series och dess mikrokontroller STM32F722IEK. Utöver pyboarden används också en CAN-transceiver samt en spänningsregulator, laddningskrets och ett batteri.

2.3.1 STM32F722IEK

Mikrokontrollen är kraftfull har, lågströmförbrukning och något som är viktigt för implementationen är att den har en CAN-kontroller. Huvudfunktionerna är som följer:

- 216 MHz Cortex M7 CPU
- 512KiB internt flash ROM och 256KiB internt RAM
- 2MiB externt QSPI flash
- CAN-interface 2.0 A och B, (11-bit samt 29-bit identifierare)

[1] En likvärdig kontrollsumma är ingen garanti för att datan är intakt, det ökar dock chanson att en misslyckad överföring upptäcks avsevärt.

2.3.2 Murata 1DX
Baserat på CYW4343 chipet, ger möjlighet att kommunicera med WiFi 802.11b/g/n och Bluetooth 5.1 BR/EDR/LE. Två typer av antenner används i lösningen tillsammans med Murata 1DX. En som satt monterad på pyboarden, det är en fraktal chip antenn, en liten SMD monterad antenn. Och en större extern antenn med högre förstärkning, MAF94149 som kopplas på en kontakt på pyboarden.

2.3.3 MCP2562
CAN-transceivern MCP2562s syfte är att signalanpassa mellan mikrokontrollern och CAN-bussens hög och låg ledningar. På STM32F722IEK är logiknivå 3.3V (hög eller “etta”) och 0V (låg eller “nolla”) på CAN-bussen är det differensen mellan ledningarna kallade hög och låg som avgör om CAN-bussen ska läsas som logisk etta eller nolla. Här krävs alltså en signalanpassning och IC-kretsen MCP2562 är används för detta.

Spänningsdifferensen V_{diff} mellan ledningarna i CAN-bussen räknas ut av MCP2562 och om de ligger över eller under V_{diff} kommer mikrokontrollen kunna avläsa om bussen är i dominat eller recessivt läge genom att läsa motsvarande 3.3V och 0V på Rx som är kopplad till mikrokontrollen. När mikrokontrollen ska skicka ut på bussen används istället Tx där samma logik används som i mottagning men riktningen är motsatt. Värden på V_{diff} med mer kan hittas i databladet för MCP2562 [4].

Det är viktigt att göra skillnad på CAN-transceivern och kontrollern, kontrollern är mer avancerad och utför det fysiska lagret i CAN-protokollet. I kontrollern sker id-prioriteringar och kontrollbits kontroll, transceivern är bara en signalanpassning.

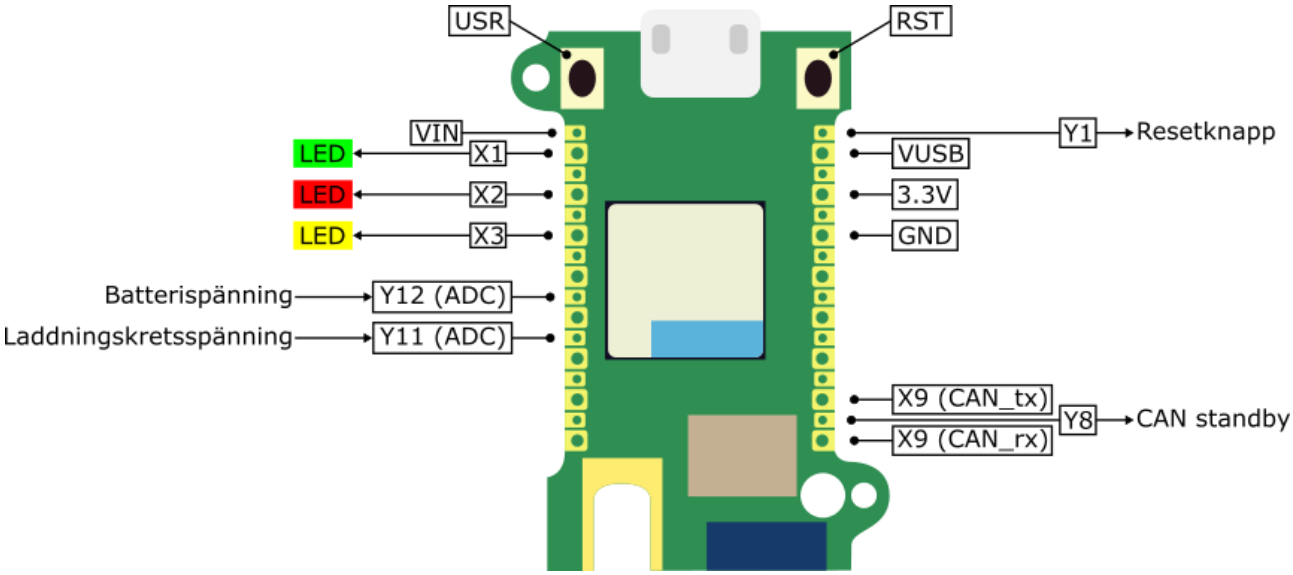
2.3.4 MIC2295
En switchande spänningsregulator som förser CAN-transceivern med en matningsspänning på 5V från ett litium-jonbatteri där spänningen kan variera mellan 3.2V till 4.2V. Eftersom Pyboarden inte har någon 5V utgång och CAN-transceivern kräver en spänning på mellan 4.5 till 5.5V för att drivas valdes detta IC-chip för att den klarar av att levererar den spänningen med den ingående spänningen på batteriet som används.

Detta ger två alternativ för att driva Pyboarden, antingen genom dess VIN som tar emot 3.3V till 4.8V eller genom dess VBUS som tar 4.8V till 5.2V. Batteriet kan direkt driva Pyboarden genom VIN eller genom MIC2295 på VBUS. Här blir strömmen som MIC2295 kan leverera den avgränsande faktorn, så den konfigurerats på kretskortet klarar den av att leverera 500mA.

2.3.5 MIC79050
Då hela kretsen ska vara bärbar och inte får drivas genom CAN-bussen krävs ett batteri och därmed en laddningskrets. Batteriet som valt är standardcellen 18650, de är mycket vanliga litium-jonbatterier. MIC79050 är tillverkad av Microchip och en vanlig applikation för detta IC-chip är laddning av en litium-joncell från en konstant 5V spänningskälla. Spänningskällan som används i kretsen är från en vanlig 5V USB-laddare med en micro-USB typ-B kontakt.

2.3.6 Pyboard

Pyboard är en serie utvecklingskort designade för användning med programmeringsspråket micropython. Prototypen byggdes på PYBD-SF2-W4F2, en pyboard som tillhör produktserien pyboard D-series. PYBD-SF2-W4F2 är baserad på mikrokontrollern STM32F722IEK tillsammans med periferienheter för WiFi-kommunikation [5]. Pyboarden har förutom ett antal GPIO-pinnar en tryckknapp för hårdvarureset kallad ”RST” och en programmerbar tryckknapp kallad ”USR” vilka kan ses i [Figur 2]. Knapparna används även för att försätta pyboarden i DFU-läge.



Figur 2. Förenklad bild av PYBD-SF2-W4F2 med knappar och pinnar som används i prototypen utmärkta.

2.4 Micropython

Micropython är en implementation av Python 3 optimerat för körning på mikrokontroller. Språket inkluderar en del av pythons standardbibliotek för snabb och enkel utveckling av programvara. En stor fördel med micropython är REPL, ett interaktivt sätt att exekvera pythonkod på ett sätt som gör det mycket lätt att testa funktioner som de inbyggda biblioteken för CAN och WiFi innan någon kod skrivs. En annan stor fördel och huvudanledningen till att Micropython används för prototypen är att det finns implementationer av Micropython skrivna till en mängd olika mikrokontrollers och nya tillkommer ständigt. Detta gör att ett eventuellt framtida byte av mikrokontroller kan ske utan större arbete med anpassning av koden.

2.4.1 Att programmera i micropython

Programmering i micropython sker genom att mikrokontrollern ansluts till en dator som en UMS, och gränssnittet mellan datorn och mikrokontrollern fungerar sedan som filsystemet i ett vanligt USB-minne. I filsystemet lägger man sedan pythonscript sparade i textformat med filnamnsändelsen .py. Vid uppstart kommer mikrokontrollern söka efter och exekvera scriptet i filen boot.py och därefter den fil som anges i boot.py.

2.5 Controller Area Network

CAN-bussen består av två ledningar kallade CAN-hög och CAN-låg, signalen tolkas utifrån spänningsdifferensen mellan ledningarna. CAN stödjer hastigheter upp till en megabit per sekund och används i applikationer där snabbhet och säkerhet är viktigt. Kommunikation på bussen sker genom broadcast, varje meddelande skickas till samtliga enheter (noder) som är kopplade på bussen. Meddelandena är försedda med ett ID och ett datafält. ID numret används generellt för att visa vilken typ

av data ett meddelande innehåller och vilken nod som är avsändare. Varje ID måste vara unikt och används även för att ge meddelanden prioritet på bussen. Det finns två typer av ramar som används i CAN-kommunikation, standard och utökade. Namnen syftar till antalet bitar ID-fältet innehåller där, standard har ett 11 bitars ID-fält och utökade har 29 bitar.

2.5.1 Micropowers val att använda CAN-bus

Enligt Ivan Majdandzic, mjukvaruutvecklare på Micropower, används CAN protokollet att kommunicera med Micropowers BMSer på grund av den höga pålitligheten och därmed säkerheten som medförs. Han förklarar att det också beror på att det är en utsprid standard inom den industrin de jobbar med, och att deras kunders system också använder sig av CAN.

CAN introducerades 1986 av Robert Bosch under en Automotive konferens [6], dess ålder är också en av anledningarna att CAN används säger Ivan Majdandzic. Det är välbeprövat och många av de buggar som uppkommer när något nytt utvecklats har nu under en längre tid större chans att bli upptäckta, det gör CAN-kommunikation stabil.

3 GENOMFÖRANDE

Under genomförandets gång har hårdvara och mjukvara utvecklats parallellt. Uppgiften har delats upp i flera mindre projekt som mot slutet kombinerats för att nå en komplett lösning. Att utgå från ett utvecklingskort som Pyboarden där både mjukvara och hårdvara redan är förberedd gör startsträckan kortare och tillät att tidigt börja testa funktioner. Men det är också det som gjorde det svårt att få full kontroll över detaljer mot slutet där optimering för prestanda försvårades då Pyboarden saknar vissa konfigurationer.

3.1 VAL AV UTVECKLINGSPLATTFORM

Det första steget efter att problemet formulerats och en kravspecifikation skrivits och godkänts av MicroPower, var att välja i vilken miljö utvecklingen ska fortsätta i. Här var det viktigaste valet vilken mikrokontroller och programspråk som skulle användas. Micropowers tekniska chef Dag Lundström och mjukvaruutvecklare Ivan Majdandzic rekommenderade Pyboarden med tillhörande Micropython som programspråk. Detta då de använt plattformen innan och att många funktioner finns färdiga för att sätta upp CAN och WiFi kommunikation. Ett annat alternativ som övervägdes var ett ESP32 utvecklingskort, den har WiFi och CAN-kontroller inbyggt, samt möjligheten att programmera i C. Det är ett billigare chip och har också funnits under en längre tid, det finns även mycket information att hitta om olika applikationer på forum och liknande.

Pyboarden valdes som utvecklingsplattform i prototypen på grund av fördelarna med att ha tillgång till REPL och att framtida byten av processor underlättas. Dessutom fanns det redan en uppsättning Pyboards tillgängliga på Micropower.

3.2 UTVECKLINGSMILJÖ

Utvecklingskortets funktioner testades först i REPL. Anslutning till REPL gjordes med terminalemulatorn Tera Term [7] i Windows samt med screen i Linux. Programmeringen skedde sedan i enklare textredigeringsprogram med hjälp av Micropythons dokumentation [8] och källkod [9].

3.3 GRÄNSSNITT

För att användaren ska kunna använda denna lösning har kretsen tre LED lampor med färgen grön, röd och gul. Grön och röd informerar om batteriets status och gul om kommunikationen. Det finns också en återfjädrande tryckknapp samt en switch, tryckknappen initierar en hårdvaruomstart och switchen bryter batteriets koppling till resten av kretsen.

3.4 SKAL

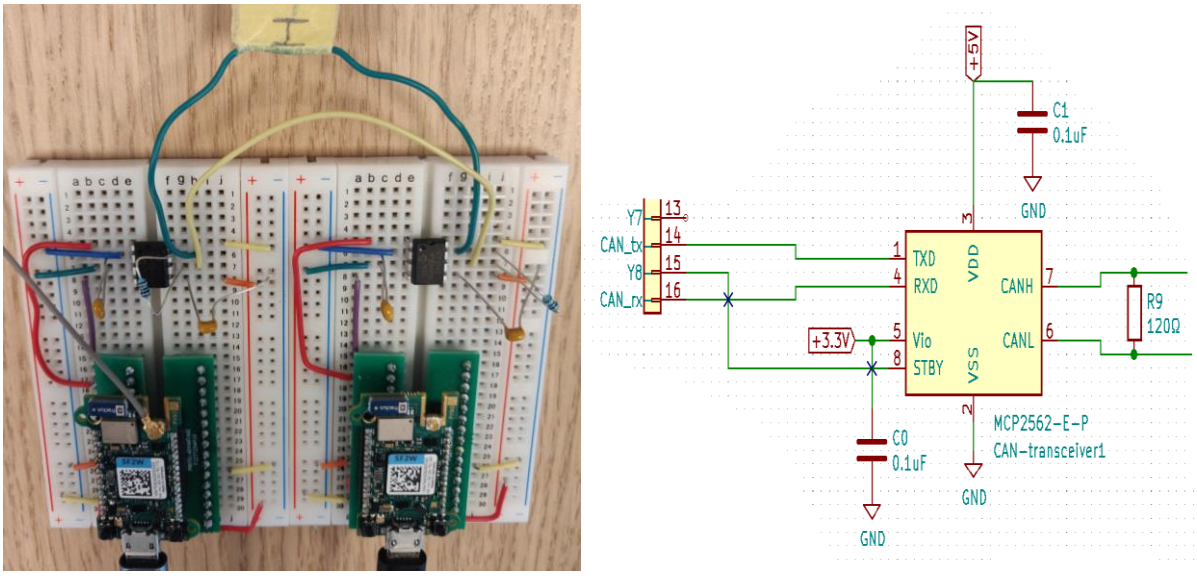
Då lösningen ska kunna användas i labbmiljö och det finns framtida planer att utveckla prototypen ger en kapsling ett mer färdigt intryck och enklare samt säkrare användning. Skalet har ritats upp i SolidWorks och 3D-printats med DLP teknik utav resin, detta ger en mycket fin finish mer likt en injektionsformad plastdetalj. Hela skalet består av fyra delar, en huvuddel där kretskortet skjuts in sedan två plintar som håller fast kortet. Till sist en lock som täcker hålet där kretskortet trycks in från.

3.5 CAN-KOMMUNIKATION

Den första funktionen som testades var att läsa och skriva CAN-meddelanden, Pyboard, CAN-transceiver, motstånd och kondensatorer kopplades upp enligt databladet för MCP2562 [10] på ett kopplingsdäck. Pyboarden drivs direkt med USB från dator och vid testet hämtades matningsspänningen till CAN-transceiver från USB genom pinnen i [Figur 2] märkt VUSB.

För att generera CAN-meddelanden används en USB till CAN adapter kallad PCAN tillsammans med mjukvaran PCAN-view [11]. Dessa produkter är tillverkade av PEAK-systems och används också för att läsa vad som händer på CAN-bussen.

CAN-kontrollen på Pyboarden initieras med en hastighet på 125 kbit/s, samma hastighet som Micropowers BMSer använder när de kommunicerar med deras konfigurationsprogram. Efter initieringen kan CAN-meddelanden läsas och skickas med hjälp av funktionerna can.read() och can.send(). Först testades CAN-kommunikation mellan PCAN och en pyboard och därefter kommunikation mellan två Pyboards kopplade på en gemensam CAN-buss så som visas i [Figur 3].



Figur 3. T.V. Två Pyboard kopplade på gemensam CAN-buss (gul och grön ledning). T.H. Elschema över CAN-transceiver mellan CAN-buss och Pyboard

3.6 WIFI KOMMUNIKATION OCH VAL AV TRANSPORTPROTOKOLL

Pyboardens WiFi modul kan konfigureras att fungera som en accesspunkt (ett eget nätverk skapas vilket andra enheter kan ansluta till) eller en station (Pyboarden ansluter till ett existerande lokalt nätverk). Detta ger två möjligheter till hur lokal WiFi-kommunikation mellan två pyboards kan fungera.

1. En pyboard fungerar som accesspunkt och en annan pyboard ansluter till denna. En fördel med detta är att kommunikation kan ske oberoende om ett lokalt nätverk existerar eller är tillgängligt för användning.
2. Två pyboards ansluter båda till samma existerande lokala nätverk. Fördelen är att räckvidden blir lika stor som nätverkets räckvidd, nackdelen är att båda pyboards måste konfigureras med lösenord och ssid för det aktuella nätverket.

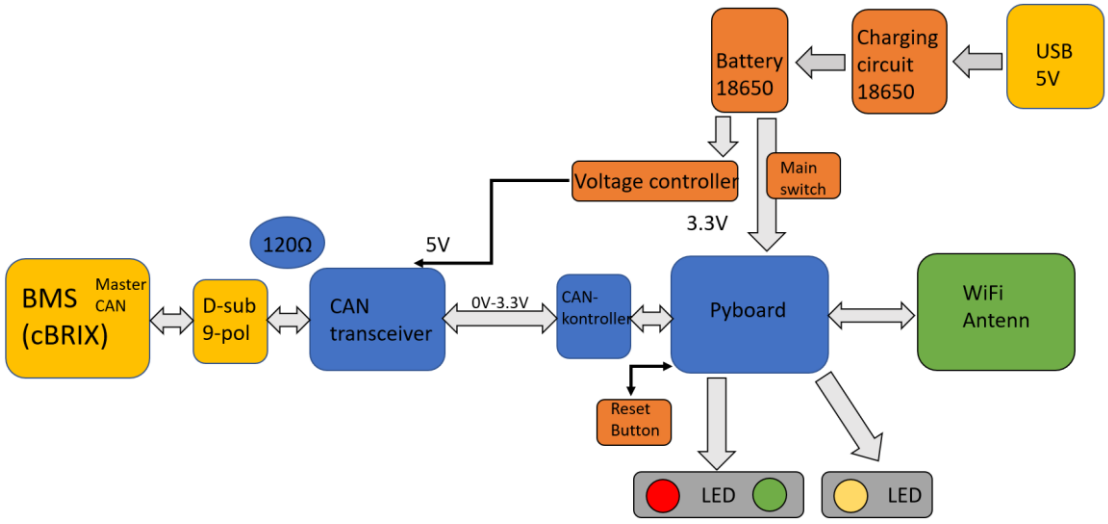
Alternativ 2 valdes för implementation i prototypen då räckvidd anses viktigare än driftmöjlighet utan tillgång till nätverk. Dessutom kan brist på tillgång till lokalt nätverk lösas med en bärbar router, en lösning som skulle ge minst lika bra räckvidd som alternativ 1.

När en nätverksanslutning väl är upprättad sker kommunikationen med hjälp av något transportprotokoll, vanligtvis UDP eller TCP. Tidigt i utvecklingsarbetet valdes TCP som protokoll då det säkerställer den överförda datans integritet. Något som är viktigt för den typ av data som prototypen skall förmedla eftersom en felaktigt konfigurerad BMS potentiellt kan skada batteriet.

När tester senare utfördes där CAN-meddelanden skickades mellan två CAN-bussar över TCP upptäcktes en fördröjning av datan på 500-1500ms vid start av kommunikation eller när CAN-meddelanden skickades med relativt låg frekvens (<10 Hz). Problemet visade sig vara Nagles algoritm, en funktion som förhindrar att TCP-paket skickas utan att först fyllas med en tillräckligt stor mängd data [12]. Då en lösning för att stänga av Nagles algoritm vid tillfället inte existerade i Micropython valdes istället UDP i kombination med CRC16 som kommunikationslösning.

3.7 Komponenterna på kretskortet

Efter att kommunikationen från CAN-buss till CAN-buss med WiFi fungerade, påbörjades konstruktionen av kretskortet. Ett flödesschema skapades som utgångspunkt till komponentval och ritning av elschema. Flödesschemat visas nedan i [Figur 4].



Figur 4. Flödesschema: hårdvara

Kretskortet kan delas upp i tre delar med motsvarande tre IC-chip, laddning, spänningsreglering och spänningsanpassning för CAN-buss. Pyboarden monteras med två 16-pin 1.27mm stiftlist, val av CAN-transceiver gjordes efter rekommendation av Jon Klasseson.

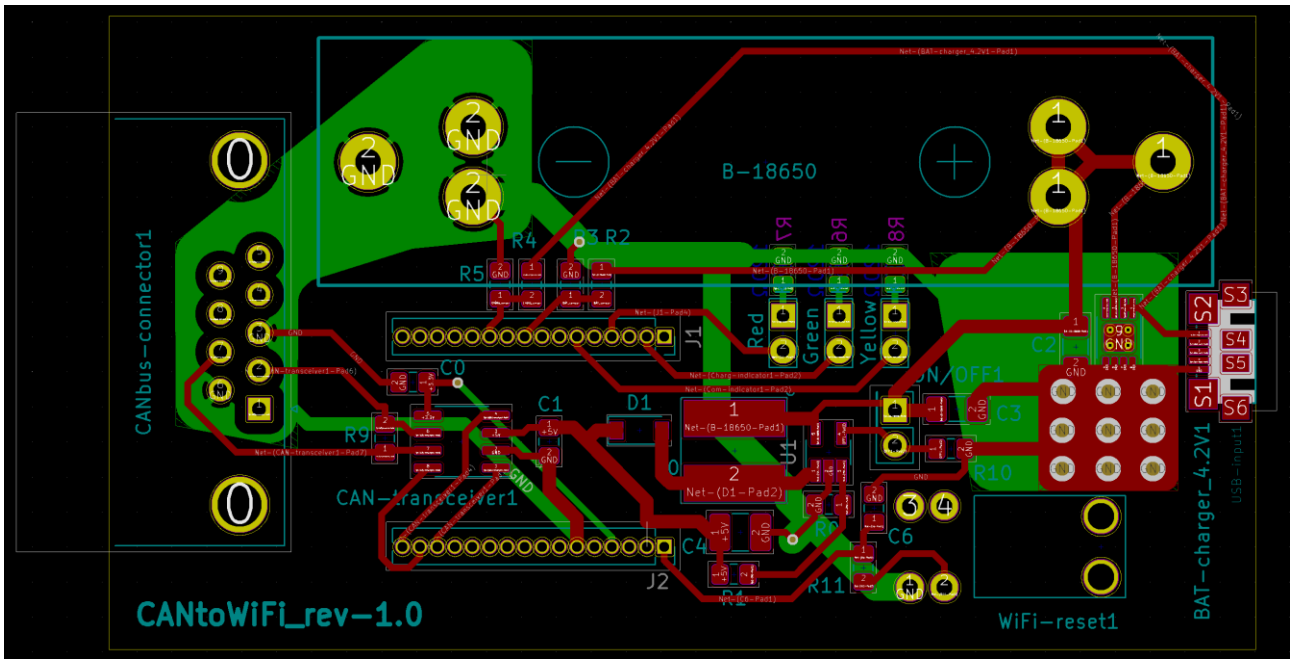
Kortet löddes för hand, så val av kapslingar skedde med det i åtanke. Spänningsregulatorn valdes efter att databladet innehöll ett applikationsexempel som stämde bra överens med vad som efterfrågades och kapsling TSOT-23-5 möjliggör också handlödning. Laddningskretsen valdes med samma metod.

Kretskortet ritades i KiCad vilket är ett opensource program som kan används genom hela processen. Ritning av elschema, listning av komponenter sedan konstruktion av layouten på PCBn och slutligen

generering av greberfiler som tillverkarna av PCBn använder. Ritningen av elschemat gjordes utifrån vad databladen för de tillhörande komponenterna krävde.

Då elschemat ritas behöver alla komponenter få ett fotavtryck vilket är kopplat till vilken kapsling komponenten har. Som tidigare nämnts, handlöds kortet vilket gjorde att större fotavtryck var att föredra. De flesta komponenter är av SMD typ vilket ger ett bättre pris och ett större utbud att välja från. Efter att alla komponenter har ett motsvarande fotavtryck kan designen av layouten på PCBn påbörjas.

Den färdiga PCBn som visas i [Figur 5] är ett två-lagers kort där framsidans kopparlager syns i rött och baksidan i grön. Det gula är genomgående hål som används för att montera komponenter men de kan också användas som vias där man vill ta ledningen mellan topp- och botten lagret. Vian till höger om kondensatorn C0 i figuren nedan är ett exempel på en sådan användning.



Figur 5. Ritning över PCB layout.

3.8 MJUKVARA

Mjukvarudelen av projektet beskrivs i detta avsnitt. Arbetet bestod till största delen av att utveckla pythonprogrammet som körs på pyboarden, men även en mindre modifikation av pyboardens firmware, vilken är skriven i C, behövde göras.

3.8.1 Firmware

Vid genomförandet av projektet uppdaterades mikrokontrollern med den senaste tillgängliga firmwaren, vilken hämtades från github [9]. Det upptäcktes att en funktion för att avläsa om inkommande CAN-meddelanden var av typ standard eller utökad ram saknades. CAN-meddelanden som prototypen förmedlar måste vara identiska på båda CAN-bussar, när mjukvaran skall återskapa ett meddelande mottaget på WiFi för att skicka på sin lokala CAN-buss måste därför information om vilken typ av ram det ursprungliga meddelandet använde finnas.

Modifikation av firmware

Följande modifikation av Micropythons källkod gjordes för att läsa inkommande CAN-meddelandes typ av ram:

I filen *"pyb_can.c",* med sökvägen *"micropython-master/ports/stm32/pyb_can.c",* ändrades funktionen *"STATIC(mp_obj_t pyb_can_recv(size_t n_args, const mp_obj_t *pos_args, mp_map_t *kw_args)"* enligt [Figur 6]. Efter kompilering och omflashning av mikrokontrollern resulterade detta I att funktionen för att läsa inkomna CAN-meddelanden returnerar ett värde som indikerar utökad eller standard ram.

```
549
550     // Populate the first 3 values of the tuple/list
551     #if MICROPY_HW_ENABLE_FDCAN
552     items[0] = MP_OBJ_NEW_SMALL_INT(rx_msg.Identifier);
553     items[1] = rx_msg.RxFrameType == FDCAN_REMOTE_FRAME ? mp_const_true : mp_const_false;
554     items[2] = MP_OBJ_NEW_SMALL_INT(rx_msg.FilterIndex);
555     #else
556     items[0] = MP_OBJ_NEW_SMALL_INT((rx_msg.IDE == CAN_ID_STD ? rx_msg.StdId : rx_msg.ExtId));
557     items[1] = rx_msg.RTR == CAN_RTR_REMOTE ? mp_const_true : mp_const_false;
558     items[2] = MP_OBJ_NEW_SMALL_INT(rx_msg.IDE);    // Modifierad kod
559     //items[2] = MP_OBJ_NEW_SMALL_INT(rx_msg.FMI);  // Original   kod
560     #endif
561
```

Figur 6. Modifikation av Micropythons källkod (Rad 559 ersattes med rad 558)

Kompilering och flashning av firmware

En kompilator för processorn Microptyhon ska byggas för måste vara installerad på datorn som bygger. I projektet användes arm-kompilatorn arm-none-eabi-gcc. Instruktioner för att bygga Micropython hittas i källkoden i mappen för processorn som firmware skall byggas till. I projektet användes följande kommandon på för att bygga firmware för Pyboard D-series på en dator med Linux som operativsystem [13].

```
1 $ cd micropython
2 $ make -C mpy-cross
3 $ cd ports/stm32
4 $ make submodules
5 $ make BOARD=PYBD_SF2
6 $ sudo make BOARD=PYBD_SF2 deploy
```

Rad 6 bränner den kompilerade firmwaren till en ansluten Pyboard i DFU-läge. För att sätta Pyboarden i DFU-läge hålls USR-knappen inne medan RST-knappen trycks ned, en LED på Pyboarden kommer då att cykla: röd → grön → blå → vit. När LED lampan är vit släpps USR-knappen. LED lampan kommer då blinka

röd för att indikera att Pyboarden är i DFU-läge. När flashningen är klar kommer Pyboarden att starta om i vanligt läge [14].

3.8.2 Program

När pyboarden startar sker en initiering av globala variabler och CAN-kontroller och sedan försöker programmet ansluta till ett lokalt nätverk. Nätverkets ssid, lösenord samt pyboardens nätverksinställningar hämtas från filen ”Settings.py”. Om programmet inte lyckas ansluta till nätverket kommer pyboarden starta om och försöka igen. Under uppstart och nätverksanslutning är de tre externa lysdioderna tända.

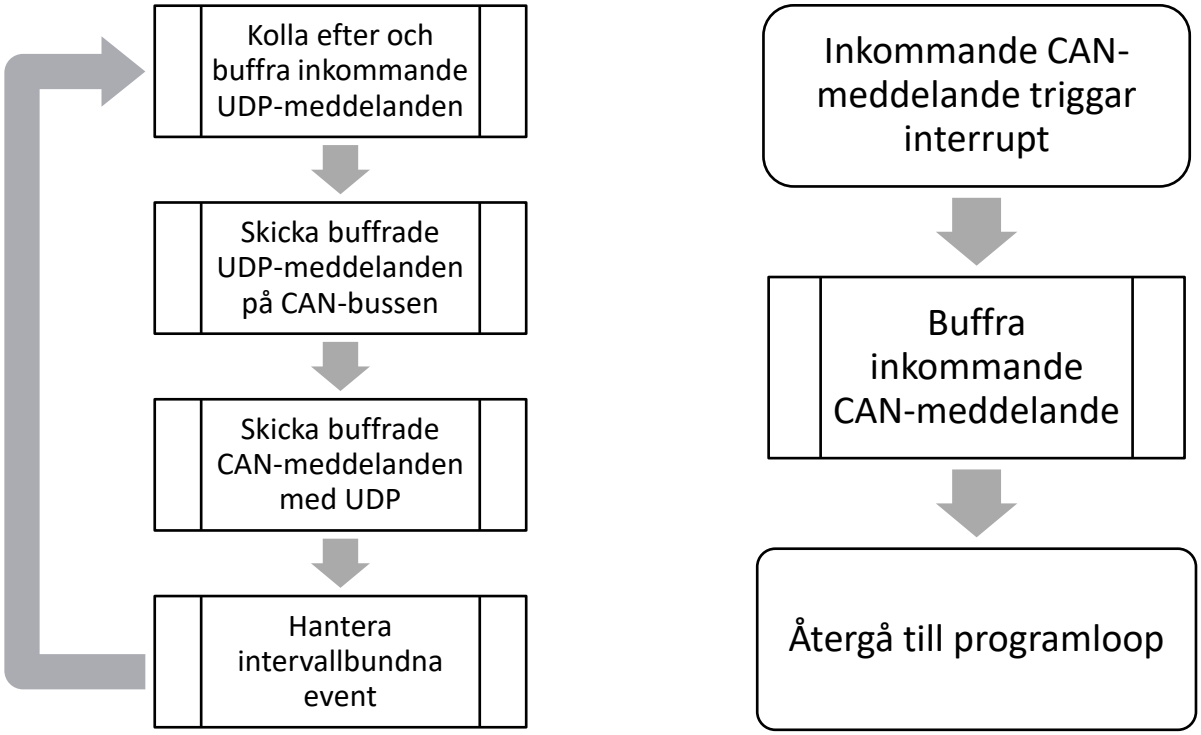
Efter lyckad anslutning initieras tre UDP-sockets:

- conn: Används för överföring av ramen som CAN-meddelanden skickas med.
- ackSocket: Används för att bekräfta en lyckad överföring av CAN-meddelanden på conn, eller för att begära en ny överföring av samma meddelande om datan blev korrupt.
- hbSocket: Används för att skicka och ta emot ”hjärtslag” till och från den andra pyboarden. Mottagna hjärtslag indikerar att anslutningen är obruten.

Därefter släcks röd och grön lysdiod för att indikera att noden är aktiv och programloopen startas.

Programloop

Programloopen som visas i [Figur 7] körs tills pyboard stängs av. Om programmet kraschar kommer pyboarden att starta om och återgå till programloopen.



Figur 7. Flödesschema: programloop.

Om ett CAN-meddelande kommer in lagras den av CAN-kontrollern i en hårdvaru-FIFO-buffert, denna buffert rymmer endast tre meddelanden, om ett meddelande anländer medan bufferten är full slängs det. Vid inkommande CAN-meddelanden triggas ett avbrott som kör en funktion vars syfte är att tömma FIFO-bufferten och lagra meddelandena på en större buffert i arbetsminnet.

Programloopen kan logiskt delas in i fyra sektioner:

Buffring av inkommande UDP

conn socketen kollas efter inkommande data, om data kommer antas det vara början på en inkommande ram och 18 bytes tas emot. Ramen packas upp och räknaren kontrolleras. Om räknaren är samma som föregående mottaget meddelandes räknare slängs ramen och programloopen fortsätter. Därefter packas bifogad kontrollsumma upp och jämförs med en kontrollsumma som beräknas på byte 0–14 i ramen. Om kontrollsummorna inte är likvärdiga skickas en begäran om att samma meddelande skall skickas igen på ackSocket och programloopen fortsätter. Om kontrollsummorna är likvärdiga packas resterande datafält i ramen upp och lagras i en buffert för utgående CAN-meddelanden. Efter lagring skickas en konfirmation om lyckad överföring på ackSocket och programloopen fortsätter.

Skicka utgående CAN-meddelanden

Om utgående CAN-meddelanden finns tas det äldsta meddelandet från bufferten och skickas på CAN-bussen. Om CAN-kontrollern är upptagen läggs meddelandet tillbaka på bufferten och programloopen fortsätter.

Skicka utgående UDP-meddelanden

Om bufferten för inkomna CAN-meddelanden inte är tom, tas det äldsta meddelandet från bufferten. Meddelandet packas i en ram och en kontrollsumma en räknare läggs till på slutet av ramen. Sedan skickas hela ramen på conn och programmet väntar på en bekräftelse från mottagande pyboard på ackSocket i högst 100 ms. Om ingen bekräftelse fås läggs meddelandet tillbaka på bufferten och programloopen fortsätter. Om bekräftelse fås om att mottaget meddelande är korrupt skickas samma meddelande igen. Om bekräftelse fås om lyckad överföring fortsätter programloopen.

Intervallbundna event

Varje programloop kontroleras ett antal timers och kod exekveras om dessa någon av dessa passerar ett fast intervall. Följande sker med ett intervall av...

500 millisekunder:

> Pyboarden skickar och tar emot ett hjärtslag på hbSocket. Om nodens batterinivå är låg togglas den röda LEDn, om batteriet laddas blinkar den gröna LEDn. Om batteriet laddas och är fullladdat lyser den gröna LEDn konstant.

1 sekund:

> Om ett hjärtslag inte tagits emot från den andra pyboarden den närmsta sekunden togglas den gula LEDn. Laddningskretsens spänning mäts för att upptäcka laddning.

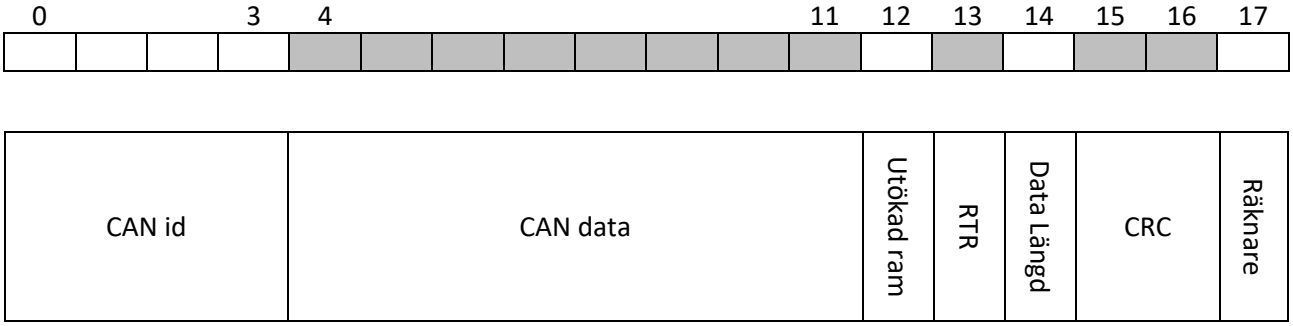
10 sekunder:

> Batterispänningen mäts och variabler sätts för att indikera om batteriet är fullt eller behöver laddas.

Ram

Ramen som CAN meddelanden skickas med över UDP beskrivs i detalj här. Byte 0–13 är en spegling av datastrukturen Micropythons inbyggda bibliotek returnerar när CAN-meddelanden tas emot. Booleaner packades i en byte vardera eftersom Micropythons standardbibliotek inte stödjer packning av bitar. Ramen visas nedan i [Figur 8].

- Byte 0–3 innehåller CAN-identifieringsnummer
- Byte 4–11 innehåller CAN meddelandet, om meddelandet är kortare än 8 bytes högerjusteras datan och resterande bytes fylls med nollor.
- Byte 12 innehåller en boolean som indikerar om meddelandet har utökat identifieringsnummer.
- Byte 13 innehåller en boolean som indikerar om meddelandet är en meddelandebegäran (remote frame)
- Byte 14 innehåller längden på CAN meddelandet i antal bytes
- Byte 15–16 innehåller en CRC som beräknats på byte 0–14
- Byte 17 innehåller en räknare som ökar för varje meddelande tills den överflödar och börjar om på 0. Räknarens syfte är att sortera bort meddelanden som kommer fram mer än en gång



Figur 8. Visualisering av byte-fält i ramen som används vid UDP kommunikation på socket conn.

4 RESULTAT

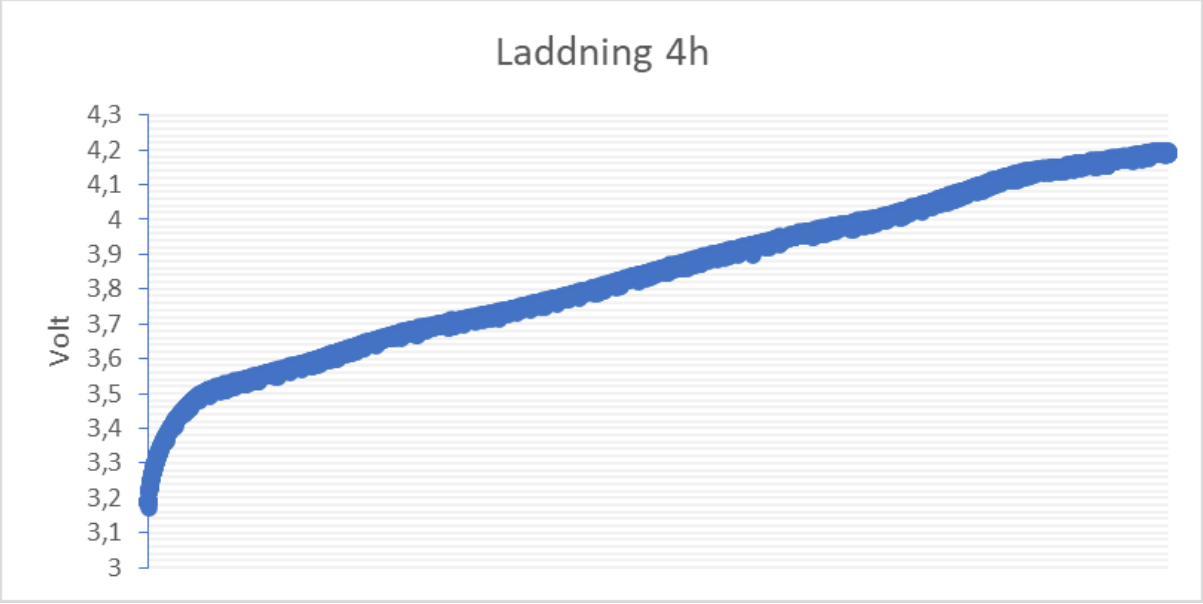
Här presenteras resultatet från ett laddningstest, flertalet olika kommunikationstester samt ett räckviddstest med och utan extern antenn.

4.1 LADDNINGSKRETS

När batteriet får en spänning på under 3.3V kommer den röda LED lampan börja blinka, laddningen startas när en 5V USB kopplas in och då släcks den röda LED lampan och den gröna börjar blinka. [Figur 9] visar en laddningscykel under 4 timmar.

Laddningskretsen med MIC79050 laddar under dess första fas med 580mA enligt mätningar med multimeter, då batterit nått en spänning på 4.2V påbörjas fas två där spänningen hållas konstant allt eftersom strömmen minskar. Detta heter konstant ström konstant spänning laddning, och är vanlig metod för att ladda litium-jonbatterier.

Laddningsstatus på batteriet utläses genom att mäta spänningen på batteriet och detektera om en 5V USB är inkopplad. När batteriet laddas, alltså 5V USB är inkopplad och batterispänningen är under 4.2V, kommer den gröna LED lampan blinka och när batteriet når 4.2V lyser den gröna LED lampan.



Figur 9. Spänningskurva på ett Samsung 18650 (3400mAh) under en laddning från 3,2V till 4,2V

4.2 MATERIALKOSTNAD

Låg kostnad på prototypen har inte varit av hög prioritet men onödigt höga kostnader har undvikits genom avvägningar vid komponentval. En uppdelning av materialkostnaden för en nod kan ses i [Tabell 1]. Den totala materialkostnaden för en färdig nod är 1391,92kr men detta inkluderar inte fraktkostnaderna för leverans till Micropower. Notera att en kommunikationslösning kräver två noder för att fungera. I [Tabell 1] kan ses att en stor del av kostnaden kommer från utvecklingskortet som användes. En möjlighet för att få ner kostnaden vore att placera mikrokontrollern direkt på det i projektet utvecklade kretskortet.

Kategori	Beskrivning	Antal	Pris
Utvecklingskort	Pyboard från MicroPython	1	530
Kapsling	4st delar, 3D-printad resin	1	329
Kretskort	1,6mm dubbelsidigt FR4 laminat	1	330
Komponenter	Laddningskrets	1	25,3
Komponenter	CAN-transceiver	1	9,5
Komponenter	Spänningsregulator	1	9,95
Komponenter	Kondensatorer	6	9,97
Komponenter	Resistorer	12	4,5
Komponenter	Spole	1	7,69
Komponenter	Diod	1	2,8
Komponenter	Knappar	2	30,1
Komponenter	LED lampor	3	10,08
Komponenter	Kontakter	5	49,35
Komponenter	Batterihållare	2	5,73
Komponenter	18650 batteri	1	37,95
Färdig prototyp	Materialkostnad för en nod	1	**1391,92**

Tabell 1. Materialkostnad för en färdig nod.

4.3 TESTER

För att mäta lösningens prestanda utfördes tre typer av test. Ett stresstest där CAN-meddelanden skickades med hjälp av PCAN. Ett kommunikationstest där en BMS avlästes och konfigurerades, samt ett räckviddstest där den på pyboarden inbyggda antennen jämfördes med ett externt alternativ.

4.3.1 CAN-buss till CAN-buss stresstest

Meddelande skickades mellan två PCAN CAN till USB adaptrar kopplade på två CAN-bussar som var anslutna till varandra genom projektets kommunikationslösning. Först testades envägskommunikation, där endast en PCAN skickade meddelanden och den andra PCAN tog emot meddelanden. Resultatet presenteras nedan i [Tabell 2].

Intervall mellan skickade meddelanden	100 ms		50 ms		10 ms		5 ms		1 ms	
	Mottagna	Skickade	Mottagna	Skickade	Mottagna	Skickade	Mottagna	Skickade	Mottagna	Skickade
Standard ram.	1006	1002	1003	1001	1047	1046	1074	1074	1517	1516
- andel tappade		0%		0%		0%		0%		0%
Utökad ram.	1008	1005	1001	1001	1056	1056	1074	1074	1439	1444
- andel tappade		0%		0%		0%		0%		0,4%

Tabell 2. Stresstestresultat av envägskommunikation.

Under testet upptäcktes att fler meddelande togs emot än vad som skickades, en möjlig förklaring till detta ges i kapitel [5.2]. Mottagande PCAN visade att meddelanden kom in med ett intervall på 1–10 ms, vilket innebär att bufferten för inkomna meddelanden kommer fyllas om meddelanden skickas oavbrutet med ett intervall lägre än 5 ms.

Tvåvägskommunikation testades sedan genom att låta båda PCAN verktyg skicka och ta emot meddelande samtidigt. Resultatet visas nedan i [Tabell 3].

Intervall mellan skickade meddelanden	100 ms		50 ms		10 ms		5 ms		1 ms	
	Mottagna	Skickade	Mottagna	Skickade	Mottagna	Skickade	Mottagna	Skickade	Mottagna	Skickade
Standard ram.	997	995	966	961	957	1207	588	1057	1598	3239
	997	997	515	974	900	1260	743	1025	1491	3048
- andel tappade		0,1%		23,5%		24,7%		36,1%		50,9%
Utökad ram.	1012	1007	484	920	1028	1026	899	892	775	4366
	1007	1001	893	954	1039	1629	988	986	791	4070
- andel tappade		0%		26,5%		22,1%		0%		81,4%

Tabell 3. Stresstestresultat av tvåvägskommunikation.

4.3.2 Kommunikationstest med BMS

Flera tester utfördes mot en BMS av samma typ som lösningen är tänkt att användas med. Vid testerna användes Micropowers egna mjukvara för att konfigurera och avläsa en BMS.

4.3.3 Räckvidd

Räckvidden testades genom att pinga BMSen med Micropowers egna verktyg. Testet utfördes med och utan extern antenn[2]. Testet upprepades med den pingande datorn i olika rum medan BMSen var stationär. Rummen som användes var:

- Samma rum som BMSen, access point fanns tillgänglig i rummet.
- Ett närliggande rum, access point fanns tillgänglig i rummet.
- Ett rum längre bort (Byggnadens entré), ca 15 m till närmsta access point och flera väggar mellan.

Resultatet av testet presenteras nedan i [Tabell 4].

Antenn typ	Rum	Tappade ping	Skickade ping	Andel tappade meddelanden	Meddelande per sekund
Intern	Samma	0	1000	0%	239
	Närliggande	0	1000	0%	238
	Längre bort	9	1000	0,9%	167
Extern	Samma	0	1000	0%	238
	Närliggande	0	1000	0%	239
	Längre bort	2	1000	0,2%	205

Tabell 4. Resultat av räckviddstest.

[2] Extern antenn som användes vid testet: MAF94149

5 DISKUSSION

I sin helhet ser vi att detta blev ett lyckat projekt: resultatet är tillfredsställande och vi kunde leverera 6st prototyper till Micropower. Men det finns många aspekter som skulle kunna förbättras och problem som kunnat lösas med andra metoder.

5.1 VAL AV MIKROKONTROLLER

Något som tidigt var uppe i diskussion när projektet startades var val av mikrokontroller, efter en undersökning stod det mellan Pyboard med sin STM32F722IEK chip och ESP32 chipet. Båda mikrokontrollerna har stöd för WiFi och CAN, STMen är dyrare men Micropower har använt Pyboarden innan till andra projekt vilket var anledningen till att det blev den som användes. I efterhand hade antagligen ESP32 varit ett bättre alternativ, den skulle ha gett möjligheten att skriva i C och dokumentationen på ESP32 är mycket mer täckande då det är ett äldre chip. Fördelen med att skriva i C är att vi var vana vid det språket från början, möjligen hade det krävts mer jobb i nätverkskonfigurationen och inställningar av sockets, men det är också här fördelen skulle kommit in. MicroPython ger en möjligheten att enkelt komma igång men begränsar möjligheten att gå in på detaljnivå och göra ändringar. Detta skapade problem mot slutet då prestandan skulle förbättras, vi tror att om vi använt ESP32 hade vi haft fler möjligheter i slutet att forma inställningarna noggrannare och förbättra prestandan ytterligare.

En annan fördel med ESP32 hade varit att det är ett chip som går att handlöda vilket hade öppnat upp möjligheten att göra en billigare prototyp. Alla komponenter skulle då köpas in lösa och monterats på ett enda egenutvecklat kretskort vilket hade tagit bort behovet av att köpa in ett separat utvecklingskort för varje prototyp, och därmed undvika den dyraste delen av prototypen.

5.2 RESULTAT

Testerna visar att kommunikationen fungerar mycket bra när trafiken går i en riktning. Men inte så bra när kommunikationen sker i båda riktningar. Att tänka på är dock att motsvarande intervallkategorier i stresstesterna för en- och tvåvägskommunikation inte innebär samma mängd meddelande per sekund på CAN-bussen. Intervallet avser hastigheten med vilken meddelande skickades från vardera nod, alltså är mängden trafik på bussen dubbelt så stor vid tvåvägskommunikation som vid envägskommunikation.

Stresstestet med tvåvägskommunikation visade stora variationer i prestandan, detta beror på en kaskadeffekt som uppstår när ett meddelande tappas. Eftersom kommunikationslösningen försöker sända om tappade meddelande gång på gång till den lyckas, byggs snabbt en kö upp när något går fel. Dessutom är lösningen byggd utan parallellism, och den avbrottsbaserade hanteringen av inkommande CAN-meddelanden leder till att en nod prioriterar inkommande CAN-meddelanden. Om meddelande då kommer för tätt hinner noden inte skicka meddelanden till den andra noden över WiFi och den interna bufferten blir till slut full.

Vid kommunikation med BMSen kommer meddelanden in med kort intervall men sällan i samma mängder som vid stresstesten. Dessutom sker kommunikationen mellan BMS och PC på ett sätt som inte innebär så mycket simultan tvåvägskommunikation utan mer i stil av en konversation där en part skickar en förfrågan och den andra svarar. Testet mot BMSen visade även att prestandan var god nog för den användning kommunikationslösningen är tänkt för.

Att fler meddelanden mottogs än skickades tror vi kommer sig av att meddelande läggs tillbaka på bufferten om väntan på en bekräftelse från mottagande nod tar längre än 100 ms. Om meddelandet faktiskt kom fram men bekräftelsen dröjde, till exempel på grund av att ett inkommande CAN-meddelande

triggar en interrupt. Skulle det innebära att samma meddelande skickas igen nästa programloop. Testerna visar att dubbletter är relativt ovanliga (mindre än 1 på 100). Om repeterade meddelanden är ett problem beror på hur mjukvaran som använder kommunikationslösningen är konstruerad.

5.3 Framtidsutsikter

Något som diskuterats från start av projektet och som projektet skulle kunna vara starten av är att implementera denna lösning direkt på kretskortet på framtida BMSer. Om varje BMS hade haft hård och mjukvaran att genom WiFi konstant kommunicera mot en webbserver hade stora mängder testdata kunnat loggas. Det hade också gjort det möjligt att nå alla batterisystem som är inom ett WiFi nätverk från var som helst där man har internetuppkoppling.

5.4 Miljö och etik

Att möjliggöra dataloggning av batterimoduler under drift kan ge insikter och data som bidrar till framtida utveckling, användning och innovation inom batteriteknik. Och bättre batteriteknik kan innebära mer energieffektiva och därmed miljövänliga energilagringslösningar. Med utvecklad batteriteknik kan förnyelsebar energi lättare utnyttjas. Längre livstid på batterier är även en möjlig påföljd som har en positiv påverkan på miljöaspekter.

Om denna lösning skulle vidareutvecklas som diskuterats i avsnitt [5.3] skulle webbservern användas till att programmera eller diagnostisera batterisystem från en och samma plats. Genom att distansera operatören av mjukvaran som konfigurerar batteriet, från batteriet och maskineriet batteriet driver förebygger man även potentiella risker operatören kan utsättas för i en industrimiljö. Operatören behöver inte heller göra resor till de olika plasterna där batterisystemen är i användning.

6 REFERENSER

[1] Micropower Group AB, ”lithium-ion-solution,” 2020. [Online]. Available: https://micropower-group.se/business-area/lithium-ion-solution/. [Använd 05 05 2020].

[2] gridconnect, ”gridconnect.com,” gridconnect INC, [Online]. Available: https://www.gridconnect.com/products/can-wifi-wireless-wi-fi-can-diagnostic-monitoring-development-tool. [Använd 03 06 2020].

[3] kvaser, ”kvaser.com,” kvaser, [Online]. Available: https://www.kvaser.com/product/kvaser-blackbird-v2/. [Använd 03 06 2020].

[4] Microchip Technology Inc., ”Microchip - MCP2562 product page,” 25 03 2014. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/20005167C.pdf. [Använd 22 05 2020].

[5] George Robotics Limited, ”PYBD-SF2-W4F2,” [Online]. Available: https://store.micropython.org/product/PYBD-SF2-W4F2. [Använd 08 05 2020].

[6] C.-c. Organisation, ”can-cia.org,” Okänt. [Online]. Available: https://www.can-cia.org/can-knowledge/can/can-history/. [Använd 23 04 2020].

[7] TeraTerm Project, ”Tera Term,” 07 01 2020. [Online]. Available: https://ttssh2.osdn.jp/index.html.en. [Använd 03 05 2020].

[8] Damien P. George et al, ”MicroPython documentation,” 2020. [Online]. Available: https://docs.micropython.org/en/latest/#. [Använd 2020].

[9] Damien P. George et al, ”MicroPython,” 2020. [Online]. Available: https://github.com/micropython/micropython. [Använd 2020].

[10] M. T. Inc, ”Microchip,” 2014. [Online]. Available: www.microchip.com/mymicrochip/filehandler.aspx?ddocname=en561044. [Använd 27 04 2020].

[11] PEAK-System, ”PEAK-System.com,” [Online]. Available: https://www.peak-system.com/PCAN-USB.199.0.html?&L=1. [Använd 08 05 2020].

[12] S. Cheshire, ”TCP Performance problems caused by interaction between Nagle’s Algorithm and Delayed ACK,” 20 05 2005. [Online]. Available: http://www.stuartcheshire.org/papers/NagleDelayedAck/. [Använd 14 05 2020].

[13] Damien P. Geroge et al, ”micropython/tree/master/ports/stm32,” [Online]. Available: https://github.com/micropython/micropython/tree/master/ports/stm32. [Använd 19 05 2020].

[14] Daimen. P. George et al, ”Pyboard-Firmware-Update,” [Online]. Available: https://github.com/micropython/micropython/wiki/Pyboard-Firmware-Update. [Använd 19 05 2020].

A APPENDIX 1

Detta appendix innehåller instruktioner för konfigurering och användning av kommunikationslösningen. Avsnittet är tänkt att kunna förstås utan att ha läst den fullständiga rapporten.

A.1 KONFIGURERING AV NODER

- Se till att switchen på kapslingen [Figure 1] är av.

Figure 1.Kapsling framsida

- För att konfigurera noderna för användning med det tillgängliga nätverket kopplas pyboarden in på en dator med micro USB kontakten som sitter på pyboarden (röd cirkel I [Figure 2]) OBS: Inte samma micro USB kontakten som finns tillgänglig på skalet.

Figure 2. Pyboard

- Pyboarden kommer nu finnas tillgänglig som ett skrivbart minne I filutforskaren.
  (Om inte pyboarden syns kan drivrutinerna behöva uppdateras till USB Mass Storage)

A.1.1 Nätverksinställningar

- Öppna filen ”Settings.py” i en textredigerare och uppdatera fälten med ssid, lösenord, ip-adress, för aktuell nod o.s.v.
  Hostname anger IP-adressen till noden som skall kommuniceras med.
- Spara filen, dra ut USB-kabeln och starta pyboarden med switchen på kapslingen.

A.2 Användning

För att använda noderna, slå på switchen på kapslingen, tre LED lampor på kapslingens ovansida skall börja lysa.

Noden försöker nu ansluta till nätverket med inställningarna givna i ”Settings.py”. Om noden inte kan ansluta till nätverket kommer noden starta om efter 10 sekunder. Detta kan ses genom att LED lamporna slocknar en kort stund.

När noden lyckats ansluta till det lokala nätverket slocknar röd och grön LED lampa. Den gula lampan kommer nu att blinka fram tills att kontakt med motsvarande nod är etablerad, därefter lyser den gula lampan konstant.

Om kontakten mellan noderna bryts under användning kommer den gula lampan att blinka tills kontakt kan återupprättas.

Noden försöker hela tiden återuppta anslutningen om den skulle tappas eller nod startas om.

A.2.1 Kommunikation

För att använda noderna kopplar man dom till respektive CAN-buss med D-sub kontakten som sitter på noden.

Noderna är med undantag av IP-adressen identiska och kan användas både med BMS och PC.

A.2.2 Laddning

För att ladda det inbyggda batteriet som driver noden kopplas 5V USB till micro USB kontakten på framsidan av kapslingen som syns i [Figure 1].

Batteriet kommer att laddas till 4.2 V.

Låg batterinivå indikeras genom att den röda LED lampan blinkar.

Laddning indikeras genom att den gröna LED lampan blinkar.

Fulladdat batteri indikeras genom att den gröna LED lampan lyser konstant.

A.2.3 LED lampor

Ovanpå noden sitter tre LED lampor som indikerar batteri- och kommunikationsstatus.

Vid uppstart, medan noden försöker ansluta till nätverket lyser alla tre lampor konstant.

Gul lampa

Den gula lampan indikerar kommunikationsstatus.

När den gula lampan blinkar betyder det att ingen kontakt kan fås med mottagande nod.

När den gula lampan lyser konstant finns kontakt med mottagande nod och kommunikation borde fungera som tänkt.

Grön lampa

Den gröna lampan används för att visa laddningsstatus.

När den gröna lampan blinkar laddas batteriet.

När den gröna lampan lyser konstant under laddning, betyder det att batteriet är fulladdat.

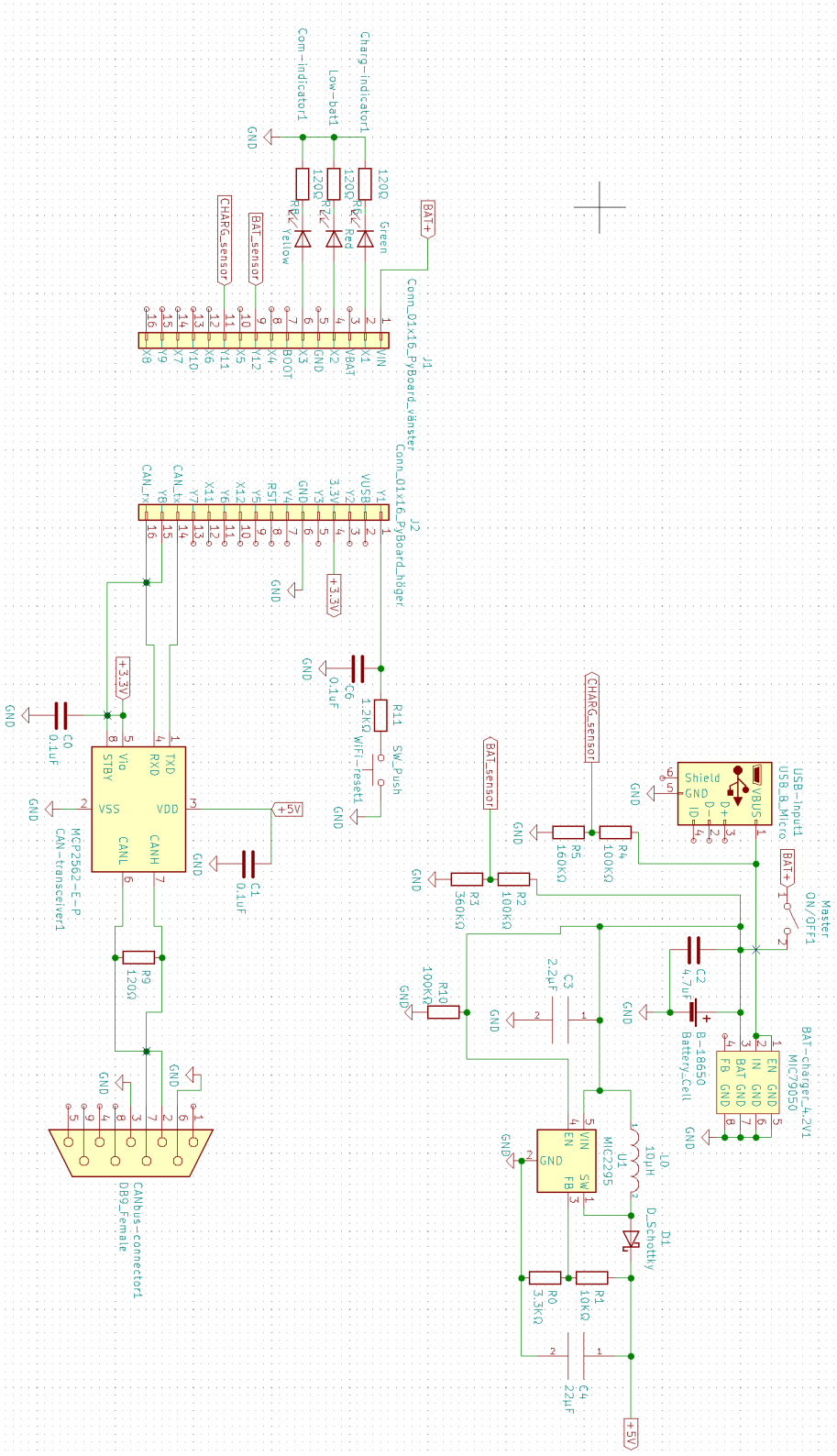
Röd lampa

Den röda lampan används för att visa batteristatus.

När den röda lampan blinkar är batterinivån låg.

B Appendix 2

B.1 Kretsschema



C APPENDIX 3

C.1 PROGRAMKOD

Samtliga pythonfiler nödvändiga för att återskapa en kommunikaitonsnod från projektet återfinns här.

C.1.1 Settings.py

```
1.  #Settings.py
2.
3.  #Set WiFi settings here
4.  ssid = 'ExampleSSID'
5.  password = 'Password123'
6.
7.  #set IP config for node here, must fit network
8.  IPconfig = ('192.168.128.35', '255.255.255.0', '192.168.128.1',
    '192.168.128.1')
9.
10.  #Set IP-adress of receiving node
11.  hostname = '192.168.128.34'
12.
13.  #Set ports to be used, must fit receiving node
14.  port = 24519
15.  hbPort = port+1 #Heartbeat port
16.  ackPort = port+2 #Acknoledgement port
17.
18.  #set to True if external antenna is used
19.  extAntenna = False
```

C.1.2 boot.py

```
1.  #boot.py -- run on boot-up
2.
3.  #imports
4.  import os
5.  import machine
6.  import pyb
7.  import utime
8.  import usocket
9.  import network
10.  import math
11.  import uselect
12.  from pyb import CAN
13.  import struct
14.  import micropython
15.  from machine import Pin
16.  from machine import ADC
17.
18.  micropython.alloc_emergency_exception_buf(100) #needed if error occurs
   during interrupt handling
19.
20.
21.  #Set clockrates, (if changed, can.init will need to be changed in
   main.py, see micropython documentation for settings)
22.  #pyb.freq(64000000, 64000000, 8000000, 32000000) #old settings
23.  pyb.freq(168000000, 168000000, 42000000, 84000000)#sys-clock,hcl-clock,
24.
25.  #set country
26.  pyb.country('SE') # ISO 3166-1 Alpha-2 code, eg US, GB, DE, AU
27.
28.  pyb.main('main.py') # main script to run after this one
```

C.1.3 inits.py

```
1.  #inits.py
2.  import pyb
3.  from machine import Pin
4.  from machine import ADC
5.
6.  #states
7.  IS_CHARGING = False
8.  IS_LOWBATTERY = False
9.  IS_FULLCHARGE = False
10.  RESET_IS_PRESSED = True
11.
12.  #timeEvents - timers that reset in even intervals
13.  time10s    = pyb.millis()
14.  time1000ms = pyb.millis()
15.  time500ms  = pyb.millis()
16.
17.  #timers
18.  timeCanAccess = pyb.millis() #stores last time can controller was
    accessed
19.  timeHeartbeat = pyb.millis() #last time a heartbeat signal was received
20.
21.  #variables
22.  batteryVoltage = 0
23.
24.  #user button
25.  resetBtn = Pin('Y1', Pin.IN, pull=Pin.PULL_UP) #Userknapp
26.  #resetBtn.value() #get value, is 0 when button is pressed
27.  timeResetBtn = pyb.millis() #Sets time when reset button is pushed
28.  resetBtnCounter = 0
29.
30.
31.  #onboard LEDs init
32.  redPyb = pyb.LED(1)
33.  greenPyb = pyb.LED(2)
34.  bluePyb = pyb.LED(3)
35.  #external LED pin init
36.  green = Pin('X1', Pin.OUT)
37.  red = Pin('X2', Pin.OUT)
38.  yellow = Pin('X3', Pin.OUT)
39.  #Analog to digital input pins
40.  BATsens = ADC('Y12') #Gives battery voltage
41.  CHARGEsens = ADC('Y11') #Gives 0 if not charging, >30 000 if charging
42.
43.  #Can stanby pin
44.  StandbyCAN = Pin('Y8', Pin.OUT)
45.  StandbyCAN.value(0)
46.
47.  #Can object
48.  can = pyb.CAN(1)
49.
50.  checkBatteryInterval = 5000; #interval of battery charge check
51.  timeCheckBattery = pyb.millis() #Time at last battery charge check
52.  timeBatteryBlink = pyb.millis() #Time at last battery warning blink
53.  LOWBATTERY = False #True if battery charge is low
```

C.1.4 func_defs.py

```
1.  #func_defs.py
2.  import os
3.  import machine
4.  import pyb
5.  import utime
6.  import usocket
7.  import network
8.  import math
9.  import uselect
10.  from pyb import CAN
11.  import struct
12.  import micropython
13.
14.  from inits import *
15.  from Settings import *
16.
17.  def wlan_init(APorST,ssid,password,IPconfig):
18.          wlan = network.WLAN()
19.          if(APorST == 'AP'):
20.                  st = network.WLAN(network.STA_IF)
21.                  st.disconnect()
22.                  st.active(False)
23.                  print('1')
24.                  while st.active() == True:
25.                          utime.sleep(0.1)
26.                  ap = network.WLAN(network.AP_IF)
27.                  ap.active(False)
28.                  ap.disconnect()
29.                  print('2')
30.                  while ap.active() == True:
31.                          utime.sleep(0.1)
32.                  #ap.config('mac')         # get the MAC address
33.                  ap.config(antenna=0)     # select antenna, 0=chip,
   1=external
34.                  ap.config(essid=ssid)
35.                  pyb.delay(500)
36.                  ap.active(True)
37.                  print('Setting up access point...')
38.                  while ap.active() == False:
39.                          utime.sleep(0.1)
40.                  print(ap.ifconf-ig())
41.                  wlan = ap
42.
43.          elif(APorST == 'ST'):
44.                  print('Setting up station')
45.  #               ap = network.WLAN(network.AP_IF)
46.  #               ap.disconnect()
47.  #               ap.active(False)
48.  #               print('1')
49.  #               while ap.active() == True:
50.  #                       utime.sleep(0.1)
51.                  st = network.WLAN(network.STA_IF)
52.  #               st.active(False)
53.  #               st.disconnect()
54.  #               print('2')
55.  #               while st.active() == True:
56.  #                       utime.sleep(0.1)
57.                  #st.config('mac')         # get the MAC address
58.                  if extAntenna == True:
```

```
59.                         st.config(antenna=1)    # select antenna, 0=chip,
   1=external
60.                 else:
61.                         st.config(antenna=0)
62.                 st.config(essid=ssid)
63.                 pyb.delay(500)
64.                 st.ifconfig(IPconfig)
65.                 if not st.isconnected():
66.                         print('connecting to network...')
67.                         st.active(True)
68.                         st.connect(ssid,password)
69.                         timeoutTimer = pyb.millis()
70.                         while not st.isconnected():
71.                                 utime.sleep(0.2)
72.                                 if pyb.millis() - timeoutTimer > 10000:
73.                                         print('Cannot connect to network,
   running hardware reset!')
74.                                         utime.sleep(0.5)
75.                                         machine.reset()
76.                 print('network config:', st.ifconfig())

77.                 wlan = st
78.         return wlan
79.
80.
81.  def toggle(pin):
82.         if pin.value() == 1 :
83.                 pin.value(0)
84.         else :
85.                 pin.value(1)
86.
87.
88.  def crc16(data: bytes, poly=0x8408):
89.      data = bytearray(data)
90.      crc = 0xFFFF
91.      for b in data:
92.          currentByte = 0xFF & b
93.          for _ in range(0, 8):
94.              if (crc & 0x0001) ^ (currentByte & 0x0001):
95.                  crc = (crc >> 1) ^ poly
96.              else:
97.                  crc >>= 1
98.              currentByte >>= 1
99.      crc = (~crc & 0xFFFF)
100.     crc = (crc << 8) | ((crc >> 8) & 0xFF)
101.
102.     return crc & 0xFFFF
```

C.1.5 main.py

```
1.  #main.py
2.
3.  #imports
4.  from func_defs import *
5.  from inits import *
6.  from Settings import *
7.
8.  #Turn on all LEDs during startup
9.  green.value(1)
10.  greenPyb.off()
11.  red.value(1)
12.  redPyb.on()
13.  yellow.value(1)
14.  bluePyb.off()
15.
16.
17.  StandbyCAN.value(0) #Set can tranceiver to active, if set to 1 the can
   tranceiver will enter low power standby mode
18.  pyb.Pin('EN_3V3').on() #Set voltage output for can tranciver
19.
20.  #init can controller, If clock frequency is changed or CANbus bitrate is
   changed,
21.  #can.init will need to be changed on multiple locations in this document
22.  #can.init(mode=pyb.CAN.NORMAL, extframe=False, prescaler=4, sjw=1,
   bs1=13, bs2=2) #old settings
23.  can.init(mode=pyb.CAN.NORMAL, extframe=False, prescaler=16, sjw=1,
   bs1=14, bs2=6, auto_restart=True)
24.  can.setfilter(0, CAN.MASK16, 0, (0, 0, 0, 0)) #set can filter to allow
   all IDs
25.
26.  #Buffer where incoming and outgoing messages are stored
27.  canInBuffer = []
28.  canOutBuffer = []
29.
30.
31.
32.  #-----------Buffer incoming CAN messages-------------#
33.  #Incoming messages trigger an interrupt that schedules a function
34.  #that empties the CAN FIFO into the canInBuffer
35.
36.
37.  def incomingCan(reason):
38.          while can.any(0): #empty FIFO into buffer
39.                  canInBuffer.insert(0,can.recv(0))
40.
41.
42.  def my_canbus_interrupt_test(bus, reason):
43.          # Schedule a task to be handled soon
44.          if reason == 0:
45.                  #print('pending')
46.                  micropython.schedule(incomingCan, reason)
47.                  return
48.          if reason == 1:
49.                  #print('full')
50.                  micropython.schedule(incomingCan, reason)
51.                  return
52.          if reason == 2:
53.                  #print('overflow')
54.                  bluePyb.on()
```

```
55.                 micropython.schedule(incomingCan, reason)
56.                 return
57.
58.  #link interrupt function to CAN FIFO
59.  can.rxcallback(0, my_canbus_interrupt_test) #function to be called when
   a CAN message enters FIFO 0
60.
61.
62.  #Connect to WiFi network
63.  wlan = wlan_init('ST',ssid,password,IPconfig)
64.  print('setting local ip to', IPconfig[0])
65.  wlan.ifconfig(IPconfig) #Configure IP
66.
67.  #catch crashes
68.  try:
69.
70.          resetBtnCounter = 0 #Used for triggering hardware reset
71.
72.          #Setup all sockets as UDP and bind to pollers
73.          #setup main communication socket
74.          conn = usocket.socket(usocket.AF_INET, usocket.SOCK_DGRAM) #UDP
75.          conn.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR, 1)
   #Make socket reusable
76.          #bind socket to port
77.          addr = usocket.getaddrinfo("0.0.0.0",port)[0][-1]
78.          conn.bind(addr)
79.          #create poller
80.          poller = uselect.poll()
81.          poller.register(conn, uselect.POLLIN)
82.
83.
84.          #setup heartbeat socket
85.          #used to check connection to receiving node
86.          hbSocket = usocket.socket(usocket.AF_INET, usocket.SOCK_DGRAM)
   #UDP
87.          hbSocket.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR, 1)
   #Make socket reusable
88.          hbAddr = usocket.getaddrinfo("0.0.0.0",hbPort)[0][-1]
89.          hbSocket.bind(hbAddr)
90.          hbPoller = uselect.poll()
91.          hbPoller.register(hbSocket, uselect.POLLIN)
92.
93.          #setup acknowledgement socket
94.          #Receives UDP transfer acknowledgement from receiving node
95.          ackSocket = usocket.socket(usocket.AF_INET, usocket.SOCK_DGRAM)
   #UDP
96.          ackSocket.setsockopt(usocket.SOL_SOCKET, usocket.SO_REUSEADDR, 1)
   #Make socket reusable
97.          ackAddr = usocket.getaddrinfo("0.0.0.0",ackPort)[0][-1]
98.          ackSocket.bind(ackAddr)
99.          ackPoller = uselect.poll()
100.         ackPoller.register(ackSocket, uselect.POLLIN)
101.
102.
103.         if conn:
104.
105.                 #indicate program running with exclusive yellow light
106.                 redPyb.off()
107.                 greenPyb.on()
108.                 yellow.value(1)
109.                 green.value(0)
```

```
110.                red.value(0)
111.
112.                lastMessageCounter = 0 #Saves counter of last recv
    message on wifi
113.                iteration = 0 #program loop counter
114.
115.                #PROGRAM LOOP START
116.                while True:
117.                        iteration += 1 #program loop counter
118.
119.
120. #----------Buffer incoming UDP messages-------------#
121.
122.                        #poll for incoming message
123.                        fdVsEvent = poller.poll(0)
124.                        for descriptor, Event in fdVsEvent:
125.                                if Event & uselect.POLLIN:
126.                                        msg_bs = b'' #Empty bytestring
    that will store message
127.                                        Recv_lenght=18 #Lenght of message
    to receive
128.                                        while Recv_lenght > 0: #Receive
    Recv_lenght bytes
129.                                                msg_bs +=
    conn.recv(Recv_lenght) #Add bytes to msg_bs
130.                                                Recv_lenght = Recv_lenght
    - len(msg_bs) #Update amount of bytes left to receive
131.
132.                                        #frame: IIII MMMMMMMM E R L CC c
133.                                        #where: I=id, M=message, E=ext,
    R=RTR, L=message lenght in bytes, C=CRC, c=counter 0-255
134.                                        #Unpack message according to
    frame^
135.
136.                                        #Check if counter is different
    from last message recived, if not then something is wrong :)
137.                                        counter = struct.unpack('B',
    msg_bs[17:18])[0]
138.                                        if lastMessageCounter != counter:
139.                                                lastMessageCounter =
    counter
140.                                                #check CRC
141.                                                CRC = crc16(msg_bs[0:15])
142.                                                if CRC ==
    struct.unpack('H',msg_bs[15:17])[0]:
143.
144.                                                        #unpack ID, lenght,
    message, extID and RTR
145.                                                        msgID   =
    struct.unpack('I', msg_bs[0:4])[0]
146.                                                        msgLen  =
    struct.unpack('B', msg_bs[14:15])[0]
147.                                                        msg = msg_bs[(12-
    msgLen):12] #unpack only msgLen bytes
148.                                                        isExtFrame =
    msg_bs[12:13] #unpack External frame identifier
149.                                                        RTR = msg_bs[13:14]
    #Unpack RTR
150.
151.                                                        #insert unpacked
    message into buffer
```

```
152.
    canOutBuffer.insert(0,(msgID,msg,isExtFrame,RTR))
153.
154.                                                    #send
   acknoledgement of successfull transmission
155.
    ackSocket.sendto(b'A',(hostname,ackPort))
156.
157.                                                else:
158.                                                    #send
   acknoledgement of failed CRC
159.
    ackSocket.sendto(b'C',(hostname,ackPort))
160.
161.
162. #------------Send Buffered CAN messages--------------#
163.
164.                         #check if canOutBuffer contains any messages
165.                         if len(canOutBuffer) > 0:
166.                                 #pop canOutBuffer
167.                                 (msgID,msg,isExtFrame,RTR) =
   canOutBuffer.pop()
168.
169.                                 #check if message has extended ID
170.                                 #if ID is extended the can module need to
   be reinitated to send with extended ID
171.                                 if isExtFrame == b'1':
172.                                         can.init(mode=pyb.CAN.NORMAL,
   extframe=True, prescaler=16, sjw=1, bs1=14, bs2=6, auto_restart=True)
173.                                         #can.init(mode=pyb.CAN.NORMAL,
   extframe=True, prescaler=4, sjw=1, bs1=13, bs2=2) #gamla inställnigar som
   funkar
174.
175.                                 #check if message is RTR and send
   accordingly
176.                                 if RTR == b'1':
177.                                         try:
178.
    can.send(msg,msgID,rtr=True)
179.                                         except Exception:
180.                                                 #if the CAN controller is
   busy, rebuffer the message and try later
181.
    canOutBuffer.append((msgID,msg,isExtFrame,RTR))

182.                                 else:
183.                                         try:
184.                                                 can.send(msg,msgID)
185.                                         except Exception:
186.                                                 #if the CAN controller is
   busy, rebuffer the message and try later
187.
    canOutBuffer.append((msgID,msg,isExtFrame,RTR))
188.
189.                                 #reinit CAN controller to a setting that
   can receive both extended and normal ID frames
190.                                 can.init(mode=pyb.CAN.NORMAL,
   extframe=False, prescaler=16, sjw=1, bs1=14, bs2=6, auto_restart=True)
191.                                 #can.init(mode=pyb.CAN.NORMAL,
   extframe=False, prescaler=4, sjw=1, bs1=13, bs2=2) #gamla inställnigar som
   funkar
```

```
192.
193.
194. #------------Send Buffered UDP messages--------------#
195.
196.
197.         #frame: IIII MMMMMMMM E R L CC c
198.         #där: I=id, M=message, E=ext, R=RTR, L=message lenght in bytes,
   C=CRC, c=counter 0-255
199.
200.                         #check if canInBuffer contains any messages
201.                         if len(canInBuffer) > 0:
202.                                 canRecvTuple = canInBuffer.pop()
203.
204.                                 #construct bytestring that will be sent
205.
206.                                 #check if message is RTR
207.                                 RTR = b'0'
208.                                 if canRecvTuple[1]:
209.                                         RTR = b'1'
210.
211.                                 #check if message is extended ID
212.                                 isExtFrame = b'0'
213.                                 if canRecvTuple[2] == 4:
214.                                         isExtFrame = b'1'
215.
216.                                 #pack message inte bytestring
217.                                 msgID_bs =
   struct.pack('I',canRecvTuple[0])    #message ID, packed in 4 unsigned bytes
218.                                 msg = canRecvTuple[3] #message is already
   bytestring
219.                                 msgLen = struct.pack('B',len(msg)) #Save
   message length in bytes, packed in one Unsigned byte
220.                                 #add padding to msg if needed so it's
   always 8 bytes
221.                                 while len(msg) < 8:
222.                                         msg = b"\0" + msg
223.
224.                                 #combine bytestrings
225.                                 msg_bs = msgID_bs + msg + isExtFrame + RTR
   + msgLen #message bytestring
226.
227.                                 CRC = crc16(msg_bs) #Calculate Checksum
228.                                 msg_bs += struct.pack('H',CRC) #Add
   checksum to message, packed as 2 unsigned bytes
229.
230.                                 msg_bs += struct.pack('B', iteration %
   255) #Add counter to message, packed as one unsigned byte (0-255)
231.
232.
233.                                 #Try to send can message over UDP and
   await acknoledgement from
234.                                 #receiving socket. If transfer is
   corrupted (crc failed)
235.                                 #Try sending the same message again.
236.                                 #If waiting for acknoledgent times out or
   there is a counter error,
237.                                 #Push the message back on the canInBuffer
   and continue the program
238.                                 while True:
239.
    conn.sendto(msg_bs,(hostname,port)) #send message as UDP
```

```
240.                                        exitLoop = False
241.                                        ackRecv = False
242.                                        fdVsEvent = ackPoller.poll(100)
   #poll for acknoledgement, will wait for 100ms for incoming ack, lowering
   this might cause messages to arrive in wrong order but speed up overall
   transmission
243.                                        for descriptor, Event in
   fdVsEvent:
244.                                                if Event & uselect.POLLIN:
245.                                                        #receive ack on
   socket
246.                                                        ack =
   ackSocket.recv(1)
247.                                                        ackRecv = True

248.                                                        if(ack == b'A'):
   #message transfer successfull, exit loop
249.                                                                exitLoop =
   True
250.                                                        if (ack == b'C'):
   #CRC error, try sending again
251.                                                                pass
252.                                        if ackRecv == False: #ack timeout,
   rebuffer message
253.
     canInBuffer.append(canRecvTuple)
254.                                                break
255.                                        if exitLoop == True:
256.                                                break

257.
258.
259. #==================TIME EVENTS=====================#
260. #TIME EVENTS: Stuff that happens every...
261. #500 ms
262. if (pyb.millis() - time500ms > 500):
263. time500ms = pyb.millis() #reset timer
264.
265. #if battery is charging and not full charge
266. #blink green light
267. if IS_CHARGING and not IS_FULLCHARGE:
268. toggle(green)
269. red.value(0)
270. elif IS_LOWBATTERY: #if battery is not charging and low charge, blink
   red light
271. toggle(red)
272.
273. #if battery is charging and full charge, show constant green light
274. if IS_FULLCHARGE and IS_CHARGING:
275. green.value(1)
276. #if battery is not charging, turn of green light
277. if not IS_CHARGING:
278. green.value(0)
279.
280. #Send and receive heartbeat to see if connection is working
281. hbSocket.sendto("H",(hostname,hbPort))
282. hbEvent = hbPoller.poll(0)
283. for descriptor, Event in hbEvent:
284. if Event & uselect.POLLIN:
285. heartbeat = hbSocket.recv(1)
286. timeHeartbeat = pyb.millis()
```

```
287.
288.
289. #1 s
290.                         if (pyb.millis() - time1000ms > 1000):
291. time1000ms = pyb.millis() #reset timer
292.
293. #check if heartbeat has been recived this second
294. if (pyb.millis() - timeHeartbeat > 1000):
295. #if connection is lost, blink yellow light
296. print("Connection lost to receiving node.")
297. toggle(yellow)
298. else:
299. yellow.value(1)
300.
301. #check if battery charger is connected
302. if (CHARGEsens.read_u16() > 50000):
303. IS_CHARGING = True
304. else:
305. IS_CHARGING = False
306.
307.
308. #10 s
309. if (pyb.millis() - time10s > 10000):
310. time10s = pyb.millis()
311.
312. #read battery voltage and set state accordingly
313. batteryVoltage = BATsens.read_u16()
314. if batteryVoltage < 48000:
315. IS_LOWBATTERY = True
316. else:
317. IS_LOWBATTERY = False
318. if batteryVoltage > 61000:
319. IS_FULLCHARGE = True
320. else:
321. IS_FULLCHARGE = False
322.
323. #reset reset button counter
324. resetBtnCounter = 0
325.
326.
327.
328. #=================RUN EVERY LOOP====================#
329.
330. #check if reset Button is pressed and add to resetBtnCounter,
331. #if counter goes over a certain value a hardware reset is triggered.
332. #It's done this way because the value on the resetBtn pin flutters
   sometimes
333. #The reason for this is unknown
334. if resetBtn.value() == 0:
335. resetBtnCounter += 1
336.
337. if resetBtnCounter > 1000:
338. print("reset button pressed, running hardware reset!!!")
339. utime.sleep(0.5)
340. machine.reset()
341.
342. except Exception:
343.         print('Crash caused by exception:', Exception)
344.         utime.sleep(0.5)
345.         machine.reset()
```

```
346.
```