



Faculteit Bedrijf en Organisatie

De mogelijkheden van Swift voor Arduino en Raspberry Pi binnen de context van Internet of Things:
onderzoek en proof of concept

Martijn Bogaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Steven Van Impe
Co-promotor:
Carl Peto

Instelling: Swift for Arduino

Academiejaar: 2020-2021

Tweede examenperiode

Faculteit Bedrijf en Organisatie

De mogelijkheden van Swift voor Arduino en Raspberry Pi binnen de context van Internet of Things:
onderzoek en proof of concept

Martijn Bogaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Steven Van Impe
Co-promotor:
Carl Peto

Instelling: Swift for Arduino

Academiejaar: 2020-2021

Tweede examenperiode

Woord vooraf

Voor mij is de brug tussen soft- en hardware altijd een belangrijke bron van interesse geweest. Ik schreef deze scriptie dan wel na drie jaar toegepaste informatica gevolgd te hebben, ik ben de hardwarekant van het verhaal niet uit het oog verloren. Mijn plan bestaat er dan ook in om na mijn huidige opleiding nog een masterdiploma in de industriële wetenschappen te behalen en zo mijn vakkennis uit te breiden. Om die reden wilde ik met mijn scriptie alvast wat terrein verkennen en toen meneer Van Impe op de proppen kwam met een voorstel dat rond de aansturing van elektronica-apparaten draaide, wist ik dat ik moest toehappen. Bij deze wil ik dus allereerst meneer Van Impe bedanken. Zonder de aanzet was ik zelf nooit op het idee gekomen waaruit deze scriptie uiteindelijk is voortgevloeid. Daarnaast ben ik meneer van Impe ook ontzettend dankbaar voor de positieve en waardevolle feedback doorheen het schrijfproces, maar vooral ook voor het openen van een nieuwe wereld voor mij.

Meneer Van Impe bracht me immers in contact met Carl Peto, de bezieler van het Swift for Arduino-project, en de volledige daarbij horende community. Vooral bij het opstellen van mijn proof of concept, heeft Carl mij ongelooflijk geholpen. Ik hoefde Carl nog maar een berichtje te sturen of ik kreeg vijf minuten later al een antwoord in het lang en in het breed, vaak vergezeld van een codevoorbeeld of twee. Carls expertise heeft mij versteld doen staan, in die mate zelfs dat ik in mijn conclusie moest laten optekenen dat een bepaald deel van de doelgroep zich in bepaalde omstandigheden beter niet aan het onderwerp waagt, het zou het petje van de *gemiddelde softwareontwikkelaar* immers te boven gaan. Dat ik ook Carl ontzettend dankbaar ben, behoeft dus geen verder betoog.

Het is ongetwijfeld een ontzettend cliché, maar daarom niet minder waar: ik wil ook mijn mama, papa, zus, broer en vriendin bedanken. Ik ben niet altijd een even open boek wat school betreft, maar toch hebben ze allemaal een grote interesse getoond in mijn scriptie.

Geen van hen is zelf bekend met het onderwerp, maar dat maakte dat hun monden des te dieper openvielen toen ze het IoT-apparaatje uit mijn proof of concept in werking zagen - zeker toen de lichtjes van kleur veranderden. Al die geïnteresseerde blikken werkten alleen maar motiverend.

Ten slotte wil ik ook nog u, de lezer, bedanken om toch tenminste al het voorwoord achter de kiezen te hebben. Hopelijk heb ik u voldoende geprikkeld om verder te lezen en houdt u er achteraf iets aan over.

Samenvatting

Het Internet of Things is aan een opmars bezig. Het spectrum aan mogelijkheden van steeds meer alledaagse apparaten wordt almaar uitgebreider. Die evolutie kan echter niet alleen toegeschreven worden aan een grotere interesse van internationale bedrijven in het vakgebied, maar vooral aan een steeds groter wordende markt voor embedded systems, gericht aan particulieren. Microcontrollers en in het bijzonder Arduino-apparaten, zijn misschien wel de grootste katalysators in dit verhaal. Veel hobbyisten hebben ergens wel een Arduino liggen, maar ook kleinschalige softwarebedrijven durven al eens in de richting van de microcontroller te kijken. Een softwareoplossing in combinatie met een IoT-apparaat aangeboden krijgen, kan voor veel klanten immers als muziek in de oren klinken. Toch wagen veel ontwikkelaars de sprong naar IoT nog niet. De boosdoener? Taal. De dag van vandaag beheersen ontwikkelaars immers vooral moderne programmeertalen en dus in mindere mate C of C++. Maar laat net die twee dé talen bij uitstek zijn om Arduino's of andere microcontrollers mee aan te sturen. Er zit voor hobbyisten en softwarebedrijven dus niks anders op dan zichzelf of de werknemers om te scholen. Of toch niet?

Deze scriptie doet onderzoek naar de mogelijkheden van zo'n *moderne programmeertaal* bij de ontwikkeling van IoT-apparaten, met name Swift. In eerste instantie wordt het begrip *Internet of Things* in het lang en in het breed van context voorzien en wordt bekeken hoe een Arduino, alsook een Raspberry Pi - zo'n ander embedded system - precies werken en wat hun rol binnen de wereld van IoT is. Ook wordt Swift for Arduino voorgesteld. Dit softwaresysteem blijkt immers dé oplossing voor het vooropgestelde probleem te zijn. Het stelt ontwikkelaars in staat om aan C en C++ voorbij te gaan en Swift te gebruiken voor de aansturing van Arduino's. In een proof of concept wordt alle verworven informatie ten slotte ingezet voor de ontwikkeling van een nieuw IoT-apparaat. Concreet gaat het over een slimme prullenbak die weegt in hoeverre ze gevuld is en op basis daarvan een lichtsignaal de wereld instuurt. Om dit te bewerkstelligen, wordt gebruikgemaakt van een

Arduino. De prullenbak is daarnaast ook in staat haar bekomen meetwaarden op te slaan in een databank, zodat die later geraadpleegd kunnen worden door een extern apparaat, zoals een smartphone. Wegens haar standaard meegeleverde communicatiecapaciteiten, is een Raspberry Pi het apparaat dat hiervoor instaat.

Uit de proof of concept blijkt dat Swift wel degelijk in staat is een volledig IoT-apparaat aan te sturen. Toch geldt er één voorwaarde: de code moet van nul opgebouwd worden. Veel ingewikkelde elektronica-componenten rekenen immers op reeds bestaande code in de vorm van een library, die in de meeste gevallen uitsluitend opgesteld is met het oog op het Arduino-ecosysteem. Swift for Arduino valt daar niet onder en maakt het de gemiddelde softwareontwikkelaar dus enorm moeilijk toch dergelijke libraries aan te wenden. Moraal van het verhaal? Indien het plan inhoudt een IoT-apparaat te ontwikkelen en de daarbij horende Arduinocode vanaf nul op te bouwen, is Swift zeker een waardevolle optie voor hobbyisten en softwarebedrijven. Maar indien toch op externe Arduino libraries gesteund moet worden, haken softwarebedrijven beter af. Voor hobbyisten, daarentegen, lijkt die manier van werken wel nog steeds een interessante uitdaging. Het goede nieuws is echter dat Swift for Arduino volop in ontwikkeling is, dus de kans dat die laatste voorwaarde op termijn geschrapt kan worden, wordt mettertijd groter.

Abstract (English)

The Internet of Things is on the rise. The spectrum of possibilities offered by more and more everyday devices is constantly expanding. However, this development is not only due to the increased interest of international companies in the field, but mainly to an ever-expanding market for embedded systems aimed at private individuals. Microcontrollers, and in particular Arduino devices, are perhaps the biggest catalysts in this story. Many hobbyists probably have an Arduino lying around somewhere, but more and more small-scale software companies are also taking a look at the microcontroller. After all, offering a software solution in combination with an IoT device can be music to many customers' ears. Still, many developers do not yet take the plunge into IoT. The culprit? Language. These days, developers mainly master modern programming languages and therefore, to a lesser extent, C or C++. Yet these two languages are the languages of choice for controlling Arduinos or other microcontrollers. This implies that hobbyists and software companies have no other option than to retrain themselves or their employees. Or have they?

This thesis investigates the possibilities of such a *modern programming language* with regard to the development of IoT devices, in particular Swift. It begins by diving into the world of the Internet of Things and looks at how exactly an Arduino and a Raspberry Pi - another embedded system - work and fit into this world. Swift for Arduino is also introduced, since the software system appears to be the solution to the proposed problem. It enables developers to bypass C and C++ and use Swift to control Arduinos. Finally, as part of a proof of concept, all the information acquired is used to develop a new IoT device. In concrete terms, the project presents a smart waste bin that weighs how full it is and sends a light signal into the world on the basis of that weight. An Arduino is used to achieve this. The bin is also capable of storing its measurements in a database. This makes it possible for an external device, such as a smartphone, to consult these measurements later on. Because of its standard communication capabilities, a Raspberry Pi is the device

to resort to in order to achieve this purpose.

The proof of concept shows that Swift is indeed capable of controlling a complete IoT device. However, there is one condition: the code must be built from scratch. Many complex electronics components rely on existing code in the form of a library, which in most cases has been written exclusively for the Arduino ecosystem. Swift for Arduino is not part of this and therefore makes it extremely difficult for the average software developer to use such libraries. Moral of the story? If the plan is to develop an IoT device and build the associated Arduino code from scratch, Swift is certainly a valuable option for hobbyists and software companies. However, if it involves relying on external Arduino libraries, software companies are better off not using it. For hobbyists, on the other hand, this approach still seems to be an interesting challenge to take on. The good news, however, is that Swift for Arduino is in full development, which means that as time passes, this last condition might one day be scrapped.

Inhoudsopgave

1	Inleiding	19
1.1	Probleemstelling	20
1.2	Onderzoeks vraag	21
1.3	Onderzoeksdoelstelling	21
1.4	Opzet van deze bachelorproef	22
2	Stand van zaken	23
2.1	Internet of Things	23
2.1.1	Communicatie en samenwerking	24
2.1.2	Adresseerbaarheid	25
2.1.3	Identificatie	25
2.1.4	Sensing	25
2.1.5	Actuation	26

2.1.6	Informatieverwerking	26
2.1.7	Lokalisatie	27
2.1.8	User interface	27
2.1.9	Opmerkingen	28
2.1.10	Architectuur	28
2.2	De motors van IoT-apparaten	29
2.2.1	Arduino	30
2.2.2	Raspberry Pi	36
2.2.3	Een vergelijking	37
2.3	Swift for Arduino	39
2.4	µSwift	41
2.4.1	Wat µSwift niet ondersteunt	42
2.4.2	Integers	42
2.4.3	Characters	44
2.4.4	Strings	45
2.4.5	Arrays	46
3	Methodologie	49
4	Een slimme prullenbak als proefkonijn	51
4.1	Situatieschets	51
4.2	Weegschaal	52
4.2.1	Hardware	52
4.2.2	Software	52
4.2.3	Conclusie	61

4.3 Communicatie	61
4.3.1 Arduino	62
4.3.2 Raspberry Pi	65
5 Conclusie	71
5.1 De drie pijlers van het ontwikkelproces	71
5.1.1 Vanaf nul beginnen	72
5.1.2 Bouwen op een bestaande library	72
5.1.3 Verbinden met het internet	73
5.2 Onderzoeksvragen beantwoord	73
A Onderzoeksvoorstel	77
A.1 Introductie	77
A.2 State-of-the-art	78
A.2.1 Internet of Things	78
A.2.2 Arduino en Raspberry Pi	79
A.2.3 Swift for Arduino	80
A.3 Methodologie	81
A.4 Verwachte resultaten	82
A.5 Verwachte conclusies	83
B Code slimme prullenbak	85
B.1 Weegschaal	85
B.1.1 Startpunt	85
B.1.2 HX711_ADC library	91
B.1.3 Extra bestanden	109

B.2 Communicatie	114
B.2.1 Arduino	114
B.2.2 Raspberry Pi	116
Bibliografie	123

Lijst van figuren

2.1	Netwerktopologieën volgens Gratton (2013)	25
2.2	Architectuur van een systeem	27
2.3	Architectuur van een microprocessor	27
2.4	IoT-architectuur volgens Mattern en Floerkemeier (2010)	29
2.5	Alternatieve IoT-architectuur	30
2.6	Arduino Uno Rev3 (clone)	31
2.7	PWM-signalen	33
2.8	Arduino IDE	34
2.9	Raspberry Pi 4 Model B	39
2.10	Swift for Arduino IDE	41
2.11	Architectuur compiler Swift for Arduino	41
4.1	Opstelling weegschaal	53
A.1	Swift for Arduino IDE	81

Lijst van tabellen

2.1	Vergelijking Arduino Uno Rev3 en Raspberry Pi 4 Model B	38
B.1	Herkomst codebestanden weegschaal	86
B.2	Herkomst codebestanden communicatie	114

Lijst van codevoorbeelden

2.1	LED Flasher project in C (McRoberts, 2010)	36
2.2	Overflow operatoren in Swift for Arduino	44
2.3	Characters in Swift for Arduino	44
2.4	Characters in Swift for Arduino (vóór build)	45
2.5	String Buffers in Swift for Arduino (Peto & Kay, 2021)	46
2.6	Arrays met vaste waarden in Swift for Arduino	46
2.7	Arrays met dynamisch toegewezen waarden in Swift for Arduino .	47
4.1	Greep uit originele <i>shims.h</i>	55
4.2	Regels 7 t.e.m. 12 uit <i>shims.h</i>	56
4.3	Regels 21 t.e.m. 60 uit <i>HX711_ADC.cpp</i>	58
4.4	Regels 3 t.e.m. 22 uit <i>main.swift</i>	63
4.5	Regels 61 t.e.m. 65 uit <i>main.swift</i>	63
4.6	Regels 25 t.e.m. 37 uit <i>main.swift</i>	64
4.7	Regels 40 t.e.m. 59 uit <i>main.swift</i>	65
4.8	Regels 49 t.e.m. 58 uit <i>main.swift</i> (Roest, 2020b)	67
4.9	Regels 53 t.e.m. 62 uit <i>SerialHandler.swift</i> (Roest, 2020c)	68
4.10	Regels 64 t.e.m. 83 uit <i>SerialHandler.swift</i> (Roest, 2020c)	68
4.11	Regels 105 t.e.m. 119 uit <i>SerialHandler.swift</i> (Roest, 2020c)	69

4.12	Regels 23 t.e.m. 28 uit <i>main.swift</i> (Roest, 2020b)	69
4.13	Regels 60 t.e.m. 80 uit <i>SQLiteHandler.swift</i> (Roest, 2020d)	70
B.1	Weegschaal - <i>main.swift</i>	85
B.2	Weegschaal - <i>HX711_ADC.cpp</i>	91
B.3	Weegschaal - <i>HX711_ADC.h</i>	104
B.4	Weegschaal - <i>config.h</i>	108
B.5	Weegschaal - <i>main.h</i>	109
B.6	Weegschaal - <i>shims.h</i>	110
B.7	Communicatie - <i>main.swift</i>	114
B.8	Communicatie - <i>main.swift</i> (Roest, 2020b)	116
B.9	Communicatie - <i>SerialHandler.swift</i> (Roest, 2020c)	117
B.10	Communicatie - <i>SQLiteHandler.swift</i> (Roest, 2020d)	120

1. Inleiding

Toen op 18 februari 2021 NASA's nieuwste rover, Perseverance, succesvol op Mars landde, werd dit uitgebreid besproken op sociale media, tijdens nieuwsbulletins op radio en tv en op de voorpagina's van kranten. De hele wereld reageerde enthousiast. Niet alleen het technisch huzarenstukje werd met veel bewondering becommentarieerd, ook het doel van de rover en de omkaderende missie sprak tot de verbeelding. Leven zoeken op Mars. Of dat leven er ooit geweest zou kunnen zijn, is natuurlijk nog de vraag. Dat er ooit leven kon ontstaan op onze eigen planeet, is immers al een ongelooflijk gegeven. Dat dat prille leven uiteindelijk zou uitgroeien tot datgene wat deze aarde tegenwoordig rijk is, kan gerust een mirakel genoemd worden.

De meest indrukwekkende uitkomst van dat ontwikkelproces is misschien wel de mens. De mens is er doorheen haar bestaan immers in geslaagd zichzelf het leven gevoelig makkelijker te maken. De ontwikkeling van talen stelde de mens in staat ondubbelzinnig met elkaar te communiceren, de uitvinding van het wiel maakte transport plots kinderlijk eenvoudig en de ontdekking dat gewassen gekweekt konden worden, maakte van de mens een sedimentair wezen en bracht een exponentiële groei in bevolkingsaantallen met zich mee. Ook *moderne* uitvindingen of ontdekkingen zoals de boekdrukkunst, elektriciteit en de computer, brachten de mens alleen maar meer comfort.

Deze scriptie doet in zekere zin onderzoek naar technieken en technologieën die het leven van de mens verder vergemakkelijken. Grotendeels staat de ontwikkeling van *Internet of Things*- of IoT-apparaten centraal. Dergelijke *slimme* apparaten vervullen vaak één bepaalde taak en zorgen er zo voor dat gebruikers diezelfde taak niet zelf hoeven uit te voeren. Uiteraard zijn er al onnoemelijk veel teksten geschreven die onderzoek doen naar IoT en lezers de kneepjes van het vak leren. Daarom belicht deze scriptie IoT in de eerste plaats vooral op een - op het moment van schrijven - vrij vernieuwende

en misschien wel experimentele manier. In plaats van gevestigde, maar soms ietwat verouderde programmeertalen te hanteren, wordt in deze scriptie onderzocht hoe IoT-apparaten ontwikkeld kunnen worden door gebruik te maken van een moderne, maar vooral toegankelijker taal: *Swift*. Op die manier draait deze scriptie rond een cluster van technologieën die de afgelopen jaren heeft bewezen het leven van de mens gevoelig te vergemakkelijken - IoT - maar in hogere mate rond een nog nieuwere technologie die het technische gebruik van datzelfde onderwerp op haar beurt ook nog eens vergemakkelijkt.

1.1 Probleemstelling

De ontwikkeling van geautomatiseerde apparaten is iets van alle tijden. Toch is het slechts sinds enkele jaren dat die *wetenschap* toegankelijk is geworden voor het *grote publiek*. Een apparaat zoals Perseverance hoort uiteraard niet thuis in die categorie, maar nu microcontrollers en zelfs volledige computers aangekocht kunnen worden voor de spreekwoordelijke prijs van een appel en een ei, kan iedereen zijn of haar droomapparaat ineens knutselen. Die microcontrollers en *singleboardcomputers* zijn immers zo verpakt dat de pure elektronica gerelateerde zijde van het verhaal verrassend weinig inhoudt. Wat deze apparaten dan toch zo doeltreffend maakt, draait volledig rond programmeren. Zolang de verschillende componenten die aangesloten zijn de microcontroller of computer in kwestie het toelaten, kan op deze laatste eender welk gedrag geprogrammeerd worden. Dit geeft echter meteen ook aan welke *stiel* de ontwikkelaar wel baas moet zijn: de programmeertaal die het gekozen apparaat begrijpt.

Het is precies bij die programmeertalen dat het schoentje knelt. De bovenvermelde singleboardcomputers begrijpen vaak nog meerdere talen, maar vooral microcontrollers laten de ontwikkelaar geen keuze en dienen op de koop toe gewoonlijk aangestuurd te worden met een taal die niet meer van deze tijd is. Tegenwoordig kunnen veel mensen overweg met één of meerdere programmeertalen die ontworpen zijn om moderne websites of zelfs computer- of mobiele applicaties te bouwen, niet met talen die ontworpen zijn in de vorige eeuw. Daarnaast kan de mogelijkheid microcontrollers aan te sturen aan de hand van een moderne taal ook kleine bedrijven in staat stellen hun klanten de extra service aan te bieden om naast de ontwikkeling van een gepersonaliseerde applicatie ook een bijhorend IoT-apparaat te ontwikkelen. Zonder het aanwerven van gespecialiseerd personeel, is het immers onwaarschijnlijk dat een klein bedrijf de tijd en middelen zou willen of zelfs kunnen besteden om haar eigen werknemers een verouderde taal aan te leren. De voordelen die die extra service voor de klanten zou opleveren, lijken niet op te wegen tegen de nadelen.

Uiteraard bestaan er een heleboel talen die in aanmerking zouden komen om onderzoek naar te doen, maar voor deze scriptie is de keuze op het reeds eerder aangehaalde Swift gevallen. Swift is de taal die gebruikt wordt om onder meer macOS- en iOS applicaties te bouwen en wordt daarom ook beheerst door een grote groep mensen. Ook binnen softwarebedrijven zijn er meer dan vaak iOS-ontwikkelaars aanwezig, wat zo'n bedrijf meteen in staat stelt die werknemers in te zetten voor de eventuele ontwikkeling van een IoT-apparaat.

Concreet heeft deze scriptie twee doelgroepen. Aan deze ene kant zijn er de *hobbyisten*, mensen die in hun vrije tijd graag bezig zijn met het ineen knutselen van apparaatjes, maar zonder over een uitgebreide kennis van programmeertalen te beschikken. Zij die Swift of een gelijkaardige programmeertaal reeds machtig zijn, kunnen zonder veel problemen aan de slag met hun eigen IoT-apparaten. Zij die Swift nog niet machtig zijn, kunnen met een gerust hart de taal onder de knie proberen te krijgen, aangezien ze naast de bouw van IoT-apparaten ook aangewend kan worden om iOS-applicaties te bouwen. Aan de andere kant zijn er de eerder kleine softwarebedrijven die iOS-ontwikkelaars in huis hebben. Die bedrijven zouden dankzij de technologieën en technieken beschreven in deze scriptie in staat moeten zijn IoT-apparaten te ontwikkelen naast de software die ze al leveren, zonder daarvoor gespecialiseerd personeel in huis te moeten halen.

1.2 Onderzoeksvraag

Hoewel alle hierboven beschreven bevindingen ervan uitgaan dat het effectief mogelijk is IoT-apparaten te ontwikkelen aan de hand van Swift, staat dat uiteraard (nog) niet vast en moet deze scriptie daar verder klare wijn over schenken. Centraal staat de vraag in hoeverre bestaande technologieën het toelaten Swift te gebruiken voor de aansturing van IoT-apparaten. Om deze onderzoeksvraag meer context te geven, worden verder concrete antwoorden gezocht op de deelvragen tot wat Swift voor IoT in staat is in vergelijking met de gevestigde programmeertalen, welke technologie - Arduino of Raspberry Pi - de voorkeur geniet wanneer gewerkt wordt met Swift en wat de specifieke mogelijkheden zijn van Swift voor Arduino.

1.3 Onderzoeksdoelstelling

Dat het geen sinecure is om zwart op wit antwoorden te formuleren op bovenstaande vragen, behoeft geen betoog. Om toch enigszins aansluiting te vinden bij de absolute waarheid, wordt in het tweede deel van deze scriptie een proof of concept uitgewerkt vanuit het perspectief van een hobbyist of applicatieontwikkelaar met een bescheiden kennis van Swift. De concrete invulling van deze proof of concept berust uiteraard grotendeels op de fantasie van haar bedenker, maar tracht toch gebruik te maken van vaak gekozen componenten en systemen, zodanig dat uiteindelijk een zo algemeen mogelijk oordeel geveld kan worden.

Bedoeling is om een *slimme* prullenbak te ontwikkelen aan de hand van een Arduino en Raspberry Pi. Op de Arduino worden enkele elektronica-componenten aangesloten: een viertal gewichtsensoren en een lichtstrip. De gewichtsensor stelt de Arduino in staat na te gaan in hoeverre de prullenbak haar maximale capaciteit bereikt heeft en geeft dat ook visueel weer aan de hand van de lichtjes. Een rode kleur geeft aan dat de grenswaarde overschreden is, terwijl een oranje kleur de gebruiker waarschuwt dat die situatie gevaarlijk dichtbij komt. In alle andere gevallen licht de ledstrip niet op. Daarnaast wordt een Raspberry Pi aan de opstelling toegevoegd. Deze *minicomputer* moet de internetverbinding

van de prullenbak verzorgen en maakt er op die manier meteen ook een IoT-apparaat van. Zonder al te veel in technische details te treden, krijgt de Raspberry Pi het huidige gewicht van de inhoud van de prullenbak doorgestuurd en slaat het op in een databank. Een ander programma dat ook gehuisvest is op diezelfde Raspberry Pi, kan de databank vervolgens raadplegen en de waarden op een gestructureerde manier beschikbaar maken voor de buitenwereld. Ten slotte vervolledigt een eenvoudige iOS-applicatie de opstelling. Deze applicatie stelt haar gebruiker in staat in één oogopslag na te gaan of de prullenbak geleegd moet worden.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeks domein, op basis van een literatuurstudie.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeks vragen.

In Hoofdstuk 4 wordt een proof of concept opgesteld waarbij een volledig IoT-apparaat ontwikkeld wordt. Het apparaat in kwestie is een slimme prullenbak die zowel met enkele elektronische componenten als via het internet met de buitenwereld kan communiceren.

In Hoofdstuk 5, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeks vragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

Het onderwerp waar deze scriptie onderzoek naar doet, steunt uiteraard op een heleboel reeds bestaande ideeën en technologieën. Zo bestaat het achterliggende idee van het *Internet of Things* reeds sinds lang. Elektronica borden zoals de *Arduino* en *Raspberry Pi* zijn ondertussen ook al een hele tijd beschikbaar en worden al even lang gebruikt om IoT-apparaten te maken. Ook de programmeertaal die hier centraal staat - *Swift* - is inmiddels al enkele jaren vorhanden en tijdens haar relatief korte bestaan zijn al meerdere onafhankelijke platformen ontwikkeld die op de taal steunen. Een voorbeeld van zo'n platform is *Swift for Arduino*, dat ontwikkelaars in staat stelt Swift te gebruiken om Arduino-apparaten aan te sturen. Uiteraard zal dit platform dan ook een belangrijke rol spelen tijdens het verdere verloop van deze scriptie.

Met uitzondering van *Swift for Arduino* worden alle bovenvermelde zaken in het tweede deel van deze scriptie aangewend zonder verder in te gaan op hun praktische invulling of technologische werking. Daarom worden deze onderwerpen in dit hoofdstuk allereerst voorzien van een stand van zaken.

2.1 Internet of Things

Het *Internet of Things* - Internet der dingen, Internet of Objects of kortweg IoT - omvat het geheel van met elkaar verbonden alledaagse apparaten. Die apparaten zijn uitgerust met een zekere vorm van *intelligentie* die de standaardfunctionaliteit van het apparaat in kwestie uitbreidt. Ze zijn ook verbonden met het internet, wat ze in staat stelt te communiceren met mensen en andere apparaten. Samen vormen al deze apparaten het IoT en zorgen ze ervoor dat het internet nog sterker in het alledaagse leven van de mens wordt gebracht. Zo

kan het IoT een deurbel in staat stellen een melding te sturen naar een smartphone wanneer ze ingedrukt wordt, kunnen de binnenuitschakelaars bestuurd worden vanaf een smartwatch en is het mogelijk voor een koelkast om automatisch verse groenten te bestellen wanneer de huidige voorraad op is. (Xia e.a., 2012)

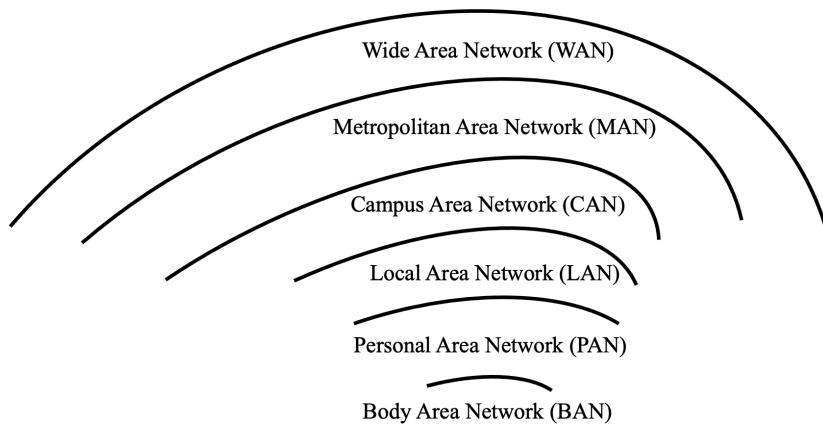
Het is bijzonder moeilijk een concrete technische invulling te geven aan IoT. Het idee heeft zich doorheen de tijd ontwikkeld en bestaat daarom uit een lappendeken aan technologieën die een brug vormen tussen de digitale en fysieke wereld. Het ene IoT-apparaat kan zo berusten op een bepaalde communicatietechnologie waarvan bij het andere geen sprake is. IoT manifesteert zich in meerdere vormen. Toch waagden Mattern en Floerkemeier (2010) zich eraan de nodige IoT-technieken en -technologieën af te bakenen en te categoriseren. Hieronder volgen hun bevindingen.

2.1.1 Communicatie en samenwerking

Verschillende apparaten die met elkaar samenwerken om een doel te bereiken, is misschien wel de basis van IoT. Maar om die samenwerking mogelijk te maken, moeten die apparaten uiteraard in staat zijn met elkaar te communiceren. Verschillende technologieën kunnen die communicatie tot stand te brengen. Vaak is er sprake van draadloze technologieën zoals GSM, Wi-Fi en Bluetooth. Vooral die laatste twee spelen een belangrijke rol in *Wireless Personal Area Networks* (WPANs). (Mattern & Floerkemeier, 2010)

De term WPAN verwijst naar een *Personal Area Network* (PAN) waar uitsluitend sprake is van draadloze technologieën. PAN verwijst op haar beurt naar het soort netwerk waar de apparaten die er deel van uitmaken zich dicht bij elkaar bevinden - niet verder dan 30 meter - en aan één persoon toebehoren. Een iMac waaraan een toetsenbord en muis verbonden zijn, in contact staat met een iPhone via AirDrop en kan communiceren met een printer in de dichte nabijheid, is een voorbeeld van een opstelling die onder de noemer PAN valt. Uiteraard bestaan er nog heel wat meer netwerktopologieën. De afstand tussen de apparaten die deel uitmaken van het netwerk in kwestie, is de belangrijkste factor om de bijhorende netwerktopologie aan te duiden. Zonder verder in te gaan op hun concrete invulling, geeft Figuur 2.1 enkele netwerktopologieën weer, geordend volgens omvang. Een belangrijke notie is dat de apparaten die tot een netwerk behoren, niet automatisch als IoT-apparaten beschouwd kunnen worden. De omgekeerde redenering geldt wel: elke opstelling van IoT-apparaten vormt een netwerk en kan bijgevolg bestempeld worden met een bepaalde netwerktopologie. (Gratton, 2013)

Hierboven werd kort gewag gemaakt van Bluetooth. Toch is de vermelding van die technologie niet onbelangrijk. Het geeft immers aan dat een apparaat dat uitsluitend via Bluetooth communiceert en dus niet direct in contact staat met het internet, toch onder de noemer Internet of Things geplaatst mag worden. Dit geeft aan dat de apparaten binnen een IoT-opstelling onderling gelijk welke communicatietechnologie mogen gebruiken. Ongeacht wat de term impliceert, hoeven IoT-apparaten dus niet per se over een internetverbinding te beschikken.



Figuur 2.1: Netwerktopologieën volgens Gratton (2013)

2.1.2 Adresseerbaarheid

Apparaten moeten andere apparaten binnen dezelfde IoT-opstelling kunnen opzoeken om mee te kunnen interageren. Om die reden moet elke apparaat een of andere vorm van adres met zich meedragen. Door gebruik te maken van zo'n adres of een deel ervan, kan een apparaat een ander apparaat of een groep andere apparaten aanspreken. (Mattern & Floerkemeier, 2010)

Daar apparaten met een directe internetverbinding per definitie aangeduid worden met een IP-adres, lenen internettechnologieën zoals Wi-Fi en Ethernet zich er uitstekend toe gebruikt te worden voor onderlinge communicatie binnen een IoT-omgeving.

2.1.3 Identificatie

Dat een IoT-apparaat identificeerbaar moet zijn, hangt uiteraard nauw samen met haar adresseerbaarheid. Toch dekt dat laatste de lading niet helemaal, aangezien adresseerbaarheid niet noodzakelijk slaat op de uniciteit van een IoT-apparaat. Elk IoT-apparaat moet uniek aangesproken kunnen worden zodat ondubbelzinnig uitgemaakt kan worden welke informatie van welk apparaat komt en welk apparaat welke informatie moet ontvangen. Het is ook zo dat een apparaat niet per se op een energiebron moet rekenen om van waarde te zijn binnen een IoT-opstelling. Om die reden worden IoT-apparaten vaak als *objecten* aangeduid. Apparaten zonder energiebron of passieve objecten kunnen uniek geïdentificeerd worden door gebruik te maken van *Near Field Communication* of NFC of een barcode. (Mattern & Floerkemeier, 2010)

2.1.4 Sensing

Vooraleer een IoT-apparaat betekenisvol kan zijn voor haar gebruiker, moet het op de een of andere manier informatie uit haar omgeving kunnen oppikken. Daarom beschikken IoT-apparaten over sensoren die bepaalde vormen van informatie kunnen capteren. Denk

daarbij aan lichtsensoren, temperatuursensoren of het registeren van een druk op een knop. (Mattern & Floerkemeier, 2010)

2.1.5 Actuation

Een IoT-apparaat moet niet alleen invloeden uit haar omgeving oppikken, op de een of andere manier moet ze zelf ook invloed uitoefenen op die omgeving. De componenten die aanwezig zijn op het apparaat, kunnen door het apparaat zelf aangestuurd worden en de omgeving op die manier in bepaalde mate beïnvloeden. Dit kan gaan van het aanzetten van een lamp tot het aansturen van een motor. (Mattern & Floerkemeier, 2010)

2.1.6 Informatieverwerking

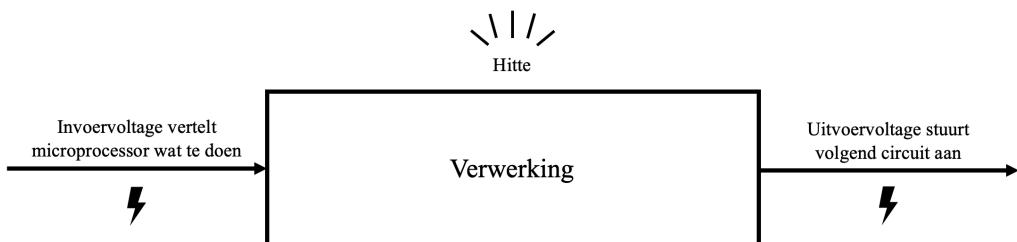
Het ligt voor de hand dat het uitlezen en aansturen van componenten en het versturen en ontvangen van informatie niet zomaar gebeurt. Elk IoT-apparaat heeft nood aan een centraal systeem dat op basis van invoergegevens de nodige berekeningen en beslissingen kan doorvoeren om ten slotte een uitvoer te voorzien. Er is ook nood aan een vorm van geheugen zodat op zijn minst de invoer-, uitvoer-, en tussengegevens, alsook de programma-instructies, al dan niet tijdelijk opgeslagen kunnen worden. (Mattern & Floerkemeier, 2010)

Het idee van een invoer, verwerking en uitvoer, vormt de basis van computers. Vaak wordt dit drietal aangeduid met de term *systeem*. De architectuur van een systeem kan erg eenvoudig gevisualiseerd worden, zo getuige Figuur 2.2. In de praktijk worden invoer en uitvoer bepaald door voltages, waarna sprake is van een *microprocessor*. Figuur 2.3 toont de architectuur van zo'n microprocessor. Een microprocessor op zich stelt weinig voor. Om functioneel te zijn, heeft ze nood aan extra elektronische circuits, zoals *input/output*-poorten (I/O devices), timers en uiteraard geheugen. Deze bestanddelen kunnen samengebracht worden door ze simpelweg aaneen te hangen. In dat geval is er sprake van een *computer*. De verschillende circuits van een computer kunnen gemakkelijk uitgebreid of vervangen worden, waardoor haar snelheid en geheugen gevoelig opgewaardeerd kunnen worden. Het is ook mogelijk de nodige bestanddelen - microprocessor inclusief - op één enkele chip samen te brengen. Nu is er sprake van een *microcontroller*. Microcontrollers hebben qua prijs en fysiek ruimtegebruik een streepje voor op computers, maar stellen hun gebruikers niet zomaar in staat aan de snelheid of geheugenruimte te tasten. *What you see is what you get.* (Crisp, 2004)

In de praktijk is de scheidingslijn tussen een computer en microcontroller nogal vaag. Beide systemen zijn doorheen de tijd flink geëvolueerd en doen dat nog steeds, waardoor hun definities praktisch voortdurend veranderen. Wat wel vaststaat, is dat ze vaak - zeker in het geval van IoT - ondergebracht worden in een of andere behuizing. Op die manier is er bijna sprake is van een *onzichtbare* computer - met computer hier in de algemene zin van het woord gebruikt. Een computer of microcontroller die op die manier ingebet is in het uiteindelijke apparaat, wordt een *embedded system* genoemd. (Marwedel, 2010)



Figuur 2.2: Architectuur van een systeem



Figuur 2.3: Architectuur van een microprocessor

2.1.7 Lokalisatie

Hoewel de hierboven en -onder opgesomde IoT-technieken en -technologieën in principe niet allemaal aanwezig hoeven te zijn in de opstelling van een IoT-apparaat, komen ze allemaal vrij vaak voor. Met het onderwerp dat in deze sectie besproken wordt, is dat minder het geval. Het is immers vrij eenvoudig voor te stellen dat bepaalde IoT-apparaten er geen nood aan hebben gelokaliseerd te kunnen worden. Zo is het nogal zinloos de precieze locatie van een slimme koelkast te kunnen achterhalen. Toch is lokalisatie bij bepaalde IoT-apparaten wel degelijk een vereiste. Er bestaan bijvoorbeeld kleine tags die in een handtas terecht kunnen, waardoor die handtas bij verlies gemakkelijk teruggevonden kan worden via een smartphone. Het spreekt voor zich dat die tags gelokaliseerd moeten kunnen worden. De meest voor de hand liggende technologie die geschikt is dit tot stand te brengen, is uiteraard GPS, maar ook het GSM-netwerk of *ultra-wideband* (UWB) komen hiervoor in aanmerking. (Mattern & Floerkemeier, 2010)

2.1.8 User interface

Ten slotte moet een gebruiker uiteraard nog met zijn of haar IoT-apparaat aan de slag kunnen gaan. De gebruiker heeft nood aan een soort raakvlak of *user interface* om met het apparaat te communiceren. Die redenering geldt ook in de omgekeerde richting: het apparaat moet met de gebruiker kunnen communiceren. Zo'n user interface kan verschillende vormen aannemen. Hoogstwaarschijnlijk is de meest voorkomende vorm die van een applicatie op een smartphone. Smartphones zijn immers ontzettend multifunctioneel en kunnen op ontelbaar veel manieren informatie van de gebruiker opvragen en op nog eens

zoveel manier informatie weergeven. Het kan echter veel abstracter. Zo kan bij apparaten die alleen aangestuurd kunnen worden via stemcommando's, de geluidsgolfdoorlatende ruimte waarin gebruiker en apparaat zich bevinden, aangeduid worden als de user interface. (Mattern & Floerkemeier, 2010)

2.1.9 Opmerkingen

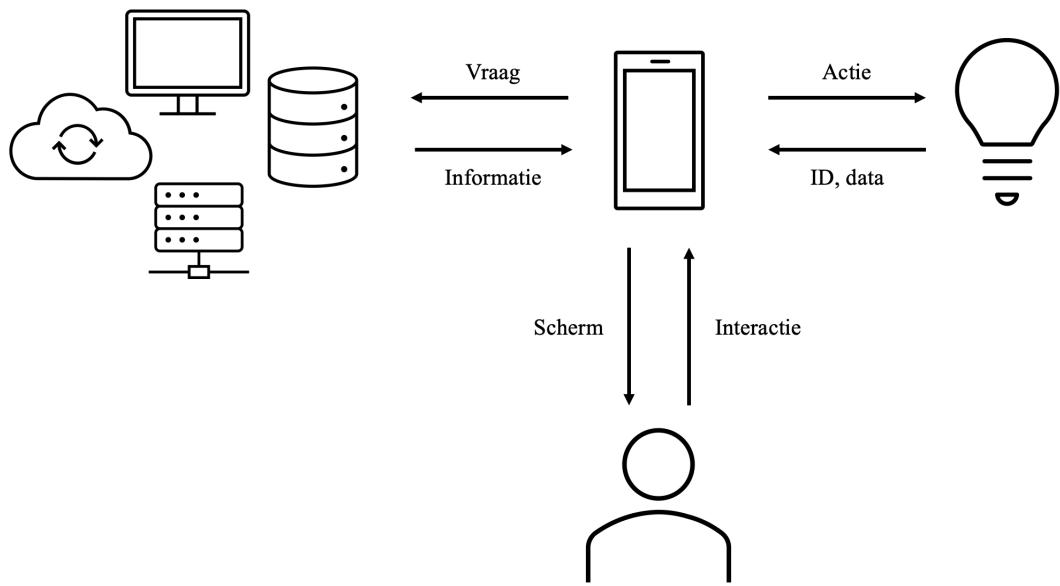
Het werd hierboven al eens aangehaald, maar dient toch nog eens in de verf gezet te worden: niet alle vermelde technieken en technologieën zijn nodig om van een volwaardig IoT-apparaat te spreken. De meeste concrete toepassingen gebruiken er slechts een subset van. Zelfs informatieverwerking is geen vereiste. Zo kunnen passieve objecten onderdeel zijn van een IoT-opstelling - bijvoorbeeld via barcodes - zonder zelf verder berekeningen uit te moeten voeren. Zij beschikken bijgevolg niet over een embedded system. (Mattern & Floerkemeier, 2010)

Wat ook aangehaald werd, is de bijzondere situatie waarbij een IoT-opstelling op geen enkele manier met het internet verbonden is. Hoe uitzonderlijk die situatie ook moge zijn, ze is per definitie mogelijk en bestaat ook. Daarbovenop is het geen vereiste dat een IoT-opstelling die wel beschikt over een internetverbinding ook toegang heeft tot het Internet - met hoofdletter - of *world wide web*. Een IoT-omgeving die bestuurd of geraadpleegd moet kunnen worden vanaf eender waar, moet uiteraard in contact staan met het Internet, maar het kan evengoed voorvallen dat een andere IoT-omgeving uitsluitend binnen het lokaal netwerk beschikbaar moet zijn om haar doel te bereiken. Een slimme deurbel met camera hoeft bijvoorbeeld niet per se in verbinding te staan met het Internet - hoewel dat uiteraard wel mogelijk is. Het volstaat in dat geval om de video- en geluidssignalen enkel binnenshuis beschikbaar te maken. (Mattern & Floerkemeier, 2010)

2.1.10 Architectuur

Een standaardarchitectuur voor IoT-opstellingen aanreiken, is moeilijk. De nodige bestanddelen kunnen immers in verschillende configuraties aaneen geschakeld worden en toch blijven voldoen aan de hierboven beschreven ideeën en eisen. Mattern en Floerkemeier (2010) stellen een opstelling voor zoals gevisualiseerd in Figuur 2.4. Dat hier gekozen wordt voor een smartphone als user interface, is van weinig belang. Wat wel van belang is, is de centrale rol die de user interface bekleedt. Een persoon staat - weinig verrassend - in direct contact met de user interface die op haar beurt - en dit is wel best opvallend - met zowel het IoT-apparaat als de aanwezige netwerkapparaten in contact staat. Alle communicatie verloopt met andere woorden via de user interface. Het moet nog eens onderstreept dat die netwerkapparaten samen een internet vormen, niet per se het Internet. In principe is het echter niet noodzakelijk dat die netwerktaak van de opstelling aanwezig is. Een opstelling die enkel uit een user interface en IoT-apparaat bestaat, kan per definitie nog steeds onder de noemer IoT geplaatst worden.

Een alternatieve en in de praktijk misschien meer voorkomende architectuur, is gevisualiseerd in Figuur 2.5. Hier bekleedt het internet - of een andere technologie die informatie



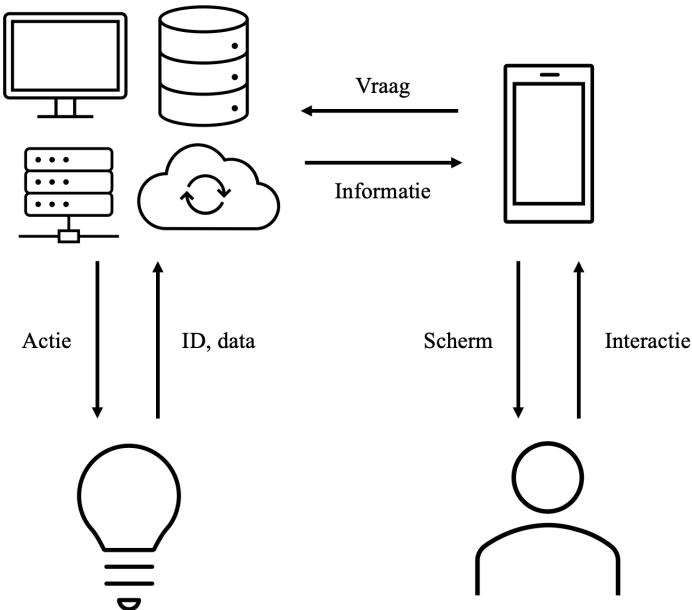
Figuur 2.4: IoT-architectuur volgens Mattern en Floerkemeier (2010)

kan doorsluizen zonder beroep te doen op IP-adressen - de centrale rol. De gebruiker staat nog steeds in contact met de user interface, maar die user interface steunt volkomen op het aanwezige netwerk om de communicatie met het IoT-apparaat of de IoT-apparaten te verzorgen. Wanneer dergelijke opstelling ook toegang heeft tot het Internet, komt meteen het grote voordeel van deze architectuur bovenrijzen. Dankzij het feit dat het Internet wereldwijd toegankelijk is, kunnen alle IoT-apparaten die deel uitmaken van de opstelling immers geraadpleegd en bestuurd worden van eender waar. Ten slotte is het belangrijk bij deze architectuur op te merken dat de netwerktak niet zomaar weggeleggen kan worden. Zonder kan de persoon immers niet communiceren met het apparaat - fysieke handelingen buiten beschouwing gelaten - waardoor van IoT geen sprake meer is.

2.2 De motors van IoT-apparaten

Met uitzondering van passieve objecten, rekent elk IoT-apparaat op een motor. Hoewel de term motor in deze context geen wetenschappelijke basis heeft en verder dus niet zal opduiken in wetenschappelijke teksten die spreken over gelijkaardige onderwerpen, is haar gebruik hier zeker geoorloofd. Zoals een motor bij voertuigen van levensbelang is, dient ook een IoT-apparaat op de juiste manier aangestuurd te worden. Om toch wat meer aansluiting te vinden bij die andere wetenschappelijke teksten, wordt gekozen vanaf nu gebruik te maken van de eerder vernoemde term *embedded system*.

Het spreekt voor zich dat de keuze van embedded system van groot belang is. Om zoveel mogelijk controle te hebben over het uiteindelijke IoT-apparaat, kan ervoor gekozen worden het embedded system volledig (of gedeeltelijk) zelf te ontwikkelen. Dit betekent dat er bijvoorbeeld vrij gekozen kan worden hoe groot het geheugen moet zijn, hoe



Figuur 2.5: Alternatieve IoT-architectuur

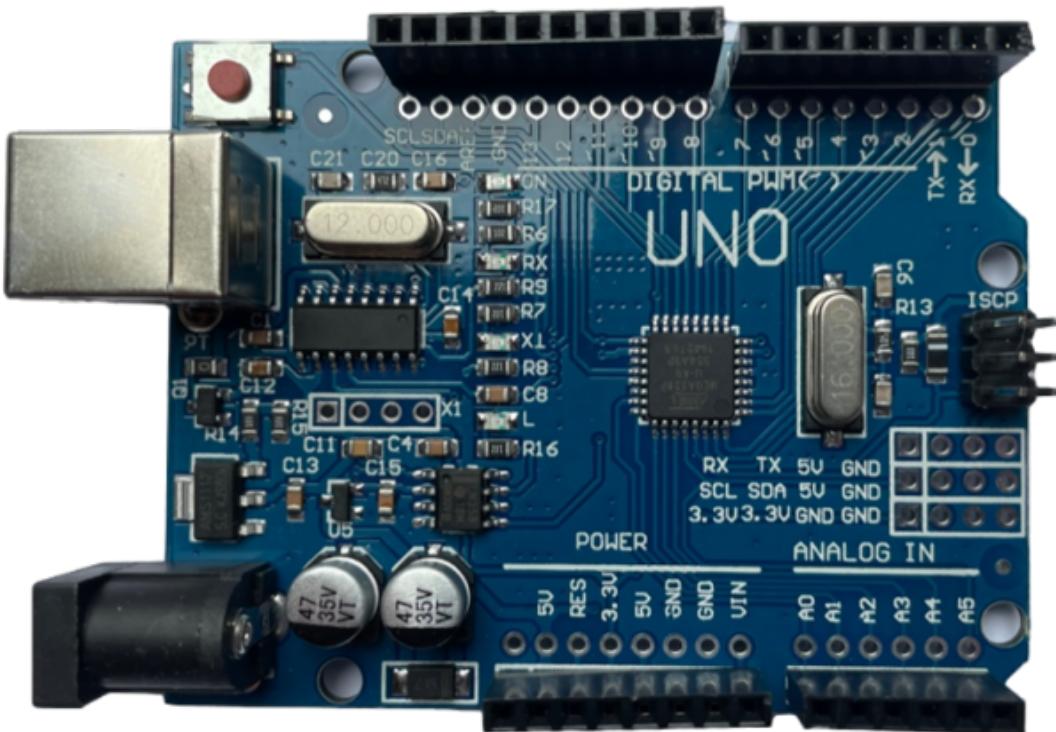
snel de (micro)processor dient te werken, hoeveel plaats alles samen mag innemen en hoeveel elke onderdeel mag kosten. Het is echter duidelijk dat een gevorderde kennis van elektronica vereist is om dit alles uit te denken en ineen te steken tot een bruikbaar product. Het doelpubliek van deze scriptie bestaat ook helemaal niet uit doorgewinterde elektronica-ingenieurs, dus moet er gezocht worden naar alternatieven.

Gelukkig bestaan die alternatieven. Tegenwoordig zijn heel wat *kant-en-klare* embedded systems beschikbaar op de markt. Er is keuze tussen allerlei merken, configuraties, prijskaartjes, vormen en maten. Toch zijn er twee namen die erbovenuit springen: Arduino en Raspberry Pi. Ondanks een gering verschil in grootte, lijken beide uitzonderlijk goed op elkaar. Toch zijn hun configuratie en dus ook hun werking erg verschillend. In de volgende secties worden beide apparaten apart voorgesteld om uiteindelijk kort naast elkaar gezet te worden.

2.2.1 Arduino

Arduino kent zijn oorsprong in 2005 in het *Interaction Design Institute Ivrea* in Italië. Daar zochten docenten naar een oplossing voor het probleem waarbij studenten elektro-nicacomponenten softwarematig wilden aansturen maar daarbij tegengehouden werden door hardwarekwesties. Er moest een soort prototype ontwikkeld worden waarop elk nieuw project kon verder bouwen. Er waren twee vereisten: het apparaat moest aangestuurd kunnen worden vanaf een Mac en moest goedkoop zijn. Om die redenen werd er vertrokken vanaf een programmeertaal en -omgeving die ontwikkeld werden aan het *Massachusetts Institute of Technology* (MIT), genaamd *Processing*¹. Een student slaagde er

¹<https://processing.org>



Figuur 2.6: Arduino Uno Rev3 (clone)

vervolgens in een microcontroller via USB met een computer te verbinden en bouwde een *Application programming interface* (API), zodat de communicatie tussen microcontroller en computer in het vervolg eenvoudig gestuurd kon worden. De API werd *Wiring*² gedoopt. Het volledige pakket - de software en de microcontroller met haar omringende circuits - werd volledig open source gepubliceerd en kreeg de naam *Arduino*³. (Severance, 2014)

Dankzij de beslissing om Arduino open source te maken, konden mensen thuis zelf aan de slag, zolang ze over een compatibele microcontroller beschikten. Op die manier won het platform snel aan populariteit. Ondertussen verkoopt Arduino zelf volledig geprepareerde bordjes via haar webshop⁴. Er is keuze tussen verschillende configuraties, maar het standaardmodel is de Arduino Uno. Consumenten kunnen er ook voor opteren aan de slag te gaan met een bord dat niet door Arduino zelf ontwikkeld is, een zogeheten *clone*, doorgaans aangeboden aan een lagere prijs. Figuur 2.6 toont een clone van een Arduino Uno Rev3. (McRoberts, 2010)

Hoewel de verschillende configuraties van Arduinobordjes allemaal zijn opgebouwd rond één basistechnologie, kunnen ze qua werking en gebruik uiteenlopen. Niettemin zal elke verdere verwijzing naar een Arduinobord vanaf dit punt in *facto* verwijzen naar een Arduino Uno, dit om de tekst overzichtelijk en leesbaar te houden.

²<http://wiring.org.co/>

³<https://www.arduino.cc/>

⁴<https://store.arduino.cc/>

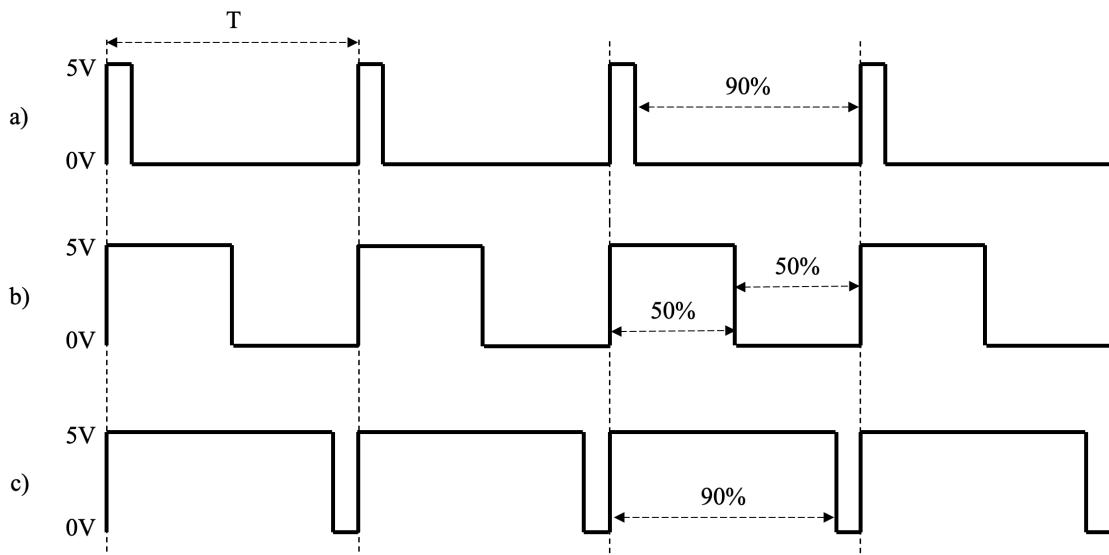
Zoals Figuur 2.6 duidelijk aangeeft, bevat een Arduino heel wat plaatsen waar andere componenten aan gekoppeld kunnen worden. Zo'n in- en uitgangen worden aangeduid met de term *pin*. Over het algemeen is elke pin tot hetzelfde in staat: lees een waarde af (*read*) of voer een waarde in (*write*). Toch zijn vanuit technisch oogpunt niet alle pinnen gelijk. (McRoberts, 2010)

Een digitaal signaal lezen of schrijven, kunnen nagenoeg alle pinnen. Het idee achter zo'n digitaal signaal is vrij eenvoudig en behelst niet meer dan een elektrisch signaal dat zich boven of onder een bepaald voltage bevindt. Is de signaalsterkte hoger dan die grens, is er digitaal sprake van HIGH. In het andere geval, is er sprake van LOW. Indien een pin ingesteld dient te worden op HIGH, zal een Arduino 5V uitsuren op de pin in kwestie en indien een andere pin een 5V-signaal binnenkrijgt, zal de Arduino dat beschouwen als HIGH. Wanneer 0V wordt uitgestuurd of gelezen, is er dan weer sprake van LOW. Deze digitale signalen lenen zich er bijvoorbeeld perfect toe om na te gaan of een schakelaar aan of uit is en om een licht in of uit te schakelen, maar het wordt al snel duidelijk dat dit systeem niet volstaat voor alle use cases. (McRoberts, 2010)

Om niet langer gebonden te zijn aan signalen die slechts twee waarden kunnen aannemen, zijn enkele pinnen ook in staat analoge signalen te lezen of te schrijven. Pinnen A0 tot en met A5 kunnen alle waarden⁵ tussen 0V en 5V lezen, wat het bijvoorbeeld mogelijk maakt na te gaan in hoeverre een potentiometer - een soort draaiknop - opengedraaid is. Alle pinnen die aangeduid zijn met een tilde (~), kunnen dan weer alle waarden⁵ tussen diezelfde grenzen uitsuren. Echter, een digitaal apparaat zoals een Arduino analoge signalen laten uitsuren, is in principe niet mogelijk. Om dit probleem te overkomen, maakt een Arduino gebruik van een slimme kunstgreep, genaamd *Pulse Width Modulation* (PWM). In het kort komt het erop neer dat een digitaal signaal erg snel uitgeschakeld wordt binnen een bepaalde periode. Hoe snel dat precies gebeurt, geeft de sterkte van het beoogde analoog signaal aan. De beste manier om PWM te illustreren, is door gebruik te maken van een blokgolf. Figuur 2.7a toont bijvoorbeeld een blokgolf waar het digitaal signaal 10% van elke periode (T) op HIGH staat en de rest ervan op LOW. Of nog: een LED-lichtje dat aangesloten is op een van de PWM-pinnen en een signaal binnenkrijgt zoals geïllustreerd in Figuur 2.7b, zal half zo sterk branden als tot wat ze maximaal in staat is. Ten slotte is het interessant vast te stellen dat een analoog signaal met waarde 0 het gevolg is van een PWM-signaal dat onafgebroken op LOW staat en een analoog signaal met maximale waarde voortkomt uit een PWM-signaal dat uitsluitend HIGH-waarden produceert. (McRoberts, 2010) (Sobota e.a., 2013) (Barr, 2001)

Het hangt natuurlijk van persoon tot persoon af in hoeverre de grenzen van een bepaald Arduinobordje opgezocht worden. Echte kenners zijn ongetwijfeld in staat hun bordjes de meest buitengewone functies te laten uitvoeren door enkel en alleen gebruik te maken van losse componenten en hun eigen circuits. Het is uiteraard weinig vanzelfsprekend dat iedereen zo'n ingewikkelde systemen ineen kan steken, laat staan bedenken, dus wordt er vaak gebruik gemaakt van *shields* om de capaciteiten van Arduino's uit te breiden met minder toegankelijke technologieën. Shields zijn aparte printplaatjes die eenvoudig bovenop een Arduino geplaatst kunnen worden door ze aan te sluiten op diens pinnen. De

⁵Mits een bepaalde stapgrootte of resolutie



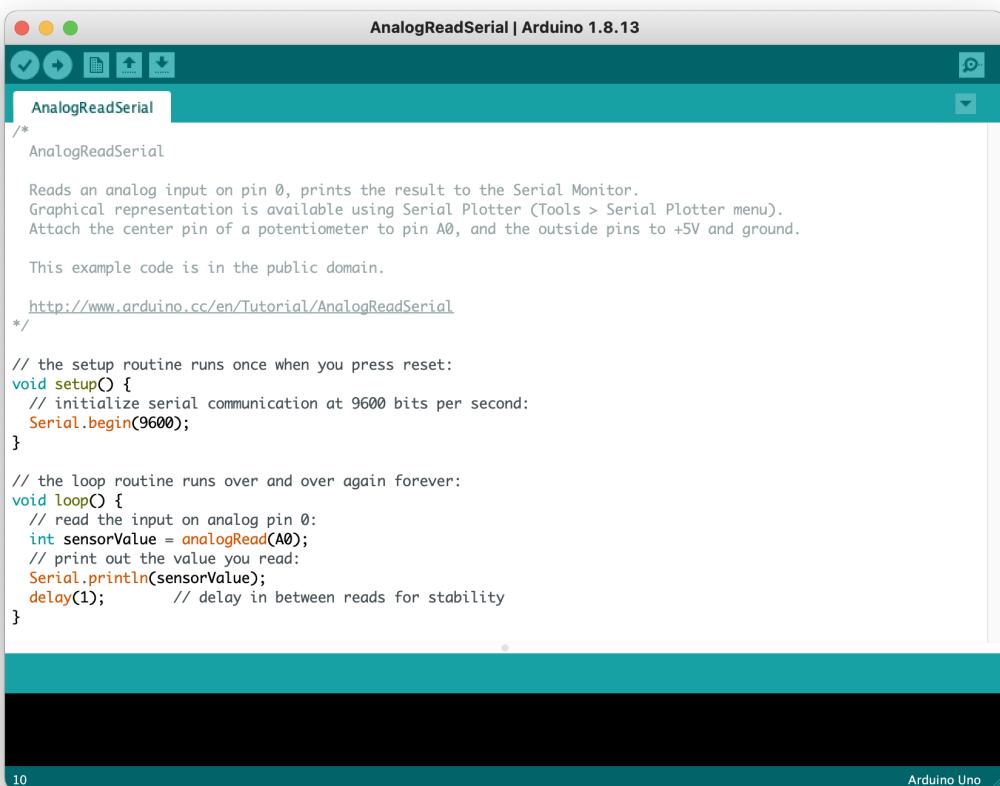
Figuur 2.7: PWM-signalen

shields beschikken zelf ook over pinnen, zodat andere componenten nog steeds gebruikt kunnen worden. Dit maakt het meteen ook mogelijk één Arduino van meerdere shields te voorzien. Het aantal shields dat mogelijk gebouwd zou kunnen worden, is ontelbaar, maar enkele voorbeelden van reeds bestaande zijn GPS-ontvangers, LCD-schermen en modules die een Ethernetverbinding tot stand kunnen brengen. (McRoberts, 2010)

Het gemak waarmee componenten op een Arduino aangesloten kunnen worden, is natuurlijk een enorm voordeel ten opzichte van het zelf construeren van componenten rondom een centrale microcontroller, maar daarmee is bij lange nog geen sprake van een werkend apparaat. Om dat te bereiken, moet eerst een programma geschreven worden. Het is precies daar waar misschien wel het grootste voordeel van Arduino ligt. Arduino biedt immers zelf een tool aan om code te schrijven, om te zetten naar begrijpbare instructies voor haar microcontroller en die instructies er ten slotte naar weg te schrijven, de *Arduino IDE*⁶. Die laatste twee stappen gebeuren trouwens volledig automatisch dankzij een druk op een knop. De Arduino IDE begrijpt uitsluitend Wiring. Wiring is - zoals eerder reeds aangehaald - een API die speciaal ontwikkeld werd voor Arduino en is een subset van de taal C. Een programma ontwikkeld voor Arduino en dus geschreven in Wiring (of C), wordt een *sketch* genoemd. De Arduino IDE biedt reeds een resem van die sketches aan voor vaak terugkerende opstellingen, gaande van een flikkerend LED-lichtje tot het tot stand brengen van een internetverbinding via Wi-Fi. Uiteraard kan elke sketch pas naar behoren werken als de nodige componenten of shields aanwezig zijn. Maar het kan ook omgekeerd: een fabrikant van componenten of zelfs volledige shields, kan haar eigen programmacode beschikbaar stellen, zodat klanten die maar hoeven te importeren in hun Arduino IDE om er vervolgens zelf mee aan de slag te gaan. Figuur 2.8 toont een schermafbeelding van de Arduino IDE. (McRoberts, 2010) (Patnaikuni, 2017)

De opbouw van een sketch is belachelijk eenvoudig. Om te werken, heeft een sketch op

⁶<https://www.arduino.cc/en/software>



The screenshot shows the Arduino IDE interface with the title bar "AnalogReadSerial | Arduino 1.8.13". The code editor contains the "AnalogReadSerial" sketch. The code reads an analog input from pin A0 and prints it to the Serial Monitor. It includes comments explaining the setup and loop routines. The bottom status bar shows "10" on the left and "Arduino Uno" on the right.

```
/*
 * AnalogReadSerial
 */
AnalogReadSerial

/*
 * Reads an analog input on pin 0, prints the result to the Serial Monitor.
 * Graphical representation is available using Serial Plotter (Tools > Serial Plotter menu).
 * Attach the center pin of a potentiometer to pin A0, and the outside pins to +5V and ground.
 *
 * This example code is in the public domain.
 * http://www.arduino.cc/en/Tutorial/AnalogReadSerial
 */

// the setup routine runs once when you press reset:
void setup() {
    // initialize serial communication at 9600 bits per second:
    Serial.begin(9600);
}

// the loop routine runs over and over again forever:
void loop() {
    // read the input on analog pin 0:
    int sensorValue = analogRead(A0);
    // print out the value you read:
    Serial.println(sensorValue);
    delay(1);          // delay in between reads for stability
}
```

Figuur 2.8: Arduino IDE

zijn minst nood aan een `setup()`- en `loop()`-functie. De `setup()`-functie wordt één keer aangeroepen, helemaal in het begin van het programma. Bedoeling is in deze functie alles klaar te zetten vooraleer het hoofdprogramma wordt opgestart. Typische instructies die in deze functie kunnen voorkomen, zijn het initialiseren van pinnen als invoer- of uitvoerpinnen en hoe snel de seriële communicatie dient te verlopen. Seriële communicatie gebeurt via de seriële poort van de Arduino en is handig om tijdens de uitvoering van een programma eenvoudig informatie te verzenden naar en te ontvangen van de Arduino. De `loop()`-functie, anderzijds, wordt opgeroepen nadat de `setup()`-functie volledig uitgevoerd is en blijft zich vervolgens constant herhalen. (McRoberts, 2010)

Codevoorbeeld 2.1 werd samengesteld door McRoberts (2010) en toont een sketch waarin een LED-lichtje afwisselend een seconde aan- en een seconde uitstaat. Op lijn 1 wordt een variabele geïnitialiseerd. Het is een goede gewoonte om de nummers van de pinnen die gebruikt zullen worden in de rest van het programma, vooraf telkens in een variabele te steken. Op die manier hoeft enkel de waarde van de variabele in kwestie aangepast te worden, als besloten wordt dezelfde functionaliteit door een andere pin uit te laten voeren. Op lijn 4 wordt de functie `pinMode(ledPin, OUTPUT)` vervolgens opgeroepen. Hier krijgt de Arduino te horen dat de pin met het nummer 10 signalen zal uitvoeren (`OUTPUT`). Logischerwijs kan er ook voor geopteerd worden een uitvoerpin aan te wijzen. Dat gebeurt aan de hand van `INPUT`. Ten slotte is het de beurt aan de `loop()`-functie. Op lijn 8 wordt de functie `digitalWrite(ledPin, HIGH)` opgeroepen. Zoals de naam suggereert, schrijft deze een digitale waarde naar de meegegeven pin. In dit geval is dat dus pin 10 en wordt deze op `HIGH` gezet. Of nog: er wordt een spanning van 5V op pin 10 geplaatst. Als een LED-lichtje aangesloten is op pin 10, zal het in dit geval dus branden. Vervolgens wordt `delay(1000)` opgeroepen. Deze functie deelt de Arduino eenvoudigweg mee 1000 milliseconden - of 1 seconde - te wachten vooraleer naar de volgende instructie te gaan. Die volgende instructie bevindt zich op lijn 10 en is reeds bekend. Deze keer wordt pin 10 echter op `LOW` gezet, het lichtje gaat dus uit. Op lijn 11 wordt nogmaals 1 seconde stilgestaan, waarna teruggekeerd wordt naar lijn 8 en het licht weer ingeschakeld wordt.

```

1 int ledPin = 10;
2
3 void setup() {
4     pinMode(ledPin, OUTPUT);
5 }
6
7 void loop() {
8     digitalWrite(ledPin, HIGH);
9     delay(1000);
10    digitalWrite(ledPin, LOW);
11    delay(1000);
12 }
```

Codevoorbeeld 2.1: LED Flasher project in C (McRoberts, 2010)

Uiteraard biedt Wiring nog een heleboel andere basisfuncties aan, maar aangezien deze scriptie geen startersgids voor Arduino is, wordt daar verder niet op ingegaan. De documentatie op de website van Arduino biedt een overzicht van alle beschikbare functies, constanten en structuren⁷.

Om gebruik te maken van alle softwaregerelateerde zaken die Arduino aanbiedt, is het opvallend genoeg geen vereiste een Arduinobordje aan te schaffen. Al het harde werk wordt immers opgekapt door haar microcontroller, een *ATMega328P*⁸. Zolang een ontwikkelaar beschikt over dergelijk exemplaar, kan hij of zij alle componenten die verder nodig zijn om de use case in kwestie te vervullen, naar eigen believen uitkiezen, aansluiten en positioneren. Met het oog op die manier een IoT-apparaat ineen te knutselen, levert dit heel wat mogelijkheden op wat betreft ruimtegebruik. (McRoberts, 2010) (Patnaikuni, 2017)

2.2.2 Raspberry Pi

Net zoals dat bij Arduino het geval is, valt er over Raspberry Pi heel wat te vertellen, misschien zelfs meer. Maar aangezien Raspberry Pi niet de focus is van deze scriptie, probeert deze sectie slechts een basisbegrip van het onderwerp over te brengen.

Om het kortweg te stellen, slaat Raspberry Pi op een gamma minicomputers. *Mini* omdat een Raspberry Pi minder krachtig, minder geheugen en - *tout court* - minder mogelijkheden heeft dan een doorsnee computer, zoals een laptop, vaste computer of zelfs smartphone, maar *computer* omdat ze op al die vlakken toch gevoelig sterker scoort dan een Arduino. Daarnaast is een Arduino - zoals reeds vermeld - niet veel meer dan een microcontroller en hangen ontwikkelaars dus vast aan een eerder beperkt arsenaal aan tools om dergelijk

⁷<https://www.arduino.cc/reference/en/>

⁸Datasheet ATMega328P: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

apparaat aan te sturen. Bij Raspberry Pi is het tegengestelde waar. (Patnaikuni, 2017) (Maksimovic e.a., 2014)

Wat de term *computer* precies inhoudt, daar bestaan veel meningen over. Zo zal een doorsnee computergebruiker een Raspberry Pi, enkel bij haar aanblik, waarschijnlijk niet meteen in verband brengen met zijn of haar laptop of vaste computer. Qua uitzicht heeft een Raspberry Pi immers weinig weg van een fraai vormgegeven computer, zoals die in een winkel als MediaMarkt verkocht worden. Maar laat diezelfde persoon aan de slag gaan met een volledig opgestelde Raspberry Pi, en die eerste indruk zal al snel vergeten zijn. Een Raspberry Pi beschikt immers over een besturingssysteem of *operating system* (OS), *Raspian*. Raspian is gebaseerd op Linux, een OS dat qua uitzicht en gebruik erg veel weg heeft van macOS en Windows. Ze beschikt met andere woorden over een grafische interface en biedt de mogelijkheid een muis, toetsenbord en scherm aan te sluiten om er gebruik van te maken. Figuur 2.9 toont een Raspberry Pi 4 Model B. Dit model wordt onder andere te koop aangeboden via haar officiële webshop⁹ en wordt daar - al dan niet onbescheiden - meteen ook aangeprezen als *desktop computer*. Het is meteen duidelijk dat de Raspberry Pi een grotere verscheidenheid aan ingangen heeft dan de Arduino die afgebeeld wordt in Figuur 2.6. Een van die ingangen is een Ethernetpoort, waardoor een Raspberry Pi van internet voorzien, snel gebeurd is. Daarnaast staan de vier USB-ingangen toe een ontelbaar aantal andere apparaten aan de opstelling toe te voegen en vertolkt een eenvoudige micro SD-kaart de rol van vast geheugen. Ten slotte beschikt een Raspberry Pi, net zoals een Arduino, over een reeks pinnen waarop meerdere losse elektronica-componenten aangesloten kunnen worden. Deze pinnen worden aangeduid met de naam *General Purpose Input/Output* (GPIO) en maken van een Raspberry Pi een geschikte kandidaat om als embedded system in een IoT-apparaat te dienen. (Maksimovic e.a., 2014)

2.2.3 Een vergelijking

Arduino en Raspberry Pi worden vaak in één adem genoemd, maar ondanks enkele gelijkenissen, verschillen ze op veel vlakken erg veel van elkaar. Zoals gebleken is, zijn beide systemen geschikt om als embedded system in een IoT-apparaat te dienen. Maar welke van de twee de voorkeur krijgt, hangt af van de use case.

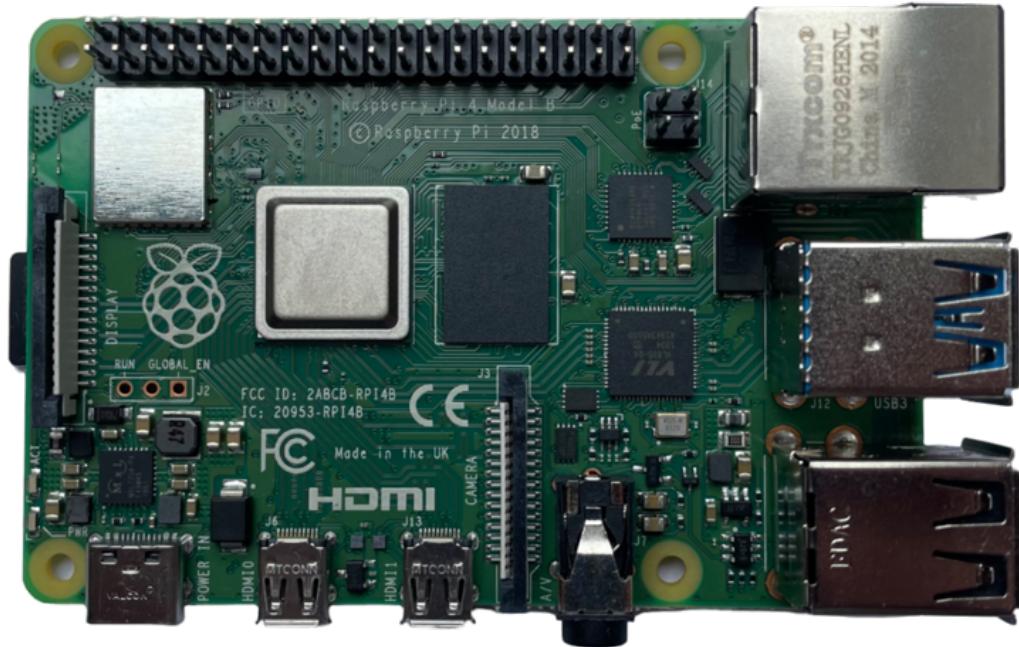
In Tabel 2.1 worden enkele beduidende kenmerken van enerzijds een Arduino Uno Rev3 en anderzijds een Raspberry Pi 4 Model B, naast elkaar geplaatst. Waar van belang, wordt per eigenschap de meest aantrekkelijke waarde in het vet aangeduid. De opsomming is gebaseerd op onderzoekswerk van Patnaikuni (2017) Maksimovic e.a. (2014).

Tabel 2.1 toont aan dat een Arduino zich er goed toe leent met elektronische componenten te communiceren zonder al te veel ruimte in beslag te nemen. Als er dan nog eens voor geopteerd wordt in productie enkel de microcontroller te gebruiken, kunnen grootte, gewicht en prijs nog gevoelig dalen. Een Raspberry Pi, aan de andere kant, lijkt dan weer erg geschikt voor het uitvoeren zware taken en onderhouden van communicatie met de buitenwereld. Ook het feit dat ze aan de hand van een besturingssysteem aangestuurd wordt,

⁹<https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>

	Arduino	Raspberry Pi
Grootte (mm)	75 x 53 x 15	86 x 54 x 17
Gewicht (g)	30	45
Prijs (€)	29,95	39,95
Microcontroller	ATmega328P	-
Processor	-	Quad core Cortex-A72
RAM-geheugen	2KB	2GB, 4GB of 8GB
Flashgeheugen	32KB	Afh. van micro SD-kaart
Digitale I/O-pinnen	14	26
Analoge invoerpinnen	6	0
PWM-pinnen	6	4
USB-poorten	1 (seriële communicatie)	4
Ethernet mogelijk?	Via shield	Ja
Wi-Fi mogelijk?	Via shield	Via extra component
Bluetooth mogelijk?	Via shield	Ja
Besturingssysteem	-	Raspbian (& enkele andere)
Programmeertalen	Wiring, C (& enkele andere)	Ontelbaar veel

Tabel 2.1: Vergelijking Arduino Uno Rev3 en Raspberry Pi 4 Model B



Figuur 2.9: Raspberry Pi 4 Model B

zorgt ervoor dat in wezen enkel haar eigen reken- en geheugencapaciteit een Raspberry Pi in de weg kunnen staan.

De hierboven beschreven bevindingen illustreren dat - nogmaals - voor de ontwikkeling van een IoT-apparaat de keuze tussen embedded systems gebaseerd moet worden op de use case. Sommige use cases zijn echter in die mate uitgebreid dat ze vooral gebaat zijn bij een constructie waarvan zowel een Arduino als een Raspberry Pi deel uitmaakt. Dergelijk opstelling is perfect te realiseren dankzij de voor seriële communicatie geschikte USB-poort van de Arduino en een van de vele USB-poorten van de Raspberry Pi. Op die manier kan de Arduino bijvoorbeeld instaan voor de communicatie met enkele componenten zoals drukknoppen en LED-lichtjes, terwijl de Raspberry Pi het *zware* rekenwerk verricht en de beoogde data via het internet opslaat in een databank en ter beschikking van de buitenwereld stelt. (Roest, 2020a) (Sobota e.a., 2013)

2.3 Swift for Arduino

Met de bovenstaande informatie kan perfect aan de slag gegaan worden, ware het niet dat deze scriptie verder wil gaan dan dat. Een Arduino en de Arduino IDE lijken onlosmakelijk verbonden met elkaar, maar schijn bedriegt. Het werd al aangehaald dat de echte kracht van een Arduino in haar microcontroller zit, de ATmega328P. Deze microcontroller behoort tot de familie van de AVR-chips en verwacht dus ook instructies die geschreven zijn in

machinecode bedoeld voor de AVR-architectuur. Dat maakt het in principe mogelijk gelijk welke AVR-microcontroller aan te sturen via de Arduino IDE. Maar zelfs dat is niet het volledige verhaal. De taken van de Arduino IDE kunnen perfect overgenomen worden door één of meerdere andere programma's. Code schrijven kan in gelijk welke tekstverwerker, een compiler moet die code omzetten naar AVR-machinecode en ten slotte moet een systeem die machinecode op de microcontroller schrijven. De *Swift for Arduino* IDE is een applicatie voor macOS die, net zoals de Arduino IDE, al deze zaken ondersteunt. Er is echter één groot verschil, Swift for Arduino verwacht geen C-code, maar wel Swift. (Peto, 2017)

Swift for Arduino werd en wordt nog steeds ontwikkeld door Carl Peto. Via de officiële website van het project¹⁰ kan de IDE aangekocht worden voor \$40, maar ze is ook verkrijgbaar via de Mac App Store¹¹. Die laatste biedt de IDE één maand gratis aan en vraagt daarna \$3 per maand.

De Swift for Arduino IDE wordt afgebeeld in Figuur 2.10 en is vrij eenvoudig te gebruiken. Het voornaamste paneel bevindt zich in het midden en bevat de code van het geopende project. Het meest linkse paneel is vrij uniek in de wereld van de IDE's, want ze herbergt een heleboel korte stukjes code die zo in de projectcode gesleept kunnen worden. Het meest rechtse paneel, vervolgens, toont de voortgang van het compileer- en uploadproces, terwijl het onderste paneel de uitvoer van de seriële communicatie van de aangesloten microcontroller weergeeft. Bovenaan, ten slotte, bevinden zich enkele knoppen die onder andere toelaten de juiste microcontroller te selecteren en het compileer- en uploadproces te starten. (Peto, 2020) (Roest, 2020a)

De tekstverwerkingscapaciteiten en het uploadsysteem van de Swift for Arduino IDE zijn best indrukwekkende onderdelen, maar zijn op zich geen nieuwe uitvindingen. Wat wel uit de grond gestampt moest worden, was het compileerproces. Om dit werkende te krijgen, maakte Carl Peto op een slimme manier gebruik van enkele bestaande processen. Waar nodig werkte hij die wat bij om ze ten slotte achtereen te schakelen. Het startpunt in het compileerproces is uiteraard de Swiftcode. Swift for Arduino neemt deze code en laadt ze in de door Apple ontworpen - en ondertussen open-source - Swiftcompiler¹². Die Swiftcompiler is op haar beurt gebouwd op de *LLVM*¹³-architectuur. LLVM is zelf ook open-source en haar compilersysteem kan compatibele code omzetten naar *LLVM Intermediate Representation (LLVM IR)*. LLVM IR is een soort low-level tussentaal die universeel is voor alle LLVM-compatibele compilers. Dit omzetten van high-level code - in dit geval Swiftcode - naar LLVM IR is de eerste stap in het compileerproces en wordt ook wel aangeduid met de term *frontend*. De tweede stap, vervolgens, bestaat erin de LLVM IR-code te optimaliseren. Dit optimaliseerproces wordt uitgevoerd door de *LLVM optimizer*¹⁴ en verwijdert bijvoorbeeld ongebruikte variabelen of functies. In de derde en meteen ook de laatste stap moet de geoptimaliseerde LLVM IR-code uiteindelijk nog omgezet worden naar machinecode die begrepen kan worden door het doelapparaat. In

¹⁰<https://www.swiftforarduino.com/>

¹¹<https://apps.apple.com/be/app/s4a-ide/id1334769565>

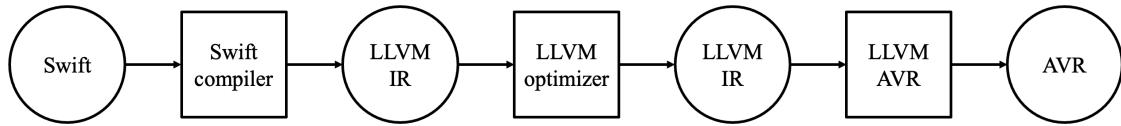
¹²<https://swift.org/swift-compiler/>

¹³<https://llvm.org>

¹⁴<https://llvm.org/docs/CommandGuide/opt.html>

The screenshot shows the Swift for Arduino IDE interface. The main window is titled "Untitled". The code editor displays Swift code for an AVR microcontroller. The code includes basic pin operations (Set Pin Mode Output, Set Pin Mode Input), digital I/O (Write Pin High, Write Pin Low, Digital Read), conditional statements (Digital Read in an if statement), arithmetic operations (Add 1 to a value, Subtract 1 from a value), setting constants (Set a Constant), and timing (Wait). The code also includes comments about Libraries and the Main Loop. The serial output window on the right shows the command to start a simulation: "started simduino with virtual UART on /dev/ttys000". Below the code editor, there are tabs for "Build", "Run", "Reboot", "Simulator", "Programmer", "Simulator", "Show Errors", and "Show Hide".

Figuur 2.10: Swift for Arduino IDE



Figuur 2.11: Architectuur compiler Swift for Arduino

dit geval is dat doelapparaat een AVR-microcontroller. Dankzij het feit dat LLVM open-source is, zijn enkele ontwikkelaars erin geslaagd een systeem te bouwen dat precies die laatste stap uitvoert: *AVR backend*¹⁵. De compiler die aanwezig is in de Swift for Arduino IDE is in wezen dus een proces van processen en kan zelfs aangeduid worden als de bestaansreden van deze scriptie. Zonder de mogelijkheid Swiftcode om te zetten naar AVR-machinecode, was deze scriptie er immers nooit geweest. Het volledige hierboven beschreven compileerproces wordt in Figuur 2.11 nog eens visueel weergegeven. (Peto, 2017)

2.4 µSwift

Apple ontwikkelde Swift uiteraard in functie van haar eigen besturingssystemen (iOS, macOS, watchOS en tvOS) en dus niet met het oog op het aansturen van microcontrollers. Dit impliceert meteen ook dat Swifts standaardcompiler verwacht dat het apparaat waarvoor

¹⁵https://llvm.org/doxygen/md__lib__Target_AVR_README.html

ze machinecode genereert, toch enigszins over het uitgebreide geheugen en de hoge verwerkingskracht van een Apple-apparaat - denk aan een iPhone of MacBook - beschikt. Het ligt voor de hand dat de gemiddelde microcontroller niet aan die eisen voldoet. Daarom zal Swift for Arduino niet elk Swiftproject compileren, maar enkel die projecten die geschreven zijn in μ Swift¹⁶ ¹⁷. (Peto, 2018)

μ Swift is Swift, maar steunt enkel op de meest fundamentele onderdelen van de zogeheten *Swift Standard Library*. Bouwstenen die rekenen op veel of dynamisch geheugengebruik, hebben in μ Swift een ietwat afgezwaktere tegenpool of werken zelfs helemaal niet. Het spreekt ook voor zich dat de verschillende uitgebreide libraries die Xcode ondersteunt - zoals de libraries die het bouwen van iOS-apps mogelijk maken - niet herkend worden door Swift for Arduino. Om verwarring te voorkomen, beschikt de Swift for Arduino IDE over een *Swift For Arduino Help*-venster, waarin beschreven wordt wat de IDE wel en vooral niet ondersteunt. De tekst werd opgesteld door Peto en Kay (2021) en de belangrijkste zaken eruit worden hieronder opgesomd. (Peto, 2018)

2.4.1 Wat μ Swift niet ondersteunt

De volgende zaken ondersteunt Swift for Arduino niet en zal de bijhorende compiler dus verplichten build errors te genereren.

- Klassen of overerving
- Existential containers
- Niet-gespecialiseerde generics
- Reflection
- `CustomStringConvertible`
- String-interpolatie

Het is belangrijk op te merken dat Swift for Arduino volop in ontwikkeling is en het daarom perfect mogelijk is dat bepaalde zaken uit de opsomming hierboven uiteindelijk toch ondersteund worden. Zo is versie 4.4.1 de meest recente versie op moment van publicatie van deze scriptie en ondersteunt deze reeds zaken zoals tuples, structs, enums, protocols en generics, daar waar dat in vroegere versies nog niet het geval was. Deze opmerking geldt uiteraard ook voor de volgende subsecties. (Peto & Kay, 2021)

2.4.2 Integers

Numerieke waarden spelen een belangrijke rol bij de aansturing van microcontrollers. Zo moeten de juiste pinnen aangesproken worden, moeten er bepaalde waarden naar de pinnen gestuurd worden en moeten de waarden die de pinnen zelf uitsturen ook gelezen worden. Integers - gehele getallen - zijn hiervoor de perfecte instrumenten. Echter, wanneer een integer in Xcode geïnstantieerd wordt, zal de compiler deze behandelen als een `Int64`-datatype - een geheel getal tussen -9 223 372 036 854 775 808 en 9 223 372 036 854 775

¹⁶<http://uswift.io/>

¹⁷<https://github.com/carlos4242/uSwift>

807 (grenzen inbegrepen). Het is meteen duidelijk dat een microcontroller met dergelijke grote waarden bestoken, om problemen vragen is. Bovendien lijkt het weinig waarschijnlijk dat eender welk microcontrollerproject nood heeft aan zo'n enorme getallen. Het aantal pinnen op een Arduino is bijvoorbeeld beperkt, dus kan er perfect gebruik gemaakt worden van een `UInt8` - een natuurlijk getal tussen 0 en 255 (grenzen inbegrepen) - om een pin aan te duiden. (Peto & Kay, 2021)

Swift for Arduino beschikt standaard al over twee *typealiases*. Typealiases worden gebruikt om nieuwe datatypes te definiëren aan de hand van bestaand datatypes. `public typealias Pin = UInt8` is een van de twee en zorgt ervoor dat elk getal gedefinieerd als `Pin`, eigenlijk een `UInt8` is. Functies als `pinMode(pin: Pin, mode: Bool)` en `digitalWrite(pin: Pin, value: Bool)` verwachten ook zo'n `Pin`-datatype en zullen een compilefout veroorzaken als ze een ander integertype krijgen als parameter. `public typealias IntegerLiteralType = UInt8` is de andere typealias en garandeert zelfs nog meer *type safety*. Hier wordt geen nieuw type gedefinieerd, maar wordt een reeds bestaand datatype - `IntegerLiteralType` - gespecificeerd als `UInt8`. Aangezien `IntegerLiteralType` het meest algemene datatype van een getal is, zorgt de typealias ervoor dat elk getal waarbij het datatype niet gedefinieerd is, automatisch wordt aangeduid als `UInt8`. Vroegere versies van Swift for Arduino waren niet standaard voorzien van deze typealias, waardoor de typealias bij veel projecten die te vinden zijn op het internet, nog expliciet wordt meegegeven als eerste lijtje van de code. (Peto & Kay, 2021)

Ondanks alle bovenbeschreven opvangnetten, blijft het aangeraden elke variabele of constante explicet te voorzien van een datatype. Het is immers goed mogelijk dat ook andere datatypes buiten `UInt8` nodig zijn. `Int8` of `Int16` ondersteunen bijvoorbeeld ook negatieve getallen, of nog: aangezien een `UInt8` niet volstaat om alle mogelijke waarden te beschouwen bij het lezen van waarden op een analoge pin, geeft de niet-onbelangrijke methode `analogReadSync(pin: Pin)` een `UInt16` terug. Het is duidelijk dat het goed aanduiden en bijhouden van datatypes van cruciaal belang is om compilefouten te voorkomen. (Peto & Kay, 2021)

Daar waar het bij 64-bitapparaten misschien minder zorgwekkend is, dient er bij 8- en 16-bitgetallen voldoende aandacht besteed worden aan *overflow*. Het is immers snel gebeurd dat een op het eerste zicht onschuldige optelling resulteert in een getal dat buiten de grenzen van het datatype in kwestie ligt. Om de daarbij horende compilefouten te voorkomen, biedt Swift de overflow operatoren `&+`, `&-` en `&*` aan, als tegenhangers van respectievelijk `+`, `-` en `*`. Daar waar Swiftontwikkelaars die overflow operatoren hoogstzelden gebruiken, zijn ze in het geval van microcontrollers van groot belang. Een microcontroller stopt immers volledig met werken bij een compilefout. Aangezien het minder erg lijkt dat een bepaald getal een onvoorziene waarde aanneemt dan dat haar microcontroller volledig stilvalt, koos Carl Peto ervoor de overflow operatoren standaard te maken in Swift for Arduino. Zo zal Codevoorbeeld 2.2 in Swift for Arduino de waarde `0` naar de console schrijven, terwijl het in Xcode een compilefout op de tweede lijn zou geven. (Peto & Kay, 2021)

```

1 var value: UInt8
2 value = UInt8.max + 1
3 print(value)

```

Codevoorbeeld 2.2: Overflow operatoren in Swift for Arduino

2.4.3 Characters

Aangezien μSwift slechts een subset is van Swift, mist ze enkele elementen uit de *Swift Standard Library*. Een van die elementen is het type `Character`. Characters zijn alleenstaande symbolen als `a`, `D`, `@` en `5` (niet als integer), en worden normaliter door de compiler omgezet naar hun overeenkomstige hexcode, zoals gedefinieerd in het Unicode-alfabet. Maar aangezien Swift for Arduino's Swift Standard Library niet over de juiste elementen beschikt, is haar compiler niet in staat die omzetting uit te voeren. Om toch met dergelijke alleenstaande symbolen aan de slag te kunnen, dienen developers de hexcodes van de nodige characters reeds in de code zelf aan te geven. Codevoorbeeld 2.3 toont hoe het gebruik van characters eruitziet in Swift for Arduino. (Peto & Kay, 2021)

```

1 if (charReadFromSerial == 0x58) { // 'X'
2     // do processing
3 }

```

Codevoorbeeld 2.3: Characters in Swift for Arduino

Gelukkig is het niet nodig om bij elk gebruik van een character op zoek te gaan naar haar overeenkomstige hexcode, want vooraleer Swift for Arduino haar compiler aan het werk zet, zal een zogeheten *AVR Sanitizer* de code afschuimen en eventuele characters in tekstvorm automatisch omzetten. Hierbij moeten echter twee belangrijke kanttekeningen gemaakt worden. Allereerst moeten de characters die in tekstvorm geschreven zijn, tussen **enkele** aanhalingstekens geplaatst worden. Characters tussen dubbele aanhalingstekens zullen niet als `Character`, maar als `String` beschouwd worden (zie Subsectie 2.4.4). In de tweede plaats zal de AVR Sanitizer de Swiftcode zelf aanpassen, dus een character in tekstvorm zal na een eerste build reeds vervangen zijn door haar overeenkomstige hexcode. Om toch enigszins overzicht te bewaren in de code, is het sterk aangeraden om naast een character haar tekstuele representatie in commentaar te zetten. Codevoorbeeld 2.4 toont hoe een character eruitziet vooraleer gebuild wordt, terwijl Codevoorbeeld 2.3 toont hoe diezelfde code eruit zal zien nadat de AVR Sanitizer haar werk heeft verricht. (Peto & Kay, 2021)

```
1 if (charReadFromSerial == 'X') { // 'X'  
2     // do processing  
3 }
```

Codevoorbeeld 2.4: Characters in Swift for Arduino (vóór build)

2.4.4 Strings

Zolang te tekst waarmee gewerkt wordt beperkt blijft in grootte, ondersteunt Swift for Arduino het gebruik van *strings* verrassend goed. Enerzijds zijn er de statische strings, die eenmalig vastgelegd worden bij de start van het programma en daarna niet meer aangepast kunnen worden. Deze strings kunnen eenvoudigweg gedeclareerd worden tussen **dubbele aanhalingstekens**, maar geen enkele. Zoals reeds vermeld in Subsectie 2.4.3 worden symbolen tussen enkele aanhalingstekens immers automatisch vervangen door een hexcode. In het geval er meerdere symbolen tussen de enkele aanhalingstekens staan, zal enkel het eerste symbool als character beschouwd worden en dienovereenkomstig enkel haar hexcode opgehaald worden. (Peto & Kay, 2021)

Aangezien het af en toe toch nodig is strings aan te passen tijdens de uitvoering van het programma, biedt Swift for Arduino *String Buffers* aan. String Buffers krijgen bij hun initialisatie een vaste grootte toegewezen, waarbinnen vervolgens dynamisch gewerkt kan worden. De functie `stringStartNew(bufferSize: UInt)` retourneert een nieuwe String Buffer met de grootte in bytes meegegeven als parameter. In Codevoorbeeld 2.5 wordt het gebruik van String Buffers geïllustreerd. In dit geval wordt een nieuwe String Buffer met een grootte van 35 bytes aangemaakt. De string wordt vervolgens herhaaldelijk uitgebreid om uiteindelijk naar de seriële poort gestuurd te worden in lijn 9. In lijn 10, ten slotte, wordt de String Buffer uit het geheugen van de microcontroller verwijderd. Dit is van uiterst groot belang, aangezien String Buffers sowieso al vrij veel geheugen innemen, maar vooral omdat er geen controlemechanisme is ingebouwd die een eventuele *memory overload* kan voorkomen. (Peto & Kay, 2021)

```

1 let tempC: UInt8 = 55
2
3 if var buffer = stringStartNew(bufferSize: 35) {
4     buffer += "The temperature is... "
5     buffer += tempC
6     buffer += "C, "
7     buffer += Float(tempC) * 9.0 / 5.0 + 32.0
8     buffer += "F"
9     print(buffer)
10    buffer.release()
11 }

```

Codevoorbeeld 2.5: String Buffers in Swift for Arduino (Peto & Kay, 2021)

2.4.5 Arrays

Weinig verrassend staat het gebruik van arrays in Swift for Arduino ook volledig in het teken van bedachtzaam geheugengebruik. Codevoorbeeld 2.6 toont hoe een statische array gedefinieerd en vervolgens herhaaldelijk aangeroepen wordt. De array wordt in lijn 1 gedefinieerd als constante, dus is het niet mogelijk verder in het project nog aanpassingen door te voeren aan de array. (Peto & Kay, 2021)

```

1 let testArray = [1,5,9,112,0,1,4]
2
3 for i in testArray {
4     print(i)
5 }
6
7 testArray.deallocate()

```

Codevoorbeeld 2.6: Arrays met vaste waarden in Swift for Arduino

Codevoorbeeld 2.7, daarentegen, toont een array die gedefinieerd is als variabele en dus wel aanpasbaar is. Verder wordt de array bij aanvang geïnitialiseerd met een vaste lengte - in dit geval 9 - en een waarde dat in eerste instantie aan elke plaats in de array toegekend wordt - in dit geval 0. Lijn 5 wordt vervolgens herhaaldelijk aangeroepen om elke plaats in de array van een nieuwe waarde te voorzien.

```
1 var length: Int = 9
2 var testArray = [UInt8](repeating: 0, count: &length)
3
4 for i: UInt8 in 0..
```

Codevoorbeeld 2.7: Arrays met dynamisch toegewezen waarden in Swift for Arduino

Net zoals dat met String Buffers het geval is, moeten arrays uit het geheugen gehaald worden na gebruik. In codevoorbeelden 2.6 en 2.7 wordt daar respectievelijk in lijnen 7 en 12 de functie `deallocate()` voor opgeroepen. Enkel bij arrays die globaal gedefinieerd en vervolgens doorheen de volledige levensduur van het programma gebruikt worden, is dat vrijmaken van het geheugen in principe niet nodig. (Peto & Kay, 2021)

3. Methodologie

In de hoop antwoorden te vinden op de verschillende deelvragen van deze scriptie, kan op verschillende manier te werk gegaan worden. Grondige vergelijkende studies lijken de meest geschikte methode om uit te zoeken tot wat Swift voor IoT in staat is in vergelijking met de gevestigde programmeertalen en welke technologie - Arduino of Raspberry Pi - de voorkeur geniet wanneer gewerkt wordt met Swift. Om daarnaast uit te maken wat de specifieke mogelijkheden van Swift voor Arduino zijn, zou dan weer vooral ingezet moeten worden op een lijvige opsomming.

Dat dergelijke onderzoeksmethoden interessante resultaten zouden opleveren, staat buiten kijf. Toch zou geen van hen een concreet antwoord kunnen bieden op de centrale vraag in hoeverre bestaande technologieën het toelaten Swift te gebruiken voor de aansturing van IoT-apparaten. Een beter plan van aanpak lijkt daarom een uitgebreide proof of concept op te zetten. Deze methode zou immers niet alleen een beter antwoord op de hoofdvraag formuleren, maar ook het doelgroep van deze scriptie - hobbyisten en softwarebedrijven met iOS-ontwikkelaars aan boord - van tastbare technieken én een handig naslagwerk voorzien. Een geschikt proof of concept kan echter niet eender welk project voorstellen. Belangrijk is dat de eerder vernoemde deelvragen binnen beschouwing blijven. In Hoofdstuk 4 wordt daarom een proof of concept uitgewerkt die met al die zaken rekening houdt.

Concreet wordt een *slimme* prullenbak ontwikkeld die haar gebruiker er op verschillende manieren van op de hoogte kan houden of ze al dan niet geleegd moet worden. De ontwikkeling van het IoT-apparaat is tweeledig: enerzijds wordt het gewicht van de inhoud van de prullenbak gewogen en anderzijds wordt de kleur van een ledstrip aangepast naargelang dat gewicht, alsook een databank bijgewerkt met de meest recente meetwaarde.

Het eerste deel van de proof of concept focust zich uitsluitend op de *weegschaal*. Daarvoor wordt gebruik gemaakt van een reeds bestaande library, die ontwikkeld werd met uitsluitend Arduino-georiënteerde technologieën in gedachten. Om een Arduino aan te sturen met Swift, is er weinig andere keuze dan op Swift for Arduino te rekenen. De uitdaging bestaat er dus in de *kernbestanden* uit die library in de eerste plaats werkende te krijgen binnen de Swift for Arduino IDE en ermee aan de slag te gaan. Dit is niet alleen nodig voor de volgende stappen, maar moet ook al een goed idee geven van de mate waarin Swift for Arduino letterlijke Arduino-georiënteerde bestanden zonder (al te veel) aanpassingen kan begrijpen. In de tweede plaats wordt een uitgewerkt voorbeeld die beroep doet op die *kernbestanden* en meegeleverd wordt met de originele library, vertaald naar Swift, in de hoop ook een antwoord te bekomen op de vraag hoe Swift for Arduino zich vergelijkt tegenover de pure softwarekant van Arduino.

Het tweede deel, ten slotte, focust zich op de communicatie van het IoT-apparaat naar de gebruiker en het internet. Het Swift for Arduino-project dat hiermee gepaard gaat, is uniek aan deze scriptie en moet dus een goed idee geven van tot wat Swift for Arduino op zich in staat is. Allereerst wordt de ledstrip aangesproken. Is de prullenbak te vol, moet die rood oplichten, terwijl een oranje kleur aangeeft dat het overschrijden van de grens dichterbij komt. Niet-oplichtende lichtjes betekenen dat de limiet van de prullenbak nog niet binnen bereik is. In het laatste deel van de proof of concept, ten slotte, wordt de internetverbinding op poten gezet. Om dat te bereiken, wordt een Raspberry Pi bij de opstelling betrokken. Deze moet de waarden die de Arduino meet, verwerken en opslaan in een databank. De Arduino en Raspberry Pi worden met andere woorden niet lijnrecht tegenover elkaar gezet, maar werken eerder samen om een gemeenschappelijk doel te bereiken. Uit Sectie 2.1.10 bleek immers al dat de twee wel in een bepaald aantal gelijkaardige functionaliteiten kunnen voorzien, maar vooral dat elk haar unieke kwaliteiten heeft: een Raspberry Pi is goed in het (tegelijk) draaien van zwaardere programma's en kan eenvoudiger met verschillende communicatietechnologieën overweg, terwijl een Arduino vooral goed scoort in haar eenvoud, namelijk op een overzichtelijke en voorspelbare manier verschillende elektronica componenten aansturen.

Dankzij de ondervindingen uit Hoofdstuk 4 moet de daarop aansluitende conclusie (Hoofdstuk 5) in staat zijn duidelijk uit te maken in welke omstandigheden Swift for Arduino een al dan niet waardig alternatief vormt voor Arduino's softwarepakketten. Een proof of concept is dus onontbeerlijk.

4. Een slimme prullenbak als proefkonijn

Om Swift for Arduino echt aan de tand te voelen, wordt een IoT-apparaat ontwikkeld aan de hand van het systeem. In dit hoofdstuk worden alle stappen van dit ontwikkelproces overlopen en voorzien van uitleg.

4.1 Situatieschets

Het plan voor het IoT-apparaat is als volgt: enkele gewichtsensoren of *load cells* worden aangesloten op een Arduino, die onophoudelijk het gewicht dat de sensoren opvangen, controleert. Elke nieuwe waarde die binnenkomt, wordt daarbij vergeleken met een vaste grens om op basis daarvan beslissingen te nemen die invloed hebben op de andere componenten die ook aangesloten zijn op de Arduino. Zo moet de kleur van een digitale ledstrip veranderen op basis van die vergelijking: een rode kleur betekent dat de grenswaarde overschreden is, terwijl een oranje kleur aangeeft dat het overschrijden van die grenswaarde gevaarlijk dichtbij komt. In alle andere gevallen is er geen nood aan een oplichtende ledstrip.

Ook aangesloten op de Arduino, is een Raspberry Pi. Elke nieuwe waarde van de gewichtsensors die binnenkomt, moet de Arduino niet alleen vergelijken met de vaste grenswaarde, maar ook naar haar seriële poort sturen. De Raspberry Pi, die via een USB-verbinding aangesloten is op die seriële poort, vangt die waarden vervolgens op om ze ten slotte naar een databank te schrijven.

Verder is het de bedoeling om de data in die databank op te laten vragen door een server - dit kan diezelfde Raspberry Pi zijn, maar evengoed een ander apparaat - en aan te bieden als API. Die API stelt andere apparaten, zoals een smartphone, in staat de originele waarden van de gewichtsensoren te raadplegen en tot bij de eindgebruiker brengen. Deze laatste

stappen worden echter niet verder uitgewerkt in deze scriptie, aangezien ze buiten haar bestek vallen.

De context waarin deze opstelling gebruikt wordt, hangt uiteraard af van haar doel. In deze scriptie wordt er verder echter vanuit gegaan dat het hierboven beschreven IoT-apparaat een soort slimme prullenbak is, die aan haar gebruikers aangeeft of ze al dan niet geleegd moet worden. En inderdaad, ook de term *IoT-apparaat* is hier op haar plaats, want deze opstelling voldoet wel degelijk aan de kenmerken beschreven in Sectie 2.1.

4.2 Weegschaal

4.2.1 Hardware

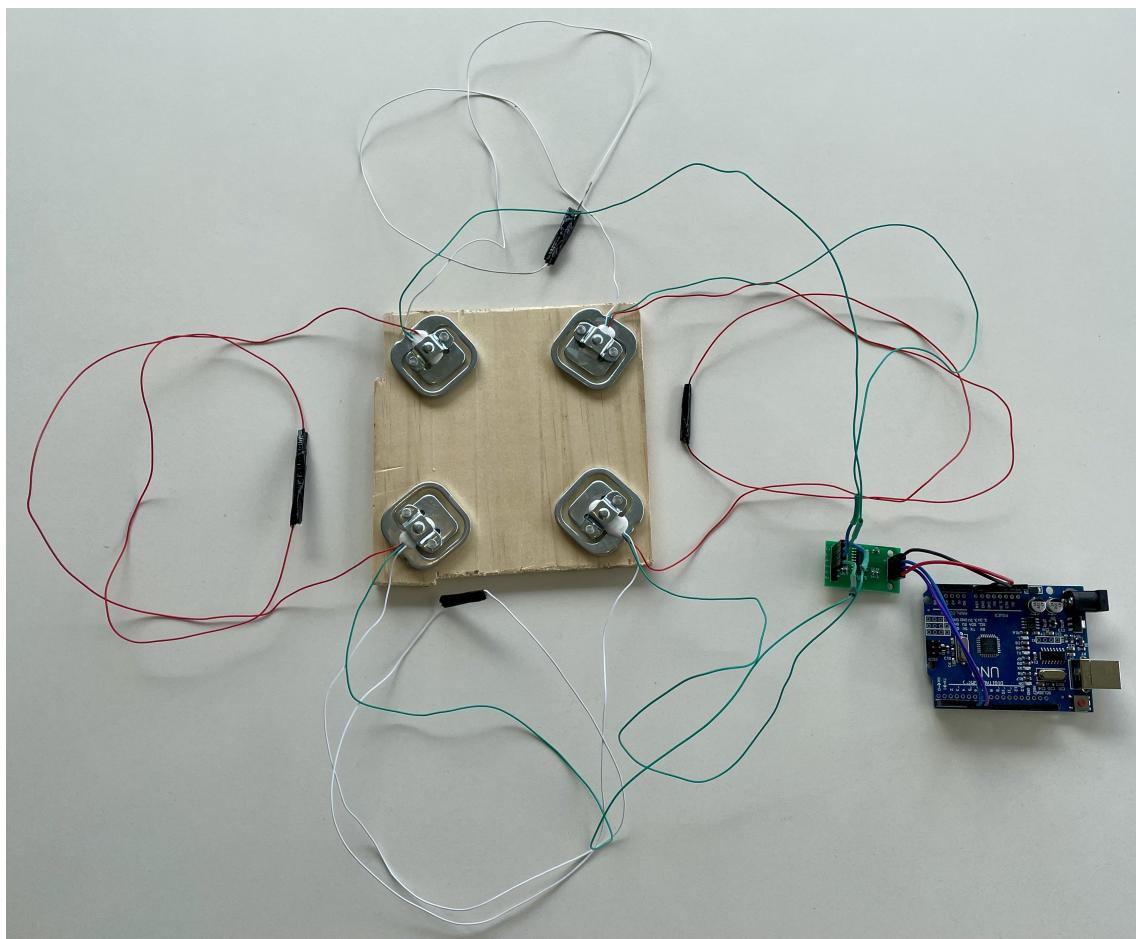
Allereerst is er de constructie van de gewichtsensoren. Samen vormen zij de weegschaal van de prullenbak. De opstelling waarmee hieronder verder gewerkt wordt, is gebaseerd op de uitleg uit een online gids en video van Indrek Luuk¹ ². De gids stelt drie mogelijke opstellingen voor: een opstelling met vier gewichtsensoren, een met twee en een die bestaat uit één sensor. In deze scriptie wordt ervoor gekozen met vier sensoren aan de slag te gaan, aangezien die de meeste stabiliteit in de uiteindelijke volledige constructie garanderen. Deze vier sensoren op zich, volstaan echter niet. Afhankelijk van de druk die erop uitgeoefend wordt, stuurt dergelijke sensor immers een variabel voltage uit, maar de verandering in voltage is zo miniem dat een microcontroller deze niet kan oppikken. Om die reden wordt de uitvoer van de vier sensoren eerst opgepikt door een ander geïntegreerd circuit, de *HX711 amplifier module*. Het circuit op dit bordje vergroot de binnenkomende voltageveranderingen zodanig, dat de microcontroller waarop ze aangesloten is, toch aan de slag kan met de gewichtsensoren. Na wat soldeerwerk, zit de uiteindelijke opstelling eruit zoals aangegeven in Figuur 4.1. De figuur illustreert dat de HX711 amplifier module aangesloten is op een van de 5V- en dus ook op een van de GND-pinnen van de Arduino. Het bordje heeft ook nood aan het kloksignaal van de Arduino, dat krijgt het door via pin 4 en de uiteindelijke dataoverdracht gebeurt via pin 5. Het is uiteraard geen verplichting dat pinnen 4 en 5 gebruikt worden, ook andere pinnen kunnen aangesproken worden. (Luuk, 2020)

4.2.2 Software

Wat hardware betreft, is het verhaal klaar. Tijd dus voor de software. Indien een ontwikkelaar ervoor kiest hiervoor met de Arduino IDE aan de slag te gaan, heeft hij of zij weinig voorbereidingswerk voor de boeg. Er zijn immers meerdere Arduino-libraries voorhanden die ontwikkelaars in staat stellen eenvoudig met de HX711 amplifier module aan de slag te gaan. Dergelijke library bestaat voornamelijk uit header files en bestanden in C of C++, waarnaar verwezen wordt als bronbestanden en die nieuwe functies ter beschikking stellen.

¹<https://circuitjournal.com/50kg-load-cells-with-HX711>

²<https://www.youtube.com/watch?v=LIuf2egMioA>



Figuur 4.1: Opstelling weegschaal

Daarnaast beschikt zo'n library zelfs vaak over enkele voorbeeldsketches, waar ontwikkelaars zich op kunnen baseren voor hun eigen project. In deze scriptie wordt er aan de slag gegaan met de *HX711_ADC* library³ van Olav Kallhovd. Naast de vaste bronbestanden, biedt die effectief enkele voorbeeldsketches aan. Een daarvan is een programma om de weegschaal te kalibreren.

Uiteraard is het doel van deze scriptie om een Arduino aan te sturen door gebruik te maken van Swift, dus van de Arduino IDE zal helaas geen gebruik gemaakt kunnen worden. Het goede nieuws is echter dat de Swift for Arduino IDE sinds versie 4.3 de mogelijkheid biedt bestanden in C of C++ - ook header files horen daarbij - te importeren. Toch betekent dat niet dat alle bronbestanden van een library zomaar geïnterpreteerd zullen kunnen worden door de Swift for Arduino compiler. Veel van die bronbestanden rekenen immers op dependencies die standaard niet meegeleverd worden met de Swift for Arduino IDE. Dat is ook meteen het geval met de HX711_ADC library. De HX711_ADC library beschikt immers over drie bronbestanden - *HX711_ADC.cpp*, *HX711_ADC.h* en *config.h* - waarvan de eerste twee beginnen met het statement `#include <Arduino.h>`. Een optie zou kunnen zijn om het bestand *Arduino.h* - die is gewoon te vinden online - eenvoudigweg zelf te importeren, maar daarmee is de kous niet af. Die ene header file steunt op haar beurt immers op andere dependencies, die op haar beurt weer op andere dependencies steunen, en ga zo maar door. In principe is het mogelijk de volledige lijst van dependencies te importeren, totdat de meest kernachtige bronbestanden vorhanden zijn, maar het is duidelijk dat dat huzarenstukje niet de juiste weg is.

Aangezien de meeste bronbestanden slechts steunen op een beperkt aantal functies uit de uitgebreide lijst die een header file als *Arduino.h* aanbiedt, is het een beter idee om die functies meteen *om te leiden* naar hun equivalenten low-level C-functies. In de GitHub repository van de Swift for Arduino Community⁴ bevinden zich reeds enkele projecten waarin gelijkaardige zaken bewerkstelligd worden. Het bestand dat daarbij telkens terugkeert, heet *shims.h*⁵ en werd opgesteld door Carl Peto. Code voorbeeld 4.1 toont een greep uit die header file en toont aan hoe functies als `digitalRead`, `digitalWrite` en `pinMode` - die normaal gezien terug te vinden zijn in *Arduino.h* - worden omgeleid naar de respectievelijk C-functies `_digitalRead`, `_digitalWrite` en `_pinMode`. Dankzij de declaratie `import AVR` bovenaan het *main.swift*-bestand - het startpunt van elk project in Swift for Arduino - heeft de Swift for Arduino compiler wel toegang tot die C-functies. Naast functies worden in *shims.h* ook bepaalde datatypes die uniek zijn aan het Arduino-ecosysteem, omgeleid naar voor Swift for Arduino begrijpbare C-datatypes.

³https://github.com/olkal/HX711_ADC

⁴<https://github.com/swiftforarduino/community>

⁵<https://github.com/swiftforarduino/community/blob/master/contributed%20unsafe%20modules/MAX30105/shims.h>

```
1 #ifndef digitalRead
2 #define digitalRead _digitalRead
3 #endif
4
5 #ifndef digitalWrite
6 #define digitalWrite _digitalWrite
7 #endif
8
9 #ifndef pinMode
10 #define pinMode _pinMode
11 #endif
```

Codevoorbeeld 4.1: Greep uit originele *shims.h*

Het *shims.h*-bestand naast de reeds aanwezige bronbestanden van de HX711_ADC library importeren en de statements `#include <Arduino.h>` vervangen door `#include "shims.h"`, volstaat helaas nog steeds niet. Swift for Arduino onderging immers al meerdere grootschalige updates sinds *shims.h* voor het laatst werd bijgewerkt, waardoor bepaalde zaken niet meer werken. Daarnaast werd de versie die te vinden is op GitHub, opgesteld met het oog op een ander project dan datgene waarrond deze proof of concept draait. Om de library bronbestanden toch werkende te krijgen, moeten dus nog enkele veranderingen en toevoegingen doorgevoerd worden aan *shims.h*. Die aanpassingen werden opgesteld in overleg met Carl Peto en zijn te raadplegen in het vernieuwde bestand, dat na te kijken is in Codevoorbeeld B.6. De belangrijkste aanpassingen bevinden zich tussen regels 7 en 21 en worden weergegeven in Codevoorbeeld 4.2.

```

1 #define interrupts sei
2 #define noInterrupts cli
3 #define byte uint8_t
4 #define yield()
5 #define micros() 0
6 #define boolean bool
7
8 #define HIGH (bool)1
9 #define LOW (bool)0
10 #define OUTPUT (bool)1
11 #define INPUT (bool)0
12 #define WHILE_LOW (unsigned char)0
13 #define CHANGING_EDGE (unsigned char)1
14 #define FALLING_EDGE (unsigned char)2
15 #define RISING_EDGE (unsigned char)3

```

Codevoorbeeld 4.2: Regels 7 t.e.m. 12 uit *shims.h*

Dankzij het vernieuwde *shims.h*-bestand is het eindelijk mogelijk om de drie bronbestanden van de HX711_ADC library - mits die te hebben voorzien van de juiste `include`-statements - foutloos te laten compileren door Swift for Arduino. De bronbestanden heten *HX711_ADC.cpp*, *HX711_ADC.h* en *config.h*, en zijn respectievelijk te raadplegen in Codevoorbeelden B.2, B.3 en B.4. Wat betreft die eerste, is het interessant enkele zaken kort onder de loep te nemen. Zo roept regel 347 van Codevoorbeeld B.2 de functie `noInterrupts()` op. Volgens de Arduino-documentatie⁶ zorgt deze functie ervoor dat het programma geen interrupts meer toestaat, dit kan handig zijn wanneer kritieke taken uitgevoerd moeten worden. Regel 360 roept dan weer `interrupts()`⁷ op, zodat interrupts weer toegelaten worden. Zoals gezegd, kent Swift for Arduino deze functies helemaal niet, dus moet er op zoek gegaan worden naar een alternatief. Dat alternatief wordt gelukkig aangeboden door de AVR library, waartoe *main.swift* toegang heeft, mits het juiste `import`-statement. De functies `cli()`⁸ en `sei()`⁹ hebben precies dezelfde functionaliteit als hun respectievelijke Arduino-equivalenten, waardoor zij de perfecte alternatieven zijn. Dat geeft meteen ook de bestaansreden van regels 2 en 3 van Codevoorbeeld 4.2 aan.

Terug naar Codevoorbeeld B.2, want daar worden nog twee methodes aangeroepen die een woordje uitleg verdienen. In regel 103 wordt `yield()` aangeroepen, ook deze functie komt uit het Arduino-arsenaal, maar heeft een minder duidelijke documentatie¹⁰. Aangezien de functie geen al te grote impact lijkt te hebben op de werking van het

⁶<https://www.arduino.cc/reference/en/language/functions/interrupts/nointerrupts/>

⁷<https://www.arduino.cc/reference/en/language/functions/interrupts/interrupts/>

⁸http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html#ga68c330e94fe121eba993e5a5973c3162

⁹http://www.nongnu.org/avr-libc/user-manual/group__avr__interrupts.html#gaad5ebd34cb344c26ac87594f79b06b73

¹⁰<https://www.arduino.cc/en/Reference/SchedulerYield>

programma, werd in samenspraak met Carl Peto beslist die functie van geen alternatief te voorzien, vandaar regel 4 van Codevoorbeeld 4.2. Ongeveer dezelfde redenering kan gevuld worden voor de functie `micros()`, die in Codevoorbeeld B.2 onder andere in regel 342 aangeroepen wordt. Hoewel de documentatie¹¹ in dit geval toch een iets duidelijker beschrijving van haar functionaliteit geeft, lijkt de functie ook hier weer van weinig groot belang voor de werking van het huidige project. Omdat de originele functie een `unsigned long` teruggeeft, werd ervoor geopteerd standaard een nulwaarde te retourneren, zoals geïllustreerd door regel 5 in Codevoorbeeld 4.2.

De bronbestanden zijn geïmporteerd en werken. Tijd dus om een effectief programma te schrijven in `main.swift`. Aangezien het niet onbelangrijk is de weegschaal te kalibreren, wordt de originele `Calibration.ino`-sketch¹² allereerst herschreven in Swift. Uiteraard maakt deze gebruik van functies uit de bronbestanden, dus om daarmee aan de slag te gaan in `swift.main`, moet deze toegang hebben tot `HX711_ADC.cpp`. Helaas kan ook dit niet zomaar, maar gelukkig is de Swift for Arduino IDE hierop voorzien. Door in de menubalk van de IDE te navigeren naar *Project > Advanced > Add Clang Import Header*, genereert de IDE een nieuwe header file met de naam `main.h`. In dit bestand kunnen de functies die een bronbestand beschikbaar stelt en opgeroepen moeten kunnen worden in `main.swift`, worden gedeclareerd. Er blijft echter nog één probleem over: de functies in `HX711_ADC.cpp` zijn niet zomaar beschikbaar voor de buitenwereld. Gelukkig valt hier ook vrij eenvoudig een mouw aan te passen. De sleutel is te vinden in regels 21 tot en met 60 van Codevoorbeeld B.2, uitgeknippt en geplakt in Codevoorbeeld 4.3. Alles wat gedeclareerd is binnen `extern "C"`, kan immers opgevraagd worden door het `main.h`-bestand. De meeste functiedeclaraties in Codevoorbeeld 4.3 roepen zo de overeenkomstige C++-functies op en geven in bepaalde gevallen de geretourneerde waarde verder terug. Maar het kan ook uitgebreider: in regels 2 tot en met 15 wordt de functie `setupLoadCell()` gedeclareerd en die doet heel wat meer dan slechts één functie aanroepen. Deze `setupLoadCell()` is letterlijk opgebouwd uit enkele regels uit de `Calibration.ino`-sketch, die de eerste paar instellingen voor de HX711 amplifier module doorvoeren. In principe zou dit volledig opnieuw opgebouwd kunnen worden in Swift, maar daar is weinig noodzaak toe. Alle statements binnen de functie blijken perfect uitvoerbaar te zijn met de resources en dependencies die ze voorhanden hebben, dus levert het alleen maar tijdsverlies op door ze te laten als ze zijn. Codevoorbeeld B.5, ten slotte, toont de inhoud van het eerder genoemde `main.h` en laat zien hoe de nodige functies beschikbaar kunnen worden gesteld voor `main.swift` door eenvoudigweg hun namen, retourwaarden en eventuele parameters te declareren.

¹¹<https://www.arduino.cc/reference/en/language/functions/time/micros/>

¹²https://github.com/olkal/HX711_ADC/blob/master/examples/Calibration/Calibration.ino

```

1  extern "C" {
2      bool setupLoadCell() {
3          LoadCell.begin();
4          unsigned long stabilizingtime = 2000; // precision right after power-up can be improved by adding a few seconds of stabilizing time
5          boolean _tare = true; //set this to false if you don't want tare to be performed in the next step
6          LoadCell.start(stabilizingtime, _tare);
7          if (LoadCell.getTareTimeoutFlag() || LoadCell.getSignalTimeoutFlag()) {
8              return false;
9          }
10         else {
11             LoadCell.setCalFactor(1.0); // user set calibration value (float), initial value 1.0 may be used for this sketch
12         }
13         while (!LoadCell.update());
14         return true;
15     }
16     uint8_t updateLoadCell() {
17         return LoadCell.update();
18     }
19     void tareNoDelayLoadCell() {
20         LoadCell.tareNoDelay();
21     }
22     bool getTareStatusLoadCell() {
23         return LoadCell.getTareStatus();
24     }
25     bool refreshDataSetLoadCell() {
26         return LoadCell.refreshDataSet();
27     }
28     float getNewCalibrationLoadCell(float known_mass) {
29         return LoadCell.getNewCalibration(known_mass);
30     }
31     float getDataLoadCell() {
32         return LoadCell.getData();
33     }
34     float getCalFactorLoadCell() {
35         return LoadCell.getCalFactor();
36     }
37     void setCalFactorLoadCell(float cal) {
38         LoadCell.setCalFactor(cal);
39     }
40 }
```

Codevoorbeeld 4.3: Regels 21 t.e.m. 60 uit *HX711_ADC.cpp*

Het eigenlijke vertaalwerk van de *Calibration.ino*-sketch in Wiring naar Swift, is eerder eenvoudig in vergelijking met alle hierboven beschreven stappen - zeker voor ontwikkelaars die C of C++ niet machtig zijn. De documentatie die de Swift for Arduino IDE aanbiedt - te raadplegen door in de menubalk naar *Help > Documentation* te navigeren - en de code snippets die te vinden zijn in het linkerpaneel van de IDE, kunnen hierbij als handige naslagwerken dienen. Het is wel aangewezen het project gereeld te compileren, aangezien de documentatie vaak wat achterloopt op nieuwe updates voor de IDE, waardoor de compiler ettelijke functies soms lichtjes anders interpreteert dan zoals ze beschreven worden in de documentatie. Gelukkig worden eventuele foutberichten van duidelijke context voorzien in het rechterpaneel van de IDE en wordt snel duidelijk waar het schoentje wringt.

Codevoorbeeld B.1 toont de uiteindelijke *vertaling* van de originele sketch. Het is wenselijk daar kort het een en ander uit te bespreken. De eerste stop is regel 2, want daar gebeurt iets erg belangrijks. Dankzij het *import AVR*-statement kan het programma immers alles doen wat zo typisch is aan het aansturen van microcontrollers: pinnen bestempelen als INPUT of OUTPUT, waarden van pinnen lezen, waarden naar pinnen verzenden, de seriële poort besturen, enzovoort. Een Swift for Arduino-project zonder oproep naar de AVR libary is met andere woorden vrij zinloos.

De volgende regel waar iets noemenswaardig gebeurt, vervolgens, is regel 11. In regel 11 wordt immers de eerder besproken functie `setUpLoadCell()` opgeroepen. Zoals verwacht, zal de microcontroller hier regels 85 tot en met 98 van Codevoorbeeld 4.3 uitvoeren, om de draad daarna terug op te pikken in *main.swift*. Maar vooraleer de `loop()`-functie - in Swift for Arduino aangeduid als een simpele `while true` - kan worden aangevat, roept regel 18 eerst nog de functie `calibrate()` op. De declaratie van `calibrate()` start in regel 56, maar echt interessant wordt het pas vanaf regel 64. Aangezien het programma vanaf hier seriële invoer verwacht van de gebruiker - in de Swift for Arduino IDE kan er in het uitvoerpaneel (onderste paneel) ook gewoon getypt worden - moet het blijven stilstaan, vandaar de `while !_resume`. De functie `available()` controleert vervolgens of er invoer beschikbaar is en zo ja, ruimt plaats voor de functie `read()`. Zoals de naam suggereert, leest `read()` de eerstvolgende waarde die binnenkomt op de seriële invoer. Deze waarde wordt teruggegeven als `UInt8`-representatie van de Unicode-waarde van het ingevoerde karakter, waardoor in regels 69 eenvoudig gecontroleerd kan worden of het verwachte karakter ingevoerd werd. Even verder in regel 83, vervolgens, wordt de functie `getMultiDigitFloatFromSerial()` opgeroepen. Regel 104 is de plaats waar diens functiedeclaratie start en tevens de start van enkele vrij ongewone gebeurtenissen. Bedoeling van deze functie is immers om een kommagetal uit de seriële invoer te lezen, op zich geen aartsmoeilijke opdracht, ware het niet dat Swift for Arduino enkel de eerder besproken functie `read()` aanbiedt, een functie die slechts één karakter per keer kan lezen. Om dat probleem te omzeilen, wordt in `getMultiDigitFloatFromSerial()` gebruik gemaakt van een array, die vervolgens stukje bij beetje wordt opgevuld. Bedoeling is om een `Float` te construeren die uit zes cijfers bestaat - vier cijfers voor en twee cijfers na de komma - door van links naar rechts te werken. Per cijfer wordt `read()` opgeroepen, wordt het ingevoerde karakter geconverteerd naar een `Float`, wordt die `Float` vermenigvuldigd met de juiste macht van tien en wordt die uitkomst uiteindelijk op de juiste plaats in de array opgeslagen. Ten

slotte worden alle waarden in die array bij elkaar opgeteld, waarna de geconstrueerde `Float` uiteindelijk geretourneerd wordt.

Terug naar `calibrate()`, regel 89. Daar wordt de ingevoerde massa van het gewicht op de weegschaal - de waarde die dus werd bekomen aan de hand van de hierboven beschreven functie - als parameter meegegeven met `getNewCalibrationLoadCell(known_mass)`. Dit is een functie die gedeclareerd is in `HX711_ADC.cpp`, dus daar wordt het werk uitgevoerd om de kalibratiewaarde te bekomen. De `Float` die deze functie teruggeeft, is het getal waar het in dit project allemaal om draait: door deze waarde te gebruiken in alle andere projecten van dezelfde weegschaal, zal de HX711 amplifier module correcte en zinvolle meetwaarden afleveren. Precies om die reden is het aangewezen deze waarde op te slaan in de EEPROM van de Arduino. De EEPROM van een Arduino is een kleine hoeveelheid vast geheugen die blijft bestaan, ook als het apparaat wordt voorzien van een nieuw programma of uitgeschakeld wordt. In regel 95 van Codevoorbeeld B.1 wordt daarvoor de functie `saveValueInEEPROM(newCalibrationValue)` opgeroepen. De declaratie van deze functie start in regel 158 en vraagt de gebruiker eerst of hij of zij de kalibratiewaarde wel effectief wil opslaan. Zo ja, wordt in regel 169 `writeEEPROM(address: calVal_eepromAddress, value: UInt8(value))` opgeroepen. Deze functie doet precies wat ervan verwacht kan worden, maar komt qua functionaliteit helaas tekort in dit project. Het is immers enkel mogelijk `UInt8`-waarden op te slaan en geen `Float`-datatypes. Om dit probleem te overkomen, zou eventueel getracht kunnen worden de `Float` op te breken in aparte cijfers en die als `UInt8`-waarden op te slaan op meerdere adressen van de EEPROM. Maar met het oog op tijdverlies, werd er in deze scriptie voor gekozen dat pad niet te bewandelen en vrede te nemen met het feit dat het schrijven van de kalibratiewaarde naar de EEPROM in dit project niet naar behoren werkt.

Dit betekent meteen ook dat alle noemenswaardige zaken uit de `calibrate()`-functie overlopen zijn en er ten slotte nog snel een blik geworpen kan worden op de `loop()`-functie of `while true`-loop die start in regel 20. Allereerst wordt aan de hand van de uitkomst van `updateLoadCell()` nagegaan of er een nieuwe meetwaarde beschikbaar is. In het positieve geval wordt die waarde opgehaald en naar de seriële uitvoer geschreven, waarna er een vertraging van 100 milliseconden wordt ingebouwd, met als doel de seriële uitvoer niet te sterk te beladen. Vanaf regel 38 wordt vervolgens gecontroleerd of er een waarde is binnengekomen op de seriële invoer en als dat effectief zo is, wordt die vergeleken met de karakters `t`, `r` en `c`. Indien de invoer overeenkomt met een van die karakters, wordt respectievelijk de nulwaarde van de weegschaal opnieuw ingesteld, wordt het kalibratieproces opnieuw doorlopen of wordt de gebruiker gevraagd naar een eigen kalibratiewaarde. In het eerste geval wordt de functie `tareNoDelayLoadCell()` opgeroepen. Ook dit is een functie die zich afspeelt in `HX711_ADC.cpp`, maar onderbreekt het programma daar niet voor. De while-loop kan gewoon doorgaan en in regel 51 wordt de vooruitgang ervan gecontroleerd. In het geval de nulwaarde van de weegschaal effectief opnieuw is ingesteld, wordt de gebruiker daarvan op de hoogte gebracht, waarna de while-loop gewoon opnieuw begint.

4.2.3 Conclusie

Zonder de taak van de eindconclusie van deze scriptie in zijn geheel te ondermijnen, is het toch al eens de moeite een korte conclusie van het hierboven beschrevene op te stellen. Vooreerst is het weinig waarschijnlijk dat veel ontwikkelaars zo'n verregaande kennis hebben van C en C++ om wat beschreven is in het eerste deel van Subsectie 4.2.2 op eigen houtje uit te pluizen. Het probleem is ook dat wat in die subsectie beschreven is, niet als lichtend voorbeeld gebruikt kan worden. Elk project is immers verschillend en elk project steunt op andere libraries. De beste situatie is natuurlijk die situatie waarbij geen C- en/of C++-libraries nodig zijn, maar indien dat toch het geval is, lijkt het beter de library in kwestie van in het begin te vertalen naar Swift en daar verder op te steunen. Uiteraard is dit een tijdverslindende taak, maar zich uitsluitend verhalen op Swiftcode, maakt het uiteindelijke programma niet alleen overzichtelijker, maar vooral ook minder kwetsbaar voor bugs en veiligheidsproblemen. Daarnaast kan men zich de vraag stellen of het überhaupt zoveel extra tijd vergt de library in kwestie te vertalen, gezien de stevig portie tijd die ook het doen compileren van diens bronbestanden vraagt. Om een keuze te maken, moeten de voor- en nadelen per project tegen elkaar afgewogen worden. Feit is wel dat indien er effectief nood is aan C- of C++-libraries, extra tijd moet worden voorzien wanneer gekozen wordt voor de Swift for Arduino IDE boven de Arduino IDE.

Vooraleer een punt te zetten achter alles wat de weegschaal betreft, moet nog kort even meegegeven worden dat veel geïntegreerde circuits die verkocht worden via webshops, meer dan eens niet naar behoren werken. Dit bleek ook het geval te zijn bij de HX711 amplifier module waarmee gewerkt werd in de bovenstaande paragrafen. Concreet bleken de meetwaarden die de module beschikbaar stelde, van weinig logica te getuigen en ontzettend volatiel te zijn. Wanneer de module aangesloten wordt, ontvangen zowel het originele project voor de Arduino IDE als het project voor de Swift for Arduino IDE dezelfde soort waarden, wat aangeeft dat het probleem zich - gelukkig - niet bij de software, maar bij de hardware bevindt. Dat maakt dat deze tegenvaller op zich geen grote gevolgen heeft voor het onderzoek en het verdere verloop van deze scriptie, maar toch moet voor de volgende sectie lichtjes bijgestuurd worden. Simpel gezegd, zal de weegschaal vervangen worden door een potentiometer. Dit impliceert dat de kleur van de ledstrip zal afhangen van de mate waarin de potentiometer opengedraaid is en dat de waarden die naar de Raspberry Pi worden gestuurd, afkomstig zullen zijn van het alternatieve component.

4.3 Communicatie

Het tweede deel van de proof of concept slaat op het voorzien van communicatiecapaciteiten aan het IoT-apparaat. Enerzijds moet de Arduino lichtsignalen afgeven om de gebruiker fysiek en direct op de hoogte te stellen van de status van de prullenbak. Anderzijds moeten de meetwaarden van de weegschaal - in dit geval gesimuleerd door een potentiometer - doorgestuurd worden naar een Raspberry Pi, zodat die laatste de data beschikbaar kan stellen via het internet.

4.3.1 Arduino

De Arduino is het startpunt van heel deze operatie, dus lijkt het geen slecht idee eerst te focussen op een nieuw Swift for Arduino-bestand. De inhoud van dat bestand wordt gepresenteerd in Codevoorbeeld B.7 en start meteen met het instellen van enkele pinnen. Om het lezen wat aangenamer te maken, toont ook Codevoorbeeld 4.4 die statements, samen met de rest van de *setup*. Er wordt in de eerste plaats voor gekozen de ledstrip aan te sluiten op pin 12 en die logischerwijs te markeren als OUTPUT-pin, terwijl de potentiometer verwacht wordt op de analoge pin A0. Aangezien alle datapinnen standaard aanzien worden als INPUT-pinnen, is het in principe niet nodige pin A0 als dusdanig te markeren. Met het oog op het bewaren van overzicht, gebeurt dat in regel 4 alsnog. In regel 8 wordt het aantal lichtjes dat de aangesloten ledstrip telt, ingesteld, waarna die waarde in de volgende regel meteen gebruikt wordt om de ledstrip in zijn geheel voor te stellen aan het programma. De parameters die meegegeven worden met de functie `iLEDFastSetup(...)`, zijn uiteraard afhankelijk van de gebruikte ledstrip en het project in kwestie. In regel 12, vervolgens, wordt de variabele `currentColorLeds` van het type `iLEDFastColor` ingesteld op `iLEDOff`. Het datatype `iLEDFastColor` is eigenlijk niets anders dan een typealias voor `UInt32`, terwijl `iLEDOff` die *kleur* represeneert waarbij de lichtjes van een ledstrip gedoofd zijn. Vooraleer de while-loop aan de beurt is, wordt de seriële poort in regel 16 geïnitialiseerd, om kort daarna, in regel 20, meteen gebruikt de worden in de vorm van de aanroep naar `sendMassWeightToSerial(currentMassWeight)`. Codevoorbeeld 4.5 bevat de declaratie van deze functie en toont dat ze niets meer doet dan de meegegeven waarde naar de seriële uitvoer te schrijven, geflankeerd door vierkante haakjes. Deze haakjes maken een apparaat dat de seriële uitvoer van de Arduino in de gaten houdt - een Raspberry Pi in het geval van deze proof of concept - duidelijk wanneer een inkomende waarde start en eindigt. Ook interessant om kort even bij stil te staan, is hoe de ingebouwde functies `print(...)` verschillende vormen aannemen. In het geval een statische string wordt meegegeven, moet de parameternaam expliciet vermeld worden, terwijl dat geen vereiste is voor `UInt16`-waarden. De documentatie stelt zo een heleboel verschillende gedaantes van dezelfde functie voor, in die mate zelfs dat bepaalde daarvan volledig dezelfde functionaliteit leveren en - vooral - precies dezelfde parameterwaarden verwachten. Wat dat betreft, kan Swift for Arduino weleens een poetsbeurt gebruiken ...

```

1 // PIN SETUP
2 let potentiometerPin: Pin = A0
3 let ledStripPin: Pin = 12
4 pinMode(pin: potentiometerPin, mode: INPUT)
5 pinMode(pin: ledStripPin, mode: OUTPUT)
6
7 // LED STRIP SETUP
8 let amountOfLeds: UInt16 = 60
9 iLEDFastSetup(pin: ledStripPin, pixelCount: amountOfLeds,
    hasWhite: false, grbOrdered: true)
10
11 // VARIABLES SETUP
12 var currentColorLeds: iLEDFastColor = iLEDOff
13 var currentMassWeight: UInt16 = 0
14
15 // SERIAL SETUP
16 SetupSerial()
17 // Time for serial port to stabilize
18 delay(milliseconds: 250)
19 // Already write start value to serial
20 sendMassWeightToSerial(currentMassWeight)

```

Codevoorbeeld 4.4: Regels 3 t.e.m. 22 uit *main.swift*

```

1 func sendMassWeightToSerial(_ value: UInt16) {
2     print(staticString: "[", addNewline: false)
3     print(value, addNewline: false)
4     print("]")
5 }

```

Codevoorbeeld 4.5: Regels 61 t.e.m. 65 uit *main.swift*

Tijd dan voor Codevoorbeeld 4.6, want die toont de while-loop. In regel 2 gebeurt meteen iets interessants, daar wordt immers een asynchrone functie opgeroepen. *Asynchroon* moet hier wel tussen aanhalingsstekens geplaatst worden, want aangezien een Arduino haar programmacode slechts op één thread uitvoert, kan er van echte *asynchroniciteit* geen sprake zijn. Om toch enigszins wat in de buurt te komen van dat concept, maakt een Arduino gebruik van interrupts. In het geval van de functie die aangeroepen wordt in regel 2 - `analogReadAsync(...)` - manifesteert dit zich als volgt: de Arduino wordt opgedragen de waarde op de aangegeven pin door te geven en in afwachting van dat antwoord, worden de volgende paar statements alvast uitgevoerd. Later - in elektronische termen gaat

het hier over een fractie van een seconde - wanneer het antwoord het programma bereikt, wordt een interrupt uitgestuurd en wordt het programma onderbroken om de statements die mee zijn gegeven aan de functieaanroep van de asynchrone functie waar alles mee begon, uit te voeren. Hierna gaat het programma gewoon weer verder waar het gebleven was. Critici kunnen opperen - en gelijk hebben ze - dat het gebruik van een asynchrone functie in het geval van deze while-loop, weinig zin heeft. De functie wordt immers niet gevuld door een andere functieaanroep, behalve dan door `delay(milliseconds: 1000)`, die de while-loop gedurende één seconde zelfs onderbreekt. Toch werd in dit project voor de asynchrone functie gekozen, aangezien het onderwerp van interrupts een waardevolle toevoeging is aan deze scriptie.

```

1 while true {
2     analogReadAsync(pin: potentiometerPin) { value in
3         changeLedStripColorDependingOn(massWeight: value)
4
5         // Only write to serial when necessary
6         if value != currentMassWeight {
7             sendMassWeightToSerial(value)
8             currentMassWeight = value
9         }
10    }
11
12    delay(milliseconds: 1000)
13 }
```

Codevoorbeeld 4.6: Regels 25 t.e.m. 37 uit *main.swift*

Verder naar regel 3. Daar wordt `changeLedStripColorDependingOn(massWeight: value: UInt16)` aangeroepen. De declaratie van deze functie wordt afgebeeld in Codevoorbeeld 4.7. De eerste paar regels controleren de meegegeven waarde en geven op basis daarvan aan welke kleur de lichtjes in de ledstrip moeten aannemen. Constanten `iLEDRed` en `iLEDoff` zijn slechts een greep uit de kleuren die Swift for Arduino reeds standaard aanbiedt, maar uiteraard moet een ontwikkelaar ook in staat zijn nieuwe kleuren te declareren. Om dat te bewerkstelligen, bestaat de functie `iLEDFastMakeColor(red: UInt8, green: UInt8, blue: UInt8, white: UInt8)`. Moeilijk om uit te leggen, is die niet: door eenvoudigweg de RGB-waarden en eventueel een witwaarde mee te geven, kan elke kleur van de regenboog geproduceerd worden. Maar vooraleer die kleur effectief ingesteld kan worden, controleert regel 12 eerst nog of dat effectief nodig is. Het is immers overbodig en te belastend voor de ledstrip om bij elke nieuwe meetwaarde opnieuw ingesteld te worden. Als toch van kleur veranderd moet worden, echter, wordt `iLEDFastWritePixel(color: colorToDisplay)` in regel 14 zoveel keer aangeroepen als er lichtjes zijn. Deze functie doet precies wat haar naam impliceert te doen, maar heeft toch even tijd nodig om volledig tot ontplooiing te komen in de fysieke wereld. Daarom wordt het programma in regel 17 kort gepauzeerd,

om daarna weer te hervatten.

```
1 func changeLedStripColorDependingOn(massWeight value: 
2     UInt16) {
3     let colorToDisplay: iLEDFastColor
4     if value > 700 {
5         colorToDisplay = iLEDRed
6     } else if value > 400 {
7         colorToDisplay = iLEDFastMakeColor(red: 255,
8             green: 165, blue: 0, white: 0)
9     } else {
10        colorToDisplay = iLEDOff
11    }
12
13    // Only change color when necessary
14    if colorToDisplay != currentColorLeds {
15        for _ in 1...amountOfLeds {
16            iLEDFastWritePixel(color: colorToDisplay)
17        }
18        // Time for LEDs to change color
19        delay(microseconds: 6)
20        currentColorLeds = colorToDisplay
21    }
22 }
```

Codevoorbeeld 4.7: Regels 40 t.e.m. 59 uit *main.swift*

Wanneer de Arduino de gebruiker op de hoogte heeft gesteld van de huidige meetwaarde door gebruik te maken van de ledstrip, is het tijd om diezelfde waarde naar de seriële poort te schrijven. Toch is dit niet altijd nodig. Het apparaat aan de andere kant van de *lijn* blijven bestoken met dezelfde waarde, is immers teveel van het goede. Daarom wordt in regel 4 van Codevoorbeeld 4.6 eerst gecontroleerd of de nieuwe waarde wel verschilt van de vorige. Zo ja, wordt in regel 7 `sendMassWeightToSerial(_ value: UInt16)` opgeroepen - deze functie is reeds bekend - en wordt de waarde van `currentMassWeight` bijgewerkt. Om de ledstrip en seriële poort niet te overbelasten - en omdat het volstrekt onnodig zou zijn - wordt de while-loop in regel 12, ten slotte, nog even stilgelegd. Concreet betekent dit dat de Arduino ongeveer elke seconde een nieuwe meetwaarde ophaalt en - indien nodig - de ledstrip en seriële poort aanspreekt.

4.3.2 Raspberry Pi

De Raspberry Pi op zich is niet de focus van deze scriptie, toch is ze van groot belang bij de ontwikkeling van de slimme prullenbak. De minicomputer staat immers in voor

de internetcommunicatie. Gelukkig is het project dat hiervoor nodig is, reeds geschreven. Axel Roest stelde ooit een eigen IoT-apparaat voor¹³ en stelde de daarbij horende bronbestanden ter beschikking via zijn GitLab account¹⁴. De opstelling die hij bedacht voor zijn project, is zo eenvoudig en toch effectief, dat het de basis vormt van de proof of concept uit dit hoofdstuk. Alles wat Arduino betreft, is natuurlijk uniek voor elk IoT-apparaat, dus die bestanden zijn niet met elkaar te vergelijken. Maar wanneer het over de communicatie naar de Raspberry Pi en diens eigen werking gaat, verwateren de verschillen. In die mate zelfs dat Axel Roests originele project voor de Raspberry Pi, zonder aanpassingen - behalve dan wat de databankinstellingen betreft - toegevoegd kan worden aan de opstelling van de slimme prullenbak. Ook de bronbestanden van dit project zijn terug te vinden in een GitLab repository¹⁵ en vormen samen een Swiftproject dat als een soort *daemon* uitgevoerd kan worden op een macOS- of Linuxsysteem. Om de applicatie op te starten, dient het commando `plantinject <serialport device>` uitgevoerd te worden in een terminalvenster. Parameter `<serialport device>` moet uiteraard vervangen worden door de naam van de seriële poort waarop de seriële communicatie van de Arduino binnenkomt, waardoor het uiteindelijke commando eruit zal zien in trend van `plantinject /dev/cu.usbmodem14101`. Een eenvoudige USB-verbinding tussen Arduino en Raspberry Pi volstaat al om die seriële communicatie op poten te zetten. (Roest, 2020a)

Roests Swiftproject lijkt op het eerste zicht vrij indrukwekkend wegens de vele bestanden en lijnen code, maar een tweede blik leert al snel dat de opbouw eigenlijk vrij eenvoudig is. Er zijn drie hoofdcomponenten: de startplaats van het programma, het systeem dat de waarden die binnenkomen op de meegegeven seriële poort, capteert en het systeem die die waarden naar de ingestelde databank schrijft. De codebestanden die die drie zaken bewerkstelligen, heten respectievelijk `main.swift`, `SerialHandler.swift` en `SQLiteHandler.swift`, en worden weergegeven door Codevoorbeelden B.8, B.9 en B.10. Hieronder worden de belangrijkste zaken uit die bestanden kort van een uitleg voorzien.

Beginnen doet men in het begin en in dit geval is dat `main.swift`, waarvan Codevoorbeeld 4.8 de belangrijkste regels toont. In de eerste regel gebeurt meteen iets interessants: een `SerialHandler` - die in moet staan voor de seriële communicatie - wordt ingesteld. Daarbij worden twee zaken meegegeven: allereerst de eerder aangeduidde seriële poortnaam en in de tweede plaats een functie die de `SerialHandler` later kan aanroepen om een nieuwe waarde naar de databank te schrijven. In regel 7, vervolgens, wordt de functie `waitForSerial()` op de `SerialHandler` aangeroepen. Vooraleer naar die klasse te kijken, echter, verdient regel 10 toch ook nog even een kort *moment de gloire*. Dankzij de functie die daar opgeroepen wordt, blijft het script immers draaien, wat betekent dat zolang het niet beëindigd wordt, `serialHandler` haar functie `waitForSerial()` blijft afvuren.

¹³<https://www.youtube.com/watch?v=09ro41ECED8>

¹⁴<https://gitlab.com/axello>

¹⁵<https://gitlab.com/axello/plantinject>

```
1 guard let serialPortHandler = SerialHandler(device:
2     serialDevice, injector: sqliteInjector) else {
3 }
4 print("Hello, serial device: \(serialDevice)")
5
6 serialHandler = serialPortHandler
7 serialHandler.waitOnSerial()
8
9 // When backgrounding daemon style, use & to keep script
10 RunLoop.main.run()
```

Codevoorbeeld 4.8: Regels 49 t.e.m. 58 uit *main.swift* (Roest, 2020b)

SerialHandler dus. Codevoorbeeld B.9 toont de declaratie van deze klasse en herbergt daarom ook de declaratie van `waitOnSerial()`. Ook Codevoorbeeld 4.9 toont deze functie en meteen springt in het oog dat er toch een klein verschil gemaakt wordt tussen macOS- en Linuxapparaten. Aangezien deze proof op concept met een Raspberry Pi werkt, wordt verder gefocust op de functieaanroep `linuxBackgroundRead()`, afgebeeld in Codevoorbeeld 4.10. Deze functie leest enkele karakters die binnengekomen zijn op de seriële poort (zie regel 7) en stuurt die in regel 11 door naar `processLine(_line: String)`. Deze laatste wordt tentoongesteld in Codevoorbeeld 4.11, maar behoeft toch een korte samenvatting van wat daar precies gebeurt: er wordt op zoek gegaan naar een begin-tag - zijnde `/` - en even verder naar de eind-tag - `/`. De string van karakters - de meetwaarde van de Arduino dus - tussen beide tags wordt ten slotte geretourneerd, vergezeld door de rest van de binnengekomen string, zodat die data in de volgende oproep van dezelfde functie verder verwerkt kan worden. Terug in Codevoorbeeld 4.10, wordt in lijn 13, ten slotte, de bekomen meetwaarde meegegeven met de functie die bij de aanroep van `SerialHandler` werd meegegeven. Ter herinnering: dit is de functie die instaat voor de oproepen naar de databankfuncties en wordt gedeclareerd in Codevoorbeeld 4.12. En effectief, in regel 3 wordt daar een `SQLiteHandler` gevraagd de functie `write(sensorId: 1, sensorValue: Double(intValue))` uit te voeren. Vooraleer haar declaratie in meer detail te bekijken, echter, is het aangewezen toch even stil te staan bij de parameter met naam `sensorId`. In dit geval wordt de waarde `1` meegegeven, maar in de situatie waarin meerdere slimme prullenbakken dezelfde databank aanspreken, zou elk van hen uiteraard een uniek identificatienummer moeten hebben. Aangezien deze proof of concept slechts één prullenbak beschouwt, volstaat identificatienummer `1` echter.

```

1 func waitOnSerial() {
2 #if os(Linux)
3     linuxBackgroundRead()
4
5 #elseif os(OSX)
6     if let port = port {
7         self.backgroundRead(port: port, dataHandler: self
8             .dataInjector)
9     }
10 #endif
11 }
```

Codevoorbeeld 4.9: Regels 53 t.e.m. 62 uit *SerialHandler.swift* (Roest, 2020c)

```

1 func linuxBackgroundRead() {
2     guard let port = port else { return }
3
4     var line = ""
5     while true {
6         do{
7             let data = try port.readData(ofLength: 2)
8             if let text = String(data: data, encoding: .
9                 utf8) {
10                 line.append(text)
11             }
12             let (value, rest) = processLine(line)
13             if let value = value {
14                 dataInjector(value)
15             }
16             line = rest
17         } catch {
18             print("Error: \(error)")
19         }
20     }
21 }
```

Codevoorbeeld 4.10: Regels 64 t.e.m. 83 uit *SerialHandler.swift* (Roest, 2020c)

```

1 func processLine(_ line: String) -> (String?, String) {
2     if let tagIndex = line.firstIndex(of: startTag),
3         tagIndex < line.endIndex {
4         let startOfSentence = line.index(after: tagIndex)
5         let firstSentence = line[startOfSentence...]
6
7         if let endTagIndex = firstSentence.firstIndex(of:
8             endTag) {
9             let endOfValue = firstSentence.index(before:
10                endTagIndex)
11            let value = firstSentence[...endOfValue]
12            let afterEndTagIndex = firstSentence.index(
13                after: endTagIndex)
14            let restString = firstSentence[
15                afterEndTagIndex...]
16            return (String(value), String(restString))
17        }
18    }
19    return (nil, line)
20 }
```

Codevoorbeeld 4.11: Regels 105 t.e.m. 119 uit *SerialHandler.swift* (Roest, 2020c)

```

1 func sqliteInjector(_ string: String) {
2     if let intValue = Int(string) {
3         db.write(sensorId: 1, sensorValue: Double(
4             intValue))
5     }
6     print("value: \(string)")
```

Codevoorbeeld 4.12: Regels 23 t.e.m. 28 uit *main.swift* (Roest, 2020b)

Om af te sluiten, is er dus de klasse `SQLiteHandler`, weergegeven in Codevoorbeeld B.10. In de initializer wordt `setup()` aangeroepen, die op haar beurt `createTable()` aanroept. Startend in regel 40, vraagt deze functie de databank een nieuwe tabel voor alle meetwaarden te genereren, mocht deze nog niet bestaan. Vanaf regel 60, vervolgens, wordt de eerdergenoemde functie `write(sensorId: Int, sensorValue: Double)` gedeclareerd, die op die tabel steunt. Ook Codevoorbeeld 4.13 bevat die functiedeclaratie. Concreet wordt de tabel uitgebreid met een nieuwe record. De primaire sleutel buiten beschouwing gelaten, bestaat deze record uit drie waarden: het identificatie-

nummer van de sensor, de huidige datum en tijd, en - uiteraard - de nieuwe meetwaarde. Op die manier is het volledige proces van prullenbak wegen tot de bekomen meetwaarde beschikbaar stellen via het internet, meteen ook doorlopen. Axel Roest ging zelfs verder door - gebruik makend van *Vapor*¹⁶ - tevens een backendsysteem¹⁷ op te stellen in Swift en ontwikkelde zelfs een iOS-applicatie¹⁸ om de API die die backend beschikbaar stelt, te raadplegen. Dit alles toont aan dat het perfect mogelijk is een IoT-apparaat te ontwikkelen door enkel en alleen te steunen op Swift, maar of het ook de meest effectieve manier vormt, is voer voor de conclusie.

```

1 func write(sensorId: Int, sensorValue: Double) {
2     let query = "INSERT INTO Reading (sensor, date, value
3 ) VALUES (:1, :2, :3);"
4     let now = dateFormatter.string(from: Date())
5
6     do {
7         let sqlite = try SQLite(dbPath)
8         defer {
9             sqlite.close()
10    }
11        try sqlite.execute(statement: query) {
12            (stmt:SQLiteStmt) -> () in
13
14            try stmt.bind(position: 1, sensorId)
15            try stmt.bind(position: 2, now)
16            try stmt.bind(position: 3, sensorValue)
17        }
18    } catch (let error) {
19        print("error inserting reading: \(error)")
20    }
21 }
```

Codevoorbeeld 4.13: Regels 60 t.e.m. 80 uit *SQLiteHandler.swift* (Roest, 2020d)

¹⁶<https://vapor.codes/>

¹⁷<https://gitlab.com/axello/plants>

¹⁸<https://gitlab.com/axello/plantstatus>

5. Conclusie

Hoewel het onmogelijk is met één proof of concept alle mogelijke modi operandi van Swift voor Arduino aan te raken, was die uit Hoofdstuk 4 toch divers genoeg om een overkoepelende conclusie te trekken. Het ontwikkelproces van de slimme prullenbak omvatte immers alle procedures die aan bod komen bij de ontwikkeling van eender welk IoT-apparaat en daarbij uitsluitend te steunen op Swift. Die verschillende procedures kunnen gegroepeerd worden onder drie pijlers.

5.1 De drie pijlers van het ontwikkelproces

Allereerst is er die pijler die er sowieso bij elk ontwikkelproces aan te pas komt: het vanaf nul opbouwen van een programma. Elke project bevat immers wel een knop, een lampje of een andere losse component, die gelezen of aangestuurd moet worden. Het is echter evengoed mogelijk, maar geen vereiste, dat ook gesteund wordt op reeds bestaande code in de vorm van een library. Aangezien de proof of concept uit Hoofdstuk 4 heeft doen blijken dat dit in het geval van Swift for Arduino geen voor de hand liggende procedure is, kan dit meteen als tweede pijler beschouwd worden. De derde en laatste pijler, ten slotte, omhelst alles wat met het vastleggen van een internetverbinding te maken heeft. Een apparaat zonder internetverbinding kan immers bezwaarlijk onder de noemer *Internet of Things* geplaatst worden.

5.1.1 Vanaf nul beginnen

Net zoals dat bij schilderkunst het geval is, is een leeg canvas het begin van een programma. Om een idee werkelijkheid te laten worden, moet een ontwikkelaar code schrijven, vanaf nul. Het programma dat de slimme prullenbak doet oplichten en waarden naar haar seriële poort doet schrijven, is een voorbeeld van zo'n programma dat voor het ontstaan van het idee erachter, ook maar gewoon een leeg canvas was. Ondanks het feit dat dus vanaf nul gestart moest worden, verliep het ontwikkelproces van deze pijler bijzonder vlot. Binnen de kortste keren was het lege canvas omgetoverd tot een volledig werkzaam project. En hoewel dit ontwikkelaars met ervaring binnen de wereld van microcontrollers waarschijnlijk niet zal verrassen, is dit toch een belangrijke vaststelling. Het geeft immers aan dat de Swift for Arduino IDE en μ Swift even vlot te hanteren zijn als de Arduino IDE en Wiring. De Swift for Arduino-syntaxis is zelfs erg gelijkaardig aan de syntaxis van de Wiring API, waardoor de stap naar Swift for Arduino bijzonder klein is. Daarnaast is ook de meegeleverde documentatie erg duidelijk opgesteld en dus uiterst behulpzaam, ondanks het feit dat Swift for Arduino bepaalde delen ervan al voorbijgestreefd heeft.

Uiteraard is niet alles rozengeur en maneschijn. Hier en daar bleek uit de proof of concept immers dat Swift for Arduino toch enkele zaken mist in vergelijking met wat de Arduino IDE aanbiedt. Zo bleek dat enkel losse karakters op de seriële poort gelezen kunnen worden, geen volledige strings, en dat uitsluitend `UINT16`-waarden naar een Arduino's EEPROM geschreven kunnen worden, daar waar Wiring onder andere ook `float`-waarden toelaat. Ongetwijfeld zal Swift for Arduino nog wel van die kinderziektes hebben, maar de proof of concept heeft ook aangetoond dat die mits wat slimmigheidjes best te omzeilen zijn. Daaruit kan ook meteen opgemaakt worden dat ontwikkelaars die graag aan de slag willen gaan met microcontrollers, trouw aan Swift willen blijven én projecten vanaf nul willen opbouwen, zonder problemen aan de slag kunnen met Swift for Arduino.

5.1.2 Bouwen op een bestaande library

Dat laatste kan helaas niet gezegd worden over de situatie waarbij gebouwd moet worden op een bestaande library. Uiteraard gaat het hier dan niet over libraries die reeds voor Swift for Arduino geschreven zijn, maar voor die groep in C of C++, vaak opgesteld met het oog op de Arduino IDE en dus gebruik makend van haar Wiring API. Hoe meer zo'n library immers steunt op dat laatste, hoe ingewikkelder het wordt die werkende te krijgen binnen de Swift for Arduino IDE, zo bleek ook uit het ontwikkelproces van de weegschaal uit de proof of concept. De ene library vergt meer aanpassingswerk dan de andere, maar in wezen komt het erop neer dat enkel ontwikkelaars met kennis van C en de erg low-level AVR-library, dit karwei echt op zichzelf tot een goed einde kunnen brengen. En aangezien softwarebedrijven met Swiftontwikkelaars aan boord en hobbyisten de doelgroepen van deze scriptie zijn, is de conclusie snel gemaakt dat de Arduino IDE nog steeds *the way to go* is wanneer er een library aan te pas komt.

Er is uiteraard ook gewoon de optie om de library in kwestie zelf opnieuw vanaf nul op te bouwen door deze te vertalen naar Swift. Subsectie 5.1.1 gaf al aan dat dit perfect mogelijk is en Carl Peto is ook een grote voorstander van deze opzet, maar uiteraard vergt zo'n

vertaalklus gevoelig veel extra tijd. Voor een softwarebedrijf - waar een goede tijdbesteding een van de hoogste deugden is - lijkt de Arduino IDE dan ook nog steeds de beste optie, maar voor hobbyisten liggen de kaarten anders. Hobbyisten zijn per definitie immers liefhebbers van een bepaalde hobby, dus wat extra tijd besteden aan een vrijetijdsproject, is voor hen geen rem - integendeel. Ook de Swift for Arduino community zou ontzettend tevreden zijn met enkele nieuwe leden en hun vertaalde libraries aan boord, want hoe meer Swift for Arduino libraries vrij beschikbaar zijn, hoe meer mensen Swift for Arduino kunnen ontdekken en ook deel gaan uitmaken van de community. Op die manier wordt een vicieuze cirkel in gang gezet en is het probleem van libraries op een bepaald moment misschien wel geen struikelblok meer voor softwarebedrijven die met Swift for Arduino aan de slag willen.

5.1.3 Verbinden met het internet

Deze scriptie focust zich op IoT-apparaten, dus mogen ook hun internetverbindingen niet uit het oog verloren worden. Een internetverbinding tot stand brengen, kan op meerdere manieren. Zo bestaan er Arduino libraries die in combinatie met het geschikte shield een Wi-Fi- of Ethernetverbinding kunnen bewerkstelligen. Echter toonde Subsectie 5.1.2 al aan waarom deze scriptie deze werkwijze niet aanmoedigt. Beter is om te steunen op een apparaat dat standaard van internetcapaciteiten is voorzien. De slimme prullenbak uit de proof of concept maakt hiervoor gebruik van een Raspberry Pi. Hoewel een Raspberry Pi meerdere programmeertalen ondersteunt en deze scriptie er vooral op gericht is Swift in combinatie met Arduino-apparaten te gebruiken, bewees de proof of concept dat de minicomputer zelfs nog beter met Swift om kan gaan. Bij het installeren van de taal wordt immers de volledige Swift Standard Library meegegeven, en dus niet slechts een onderdeel ervan, zoals dat bij Swift for Arduino wel het geval is. Van alle pijlers die bij het ontwikkelproces van een IoT-apparaat in Swift komen kijken, is deze laatste dus de meest eenvoudige en voor de hand liggende om in de praktijk te brengen. Dit geldt zowel voor softwarebedrijven als voor hobbyisten.

5.2 Onderzoeksvragen beantwoord

Tijd nu voor het orgelpunt van deze scriptie, tijd voor het beantwoorden van de onderzoeksvragen. Het is belangrijk hierbij nogmaals te onderstrepen dat alle conclusies die voortvloeien uit ervaringen met Swift for Arduino, gebaseerd zijn op versie 4.4 van de IDE en daarom niet per se getrokken kunnen worden uit ervaringen met eventuele toekomstige versies. Meer zelfs, de kans is vrij groot dat toekomstige versies van de IDE qua functionaliteit steeds meer zullen toegroeien naar de standaard Arduino IDE, waardoor de twee op een bepaald moment misschien wel praktisch inwisselbaar zijn.

Zover is het echter nog niet, want zeker wat het gebruik van libraries betreft, laat Swift for Arduino nog heel wat te wensen over. Deze vaststelling geeft meteen aanleiding tot het beantwoorden van de deelvraag wat de specifieke mogelijkheden zijn van Swift voor Arduino. Omdat het onbegonnen werk is een lijst van *features* op te sommen en het

definiëren van zo'n *feature*, eenvoudigweg onmogelijk is, wordt een vergelijking getrokken met de Arduino IDE. Wanneer van nul wordt begonnen en dus gebruik gemaakt moet worden van de typische microcontrollerfuncties - zoals het lezen en schrijven van waarden van en naar pinnen - is Swift for Arduino perfect in staat te bewerkstelligen wat de Arduino IDE haar gebruikers aanbiedt te doen. Hier en daar moet misschien een kleine *omweg* gemaakt worden, maar dat doet niet af van Swift for Arduino's capaciteiten. Aan de andere kant, echter, werd reeds aangegeven dat het systeem op moment van publicatie van deze scriptie nog tekortschiet wanneer beroep gedaan moet worden op externe libraries. Op dat vlak is de Arduino IDE niet van haar troon te stoten.

Die laatste conclusie is meteen ook onderdeel van het antwoord op de deelvraag tot wat Swift voor IoT in staat is in vergelijking met de gevestigde programmeertalen. Vooraleer hier een antwoord op te formuleren, echter, moet nog eens onderstreept worden dat deze scriptie hoofdzakelijk doelt op het gebruik van Arduino en in bredere zin microcontrollers. Hoewel het dus mogelijk is IoT-apparaten te ontwikkelen door uitsluitend gebruik te maken van een apparaat zoals een Raspberry Pi, vormt deze niet de focus van deze scriptie. En eenmaal het vizier dus volledig op Arduino gericht is, blijft er weinig keuze over dan grotendeels gebruik te maken van Swift for Arduino om *Swift voor IoT* echt te bewerkstelligen. Daarbovenop is de tegenpool van Swift for Arduino ontegensprekelijk de Arduino IDE en kan met *de gevestigde programmeertalen* logischerwijs verwezen worden naar C, C++ en Wiring - ook al is deze laatste strikt genomen geen programmeertaal. Dat maakt meteen ook dat de antwoorden op de vorige deelvraag hier gerecycleerd kunnen worden: de aansturing van losse componenten door gebruik te maken van de typische microcontrollerfuncties komt wel degelijk erg goed overeen met hoe het in de Arduino IDE zou gebeuren, maar wanneer er externe libraries bij betrokken worden, schiet Swift for Arduino gevoelig tekort. Dan is er tenslotte nog de internetverbinding. Behalve door er erg veel werk aan te besteden, is het niet mogelijk de Arduino zelf met het internet te verbinden. Toch hoeft dat in principe geen probleem te zijn, want een Raspberry Pi hiervoor aan de opstelling toevoegen, lijkt sowieso een beter idee. Op die manier kunnen fysieke *bewegingen* van het uiteindelijk IoT-apparaat en de internetgerelateerde functionaliteiten strikt van elkaar gescheiden worden. Daarnaast beschikt een Raspberry Pi hoe dan ook over veel meer van die *internetgerelateerde functionaliteiten* dan een Arduino, ongeacht het aantal shields en libraries waarop die laatste rekent. Swift voor IoT is met andere woorden dus perfect in staat te realiseren wat de *gevestigde programmeertalen* kunnen bewerkstelligen, behalve dan - en het wordt een terugkerend gegeven - wanneer er externe libraries in het spel zijn.

De deelvraag welke technologie - Arduino of Raspberry Pi - de voorkeur geniet, kan dan weer kort beantwoord worden. De proof of concept liet immers al uitschijnen dat elk systeem haar voordelen heeft en dat de twee dus best tot hun recht komen als ze samenwerken. Hoewel een ongetraind oog dit op het eerste gezicht misschien zou tegenspreken, is een van de twee als *beste* uitroepen, onbegonnen werk. Ze zijn immers niet te vergelijken.

Ten slotte is er nog de centrale vraag, de vraag waar heel deze scriptie om draaide: in hoeverre laten bestaande technologieën het gebruik van Swift voor de aansturing van IoT-apparaten toe? Gelukkig is het antwoord hierop niet ver weg, de deelvragen hebben de weg ernaartoe immers volledig geplaveid. Het antwoord klinkt dan ook erg herkenbaar. Zolang

er vanaf nul gestart wordt, is alles mogelijk. Losse componenten kunnen aangestuurd worden en het opzetten van een internetverbinding, is geen probleem. Zowel softwarebedrijven en hobbyisten kunnen dus prompt op deze manier aan de slag gaan. Echter, wanneer er per se een externe library aan te pas moeten komen, geldt deze redenering niet. Of toch niet voor softwarebedrijven. Voor hen is het de tijdsbesteding en het risico immers niet waard om de library ofwel werkende te krijgen binnen Swift for Arduino, ofwel te vertalen naar Swift. Zij houden beter vast aan de Arduino IDE. Voor hobbyisten, daarentegen, is die keuze natuurlijk volledig persoonlijk. Maar net omdat ze *hobbyist* zijn, lijkt het voor hen zeker de moeite waard het pad van Swift for Arduino te bewandelen. Deze kan misschien wat uitdagender blijken, maar een IoT-apparaat ermee aan de praat krijgen, geeft des te meer voldoening.

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

De dag van vandaag wordt het internet niet meer alleen gebruikt door ingewikkelde computersystemen zoals laptops, vaste computers en smartphones, ook apparaten met een minder uitgebreid spectrum aan mogelijkheden gebruiken het communicatiemedium steeds meer en meer. Die verzameling van via het internet verbonden apparaten wordt het *Internet of Things* of kortweg *IoT* genoemd. Een lamp die bestuurd wordt via een mobiele applicatie, een garagepoort die opengaat wanneer een bewoner de oprit oprijdt of een koelkast die automatisch eten bestelt wanneer nodig, zijn voorbeelden van apparaten die onder deze noemer vallen. Allemaal hebben ze een internetverbinding nodig om te functioneren.

Het ontwikkelen van IoT-apparaten is tweeledig. Enerzijds is er het puur fysieke aspect, de nodige elektronica-componenten moeten met elkaar en met het computerbrein van het apparaat verbonden worden. Elektronica-ingenieurs zijn hiervoor opgeleid. Anderzijds moet dat computerbrein geprogrammeerd worden, zodat de verschillende elektronica-componenten op een correcte manier samenwerken en de verhoopte functionaliteit tot stand brengen. In de praktijk zijn het ook vaak diezelfde elektronica-ingenieurs die zich daarmee bezighouden. Nochtans zou iemand die puur opgeleid is als informaticus misschien meer kunnen betekenen op dat vlak, ware het niet dat veel van die informatici de programmeertalen om IoT-apparaten aan te sturen, niet voldoende machtig zijn. Ook bestaande softwarebedrijven die dromen met IoT aan de slag te gaan, worden om dezelfde reden

tegengehouden die droom na te jagen.

Er is duidelijk nood aan een alternatief voor de gevestigde programmatiestandaarden van IoT-apparaten. Om die reden werden enkele projecten opgestart om Arduino's en Raspberry Pi's - de twee meest gekozen *computerbreinen* voor IoT-apparaten - aan te sturen aan de hand van Swift. Swift is de standaardtaal voor de ontwikkeling van onder andere iPhone-applicaties en wordt daarom reeds beheerd door heel wat informatici. Die projecten staan echter nog in hun kinderschoenen, dus is het voor informatici en softwarebedrijven vooralsnog niet tot nauwelijks duidelijk in hoeverre deze projecten toelaten Swift voor de aansturing van IoT-apparaten te gebruiken. Deze bachelorproef zal daarom volledig rond deze vraag draaien. Specifieker zullen antwoorden gezocht worden op de deelvragen tot wat Swift voor IoT in staat is in vergelijking met de gevestigde programmeertalen, welke technologie - Arduino of Raspberry Pi - de voorkeur geniet wanneer gewerkt wordt met Swift en wat de specifieke mogelijkheden zijn van Swift voor Arduino.

A.2 State-of-the-art

A.2.1 Internet of Things

In de introductie werd gewag gemaakt van het feit dat een apparaat in verbinding moet staan met het internet om onder de noemer *Internet of Things* te vallen. Toch is dit geen vereiste, want hoewel IoT-apparaten in de praktijk meestal over een internetverbinding beschikken, kan er nog steeds sprake zijn van IoT wanneer een communicatietechnologie gebruikt wordt waarmee geen internetverbinding tot stand wordt gebracht. Om het begrip toch enigszins af te bakenen, werden een aantal kenmerken opgesteld door Mattern en Floerkemeier (2010). Hieronder worden enkele daarvan toegelicht.

- **Communicatie en samenwerking**

IoT-apparaten moeten met elkaar communiceren om samenwerking mogelijk te maken. Technologieën zoals Wi-Fi en GSM kunnen een internetverbinding tot stand brengen, maar ook Bluetooth kan bijvoorbeeld gebruikt worden om apparaten met elkaar te laten communiceren. Bij deze laatste is van een internetverbinding dan geen sprake.

- **Sensing**

Vooraleer een IoT-apparaat betekenisvol kan zijn voor haar gebruiker, moet het op de een of andere manier informatie uit haar omgeving kunnen oppikken. Denk daarbij aan lichtsensoren, temperatuursensoren of het registeren van een druk op een knop.

- **Informatieverwerking**

Een rekeneenheid, zoals een processor of microcontroller, bepaalt vervolgens wat met die opgevangen informatie gedaan moet worden. Dit is het brein van het apparaat en moet dus geprogrammeerd worden om de juiste functionaliteit te garanderen. Om variabele data en het programma zelf te herbergen, is ook opslagruimte nodig.

- **Actuation**

De rekeneenheid zal uiteindelijk één of meerdere componenten *in beweging* zetten. Dit kan gaan van het aanzetten van een lamp tot het aansturen van een motor.

- **User interface**

Een gebruiker moet op een geschikte manier met een IoT-apparaat kunnen interageren. Een gebruiker kan bijvoorbeeld naar het apparaat toe communiceren aan de hand van een simpele drukknop of een mobiele applicatie. Het apparaat zelf kan dan weer naar de gebruiker communiceren aan de hand van diezelfde applicatie of een flikkerend lichtje.

A.2.2 Arduino en Raspberry Pi

Wanneer enkel afgegaan wordt op het uiterlijk, lijken een Arduino en Raspberry Pi bijzonder goed op elkaar, maar daar houden de gelijkenissen ook op. Een Arduino is een microcontroller. Ze beschikt over een vrij eenvoudige processor, wat opslag en enkele in- en uitgangen (I/O-poorten). Zogeheten *shields* kunnen toegevoegd worden om het aantal I/O-poorten uit te breiden (bijvoorbeeld ethernet- en Wi-Fi-shields). Een Raspberry Pi, daarentegen, kan echt al een *minicomputer* genoemd worden. Haar processor- en opslagcapaciteiten bedragen een veelvoud van die van een Arduino en het standaardmodel beschikt reeds over heel wat meer I/O-poorten dan het Arduino-standaardmodel. Daarnaast bevat een Raspberry Pi nog een grafische processor (GPU), waardoor ze aan een monitor gekoppeld kan worden. Een gebruiker kan vervolgens communiceren met het systeem dankzij een muis, toetsenbord en het Linux-besturingssysteem. Dat besturingssysteem vormt dan de brug tussen de door de gebruiker geschreven programma's en de verschillende componenten die zich op de module bevinden. Bij een Arduino is van een besturingssysteem geen sprake, daar dient de programmacode meteen op module zelf geschreven te worden. Dit betekent ook meteen dat een Arduino slechts één programma kan draaien. (Patnaikuni, 2017)

Aangezien een Raspberry Pi op Linux draait, kunnen allerhande soorten programma's geschreven worden in allerhande soorten programmeertalen. Patnaikuni (2017) somde er enkele op: Python, C, C++, Java ... Het mag dan ook niet verbazen dat er reeds een relatief groot aanbod is van projecten die ook Swift werkende hebben gekregen op Raspberry Pi's¹². De library *SwiftyGPIO*³, in het bijzonder, voorziet explicet in de mogelijkheid om met Swift de verschillende poorten op een Raspberry Pi aan te sturen. Deze library is dus uitermate geschikt voor de ontwikkeling van IoT-apparaten.

In de praktijk wordt echter vaak geopteerd voor Arduino om de waarden van de componenten van een IoT-apparaat te lezen of te schrijven. Een Raspberry Pi wordt dan vaak gebruikt als centrale hub in het uitgebreidere IoT-systeem. Zo zal een Raspberry Pi informatie binnenkrijgen van één of meerdere Arduino's en die informatie vervolgens verwerken, opslaan en beschikbaar maken over het internet. Een mobiele applicatie kan die informatie ten slotte raadplegen en weergeven aan de gebruiker. Uiteraard kan informatie ook in de omgekeerde richting doorgespeeld worden. De communicatie tussen de Arduino en Raspberry Pi zelf gebeurt via de op beide apparaten aanwezige seriële poort. (Roest,

¹<https://swift-arm.com/>

²<https://github.com/uraimo/buildSwiftOnARM>

³<https://github.com/uraimo/SwiftyGPIO>

2020a)

A.2.3 Swift for Arduino

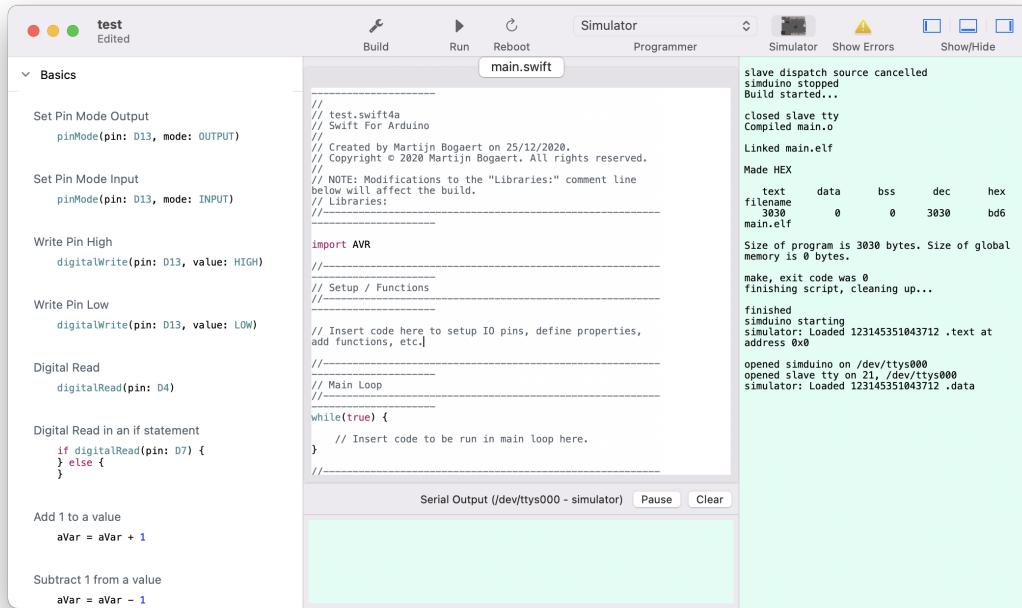
Hoewel deze bachelorproef het gebruik van Swift voor Raspberry Pi's in geen geval links zal laten liggen, zal ze hoofdzakelijk focussen op Arduino-apparaten. Echter, wat het gebruik van Swift voor Arduino betreft, zijn de mogelijkheden minder uitgebreid dan Raspberry Pi. Deze vaststelling is weinig verrassend, aangezien de beschikbare Swift-bibliotheken voor Raspberry Pi in principe niet enkel voor Raspberry Pi-systemen geschreven zijn, maar eerder voor het Linux-besturingssysteem in het algemeen. De use cases van Swift voor Linux gaan met andere woorden veel verder dan enkel het aansturen van Raspberry Pi's, reden genoeg dus voor de ontwikkelaars van de verschillende bibliotheken in kwestie om de technologie werkende te krijgen. Daarnaast zijn de verschillende besturingssystemen van Apple - waarop Swift origineel ontwikkeld werd - en Linux gebouwd op eenzelfde besturingssysteem, Unix.

De werking van een Arduino staat daarmee in schril contrast. Zoals reeds aangehaald, is er bij Arduino immers geen sprake van een besturingssysteem. Een programma dient daarom op de microcontroller zelf - *bare-metal* - geschreven te worden. Dit schrijven of *poorten* wordt normaliter uitgevoerd door de *Arduino IDE*. De Arduino IDE is een computerprogramma waarin Wiring-code geschreven kan worden. Wiring is een bibliotheek, gebouwd op de erg bekende, maar vrij moeilijk te hanteren programmeertaal C++. C++ biedt ontwikkelaars erg veel diepgang in de werking van hun programma's, maar dat betekent meteen ook dat er heel wat verkeerd kan lopen bij het schrijven ervan. Zo zijn bepaalde functies in C++ bijvoorbeeld niet type safe (i.e. wanneer het type van een bepaalde variabele in een functie niet overeenkomt met het type dat die functie verwacht, zal de ontwikkelaar daar niet op gewezen worden tijdens het coderen of bij het builden van het programma - het programma zal tijdens het runnen gewoon crashen) of is het erg moeilijk werken met geheugenruimte. Swift heeft deze problemen niet. (Peto, 2017) (Roest, 2020a)

De Arduino IDE kan daarnaast ook gebruikt worden om de geschreven code om te zetten naar leesbare machinecode voor de processor (*compilen*) en tenslotte te schrijven in het geheugen van het Arduino-apparaat. Om Swift te kunnen gebruiken om Arduino's aan te sturen, moet dus een compiler gemaakt worden die Swift-code kan omzetten naar diezelfde vorm van machinecode. Enkele jaren begon de Brit Carl Peto met dat huzarenstukje. Hij maakte gebruik van de reeds bestaande compilertechnologie, LLVM⁴, en ontwikkelde een Swift-tegenhanger van de Arduino IDE: de *Swift for Arduino*⁵ IDE. Figuur A.1 toont een schermafbeelding van de applicatie en meteen is duidelijk dat centraal plaats is voor de Swift-code. Het linkerpaneel bevat enkele korte *code snippets* die met een simpele sleepbeweging in het centrale paneel geplaatst kunnen worden. Bovenaan staan twee belangrijke knoppen: *Build* en *Run*. Bij een druk op die eerste, wordt de Swift-code gecompileerd naar machinecode die leesbaar is voor een ATmega-processor. Zo'n ATmega-processor is

⁴<https://llvm.org>

⁵<http://swiftforarduino.com/>



Figuur A.1: Swift for Arduino IDE

immers het soort processor dat een Arduino aanstuurt. De tweede knop - *Run* - schrijft de gecompileerde code dan weer in het geheugen van de aangesloten Arduino, waarna die laatste het zonet opgeladen programma meteen begint uit te voeren. Uiteraard is Swift for Arduino nog een vrij jonge technologie en is ze volop in opbouw. De GitHub-pagina⁶ van het project heeft vaak enkele issues openstaan, waaruit blijkt dat veel functionaliteiten nog ingebouwd moeten worden. (Roest, 2020a)

A.3 Methodologie

De bachelorproef zal in de eerste plaats onderzoek doen naar wat de mogelijkheden zijn van de verschillende hierboven vermelde Swift-libraries voor Raspberry Pi, alsook hoe Swift for Arduino precies is opgebouwd. Dit eerste deel zal eerder technisch zijn en probeert antwoorden te zoeken op de vraag tot wat Swift voor IoT in staat is in vergelijking met de gevestigde programmeertalen.

Vervolgens wordt een vergelijkende studie opgezet tussen de mogelijkheden van Swift voor Arduino en Raspberry Pi. De Arduino IDE beschikt over enkele voorbeeldimplementaties waarbij eenvoudige, vaak gebruikte elektronische componenten aangesproken worden. Die voorbeelden kunnen dienen als maatstaven bij die vergelijkende studie. Er wordt onderzocht welke technologie wat wel en niet tot stand kan brengen en welke technologie daar het best in slaagt. Om dit onderzoek tot stand te kunnen brengen, zal uiteraard gebruikt

⁶<https://github.com/swiftforarduino/community>

moeten worden gemaakt van een Arduino- en Raspberry Pi-apparaat. Het is hierbij niet de bedoeling om de Arduino en Raspberry Pi samen te laten werken. Op die manier kan optimaal antwoord gezocht worden op de vraag welke technologie - Arduino of Raspberry Pi - de voorkeur geniet wanneer gewerkt wordt met Swift.

Om na te gaan wat de specifieke mogelijkheden zijn van Swift voor Arduino, wordt tenslotte een proof of concept opgezet. Hierbij zal een volledig IoT-apparaat opgebouwd worden. Concreet wordt een slimme vuilbak ontwikkeld die via een lichtsysteem en iOS-applicatie kan aangeven of haar maximumcapaciteit al dan niet is bereikt. Hiervoor zullen naast een Arduino en de Swift for Arduino IDE ook een gewichtsensor en *NeoPixels*-strip nodig zijn. Dat laatste component is een strip van LED-lichtjes die op allerlei interessante en kleurrijke manieren aangestuurd kunnen worden. Er zal ook geprobeerd worden om aan de hand van een ethernet- en later een Wi-Fi-shield een connectie tot stand te brengen met een server, zodat de bijhorende iOS-applicatie kan communiceren met de Arduino. Dit proof of concept zal illustreren wat de specifieke mogelijkheden zijn van Swift voor Arduino.

Omdat niet gegarandeerd is dat Swift for Arduino reeds overweg kan met een Wi-Fi-shield - laat staan met een ethernet-shield - zal ook getracht worden eenzelfde functionaliteit tot stand te brengen door een Raspberry Pi aan de opstelling toe te voegen. De Arduino en Raspberry Pi communiceren onderling via hun seriële poorten en het is de Raspberry Pi die dan de verdere internetcommunicatie verzorgt. De data die de gewichtsensor van de Arduino leest, wordt meteen doorgestuurd naar de Raspberry Pi, die de data vervolgens uitpakt en opslaat in een SQLite databank. Daarnaast draait er op de Raspberry Pi ook een backendsysteem dankzij *Vapor*⁷. Vapor is een bibliotheek die ontwikkelaars de mogelijkheid geeft via Swift een SQLite databank te lezen, de opgehaalde data al dan niet te hergroeperen en tenslotte beschikbaar te stellen via een API. In het hier beschreven IoT-project zal een iOS-applicatie die API uiteindelijk gebruiken om de gelezen data van de gewichtsensor tot in de hand van de gebruiker te brengen. Deze opstelling waarbij Arduino en Raspberry Pi samenwerken, is gebaseerd op een gelijkaardige opstelling die werd voorgesteld door Roest (2020a). Doel is om alle hierboven beschreven functionaliteiten te implementeren door enkel en alleen gebruik te maken van Swift.

A.4 Verwachte resultaten

Het heeft er alle schijn naar dat de mogelijkheden om IoT-apparaten te ontwikkelen aan de hand van Swift, veelal beperkt zijn in vergelijking met de programmeertalen die gewoonlijk gebruikt worden voor dergelijke use cases. Volgende kanttekening moet wel gemaakt worden: vooral de mogelijkheden van de Swift for Arduino IDE zijn hoogstwaarschijnlijk een stuk geringer dan die van haar tegenhanger, de originele Arduino IDE. Wat Raspberry Pi betreft, ziet het er wel naar uit dat de beschikbare Swift-libraries het gros van de functionaliteiten aangeboden door de meer mature Python-, Java- en C-libraries, ook kan voorzien, zij het met enkele beperkingen wegens het minder ruime aanbod van bibliotheken.

⁷<https://vapor.codes>

Deze hypothese zal meteen ook invloed hebben op de vergelijkende studie tussen Swift voor Raspberry Pi en Arduino. Aangezien een Raspberry Pi in wezen een minicomputer is, zijn haar mogelijkheden sowieso reeds heel wat uitgebreider dan die van een Arduino. Toch wordt voor de ontwikkeling van IoT-apparaten vaak naar de Arduino gegrepen wegens haar simpelheid. Een Raspberry Pi gebruiken om enkele simpele componenten te lezen en aan te sturen, is vaak *overkill*. Toch lijkt de kans groot dat Swift for Arduino een nog te jonge technologie blijkt en de standaardvereisten van een IoT-apparaat - zoals een directe verbinding met het internet maken - (nog) niet kan vervullen. De verschillende Swift-libraries voor Raspberry Pi, daarentegen, slagen daar waarschijnlijk wel in.

Wat de proof of concept tenslotte betreft, werd in voorgaande paragraaf, alsook in voorstaand hoofdstuk, reeds gewag gemaakt dat de kans bestaat dat een Arduino op zich niet voldoende is om een internetverbinding tot stand te brengen wanneer Swift gehanteerd wordt. Het ziet er dan ook uit dat een Raspberry Pi aan de opstelling toegevoegd zal moeten worden. Het is immers zo goed als zeker dat de combinatie Arduino-Raspberry Pi wel volstaat om de basisfunctionaliteiten van een IoT-apparaat uit te voeren, in het geval alle nodige code in Swift geschreven wordt.

A.5 Verwachte conclusies

Het antwoord op de vraag tot wat Swift voor IoT in staat is in vergelijking met de gevestigde programmeertalen, zal tweeledig zijn. Er is immers een groot verschil tussen de mogelijkheden van Swift for Arduino en de mogelijkheden waarin de verschillende Swift-libraries voor Raspberry Pi voorzien. Het lijkt erop dat Swift for Arduino nog een lange weg af te leggen heeft om enigszins te kunnen tippen aan de functionaliteiten van de Arduino IDE en de vele bibliotheken die doorheen de jaren werden gebouwd en beschikbaar gesteld door de uitgebreide Arduino-community. Swift voor Raspberry Pi, daarentegen, wekt minder de indruk dat er gevoelig minder functionaliteiten beschikbaar zijn.

Ook wanneer Arduino en Raspberry Pi regelrecht tegenover elkaar worden gezet, heeft het er alle schijn naar dat Raspberry Pi als *winnaar* uit de bus zal komen. De argumentatie hierachter is precies dezelfde als de hierboven beschreven argumentatie. Wanneer enkel het aansturen en lezen van elektronische componenten beschouwd wordt, lijkt het erop dat beide apparaten tot hetzelfde in staat zijn. Er zou zelfs gesteld kunnen worden dat Arduino de voorkeur hier wegdraagt. Arduino werd in wezen immers gebouwd om te voldoen aan deze functionaliteiten en is daarom ook eenvoudiger om mee te werken, ook wanneer gekozen wordt voor Swift for Arduino. Aangezien deze bachelorproef echter draait rond IoT-apparaten, ziet het er toch naar uit dat de Raspberry Pi uiteindelijk aan het langste eind zal trekken. Het grote probleem met Swift for Arduino is immers dat het tot stand brengen van een internetverbinding niet gegarandeerd is. De Swift-libraries van Raspberry Pi lijken daar wel mee overweg te kunnen.

Het al dan niet slagen van de proof of concept zal uiteindelijk antwoord moeten geven op de

hoofdvraag. Is het mogelijk Swift te gebruiken voor de aansturing van IoT-apparaten? Het antwoord hierop zal hoogstwaarschijnlijk *ja* zijn. Echter, het scala aan mogelijkheden om dat doel te bereiken, lijkt heel wat minder uitgebreid dan wanneer geen veto wordt gesteld op welke programmeerta(al)en gehanteerd moet(en) worden. Een IoT-apparaat ontwikkelen in Swift door enkel en alleen gebruik te maken van Arduino, lijkt niet realistisch. Het ziet er immers niet naar uit dat Swift for Arduino op zichzelf een internetverbinding tot stand kan brengen. Wanneer een Raspberry Pi de internetfunctionaliteiten echter op zich neemt, is het verhaal anders. In dat laatste geval lijkt het wel degelijk mogelijk om Swift te gebruiken voor de aansturing van IoT-apparaten.

B. Code slimme prullenbak

B.1 Weegschaal

Alle codebestanden in deze sectie vormen samen een *.swift4a*-bestand. Een *.swift4a*-bestand is een soort archief dat de Swift for Arduino IDE kan lezen. Het project in kwestie verwacht dat een geïntegreerd circuit van het type *HX711_ADC* aan de microcontroller gekoppeld is. Bij dit bordje hoort ook een library¹ dat gemaakt werd met het oog op de Arduino IDE. Om ze bruikbaar te maken in de Swift for Arduino IDE, zijn enkele van die originele bestanden echter lichtjes gewijzigd. Tabel B.1 geeft een overzicht van de herkomst van de verschillende codebestanden die verder in deze sectie uitgeschreven zijn.

B.1.1 Startpunt

```
1 // Libraries:  
2 import AVR  
3  
4 let calVal_eepromAddress: UInt16 = 0  
5  
6 SetupSerial()  
7  
8 print("")  
9 print("Starting...")  
10
```

¹https://github.com/olkal/HX711_ADC

Herkomst	Bestanden
Projectspecifiek	main.swift main.h
HX711_ADC library (mits aanpassingen)	HX711_ADC.cpp
HX711_ADC library	HX711_ADC library.h config.h
Swift for Arduino community (mits aanpassingen)	shims.h

Tabel B.1: Herkomst codebestanden weegschaal

```

11 if setupLoadCell() {
12     print("Startup is complete")
13 } else {
14     print("Timeout, check MCU>HX711 wiring and pin
15     designations")
16     while true {}
17 }
18 calibrate() //start calibration procedure
19
20 while true {
21     var newDataReady = false
22
23     // check for new data/start next conversion:
24     if updateLoadCell() != 0 {
25         newDataReady = true
26     }
27
28     // get smoothed value from the dataset:
29     if newDataReady {
30         let i: Float = getDataLoadCell()
31         print(staticString: "Load_cell output val: ",
32               addNewline: false)
33         print(i)
34         newDataReady = false
35         delay(milliseconds: 100)
36     }

```

```
37     // receive command from serial terminal
38     if available() {
39         let inByte = read()
40         print("")
41         if inByte == 0x74 { //'t'
42             tareNoDelayLoadCell() //tare
43         } else if inByte == 0x72 { //'r'
44             calibrate() //calibrate
45         } else if inByte == 0x63 { //'c'
46             changeSavedCalFactor() //edit calibration
47             value manually
48         }
49     }
50     // check if last tare operation is complete
51     if getTareStatusLoadCell() {
52         print("Tare complete")
53     }
54 }
55
56 func calibrate() {
57     print("/**/")
58     print("Start calibration:")
59     print("Place the load cell an a level stable surface.
")
60     print("Remove any load applied to the load cell.")
61     print("Send t from serial monitor to set the tare
offset.")
62
63     var _resume = false
64     while !_resume {
65         updateLoadCell()
66         if available() {
67             let inByte = read()
68             print("")
69             if inByte == 0x74 { //'t'
70                 tareNoDelayLoadCell()
71             }
72         }
73         if getTareStatusLoadCell() {
74             print("Tare complete")
75             _resume = true
76         }
77     }
78
79     print("/**/")
```

```
80     print("Now, place your known mass on the loadcell.")
81     print("Then send the weight of this mass (i.e.
82           1000.00) from serial monitor. Send the digits one by
83           one: first the thousand, then the hundred, then the
84           ten, then the unit, then the tenth, then the hundredth
85           .")
86
87
88     let known_mass: Float = getMultiDigitFloatFromSerial()
89
90     print("Known mass is")
91     print(known_mass)
92
93     refreshDataSetLoadCell() //refresh the dataset to be
94         sure that the known mass is measured correct
95     let newCalibrationValue: Float =
96         getNewCalibrationLoadCell(known_mass) //get the new
97         calibration value
98
99     print(staticString: "New calibration value has been
100        set to: ", addNewline: false)
101    print(newCalibrationValue, addNewline: false)
102    print(", use this as calibration value (calFactor) in
103        your project sketch.")
104
105    saveValueInEEPROM(newCalibrationValue)
106
107    print("End calibration")
108    print("***")
109    print("To re-calibrate, send r from serial monitor.")
110    print("For manual edit of the calibration value, send
111        c from serial monitor.")
112    print("***")
113
114 func changeSavedCalFactor() {
115     let oldCalibrationValue: Float = getCalFactorLoadCell()
116
117     print("***")
118     print(staticString: "Current value is: ", addNewline:
119           false)
120     print(oldCalibrationValue)
121     print("Now, send the new value from serial monitor, i
122           .e. 696.0")
```

```
112     let newCalibrationValue: Float =
113         getMultiDigitFloatFromSerial()
114
115         print(staticString: "New calibration value is: ",
116             addNewline: false)
116         print(newCalibrationValue)
117         setCalFactorLoadCell(newCalibrationValue)
118
118         saveValueInEEPROM(newCalibrationValue)
119
120         print("End change calibration value")
121         print("***")
122     }
123
124 func getMultiDigitFloatFromSerial() -> Float {
125     var amount: Int = 6
126     var numbers = [Float](repeating: 0.0, count: &amount)
127     var power: Int8 = 3
128
129     for i in 0..<numbers.count {
130         print(staticString: "Digit ", addNewline: false)
131         print(i + 1, addNewline: false)
132         print(staticString: ":", addNewline: false)
133         var _resume = false
134         while !_resume {
135             if available() {
136                 updateLoadCell()
137                 let char = read()
138                 print("")
139                 if char >= 48 && char <= 57 {
140                     numbers[i] = Float(char - 48) *
141                         powerOfTen(power)
142                         power = power - 1
143                         _resume = true
144                     }
145                 }
146             }
147
148     var result: Float = 0.0
149     for i in numbers {
150         result += i
151     }
152
153     numbers.deallocate()
154     return result
```

```
155 }
156
157 // Doesn't work since S4A can only write UInt8 types to
158 // EEPROM
159 func saveValueInEEPROM(_ value: Float) {
160     print(staticString: "Save this value to EEPROM
161 address ", addNewline: false)
162     print(calVal_eepromAddress, addNewline: false)
163     print("? y/n")
164
165     var _resume = false
166     while !_resume {
167         if available() {
168             let inByte = read()
169             print("")
170             if inByte == 0x79 { // 'y'
171                 writeEEPROM(address: calVal_eepromAddress
172 , value: UInt8(value))
173
174                 print(staticString: "Value ", addNewline:
175 false)
176                 print(value, addNewline: false)
177                 print(staticString: " saved to EEPROM
178 address ", addNewline: false)
179                 print(calVal_eepromAddress)
180                 _resume = true
181             } else if inByte == 0x6E { // 'n'
182                 print("Value not saved to EEPROM")
183                 _resume = true
184             }
185         }
186     }
187
188     func powerOfTen(_ power: Int8) -> Float {
189         var result: Float = 1
190         if power > 0 {
191             for _ in 0..<power {
192                 result *= 10.0
193             }
194             return result
195         }
196         if power < 0 {
197             for _ in power..<0 {
198                 result /= 10.0
199             }
200         }
201     }
202 }
```

```
196     }
197     return result
198 }
```

Codevoorbeeld B.1: Weegschaal - *main.swift*

B.1.2 HX711_ADC library

```
1  /*
2  -----
3  HX711_ADC
4  Arduino library for HX711 24-Bit Analog-to-Digital
5  Converter for Weight Scales
6  Olav Kallhovd sept2017
7  -----
8
9 // #include <Arduino.h>
10 #include <stdint.h>
11 #include "shims.h"
12 #include "HX711_ADC.h"
13
14 // standard pins:
15 const int HX711_dout = 4; //mcu > HX711 dout pin
16 const int HX711_sck = 5; //mcu > HX711 sck pin
17
18 // HX711 constructor:
19 HX711_ADC LoadCell(HX711_dout, HX711_sck);
20
21 extern "C" {
22     bool setupLoadCell() {
23         LoadCell.begin();
24         unsigned long stabilizingtime = 2000; //
precision right after power-up can be improved by
adding a few seconds of stabilizing time
25         boolean _tare = true; //set this to false if you
don't want tare to be performed in the next step
26         LoadCell.start(stabilizingtime, _tare);
27         if (LoadCell.getTareTimeoutFlag() || LoadCell.
getSignalTimeoutFlag()) {
28             return false;
29         }
30         else {
31             LoadCell.setCalFactor(1.0); // user set
```

```

    calibration value (float), initial value 1.0 may be
    used for this sketch
32     }
33     while (!LoadCell.update());
34     return true;
35 }
36 uint8_t updateLoadCell() {
37     return LoadCell.update();
38 }
39 void tareNoDelayLoadCell() {
40     LoadCell.tareNoDelay();
41 }
42 bool getTareStatusLoadCell() {
43     return LoadCell.getTareStatus();
44 }
45 bool refreshDataSetLoadCell() {
46     return LoadCell.refreshDataSet();
47 }
48 float getNewCalibrationLoadCell(float known_mass) {
49     return LoadCell.getNewCalibration(known_mass);
50 }
51 float getDataLoadCell() {
52     return LoadCell.getData();
53 }
54 float getCalFactorLoadCell() {
55     return LoadCell.getCalFactor();
56 }
57 void setCalFactorLoadCell(float cal) {
58     LoadCell.setCalFactor(cal);
59 }
60 }
61
62 HX711_ADC::HX711_ADC(uint8_t dout, uint8_t sck) // constructor
63 {
64     doutPin = dout;
65     sckPin = sck;
66 }
67
68 void HX711_ADC::setGain(uint8_t gain) //value should be
69     32, 64 or 128*
70 {
71     if(gain < 64) GAIN = 2; //32, channel B
72     else if(gain < 128) GAIN = 3; //64, channel A
73     else GAIN = 1; //128, channel A

```

```
74
75 //set pinMode, HX711 gain and power up the HX711
76 void HX711_ADC::begin()
77 {
78     pinMode(sckPin, OUTPUT);
79     pinMode(doutPin, INPUT);
80     setGain(128);
81     powerUp();
82 }
83
84 //set pinMode, HX711 selected gain and power up the HX711
85 void HX711_ADC::begin(uint8_t gain)
86 {
87     pinMode(sckPin, OUTPUT);
88     pinMode(doutPin, INPUT);
89     setGain(gain);
90     powerUp();
91 }
92
93 /* start(t):
94 *      will do conversions continuously for 't' +400
95 *      milliseconds (400ms is min. settling time at 10SPS).
96 *      Running this for 1-5s in setup() - before tare()
97 *      seems to improve the tare accuracy */
98 void HX711_ADC::start(unsigned long t)
99 {
100     t += 400;
101     lastDoutLowTime = millis();
102     while(millis() < t)
103     {
104         update();
105         yield();
106     }
107     tare();
108     tareStatus = 0;
109 }
110
111 /* start(t, dotare) with selectable tare:
112 *      will do conversions continuously for 't' +400
113 *      milliseconds (400ms is min. settling time at 10SPS).
114 *      Running this for 1-5s in setup() - before tare()
115 *      seems to improve the tare accuracy. */
116 void HX711_ADC::start(unsigned long t, bool dotare)
117 {
118     t += 400;
119     lastDoutLowTime = millis();
```

```
116     while(millis() < t)
117     {
118         update();
119         yield();
120     }
121     if (dotare)
122     {
123         tare();
124         tareStatus = 0;
125     }
126 }
127
128 /* startMultiple(t): use this if you have more than one
   load cell and you want to do tare and stabilization
   simultaneously.
129 * Will do conversions continuously for 't' +400
   milliseconds (400ms is min. settling time at 10SPS).
130 * Running this for 1-5s in setup() - before tare()
   seems to improve the tare accuracy */
131 int HX711_ADC::startMultiple(unsigned long t)
132 {
133     tareTimeoutFlag = 0;
134     lastDoutLowTime = millis();
135     if(startStatus == 0) {
136         if(isFirst) {
137             startMultipleTimeStamp = millis();
138             if (t < 400)
139             {
140                 startMultipleWaitTime = t + 400; //min
   time for HX711 to be stable
141             }
142             else
143             {
144                 startMultipleWaitTime = t;
145             }
146             isFirst = 0;
147         }
148         if((millis() - startMultipleTimeStamp) <
   startMultipleWaitTime) {
149             update(); //do conversions during
   stabilization time
150             yield();
151             return 0;
152         }
153         else { //do tare after stabilization time is up
154             static unsigned long timeout = millis() +
```

```
tareTimeOut;
155         doTare = 1;
156         update();
157         if(convRslt == 2)
158     {
159             doTare = 0;
160             convRslt = 0;
161             startStatus = 1;
162         }
163         if (!tareTimeoutDisable)
164     {
165             if (millis() > timeout)
166             {
167                 tareTimeoutFlag = 1;
168                 return 1; // Prevent endless loop if no
HX711 is connected
169             }
170         }
171     }
172 }
173 return startStatus;
174 }

175

176 /* startMultiple(t, dotare) with selectable tare:
177 *   use this if you have more than one load cell and you
178 *   want to (do tare and) stabilization simultaneously.
179 *   Will do conversions continuously for 't' +400
180 *   milliseconds (400ms is min. settling time at 10SPS).
181 *   Running this for 1-5s in setup() - before tare()
182 *   seems to improve the tare accuracy */
183 int HX711_ADC::startMultiple(unsigned long t, bool dotare
)
184 {
185     tareTimeoutFlag = 0;
186     lastDoutLowTime = millis();
187     if(startStatus == 0) {
188         if(isFirst) {
189             startMultipleTimeStamp = millis();
190             if (t < 400)
191             {
192                 startMultipleWaitTime = t + 400; //min
193                 time for HX711 to be stable
194             }
195             else
196             {
197                 startMultipleWaitTime = t;
```

```
194         }
195         isFirst = 0;
196     }
197     if((millis() - startMultipleTimeStamp) <
198         startMultipleWaitTime) {
199         update(); //do conversions during
200         stabilization time
201         yield();
202         return 0;
203     }
204     else { //do tare after stabilization time is up
205         if (dotare)
206         {
207             static unsigned long timeout = millis() +
208             tareTimeout;
209             doTare = 1;
210             update();
211             if(convRslt == 2)
212             {
213                 doTare = 0;
214                 convRslt = 0;
215                 startStatus = 1;
216             }
217             if (!tareTimeoutDisable)
218             {
219                 if (millis() > timeout)
220                 {
221                     tareTimeoutFlag = 1;
222                     return 1; // Prevent endless loop if
223                     no HX711 is connected
224                 }
225             }
226             else return 1;
227         }
228     }
229     return startStatus;
230 }
231 //zero the scale, wait for tare to finnish (blocking)
232 void HX711_ADC::tare()
233 {
234     uint8_t rdy = 0;
235     doTare = 1;
236     tareTimes = 0;
237     tareTimeoutFlag = 0;
```

```
236     unsigned long timeout = millis() + tareTimeout;
237     while(rdy != 2)
238     {
239         rdy = update();
240         if (!tareTimeoutDisable)
241         {
242             if (millis() > timeout)
243             {
244                 tareTimeoutFlag = 1;
245                 break; // Prevent endless loop if no
HX711 is connected
246             }
247         }
248         yield();
249     }
250 }
251
252 //zero the scale, initiate the tare operation to run in
//the background (non-blocking)
253 void HX711_ADC::tareNoDelay()
254 {
255     doTare = 1;
256     tareTimes = 0;
257 }
258
259 //set new calibration factor, raw data is divided by this
//value to convert to readable data
260 void HX711_ADC::setCalFactor(float cal)
261 {
262     calFactor = cal;
263     calFactorRecip = 1/calFactor;
264 }
265
266 //returns 'true' if tareNoDelay() operation is complete
267 bool HX711_ADC::getTareStatus()
268 {
269     bool t = tareStatus;
270     tareStatus = 0;
271     return t;
272 }
273
274 //returns the current calibration factor
275 float HX711_ADC::getCalFactor()
276 {
277     return calFactor;
278 }
```

```
279
280 //call the function update() in loop or from ISR
281 //if conversion is ready; read out 24 bit data and add to
282 //dataset, returns 1
283 //if tare operation is complete, returns 2
284 //else returns 0
285 uint8_t HX711_ADC::update()
286 {
287     byte dout = digitalRead(doutPin); //check if
288     //conversion is ready
289     if (!dout)
290     {
291         conversion24bit();
292         lastDoutLowTime = millis();
293         signalTimeoutFlag = 0;
294     }
295     else
296     {
297         //if (millis() > (lastDoutLowTime +
298         SIGNAL_TIMEOUT))
299         if (millis() - lastDoutLowTime > SIGNAL_TIMEOUT)
300         {
301             signalTimeoutFlag = 1;
302         }
303         convRslt = 0;
304     }
305     return convRslt;
306 }
307
308 float HX711_ADC::getData() // return fresh data from the
309 //moving average dataset
310 {
311     long data = 0;
312     lastSmoothedData = smoothedData();
313     data = lastSmoothedData - tareOffset ;
314     float x = (float)data * calFactorRecip;
315     return x;
316 }
317
318 long HX711_ADC::smoothedData()
319 {
320     long data = 0;
321     long L = 0xFFFFFFF;
322     long H = 0x00;
323     for (uint8_t r = 0; r < (samplesInUse +
324     IGN_HIGH_SAMPLE + IGN_LOW_SAMPLE); r++)
```

```
320      {
321          #if IGN_LOW_SAMPLE
322              if (L > dataSampleSet[r]) L = dataSampleSet[r];
323              // find lowest value
324          #endif
325          #if IGN_HIGH_SAMPLE
326              if (H < dataSampleSet[r]) H = dataSampleSet[r];
327              // find highest value
328          #endif
329          data += dataSampleSet[r];
330      }
331      #if IGN_LOW_SAMPLE
332          data -= L; //remove lowest value
333      #endif
334      #if IGN_HIGH_SAMPLE
335          data -= H; //remove highest value
336      #endif
337      //return data;
338      return (data >> divBit);
339
340 void HX711_ADC::conversion24bit() //read 24 bit data,
341     store in dataset and start the next conversion
342 {
343     conversionTime = micros() - conversionStartTime;
344     conversionStartTime = micros();
345     unsigned long data = 0;
346     uint8_t dout;
347     convRslt = 0;
348     if(SCK_DISABLE_INTERRUPTS) noInterrupts();
349     for (uint8_t i = 0; i < (24 + GAIN); i++)
350     { //read 24 bit data + set gain and start next
351         conversion
352         if(SCK_DELAY) delayMicroseconds(3); // could be
353         required for faster mcu's, set value in config.h
354         digitalWrite(sckPin, 1);
355         if(SCK_DELAY) delayMicroseconds(3); // could be
356         required for faster mcu's, set value in config.h
357         digitalWrite(sckPin, 0);
358         if (i < (24))
359         {
360             dout = digitalRead(doutPin);
361             data = (data << 1) | dout;
362         }
363     }
```

```
360     if(SCK_DISABLE_INTERRUPTS) interrupts();
361     /*
362      The HX711 output range is min. 0x800000 and max. 0
363      x7FFFFFF (the value rolls over).
364      In order to convert the range to min. 0x000000 and
365      max. 0xFFFFFFFF,
366      the 24th bit must be changed from 0 to 1 or from 1 to
367      0.
368     */
369     data = data ^ 0x800000; // flip the 24th bit
370
371     if (data > 0xFFFFFFF)
372     {
373         dataOutOfRange = 1;
374         //Serial.println("dataOutOfRange");
375     }
376     if (readIndex == samplesInUse + IGN_HIGH_SAMPLE +
377         IGN_LOW_SAMPLE - 1)
378     {
379         readIndex = 0;
380     }
381     if(data > 0)
382     {
383         convRslt++;
384         dataSampleSet[readIndex] = (long)data;
385         if(doTare)
386         {
387             if (tareTimes < DATA_SET)
388             {
389                 tareTimes++;
390             }
391             else
392             {
393                 tareOffset = smoothedData();
394                 tareTimes = 0;
395                 doTare = 0;
396                 tareStatus = 1;
397                 convRslt++;
398             }
399         }
400     }
401 }
```

```
402
403 //power down the HX711
404 void HX711_ADC::powerDown()
405 {
406     digitalWrite(sckPin, LOW);
407     digitalWrite(sckPin, HIGH);
408 }
409
410 //power up the HX711
411 void HX711_ADC::powerUp()
412 {
413     digitalWrite(sckPin, LOW);
414 }
415
416 //get the tare offset (raw data value output without the
417 //scale "calFactor")
418 long HX711_ADC::getTareOffset()
419 {
420     return tareOffset;
421 }
422
423 //set new tare offset (raw data value input without the
424 //scale "calFactor")
425 void HX711_ADC::setTareOffset(long newoffset)
426 {
427     tareOffset = newoffset;
428 }
429
430 //for testing and debugging:
431 //returns current value of dataset readIndex
432 int HX711_ADC::getReadIndex()
433 {
434     return readIndex;
435 }
436
437 //for testing and debugging:
438 //returns latest conversion time in millis
439 float HX711_ADC::getConversionTime()
440 {
441     return conversionTime/1000.0;
442 }
443
444 //for testing and debugging:
445 //returns the HX711 conversions ea seconds based on the
446 //latest conversion time.
447
448 //The HX711 can be set to 10SPS or 80SPS. For general use
```

```
    the recommended setting is 10SPS.  
445 float HX711_ADC::getSPS()  
446 {  
447     float sps = 1000000.0/conversionTime;  
448     return sps;  
449 }  
450  
451 //for testing and debugging:  
452 //returns the tare timeout flag from the last tare  
    operation.  
453 //0 = no timeout, 1 = timeout  
454 bool HX711_ADC::getTareTimeoutFlag()  
455 {  
456     return tareTimeoutFlag;  
457 }  
458  
459 void HX711_ADC::disableTareTimeout()  
460 {  
461     tareTimeoutDisable = 1;  
462 }  
463  
464 long HX711_ADC::getSettlingTime()  
465 {  
466     long st = getConversionTime() * DATA_SET;  
467     return st;  
468 }  
469  
470 //override the number of samples in use  
471 //value is rounded down to the nearest valid value  
472 void HX711_ADC::setSamplesInUse(int samples)  
473 {  
474     int old_value = samplesInUse;  
475  
476     if(samples <= SAMPLES)  
477     {  
478         if(samples == 0) //reset to the original value  
479         {  
480             divBit = divBitCompiled;  
481         }  
482         else  
483         {  
484             samples >>= 1;  
485             for(divBit = 0; samples != 0; samples >>= 1,  
        divBit++);  
486         }  
487         samplesInUse = 1 << divBit;
```

```
488
489         //replace the value of all samples in use with
490         //the last conversion value
490         if(samplesInUse != old_value)
491         {
492             for (uint8_t r = 0; r < samplesInUse +
493                 IGN_HIGH_SAMPLE + IGN_LOW_SAMPLE; r++)
493             {
494                 dataSampleSet[r] = lastSmoothedData;
495             }
496             readIndex = 0;
497         }
498     }
499 }
500
501 //returns the current number of samples in use.
502 int HX711_ADC::getSamplesInUse()
503 {
504     return samplesInUse;
505 }
506
507 //resets index for dataset
508 void HX711_ADC::resetSamplesIndex()
509 {
510     readIndex = 0;
511 }
512
513 //Fill the whole dataset up with new conversions, i.e.
514 //after a reset/restart (this function is blocking once
515 //started)
516 bool HX711_ADC::refreshDataSet()
517 {
518     int s = getSamplesInUse() + IGN_HIGH_SAMPLE +
519             IGN_LOW_SAMPLE; // get number of samples in dataset
520     resetSamplesIndex();
521     while ( s > 0 ) {
522         update();
523         yield();
524         if (digitalRead(doutPin) == LOW) { // HX711 dout
525             pin is pulled low when a new conversion is ready
526             getData(); // add data to the set and start
527             next conversion
528             s--;
529         }
530     }
531     return true;
```

```

527 }
528
529 //returns 'true' when the whole dataset has been filled
530 // up with conversions, i.e. after a reset/restart.
530 bool HX711_ADC::getDataSetStatus()
531 {
532     bool i = false;
533     if (readIndex == samplesInUse + IGN_HIGH_SAMPLE +
533         IGN_LOW_SAMPLE - 1)
534     {
535         i = true;
536     }
537     return i;
538 }
539
540 //returns and sets a new calibration value (calFactor)
541 // based on a known mass input
541 float HX711_ADC::getNewCalibration(float known_mass)
542 {
543     float readValue = getData();
544     float exist_calFactor = getCalFactor();
545     float new_calFactor;
546     new_calFactor = (readValue * exist_calFactor) /
547     known_mass;
547     setCalFactor(new_calFactor);
548     return new_calFactor;
549 }
550
551 //returns 'true' if it takes longer time then '
552 // 'SIGNAL_TIMEOUT' for the dout pin to go low after a new
553 // conversion is started
552 bool HX711_ADC::getSignalTimeoutFlag()
553 {
554     return signalTimeoutFlag;
555 }
```

Codevoorbeeld B.2: Weegschaal - *HX711_ADC.cpp*

```

1 /*
2 -----
3 HX711_ADC
4 Arduino library for HX711 24-Bit Analog-to-Digital
5 Converter for Weight Scales
6 Olav Kallhovd sept2017
7 -----
```

```
7  */
8
9 #ifndef HX711_ADC_h
10 #define HX711_ADC_h
11
12 // #include <Arduino.h>
13 #include "shims.h"
14 #include "config.h"
15
16 /*
17 Note: HX711_ADC configuration values has been moved to
18     file config.h
19 */
20 #define DATA_SET      SAMPLES + IGN_HIGH_SAMPLE +
21           IGN_LOW_SAMPLE // total samples in memory
22
23 #if (SAMPLES != 1) & (SAMPLES != 2) & (SAMPLES != 4) &
24   (SAMPLES != 8) & (SAMPLES != 16) & (SAMPLES != 32)
25   & (SAMPLES != 64) & (SAMPLES != 128)
26   #error "number of SAMPLES not valid!"
27 #endif
28
29
30 #if          (SAMPLES == 1)
31 #define      DIVB 0
32 #elif        (SAMPLES == 2)
33 #define      DIVB 1
34 #elif        (SAMPLES == 4)
35 #define      DIVB 2
36 #elif        (SAMPLES == 8)
37 #define      DIVB 3
38 #elif        (SAMPLES == 16)
39 #define      DIVB 4
40 #elif        (SAMPLES == 32)
41 #define      DIVB 5
42 #elif        (SAMPLES == 64)
43 #define      DIVB 6
44 #elif        (SAMPLES == 128)
45 #define      DIVB 7
46 #endif
47
```

```

48 #define SIGNAL_TIMEOUT      100
49
50 class HX711_ADC
51 {
52
53     public:
54         HX711_ADC(uint8_t dout, uint8_t sck);           // constructor
55         void setGain(uint8_t gain = 128);               // value must be 32, 64 or 128*
56         void begin();                                  // set pinMode, HX711 gain and power up the HX711
57         void begin(uint8_t gain);                     // set pinMode, HX711 selected gain and power up the HX711
58         void start(unsigned long t);                  // start HX711 and do tare
59         void start(unsigned long t, bool dotare);    // start HX711, do tare if selected
60         int startMultiple(unsigned long t);          // start and do tare, multiple HX711 simultaneously
61         int startMultiple(unsigned long t, bool dotare); // start and do tare if selected, multiple HX711 simultaneously
62         void tare();                                // zero the scale, wait for tare to finish (blocking)
63         void tareNoDelay();                         // zero the scale, initiate the tare operation to run in the background (non-blocking)
64         bool getTareStatus();                      // returns 'true' if tareNoDelay() operation is complete
65         void setCalFactor(float cal);              // set new calibration factor, raw data is divided by this value to convert to readable data
66         float getCalFactor();                     // returns the current calibration factor
67         float getData();                          // returns data from the moving average dataset
68         int getReadIndex();                     // for testing and debugging
69         float getConversionTime();            // for testing and debugging
70         float getSPS();                        // for testing and debugging
71         bool getTareTimeoutFlag();             // for testing and debugging

```

```
72     void disableTareTimeout(); //  
73     for testing and debugging  
74         long getSettlingTime(); //  
75     for testing and debugging  
76         void powerDown(); //  
77     power down the HX711  
78         void powerUp(); //  
79     power up the HX711  
80         long getTareOffset(); //  
81     get the tare offset (raw data value output without the  
82     scale "calFactor")  
83         void setTareOffset(long newoffset); //  
84     set new tare offset (raw data value input without the  
85     scale "calFactor")  
86         uint8_t update(); //  
87     if conversion is ready; read out 24 bit data and add  
88     to dataset  
89         void setSamplesInUse(int samples); //  
90     override number of samples in use  
91         int getSamplesInUse(); //  
92     returns current number of samples in use  
93         void resetSamplesIndex(); //  
94     resets index for dataset  
95         bool refreshDataSet(); //  
96     Fill the whole dataset up with new conversions, i.e.  
97     after a reset/restart (this function is blocking once  
98     started)  
99         bool getDataSetStatus(); //  
100    returns 'true' when the whole dataset has been filled  
101    up with conversions, i.e. after a reset/restart  
102        float getNewCalibration(float known_mass); //  
103    returns and sets a new calibration value (calFactor)  
104    based on a known mass input  
105        bool getSignalTimeoutFlag(); //  
106    returns 'true' if it takes longer time then '  
107    SIGNAL_TIMEOUT' for the dout pin to go low after a new  
108    conversion is started  
109  
110    protected:  
111        void conversion24bit(); //if  
112    conversion is ready: returns 24 bit data and starts  
113    the next conversion  
114        long smoothedData(); //  
115    returns the smoothed data value calculated from the  
116    dataset  
117        uint8_t sckPin; //
```

```

1  /*
91   HX711 pd_sck pin                                // 
92   HX711 dout pin                                 // 
93   HX711 GAIN                                     // 
94   float calFactor = 1.0;                          // 
95   calibration factor as given in function setCalFactor(
96   float cal)
97   float calFactorRecip = 1.0;                     // 
98   reciprocal calibration factor (1/calFactor), the HX711
99   raw data is multiplied by this value
100  volatile long dataSampleSet[DATA_SET + 1];       // 
101  dataset, make volatile if interrupt is used
102  long tareOffset;
103  int readIndex = 0;
104  unsigned long conversionStartTime;
105  unsigned long conversionTime;
106  uint8_t isFirst = 1;
107  uint8_t tareTimes;
108  uint8_t divBit = DIVB;
109  const uint8_t divBitCompiled = DIVB;
110  bool doTare;
111  bool startStatus;
112  unsigned long startMultipleTimeStamp;
113  unsigned long startMultipleWaitTime;
114  uint8_t convRslt;
115  bool tareStatus;
116  unsigned int tareTimeOut = (SAMPLES +
117  IGN_HIGH_SAMPLE + IGN_HIGH_SAMPLE) * 150; // tare
118  timeout time in ms, no of samples * 150ms (10SPS + 50%
119  margin)
120  margin)
121  bool tareTimeoutFlag;
122  bool tareTimeoutDisable = 0;
123  int samplesInUse = SAMPLES;
124  long lastSmoothedData = 0;
125  bool dataOutOfRange = 0;
126  unsigned long lastDoutLowTime = 0;
127  bool signalTimeoutFlag = 0;
128 };
129
130 #endif

```

Code voorbeeld B.3: Weegschaal - *HX711_ADC.h*

```
2 -----  
3 HX711_ADC  
4 Arduino library for HX711 24-Bit Analog-to-Digital  
5 Converter for Weight Scales  
6 Olav Kallhovd sept2017  
7 -----  
8 */  
9 //number of samples in moving average dataset, value must  
10 //be 1, 2, 4, 8, 16, 32, 64 or 128.  
11 #define SAMPLES 16 //default  
12 //value: 16  
13 //adds extra sample(s) to the dataset and ignore peak  
14 //high/low sample, value must be 0 or 1.  
15 #define IGN_HIGH_SAMPLE 1 //default  
16 //value: 1  
17 #define IGN_LOW_SAMPLE 1 //default  
18 //value: 1  
19 //microsecond delay after writing sck pin high or low.  
20 //This delay could be required for faster mcu's.  
21 //So far the only mcu reported to need this delay is the  
22 //ESP32 (issue #35), both the Arduino Due and ESP8266  
23 //seems to run fine without it.  
24 //Change the value to '1' to enable the delay.  
25 #define SCK_DELAY 0 //default  
26 //value: 0  
27 //if you have some other time consuming (>60us) interrupt  
28 //routines that trigger while the sck pin is high, this  
29 //could unintentionally set the HX711 into "power down"  
30 //mode  
31 //if required you can change the value to '1' to disable  
32 //interrupts when writing to the sck pin.  
33 #define SCK_DISABLE_INTERRUPTS 0 //default  
34 //value: 0
```

Codevoorbeeld B.4: Weegschaal - *config.h*

B.1.3 Extra bestanden

```
1 #include <stdbool.h>  
2
```

```

3 bool setupLoadCell();
4 uint8_t updateLoadCell();
5 void tareNoDelayLoadCell();
6 bool getTareStatusLoadCell();
7 bool refreshDataSetLoadCell();
8 float getNewCalibrationLoadCell(float known_mass);
9 float getDataLoadCell();
10 float getCalFactorLoadCell();
11 void setCalFactorLoadCell(float cal);

```

Codevoorbeeld B.5: Weegschaal - *main.h*

```

1 #include <avr/interrupt.h>
2
3 #ifndef _SHIMS_H_
4 #define _SHIMS_H_
5 extern "C" {
6
7 #define interrupts sei
8 #define noInterrupts cli
9 #define byte uint8_t
10 #define yield()
11 #define micros() 0
12 #define boolean bool
13
14 #define HIGH (bool)1
15 #define LOW (bool)0
16 #define OUTPUT (bool)1
17 #define INPUT (bool)0
18 #define WHILE_LOW (unsigned char)0
19 #define CHANGING_EDGE (unsigned char)1
20 #define FALLING_EDGE (unsigned char)2
21 #define RISING_EDGE (unsigned char)3
22
23 #ifndef uint32
24 #define uint32 uint32_t
25 #endif
26
27 #ifndef millis
28 #define millis _ticks
29 #endif
30
31 // nullable annotations are meaningless in GCC
32 #define __nonnull
33 #define __nullable

```

```
34
35 #ifndef digitalRead
36 #define digitalRead _digitalRead
37 #endif
38
39 #ifndef digitalWrite
40 #define digitalWrite _digitalWrite
41 #endif
42
43 #ifndef pinMode
44 #define pinMode _pinMode
45 #endif
46
47 #ifndef delayMicroseconds
48 #define delayMicroseconds _delayUs
49 #endif
50
51 #ifndef delay
52 #define delay _delayMs
53 #endif
54
55     void _i2cinit(unsigned char speed, unsigned char
56     premultiplier, bool activatePullups);
57     bool _i2cstart();
58     void _i2cstop();
59     bool _i2cwrite(unsigned char byte);
60     unsigned char _i2cread(bool sendAck);
61
62     // higher level I2C
63     void _i2cWriteByte(unsigned char address, unsigned
64     char byte);
65     void _i2cWriteBuffer(unsigned char address, unsigned
66     short maxMsgLen, const char * buffer, bool isString);
67     const char * __nonnull _i2cReadBuffer(unsigned char
68     address, unsigned char size, char * __nullable buffer)
69     ;
70     unsigned char _i2cReadSingleByte(unsigned char
71     address);
72     void _i2cWritePGMString(unsigned char address, const
73     char * __nonnull message, bool addNewline);
74
75     // SPI
76     // enable hardware SPI and set parameters, note: this
77     // takes over pins 11,12,13 and sets pin 10 as output (
78     // for use as SS)
79     // speed is 0 - 3, 0 is fastest, clock speed/4, 1 =
```

```
clock/16, 2 = clock/64, 3 = clock/128
71 void _setupSPIAsMaster(unsigned char speed, unsigned
char mode, bool lsb);
72 // not recommended when already running on maximum
speed (speed == 0, clock/4)
73 void _setupSPIDoubleSpeed(bool doubleSpeed);
74 // blocks until transmission and reception complete
75 unsigned char _sendSPIByteBlocking(unsigned char byte
);
76 // blocks until transmission and reception complete
77 const char * __nonnull _sendReceiveSPIBufferBlocking(
unsigned short maxMsgLen, const char * __nonnull
message, bool sendMessage, bool sendString, bool
receiveString);
78 // blocks until reception complete
79 const char * __nonnull _receiveSPIBufferBlocking(
unsigned short maxMsgLen, bool receiveString);
80 // disable hardware SPI, release pins 11,12,13, you
should now reset their input/output modes if you want
to use them
81 void __attribute__((weak)) _stopSPI();
82
83 // utils
84 uint32 _ticks();
85 void _delayMs(unsigned short delay);
86 void _delayUs(unsigned short delayUs);
87
88 // digital IO
89 bool _digitalRead(unsigned char pin);
90 void _digitalWrite(unsigned char pin, bool value);
91 void _pinMode(unsigned char pin, bool write);
92
93 /* Raw read/write. */
94 unsigned char _getPortB();
95 unsigned char _getPortC();
96 unsigned char _getPortD();
97
98 void _setPortB(unsigned char value);
99 void _setPortC(unsigned char value);
100 void _setPortD(unsigned char value);
101
102 unsigned char _getDDRB();
103 unsigned char _getDDRC();
104 unsigned char _getDDRD();
105
106 void _setDDRB(unsigned char value);
```

```
107     void _setDDRC(unsigned char value);
108     void _setDDRD(unsigned char value);
109
110     //USART
111     void _sendByte(unsigned char byte);
112     // send a hard coded string from the PROGMEM (does
113     // not use RAM)
114     void _sendString(unsigned short maxMsgLen, const char
115     * __nonnull message, bool addNewline); // must be
116     NULL terminated
117     // send a buffer from RAM (do not mix these two up!)
118     void _sendBuffer(unsigned short maxMsgLen, const char
119     * __nonnull buffer, bool addNewline); // must be NULL
120     terminated
121     bool _available();
122     unsigned char _receiveByte();
123
124     // SPI
125     // blocks until transmission and reception complete
126     unsigned char _sendSPISPIByteBlocking(unsigned char byte
127 );
128     // blocks until transmission and reception complete
129     const char * __nonnull _sendReceiveSPIBufferBlocking(
130         unsigned short maxMsgLen,
131         const char * __nonnull message,
132         bool sendMessage,
133         bool sendString,
134         bool receiveString);
135
136     // Memory efficient string buffer handling/building.
137     // note, only one string buffer can be used at a time
138     bool _stringAddSingleCharacter(unsigned char byte);
139     bool _stringAddWord(unsigned short word, bool
140     bigEndian);
141     void _stringStartNew();
142     const char * __nonnull _stringCurrentValue();
143     const char _stringCurrentLength();
144     const char _stringRemainingLength();
145     const char * __nonnull _stringAdd(const char *
146     __nullable message);
147     const char * __nonnull _stringAddFromProgmem(const
148     char * __nullable message);
149     const char * __nonnull _emptyStringBuffer();
150 }
```

Codevoorbeeld B.6: Weegschaal - *shims.h*

Herkomst	Bestanden
Projectspecifiek	main.swift
Project Axel Roest	main.swift (Roest, 2020b)
	SerialHandler.swift (Roest, 2020c)
	SQLiteHandler.swift (Roest, 2020d)

Tabel B.2: Herkomst codebestanden communicatie

B.2 Communicatie

De volgende codebestanden dienen uitgevoerd te worden op twee verschillende apparaten. Terwijl Code voorbeeld B.7 onderdeel is van een Swift for Arduino-project, zijn Code voorbeelden B.8, B.9 en B.10 onderdeel van een groter project dat gedraaid kan worden op macOS- of Linuxapparaten waarop Swift geïnstalleerd is. Het project waartoe die drie laatste bestanden behoren, werd opgesteld door Axel Roest en is in zijn geheel terug te vinden als repository op GitLab². Met het oog op transparantie, biedt Tabel B.2 een overzicht van de herkomst van de verschillende codebestanden uit deze sectie.

B.2.1 Arduino

```

1 import AVR
2
3 // PIN SETUP
4 let potentiometerPin: Pin = A0
5 let ledStripPin: Pin = 12
6 pinMode(pin: potentiometerPin, mode: INPUT)
7 pinMode(pin: ledStripPin, mode: OUTPUT)
8
9 // LED STRIP SETUP
10 let amountOfLeds: UInt16 = 60
11 iLEDFastSetup(pin: ledStripPin, pixelCount: amountOfLeds,
    hasWhite: false, grbOrdered: true)
12
13 // VARIABLES SETUP
14 var currentColorLeds: iLEDFastColor = iLEDOff
15 var currentMassWeight: UInt16 = 0
16
17 // SERIAL SETUP
18 SetupSerial()

```

²<https://gitlab.com/axello/plantinject/-/tree/main>

```
19 // Time for serial port to stabilize
20 delay(milliseconds: 250)
21 // Already write start value to serial
22 sendMassWeightToSerial(currentMassWeight)
23
24 // LOOP
25 while true {
26     analogReadAsync(pin: potentiometerPin) { value in
27         changeLedStripColorDependingOn(massWeight: value)
28
29         // Only write to serial when necessary
30         if value != currentMassWeight {
31             sendMassWeightToSerial(value)
32             currentMassWeight = value
33         }
34     }
35
36     delay(milliseconds: 1000)
37 }
38
39 // FUNCTIONS
40 func changeLedStripColorDependingOn(massWeight value: UInt16) {
41     let colorToDisplay: iLEDFastColor
42     if value > 700 {
43         colorToDisplay = iLEDRed
44     } else if value > 400 {
45         colorToDisplay = iLEDFastMakeColor(red: 255,
46 green: 165, blue: 0, white: 0)
47     } else {
48         colorToDisplay = iLEDOff
49     }
50
51     // Only change color when necessary
52     if colorToDisplay != currentColorLeds {
53         for _ in 1...amountOfLeds {
54             iLEDFastWritePixel(color: colorToDisplay)
55         }
56         // Time for LEDs to change color
57         delay(microseconds: 6)
58         currentColorLeds = colorToDisplay
59     }
60
61 func sendMassWeightToSerial(_ value: UInt16) {
62     print(staticString: "[", addNewline: false)
```

```

63     print(value, addNewline: false)
64     print("]")
65 }
```

Codevoorbeeld B.7: Communicatie - *main.swift*

B.2.2 Raspberry Pi

```

1  //
2  //  main.swift
3  //  plantinject
4  //
5  //  Created by Axel Roest on 22-02-2020.
6  //  Copyright 2020 App Academy. All rights reserved.
7  //
8
9 import Foundation
10
11 var serialDevice: String = "serial port device as
12   argument"    ///dev/cu.usbmodem141101"
13 var sqliteDbPath: String = "readings.sqlite3"      //"path
14   to sqlite3 database"
15 let db: SQLiteHandler
16 let serialHandler: SerialHandler
17
18 do {
19     db = try SQLiteHandler(databaseFile: sqliteDbPath)
20 } catch {
21     print("could not open sqlite3 database '\(
22       sqliteDbPath)\'")
23     exit(2)
24 }
25
26 func sqliteInjector(_ string: String) {
27     if let intValue = Int(string) {
28         db.write(sensorId: 1, sensorValue: Double(
29           intValue))
30     }
31     print("value: \(string)")
32 }
33
34 func sqliteInjector(data: Data) {
35     let valueString = String(data:data, encoding:.utf8)!
36     if let intValue = Int(valueString) {
```

```
33         db.write(sensorId: 1, sensorValue: Double(
34             intValue))
35     }
36 }
37
38 func getSerialPortArgument() -> String? {
39     if CommandLine.argv > 1 {
40         let serialPort = CommandLine.arguments[1]
41         return serialPort
42     } else {
43         print("Usage: plantinject <serialport device>")
44     }
45     return nil
46 }
47
48 serialDevice = getSerialPortArgument() ?? serialDevice
49 guard let serialPortHandler = SerialHandler(device:
50     serialDevice, injector: sqliteInjector) else {
51     exit(1)
52 }
53
54 serialHandler = serialPortHandler
55 serialHandler.waitOnSerial()
56
57 // When backgrounding daemon style, use & to keep script
58 // in background
59RunLoop.main.run()
```

Codevoorbeeld B.8: Communicatie - *main.swift* (Roest, 2020b)

```
1 //
2 //  SerialHandler.swift
3 //  plantinject
4 //
5 //  Created by Axel Roest on 22-02-2020.
6 //  Copyright 2020 App Academy. All rights reserved.
7 //
8 //  Update: use https://github.com/yeokm1/SwiftSerial/blob
8 //          /master/Sources/SwiftSerial.swift, as that is 100%
8 //          swift
9
10 import Foundation
11 import SwiftSerial
```

```
12
13 typealias InjectDataHandler = (String) -> (Void)
14 var port: SerialPort?
15
16 class SerialHandler: NSObject {
17     private let dataInjector: InjectDataHandler
18     private var receiving: Bool = false
19     private let startTag: Character = "["
20     private let endTag: Character = "]"
21
22     init?(device: String, injector: @escaping
23         InjectDataHandler) {
24         dataInjector = injector
25         super.init()
26         if let newPort = openPort(device) {
27             port = newPort
28             receiving = true
29         } else {
30             print("Serial device '\\"(device)'\ not found.
31         ")
32     }
33
34     func openPort(_ device: String) -> SerialPort? {
35         let port = SerialPort(path: device)
36         port.setSettings(receiveRate: .baud9600,
37             transmitRate: .baud9600, minimumBytesToRead: 0)
38         do {
39             try port.openPort()
40         } catch {
41             print("Could not open serial device '\\"(device)'.")
42         }
43         return port
44     }
45
46     deinit {
47         if let port = port {
48             port.closePort()
49             print("Port Closed")
50         }
51     }
52
53     func waitOnSerial() {
```

```
54     #if os(Linux)
55         linuxBackgroundRead()
56
57     #elseif os(OSX)
58         if let port = port {
59             self.backgroundRead(port: port, dataHandler:
60             self.dataInjector)
61         }
62     }
63
64     func linuxBackgroundRead() {
65         guard let port = port else { return }
66
67         var line = ""
68         while true {
69             do{
70                 let data = try port.readData(ofLength: 2)
71                 if let text = String(data: data, encoding
72 : .utf8) {
73                     line.append(text)
74                 }
75                 let (value, rest) = processLine(line)
76                 if let value = value {
77                     dataInjector(value)
78                 }
79                 line = rest
80             } catch {
81                 print("Error: \(error)")
82             }
83         }
84
85     func backgroundRead(port: SerialPort, dataHandler:
86     @escaping InjectDataHandler) {
87         var line = ""
88 //         sleep(1)
89         do{
90             let data = try port.readData(ofLength: 2)
91             if let text = String(data: data, encoding
92 : .utf8) {
93                 line.append(text)
94             }
95             let (value, rest) = processLine(line)
96             if let value = value {
```

```

96                     dataHandler(value)
97                 }
98                 line = rest
99             } catch {
100                 print("Error: \(error)")
101             }
102         }
103     }
104
105     func processLine(_ line: String) -> (String?, String)
106     {
107         if let tagIndex = line.firstIndex(of: startTag),
108             tagIndex < line.endIndex {
109             let startOfSentence = line.index(after:
110                 tagIndex)
111             let firstSentence = line[startOfSentence...]
112
113             if let endTagIndex = firstSentence.firstIndex
114                 (of: endTag) {
115                 let endOfValue = firstSentence.index(
116                     before: endTagIndex)
117                 let value = firstSentence[...endOfValue]
118                 let afterEndTagIndex = firstSentence.
119                     index(after: endTagIndex)
120                 let restString = firstSentence[
121                     afterEndTagIndex...]
122                 return (String(value), String(restString))
123             }
124         }
125     }
126     return (nil, line)
127 }
128 }
```

Codevoorbeeld B.9: Communicatie - *SerialHandler.swift* (Roest, 2020c)

```

1 //
2 // SQLiteHandler.swift
3 // plantinject
4 //
5 // Created by Axel Roest on 22-02-2020.
6 // Copyright 2020 App Academy. All rights reserved.
7 //
8 // On linux you have to do
9 // sudo apt-get install libsqlite3-dev
```

```
10
11 import Foundation
12 import PerfectSQLite
13
14 enum SQLiteError: Error {
15     case openingDBFailure
16     case creatingTableFailure
17 }
18
19 class SQLiteHandler {
20     var dbPath: String
21     let dateFormatter = DateFormatter()
22
23     init(databaseFile: String) throws {
24         var url = URL(fileURLWithPath: databaseFile)
25         if "sqlite3" != url.pathExtension {
26             url.appendPathExtension("sqlite3")
27         }
28         print("Readings database at: \(url.path)")
29         dbPath = url.path
30
31         let sqlite = try SQLite(dbPath)
32         defer {
33             sqlite.close()
34         }
35
36         dateFormatter.dateFormat = "yyyy-MM-dd'T'HH:mm:ss
37             // compatible with sqlite3 injection
38         try setup()
39     }
40
41     func createTable() throws {
42         let query = "CREATE TABLE IF NOT EXISTS Reading(
43             id INTEGER PRIMARY KEY AUTOINCREMENT, sensor INTEGER,
44             date TEXT, value REAL);"
45
46         do {
47             let sqlite = try SQLite(dbPath)
48             defer {
49                 sqlite.close()
50             }
51
52             try sqlite.execute(statement: query)
53             print("Reading table creation success")
54         } catch (let error) {
55             print("Failure creating database tables: \(
```

```
error)") //Handle Errors
53     }
54 }
55
56 func setup() throws {
57     try createTable()
58 }
59
60 func write(sensorId: Int, sensorValue: Double) {
61     let query = "INSERT INTO Reading (sensor, date,
62 value) VALUES (:1, :2, :3);"
63     let now = dateFormatter.string(from: Date())
64
65     do {
66         let sqlite = try SQLite(dbPath)
67         defer {
68             sqlite.close()
69         }
70
71         try sqlite.execute(statement: query) {
72             (stmt:SQLiteStmt) -> () in
73
74                 try stmt.bind(position: 1, sensorId)
75                 try stmt.bind(position: 2, now)
76                 try stmt.bind(position: 3, sensorValue)
77             }
78         } catch (let error) {
79             print("error inserting reading: \(error)")
80         }
81     }
81 }
```

Codevoorbeeld B.10: Communicatie - *SQLiteHandler.swift* (Roest, 2020d)

Bibliografie

- Barr, M. (2001). Pulse Width Modulation. *Embedded Systems Programming*, 103–104. <https://barrgroup.com/embedded-systems/how-to/pwm-pulse-width-modulation>
- Crisp, J. (2004). *Introduction to Microprocessors and Microcontrollers*. Newnes.
- Gratton, D. A. (2013). *The Handbook of Personal Area Networking Technologies and Protocols*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511979132>
- Luuk, I. (2020, april 17). *Arduino Scale with HX711 and 50kg Bathroom Scale Load Cells | Step by Step Guide*. Verkregen 12 mei 2021, van <https://www.youtube.com/watch?v=LIuf2egMioA>
- Maksimovic, M., Vujoovic, V., Davidović, N., Milosevic, V. & Perisic, B. (2014). Raspberry Pi as Internet of Things hardware: Performances and Constraints.
- Marwedel, P. (2010). *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems* (2nd). Springer Publishing Company, Incorporated. <https://doi.org/https://doi.org/10.1007/978-3-319-56045-8>
- Mattern, F. & Floerkemeier, C. (2010). From the Internet of Computers to the Internet of Things. In K. Sachs, I. Petrov & P. Guerrero (Red.), *From Active Data Management to Event-Based Systems and More: Papers in Honor of Alejandro Buchmann on the Occasion of His 60th Birthday* (pp. 242–259). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-17226-7_15
- McRoberts, M. (2010). *Beginning Arduino* (1st). Apress.
- Patnaikuni, D. (2017). A Comparative Study of Arduino, Raspberry Pi and ESP8266 as IoT Development Board. *International Journal of Advanced Research in Computer Science*, 8(5), 2350–2352. <https://doi.org/10.26483/ijarcs.v8i5.3959>
- Peto, C. (2017, augustus 10). *Swift for Arduino: An Experimental Compiler*. Verkregen 25 december 2020, van <https://skillsmatter.com/skillscasts/10773-swift-for-arduino-an-experimental-compiler>

- Peto, C. (2018, december 7). *uSwift*. <https://github.com/carlos4242/uSwift/blob/master/README.md>
- Peto, C. (2020, september 4). *Why Swift for Arduino*. Verkregen 25 april 2021, van <https://www.youtube.com/watch?v=E8WD7Wr37-U>
- Peto, C. & Kay, E. (2021, april 22). *Swift For Arduino (TM)* (computersoft.; Versie 4.4.1).
- Roest, A. (2020a, november 22). *Full stack Swift with Arduino (CocoaHeadsNL)*. Verkregen 29 november 2020, van <https://www.youtube.com/watch?v=09ro41ECED8>
- Roest, A. (2020b, november 5). *main.swift*. Verkregen 13 mei 2021, van <https://gitlab.com/axello/plantinject/-/blob/main/Sources/plantinject/main.swift>
- Roest, A. (2020c, november 5). *SerialHandler.swift*. Verkregen 13 mei 2021, van <https://gitlab.com/axello/plantinject/-/blob/main/Sources/plantinject/SerialHandler.swift>
- Roest, A. (2020d, oktober 27). *SQLiteHandler.swift*. Verkregen 13 mei 2021, van <https://gitlab.com/axello/plantinject/-/blob/main/Sources/plantinject/SQLiteHandler.swift>
- Severance, C. (2014). Massimo Banzi: Building Arduino. *Computer*, 47, 11–12. <https://doi.org/10.1109/MC.2014.19>
- Sobota, J., PiŚl, R., Balda, P. & Schlegel, M. (2013). Raspberry Pi and Arduino boards in control education [10th IFAC Symposium Advances in Control Education]. *IFAC Proceedings Volumes*, 46(17), 7–12. <https://doi.org/https://doi.org/10.3182/20130828-3-UK-2039.00003>
- Xia, F., Yang, L. T., Wang, L. & Vinel, A. (2012). Internet of Things. *International Journal of Communication Systems*, 25(9), 1101–1102. <https://doi.org/https://doi.org/10.1002/dac.2417>