IOT IN GEBOUWEN AUTOMATISATIE

PBA in de Elektronica-ICT

Eindwerk voorgedragen tot het behalen van
de graad en het diploma van bachelor in de Elektronica-ICT

Door: Olivier Van den Eede
Promotor hogeschool: Wim Dams
Promotor bedrijf: Jan Derua

Academiejaar 2016-2017
Campus De Nayer, Jan De Nayerlaan 5, BE-2860 Sint-Katelijne-Waver

LID VAN ASSOCIATIE KU LEUVEN

THOMAS MORE

Voorwoord

Ik wil graag de heer Jan Derua, mijn stagebegeleider en de heer Wim Dams, de schoolbegeleider bedanken voor het mogelijk maken van deze bachelorproef. Ze hebben mij zeer goed begeleid tijdens deze periode waardoor alles tot een goed einde gekomen is.

Inhoudstafel

Lijst van figuren

Inleiding

In dit werkstuk gaan we onderzoeken wat de mogelijkheden zijn van “Internet Of things” in de gebouwenautomatisatie.

De integratie van IoT in combinatie met domotica kan heel wat voordelen bieden, denk maar aan het sturen van elektrische apparaten afhankelijk van over/onder productie van elektriciteit. Of het automatisch bedienen van uw huis via een gsm of zelfs via voice control of Siri. Het zou zelfs mogelijk moeten zijn om te detecteren of mensen thuis zijn, en op basis van deze informatie het huis anders te bedienen.

Voor dit onderzoek zal er ook een praktische realisatie gemaakt worden die meerdere van dit soort toepassingen mogelijk moet maken op hetzelfde platform. Het hele platform zal gebaseerd zijn op een embedded Linux systeem dat communiceert met de KNX domotica bus.

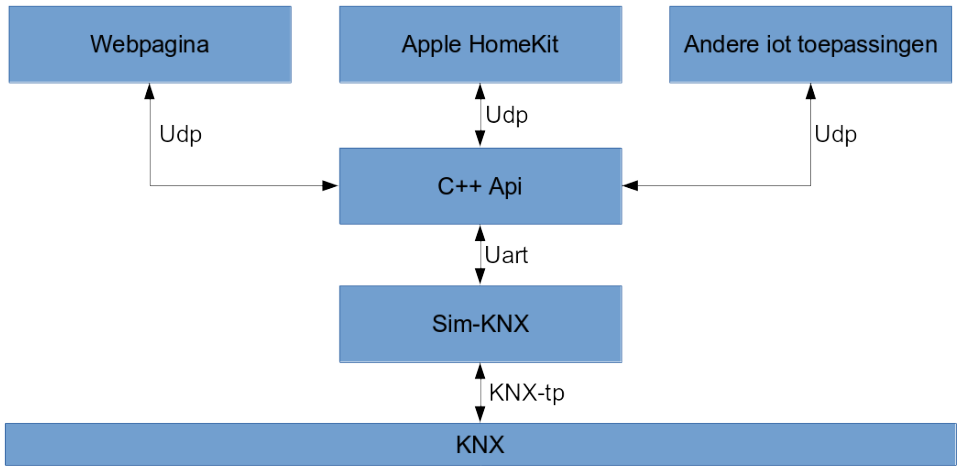
1 Doelstellingen van de bachelorproef

Het doel van dit eindwerk is het onderzoeken van de mogelijkheden van "Internet of things" of IoT binnen de wereld van de gebouwenautomatisatie en domotica.

De term "Internet of things" staat voor de wereld waarnaar we momenteel op weg zijn, waarbij alle mogelijke voorwerpen gekoppeld zullen zijn aan het internet. Dit kan gaan van kopiemachines en fototoestellen tot wasmachines. Zo kunnen deze toestellen communiceren met hun gebruikers maar vooral ook met elkaar.

Dit kan uiteraard zeer interessant zijn in de wereld van domotica omdat deze informatie van al de gekoppelde apparaten gebruikt kan worden om een gebouw veel slimmer te maken, en zo bijvoorbeeld ook veel zuiniger en efficiënter.

In deze bachelorproef zal dit principe geïmplementeerd worden voor het KNX communicatieprotocol. Er zal een embedded Linux systeem opgezet worden dat via een low level C++ applicatie zal kunnen communiceren op de KNX bus. Bovenop deze laag is het de bedoeling om "IoT" toepassingen te laten werken, deze communiceren dan langst 1 kant met het internet, en aan de andere kant met de C++ API die de berichten doorverbind met de bus. Dit principe is te zien in figuur 1.



Figuur 1: Blockschema verschillende lagen

2 KNX

2.1 KNX Association

KNX is de naam van gestandaardiseerd protocol voor een bussysteem die beschrijft hoe sensoren en actoren met elkaar communiceren. Dit protocol is de standaard in de gebouwenautomatisering. De oprichter en eigenaar van de KNX technologie is de KNX Association.

KNX is ontstaan uit convergentie van 3 vorige standaards, namelijk EHS, BatiBUS en EIB.

2.2 Bus communicatie

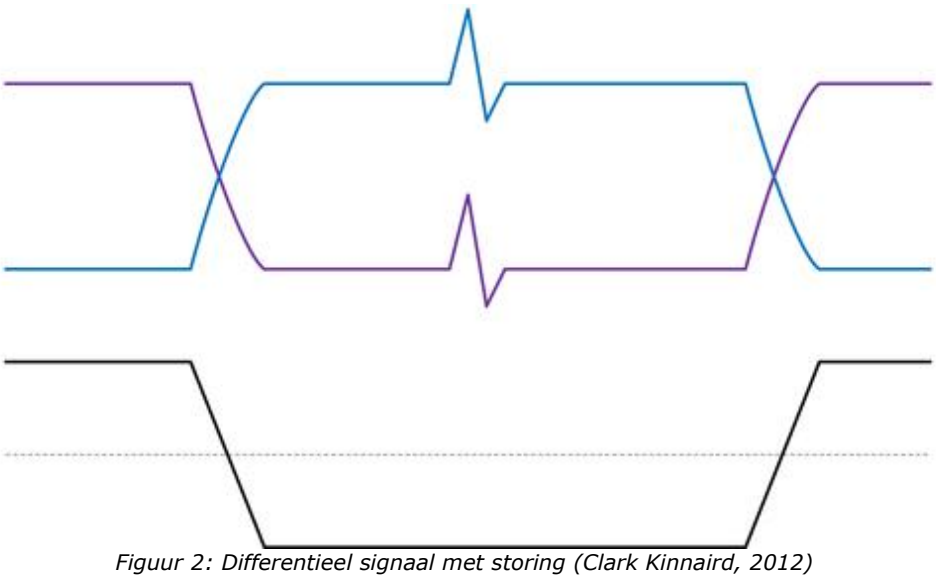
2.2.1 Communicatie media

Het KNX protocol is grotendeels gebaseerd op dat van EIB, maar is uitgebreid met een physical layer waardoor het mogelijk is om op meerdere communicatie media te werken.

De mogelijke media zijn:

- Twisted pair
- Powerline
- KNX-RF
- Infrarood
- Ethernet (KNXnet/IP).

Het belangrijkste en meest gebruikte medium is twisted pair, en dit zal dan ook gebruikt worden voor onze communicatie, en zal hieronder in detail uitgelegd worden.



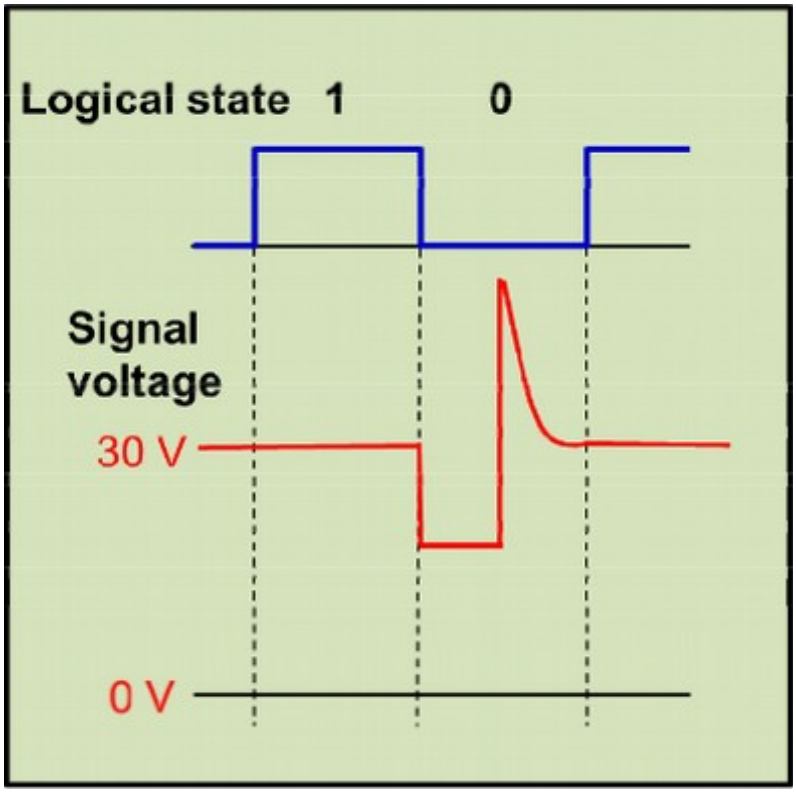
Figuur 2: Differentieel signaal met storing (Clark Kinnaird, 2012)

2.2.2 KNX-TP

KNX-TP maakt gebruik van een half-duplex twisted pair differentiële bus. Dat wil zeggen dat de data op de bus altijd bekeken wordt als het verschil tussen de 2 draden, en dat de 2 draden rond elkaar gewikkeld zijn. De toepassing van deze 2 principes voorkomt een grote hoeveelheid externe storing doordat de storing op de 2 draden evenveel zou inwerken. Doordat de storing op de 2 draden inwerkt en het verschil tussen de 2 draden genomen wordt, zal de storing niet gedetecteerd worden. Dit is te zien in figuur 2.

De bus wordt gevoed door een speciale KNX voeding die een 29V-DC spanning genereert. De reden voor de speciale voeding is dat de data bovenop deze voedingsspanning doorgestuurd wordt. Deze spanning die altijd aanwezig is op de bus, wordt gebruikt door de componenten aangesloten als voeding.

De binaire data wordt doorgestuurd bit per bit, dit gebeurd in 2 states. De bus in rust is de transmissie van een logische "1" en een "0" wordt gestuurd door een drop met daarna een opslingering van de voeding. Dit wordt weergegeven in figuur 3.

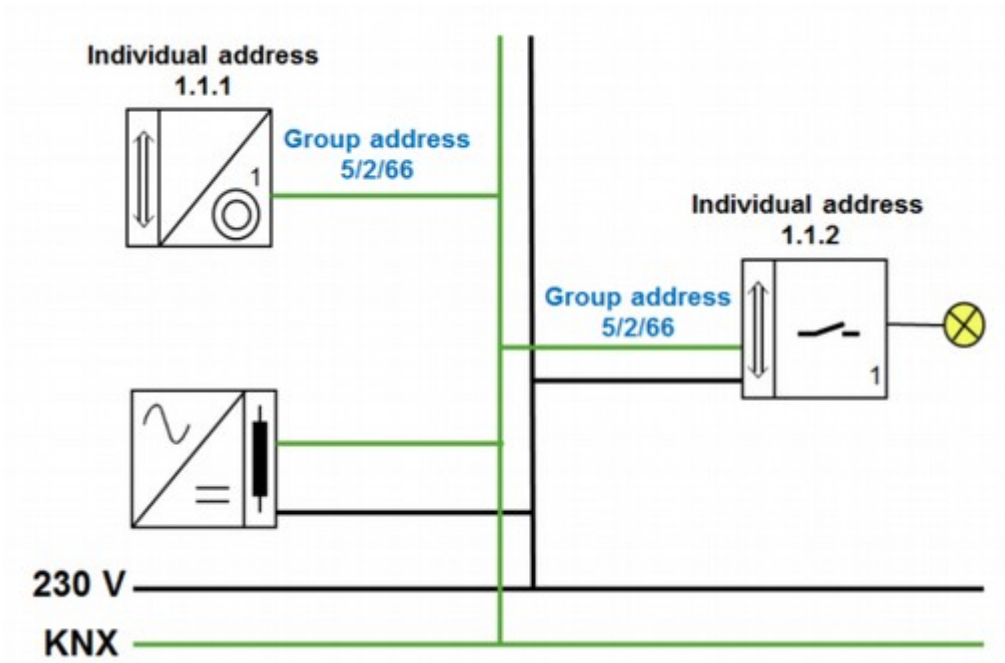


Figuur 3: Versturen van 1bit op de KNX-TP bus (Basiscursus KNX, 2016)

2.2.3 Bus topologie

De minimale opbouw van een KNX installatie bevat 3 componenten, een voeding, een sensor en een actor. De voeding is nodig voor de spanning op de bus en als voeding voor de aangesloten componenten. Alle componenten worden aangesloten op 1bus, zo kunnen al deze componenten met elkaar communiceren.

De opbouw is te zien in figuur 4.



Figuur 4: Minimale bus structuur met groep adressen (Basiscursus KNX, 2016)

Elke component op de bus dient een uniek individueel adres te hebben, maar deze worden enkel gebruikt voor het doorgeven van "programming telegrams" voor het programmeren/parametriseren van componenten. Het individueel adres wordt aan de component gegeven via de ETS software van KNX.

De echte communicatie gebeurt op basis van groep adressen, 16-bit getallen (opgesplitst in 3 delen voor structuur), waaraan de programmeur 1 functie geeft. Een voorbeeld hiervan is het aandoen van 1 lamp of lichtkring. Als een busdeelnemer dan een logische "1" of "0" zendt op het gekozen groep adres, en luistert de actor met de lamp op dit groep adres naar telegrammen. Alle communicatie gebeurd via de groep adressen, ze zijn dus de basis van alle communicatie in een KNX installatie.

Er zijn in totaal 65535 groep adressen beschikbaar voor communicatie.

3 Embedded Linux

3.1 Wat is embedded Linux

Embedded Linux is de algemene naam voor embedded systemen waarop een operating system draait dat gebaseerd is op de Linux-kernel.

De belangrijkste verschillen tussen een embedded Linux systeem en een "Gewoon" Linux systeem is de verschillen in beschikbare resources. Zo beschikt een embedded systeem niet over gigabytes aan ram en opslag, maar zijn deze eerder beperkt afhankelijk van de toepassing. Daarom wordt er een slankere versie van de Linux-kernel gebruikt waarin enkel de delen die echt nodig zijn mee inzitten, zoals network en een simpele command-line-interface maar vaak geen display manager.

3.2 Linux-kernel

De Linux-kernel is de basis van elk Linux systeem, het verzorgt de interactie met de hardware en stelt een application programming interface of API ter beschikking waarmee user programma's gemaakt kunnen worden die interactie kunnen hebben met de kernel. Deze API bestaat voornamelijk uit system call's.

De Linux-kernel heeft verschillende taken, zoals memory & network management maar ook scheduling van meerdere taken en processen waardoor het een multitasking operating system wordt.

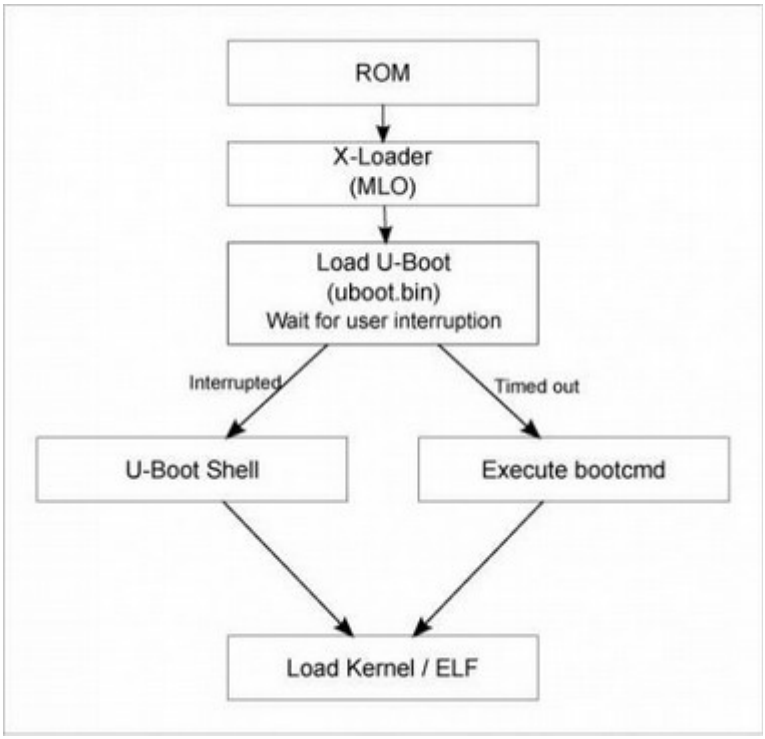
De kernel bevat zelf nog geen extra user software zoals bijvoorbeeld een shell, deze moet aangeleverd worden door het root filesystem. Om de kernel extra functionaliteit te geven wordt er gebruik gemaakt van drivers/modules. Deze drivers kunnen ofwel gecompileerd worden in de kernel of ze worden na het booten in runtime ingeladen. Dit inladen kan op verschillende manieren gebeuren, een van die manieren is het gebruik van een device tree zoals we verder in dit hoofdstuk zullen bespreken.

3.3 Boot proces

Elke vorm van embedded Linux dient opgestart te worden door een compatibele bootloader, dit is nodig omdat de Linux-kernel op een specifieke manier opgestart moet worden.

Meestal gebeurd het booten in 2 fases, een embedded controller heeft meestal een eigen mini bootlader, deze bootloader kan gaan zoeken op een ander medium (flash, SD, USB, ...) naar een andere bootloader die de user daar gezet heeft. In een embedded omgeving zal dit vaak Das U-Boot of simpelweg U-Boot zijn. Deze U-boot kan dan via zijn parameters en scripts de Linux-kernel opstarten. Het basis principe is te zien in figuur 5.

U-Boot heeft in het meest simpele geval 3 taken uit te voeren. Eerst word de Linux-kernel en de bijhorende device tree ingeladen in ram. Vervolgens worden er een paar kernel bootargumenten gespecificeerd, bijvoorbeeld de boot console waarop de boot output te zien zal zijn. Als laatste stap wordt de Linux-kernel gestart.



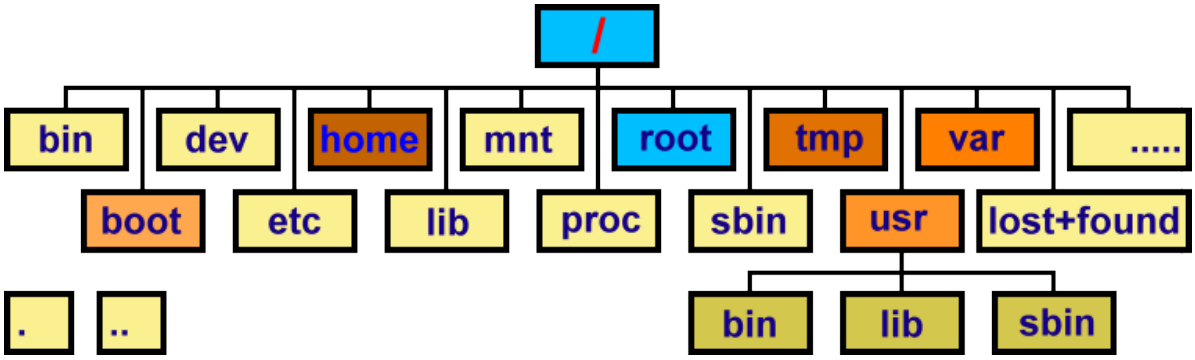
Figuur 5: Boot proces (Bharath Bhushan Lohray, 2013)

3.4 Root Filesystem

Een Linux root filesystem is het punt dat de kernel zal gebruiken om programma's en data te zoeken. Nadat de bootlader de Linux-kernel opgestart heeft, zal de kernel dit filesystem mounten als zijn root. Na het starten zal de kernel zoeken naar het init script dat zich moet bevinden in een specifieke plaats op het filesystem, daardoor kunnen alle benodigde processen opgestart worden, en krijg je een werkend systeem.

Zoals elk Linux systeem heeft ook een embedded systeem een filesystem nodig. Dit zal in veel gevallen echter geen full GNU/Linux zijn zoals Debian of Ubuntu, maar dat is zeker niet uitgesloten. Vaak wordt er gebruik gemaakt van veel lichtere software zoals Busybox in combinatie met µClib. Dit is uiteraard omdat deze gemaakt zijn om op systemen met minder resources te werken.

Busybox neemt 2 van de belangrijkste taken op zich, namelijk alle Unix-tools, zoals ls, cp en cat, maar het bevat ook het basic init system voor het opstarten van het systeem.



Figuur 6: Voorbeeld root filesystem (informatics.buzdo.com, 2017)

3.5 Device tree

Een Device tree is een data structuur, gebaseerd op nodes met sub-nodes en properties met een naam en een waarde, dat gebruikt wordt om hardware te beschrijven. Dit systeem maakt het mogelijk om 1 gecompileerde kernel te laten werken op meerdere verschillende platformen. Dit is mogelijk omdat de kernel dan niet meer alle drivers ingecompileerd krijgt, maar als laadbare modules. Als de kernel dan de device tree leest, kan hij de juiste modules/drivers inladen tijdens het booten.

Dit systeem wordt momenteel toegepast in bijna alle ARM-based embedded Linux systemen, maar is niet zo populair in de andere werelden.

De device tree wordt door de bootloader in memory geladen voordat de kernel opgestart wordt.

4 Praktische realisatie

Het doel van de realisatie is het opzetten van een embedded Linux platform dat het mogelijk moet maken voor een aantal "High-level" webapplicaties om te communiceren met de KNX bus. Door middel van dit platform zou het dus mogelijk moeten zijn om eender welke IoT toepassing te schrijven en te koppelen aan de KNX-bus.

4.1 Keuze ontwikkelomgeving

Er wordt momenteel enkel een prototype gemaakt, maar het is de bedoeling om dit in de toekomst eventueel om te vormen naar een werkend product. Daarom is het belangrijk om de juiste ontwikkelborden te kiezen zodat het mogelijk blijft om dit te realiseren.

4.1.1 Embedded Linux platform

De belangrijkste component van dit geheel is het ontwikkelbord waarop de Linux-kernel zal draaien, en uiteindelijk alles zal aansturen, en de webserver draaien.

Voor deze taak waren uiteraard een heel aantal kandidaten met elk hun eigen voor en nadelen.

De belangrijkste dingen waarnaar er gezocht is zijn:

– Open source schema's

– Gebruikte componenten zijn verkrijgbaar en mogelijk certificeerbaar

– Goede documentatie/support

– Een goede veelgebruikte Arm Cortex-A processor

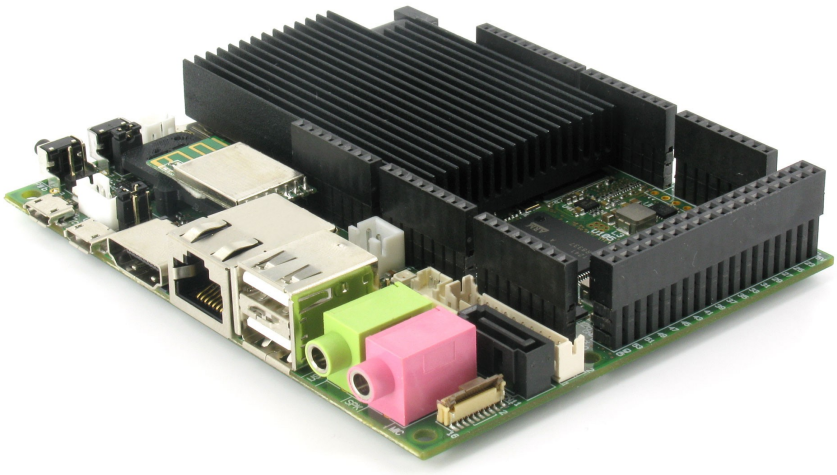
– Aanvaardbare prijs

De vergelijking van de verschillende mogelijkheden is te vinden in tabel 1.

Tabel 1: Vergelijking developement boards

Naam	Prijs	Voordelen	Nadelen
Udoo neo	50€	Veel documentatie Goedkoop Goede support Componenten goed verkrijgbaar Hardware i2c interface op cortex a9 Veelgebruikte freescale cpu	Minder peripherals voor testing
OpenRex	200€	Open source Arduino & rpi-pinout Veel i/o & pheriphirals Veelgebruikte freescale cpu	Minder documentatie beschikbaar Duur Weinig support
Udoo QUAD	135€	Veel documentatie Goede support Componenten goed verkrijgbaar Veel peripherals voor testing/uitbreiding Schema's ter beschikking Veelgebruikte freescale cpu Hardware i2c interface op cortex a9	Duurder
Raspberry pi	35,00 €	Veel documentatie Goede support Veel peripherals voor testing/uitbreiding	Geen open design Componenten niet verkrijgbaar

Het bord dat uiteindelijk gekozen is voor het prototype is het UDOO Quad bord. Dit bord voldoet aan alle voorwaarden, en was al beschikbaar op school om te gebruiken.



Figuur 7: UDOO Quad (udoo.org, 2017)

4.1.2 Keuze KNX-TP interface

Voor de communicatie met de KNX-bus is er een interface nodig om over te gaan naar de 29V bus. Om dit te doen zijn er 2 mogelijkheden.

– Uart-tp overgang

– Uart-tp overgang + µc met KNX-stack op module

Aangezien er op 4 maanden niet genoeg tijd is om een hele KNX-stack te schrijven, is er voor de 2de optie gekozen. Dit heeft ook als voordeel dat een eventuele KNX certificering makkelijker zou zijn omdat er een officiele module gekozen kan worden.

In de 2de optie met een onboard KNX-stack zijn er uiteraard verschillende mogelijkheden van verschillende fabrikanten, in tabel 2 bevinden zich een paar van de modules die beschikbaar zijn op school.

Tabel 2: Keuze KNX interface

Naam	Prijs	Voordelen	Nadelen
Tapco SIM_KNX	76€	Makkelijk testen via uart Galvanische scheiding	Duurder
Tapco kimap	26€	Goedkoper I2c interface	Geen galvanische scheiding
WEINZIERL KNX Tiny Serial Module 810-USB	90€	Galvanische scheiding Usb via fdti Makkelijk testen via uart	Duurder

De keuze is uiteindelijk gegaan naar de SIM-KNX van Tapco, de redens hiervoor zijn:

– Kleine module

– Uart interface

– Niet al te duur

– Een galvanische scheiding tussen de bus en de uart

Figuur 8: Tapco SIM-KNX (tapko.de, 2017)

4.2 Embedded Linux omgeving

Voor het embedded Linux systeem is er gebruik gemaakt van het UDOO ontwikkelbord. In eerste instantie is er gebruik gemaakt van de officiële UDOO image, deze was echter zo instabiel dat er gekozen is om een eigen Linux omgeving op te zetten.

4.2.1 Bootloader

Als bootloader is er gekozen voor Das U-Boot omdat deze ook gebruikt wordt door UDOO zelf, en deze in het algemeen zeer veel gebruikt wordt. U-Boot biedt ook support voor device trees en er is veel documentatie beschikbaar op internet.

Hiervoor is er begonnen vanaf de mainline U-Boot met een standaard configuratie voor het UDOO bord, maar dit had evengoed de configuratie voor de imx6 processor kunnen zijn.

De gecompileerde bootloader wordt dan in 2 delen op de SD-kaart gezet op specifieke plaatsen die bepaald zijn door de interne bootloader van de processor.

Om U-Boot automatisch te laten opstarten moeten we hem nog environment variabelen geven zodat hij weet hoe hij de Linux-kernel moet opstarten.

Tabel 3: U-boot argumenten

Boot-argumenten	
Naam	**Waarde**
baudrate	115200
bootargs	console=ttymxc1,115200 root=/dev/mmcblk0p2 rootwait panic=100
dtb_name	dts/oli_udoo.dtb
load_dtb	fatload mmc 0:1 0x18000000 dts/dtb_name
load_image	fatload mmc 0:1 0x12000000 zImage
start_kernel	bootz 0x12000000 – 0x18000000
bootcmd	run load_image; run load_dtb; run start_kernel

In tabel 3 is te zien dat een simpel systeem opgestart kan worden met slechts een paar kleine commando's. Het belangrijkste commando is "bootcmd", dit argument wordt automatisch uitgevoerd door U-Boot bij het opstarten. Hierin worden dan ook de andere argumenten opgeroepen zoals het inladen van de kernel en de device tree.

4.2.2 Linux-kernel

Voor de Linux-kernel is er begonnen met de kernel van de officiële UDOO image, maar er is al snel overgestapt naar een nieuwer exemplaar gecompileerd vanuit de mainline tree in combinatie met de de standaard configuratie vanuit de kernel tree voor de Freescale imx6-quad processor. Op deze manier heb ik de meeste bewegingsvrijheid qua interne en externe modules.

De belangrijkste kernel features zijn:

- Ethernet support
- I^2C device support
- UART device support
- GPIO device support

4.2.3 Device tree

Om vanuit het running Linux systeem de peripherals zoals UART, I^2C en GPIO aan te kunnen spreken, is het nodig om de Linux-kernel kenbaar te maken welke drivers hij moet laden om dit te doen. Dit gebeurd via de device tree.

Voor het maken van de device tree ben ik begonnen met het analyseren van de meegeleverde tree vanuit de officiële UDOO image, hieruit heb ik een aantal zeer belangrijke concepten gehaald die ik dan kon toepassen op 1 van de standaard tree's die zich bevind in de mainline kernel.

In de kernel-tree bevindt zich een heel basis device tree voor het UDOO bord, deze heb ik als basis gebruikt. Hieraan heb ik een paar nodes toegevoegd om I^2C support te krijgen. Zoals te zien in figuur 9 en 10, moet eerst de interface gedefinieerd worden zoals hij aangesloten is op de processor (d.m.v. de pinnen), en daarna moeten de interfaces aangemaakt worden en krijgen ze status "okay" omdat ze actief mogen zijn.

Hetzelfde is gedaan voor de UART om deze ook te kunnen gebruiken.

```
&iomuxc {
        imx6q-udoo {

                // I2c1 = external i2c
                pinctrl_i2c1: i2c1grp {
                        fsl,pins = <
                        MX6QDL_PAD_EIM_D21__I2C1_SCL            0x4001b8b1
                        MX6QDL_PAD_EIM_D28__I2C1_SDA            0x4001b8b1
                        >;
                };

                // I2c2
                pinctrl_i2c2: i2c2grp {
                        fsl,pins = <
                                MX6QDL_PAD_KEY_COL3__I2C2_SCL   0x4001b8b1
                                MX6QDL_PAD_KEY_ROW3__I2C2_SDA   0x4001b8b1
                        >;
                }
        };
};
```

Figuur 9: I2c io mapping in device tree

```
&i2c2 {
        clock-frequency = <100000>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_i2c2>;
        status = "okay";
};

&i2c1 {
        clock-frequency = <100000>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl_i2c1>;
        status = "okay";
};
```

Figuur 10: I2c definitie in device tree

4.2.4 Filesysteem

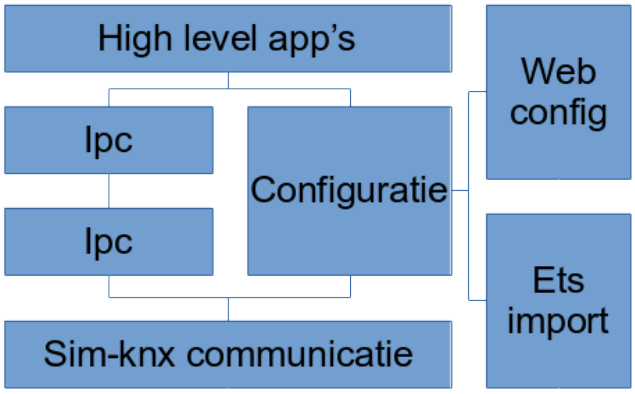
Voor het root filesysteem van het embedded Linux platform was er in eerste instantie gekozen voor het compileren van een Buildroot filesysteem. Dit is een heel basis systeem waarop enkel de minimaal benodigde programma's aanwezig zijn voor een werkend systeem. Dit heeft als grote voordeel dat er volledige controle is over elk proces dat mag en moet runnen. Dit filesysteem is dan ook aangemaakt en uitbundig gebruikt voor debuggen en testen van de kernel, de device tree en het C++ programma.

Dit systeem had echter 1 nadeel, er is namelijk geen C/C++ compiler aanwezig die op het platform zelf kan werken. Dit is normaal geen probleem omdat er gebruik kan worden gemaakt van cross-compilatie zodat alles op de laptop gecompileerd kon worden voor het target. Node.js, het platform waarmee er verder in dit werkstuk de Homekit integratie gemaakt zal worden, moet echter een aantal pakketten zelf kunnen compileren om homebridge te laten werken.

Daarom is er overgegaan naar een zelf gecompileerd Debian filesysteem omdat deze wel een eigen compiler geeft. Een extra voordeel is dat er nu gebruik gemaakt kan worden van de Debian pakkage manager.

4.3 Software

Om het geheel te laten werken is er een heel deel code geschreven. Alles kan onderverdeeld worden in verschillende lagen, deze gaan we wat dieper bespreken.

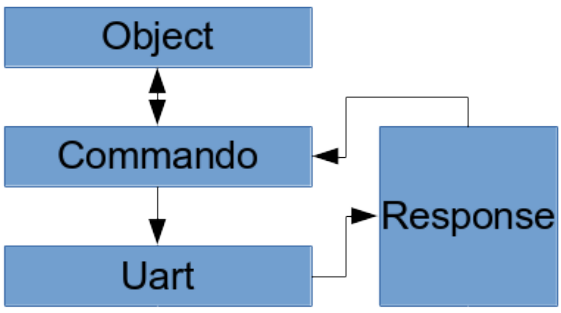


Figuur 11: De software lagen

4.3.1 SIM-KNX communicatie

De communicatie met de SIM-KNX module is een van de belangrijkste en meest complexe lagen, dit is omwille van de uitgebreide instructieset van de module. Om de complexiteit en de onduidelijkheid van de instructieset te verbergen, is er gekozen om het programma te schrijven in C++ en alles om te zetten in een aantal classen en te gaan voor een objectgeoriënteerde structuur om alles te splitsen en leesbaarder te maken.

De SIM-KNX module werkt op basis van "objecten", 1 object representeert een deelnemer op een bepaald groep adres op de bus. Heel de communicatie met de bus verloopt dus via deze objecten.



Figuur 12: De verschillende klassen

Door het gebruik van deze classen kan het volledig configureren en aansturen van SIM-KNX “objecten” gebeuren d.m.v. dit stukje code:

```
Object obj(1);

obj.setInteroperability(DPT1_BOOLEAN);   // Boolean object
obj.setSendConfig(0x00);                 // Geen autosend
obj.setType(BOOL);                       // Type voor intern
obj.sendConfiguration();

obj.setSendingAddr("1/1/1");             // Add Group adres

obj.sendBool(1);
```

Hierin wordt een object aangemaakt met nummer 1, deze wordt ingesteld als Boolean met een bepaalde configuratie, en de waarde 1 wordt naar de bus verstuurd. De ingestelde configuratie beïnvloed de manier waarop bepaalde telegrammen worden verstuurd of niet verstuurd al dan niet automatisch.

Wat er in dit stukje code eigenlijk gebeurd is dat de objecten 3 commando’s aanmaken zoals te zien in tabel 4.

Tabel 4: Basis commando's

ocs (1) 1 0 * 0 * 0	Configureer object 1 als boolean
ogs (1) 1/1/1	Set het groep adres voor object 1
ovs (1) 1	Stuur een 1 via object 1

Deze commando’s worden dan door de C-Linux termios library en het Linux tty-device dat we via de device tree aangemaakt hebben, via UART gestuurd naar de SIM-KNX module. Deze module begrijpt deze commando’s en zal de nodige acties ondernemen.

4.3.2 SIM-KNX configuratie

De bedoeling van heel het platform is om het zo open mogelijk te houden zodat meerdere applicaties naast elkaar kunnen werken en met de bus kunnen communiceren. Daarom is het belangrijk om alles zoveel mogelijk configureerbaar te maken, dit is gedaan door gebruik te maken van een dynamisch ingeladen configuratie file.

Deze file bevat een JavaScript Object Notation of JSON representatie van alle objecten die met de KNX installatie kunnen communiceren. Een object kan gezien worden als een deelnemer op een bepaald groepsadres in de KNX installatie, zo kan 1 object bijvoorbeeld het sturen van 1 lichtkring zijn.

Op basis van deze file kan het C++ programma deze objecten aanmaken in de SIM-KNX module zodat verdere communicatie mogelijk is.

Een object in de JSON file ziet er als volgt uit:

```
{
    "Naam": "Licht keuken",
    "Type": 1,
    "SendAddr": "1\/1\/1",
    "homekit": true,
    "homekitNaam": "Licht keuken",
    "uname": "lichtkeuken_1_111",
    "Soort": "Lamp",
    "SchakelObject": "0",
    "FeedbackObject": "0"
},
```

De belangrijkste parameters voor SIM-KNX zijn het type en het zendadres, alle andere parameters zijn ter extra info voor de gebruiker in de web configuratie, maar worden ook gebruikt door HomeKit zoals we in een verder hoofdstuk zullen bespreken.

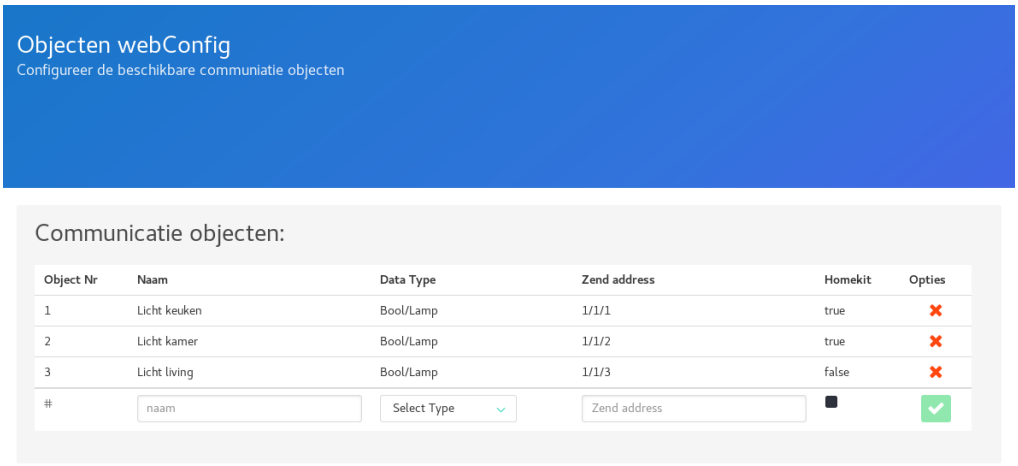
Zoals te zien is zijn er geen objectnummers toegekend aan deze objecten, dit komt omdat deze van volgorde kunnen veranderen in de loop van de tijd. Daarom is er geopteerd om de objectnummers dynamisch toe te kennen bij het opstarten van de applicatie beginnende vanaf 1 op de volgorde dat de nodes zich in de file bevinden.

Om het werken met deze tekstuele JSON files eenvoudiger te maken, zijn er 2 webtools ontwikkeld, namelijk de web configuratie en de ETS import. Deze 2 tools zijn beide beschikbaar via de welkomstpagina op de locale webserver.

4.3.2.1 Web configuratie

De meest eenvoudige vorm is een webinterface waarin alle reeds gemaakte objecten zichtbaar zijn en nieuwe objecten handmatig geconfigureerd kunnen worden.

Deze webinterface is een HTML met Javascript frontend die door middel van AJAX requests samenwerkt met een PHP backend die de configuratie omzet naar JSON en opslaat in een file.



Figuur 13: Webinterface voor configuratie

4.3.2.2 ETS import

Ets import is de meeste geavanceerde methode om de configuratie file aan te maken. Hierbij wordt er verwacht dat eerst een volledige KNX installatie geconfigureerd wordt via de ETS software. Belangrijk is hierbij dat ook de juiste datatypes toegekend worden aan de groep adressen. Het is dan de bedoeling dat vanuit ETS een file wordt geëxporteerd die alle informatie bevat over de groep adressen die beschikbaar zijn. Deze file in XML formaat dient dan ingeladen te worden in de ETS-import webinterface waarna men een aantal configuratiemogelijkheden krijgt voor het kiezen van stuurobjecten(dimmers, rolluiken en lampen), en eventueel bijhorende feedback objecten.

Deze webinterface in omwille van de complexiteit en het verwerken van file uploads opgebouwd in het Laravel PHP framework. Het gebruik van het framework heeft als voordeel dat het de tijd benodigd om deze applicatie te ontwikkelen drastisch verminderd en het project vereenvoudigd.

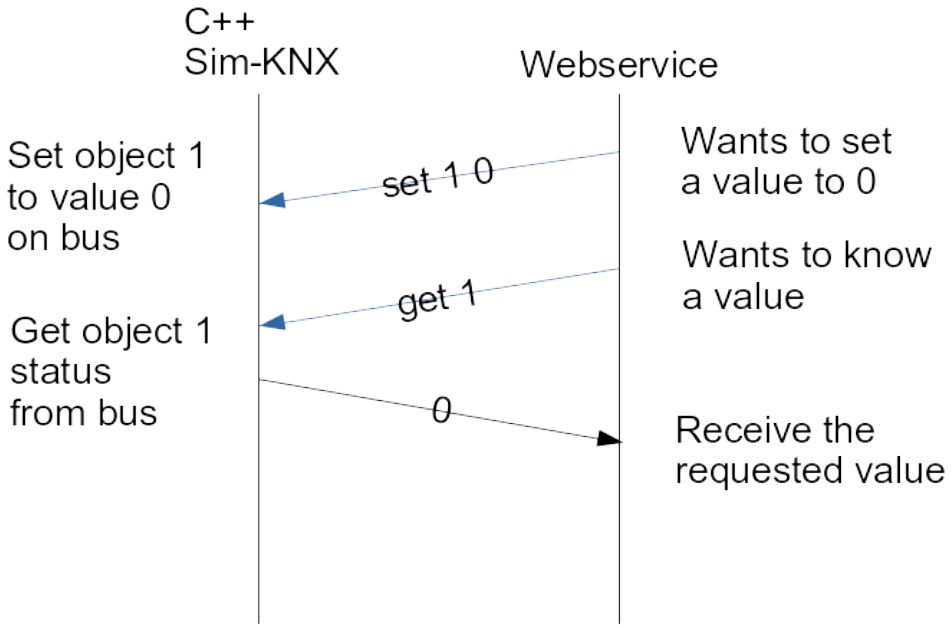
4.3.3 Interprocescommunicatie

Interprocescommunicatie of IPC is de communicatie tussen een of meerdere processen in een multitasking besturingssysteem. Dit is uiteraard een zeer belangrijke bouwsteen voor dit project, het laat immers dat meerdere applicaties via eenzelfde interface toegang krijgen naar de bus toe. Door gebruik te maken van IPC moet het mogelijk zijn om het C++ programma te koppelen aan een aantal high level applicaties waarin het mogelijk is om een aantal webservice's te programmeren voor de KNX bus.

Om de toegankelijkheid zo groot mogelijk te houden en geen mogelijke programmeertalen of platformen uit te sluiten, is er gekozen om voor de IPC gebruik te maken van het User Datagram Protocol of UDP sockets. UDP laat ons toe om op een snelle universele manier te communiceren op basis van IP-adressen en poorten. Het wordt hier lokaal gebruikt op de loopback interface, maar het is perfect mogelijk om deze UDP poort open te stellen en communicatie te hebben met andere systemen.

```
int sock = initSocket(1234);
recvlen = recvfrom(sock, buffer, 50, 0, &remaddr, &addrlen);

if(recvlen > 0)
{
    request.clear();
    parseRequest(buffer, request);
    respondTo(request, sock, remaddr);
    memset(buffer,0,50);                    // Maak buffer leeg
}
```

In bovenstaande code snippet is te zien hoe er in het C++ programma een UDP socket wordt aangemaakt, waarna er de recvfrom functie opgeroepen wordt. Deze functie is een blocking call, dit wil zeggen dat deze functie niet voortgaat totdat er data ontvangen is via UDP. Als de functie dus returnt is er normaal data ontvangen, dit wordt dan nog even nagekeken met een if-statement. Hierna wordt de ontvangen data geïnterpreteerd en het eventuele request dat zich bevond in het UDP packet wordt opgeslagen in de request variabele. Deze request variabele wordt gebruikt om de juiste actie te ondernemen, een waarde zetten op de bus of iets ervan lezen en eventueel een antwoord terug te sturen. De UDP buffer wordt tenslotte leeggemaakt voordat opnieuw de recvfrom functie opgeroepen wordt.



Figuur 14: Voorbeeld IPC met udp sockets

In Figuur 14 is een voorbeeld te zien over hoe de interprocescommunicatie juist werkt. In dit voorbeeld zijn er 2 deelnemers namelijk: de server (SIM-KNX C++ programma) en een voorbeeld webservice.

De webservice wil graag een waarde op de bus plaatsen, de programmeur van deze service heeft alles geconfigureerd en heeft een object aangemaakt in de configuratie file (hoofdstuk 4.2.2) en heeft als objectnummer 1 gekregen.

De webservice stuurt nu een UDP pakket met de volgende structuur:

"*<commando> <objNummer> <?waarde>".*

In het eerste geval is dit het commando set, dat commando zal dus de waarde 0 doorsturen via object 1.

Indien de webservice nu een waarde zou willen weten van de bus, stuurt deze een get commando samen met een objectnummer. De server zal dan de bus raadplegen en de waarde terugsturen naar de service die erom gevraagd heeft over dezelfde UDP socket.

4.3.4 Software testing

Bij het verder uitbreiden en optimaliseren van dit project zou het nodig zijn om alles verder te kunnen verfijnen, en eventuele bugs en problemen er verder uit te halen. Dit kan gebeuren door alle classes aan een aantal grondige tests te onderwerpen. Als proof of concept zal nu de basis van unit tests besproken worden, en zijn er een aantal tests geïmplementeerd als voorbeeld.

Unit testing is een veelgebruikte methode om een heel klein stukje broncode, bijvoorbeeld 1 functie, afgezonderd te testen. Er wordt een hele testsuite opgebouwd die bestaat uit een heel aantal testen die onafhankelijk zijn van elkaar uitgevoerd worden. Het doel van een unit test is om na te gaan of elk stukje van de broncode op zich zijn eigen taak correct uit kan voeren. In een later stadium geeft een uitgewerkte testsuite ook als voordeel dat bij elke wijziging aan de code nagegaan kan worden of alles nog correct werkt door alle tests uit te voeren.

Om dit in de praktijk toe te passen dient er gebruik gemaakt te worden van een zogenaamd test framework specifiek voor de gebruikte of te gebruiken programeertaal. Voor de voorbeeld tests heb ik gekozen om gebruik te maken van het Google Test framework omdat dit momenteel het meest gebruikte C++ test framework is, en er een rechtstreekse integratie bestaat met bepaalde C++ ontwikkelomgevingen.

Na het importeren en het configureren van het Make systeem, in mijn geval CMake, wordt er bij het compileren een extra executable aangemaakt. Bij het uitvoeren van dit bestand krijgt men dan het rapport te zien van alle tests.

Als voorbeeld is er een mini testsuite geschreven om de Commando klasse te testen. Deze klasse heeft als doel een commando op te bouwen dat later via UART verzonden kan worden naar de SIM-KNX module. De eerste test in de suite is te zien in volgende code snippet:

```
TEST(Commando_tests, a_commando_can_be_build_via_the_constructor)
{
    ASSERT_NO_THROW(Commando testCommando("test"));
}
```

Deze eenvoudige test probeert een nieuw object aan te maken van de Commando klasse en maakt dan een assertie, een waar of onwaar op een bepaalde plaats, dat er geen error zal plaatsvinden. In de volgende snippet is de output te zien van de test:

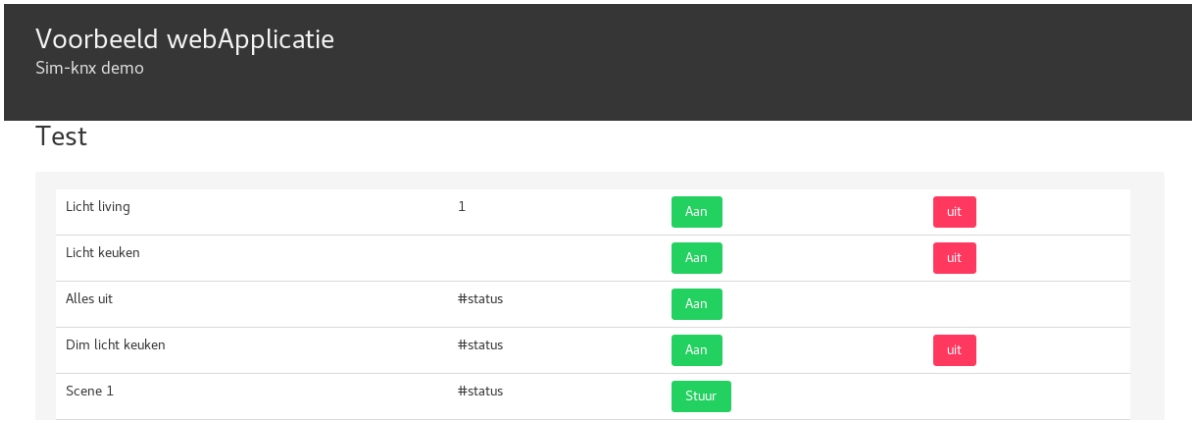
```
➜  Bachelorproef/Code/sim-knx/build/test master ✗ ./runUnitTests
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from Commando_tests
[ RUN      ] a_commando_can_be_build_via_the_constructor
[       OK ] a_commando_can_be_build_via_the_constructor 0m
[----------] 1 test from Commando_tests (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (0 ms total)
[  PASSED  ] 1 test.
```

Er is te zien dat dat de executable de testsuite start, en in dit geval de enige test uitvoert. Als alle test dan uitgevoerd zijn krijgt de gele testsuite een “passed” of een “failed” status.

4.4 Integratie van toepassingen

4.4.1 Website controle

Als eerste toepassing is er een basis web interface gemaakt met als doel het kunnen besturen van de bus via de webbrowser. De pagina is met opzet zeer eenvoudig gehouden en de configuratie gebeurt manueel via de code. Zoals te zien in figuur 15 zijn er manueel een aantal rijen aangemaakt in een tabel. Elke rij heeft een titel en 2 knoppen, deze knoppen dienen om het object aan of uit te sturen. Deze statische webpagina maakt gebruik van AJAX om het indrukken van de knoppen door te sturen naar een PHP script. Dit PHP script zal dan op zijn beurt een UDP pakket sturen zodat de data op de bus terecht kan komen.



Figuur 15: Webinterface voor besturing

Door 1 record aan te maken in de tabel zoals te zien in onderstaande code-snippet, kunnen er extra records aangemaakt worden op de webpagina. Het belangrijkste aan deze knoppen is het "@click" attribuut. Dit wordt gebruikt om aan de javascript door te geven dat in dit geval aan object 1 een "0" of een "1" gestuurd moet worden.

```
<tr>
        <td>Licht living</td>
        <td>{{ status.Living }}</td>
        <td>
                <button class="button" @click="send(1,1)">Aan</button>
        </td>
        <td>
                <button class="button" @click="send(1,0)">uit</button>
        </td>
</tr>
```

4.4.2 Apple HomeKit

HomeKit is een platform van Apple om via een IOS apparaat zoals een iPhone of een iPad op een universele manier accessoires in uw huis aan te sturen ongeacht het merk. Het laat dus toe om heel het huis te besturen niet alleen via de app, maar rechtstreeks vanuit heel het IOS apparaat en dus ook via Siri.

Het zou dus heel mooi zijn moest het KNX-IoT platform ook een HomeKit compatibel accessoire zijn. Omdat het vanaf 0 implementeren van de compatibiliteit teveel werk zou zijn voor de korte tijdsspanne van de bachelorproef, is er gekozen om vanaf een bestaand, werkend en open source platform te beginnen.

Het platform dat gebruikt is als basis is het Homebridge project. Homebridge is een lichte NodeJs server die binnen een netwerk een HomeKit API simuleert. Dit project op zich heeft geen functionaliteit, deze moet toegevoegd worden door middel van plugins. Voor meer informatie over het installeren en toevoegen van plugins kan u de documentatie lezen in de bijlage “Installeren van Homebridge”.

Om een zo open en configureerbaar mogelijke implementatie te maken voor de verbinding van Homebridge met KNX-IoT is er een plugin aangemaakt. De aangemaakte plugin is een zogenaamde “Platform plugin”, dat wil zeggen dat de plugin niet instaat voor 1 HomeKit accessoire maar dat er meerdere accessoires dynamisch aangemaakt kunnen worden.

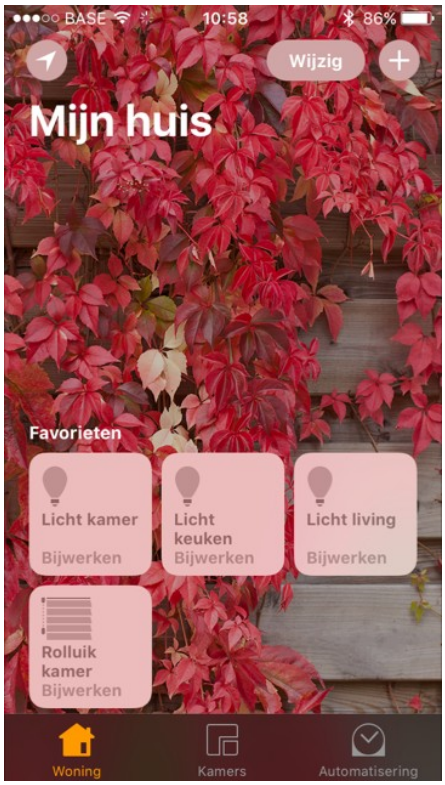
Bij het opstarten van Homebridge wordt de configuratiefile die gemaakt is door de ETS-import tool automatisch ingeladen. Uit deze configuratie worden alle objecten waar de “HomeKit” parameter op true staat toegevoegd. Deze configuratie parameters zijn besproken in hoofdstuk 4.3.2. Op basis van het “Type” en de “Soort” wordt er een nieuw accessoire aangemaakt en geconfigureerd. Dit is te zien in onderstaande code-snippet.

```
if(object.Type == 1)
{
  newAccessory.addService(Service.Lightbulb, object.homekitNaam);
}
else if(object.Type == 3)
{
  // Absoluute sturing (roluik of dimmer)
  if(object.Soort == "Dimmer")
  {
    newAccessory.addService(Service.Lightbulb, object.homekitNaam);
  }
  else if(object.Soort == "Rolluik")
  {
    newAccessory.addService(Service.WindowCovering, object.homekitNaam);
  }
}

this.configureAccessory(newAccessory);
```

Het configureren van een nieuw accessoire is de meest ingewikkelde stap. Hierbij moet er aan elk accessoire een aantal functies toegekend worden, deze functies zullen opgeroepen worden als de gebruiker data opvraagt of data wil sturen naar de bus. Dit is te zien in onderstaande code snippet.

```
// Boolean licht object
accessory.getService(Service.Lightbulb)
.getCharacteristic(Characteristic.On)
.on('set', function(value, callback) {

    platform.log("Set licht -> " + value);
    platform.sendValue(objectNumber, value);
    callback();
})
.on('get', function(callback) {

    platform.log("Get licht");
    platform.askValue(objectNumber, callback);
});
```

Zoals te zien wordt er aan 2 event's een functie gekoppeld. Deze "set" en "get" events worden automatisch opgeroepen door het Homebridge framework als er via Homekit iets veranderd aan het accessoire. In de event's worden er 2 eigen functies opgeroepen, de "sendValue" en de "askValue". Deze functies sturen zoals gewoonlijk de data over UDP naar de SIM-KNX module.



Figuur 16: Werkende HomeKit app

Literatuurlijst

Apple HomeKit. (s.a.). Gevonden op 15 mei 2017 op het internet: https://www.apple.com/ios/home/

Das U-Boot. (s.a.). Gevonden op 26 maart 2017 op Wikipedia: https://en.wikipedia.org/wiki/Das_U-Boot

Device tree. (s.a.). Gevonden op 23 maart 2017 op Wikipedia: https://en.wikipedia.org/wiki/Device_tree

Google Test. (2016). Gevonden op 26 februari 2017 op het internet: https://github.com/google/googletest

Homebridge. (2017). Gevonden op 15 mei 2017 op het internet: https://github.com/nfarina/homebridge

KNX Association. (2016). *Basiscursus KNX ETS5*

Linux kernel. (s.a.). Gevonden op 22 maart 2017 op het internet: https://www.kernel.org/

SIM-KNX documentatie. (s.a.). Gevonden op 15 februari 2017 op het internet: http://www.tapko.de/en/sim-knx.html

UDOO gegevens. (s.a.). Gevonden op 15 februari 2017 op het internet: https://www.udoo.org/

Bijlagen