

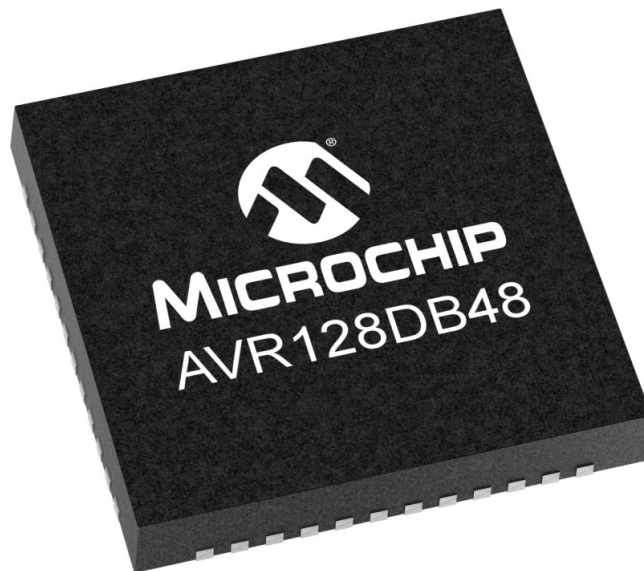
# uC-Programmeren

Op de AVR128DB48 Curiosity Nano met MPLAB X IDE en MCC

Douwe T. Schotanus met diverse bijdragen van Marco Winkelman en Anke Kuijk

Maart 2025

EDPR.22 PRM



## Reader uC-Programmeren

Versie 3.6

Domein Techniek - Engineering en Design - Opleiding Elektrotechniek

### Versiebeheer

Versie	Datum	Wijzigingen
1.0	13-02-23	Toevoegen hoofdstuk GPIO
1.1	20-02-23	Toevoegen hoofdstuk State-Machine
1.2	06-03-23	Toevoegen hoofdstuk ISR
1.3	13-03-23	Toevoegen hoofdstuk Clocks, Timers PWM
1.4	20-03-23	Toevoegen hoofdstuk ADC
1.5	27-03-23	Toevoegen hoofdstuk Systeemintegratie
1.6	03-04-23	Toevoegen handleiding losse chip programmeren
2.0	01-12-23	Handleiding overgezet naar LATEX PDF
2.1	24-01-24	Overzetten hoofdstuk Systeemintegratie naar Hardware Design
2.2	31-01-24	Herstructureren reader en updaten Week 1 en 2
2.3	13-03-24	Aanpassen practicum instructie week 5 t.b.v. MCC
2.4	14-03-24	Oplossen bugje in code week 5
2.5	18-03-24	Verplaatse communicatie naar appendix, verbeteren week 6
2.6	18-03-24	Oplossen bug in instructie en code VREF voor ADC Week 5
3.0	17-01-25	Updaten reader naar MPLAB, schrijven inleiding en verbeteren week 1
3.1	28-01-25	Splitsen GPIO in Week 1 en Week 2. Updaten week 2
3.2	30-01-25	Herschrijven week 3.
3.3	31-01-25	Herschrijven week 4.
3.4	11-02-25	Herschrijven week 5 theorie. Aanpassen nummering vragen.
3.5	26-02-25	Herschrijven week 5 practicum en week 6 theorie
3.6	04-03-25	Herschrijven week 6 practicum.

# Contents

<b>0</b>	<b>Introductie</b>	<b>5</b>
0.1	Inleiding . . . . .	5
0.2	Organisatie Practicum . . . . .	8
<b>1</b>	<b>Microcontrollers en Programmeren</b>	<b>9</b>
1.1	Vorbereiding . . . . .	9
1.2	Practicum . . . . .	14
<b>2</b>	<b>GPIO</b>	<b>17</b>
2.1	Vorbereiding . . . . .	17
2.2	Practicum . . . . .	22
<b>3</b>	<b>Interrupts</b>	<b>30</b>
3.1	Vorbereiding . . . . .	30
3.2	Practicum . . . . .	34
<b>4</b>	<b>Clocks, Timers &amp; PWM</b>	<b>40</b>
4.1	Vorbereiding . . . . .	41
4.2	Practicum . . . . .	48
<b>5</b>	<b>ADC</b>	<b>54</b>
5.1	Vorbereiding . . . . .	54
5.2	Practicum . . . . .	64
<b>6</b>	<b>State Machine</b>	<b>70</b>
6.1	Vorbereiding . . . . .	70
6.2	Practicum . . . . .	77
<b>A</b>	<b>Digitale sensors &amp; Communicationprotocols</b>	<b>85</b>
A.1	Vorbereiding . . . . .	85

A.2	Practicum . . . . .	91
<b>B</b>	<b>Losse chip en gebruik programmer</b>	<b>92</b>

# 0 Introductie

## 0.1 Inleiding

Microcontrollers (uC's) zitten tegenwoordig overal. Je kunt het zo gek niet verzinnen, of er zit een microcontroller in. Een microcontroller is een chip, die naast CPU (processor) mogelijkheden vaak verschillende hardware heeft om verschillende functionaliteiten te vervullen. Het is het ultieme voorbeeld van een flexibel component. Waar je misschien gewend bent om speciale IC's (Integrated Circuits) te gebruiken om specifieke taken te verrichten, denk bijvoorbeeld aan timer IC's, motorcontroller IC's of hexdisplay IC's, kan een microcontroller heel veel verschillende taken uitvoeren. Een groot verschil hierin is, dat je een microcontroller programmeert met behulp van software. Heel vaak zie je dat de microcontroller gebruikt wordt om als centraal besluitorgaan te functioneren. Je ziet dan ook dat de uC als centrum van je PCB (printplaat) fungeert, met verschillende chips, sensors en actuatoren er omheen. Je kunt een uC eindeloos vaak herprogrammeren, en vaak is er communicatie mogelijk tussen de uC en bijvoorbeeld je laptop.

De uC die we in dit vak gebruiken is de AVR128db48. Dit is een 8-bits microcontroller (instructiebreedte van 8-bits) met 128 kB flash geheugen en een instelbare kloksnelheid tot 24 MHz. Met dat soort statistieken kun je thuiskomen! Hoe zit dat dan? Waarom gebruiken we zo'n trage barebones uC? Nou een beetje onderzoek toont aan dat de meeste uC's helemaal niet zo heel krachtig zijn. Althans, als we ze vergelijken met bijvoorbeeld de CPU in je laptop. De CPU van de laptop waarop deze reader geschreven is, is een Intel I5 met kloksnelheid van 2,5 GHz, 16 GB RAM, en nog eens 256 GB flash (SSD) geheugen. Tevens is de instructiebreedte 64-bits en heeft hij ook nog eens 10 processorkernen. Dan lijkt die uC ineens niet meer zo spectaculair. Toch is dit een product wat nog zeer recent ontwikkeld is, en actief geproduceerd wordt. De reden daarvoor is simpel. You pay for what you get. De CPU in mijn laptop kost ongeveer 250 euro. De kosten van de AVR128dB48 is ongeveer 2,50 euro. Dat is een factor 100 verschil. Je kunt je al wel voorstellen waarom deze chip dus toch gebruikt wordt. De toepassing van een laptop CPU is heel anders dan een uC. De specificaties daarom ook. Zelfs binnen de uC wereld heb je uC's die je voor 0,10 euro kunt kopen (25 keer zo goedkoop), maar je hebt ook uC's die 100 euro kosten. Hierbij geldt weer net zo goed, je betaalt voor wat je krijgt. Denk hierbij aan hogere snelheden, meer of betere peripherals, WIFI of bluetooth mogelijkheid en energieverbruik. Daarom is het erg belangrijk om goed na te denken over wat je nodig hebt.

uC's worden gemaakt door fabrikanten. Een uC-fabrikant gebruikt zijn eigen kennis en vaak een basis CPU-architectuur zoals bijvoorbeeld ARM of RISC-V om een ontwerp te maken. Dit ontwerp kan daarnaast verschillende features bevatten. Vaak is er dan een familie van uC's voor verschillende niche's. Kijk maar eens naar figuur 1. Je ziet hier al 5 telgen uit de AVR128DB familie. Om je een illustratie te geven: Naast deze familie heeft het bedrijf Microchip (voorheen Atmel, wat onze uC produceert) al tientallen verschillende families (Atmega, PIC ... en subcategoriën). Concurrenten van Microchip, zoals bijvoorbeeld ESP (ESP32), Raspberry PI (PICO), STM (STM32), Texas Instruments (MSP) en tientallen anderen produceren ook weer allemaal eigen families van chips. Wanneer je bijvoorbeeld op Mouser kijkt naar de hoeveelheid verschillende uC's, dan zijn dit er meer dan 50000 verschillende varianten. Kortom de keuze is reuze! Vanuit dit gigantische aanbod hebben wij de AVR128DB48 gekozen, omdat dit een typische gemiddelde en betaalbare uC is, die eenvoudig te programmeren is. Deze chip heeft de meeste basisfunctionaliteiten, waarmee je kunt leren hoe alles werkt. Hier geldt nadrukkelijk, er is geen beste optie! Dit is typisch een ontwerpkeuze die afhankelijk van jouw project heel anders kan zijn! Het is daarom belangrijk dat je je in de toekomst inleest in het uC-jargon, en de mogelijkheden van verschillende uC's met elkaar vergelijkt. Prijs

is hierin extreem belangrijk. Een verschil van een paar eurocent lijkt niet zoveel, maar als je een product ontwikkelt wat 1.000.000 gemaakt wordt, heb je het toch al gauw over tienduizenden, zo niet honderdduizenden euro's aan extra kosten.

## AVR<sup>®</sup> DB MCU Family

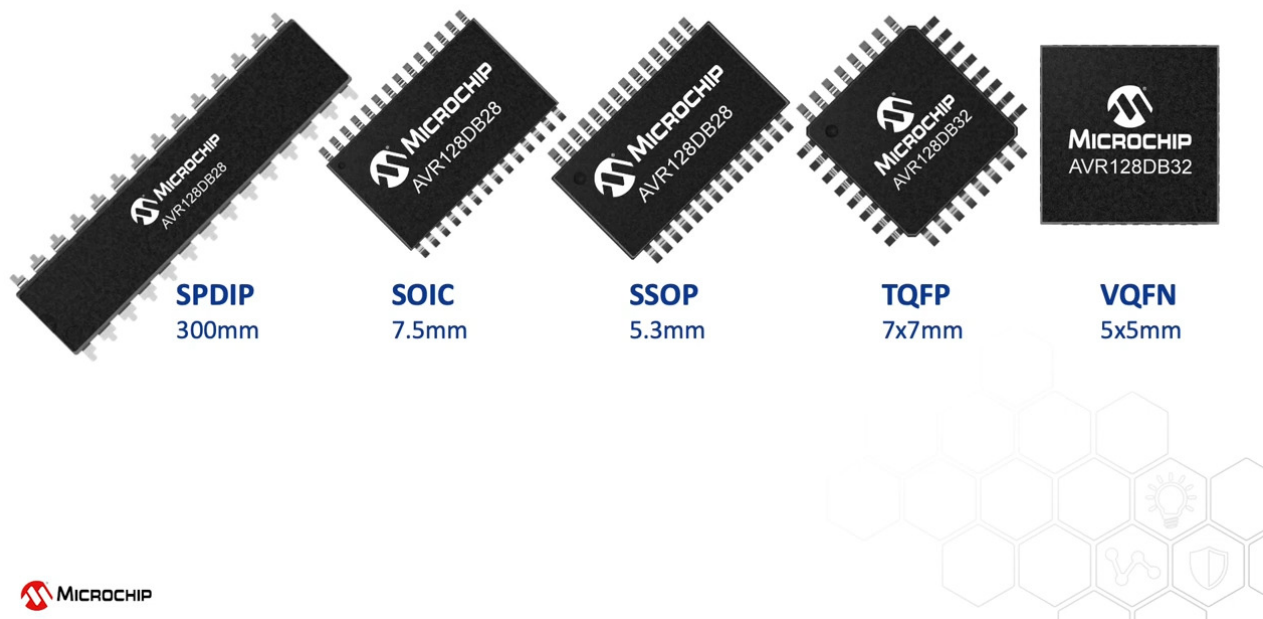


Figure 1: Een aantal telgen uit de AVR128DB familie. Kijk eens naar de afmetingen en afkorting daarboven. Kijk ook eens naar het getal achter 128DB. Onze variant de DB48 staat hier overigens niet tussen.

Het programmeren van een uC is wat lastiger dan je denkt. Zo kun je geen van de in figuur 1 weergegeven varianten direct op je computer aansluiten. Je hebt hiervoor een speciale programmeerinterface (verbinding) nodig. Meestal worden deze interfaces geleverd door de fabrikant, of gebruiken ze een veelvoorkomende standaardinterface. Deze interface bestaat uit een usb verbinding, waarmee je de computer verbindt en verschillende sporen die naar speciale toegewezen pinnen op de uC gaan. Deze chips zijn los verkrijgbaar en gaan we aan het einde van dit practicum ook gebruiken. Daarvoor gaan we gebruik maken van een zogenaamd development board. Dit development board is de curiosity nano, en heb je al eens gebruikt tijdens het vak Project Digitaal. Je ziet dit bordje afgebeeld in figuur 2.

Fabrikanten van uC's maken deze development boards zodat jij hun product kunt uitproberen en zodat het makkelijker is om de chip te debuggen en te gebruiken. Zo kun je eenvoudig een prototype maken van de software, zonder dat je eerst een PCB hoeft te fabriceren. Daarnaast kan het ontwikkelen van hardware en software parallel aan elkaar gebeuren. Dit scheelt tijd en dus ook kosten. Naast een programmeerinterface geeft het ontwikkelbord ons verschillende andere functionaliteiten. Zo is er een knopje (PB2) en een LEDje (PB3) en zijn er verschillende externe kristallen (16 MHz en 32 KHz). Ook zijn bijna alle pinnen van de DB48 beschikbaar gemaakt op de throughhole pads onder en boven. Dit maakt het mogelijk om de DB48 m.b.v. een header te verbinden met een breadboard. Een ander voorbeeld is het SMU-bordje (slimme meter uitlezer) waarop de curiosity nano gezet kan worden. Een extra bordje wat je op deze manier verbindt wordt ook wel hat (hoed) of shield (schild) genoemd. In het practicum gebruiken we ook het SMU-bordje voor extra functionaliteit. Tijdens DxC (Drive Exchange) kan het handiger zijn om de Curiosity nano los te koppelen en bijvoorbeeld te verbinden met een breadboard. Dit kun je t.z.t. doen. Let op! Een curiosity nano kost ongeveer 15-20 euro. Een losse chip kost 2,50. Het verschil dat je betaalt komt door de extra features van het ontwikkelbordje. Wees voorzichtig met dit bordje, want het is

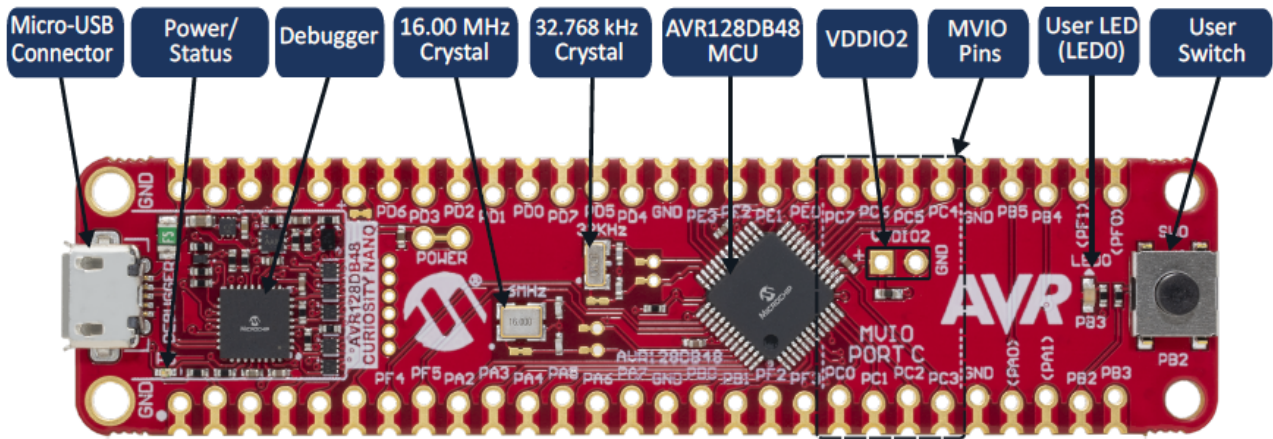


Figure 2: De Curiosity Nano. Een ontwikkelbord voor de AVR128DB48. Naast de chip (zwarte diamant) zie je een hele hoop extra electronica en zelfs nog een extra chip (programmer/debugger chip). De curiosity nano heeft een micro-usb interface waarmee hij met de computer verbonden kan worden. Daarnaast heeft hij een knopje (PB2) en een ledje (PB3).

natuurlijk net zo kwetsbaar als de losse chip! Denk dus altijd goed na voordat je ergens (extern) spanning opzet.

Zoals je misschien al wel gemerkt hebt, zijn er een hoop nieuwe dingen te leren. uC's zijn net als veel andere technische onderwerpen lastig om als los onderwerp te behandelen. Heel vaak moeten we uitstapjes maken om iets uit te kunnen leggen. Dit is helemaal niet erg, maar het kan ervoor zorgen dat het soms wat veel lijkt. Toch ga je hier veel aan hebben, want het zorgt ervoor dat je je kennis over meerdere onderwerpen met elkaar kunt verbinden. Zoals we in de introductie al verschillende andere onderwerpen hebben belicht zoals PCB's, maar ook productie. Zo ga je onderwerpen zoals sensoren en systeemontwerp ook tegenkomen in dit vak. We proberen hierin een zo sterk mogelijke koppeling te maken met het vak Hardware Design.

De reader en de docenten proberen je zo goed mogelijk te begeleiden in het leren programmeren en gebruiken van een uC. Toch is dit echt een vaardigheid die je opdoet door zelf aan de slag te gaan. Dit betekent vaak dat je zelf informatie moet opzoeken, zelf moet programmeren en wanneer je er niet uitkomt vragen moet stellen. Tegenwoordig kunnen al deze dingen gedaan worden door een LLM (Large Language Model). Deze modellen halen misschien de rauwe randjes weg, maar wees voorzichtig in het gebruik ervan. Vergeet niet om zelf na te denken, en zelf antwoord op vragen te formuleren. Soms is het frustrerend om lang vast te zitten met een probleem. Toch ga je veel meer leren door zelf dit probleem op te lossen en zelf code te schrijven. Hierin mag je je zeker laten inspireren door het werk van anderen, maar kopieer en plak niet klakkeloos. Probeer te begrijpen wat je doet, en wanneer je iets niet snapt de extra stap te zetten om dat toch te doen.

We wensen je veel programmeerplezier!

## 0.2 Organisatie Practicum

Hieronder staat een kort overzicht van de organisatie van het practicum. Voordat je met het practicumgedeelte begint, is het verplicht om de voorbereidende opdrachten thuis te maken. De docent controleert hierop. Deze voorbereiding is een belangrijk onderdeel van het leren en vormt ook onderdeel van de beoordeling voor uC-programmeren. Zie voor meer informatie de studiehandleiding .

Voordat je begint:

1. Installeer de benodigde software. Dit is MPLab X IDE.
2. Je programmeert je eigen bordje en schrijft zelf code. Samen code schrijven op 1 laptop is niet toegestaan.
3. Je schrijft een logboek waarin je zelf de vragen beantwoordt. Dit gehele logboek wordt aan het einde van het vak ingeleverd op Brightspace.
4. Je mag samenwerken. Zo mag je elkaar helpen, ideeën uitwisselen en samen de voorbereiding maken. Het is niet de bedoeling om een ander z'n werk 1-op-1 te kopiëren.
5. De verwachte inspanning is 1,5 uur voorbereiding en 3 uur practicum per week.
6. Er geldt een aanwezigheids- en inspanningsplicht bij het practicum. De docent controleert hierop. De docent zal instructie geven, wanneer er niet serieus (genoeg) gewerkt wordt. Er wordt gewerkt in een labsetting, dus we verwachten een professionele houding. Bij het herhaaldelijk in gebreke blijven, kan de docent de toegang tot het practicum aan de student ontfzeggen.
7. Wanneer je een bron gebruikt zoals een datasheet, refereer je daarnaar.

Elke week van het practicum behandelen we een gedeelte van de uC. De onderdelen die we gaan behandelen zijn :

1. Microcontrollers en Programmeren
2. GPIO
3. Interrupts (ISR)
4. Clocks, Timers & Pulse Width Modulation (PWM)
5. Analog to Digital Conversion (ADC)
6. State Machines



# 1 Microcontrollers en Programmeren

## 1.1 Voorbereiding

De voorbereiding van week 1 mag als uitzondering tijdens het practicum gemaakt worden. Alle andere voorbereidingen (week 2 t/m week 6) dienen thuis gemaakt te worden. Je laat je voorbereiding aftekenen bij de docent! Tip: deze voorbereiding kun je ook doen terwijl je MPLAB installeert.

### 1.1.1 IDE

1. Installeer [MPLAB X IDE](#) (doe dit bij voorkeur thuis).
2. Accepteer alle extra features die MPLAB wil installeren
3. Na het installeren van MPLAB X IDE start je het programma op, en installeer je eventuele updates

### 1.1.2 Microcontroller

Deze week maken we opnieuw kennis met de AVR128DB48 microcontroller (afgekort met uC) door middel van het Curiosity Nano ontwikkelbordje. We leren hoe we een nieuw project kunnen beginnen in MPLAB X IDE (Microchip Programming Lab X Integrated Development Environment). We maken een basisprogramma en leren hoe we een LED kunnen aansturen. De naam van de AVR 128 DB 48 is opgebouwd uit: AVR staat voor Alf & Vegard's RISC processor, de 128 slaat hier op de hoeveelheid geheugen (128 kB), DB voor de specifieke familie en de 48 slaat op het aantal GPIO-pinnen wat beschikbaar is.

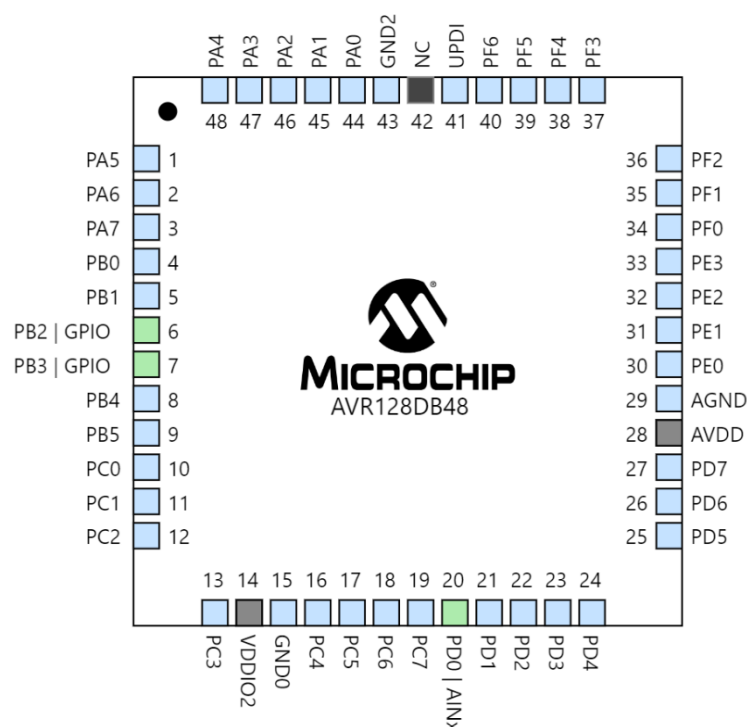


Figure 3: De Pin Package View van de AVR128DB48. We zien hier alle pinnen van deze microcontroller. Bron: MCC

Figuur 3) laat goed zien dat we over 48 pinnen beschikken. Zoals we in het hoofdstuk Introductie - Inleiding (0.1) in figuur 1 hebben laten zien, zijn er verschillende varianten beschikbaar. De 48 pins variant is een SMT (Surface Mount Technology) chip. De 28 pins variant is een DIP (Dual inline Package, dus als throughhole component) en is ook als losse chip op Windesheim beschikbaar.

Het begrip Microcontrollers ben je nu al heel wat keren tegengekomen, maar wat bedoelen we daar nou exact mee? Een microcontroller (uC) is in weze een processor (CPU) met extra functies die wij los kunnen kopen en in allerlei elektronische projecten kunnen gebruiken. In figuur 4 zien we een hardware schema van de allersimpelste weergave van een uC. Deze uC heeft een processor (CPU) die het programma (PRG) wat wij schrijven uitvoert. Daarnaast zijn er veel verschillende instellingen die we kunnen opslaan in het geheugen (MEM). Ook data en variabelen nodig voor het uitvoeren van ons programma bevindt zich in het geheugen. Daarnaast heeft de uC tal van connecties met eigen peripherals en de buitenwereld. Hetgene wat een uC vanuit elektronisch perspectief zo aantrekkelijk maakt is zijn programmeerbaarheid. We kunnen zelf een programma schrijven wat tal van complexe taken kan uitvoeren. Wanneer we dat allemaal met losse hardware hadden moeten oplossen wordt ons systeem erg complex en duur.

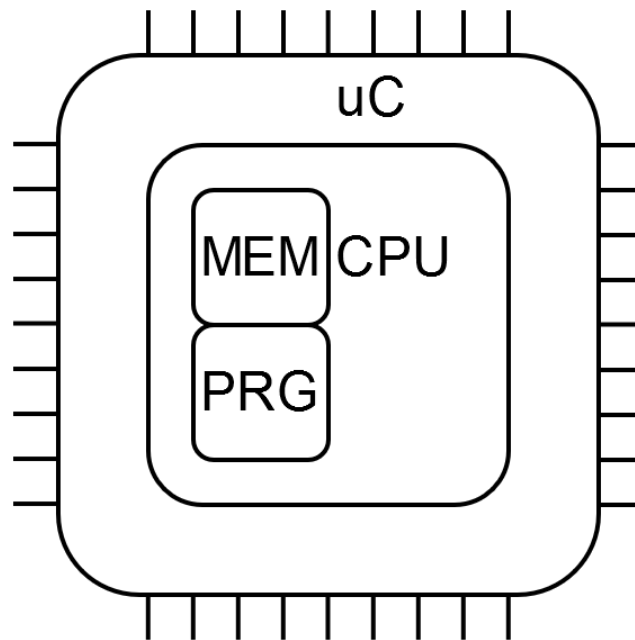


Figure 4: Hardware schema van een uC. De uC bevat een central processing unit (CPU). De CPU voert een programma (PRG) uit. Alle peripheral instellingen staan opgeslagen in speciale geheugenregisters (MEM). Het belangrijke verschil met een CPU is dat een uC verschillende hardware peripherals heeft (niet getekend) en veel connecties (IO) met de buitenwereld. Deze zijn weergegeven als losse lijntjes rondom de uC.

Zoals we in figuur 4 de uC lieten zien, verschilt hij in praktisch opzicht nog niet heel veel van een gewone standaard CPU zoals die in je laptop. Hetgene wat een uC zo sterk maakt in tal van verschillende projecten is de beschikbaarheid van veel verschillende hardware peripherals die een verscheidenheid van taken voor ons kunnen uitvoeren. Hiermee ontlasten we de CPU. Het is een erkenning dat we niet alles effectief met software kunnen oplossen. We lossen hardwareproblemen op met daarvoor geschikte hardware. Bijna al deze problemen kun je in software oplossen, maar ze maken de uC traag, energieslurpend en zijn over het algemeen minder (tijds)nauwkeurig. Voorbeelden van deze peripherals zijn GPIO (general purpose Input Output), ISR (interrupt service routine), OSC (oscillator), CLK (clock), Timer en ADC (analog to digital converter). Omdat dit allemaal in 1 chip zit, en omdat deze chip zeer breed inzetbaar is, zijn uC's extreem populair geworden. Een hoge populariteit i.c.m. grote opschaling van productie over de afgelopen 5 decennia heeft ervoor gezorgd dat ze tegenwoordig zeer goedkoop zijn.

Pak het datasheet van de AVR128DB48 erbij en navigeer naar pagina 13. Let op dat je het datasheet van de microcontroller gebruikt (losse chip), en niet het datasheet van de curiosity nano (ontwikkelbordje).

4. Licht de volgende begrippen toe: I/O, Peripherals.
5. Welke peripherals zie je allemaal in het schema op p.13 van het datasheet? Benoem er minimaal 4 en leg kort uit wat je verwacht dat ze doen. Wanneer je het niet weet, kun je altijd Googlen.

Ga nu naar pagina 16 van hetzelfde datasheet. We zien hier een schema van alle aansluitpinnen van de microcontroller.

6. Hoeveel I/O pinnen zie je?
7. De pinnen op je bordje kunnen verschillende functies hebben. Welke zijn dit? Wat doen ze?
8. In welke (functie) modus zitten I/O poorten standaard?

Kijk nu eens naar je curiosity nano (rode ontwikkelbordje). Je ziet rechts een knopje en een LED op je curiosity bordje. Wanneer nodig gebruik je bij de volgende vragen het datasheet van de curiosity nano (ontwikkelbordje).

9. Op welke pin zit de LED?
10. Hoe werkt de LED? (kijk online of in de datasheet van de curiosity nano)
11. Teken het schema van de pin + LED + voorschakelweerstand.
12. Gaat de LED aan bij een logische 1 (hoog) of een logische 0 (laag)? Licht dit toe met behulp van het schema.
13. Op welke pin zit de knop?
14. Hoe werkt het knopje?
15. Wat bedoelen we met een pull-up/pull-down weerstand?
16. Teken het schema van de knop over.
17. Krijg je bij het indrukken van de knop een logische 1 of een logische 0? Leg ook dit weer uit aan de hand van het overgetekende schema.
18. Wat is contactdender?
19. Hoe zou je op een eenvoudige (elektronische) manier contactdender kunnen voorkomen?

Omdat we straks meerdere knopjes en ledjes willen gaan gebruiken, en we maar 1 LED en 1 knopje op ons curiosity nano bordje hebben, is het tijd om verbinding te leggen met het SMU (slimme meter uitlezer) bordje.

20. Ga opzoek naar het elektrisch schema van de SMU en print deze uit op A4
21. Beschrijf de verschillende componenten die je in het schema ziet. Kun je deze componenten ook op je SMU bordje zien?

In het midden van het schema zie je 2 x 14 pinnetjes van de SMU (P1 en P2). Deze pinnetjes zijn aangesloten op je curiosity nano bordje, en dus verbonden met de AVR128DB48 microcontroller.

22. Schrijf voor elke pin ernaast met welke pin deze verbonden is. Voorbeeld: P1.2 van SMU is verbonden met PF4 (dan schrijf je er PF4 naast).

Er zijn een aantal onderdelen op het SMU-bordje dat we willen gaan gebruiken in de komende lessen: de knoppen, de LED en de potmeter. Je kunt deze componenten op dezelfde manier gebruiken als de componenten op het curiosity nano bord.

23. Kopieer onderstaande tabel in je logboek en vul in.

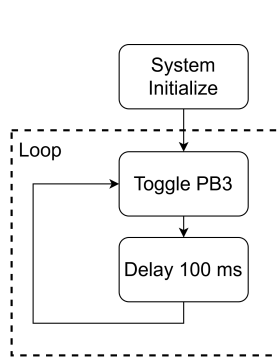
Component	Functie	Pin SMU	Pin Curiosity Nano
LED1			
LED2			
SW1			
SW2			
SW3			
Potmeter			

### 1.1.3 Programma

Zoals je wel hebt gemerkt is de wereld van uC's dol op 2/3/4 letter afkortingen (tel maar eens het aantal afkortingen in figuur 3). Dit kan zeker aan het begin erg intimiderend en lastig zijn. We gaan je helpen deze afkortingen te leren begrijpen. Mocht je iets niet snappen, vraag dit na bij de docent! De reader is zo opgebouwd dat we elke week een nieuwe peripheral behandelen. Je kunt daardoor gemakkelijk terugbladeren naar een stuk waar je meer moeite mee hebt. Om het deze week eenvoudig te houden gaan we beginnen met het überhaupt werkend krijgen van de uC en willen we dit demonstreren door een LED op de Curiosity Nano (ons ontwikkelbordje) aan te sturen. Hiervoor is er een speciale peripheral, de GPIO. Deze zullen we volgende week ook uitgebreid behandelen.

Het configureren van pinnen wordt meestal in software gedaan tijdens de configuratiefase (ook wel initialisatie of initialization genoemd). Het uitlezen, aanzetten of uitzetten van pinnen wordt in de actieve modus (ook wel active of loop genaamd) gedaan. Wanneer ik mijn uC ga programmeren zal ik bepaalde zaken maar 1 keer willen instellen, en andere zaken flexibel en continu (telkens opnieuw) willen aanpassen. Hiervoor maken we een hard onderscheid tussen de initialisatie modus (init) en de actieve modus (loop). Zie hiervoor figuur 5a en 5b.

In de linkerfiguur zien we een simpel flowchart waarin we 3 acties (blokken) hebben. We initialiseren, we togglen (aan/uitzetten) en we wachten. Daarnaast zien we een flow, namelijk de richting van de pijl. De flow bepaald in welke volgorde we acties uitvoeren. Voor meer uitleg over een



```

41  int main(void)
42  {
43      /* Initialisatie code wordt maar 1x uitgevoerd. Bijvoorbeeld het aanroepen
44       * van een speciale initialisatie functie */
45      SYSTEM_Initialize();
46
47
48      while(1)
49      {
50          /* Actieve code wordt telkens opnieuw uitgevoerd. Bijvoorbeeld het
51           * togglen (flippen) van PB3 */
52          IO_PB3_Toggle();
53          _delay_ms(100);
54      }
55  }
  
```

(a) Program Flow (flowchart) van een knipperend LEDje (PB3). (b) Bijbehorende code van een knipperend LEDje (PB3). De initialisatie van pin PB3 wordt in de functie `SYSTEM_INITIALIZE()` afgehandeld.

flowchart, kijk je in week 3 van basisprogrammeren (periode 1). In programmeren is de volgorde van acties extreem belangrijk. Wanneer we acties niet in de juiste volgorde doen, lopen we vaak het risico dat we hele andere uitkomsten krijgen dan we verwachten. Dit is heel erg vergelijkbaar met de rekenvolgorde bij wiskunde.

Rechts zien we het flowchart vertaald in code. We zien een main functie met daarin code die 1x uitgevoerd wordt (system initialize) en code die in een while loop (voor altijd) herhaald zal worden: toggle en delay. Het resultaat zal een afwisselend hoog (1) en laag (0) signaal zijn met een periodetijd van  $2 \times 100 \text{ ms} = 200 \text{ ms}$ . Dit betekent dat wanneer we deze pin aansluiten op een LED, deze LED met een frequentie van 5 Hz zal gaan knipperen. Voila, ons eerste programma! Dit programma, en een aantal andere programma's gaan we in het practicum van deze week maken.

Let op! Ook al gebruikt een flowchart net als een hardware schema blokken, het zijn twee totaal verschillende schema's! Een flowchart (zoals in figuur 5a) laat een algoritme zien en heeft altijd een richting (pijl). Het beschrijft een reeks aan acties die een processor moet ondernemen en is een abstractie van software. Een hardware schema (bijvoorbeeld figuur 4) is een abstractie van hardware en laat alle hardwarecomponenten als abstracte blokken zien, als ook de communicatie tussen deze hardware (verbindingen). Soms worden deze met richting aangegeven (richting van informatiestroom) maar vaak ook niet. Deze schema's mogen nooit door elkaar heen gebruikt worden!

24. Wat is een delay?

25. Hoe pas je dit toe in je code?

## 1.2 Practicum

### 1.2.1 MPLAB

Om te beginnen met programmeren sluit je de microcontroller aan op je PC door middel van de USB-kabel en start MPLAB X IDE. We gaan zien hoe je de LED kunt laten knipperen (de “Hello World” van microcontrollers). Maak een nieuw project aan (File → New Project (Ctrl + Shift + N)). Selecteer in het pop-up menu Microchip Embedded en Application Project. Klik nu op next. Selecteer de juiste uC: De AVR128DB48 (let op de DB, kies dus niet DA!). Onder tool kies je de USB-poort waarmee je je curiosity nano verbonden hebt. Deze zou automatisch zichtbaar moeten zijn. Zie je deze niet, laat dit dan even weten aan je docent. Kies niet No tool of simulator, in dat geval gaat je MPLAB je uC niet programmeren. Klik op next en in het volgende scherm selecteer je de laatste compiler versie van XC8 (op het moment van schrijven is dit XC8 V3.00). Klik op next en geef je project een handige naam. Tip: hanteer een week\_opdracht structuur. Bijvoorbeeld: week1\_LED. Hiermee maak je het voor jezelf makkelijker om aan het einde je projecten te vinden, wanneer je bijvoorbeeld opnieuw code wil bekijken of gebruiken. Zet een vinkje bij set as main project en haal het vinkje bij Open MCC on Finish weg.

Tip: In het verleden heb ik veel studenten gezien die hun oude code wegcommenten en dan verder gaan. Dit is geen goede coding practice. Sla je werk altijd op, en houdt het beschikbaar zodat je altijd opnieuw code kunt bekijken en gelijk kunt uitvoeren. In plaats van verder gaan in hetzelfde project met een nieuwe opdracht, maak je liever een nieuw project aan met een nieuwe herleidbare naam. Hiermee zorg je ervoor dat je gemakkelijk oude code terugvindt en kunt gebruiken. Je toekomstige ik is je dankbaar!

Wanneer je klaar bent met het opzetten van je project, genereert MPLAB een nieuw project met folderstructuur, zoals in figuur 6. Omdat we deze keer nog geen MCC gebruiken is er ook nog geen main.c voor ons gegenereerd. Deze gaan we nu aanmaken.

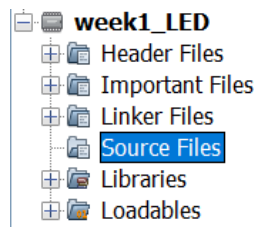


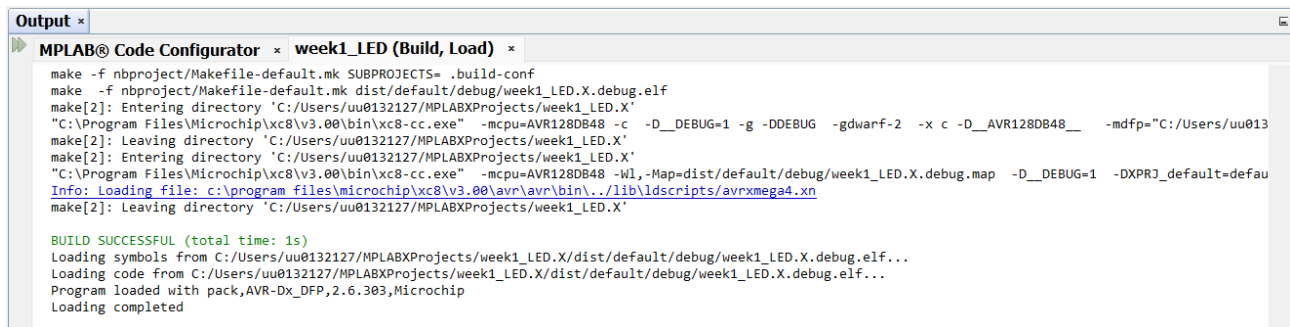
Figure 6: De projectstructuur van een nieuw project.

26. Maak een avr main.c en zorg ervoor dat deze onder Source Files staat. Noem deze main.c

We zouden de code al gelijk kunnen runnen, maar dan gebeurt er niet zoveel. Je runt de code door rechtsboven op het groene pijltje te klikken (run without debugging of Ctrl+Alt+F5). Als we dit doen zal de compiler proberen machinecode van ons programma te maken. Dit lukt alleen wanneer we syntactisch correcte code schrijven. Je krijgt dan figuur 7 te zien wanneer onze code succesvol is gecompileerd.

De code is dan syntactisch correct geschreven. Let op! Dit zegt nog niets over de werking van onze code. Als we een syntax fout (syntax error) maken, krijgen we de errormarker in figuur 8.

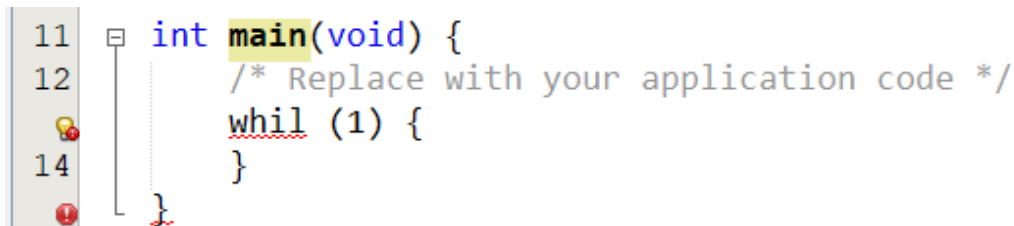
Let op! Ga nooit verder met code met syntax fouten. Los altijd eerst je (syntax) fouten op! Anders krijg je nog veel vervelender fouten. Bij syntaxfouten biedt de IDE soms een dialoogvenster aan die



```
Output x
MPLAB® Code Configurator x week1_LED (Build, Load) x
make -f nbproject/Makefile-default.mk SUBPROJECTS= .build-conf
make -f nbproject/Makefile-default.mk dist/default/debug/week1_LED.X.debug.elf
make[2]: Entering directory 'C:/Users/uu0132127/MPLABXProjects/week1_LED.X'
"C:\Program Files\Microchip\xc8\v3.00\bin\xc8-cc.exe" -mcpu=AVR128D848 -c -D__DEBUG=1 -g -DDEBUG -gdwarf-2 -x c -D__AVR128D848__ -mdfp="C:/Users/uu0132127/MPLABXProjects/week1_LED.X"
make[2]: Leaving directory 'C:/Users/uu0132127/MPLABXProjects/week1_LED.X'
make[2]: Entering directory 'C:/Users/uu0132127/MPLABXProjects/week1_LED.X'
"C:\Program Files\Microchip\xc8\v3.00\bin\xc8-cc.exe" -mcpu=AVR128D848 -Wl,-Map=dist/default/debug/week1_LED.X.debug.map -D__DEBUG=1 -DXPRJ_default=defau
Info: Loading file: c:\program files\microchip\xc8\v3.00\avr\avr\bin\..\lib\ldscripts\avrxmega4.xn
make[2]: Leaving directory 'C:/Users/uu0132127/MPLABXProjects/week1_LED.X'

BUILD SUCCESSFUL (total time: 1s)
Loading symbols from C:/Users/uu0132127/MPLABXProjects/week1_LED.X/dist/default/debug/week1_LED.X.debug.elf...
Loading code from C:/Users/uu0132127/MPLABXProjects/week1_LED.X/dist/default/debug/week1_LED.X.debug.elf...
Program loaded with pack,AVR-Dx_DFP,2.6.303,Microchip
Loading completed
```

Figure 7: De code is succesvol gebuild en gecompileerd.



```
11 int main(void) {
12     /* Replace with your application code */
13     while (1) {
14     }
```

Figure 8: Syntax Error!

vraagt of je een eerdere, werkende versie wil runnen. Doe dit nooit! Nog vervelender dan weten dat iets niet werkt, is niet weten dat iets niet werkt. Je zoekt je dan een ongeluk, omdat de uC niet doet wat je had verwacht. De IDE zal je altijd waarschuwen voor Syntaxfouten. Los deze dan ook altijd op!

Debuggen is een proces wat hier nog weer los van staat, debuggen gaat over het vinden en oplossen van functionele fouten in je code. In dat geval is de syntax wel correct, maar doet je programma niet exact wat het zou moeten doen. Dit soort fouten zijn veel lastiger te vinden. Je compiler zal hier meestal niet veel hulp in bieden. Debuggen is een (systematische) vaardigheid die je echt moet ontwikkelen.

Tip: veel studenten zie ik soms willekeurig dingen aanpassen, weghalen of toevoegen in hun code, totdat iets weer werkt. Dit is opzich niet verkeerd. Met een beetje experimenteren kun je een heel eind komen. Toch is dit voor debuggen vaak funest. De fout bevindt zich vaak op een plek waar je die niet verwacht. Wanneer je dan goed werkende code aanpast ben je nog verder van huis. Als programmeur moet je goed nadenken over wat de fout zou kunnen zijn. Hiermee kun je veel gerichter bugs smashes. Wanneer je dat doet leer je er ook nog eens meer van. Een goeie manier om dit te oefenen is om jezelf de tijd te gunnen om even na te denken over wat je doet. Heel goed werkt het als je bijvoorbeeld je buurvrouw/buurman vraagt even mee te kijken.

27. Wat is de fout in de code van figuur 8?

### 1.2.2 LED

We gaan beginnen met het simpelste programma wat we kunnen verzinnen, om te kijken of onze uC het überhaupt wel doet. We gaan de LED die op de curiosity nano zit aanzetten en daarna laten knipperen. Tijdens de voorbereiding heb je de LED van de curiosity nano bestudeerd. We hebben 2 instructies nodig om dit te doen. 1 voor het DIR register en 1 voor het OUT register. Je kunt deze registers vinden in het datasheet van de AVR128dB48 (kijk maar eens in hoofdstuk 18.4). Volgende week gaan we uitgebreid stil staan bij registers. Voor nu kun je een register zien als een speciale byte van het geheugen waarin specifieke instellingen van onze peripherals staan.

28. Wat doet het DIR register en wat doet het OUT register?
29. Geef voor beide registers aan of we deze in de initialisatiefase of in de loop aanpassen.
30. Welke specifieke instanties van het DIR en OUT register hebben we nodig, gezien de Port/pin waarop de LED zit?
31. Geef de 2 instructies waarmee je deze registers correct instelt zodat de LED aan gaat.
32. Programmeer deze instructies, run je code en zet de LED aan.

Naast de DIR en OUT registers hebben we ook DIRSET en OUTSET en hun tegenhanders (DIRCLR en OUTCLR). Deze registers kun je ook gebruiken om DIR en OUT aan te passen. In plaats van dat je een samengestelde operator gebruikt `PORTX.OUT |= 0000 0001`, gebruik je `PORTX.OUTSET = 0000 0001`. Dit is ook zo voor de DIRCLR, wat misschien een beetje verwarrend kan zijn. Daar gebruik je i.p.v. `PORTX.DIR & = ~`, `PORTX.DIRCLR =`. Let op! SET en CLR registers zijn een speciale extra feature van AVR uC's. Heel veel andere uC's hebben deze registers niet. Het is daarom belangrijk dat je ook primaire bitmanipulatie/masking begrijpt.

33. Welk van deze twee registers moet je aanpassen om de LED weer uit te zetten?
34. Op welke manier zouden we de LED kunnen laten knipperen?
35. Schrijf code waarmee je de LED 1x per seconde laat knipperen. Tip: je zult hier een extra library voor nodig hebben. Dit library heet `<util/delay.h>`.

Great success! We hebben nu geleerd hoe we de output van de uC kunnen instellen. Wat we voor de LED gedaan hebben kunnen we ook toepassen op alle andere pinnen van de uC.

36. Schrijf nu code waarmee je de 3 ledjes op het SMU bordje laat knipperen.



## 2 GPIO

### 2.1 Voorbereiding

Vorige week hebben we gezien hoe we een LED konden aansturen. Deze week gaan we de GPIO peripheral in meer detail bestuderen. De GPIO peripheral is een digitale-digitale peripheral. Dat betekent dat we een digitale signalen uit de buitenwereld omzetten naar een digitale waarde op onze uC en vice-versa. Een hardwareschema van deze peripheral op de uC is gegeven in figuur 9. Een digitaal signaal is een signaal wat slechts een 0 of een 1 kan zijn. Meestal wordt dit gerepresenteerd met een 0V (0) en een 3.3V of 5V (1). Een 0 noemen we ook wel laag, terwijl een 1 hoog wordt genoemd. Een analoog signaal daarentegen kan allerlei waardes aannemen. Zoals je misschien al wel wist, werkt een computer, en dus ook een uC op basis van nullen en enen. We zullen dus altijd een analoog signaal moeten digitaliseren voordat we er iets nuttigs mee kunnen doen. In week 5 gaan we hier dieper op in.

Omdat het per project zeer kan verschillen of je veel input nodig hebt, bijvoorbeeld bij een sensorsysteem, of juist veel output, in het geval van een LED-aansturing, hebben uC-fabrikanten bedacht om de I/O flexibel te maken. Dit betekent dat je de I/O zelf kunt instellen. Dit is extreem handig, want het geeft je de mogelijkheid om dezelfde uC voor tal van toepassingen te gebruiken. De ene keer zijn er 20 inputs voor 20 digitale sensoren. De andere 10 inputs en 10 outputs voor een robotaansturing. Door zijn flexibiliteit wordt I/O dus General Purpose IO genoemd.

Elke pin van de AVR128DB48 is individueel configureerbaar en adresseerbaar. Dat betekent dat ik elke pin apart kan instellen (initialize of configure), de waarde ervan aan kan passen (output: set of write), of mag uitlezen (input: get of read).

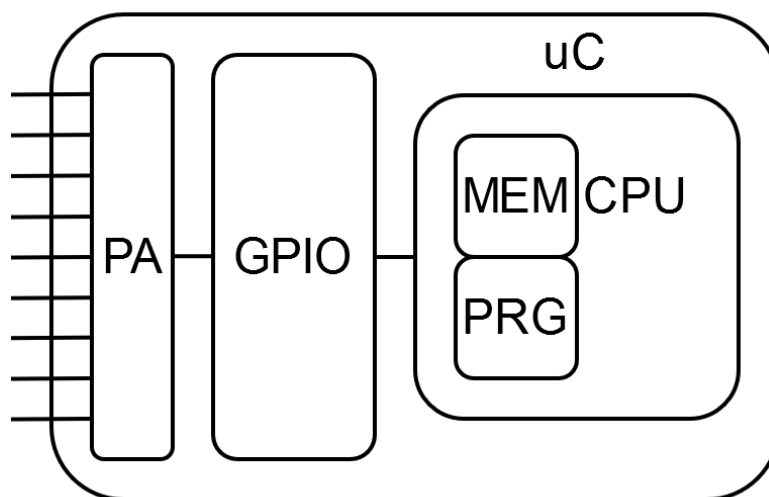


Figure 9: Hardware schema van een uC met GPIO. De uC bevat een central processing unit (CPU) die het programma (PRG) uitvoert dat wij kunnen programmeren. De GPIO (dit is een van de verschillende peripherals) stelt de verbinding naar verschillende GPIO-pinnen op de verschillende poorten (PA, PB, PC en PD) voor. Alle peripheral instellingen staan opgeslagen in geheugenregisters (MEM).

## 2.1.1 Registers

Wanneer we instellingen van peripherals (zoals de GPIO) willen configureren, doen we dit met behulp van registers. Vorige week hebben we ze al even kort gebruikt bij het programmeren van de LED. Een register is een speciale byte in het geheugen (maar kan bij sommige andere uC's ook andere groottes hebben) waarin specifieke instellingen staan. Een uC heeft honderden tot duizenden verschillende registers. Gelukkig hoeft je die meestal niet allemaal in te stellen, vaak hebben ze een standaardinstelling (met waarde 0000 0000) waarin ze niets doen of een peripheral uit laten. Door de juiste registers aan te passen, configureren en activeren we een peripheral. Dit doen we in software. Dat is heel handig, want het is erg eenvoudig dit in code uit te drukken. Daarnaast heb je ook registers die je bijvoorbeeld kunt uitlezen. De waarden daarvan wil je op hun beurt weer in je programma kunnen gebruiken.

### 18.7.2 Output Value

**Name:** OUT  
**Offset:** 0x01  
**Reset:** 0x00  
**Property:** -

Access to the Virtual PORT registers has the same outcome as access to the regular registers but allows for memory-specific instructions, such as bit manipulation instructions, which cannot be used in the extended I/O Register space where the regular PORT registers reside.

Bit	7	6	5	4	3	2	1	0
	OUT[7:0]							
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Figure 10: Een voorbeeld van een register. In dit geval het output register.

Kijk eens naar het register in figuur 10. We zien hier een snippet uit het datasheet (p.198) van het output register. Elke IO-poort (PORTA, PORTB, PORTC, PORTD, PORTE en PORTF) heeft een eigen output register waarmee we de output waarde van elke pin op die poort hoog of laag kunnen maken. Dit register staat op een speciale plek in het geheugen namelijk op address locatie:  $0x0400 + 0x01 = 0x0401$ .  $0x0400$  is het base address van deze poort.  $0x01$  is de offset (afstand vanaf de base, dus simpelweg erbij optellen) van het specifieke register OUT. Deze vind je in figuur 10 naast offset en varieert dus per register. Het handige is dat alle output registers dezelfde offset hebben. Dat betekent dat bijvoorbeeld PORTB (base:  $0x0420$ ) zijn output register op  $0x0421$  heeft staan. We kunnen dit register adresseren met behulp van een handige alias. In het geval van PORTA gebruiken we PORTA.OUT. Hierin hanteren we de structuur base.address.offset. Hiermee adresseer je het adres base + offset. Je mag dan de volgende regel code: `PORTA.OUT = 11110000` gebruiken. Dit zet de pinnen PA7 t/m PA4 op hoog en PA3 t/m PA0 op laag wanneer deze pinnen als output geconfigureerd zijn. Let op! We lezen in principe altijd van links (msb: most significant = grootste = PA7) naar rechts (lsb: least significant = kleinste = PA0). We beginnen dus ook niet op 1 te tellen! Alle base addresses staan in een heel handig .h-bestand, namelijk `ioavr128db48.h`. In figuur 11 zie je een snippet uit dit zeer uitgebreide bestand.

Je zult misschien nog wel de #define herinneren van programmeren basis. De define is een simpele preprocessor syntax. Voordat de C-code gecompileerd en daarna uitgevoerd wordt, worden alle defines in de code vervangen door de waarde die erachter staat. Dit betekent dat bijvoorbeeld `AC0` vervangen wordt door `(* (AC.t *) 0x0680`. Op die manier worden dus de juiste registers aangeroepen. De reden dat we die aliases zoals `AC0` of `PORTA` gebruiken, is om onze code makkelijk schrijf- en leesbaar te houden. Toch is het wel belangrijk om te weten dat dit dus onderwater gebeurt. Wanneer

```

3376 #define BOD      (*(BOD_t *) 0x00A0) /* Bod interface */
3377 #define VREF      (*(VREF_t *) 0x00B0) /* Voltage reference */
3378 #define MVIO      (*(MVIO_t *) 0x00C0) /* Multi-Voltage I/O */
3379 #define WDT      (*(WDT_t *) 0x0100) /* Watch-Dog Timer */
3380 #define CPUINT     (*(CPUINT_t *) 0x0110) /* Interrupt Controller */
3381 #define CRCSKAN   (*(CRCSKAN_t *) 0x0120) /* CRCSKAN */
3382 #define RTC      (*(RTC_t *) 0x0140) /* Real-Time Counter */
3383 #define CCL      (*(CCL_t *) 0x01C0) /* Configurable Custom Logic */
3384 #define EVSYS     (*(EVSYS_t *) 0x0200) /* Event System */
3385 #define PORTA     (*(PORT_t *) 0x0400) /* I/O Ports */
3386 #define PORTB     (*(PORT_t *) 0x0420) /* I/O Ports */
3387 #define PORTC     (*(PORT_t *) 0x0440) /* I/O Ports */
3388 #define PORTD     (*(PORT_t *) 0x0460) /* I/O Ports */
3389 #define PORTE     (*(PORT_t *) 0x0480) /* I/O Ports */
3390 #define PORTF     (*(PORT_t *) 0x04A0) /* I/O Ports */
3391 #define PORTMUX   (*(PORTMUX_t *) 0x05E0) /* Port Multiplexer */
3392 #define ADC0      (*(ADC_t *) 0x0600) /* Analog to Digital Converter */
3393 #define AC0       (*(AC_t *) 0x0680) /* Analog Comparator */
3394 #define AC1       (*(AC_t *) 0x0688) /* Analog Comparator */
3395 #define AC2       (*(AC_t *) 0x0690) /* Analog Comparator */
3396 #define DAC0      (*(DAC_t *) 0x06A0) /* Digital to Analog Converter */
3397 #define ZCD0      (*(ZCD_t *) 0x06C0) /* Zero Cross Detect */
3398 #define ZCD1      (*(ZCD_t *) 0x06C8) /* Zero Cross Detect */
3399 #define ZCD2      (*(ZCD_t *) 0x06D0) /* Zero Cross Detect */
3400 #define OPAMP     (*(OPAMP_t *) 0x0700) /* Operational Amplifier System */
3401 #define USART0    (*(USART_t *) 0x0800) /* Universal Synchronous and Asynchronous Receiver and Transmitter */
3402 #define USART1    (*(USART_t *) 0x0820) /* Universal Synchronous and Asynchronous Receiver and Transmitter */
3403 #define USART2    (*(USART_t *) 0x0840) /* Universal Synchronous and Asynchronous Receiver and Transmitter */
3404 #define USART3    (*(USART_t *) 0x0860) /* Universal Synchronous and Asynchronous Receiver and Transmitter */
3405 #define USART4    (*(USART_t *) 0x0880) /* Universal Synchronous and Asynchronous Receiver and Transmitter */
3406 #define TWI0      (*(TWI_t *) 0x0900) /* Two-Wire Interface */
3407 #define TWI1      (*(TWI_t *) 0x0920) /* Two-Wire Interface */
3408 #define SPI0      (*(SPI_t *) 0x0940) /* Serial Peripheral Interface */
3409 #define SPI1      (*(SPI_t *) 0x0960) /* Serial Peripheral Interface */
3410 #define TCA0      (*(TCA_t *) 0x0A00) /* 16-bit Timer/Counter Type A */
3411 #define TCA1      (*(TCA_t *) 0x0A40) /* 16-bit Timer/Counter Type A */
3412 #define TCB0      (*(TCB_t *) 0x0B00) /* 16-bit Timer Type B */
3413 #define TCB1      (*(TCB_t *) 0x0B10) /* 16-bit Timer Type B */

```

Figure 11: Een snippet uit ioavr128db48.h. PORTA is gehighlighted.

je goed bent in pointer arithmetic, zou je dit onderliggende systeem zelfs kunnen gebruiken om zeer efficiënte code te schrijven. Zover gaan we gelukkig niet in dit vak.

Je zult misschien even schrikken van al deze syntax, diepgaande uitleg en pointertovernarij. Laat je daardoor niet te snel uit het veld slaan. Je hoeft niet alles te begrijpen. Ook zul je bepaalde dingen pas gaan begrijpen wanneer je ze een keer gedaan hebt. Tijdens uC-programmeren gaan we gebruik maken van tools die ons helpen deze registers in te stellen. Toch is het wel goed om te weten wat die tools op de achtergrond voor ons doen.

## 2.1.2 Bytes & Bitwise Operators

Een bit is een 1 of een 0. Een byte is een reeks van 8 bits met enen of nullen. Een voorbeeld van een byte is 00011101. We zagen in het vorige gedeelte dat het register PORTA.OUT verantwoordelijk is voor alle outputwaardes van alle pinnen PA7 t/m PA0. Heel vaak willen we maar een of enkele bits tegelijkertijd aanpassen. We willen bijvoorbeeld PA2 aanpassen, terwijl alle andere pinnetjes ongemoeid laten. Hoe doen we dat? Daarvoor gebruiken we vaak bitmasks om een specifiek gedeelte aan te passen. Doen we dit niet goed, dan pas je dus zomaar de andere pinnetjes aan. De ervaring leert dat dit iedereen wel eens overkomt, en heel vaak de bron van bugjes zijn.

37. Wat zou er fout kunnen gaan als we op een verkeerde manier een byte gaan aanpassen?

De juiste manier om registers aan te passen is met behulp van bitmanipulatie ook wel bitmasking genoemd. Bitmasking wordt gedaan met een aantal verschillende bitgewijze operatoren (bitwise

operators). Dit zijn:  $=$ ,  $\sim$ ,  $|$ ,  $\&$ ,  $\wedge$ , en deze heb je geleerd bij programmeren basis (week 6) in periode 1. Probeer even goed voor jezelf duidelijk te maken wat het verschil tussen deze operators is, en bijvoorbeeld  $!$ ,  $||$ ,  $\&\&$ .

38. Leg voor de volgende operators uit wat ze doen:  $=$ ,  $\sim$ ,  $|$ ,  $\&$ ,  $\wedge$

39. Leg uit wat deze operators anders maakt dan  $!$ ,  $||$ ,  $\&\&$ . Tip gebruik een voorbeeld op 11 en 01.

Stel we hebben de volgende byte: 1100 1101

40. Wat is de uitkomst van  $1100\ 1101\ | \ 1111\ 0000$ ?

41. Wat is de uitkomst van  $1100\ 1101\ || \ 1111\ 0000$ ?

42. Wat is de uitkomst van  $1100\ 1101\ | \ \sim(1111\ 0000)$ ?

43. Wat is de uitkomst van  $1100\ 1101\ | \ !(1111\ 0000)$ ?

44. Wat is de uitkomst van  $1100\ 1101\ \& \ 0000\ 0000$ ?

45. Wat is de uitkomst van  $1100\ 1101\ \& \ 1111\ 1111$ ?

46. Wat is de uitkomst van  $1100\ 1101\ \&\& \ 1111\ 1111$ ?

47. Stel we willen 1 bit hoog maken in een byte, maar geen andere bits veranderen. Hoe zouden we dat aan kunnen pakken? Welke operators kun je daarvoor gebruiken? Illustreer je antwoord met een voorbeeld.

48. Beantwoord dezelfde vraag voor het laag maken ( $= 0$ ) van een bit in een byte.

Heel vaak zien we de volgende constructie (pseudocode) met samengevoegde operators:

Listing 1: Snippet (pseudocode) van Bitwise masking

```
a = 1110 1000;  
b = a;  
a = a | 0001 0000;  
b |= 0001 0000;
```

49. Is er een verschil tussen de laatste twee operaties van de code in snippet 7? Zijn a en b hetzelfde?

50. Geef de samengevoegde operator om 1 bit hoog te maken.

51. Geef de samengevoegde operator om 1 bit laag te maken.

52. Geef de samengevoegde operator om 1 bit te togglen. (0 wordt 1 en 1 wordt 0)

Alle bovengenoemde operators kunnen ook gebruikt worden om meerdere bits tegelijkertijd te veranderen. In dat geval gebruik je in plaats van 0000 0001 (doet iets met bit0) bijvoorbeeld 0110 0001 (doet iets met bit 6, bit 5 en bit 0).

Wanneer we een bit reeks gebruiken in C kunnen we dat niet doen met alleen maar nullen of enen. De compiler zou dit namelijk als decimaal getal interpreteren. De bovenstaande bitreeksen moet je dus expliciet als bitreeks benoemen, zodat je compiler dit snapt. Hiervoor gebruik je 0b. Bijvoorbeeld 0b00001010. Dit geeft de bitreeks correct door. In plaats van een bit reeks mogen we ook een hex (hexadecimaalgetal) gebruiken. Hiervoor gebruik je 0x. Daar zou de vorige reeks gelijk zijn aan 0x0A. Toch is het vaak gebruikelijker (en makkelijker voor de leesbaarheid) om een alias te gebruiken. We gaan zien tijdens het practicum dat voor alle primaire bitreeksen (waarbij er maar een 1 is en de rest nullen) er aliases zijn.

## 2.2 Practicum

### 2.2.1 Knop

De LED'je laten knipperen in week 1 was natuurlijk leuk, maar we willen meer! Nu zouden we graag input van de gebruiker willen. Een eenvoudige manier van input is de knop op de curiosity nano. Om de knop te kunnen gebruiken moet je wel de knop goed instellen. We moeten hiervoor opnieuw gaan kijken in het datasheet. Navigeer eens naar p.176.

53. Welke registers moet je instellen om de knop uit te lezen?
54. In welk register verandert iets wanneer de knop is ingedrukt/weer losgelaten?
55. Wat is de bitwaarde bij indrukken en wat is de bitwaarde bij loslaten?
56. Schrijf code waarmee je de knop kunt uitlezen.

We hebben een probleem. Je gaat er misschien van uit dat je de knop goed ingesteld hebt, maar hoe weet je dit eigenlijk? We kunnen het niet zien! Je ziet spanning niet, en al helemaal niet een register wat ergens in een geheugen zit. Dit is niet heel handig. Het is heel belangrijk dat je jezelf ontdoet van deze beperking en begint met het zichtbaar maken van de dingen die je programmeert. In het geval van een knop kun je twee dingen doen. Of we kunnen een output activeren, zoals de LED, om zichtbaar te maken dat we de knop ingedrukt hebben, of we kunnen de debug functionaliteit van de uC gebruiken om toch mee te kijken in het geheugen. We beginnen eerst met de eerste mogelijkheid, en zullen je straks de tweede laten zien.

We kunnen een if else statement gebruiken om te kijken of de knop is ingedrukt. Hiervoor gebruiken we de volgende boolean: `PORTB.IN & PIN2_bm`.

57. Leg uit hoe deze boolean kan bepalen of de knop ingedrukt is of niet.
58. Schrijf code waarmee je op basis van de knop de LED kan aansturen. Wanneer (if) de knop ingedrukt is, is de LED aan, en anders (else) uit. Zie figuur 12 voor het flowchart van je programma.
59. Run je code en bekijk of dit werkt.

Wellicht merkte je dat de knop een beetje raar functioneert wanneer je hem hebt ingedrukt. De knop werkt soms maar 1 keer en daarna blijft hij wat hangen. Soms na een tijdje doet hij het weer. Dit is een elektronisch probleem. De knop zweeft, wat wil zeggen dat het niet duidelijk is of de knop hoog of laag is, maar er ergens een beetje tussenin hangt. Wanneer we de knop indrukken, wordt de knop duidelijk verbonden met GND. Echter wanneer we de knop loslaten, wordt de knop niet zomaar uit zichzelf 3.3V. Om dit probleem op te lossen, moeten we een pull weerstand toevoegen. In dit geval een pull-up, omdat we hem omhoog trekken (naar de 3.3V). Een pulldown weerstand zou hem omlaag trekken naar de 0V (en dus niet werken voor onze knop).

60. Ga naar p.192 van het datasheet en bekijk de uitleg van het `PINCTRL` register.

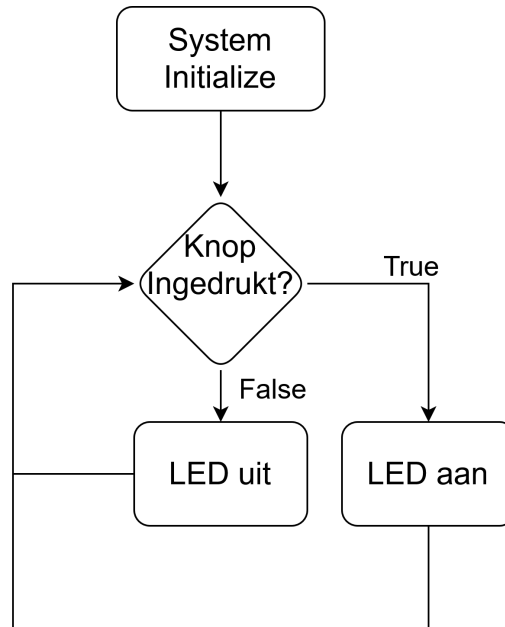


Figure 12: Een flowchart van een simpel aan uit schakelaar. De LED is aan als de knop is ingedrukt. De LED is uit als de knop niet is ingedrukt.

61. Wat doet dit register allemaal? Is dit register bedoeld voor een hele poort, of voor een individuele pin?
62. Geeft de naam van het ctrl register voor de pin aan de knop van de curiosity nano.
63. We kunnen door 1 bitje hoog te zetten de pull weerstand aanzetten. Geef deze regel code.
64. Voeg deze regel code toe op de juiste plaats (initialisatie/active) in je code en run deze opnieuw. Bestudeer het gedrag van het knopje opnieuw. Zorg ervoor dat je zeker weet dat de knop niet meer zweeft.

De manier waarop je de knop hebt geprogrammeerd noemen we polling. Dit is niet heel efficient.

65. Ga op zoek naar het begrip polling en leg uit wat daarmee bedoeld wordt.
66. Waarom is polling niet heel efficiënt/slim om te doen.?
67. Hoe zouden we dit slimmer aan kunnen pakken? Leg uit. Misschien kun je dit ook illustreren aan de hand van een voorbeeld uit je dagelijkse omgeving?
68. Welk (veelvoorkomend) onderwerp (peripheral) binnen uC-programmeren zou dit voor ons kunnen doen?

## 2.2.2 Debugging

De andere manier om te bekijken wat de waarde van de knop is met behulp van de debugging features van de curiosity nano. Veel ontwikkelbordjes bieden deze features aan, zodat jij als ontwikkelaar eenvoudiger het gedrag van de uC kunt bestuderen. Je kunt je namelijk wel voorstellen dat een ledje gebruiken om gedrag te bestuderen alleen maar in erg simpele gevallen werkt. De debugmode is een speciale run modus van de curiosity nano. Hiermee wordt de debugchip geactiveerd en kan deze intern meekijken in de processor en deze pauzeren. We kunnen in principe gebruik maken van twee hele handige debug features: pauzeren op een bepaalde plek in de code, dit wordt ook wel breakpoint genoemd en het meekijken naar de waarde van variabelen. Dit wordt ook wel watch expression genoemd. Zie figuur 13 als voorbeeld.

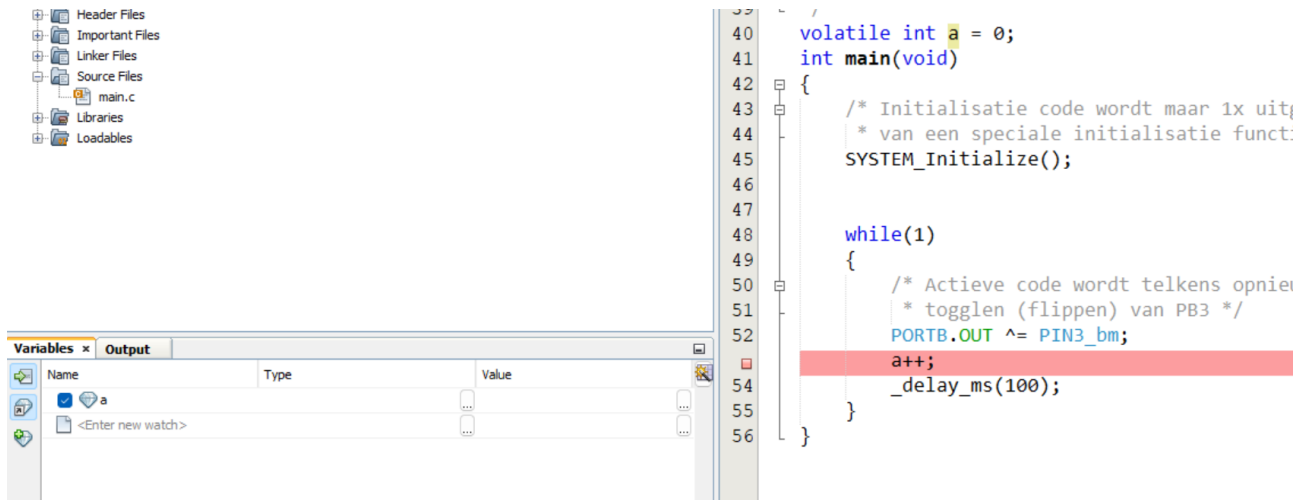


Figure 13: Een breakpoint (rode blokje op lijn 53) in de code. We hebben een watch expression toegepast op variabele a. De waarde van a is links zichtbaar, wanneer we de code runnen.

Waarom zou je je code willen pauzeren? Soms weten we niet of we op een bepaalde plaats in de code komen. Bijvoorbeeld wanneer we een complex if else statement hebben waarbij we willen controleren of we aan een conditie voldaan hebben. Door een breakpoint toe te voegen pauzeert ons programma op die plek. Daarmee weet je automatisch dat je dus aan die conditie voldaan hebt. Wanneer je in debug mode zit kun je nu je code laten stoppen wanneer je de knop hebt ingedrukt. Let op dit is een andere mode dan dat we tot nu toe onze code uitgevoerd hebben. Je kunt debuggen starten door op de knop debug main project te klikken in het lint aan de bovenkant van MPLAB.

69. Zoek uit hoe je een breakpoint kunt toevoegen en voeg deze toe op de regel dat we onze LED aanzetten en op de regel dat we onze LED uitzetten.
70. Plaats een breakpoint en run je code in debugmode. Druk op de knop en bekijk of de code pauzeert.

Wanneer je je code weer verder wil laten gaan, dien je op de continue knop te drukken. Het is mij opgevallen dat dit vaak vergeten wordt. Je uC doet niets totdat het weer instructie krijgt om door te gaan.

We willen wel eens weten hoe vaak we op de knop gedrukt hebben. Hiervoor gaan we een speciale debugvariabele aanmaken.



71. Declareer een volatile int a = 0; in je initialisatiefase. We gebruiken een volatile om ervoor te zorgen dat de compiler onze variabele niet wegoptimaliseert.
72. Laat a elke keer 1 optellen wanneer we de knop ingedrukt hebben.
73. Verplaats het breakpoint van het indrukken naar de plek waar a optelt.

We kunnen nu een watch expression plaatsen op a. Hiermee kunnen we nadat de processor pauzeert de waarde van a bekijken in het variabelenscherm (zie figuur 13 links onderin).

74. Plaats de watch expression door de variabele a te selecteren en met de rechtermuisknop erop te klikken. Kies new watch. Een alternatief is gebruikmaken van de hotkey Ctrl + Shift + F9. Check dat de variabelenscreen zichtbaar is inclusief deze variabele.
75. Voer je code uit in debugmode en druk de knop in. Bekijk a.
76. Druk op continue en druk opnieuw op de knop. Bekijk a opnieuw.

We hebben nu geleerd hoe we kunnen debuggen. Gebruik dit in je voordeel! Je kunt hiermee heel goed bugs opsporen in je code. Zeker wanneer je bijvoorbeeld een flowchart maakt zoals in figuur 12, kun je heel goed stap voor stap verifiëren of het programma daadwerkelijk een bepaalde actie in de juiste volgorde uitvoert.

### 2.2.3 MCC

Wat we tot nu toe hebben gedaan is een project op registerniveau vanaf 0 programmeren. We hebben gezien hoe we registers kunnen aanschrijven en daarmee de configuratie, input en output van de GPIO konden aanpassen. Wat je wellicht hebt gemerkt, is dat het lastig is om alles in een keer goed te doen. Het is gemakkelijk om een bepaald register, zoals de PINXCTRL met de pullup weerstand over het hoofd te zien. Voor een LED'je/ knopje is dat nog niet zo'n probleem, maar als we straks met peripherals zoals timers en interrupts gaan werken wordt dat een stuk lastiger. Deze peripherals hebben 3-10 verschillende registers waarbij deze register ook nog eens meerdere instellingen bevatten. Als er dan iets misgaat, is het een stuk lastiger om uit te vogelen wat je precies mist.

Daarnaast kan het snel onoverzichtelijk worden wanneer we alles in de main functie plaatsen. Gelukkig heeft Microchip hiervoor een GUI (Graphical User Interface) ontwikkeld wat je helpt om deze peripherals goed en gestructureerd te programmeren. Deze GUI heet MCC (Microchip Code Configurator) en is bedoeld om de initiatie/configuratie van de code te doen. Deze tool genereert op basis van gekozen instellingen de juiste registermanipulaties. Deze worden heel handig in je project geïmporteerd doordat we in onze main de functie `System.Initialize()` aanroepen. Hiermee wordt het configureren een stuk makkelijker, en houdt je je code een stuk overzichtelijker.

Vanaf nu gaan we voornamelijk de instellingen/configuratie met deze tool doen. Dat wil niet zeggen dat je vanaf nu geen registers meer gaat zien. We gaan continu bestuderen wat MCC ons aanlevert. Mocht je het toch interessant vinden om meer te leren over registermanipulatie, of je eigen libraries te willen schrijven, dan is dat uiteraard toegestaan. In de profilering Embedded Systems and Wireless Communications gaan we dit registerniveau programmeren verder toepassen.

MCC is een plugin tool die in MPLAB zit. Er is overigens ook een standalone versie beschikbaar. We gebruiken de plugin. Deze start automatisch wanneer je een nieuw project maakt en het vinkje: "start MCC after finish" laat staan. Helaas zijn de programmeurs van MCC niet heel erg gericht geweest op efficiënte code, dus het is nogal traag in het opstarten. Je kunt MCC altijd opstarten door op het blauwe hexagon met MCC in het lint aan de bovenkant van MPLAB te klikken.

77. We gaan de code voor de knop geschakelde LED opnieuw maken in MCC. Maak een nieuw project en zorg ervoor dat je deze keer het vinkje laat staan bij "start MCC after finish". Kies weer een handige naam voor je project.

Wanneer MCC is opgestart krijg je het volgende scherm te zien, zie figuur 14.

De eerste keer dat je MCC opstart zul je waarschijnlijk even overdonderd zijn door de hoeveelheid subschermpjes die tegelijkertijd opstarten. Dit is helaas een onhandige ontwerpkeuze van Microchip geweest. Zo bevatten meerdere windows dezelfde informatie en is de hoeveelheid zo hoog dat ik zelfs op mijn grote beeldscherm thuis amper de windows kan lezen. We hopen dat Microchip dit ooit leest, en haar leven betert. Tot die tijd moeten we dealen met de onhandige keuzes van anderen.

De windows die je allemaal ziet zijn:

- **Project Resources:** Hier zie je alle peripherals die op dit moment geconfigureerd worden voor je project. Heel belangrijk hier is de generate knop. Pas wanneer je daar op klikt zal er de code voor je gegenereerd worden.

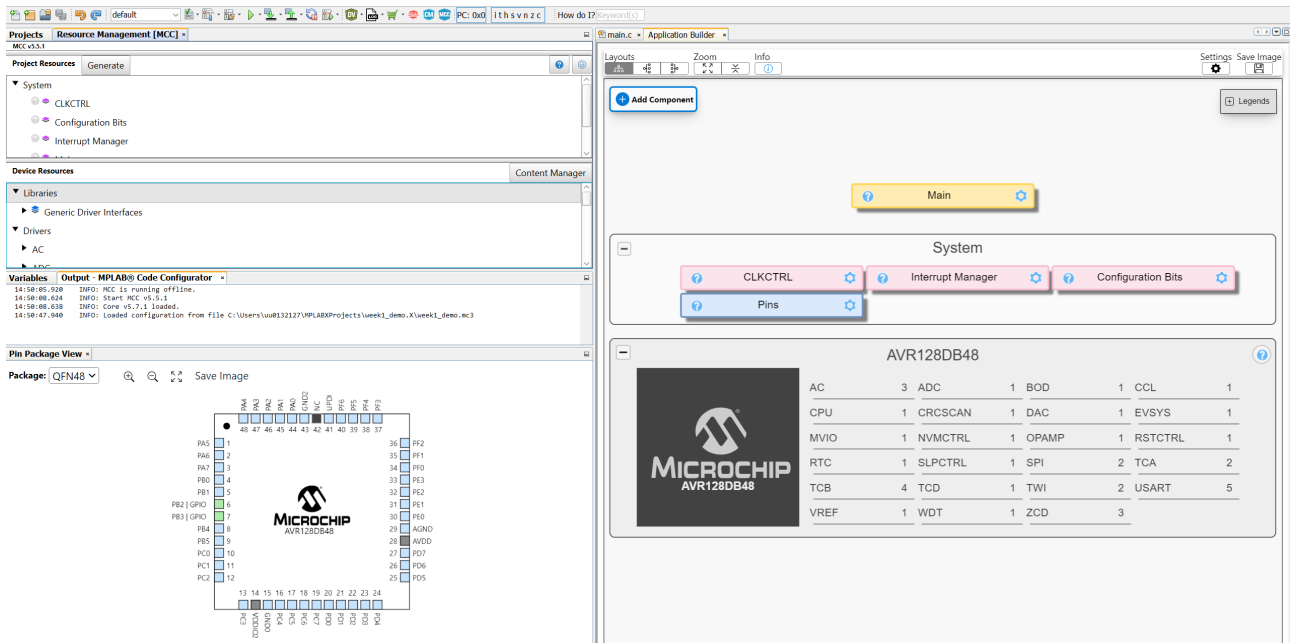


Figure 14: Microchip Code Configurator (MCC). We zien vier windows links, van boven naar beneden Project Resources, Device Resources, Terminal Output en Pin Package View en een window rechts Application Builder

- **Device Resources:** Hier staan alle peripherals die configureerbaar zijn voor onze device (AVR128DB48).
- **Terminal Output:** Hier staan soms informatieve berichten
- **Pin package view:** Hier staan alle pins van de uC. Je kunt ook zien welke pins je waarvoor gebruikt. In figuur 14 zie je dat ik PB2 en PB3 toegekend heb aan de GPIO.
- **Application Builder:** Hier zie je net als bij Project Resources alle peripherals die op dit moment geconfigureerd worden. Ook zien we net als bij device resources alle peripherals die op onze uC beschikbaar zijn.

Helaas zijn dit nog niet alle schermen die kunnen openen. Wanneer we op Pins in de Application Builder klikken, en op pin grid view onderaan het scherm krijgen we nog twee schermen extra, wat ook pijnlijk wordt in figuur 15.

- **Peripheral Editor:** Elke peripheral heeft een eigen configurator waarin (bijna) alle instellingen m.b.t. die peripheral ingesteld kunnen worden
- **Pin grid view:** Laat op een alternatieve manier alle pins zien. Deze is nodig voor een aantal acties die we uit willen voeren.

Als klap op de vuurpijl kunnen we de individuele windows wel slepen en op een iets logischer manier docken dan dat Microchip verzonnen heeft, deze instellingen worden helaas elke keer weer weggegooid wanneer we MPLAB opnieuw opstarten. We moeten er dus helaas echt mee dealen. Gelukkig zijn daar nog wel wat tools voor. Het kan bijvoorbeeld soms handig zijn om een scherm te maximaliseren. Doe dit door dubbel te klikken op de balk bovenin een scherm. Wanneer je opnieuw dubbelklikt reset de view weer terug naar de vorige.

Genoeg geklaagd! We gaan nu PB2 en PB3 als respectievelijk input en output pins configureren.

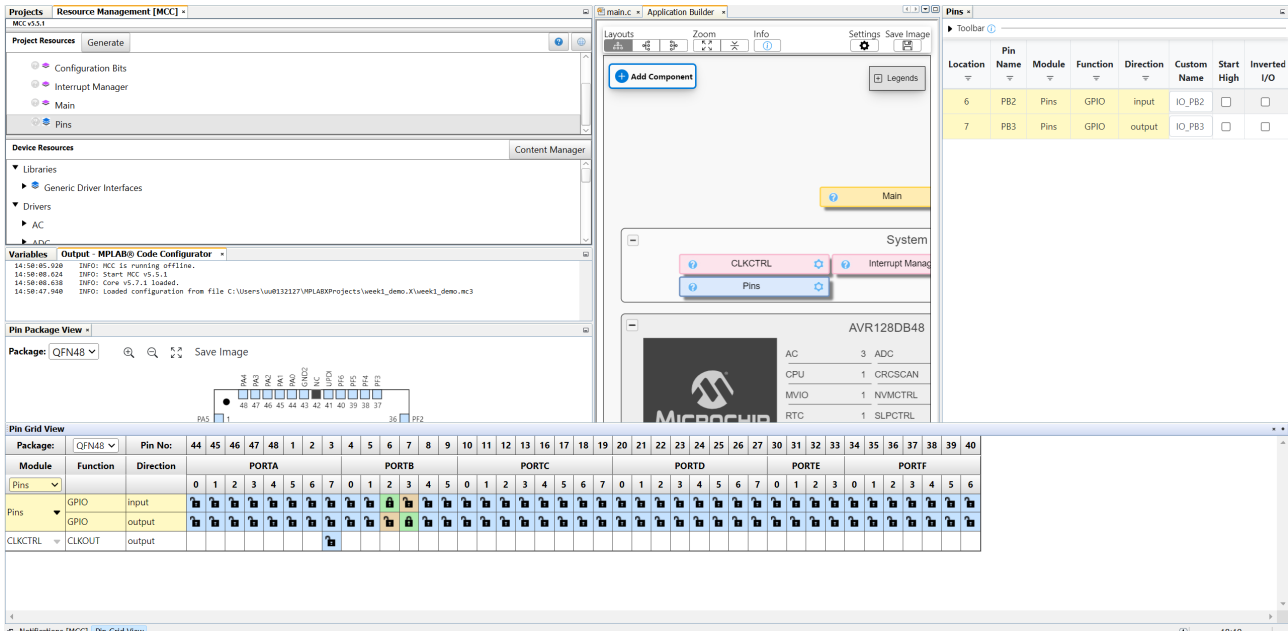


Figure 15: Screens!

78. Open de pin grid view en klik op de juiste hokjes zodat PB2 een input wordt en PB3 een Output.
79. Open de Pins view (door te klikken op Pins). Je ziet als het goed is nu hetzelfde als in figuur 15. Let op! Sommige instellingen zijn niet zichtbaar, je kunt het scherm slepen of dubbelklikken om ze zichtbaar te maken!
80. Configureer bij PB2 een pullup weerstand
81. Hernoem PB2 naar SW0 en hernoem PB3 naar LED0.
82. Klik op generate in het Project Resources scherm.

MCC heeft nu voor ons een hele hoop dingen onder water gedaan. Heel veel instellingen staan nu op de standaardconfiguratie, daar maken we ons op dit moment niet druk om. Daarnaast heeft MCC voor ons beide pinnen correct geconfigureerd en heeft het zelfs gratis en voor niets een library geschreven met een hele hoop functies waarmee we in de active mode de waarden kunnen aanpassen. Deze functies hebben de vorm: naam functienaam(). Bijvoorbeeld SW0.GetValue(). Heel handig!

Kijk eens rond in de bestanden die nu allemaal gegenereerd zijn (figuur 16).

83. Open pins.c en bestudeer de inhoud. Wat zie je allemaal. Beschrijf hoe registers geïnitieerd zijn en bekijk ook welke functies er allemaal gegenereerd zijn.
84. Hoe heet de functie waarmee we de LED0 van waarde kunnen veranderen?
85. Waar wordt de pullupweerstand geconfigureerd voor SW0?

Al deze functies zijn aanroepbaar vanuit main.c.

86. Kopieer de code uit de while lus (active mode) van je eerste project, en plak deze in de while lus van je huidige project. De configuratie moet je niet kopiëren!

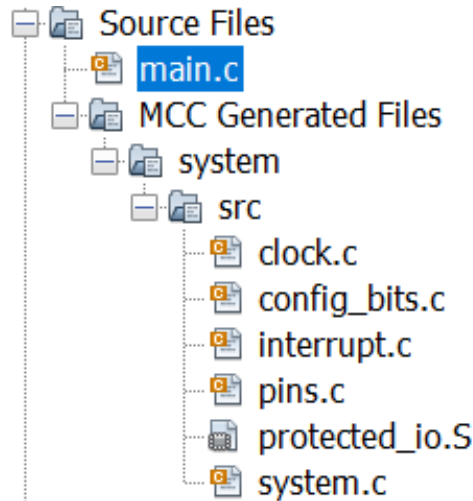


Figure 16: Gegeneerde bestanden door MCC. Elke peripheral krijgt een aparte .c. Er is ook een .h gegeneerd als header. Deze staan onder Header files.

87. Vervang de register aanpassingen door de juiste functieaanroepen.

88. Run je nieuwe project en stel vast dat deze hetzelfde gedrag vertoont als eerst.

Tip: Soms weet je even niet meer de naam van een functie. MPLAB beschikt over codecompletion. Wanneer je wat intypt en je drukt op Ctrl+Spatie, krijg je een lijst van meest voor de handliggende suggesties. Wanneer je bijvoorbeeld SW0 intypt en daarna Ctrl+Spatie, krijg je alle functie die op SW0 van toepassing zijn. Handig!

De structuur die MCC voor ons gecreëerd heeft is vrij recht-toe-recht aan. Dat is erg prettig. We kunnen dit bekijken door op de `system_initialize` functie met de rechtermuisknop te klikken, navigeren en dan go to declaration te gaan (zie figuur 17 of gebruik hotkey Ctrl+B). Hiermee gaan we naar de implementatie of declaratie van deze functie. Dat maakt het erg eenvoudig om te checken waar bepaalde code staat en wat het doet.

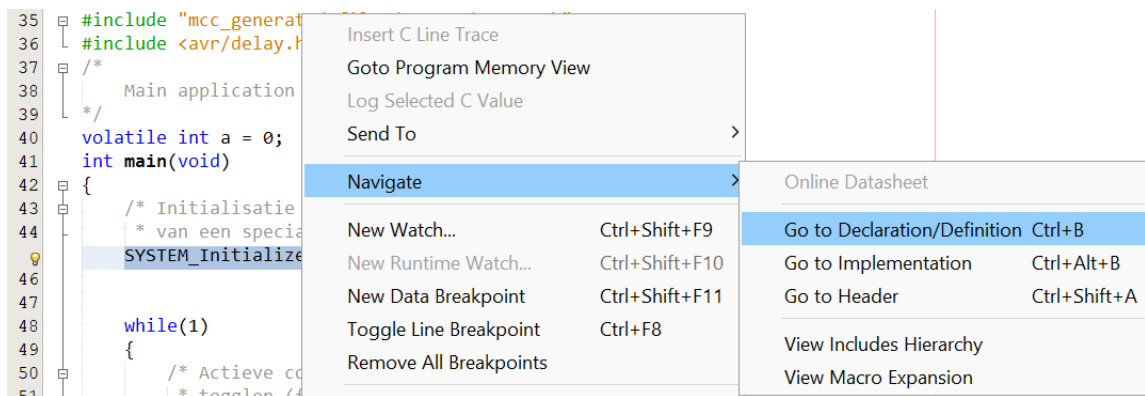


Figure 17: Zo kun je navigeren naar de declaratie/definitie van een functie.

89. Ctrl + B eens een aantal keer totdat je bij pins.c uitkomt.

90. Coding challenge: Schrijf op 1 regel code voor het aan/uitzetten van de led o.b.v. de knop. Tip: gebruik ook de functies die gegeneerd zijn.

## 3 Interrupts

### 3.1 Voorbereiding

Het doel van deze week is om een interrupt te kunnen programmeren en te gebruiken. Je begrijpt waarom een interrupt nuttig is, en kunt deze voor verschillende situaties inzetten. Ook leer je hoe je ervoor kunt zorgen dat je processor slaapt wanneer er niets te doen is. Hiermee bespaar je energie en maak je je systeem efficiënter en kan het langer meegaan. Na dit practicum verwachten we dat je interrupts en sleep modes gebruikt wanneer je je robot programmeert.

#### 3.1.1 Interrupts

Voor veel processen binnen onze uC weten we niet precies wanneer we een actie moeten laten plaatsvinden. Kijk maar eens naar figuur 18. We willen de actie 'Doe iets' uitvoeren, maar weten niet zo goed wanneer. Hoe weten we wanneer dit moment daar is? We zouden dit kunnen gaan meten.

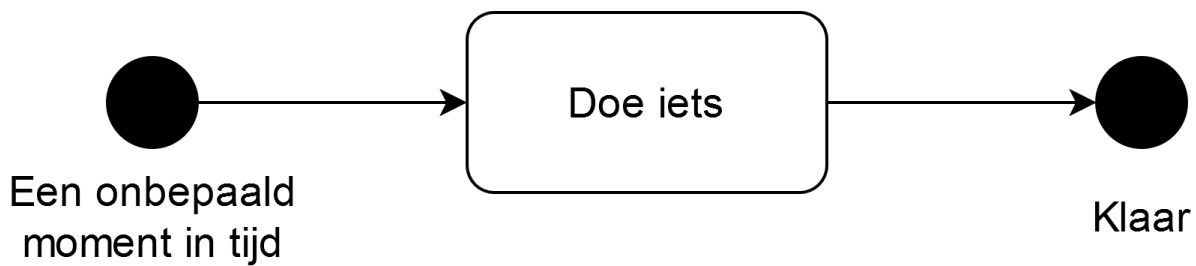


Figure 18: De actie doe iets moet op een van te voren onbekend moment in tijd uitgevoerd worden. Hoe weten we wanneer dit is?

91. Stel we wachten op een pakketje van Ali Express. Stel we zouden geen deurbel of telefoon hebben, hoe meten we dan of het pakketje er al is?
92. Benoem twee problemen met deze manier van meten
93. Leg eens uit hoe je deurbel ervoor zorgt dat we die twee problemen oplossen.

Het verschil tussen het continu checken en de deurbel is hetzelfde verschil als polling versus interruptbased. Kijk maar eens naar figuur 19. Hierin wordt gevisualiseerd wat het verschil tussen een polling systeem is en een interruptbased systeem.

94. Bekijk beide systemen en vergelijk ze met het voorbeeld van het pakketje van Ali Express.

Wat een grote tekortkoming van software algoritmes is, is dat ze niet geschikt zijn om om te gaan met onverwachte gebeurtenissen. Wanneer we niet weten of en wanneer iets gebeurt, kan het zeer lastig zijn om te bepalen of we iets moeten doen. Zoals we eerder zagen, kunnen we inbouwen dat we eindeloos lang gaan checken of iets al gebeurt is, maar dit zorgt ervoor dat we onze processorcapaciteit verspillen en heeft ook nog eens als risico dat we te laat erachter komen, omdat

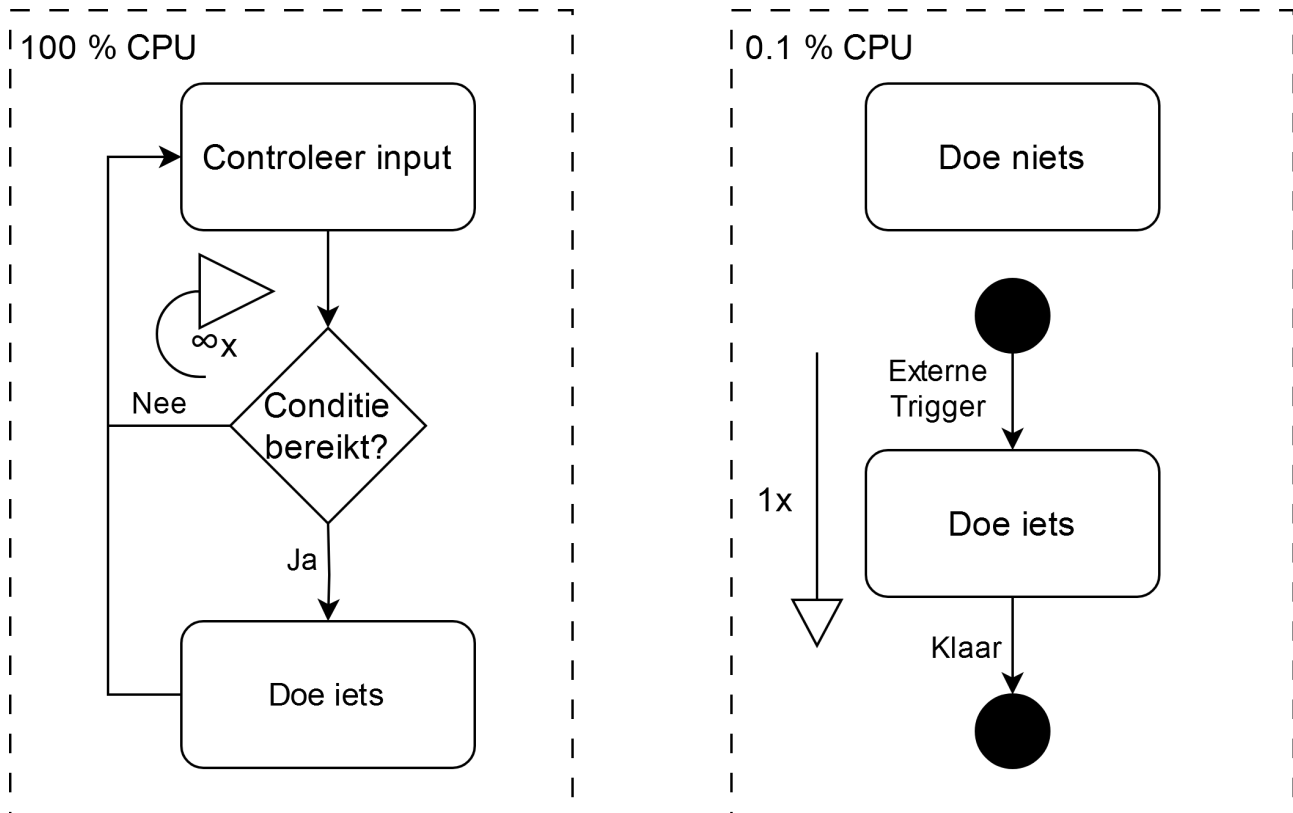


Figure 19: Het verschil tussen een polling (links) en interruptbased (rechts) systeem. Bij polling doorlopen we de kleine loop oneindig vaak, terwijl we in het interruptbased systeem pas in actie komen wanneer er een externe trigger komt.

we nog bezig waren met een moeilijke berekening. Algoritmes (flowcharts) zijn dus geen geschikte oplossing.

De oplossing voor dit probleem is de interrupt, die een signaal (vlaggetje) afvuurt, op het moment dat er iets verandert. Laten we dit eens toepassen op het voorbeeld van vorige week. In het geval van onze knop zou er een interrupt plaatsvinden op basis van een speciale trigger (signaal) van de pin aan het knopje. Op basis van die trigger kunnen we het proces in figuur 18 in gang zetten. We krijgen hiermee de situatie aan de rechterkant in figuur 19 waarin we bijvoorbeeld de hele tijd niets kunnen doen, en alleen wanneer het nodig is even de LED aan of uit doen.

Het gave van interrupts, is dat deze niet alleen op I/O-pinnen kunnen, maar dat ze door bijna alle peripherals van de uC gegenereerd kunnen worden. Deze interrupts kunnen op alle verschillende I/O pinnetjes plaatsvinden. In figuur 20 zie je een hardware overzicht. Dit geeft weer hoe de ISC (input sense configuration of interrupt service control) hardwarematig verbonden is met een peripheral zoals PORTA (PA) en direct de ISR (interrupt service routine) binnen in de CPU kan aanroepen. De ISR zorgt ervoor dat de CPU onderbroken wordt in zijn huidige werk om de gewenste handeling mogelijk te maken. Wanneer hij daarmee klaar is, gaat de CPU weer vrolijk verder met wat hij aan het doen was.

De ISC is expres als een balk getekend, om aan te geven dat deze in latere weken met meerdere peripherals verbonden zal zijn. Op die manier kun je een goede voorstelling maken van wat er hardwarematig precies gebeurt, en zie je wat voor centrale rol interrupts op onze uC spelen.

95. Lees eens wat verder over interrupts op het internet. Wat gebeurt er eigenlijk op de achtergrond?

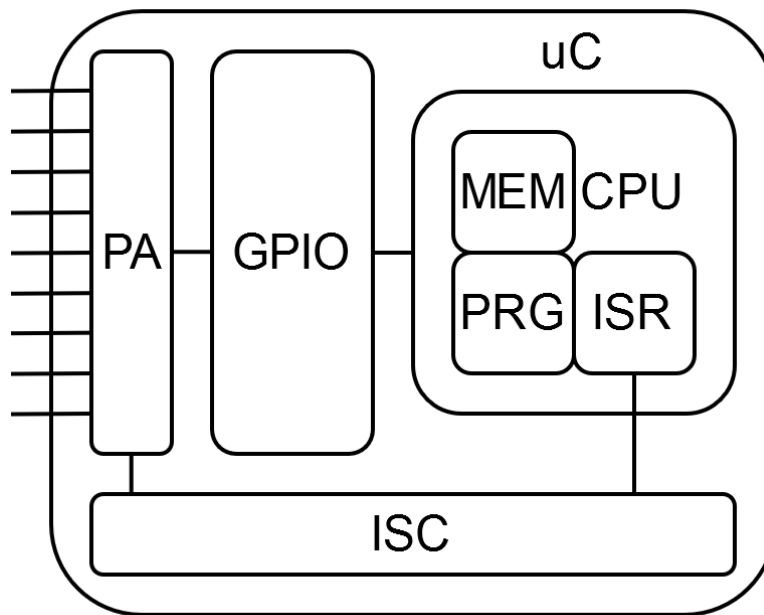


Figure 20: Hardwareschema van uC met de peripheral GPIO en ISR (interrupt service routine). We zien hier een extra blok ISC (interrupt service control) wat bepaald of interrupt gegenereerd kan worden. Dit ISC blok communiceert direct met de ISR. De ISR kan de CPU onderbreken in zijn huidige taak.

96. Ga in het datasheet op zoek naar het hoofdstuk interrupts en lees dit door

97. Welke peripherals kunnen er allemaal interrupts geven?

Interrupts zijn, net als alle andere peripherals op de uC instelbaar via registers in het geheugen. Deze stellen we weer in door in software in de initialisatiefase de juiste waarde toe te kennen. Om interrupts daadwerkelijk te gebruiken, dienen we vaak meerdere registers aan te zetten. Hiermee is de interrupt zeer goed instelbaar en kunnen we ons systeem volledig naar wens aanpassen.

98. Als je een interrupt op pin B2 aan wil zetten, wat moet je dan allemaal instellen?

99. Wat wordt er in je code aangeroepen wanneer een interrupt optreedt?

100. Wat bedoelen we met nested interrupts? Zijn nested interrupts wenselijk?

101. Waarom moet je code in een interrupt service routine functie (ISR) kort houden?

102. Wat is een interrupt vector? Hoeveel interruptvectors zijn er? Tip: H10.2 van datasheet.

Meestal wordt een GPIO interrupt getriggerd op basis van de rising en/of falling edge van een signaal op een pin.

103. Wat wordt er bedoeld met rising edge en falling edge? Illustreer dit met een tekening.

104. Stel je zou met het knopje een ledje willen aansturen. Hoe zou je dit op een slimme manier aan kunnen pakken op basis van interrupts? Tip: je hoeft maar 1x de LED aan te zetten, en 1x de LED uit te zetten.



### 3.1.2 Sleep & energie

Voor embedded systemen (de wereld van PCB's, hardware en software) is het vaak de bedoeling om zo weinig mogelijk energie te verbruiken. Je zult bijvoorbeeld je processor willen uitzetten wanneer deze niet nuttig rekenwerk hoeft te doen. Dit kan door de processor te laten slapen. Je processor en bepaalde peripherals bevinden zich op dat moment in een energiebesparende modus. Wanneer er een interrupt optreedt zal je processor wakker gemaakt worden, en zal je een bepaalde handeling uit kunnen voeren. De metafoor met de slapende student en de wekker is snel gelegd.

105. Ga opzoek naar het hoofdstuk over sleep control in je datasheet. Lees dit door
106. Welke verschillende sleepmodes heb je allemaal? Leg uit wat het voordeel is van verschillende modi.
107. Moet je nog steeds een while-lus gebruiken i.c.m. een sleep mode? Leg uit waarom wel/niet.
108. Stel je zit te wachten op een interrupt vanuit een peripheral. Welke modi zou je dan in kunnen zetten.
109. Hoe zou je vanuit het diepste niveau van slaap weer wakker kunnen worden?
110. Hoeveel clock cycli duurt het voordat je weer wakker bent?

Helaas staat er nog niet heel duidelijk in de handleiding hoe we in slaap komen. Er wordt gesproken over een SLEEP instructie, maar bij het schrijven van een SLEEP, SLEEP(), sleep() snapt je compiler helaas niet wat je bedoeld. We hebben hiervoor nog een aparte include nodig, namelijk: `#include <avr/sleep.h>`. Daarnaast moeten we ook in hebben gesteld welke slaapmodus we willen hebben, en moeten we de slaapmodus enablen. Pas dan kunnen we met een speciale functie in slaapmodus terecht komen. In de tweede helft van het practicum gaan we hier op in.

## 3.2 Practicum

### 3.2.1 Interrupts

We gaan ons programma voor de knop verbeteren door gebruik te maken van interrupts. Maak een nieuw project aan voor week 3 (week3\_int) en open dit in MCC. Let op dat je de juiste chip selecteert (AVR128DB48). Configureer de pinnen die aan LED0 en de SW0 via MCC.

111. Stel PB2 en PB3 correct in als input en output (bijvoorbeeld via de pin package view).
112. Navigeer terug naar het dashboard van MCC en klik op Pins (zie figuur 21).
113. Configureer beide pinnen via Pins. Denk eraan dat je de pullresistor op SW0 aanzet.

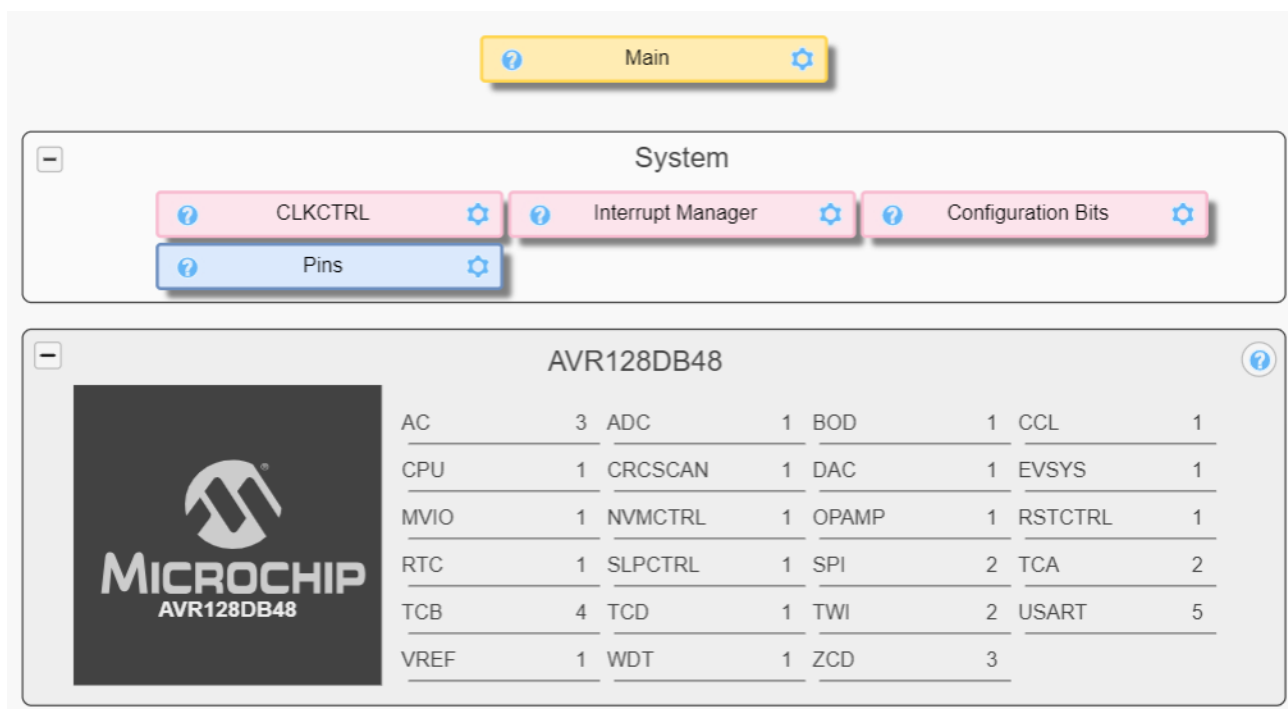


Figure 21: Dashboard van MCC. Zichtbaar zijn o.a. Pins en Interrupt Manager.

We willen nu een interrupt laten plaatsvinden op de knop. Hiervoor gaan we in MCC de interrupt instellingen aanpassen.

114. Zet input sense configuration voor de SW0 op falling edge.
115. Open de interrupt manager. Zet hier de Global Interrupt Enable (GIE) aan.
116. Genereer je project

We zijn benieuwd wat er precies gegenereerd is door MCC.

117. Navigeer naar MCC generated files/system/src/pins.c

118. Bekijk eens wat er gebeurd is met PORTB.PIN2CTRL. Welke waarde is hier toegekend?
119. Schrijf de hexwaarde om naar binair.
120. Pak het datasheet erbij en ga naar H18.4. Navigeer naar PINXCTRL.
121. Wat is er nu allemaal ingesteld in dit register? Benoem alles.
122. Stel we hadden geen falling edge maar een rising edge als triggerbron genomen. Wat zou dan de binaire invulling van dit register zijn?

Wanneer we de knop via MCC geconfigureerd hebben, krijgen we daar gratis een aantal functies bij. Dit hebben we vorige week ook gezien. We krijgen ook een functie voor het aanpassen van bovenstaande bits.

123. Probeer deze functies te vinden. Tip: de functienamen beginnen met de naam die je aan de knop hebt gegeven. Tip: gebruik Ctrl + Spatie voor code completion (suggesties).

We willen het lampje aan- en uitzetten op basis van een interrupt. Om nu een interrupt te kunnen genereren dienen we een interrupt service routine te programmeren ISR. Voor AVR processoren doe je dit door de volgende functie toe te voegen:

Listing 2: ISR

```
ISR ( interrupt_vector ) {  
  
    // Hier staat de code die uitgevoerd wordt bij het optreden  
    // van de interrupt. Aan het einde van je interrupt moet je  
    // de juiste interruptflag weer uitzetten  
}
```

Er is een hele waslijst aan interrupt vectors. Voor alle pinnen op port B is dit PORTB\_PORT\_vect. Ook de ISR wordt kant-en-klaar voor ons gegenereerd. Deze staat klaar in pins.c

124. Kijk maar eens opnieuw in pins.c en ga opzoek totdat je de ISR functie vindt.
125. Ga naar de declaration van PORTB\_PORT\_vect en bestudeer de verschillende interrupt vectors.

We worden weer genavigeerd naar ioavr128db48.h. Hier zie je een hele lijst met port vectors staan. Het kan handig zijn deze lijst te kopiëren en in een kladblok bij de hand te houden. Al deze vectors kunnen gebruikt worden als interrupt bron. Deze vectors komen overeen met allerlei verschillende peripherals van de uC. Je ziet dus dat interrupts een zeer belangrijke functionaliteit vervullen. In de komende weken gaan we een aantal van deze vectors gebruiken.

126. Sluit voor nu ioavr128db48.h en ga terug naar pins.c
127. In de ISR functie worden andere functies aangeroepen. Welke functies zijn dit? Valt je iets op aan de naam?
128. Ga naar de declaration van de functie die o.b.v. de knop zal worden aangeroepen.

Er zal je misschien iets opvallen. Je wordt genavigeerd naar boven aan in de code naar static void (\*SW0\_InterruptHandler)(void);. Dit is een speciale truc die door MCC gebruikt wordt om in application runtime de interrupt functie te kunnen aanpassen. Hoe werkt dit? Nou we hebben dus een pointer (vanwege \*) van het type SW0\_InterruptHandler(void). Dit is een bijzonder type, namelijk dit is een functie! Functiepointers dus! Zoals we gezien hebben in basisprogrammeren kunnen we naast pointers naar variabelen, waarbij we het adres van die variabele geven, ook pointers naar functies geven waarbij we het adres van die functie geven. Dit gaat ervoor zorgen dat we flexibel functies kunnen toekennen, ipv dat we vastzitten aan 1 functie.

Wat van belang is, is dat dit een global pointer is. We kunnen deze pointer dus ook naar een andere functie laten wijzen. Hiervoor heeft MCC een speciale functie gemaakt, namelijk void SW0\_SetInterruptHandler(void (\* interruptHandler)(void)).

129. Scroll maar eens wat naar beneden totdat je deze functie ziet.
130. Wat gebeurt er in deze functie?
131. Deze functie wordt op een andere plaats aangeroepen. Hier wordt een specifieke interrupthandler toegekend. Welke is dit?
132. Navigeer naar deze interrupthandler.

We hebben nu twee opties. Of je kunt gebruik maken van de default interrupthandler. Hierin plaats je de code die je wil uitvoeren. Wat je ook kunt doen, is zelf een interrupthandler maken en daarin de code plaatsen.

133. Bepaal zelf of je de defaultinterrupthandler of een eigen interrupthandler gebruikt.
134. Schrijf code in de interrupthandler die ervoor zorgt dat je LEDje togglet. Dat betekent dat hij de ene keer uitgaat de andere keer aangaat.

Wat als laatste belangrijk is in interrupts is het weer uitzetten van de interruptflag. Dit wordt ook voor ons gedaan. Navigeer opnieuw naar ISR(PORTB\_PORT\_vect) en bekijk de code onderaan de functie.

135. Bekijk de code waarin de intflags worden gecleared.
136. Ga opzoek in het datasheet naar intflags. Tip: zoek eens op intflags.
137. Lees goed wat er in de paragraaf staat. Hoe moet je een interruptflag weer uitzetten? Komt dit overeen met wat er in de code staat?

We gaan nu onze code uitproberen. Sla alles op en klik op run. Start je programma en laat deze in eerste instantie zonder breakpoints lopen.

138. Druk eens een paar keer op de knop en bestudeer het gedrag van de curiosity nano. Wat gebeurt er?

Nu ben ik benieuwd hoe het nou eigenlijk werkt onder de motorkap. Hiervoor gaan we een breakpoint plaatsen in de interrupthandler.

139. Klik in de linker kantlijn op de regel waar je je LED togglet om daar een breakpoint te plaatsen. Er komt een rood vierkantje te staan.
140. Start de AVR in debugmode. (Debug → Debug Main Project).
141. Controleer of de AVR runt (en niet gepauzeerd is).
142. Druk op SW0 en bekijk of hij stopt in de interrupthandler.
143. Druk opnieuw op play en probeer het nog eens. Let op! Wanneer onze uC gepauzeerd is, voert deze geen code uit/ kan hij nergens op reageren. We moeten hem eerst weer verder laten gaan.

We hebben nu een werkende interrupt, de interrupthandler geprogrammeerd en we kunnen zichtbaar maken dat we in deze interrupthandler terecht gekomen zijn. Nog wat belangrijke regels met betrekking tot interrupthandlers (of ISR functies). Zorg ervoor dat je zo kort mogelijk in de interrupthandler zit. De processor is echt onderbroken in zijn routine, dus dat kan gevolgen hebben op andere tijdskritieke processen. Vaak proberen we korte instructies uit te voeren. Wanneer er langere instructies nodig zijn zoals grote kopieeracties of loops, programmeer je dit in de actieve modus. Zorg er dan voor dat je bijvoorbeeld een globale boolean togglet in de interrupthandler, en dan op deze boolean iets uitvoert in je main functie. Een nog betere manier gaan we zien in week 6, waar we gaan werken met event driven state machines.

Daarnaast is het belangrijk om te weten dat wanneer een interruptbron aanstaat, deze altijd opgevangen moet worden door een ISR functie. Omdat we MCC deze functies laten genereren, gaat dit nu goed, maar een veel voorkomend probleem is, dat de processor geen ISR functie kan vinden en daardoor blijft hangen. Het ontbreken van deze functie levert namelijk geen syntax error op, en zal prima compileren. Wanneer de interrupt getriggerd wordt, kan de processor niet meer verder. Dit probleem zien we ook vaak als studenten onbedoeld interrupts aanzetten. Kortom, alle interrupts die je aanzet, moet je afvangen!

### 3.2.2 Sleepmode

Omdat we nog steeds in de `while(1)` lus zitten, en onze processor niet slaapt, gebruiken we 100% van onze processorcapaciteit. Onze processor gaat nu namelijk op volle snelheid door deze lus heen. Dat is niet heel efficiënt. Aangezien onze processor niets nuttigs aan het doen is, is dit een geschikt moment om hem te laten slapen.

144. Open MCC opnieuw
145. Onder device resources → System dubbelklik je op SLPCTRL. Deze wordt nu toegevoegd aan de application builder dashboard.
146. klik op SLPCTRL in dit dashboard zodat de SLPCTRL instellingen openen.
147. Bekijk de instellingen.
148. In principe hoeven we alleen maar de sleep enable aan te zetten. Doe dit nu.
149. Genereer je code opnieuw.
150. Navigeer nu naar MCC Generated Files → system → src → system.c. En bekijk de verschillende functies die betrekking hebben op de SLP.
151. We passen 2 registers m.b.t. SLP aan. Welk registers zijn dit?
152. Zoek in het datasheet naar het register wat de Sleep enabled en sleepmode instelt (dat is een van de twee registers van de vorige vraag). Bekijk de register description. Wat stel je precies in?

Nu we Sleep enabled hebben, kunnen we de sleepmode gebruiken. Dat betekent dat we op een bepaald moment in de code kunnen zetten dat we in slaap moeten vallen. Wanneer we slapen kunnen we met behulp van een interrupt weer gewekt worden.

153. Navigeer terug naar de main.c en voeg `#include <avr/sleep.h>` toe aan het begin van de code.
154. Voeg nu de functie `sleep_mode()` toe in de while lus.
155. Ctrl + klik nu op deze functie en lees en bekijk wat hij doet.
156. Run de code opnieuw en bekijk of hij nog steeds werkt.

Het is nog niet duidelijk of we daadwerkelijk slapen. Om dat te kunnen weten moeten we met behulp van de debug tools en breakpoints zien of onze processor daadwerkelijk gepauzeerd is. Helaas werken breakpoints niet altijd in combinatie met een functieaanroep. Daarom gaan we een debug variabele gebruiken.

157. Declareer de globale variabele: `volatile int a = 0;`
158. Plaats een `a++` onder de `sleep_mode()` toe en plaats op die regel een breakpoint. De `a++` geeft als extraatje dat we kunnen zien hoe vaak we een breakpoint hebben gehad.

159. Plaats ook een `a++` onder `LED0_toggle()` in de interrupthandler. Verplaats het breakpoint wat op de `LED0_toggle()` stond en plaats deze nu op `a++`.

Het is belangrijk om te beseffen dat we niet altijd een breakpoint kunnen plaatsen. Daarom zijn debugvariabelen zoals we hierboven aangemaakt hebben erg handig. Gebruik deze dus ook in de toekomst!

160. Debug je project opnieuw.

161. Druk op SW0

162. Beschrijf nu de volgorde van acties en in welke volgorde die uitgevoerd worden.

163. Teken een flowchart waarin je de volgorde van deze acties aangeeft.

Je weet nu hoe je interrupts in combinatie met de sleep modus kunt gebruiken. Je kunt hierna dit ook gaan toepassen op andere interruptbronnen zoals de ADC, timers of RTC.

## 4 Clocks, Timers & PWM

Het doel van dit practicum is het leren werken met oscillatorbronnen, clocks en timers van een microcontroller. Je begrijpt wat een oscillator, clock en timer zijn. Daarnaast kun je de clock en timer instellen. Met de timer kun je zelfs een Pulse-Width-Modulated (PWM) signaal generen. Op basis hiervan kun je weer verschillend peripherals hardwarematig aansturen, terwijl de uC iets anders doet of zelfs slaapt. Je snapt wat het nut is van verschillende instellingen, en kunt deze naar wens aanpassen. Figuur 22 laat zien hoe deze nieuwe peripherals plaatsnemen in de uC-architectuur. We zien hier dat de CLK en Timer meerdere verbindingen aangaan met niet alleen de CPU, maar ook andere hardware peripherals. Hiermee is duidelijk dat het systeem dus los van de CPU kan functioneren. Let er wel op dat we niet alle verbindingen tekenen. Een schema zoals in figuur 22 is bedoeld ter verduidelijking, maar inherent incompleet. Refereer naar p13 van het datasheet (Block Diagram) voor een completer en factueel juist overzicht.

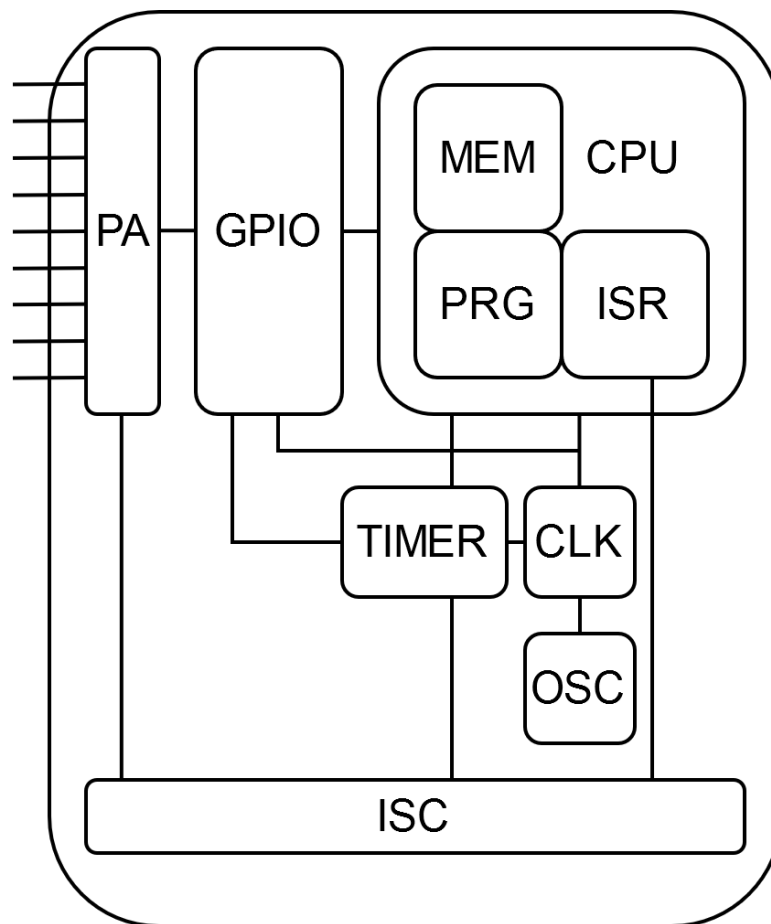


Figure 22: Hardwareschema van de uC. Nu met oscillatorbron (OSC), Clock System (CLK) en Timer. De OSC dient als bron voor de CLK. De CLK is verbonden met de CPU (klokbron voor de processor), de Timer en GPIO. De Timer is weer verbonden met ISC (als interrupt bron), de CPU en GPIO. Via de GPIO kunnen de CLK en Timer aangeboden worden op de pinnen. Andersom kunnen deze ook weer extern aangestuurd worden.



## 4.1 Voorbereiding

Timers, clocks (CLK) en oscillators (OSC) zijn verschillende begrippen die belangrijk zijn voor het begrip tijd op een microcontroller. Het kan misschien verwarrend zijn aan het begin hoe deze met elkaar samenwerken. Daarom hieronder een overzichtje

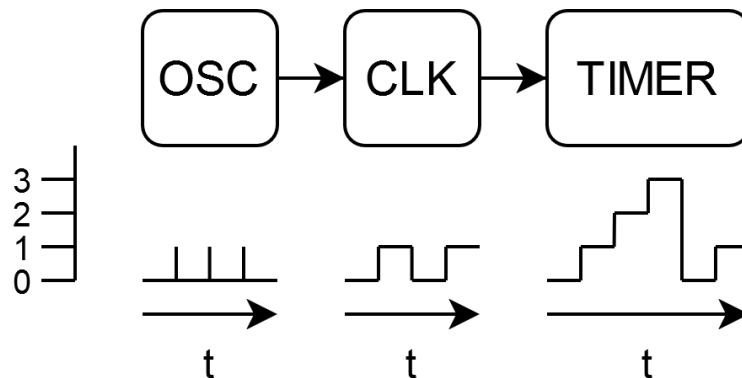


Figure 23: De OSC dient als bron voor de CLK. De CLK dient weer als bron voor de timer. Onder elke peripheral staat het uitgangssignaal wat die produceert. De OSC produceert periodiek een trigger. Deze trigger wordt door de CLK omgezet in een blokvormig kloksignaal. Dit signaal wordt weer door de timer gebruikt om een counter synchroon bij te houden.

We zien in figuur 23 dat alledrie de peripherals een specifiek iets doen. Hieronder beschrijven we deze in meer detail.

**Oscillator** Dit is letterlijk een trillingsbron. Dit kan bijvoorbeeld een kristal zijn wat met een bepaalde frequentie trilt. Doordat de trillingen periodiek zijn kan het als bron voor een clock dienen.

**Clock** Een clock is de hardware laag boven een oscillator. De clock is in principe niets anders dan een blok golf van een bepaalde frequentie. Frequentie betekent dat we per seconde zo vaak de gehele blok golf (hoog en laag) zien. Denk bijvoorbeeld aan 1 MHz, 4 MHz of 32 kHz. Deze frequentie kun je aanpassen met een prescaler divider. Je kunt er bijvoorbeeld voor kiezen om de frequentie 2/4/8/16 keer zo laag te maken. Jouw microcontroller heeft veel verschillende processen gekoppeld aan de klok. Dit betekent dat deze processen synchroon zijn. Ze laten deze acties gelijk lopen met de klokslag. Met klokslag bedoelen we een opgaande of afgaande flank. Elke keer dat de klok slaat zal er bijvoorbeeld een nieuwe instructie uitgevoerd worden door de ALU (arithmetic and logic unit in de CPU). Of bij elke klokslag wordt een deel van de ADC (analog to digital conversion) berekening gedaan (hierover in week 5 meer). Bijna alle peripherals hebben een clocksignaal nodig voor hun functionaliteit. Dit geldt ook voor de timer.

**Timer** Een timer kunnen we gebruiken om een verstreken tijd bij te houden. Dit doen we letterlijk door te tellen (counter). We gaan zie hoe dit gebruikt kan worden om bijvoorbeeld PWM, alarmen en timeouts te genereren. Een Timer is in de basis niets anders dan een counter. De Timer kan in verschillende modi tellen. In de basis telt de timer elke klokslag een omhoog. De Timer heeft een maximale telwaarde, waarna hij weer op 0 begint (overflow). Je kunt deze timer zelf configureren. Bijvoorbeeld tot hoever hij doortelt, of hij in omgekeerde richting telt, en wat er gebeurt als er een overflow plaatsvindt. Denk hierbij aan het triggeren van een interrupt, of het genereren van een PWM.

#### 4.1.1 Clocks & Oscillators

164. Ga in de datasheet van de microcontroller op zoek naar het hoofdstuk clocks en lees dit door. Beantwoord op basis van de tekst in dat hoofdstuk de volgende vragen.
165. Hoe ziet een kloksignaal eruit? Teken dit en geef hierin aan wat de waarde (hoog/laag of spanning) over tijd is. Geef ook aan wat een periode van de clock is.
166. Geef de naam van het register waarin we clock instellingen aan kunnen passen.
167. Welke oscillatoren hebben we? Geef de naam, en de frequentie.
168. Hoe werkt een prescaler divider? Illustreer dit met een voorbeeld.
169. Waarom moeten we voorzichtig zijn met het aanpassen van de instellingen van de (main) clock?

In plaats van dat we het kloksignaal willen delen, kunnen we vaak ook aan de peripheral kant het inkomende kloksignaal opnieuw delen. Heel vaak wordt die aanpak geprefereerd boven het delen van de (main) clock i.v.m. het nadeel wat je hierboven hebt vastgesteld. In het hoofdstuk over clocks wordt overigens ook nog gesproken over een cpu clock en de peripheral clock. Deze zitten doorverbonden aan de main clock. Helaas kan de AVR128 de peripheral clock niet apart aanpassen. Veel andere uC's hebben deze mogelijkheid wel.

Wanneer we de clockinstellingen willen aanpassen, gebeurt dat uiteindelijk weer via registers. Ook wanneer je MCC gebruikt. De clockregisters zijn echter bijzondere registers. Deze zijn beschermd met een zogenaamd CCP (let op! Dezelfde afkorting wordt ook gebruikt voor capture and compare. Deze hier niet mee verwarren).

170. Leg uit wat de CCP is, en wat voor bescherming dit biedt.

### 4.1.2 Timers

Timers zijn, zoals we al eerder hebben vermeld, niet anders dan counters (tellers). Ze tellen vanaf 0, 1, 2, ... tot de maximale waarde en beginnen dan weer opnieuw. Dat betekent dat de timer telt van 0 t/m 65535 ( $2^{16} - 1$ ) (single output mode, 16-bits). Normaalgesproken telt de timer door tot de hoogste waarde, waarna hij weer vanaf 0 begint. We noemen dit timer overflow (OVF). Omdat de timer tot vrij hoge getallen doortelt, is het sneller om de waarde in hexadecimaal te geven. Dat is dan bijvoorbeeld 0x0000 voor 0 en 0xFFFF voor 65535. In figuur 24 is dit schematisch weergegeven.

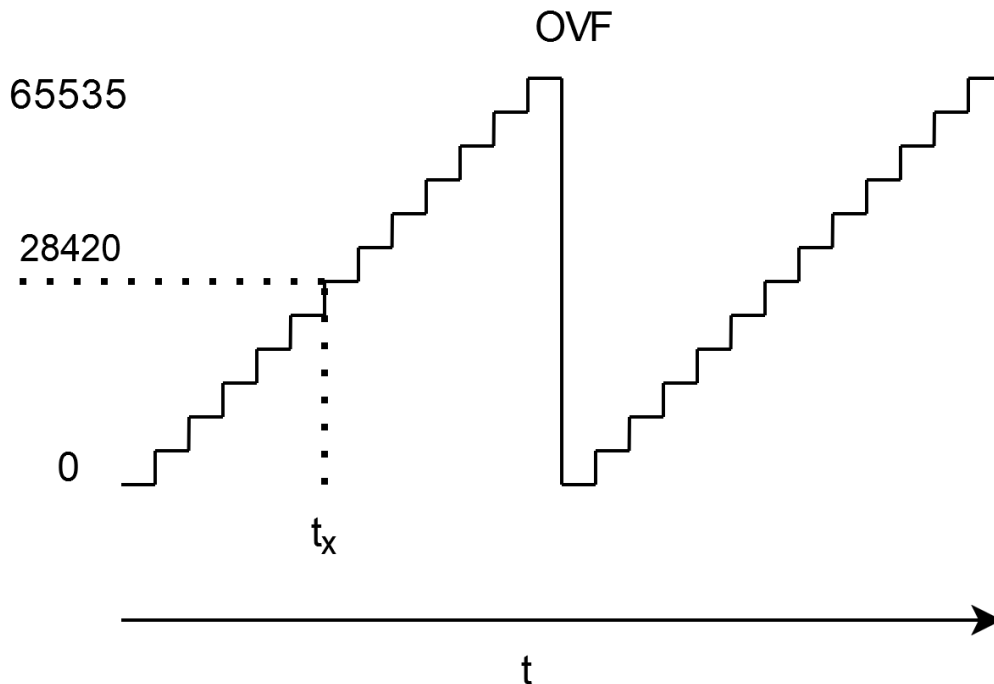


Figure 24: Een Timer telt van 0 t/m 65535. De timer (counter) waarde kunnen we op elk moment ( $t_x$ ) meten door een bepaald register te raadplegen. Bij OVF (overflow) reset de waarde naar 0 en telt hij weer van voren af aan.

De OVF is een hele handige hardwaretrigger die we weer kunnen gebruiken voor andere peripherals. Deze kan ook dienen als interrupttrigger. Dat betekent dat we de uC periodiek kunnen wekken waarmee we een periodieke actie kunnen laten uitvoeren in een interrupt service routine (ISR).

De timer telt op het ritme van de klok. Dat betekent dat elke klokslag (1x per periode) de timer 1 erbij telt. Wel kan het zijn dat we een extra prescaler divider toepassen, zodat we langzamer tellen. De prescaler divider heeft als effect dat de timer met die factor langzamer telt. Het is ook mogelijk om de OVF naar voren te halen. Dit kunnen we doen als we bijvoorbeeld minder ver willen tellen en sneller een OVF willen genereren. Door hiermee te spelen kunnen we de timing zo tweaken dat deze aan onze wensen voldoet.

171. Stel we zouden de timer laten tellen tot 0x61A8. De clockbron is 4 MHz en we hebben een prescaler van divide by 16. Wat is de periodetijd van de timer (dus hoe lang duurt het voordat hij een OVF krijgt)?

Op onze uC zitten verschillende timers (TCA, TCB en TCD) die telkens net iets anders werken. We gaan dit practicum aan de slag met timer A (TCA).

172. Navigeer naar het hoofdstuk van timer TCA.

173. Skim eens wat door dit hoofdstuk, en bekijk ook de register description.

We zien als eerste een lijst met features van deze timers. Zo is de timer 16-bits, heeft 3 compare channels en waveforms of interrupts genereren. Ook is er een mogelijkheid tot het splitten van de timer.

In tegenstelling tot timer A (TCA) heeft timer B (TCB) een capture mode. Deze gaan we niet gebruiken in dit practicum, maar het is wel handig om te weten waarvoor je dit zou kunnen gebruiken.

174. Bekijk het volgende document: [Using the CCP module](#) en navigeer naar het hoofdstuk over compare mode.

175. Leg in je eigen woorden uit wat een compare doet.

176. Leg in je eigen woorden uit wat een capture doet.

Stel je wil een timer gaan gebruiken voor de volgende toepassingen. Gebruik je dan een capture, of gebruik je dan een compare?

177. Je wil elke 1024 klokslagen een meting doen

178. Je wil de tijd bijhouden tussen 2 metingen

179. Je wil een PWM genereren op een I/O pin

180. Je wil periodiek de ADC uitlezen

We kunnen de timers in compare mode aansluiten op een interrupt. De interrupt kan triggeren op een overflow, maar ook wanneer er een zelf ingestelde waarde in de timer bereikt is. Dit stel je, je raadt het al, weer in via registers.

181. Wat zijn de interrupt vectors van de verschillende timers?

### 4.1.3 PWM

Een pulse width modulated signal (PWM) is niets anders dan een blok golf waarvan we de dutycycle kunnen bepalen. Een aantal voorbeelden van PWM signalen zie je in figuur 25. Je ziet hier dat we een digitaal signaal hebben wat hoog of laag is.

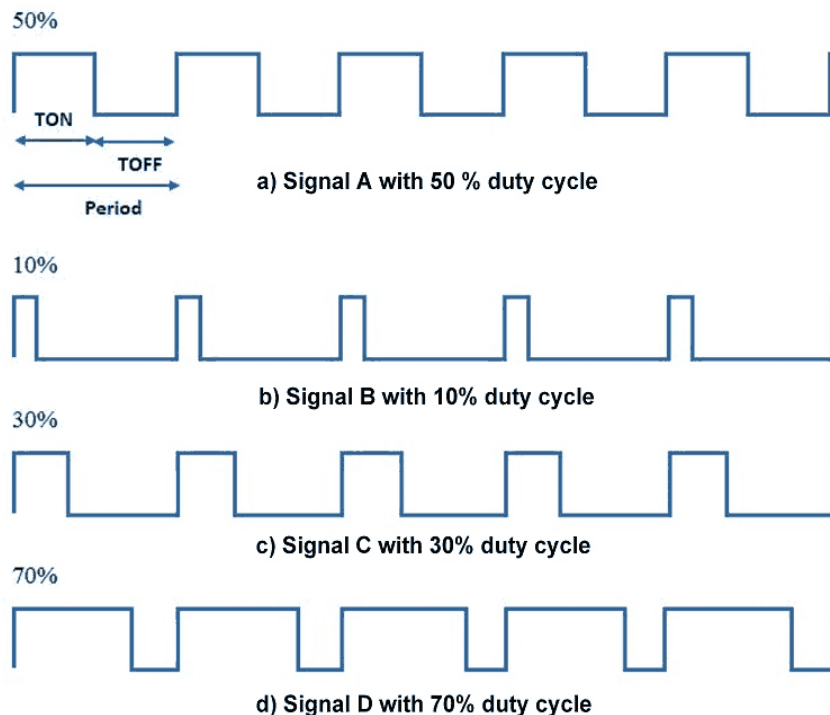


Figure 25: PWM voor verschillende duty cycles (bron [electronicwings.com](http://electronicwings.com))

De tijd dat we hoog zijn noemen we  $T_{ON}$  de tijd dat we laag zijn noemen we  $T_{OFF}$ . De som van deze twee is de periode. De dutycycle geeft als percentage weer wat de verhouding  $T_{ON}$  tot de totale periode tijd is. Je berekent deze als volgt:

$$duty cycle = \frac{T_{ON}}{T_{ON} + T_{OFF}} \cdot 100\% \quad (1)$$

Een PWM signaal is een digitaal surrogaat (vervanging) van een analoog signaal. Door de dutycycle te variëren van heel laag naar heel hoog, krijgen we iets wat in veel gevallen overeenkomt met een bepaalde analoge spanning. Dit principe kan bijvoorbeeld gebruikt worden om een LED zo snel te laten knipperen dat we niet zien dat de LED een bepaalde tijd aan staat, en een bepaalde tijd uitstaat. Hiervoor moeten we wel zorgen dat we een hooggenoege frequentie toepassen. Het menselijk oog neemt maar enkele tientallen Hz weer, dat betekent dat wanneer we een frequentie van een kHz of hoger hebben, we dit niet gaan waarnemen. Het effect is een lineair dimbare LED! Probeer dat even goed te beseffen. We hebben van een zeer nonlineair component als de LED een lineaire control op toegepast. De felheid wordt uiteindelijk bepaald door de dutycycle, waarbij een dutycycle van 0% volledig uit is, en een dutycycle van 100% volledig aan is (overigens is dit toevallig bij de LED op de curiosity nano precies andersom door de manier van aansluiten).

De PWM kunnen we voor nog veel meer toepassingen gebruiken. Deze kan ook gebruikt worden om snelle schakelaars aan en uit te zetten. In Hardware Design gaan we zien hoe we exact hetzelfde principe kunnen toepassen op een transistor en hoe dit gebruikt kan worden om met een H-brug een motor te besturen. Daarnaast zie je dit in tal van elektronische systemen terug zoals bijvoorbeeld

in vermogensschakelaars die met behulp van buck-boostconverters de spanning efficiënt kunnen omzetten.

Het PWM signaal is zeer eenvoudig te maken, door handig gebruik te maken van een aantal eigenschappen van de timer. Figuur 26 laat zien hoe dit proces in zijn werk gaat.

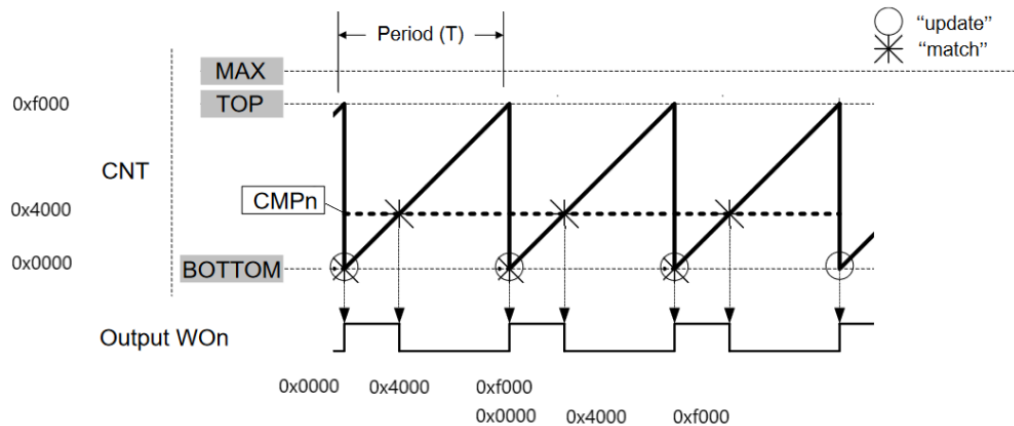


Figure 26: PWM in single output mode. Aangepast o.b.v. figuur 23-10 uit datasheet AVR128DB (bron: microchip.com)

Wanneer we een PWM in single slope mode zetten kunnen we deze als volgt instellen. We kiezen een periode tijd tot waar hij telt (TOP), en we kiezen een waarde voor het compare register (CMP, compare waarde). De output van de timer (waveform output n, of WOn) zal hoog blijven totdat we de waarde van het compare register voorbij gaan. Daarna wordt de WOn waarde laag, totdat we voorbij de periode tijd zijn en weer opnieuw beginnen te tellen. We kunnen de periode tijd niet als tijd meegeven, maar moeten zelf op basis van de klokfrequentie bepalen wat deze zou moeten zijn. Hetzelfde geldt voor de dutycycle. Wanneer we een bepaalde dutycycle willen realiseren, dienen we het compare register de volgende waarde te geven:  $CMP_n = TOP \cdot dutycycle$ . Daarnaast is belangrijk om te beseffen dat de frequentie van onze timer  $f_{timer}$  gelijk is aan  $f_{timer} = \frac{f_{klok}}{TOP}$ , waarin  $f_{klok}$  de klokfrequentie is.

In figuur 26 zagen we dat de output op WOn eerst 1 is, totdat de waarde van de CMPn 0x4000 (16384) bereikt wordt door de counter. Daarna is de waarde van WOn 0 totdat we de TOP of maximale waarde van de timer bereiken. In dat voorbeeld is deze gekozen op 0xF000 (61440). De dutycycle is in dit geval  $\frac{CMP_n}{TOP} \cdot 100\% = \frac{0x4000}{0xF000} \cdot 100\% = 26,67\%$ .

182. Stel we kiezen een TOP waarde van 20000 en CMP waarde van 5000. Wat is de dutycycle?
183. Stel we willen een timer frequentie van minimaal 1kHz. Wat moet dan in bovenstaande de klokfrequentie minimaal zijn?
184. Stel we kunnen niet deze eis behalen met een 2 MHz klok. Hoe zouden we de timer kunnen aanpassen dat deze toch deze frequentie en juiste dutycycle heeft?

De output van de timer (WOn) kan rechtstreeks op een GPIO pin aangeboden worden. Hiermee stuur je verschillende componenten aan. Wanneer je je LED of motor wil aansturen varieer je normaalgesproken niet de frequentie (periodetijd, of TOP) van de PWM, maar de dutycycle (CMP).

185. Stel we zouden een LED aansturen met een PWM met periodetijd van 0xFFFF. We zetten de dutycycle in de code op 0x1000. Geef de dutycycle als percentage weer.

186. Wat verandert er aan de LED wanneer we de dutycycle hoger of lager maken?

Om de output van onze timer WOn te koppelen aan een bepaalde I/O poort dienen we dit eerst in te stellen met de PORTMUX.

187. Open het hoofdstuk over PORTMUX in het datasheet en lees wat er in de eerste alinea staat.

188. Wat doet de PORTMUX?

189. Scroll naar beneden en ga naar TCAROUTEA.

TCAROUTEA bepaalt hoe de uitgangen van onze timer doorverbonden worden met de GPIO pinnen. We zien hier dat we een TCA0 en TCA1 timer hebben. Als we tijdens het practicum de timer gaan configureren in MCC moeten we goed onthouden welke we gebruiken. De verschillende timers kunnen namelijk niet op alle poorten aangesloten worden. We zien een tabel met als kolommen WO0 t/m WO5. WOn (waveform output n) is direct gekoppeld aan de compare register n van onze timer. De rijen in de tabel komen overeen met de verschillende I/O poorten (A t/m G). In de cellen zien we de verschillende pinnen. Dit betekent dat we een specifieke WO kunnen koppelen aan een specifieke pin. Denk dus bijvoorbeeld aan onze LED (PB3).

190. Hoeveel compare registers had Timer A?

Wat je wellicht hebt vastgesteld, is dat de hoeveelheid compare registers van onze timer in eerste instantie niet overeenkomt met de hoeveelheid WO. Om alle WO signalen te kunnen gebruiken, moeten we de compareregisters splitsen. Dat betekent dat de ene byte voor  $WO_n$  gebruikt wordt en de tweede byte voor  $WO_{n+3}$ . Hiermee verdubbelen we het aantal PWM's, ten koste van de nauwkeurigheid van de dutycycle. Immers in plaats van 16-bits (2 bytes) gebruiken we nu 8-bits (1 byte). Helaas, geen gratis lunch.

Een deel van de WO is dus niet in standaardmodus (single output mode) te gebruiken. Helaas valt onze LED op de curiosity nano (PB3) hier ook onder. Deze valt onder WO/3. In single output mode heeft de timer maar 3 compare channels die verbonden zijn aan WO/0, WO/1 en WO/2. In single output mode hebben we 3 PWM signalen van 16-bits resolutie.

Wanneer we de LED willen aansturen moeten we toegang krijgen tot WO/3 t/m WO/5. Hiervoor moeten we de timer in splitmode zetten. In de Splitmode splitst de uC onze compare channels en kunnen we 6 PWM signalen van 1 byte resolutie maken.

191. Vergelijk de resolutie van een 16 bits met een 8 bits. Welke orde van grootte wordt de resolutie anders? Tip: de resolutie van 8-bits is veel lager dan  $\frac{1}{2}$  van 16-bits.

We maken dus een afweging tussen resolutie en hoeveelheid PWM-signalen. Let er op dat je voor je Drive Exchange robot hier een juiste afweging maakt. Je kunt namelijk niet alle poorten van je uC gebruiken voor een 16-bits PWM. Anderzijds kan een 8-bits PWM wellicht een te lage resolutie bieden voor een goede aansturing bij professionele apparatuur zoals omvormers.

## 4.2 Practicum

### 4.2.1 Clocks

Maak een nieuw project aan en open MCC (week4\_clk). We stellen PB3 weer in als LED0 via de pinmux. Ga nu naar CLKCTRL. In CLKCTRL hebben we veel verschillende instellingen. Een gedeelte hiervan zie je ook al in figuur 27.

▼ Software Settings	
Generate Initializer Code	Initialize all registers

▼ CLKCTRL Settings	
Main Clock (Hz)	4000000
Clock Selection	Internal high-frequency oscillator
Internal Oscillator Frequency	1-32MHz internal oscillator
Oscillator Frequency Selection	4 MHz system clock (default)
Multiplication Factor	PLL is disabled
External Clock Source for PLL	OSCHF
External Clock (Hz)	1 ≤ 3000000
Prescaler Enable	<input type="checkbox"/>
Prescaler Division	2X
System Clock Out Enable	<input type="checkbox"/>
Timebase	4

Figure 27: CLKCTRL. We zien hier een gedeelte van de grote hoeveelheid instelling m.b.t. de clock.

We beginnen met de CLKCTRL Settings. Hierin kunnen we de klokbron selecteren en kiezen of we een prescaler willen. Tevens kunnen we onze CLKOUT configureren. Dan wordt de MCLOCK doorverbonden met pin A7. Dit kun je vinden in het datasheet, en wordt ook gelijk zichtbaar in de pin package view.

192. Controleer of de aanname klopt dat we de klokoutoput op A7 kunnen krijgen, door de system clock out aan te zetten.
193. We gaan nu spelen met de prescaler. Zet de prescaler aan (PEN) en genereer je project.
194. Plak de volgende code in je project.
195. Run je code en bekijk met welke frequentie de LED knippert



Listing 3: Code voor een knipperende LED

```
#include "mcc_generated_files/system/system.h"

volatile unsigned int a = 0;

int main(void)
{
    SYSTEM_Initialize();

    while (1) {
        a++;

        if(a >= 40000){
            a = 0;
            LED0_Toggle();
        }
    }
}
```

196. Verander de prescaler door reconfiguration van je ATMEEL start project. Probeer met waardes van 4, 8, 16, 32. Bekijk telkens opnieuw met welke frequentie je LED knippert.
197. Wat constateer je hierdoor over de snelheid van de CPU?
198. Zet de prescalar divider weer uit.

Standaard staat onze oscillatorbron op de high frequency (internal) oscillator.

199. Wat is de standaardfrequentie waarmee deze oscilleert?
200. Kies nu de oscillatorbron met een frequentie 32,678 kHz.
201. Waarom heeft deze bron de specifieke frequentie van 32,678 kHz en niet gewoon 30 kHz?
202. Genereer je project opnieuw en upload je code naar de uC. Wat voor effect heeft dit op de CPU snelheid?
203. Zet je oscillatorbron weer op de high frequency oscillator.
204. Klik eens op Oscillator Frequency selection.
205. Stel we zouden frequency select aanpassen naar 16, wat zou er dan gebeuren met onze CPU snelheid?
206. Controleer je vermoeden door de instelling aan te passen. Genereer je project opnieuw en upload deze opnieuw. Observeer wat de snelheid van het knipperen is, en concludeer wat dit zegt over de snelheid van de CPU.
207. Kunnen we onze uC dus ook sneller maken?
208. Waarom draait onze uC standaard niet op een hogere frequentie?

## 209. Zet de instellingen weer terug naar hun basiswaarden

Zoals je dus misschien hebt vastgesteld zijn er een hoop instellingen die we kunnen kiezen. Vaak wordt de klok door onervaren programmeurs met rust gelaten. Dat is begrijpelijk, gezien het feit dat veel instellingen zeer niche toepassingen hebben, terwijl ze toch vaak onbedoeld andere aspecten van de uC beïnvloeden. Toch kan het handig zijn om over de klokinstellingen te leren, zodat je echt het maximale uit je uC kunt halen. De uC biedt ons deze flexibiliteit met een belangrijke reden, de uC is multipurpose, dus we moeten zelf bepalen hoe we hem instellen. Deze instelling zullen altijd een trade-off zijn tussen snelheid en energieverbruik.

### 4.2.2 Timers

Maak een nieuw project aan (week4\_timer). Configureer nog geen output op LED0. We voegen nu een timer toe. Dit doen we door onder Device Resources naar Timer te gaan en dubbel te klikken op TCA0. Zie ook figuur 28.

210. Voeg TCA0 toe aan je project.

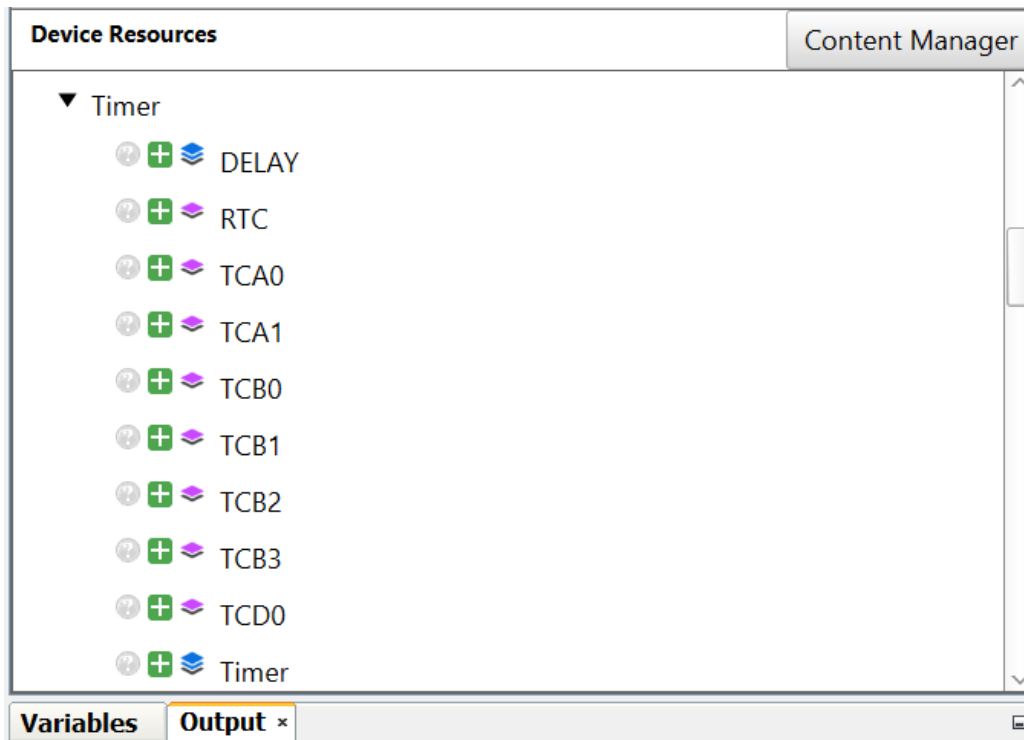


Figure 28: Het toevoegen van een timer peripheral

Via Device Resources kunnen we elke peripheral op onze uC kiezen en toevoegen aan ons project. Deze 'resources' zijn gebonden aan onze uC. Andere uC's zullen dan ook andere keuzes hebben. Je ziet al dat er een hele waslijst aan verschillende timers is, maar ook heel veel andere peripherals zijn beschikbaar. Voel jezelf ook vrij om daar eens in te neuzen om een idee te krijgen over wat je allemaal met je uC kunt. Standaard staan al deze peripherals niet ingesteld. De reden is weer simpel, wat je niet gebruikt staat uit. Pas als je iets nodig hebt, voeg je het toe. Dit houdt niet alleen je uC energiezuinig, het houdt je projectstructuur ook erg overzichtelijk. Wanneer je een peripheral niet meer nodig is, verwijder deze dan ook weer uit je project. dit doe je onder project resources op het minnetje te klikken.

Als het goed is, is TCA0 nu zichtbaar in de application builder en opent het configuratiescherm van TCA0. Wanneer het configuratiescherm niet zichtbaar is, klik je even dubbel op TCA0 in de application builder. Je krijgt nu het scherm, zoals in figuur 29 zichtbaar is, te zien.

211. We configureren de timer nu door de volgende stappen uit te voeren.

- (a) Zet de Timer Mode op Split Mode
- (b) Zet compare channel 3 (WO/3) aan.

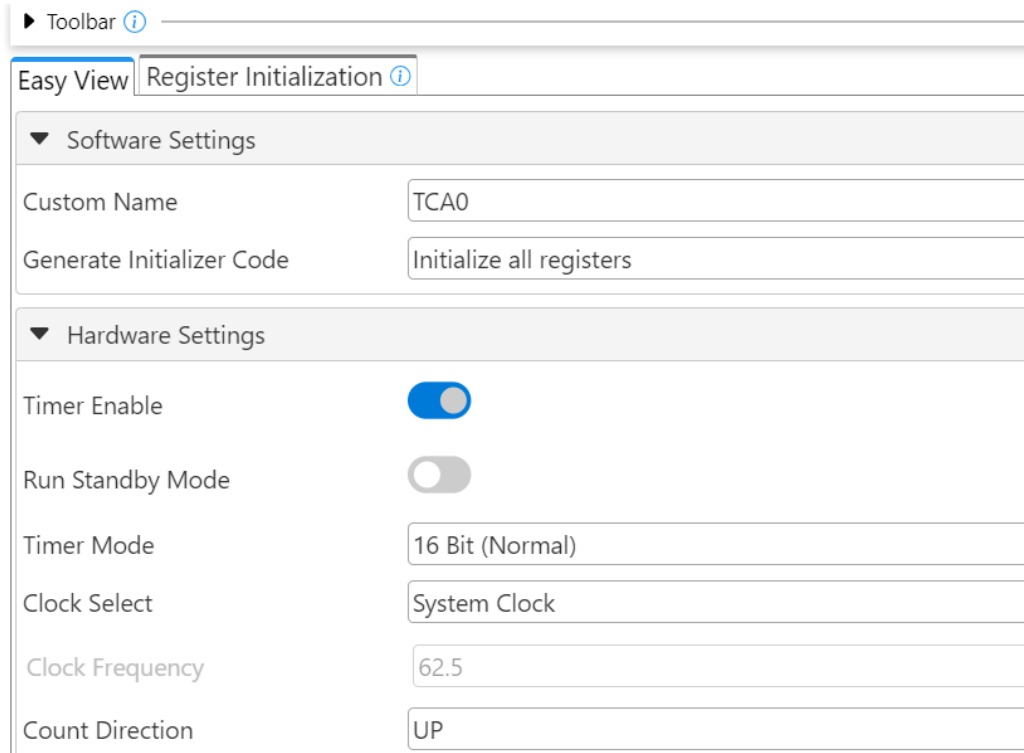


Figure 29: Een deel van het configuratiescherm van TCA0.

(c) Zet de dutycycle op 80%.

(d) In pin package view klik je op PB3 en koppel je WO/3 aan deze pin.

212. Genereer je project en klik op run. Afhankelijk van de gekozen waarde van HCMP0 zal de LED variëren in intensiteit. Let op: De LED staat aan wanneer we een logisch hoog laag signaal krijgen. Dat betekent dat de LED uit staat bij een dutycycle van 100

213. pas de dutycycle aan naar 90% en genereer en run opnieuw. Observeer wat de LED doet.

214. pas de dutycycle aan naar 95% en genereer en run opnieuw. Observeer wat de LED doet.

Het is niet heel efficiënt om elke keer als we de dutycycle willen aanpassen, we het project opnieuw moeten genereren. Het zou veel makkelijker zijn wanneer we dat in code konden doen. Om dit te kunnen doen moeten we weten welk register op de achtergrond aangepast is.

215. Ga terug naar het TCA0 configuratiescherm.

216. Scroll naar boven en klik op het tabje 'Register initialization'.

We komen nu op een andere pagina waar alle registers m.b.t. de timer TCA0 staan. Hier kunnen we precies zien wat er allemaal ingesteld wordt. De meeste registers worden geïnitieerd op 0x00 (oftewel 0b00000000). Dat betekent gewoon dat we er niets mee doen. Sommige registers worden wel aangepast.

217. Welke registers worden allemaal ingesteld?

We wisten dat het CMP (compare) register gebruikt werd om de dutycycle in te kunnen stellen. Oorspronkelijk hadden we er 3 (16-bits), maar nu dus 6 (8-bits) omdat we in splitmode zitten. Om de naamgeving consistent te houden, is ervoor gekozen elk van deze 3 16-bits registers te splitsen in een specifiek laag en hoog compare registers. Je hebt dus LCMP0, LCMP1, LCMP2, HCMP0, HCMP1 en HCMP2. Een van deze registers bevat een waarde anders dan 0.

218. Welke register is dit?

Dit register zorgt voor WO/3 (de andere registers horen dus bij WO/0, WO/1, WO/2, WO/4 en WO/5). We gaan dit register nu niet hier aanpassen, maar we gaan deze informatie gebruiken om in onze code de juiste functie aan te roepen om dit register te kunnen wijzigen.

219. Ga terug naar je main.c

220. Typ in TCA0 (dit is de naam van je timer. Je had hier overigens ook een andere naam voor kunnen instellen als custom name in het configuratiescherm in MCC).

221. Gebruik 2x Ctrl+Spatie zodat een lijst aan relevante functies verschijnt.

222. Bekijk de lijst en probeer de juiste functie te vinden die ervoor zorgt dat het juiste compare register aangepast wordt.

Wanneer je door de lijst scrollt, zal je opvallen dat de functie er niet tussen staat! Helaas heeft MCC niet aan alles gedacht. Om dit register toch aan te passen, kunnen we twee dingen doen. Het register direct aanschrijven of een functie maken die dat voor ons doet en die aanroepen.

223. Pas een van beide methodes toe en pas het juiste compareregister aan in je software. Let op dat je een waarde van 0 t/m 255 kunt toekennen, i.v.m. de grootte van een byte.

224. Demonstreer dat je code werkt door het register twee keer achter elkaar aan te passen met telkens een korte delay er tussen. Hiermee laat je de LED bijvoorbeeld helder en minder helder knipperen. Dit zou goed zichtbaar moeten zijn op je LED.

Nu we weten hoe we de PWM in realtime kunnen aanpassen, kweekt dat ruimte voor verschillende toepassingen. We kunnen een dimbare led maken.

225. Maak een programma waarin je de LED steeds feller maakt. Elke 10 ms moet er 1 bij de compare waarde opgeteld worden waardoor de dutycycle steeds wat groter wordt. Na 255 moet hij weer op 0 beginnen.

## 5 ADC

Het doel van dit practicum is om te leren werken met de Analog to Digital Conversion (ADC). Je leert hoe je een ADC kunt gebruiken. Je leert hoe je verschillende analoge signalen kunt uitlezen. Deze signalen kun je na dit practicum versturen via de UART zodat je gemakkelijk kunt debuggen. Je combineert deze kennis met het practicum Hardware Design waarmee je verschillende sensoren kunt uitlezen.

### 5.1 Voorbereiding

#### 5.1.1 Signalen

Voordat we kunnen begrijpen wat de ADC doet, moeten we eerst begrijpen hoe wat het verschil tussen analoog en digitaal is.

226. Leg in je eigen woorden uit hoe de amplitude (data) in een analoog en een digitaal signaal vormgegeven wordt.

Signalen bestaan in tijd. Hier hebben we twee belangrijke categorieën. Discreet en continu. Een discreet signaal is een signaal wat een **eindige nauwkeurigheid** in waarde en tijd heeft, een continu signaal is een signaal wat **oneindige nauwkeurigheid** in waarde en tijd heeft. Kijk ter illustratie maar eens figuur 30.

227. Leg aan de hand van figuur 30 in je eigen woorden uit wat een discreet en een continu signaal is.

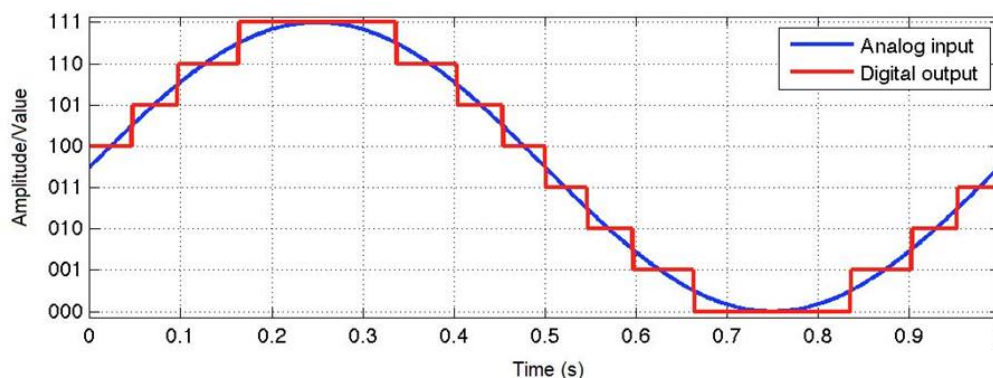


Figure 30: Een continu (analog) en een discreet (digital) signaal (bron: Arrow Electronics).

Digitale signalen zijn altijd discreet. De data die je in bits opslaat is gekoppeld aan een klok signaal (synchroon). Een analoog signaal is per definitie niet discreet maar continu in de tijd. De data die het signaal bevat een continu veranderend signaal (asynchroon).

Wanneer we van het analoge domein naar het digitale domein willen gaan, moeten we periodiek een sample nemen omdat we maar een beperkte resolutie in tijd hebben. De reden hiervoor is simpel. Meten kost tijd. Voordat we een goed sample genomen hebben, is er een bepaalde tijd verstreken waarin het gemeten signaal alweer van waarde veranderd kan zijn.

228. Leg uit wat er bedoeld wordt met sampling.

Een belangrijke eigenschap van discrete signalen is resolutie. We bedoelen hiermee resolutie in waarde (bijvoorbeeld spanning) en resolutie in tijd. Om een signaal goed weer te geven, dienen we duidelijkheid te hebben over wat een bit precies betekent. Is dit 0,1V, 0,36V, 2,4V etc. Datzelfde geldt voor tijd, 0,1s, 0,0001s of 0,25s. Vaak wordt tijd niet apart gemeten, maar triggeren we de meting op een periodieke interval. Wanneer we dit doen, kunnen we op basis van dat interval bepalen hoe het signaal er over tijd uitziet.

229. Leg uit wat er fout gaat als we een signaal willen meten en we niet weten wat de waarde- of tijdsresolutie is.

Het nemen van een sample moet altijd getriggerd worden. Dit betekent letterlijk het starten van de meting. In het geval van het meten van een signaal over tijd gebeurt dit synchroon (op periodieke intervallen). In tegenstelling tot wat we hier boven beweren kan het soms ook nuttig zijn om op basis van een niet synchrone triggerbron een sample te nemen. Dit is heel vaak het geval wanneer we tijdens een interrupt iets willen meten.

230. Geef een voorbeeld van een meting die niet synchroon loopt met de tijd.

Een voorbeeld van een synchrone trigger is de timer, een asynchrone trigger is bijvoorbeeld een trigger gegenereerd vanuit software. Let op! Beide manieren van samplen kunnen we plotten in een grafiek. Toch zegt alleen een synchrone sample iets over tijd, terwijl asynchroon dat dus absoluut niet doet. Een asynchrone trigger kan over 1s komen, maar net zo goed pas over 200s. Een synchrone trigger zal altijd op hetzelfde interval gebeuren.

### 5.1.2 ADC

De ADC zet analoge (continue) signalen om naar digitale (discrete) signalen. Het is een hardware peripheral die we net als alle andere peripherals via registers kunnen instellen. We gebruiken de ADC om analoge samples te nemen van een allerlei aan sensoren.

De ADC heeft voor zijn interne werking een snelle klok nodig om de conversie uit te kunnen voeren. Zoals we hiervoor zagen gebruiken we een trigger om de ADC te starten. Nadat de ADC gestart is zal hij een sample nemen op een bepaalde pin. Het samplen kost tijd en wordt synchroon uitgevoerd. Hiervoor heeft de ADC een kloksignaal nodig. In figuur 31 zien we hoe de ADC peripheral met allerlei andere peripherals, waaronder de CLK verbonden is. Zo kan de ADC vanuit verschillende bronnen zoals een timer of de CPU getriggered worden. De ADC samplet op de GPIO en kan het resultaat doorgeven aan de CPU. Ook kan de ADC een interrupt (via de ISC) genereren.

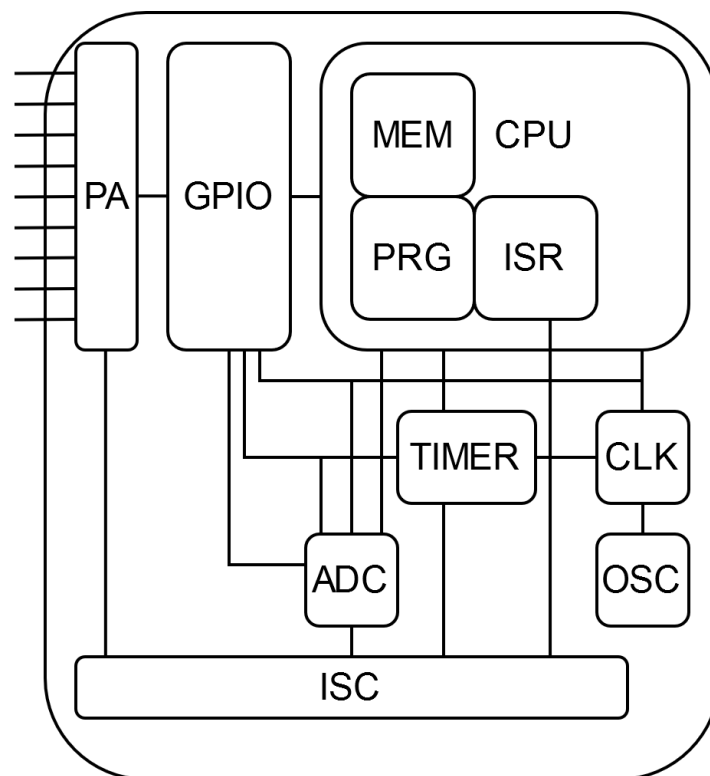


Figure 31: Het geüpdatet hardware schema met de ADC erbij. De ADC zit verbonden met de CPU, de GPIO, de Timer, de CLK en ISC. Niet alle verbindingen zijn getekend, dit i.v.m. zeer rap toenemende complexiteit.

Om een analoog naar digitaal conversie uit te voeren, zijn er verschillende technieken. Voorbeelden hiervan zijn Direct-Conversion, Successive Approximation (SAR) en sigma-delta. Om helemaal deze technieken uit te diepen gaat iets te ver voor dit vak. We kunnen voor nu deze conversie als blackbox aannemen, zolang we er rekening mee houden dat onze ADC altijd een eindige resolutie heeft in amplitude en tijd. In figuur 32 staat dit zeer ingewikkelde model weergegeven.

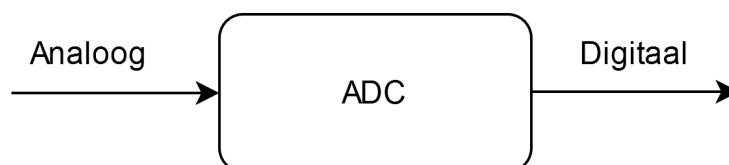


Figure 32: Het blackbox model van de ADC.

We kunnen nooit ‘perfect’ een analoog signaal meten. De precisie van een specifieke ADC (bits-



resolutie) en snelheid (tijds-resolutie) bepaalt hoe precies onze meting zal zijn. We gaan nu kijken naar de ADC van de AVR128DB.

231. Open het datasheet en ga naar het hoofdstuk over de ADC.

We lezen dat de ADC een resolutie heeft van 12 bits. Dat betekent dat het ingangsspanningsbereik wordt verdeeld in  $2^{12}$  niveaus. Denk bijvoorbeeld aan je digitale multimeter (DMM). Als je die in het bereik 2V zet, dan kan deze meten tussen -1.999 V en +1.999 V, in stapjes van 0.001 V. Dat zijn dus 4000 stapjes. Maar wat stelt zo'n bit nou precies voor? Wat je in Figuur 30 op de Y-as zag, was een 3 bits notatie. Hier wordt elk mogelijk niveau van onze digitale signaal met een unieke 3-bits combinatie weergegeven. Wel moeten we rekening houden met het spanningsbereik van onze ADC. Je kunt dit bereik zelf instellen. Wij gaan gebruik maken van een meetbereik tussen GND en de voedingsspanning (3.3V). Een 000 codering zou bijvoorbeeld staan voor 0V, en een 111 codering voor 3.3V. Onze ADC heeft een bitsresolutie van 12, dat betekent dat bij ons de 0V als 0000 0000 0000 wordt weergegeven, en de 3.3V als 1111 1111 1111. Afhankelijk van de gekozen referentiespanning zal een bepaalde bitcodering met een andere analoge spanning overeenkomen. Wanneer we een hogere of lagere spanning kiezen, zal die overeenkomen met de maxima. We maken een kort uitstapje naar het ADC0 Reference Register (21.5.1).

232. Op welke referentiespanning staat de ADC standaard ingesteld?

233. Op welke resolutie kan onze ADC meten in het geval van de standaard referentiespanning?

234. Neem de volgende tabel over en vul deze in voor de verschillende referentiespanningen

Reference Voltage (V)	Resolutie van 1 bit (V)
1.024	
2.048	
4.096	
2.500	
3.3 (VREFA)	

De laatste optie, namelijk de VREFA (3.3V) zorgt ervoor dat onze ADC de voedingsspanning van de uC als referentie gebruikt. Dat is dezelfde spanning die we op een output poort zetten, wanneer we deze hoog maken. Wanneer we straks in het practicum bezig gaan met de potentiometer op de SMU-print is het handig dat je je voedingsspanning gebruikt voor de ADC als referentie.

235. Leg uit waarom dit zo is.

We gaan terug naar het hoofdstuk over de ADC. Wanneer we onze ADC straks geconfigureerd hebben, kunnen we een analoge conversie starten door deze vanuit software of hardware (event system) triggeren. Het eventsystem gaan we later behandelen. De ADC zal deze conversie uitvoeren waarna het resultaat opgeslagen wordt in een bepaald register.

236. In welk register wordt het resultaat van een AD conversie geplaatst?

In de datasheet staat dat we tot 22 verschillende inputs hebben voor onze ADC. Dit betekent niet dat we 22 aparte ADC's hebben. Toch kunnen we met 1 ADC meerdere analoge inputs uitlezen

237. Op welke manier zouden we 22 verschillende inputs kunnen uitlezen?

Wanneer we meerdere analoge uitlezingen met 1 ADC willen doen kan dat nadelig zijn voor de sampling rate.

238. Leg in je eigen woorden uit waarom dit zo is.

Beantwoord de volgende vraag zonder het datasheet. We weten dat een AD conversie tijd kost.

239. Op welke manier zouden we kunnen weten wanneer de AD conversie klaar is?

Figuur 33 laat zien hoe we een systeem kunnen opzetten waarin we een continue sampling van een analoge pin kunnen krijgen. Links zien we een analoge input (spanning) die met behulp van de ADC omgezet wordt naar een digitaal signaal. De ADC samplet op basis van een trigger die hardwarematig vanuit de timer via het event systeem naar de ADC geleid wordt. Wanneer de conversie klaar is, genereert de ADC een interrupt die de CPU vertelt dat de ADC klaar is. Het digitale resultaat kan dan gebruikt worden door de CPU, en dus in software verder verwerkt worden.

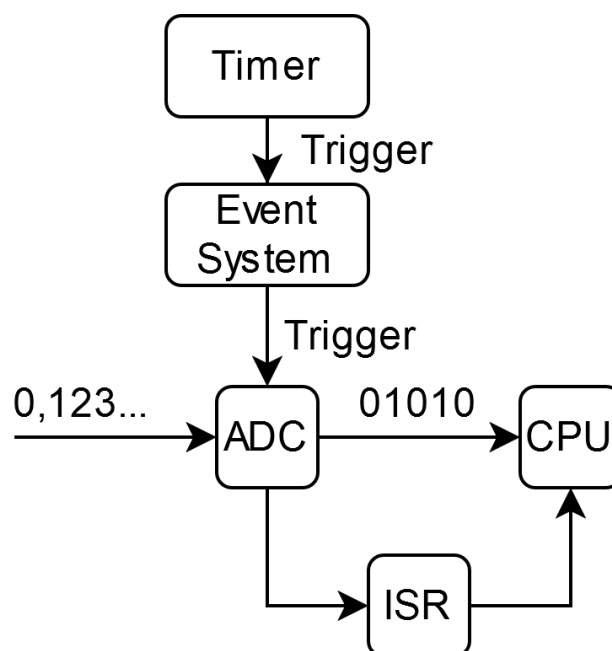


Figure 33: Een timer genereert een periodieke trigger. Deze wordt via het event system doorgeschakeld naar de ADC. Op basis van de trigger samplet de ADC een externe bron (bijvoorbeeld een pin). Het digitale resultaat genereert een interrupt en wordt doorgegeven aan de CPU.

### 5.1.3 Event System

Stel je wil bellen met een kameraad. Dan toets je zijn telefoonnummer in en krijg je hem zeer snel aan de lijn. Dit systeem van directe verbinding maakt dat we snel terecht komen bij de juiste persoon, zonder dat er iemand tussen hoeft te zitten. Vroeger was dit wel anders. Wanneer je toen wilde telefoneren, werd je door een telefoonoperator doorverbonden, zodat je bij de juiste lijn uitkwam. Dit proces vindt tegenwoordig volledig automatisch plaats.

Deze parallel vindt ook plaats op de uC. Kijk maar in figuur 34. Hier zie je de twee bovenstaande modellen geïllustreerd. Links is het langzame model via de operator (CPU), en rechts het directe snelle en automatische model via het event (EVNT) system.

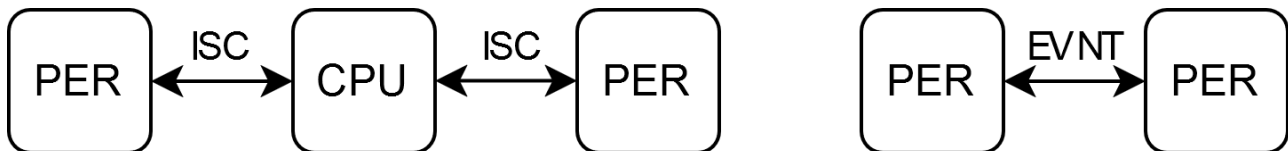


Figure 34: Links zien we een systeem waar we via de CPU worden doorverbonden. Rechts zien we een directe verbinding tussen 2 peripherals (PER). Het is bijna altijd sneller om een directe verbinding te hebben, dan eerst via een CPU te moeten.

Waarom is een directe verbinding nou sneller? De reden hiervoor zit hem er voornamelijk in, dat wanneer we via de CPU met een andere peripheral gaan praten, we dit via de software moeten doen. De CPU heeft maar een beperkte verwerkingssnelheid. Deze wordt ook nog eens gedeeld met allerlei andere processen. Dat betekent dat timing en snelheid niet altijd optimaal zullen zijn. Het is veel sneller, efficiënter en nauwkeuriger om een directe hardware verbinding tussen twee peripherals te hebben. Dit systeem is op onze uC uitgevoerd met de event system (of event bus).

In figuur 33 zagen we voor het eerst het event system. In figuur 35 zien we dit systeem in een nieuw jasje, waarin we het samen met de databus hebben afgebeeld. Bussen worden gebruikt om eenvoudig te kunnen communiceren met veel verschillende peripherals. Het gebruik van een bus maakt het overzicht niet alleen makkelijker te begrijpen, het is daarnaast qua architectuur veel eenvoudiger te maken. Zo hoeven er geen aparte verbindingen meer te zijn tussen elke peripheral naar elk andere peripheral, maar zijn er per peripheral slechts enkele verbindingen naar de bus voldoende. In het geval van bijvoorbeeld 20 peripherals, zou je met een bussysteem hooguit  $2 \cdot 20 = 40$  verbindingen hebben. In een direct peer 2 peer connection, zou je er  $19 + 18 + 17 + \dots + 1 = 190$  verbindingen nodig hebben. Dit verschil wordt per extra peripheral groter. Daarnaast zouden die peer 2 peer verbindingen allemaal kris kros door elkaar heen lopen, wat het systeem er niet bepaald overzichtelijker van maakt. Toch is het communiceren via de bus, ondanks dat we een tussenpersoon (de eventbus) hebben, veel sneller dan communiceren via de uC. Dit heeft, zoals we al eerder aangaven, te maken met dat we geef softwareverwerking, maar hardwarematige implementatie hebben.

Naast dat het event system sneller en nauwkeuriger is, is het ook energiezuiniger. Dit komt doordat de CPU überhaupt niet wakker hoeft te worden. Terwijl peripherals met elkaar praten, kan de CPU slapen (sleepmode). Dit is nog gunstiger voor het energieverbruik. In veel embedded toepassingen is het namelijk van belang om zo weinig mogelijk energie te verbruiken. Dus elk moment van CPU-tijd is dan vaak erg kostbaar. Wanneer we dat kunnen verminderen, zorgt dat voor een langere batterijtijd, en dus minder afval en een betere gebruikservaring.

Om het event system op een goede manier te laten werken, moeten we wel afspraken maken over hoe peripherals met elkaar communiceren. Wanneer we kijken naar de implementatie van het

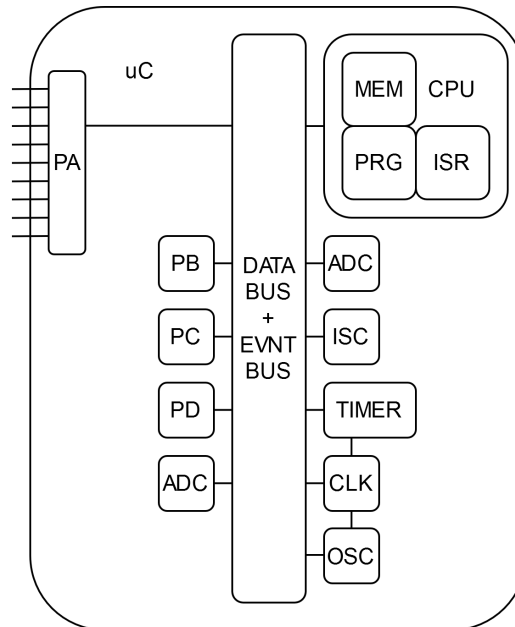


Figure 35: Een vereenvoudigde weergave van het uC-schema uit het datasheet (p.13). We zien hier dat er een event en een databus is, waar bijna alle peripherals (blokken) aan verbonden zijn.

event system op onze uC, dan zien we dat dit gedaan wordt met een event source, een event channel en een event destination. We illustreren de werking hiervan aan de hand van een voorbeeld naar het model in figuur 33. Dit voorbeeld gaan we ook programmeren in het practicum. Stel we willen de lijnsensor uitlezen. Om dit te kunnen realiseren dienen we een periodieke triggerring van onze ADC te maken. Dit kunnen we doen met een timer. Deze timer kan naast een interrupt ook een event genereren, waarop onze ADC geabonneerd kan zijn. Abboneren betekent dat we dus gevoelig zijn voor dat event, en een handeling starten wanneer dat event optreedt. Deze handeling staat soms vast (bij de ADC is dit sample nemen), soms kunnen we daar nog weer verschillende acties op laten uitvoeren. Om een abonnement tussen event source en destination fysiek te kunnen creëren, moeten we een kanaal tussen deze twee maken. Dit kanaal is de event channel, en kan dus flexibel ingesteld worden afhankelijk van de gewenste verbinding. Op het moment dat de timer het event genereert zal de ADC dus automatisch starten doordat dit doorgegeven wordt via deze event channel. Wanneer de ADC klaar is genereert hij een interrupt. Het resultaat van de ADC kan daarna bijvoorbeeld door de software verder afgehandeld worden.

Alhoewel bovenstaande methode nog niet het meest efficiënt is, besparen we al aanzienlijk door niet meer vanuit software de ADC te starten. Wanneer we bijvoorbeeld 100 samples per seconde zouden nemen, scheelt dat al aanzienlijk in het energiegebruik van onze processor. Wanneer we het ADC resultaat ook nog zouden afhandelen met een event, besparen we nog meer.

#### 5.1.4 Real Time Clock

De realtime clock (RTC) is een speciale nauwkeurige clock met timerfuncties. Deze staat standaard aangesloten op de oscillatorfrequentie van 32,678 kHz. Deze frequentie wordt behaald door middel van een interne oscillator bron, maar kan eventueel ook met het quartz kristal op de PCB van de Curiosity Nanoverbonden worden. De 32,678 kHz frequentie is een veelgebruikte standaard voor RTC's, aangezien 32678 gelijk is aan  $2^{15}$ . Wanneer we in dat geval dividers toepassen (dus deze frequentie delen door een 2-macht), kunnen we allerlei handige frequenties krijgen. Denk bijvoorbeeld aan 1024 x per seconde (delen door 32) of 128x per seconde (delen door 8192).

Met de RTC kunnen we daarom zeer eenvoudig tijdsgebonden triggers genereren. Deze kunnen we allerlei kanten opsturen via het event system. We kunnen namelijk verschillende peripherals aan de RTC verbinden die bijvoorbeeld 4x per seconde moeten gebeuren. Dit is erg handig wanneer we bijvoorbeeld periodiek een ADC willen samplen. De voordelen van de RTC zijn dat hij erg nauwkeurig en energiezuinig is.

In principe geldt dat als je iets periodieks, energiezuinig wilt triggeren je typisch een RTC gebruikt. De gewone timers gebruik je vooral als de exacte timing/periodiciteit of energieverbruik iets minder belangrijk is. Een PWM zal je dus met een timer generen, en niet met de RTC, want het maakt voor de werking van de PWM niet zoveel uit of de timer nauwkeurig is.

We gaan nu niet verder de diepte in op de RTC. Je krijgt instructies tijdens het practicumgedeelte hoe je de RTC kunt gebruiken.

### 5.1.5 UART/USART & Data Visualiser

We breiden het model wat we hierboven demonstreerden wat verder uit. Hoe is onze data eigenlijk zichtbaar? We kunnen een breakpoint plaatsen en via de debugger meekijken. Dat werkt aardig goed, maar zorgt toch voor een relatief onhandige situatie. Elke keer dat we een nieuwe meting doen moet de debugger de processor pauzeren en moeten we dit tekstueel bekijken op onze pc. Handiger zou zijn als we dit zouden kunnen streamen via een simpele interface en dit kunnen visualiseren. Dit kan! Heel eenvoudig met een UART verbinding en de Data Visualiser van MPLAB. Het hele systeem is nu zichtbaar in figuur 36.

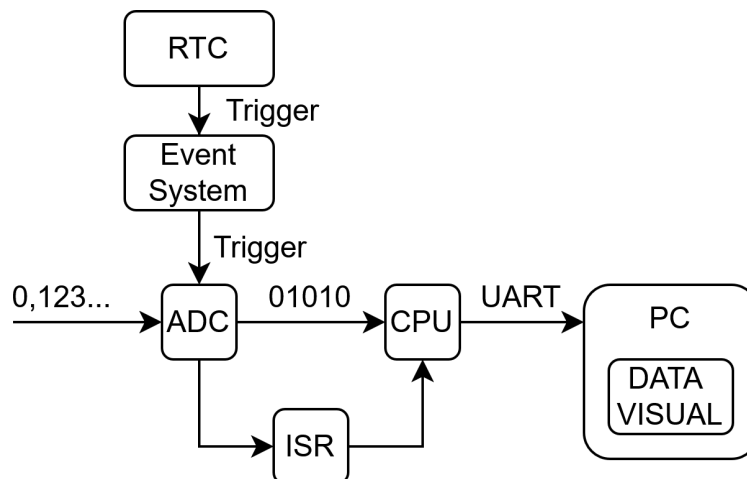


Figure 36: We zien dezelfde chain als eerder. Echter sturen we nu het signaal door via de UART naar de PC. Op de PC laten we het signaal visueel zien met behulp van de Data Visualiser

De U(S)ART (Univeral (Synchronous and) Asynchronous Receiver and Transmitter) is een serieel communicatieprotocol wat we kunnen gebruiken om data te versturen. We kunnen deze interface in combinatie met de Data Visualiser gebruiken om in realtime data uit te kunnen lezen. We gaan niet heel erg de diepte in met hoe een UART verbinding in detail werkt. Wat wel goed om te weten is, is dat een UART een digitaal signaal is, instelbare BAUD-rate heeft (hoeveel data versturen we elke seconde) en twee lijnen RX (receive) en TX (transmit) gebruikt om beide kanten op te kunnen communiceren. In het voorbeeld uit figuur 36 gebruiken we alleen de transmit. We sturen in dit practicum geen data terug naar de uC, maar dit kan wel!

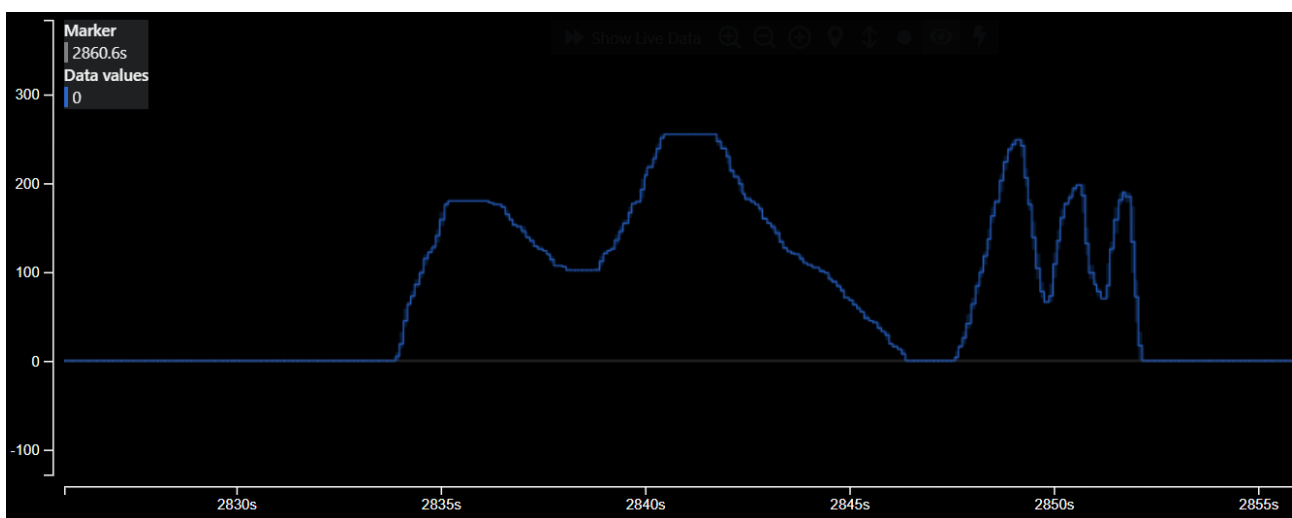


Figure 37: Analoge uitlezing van de potentiometer, verstuurd d.m.v. 8-bit UART en geplot met Data Visualiser

Wanneer we een UART-verbinding opzetten, komt er een communicatiekanaal tot stand waar continu nieuwe informatie van A naar B gestuurd wordt. De Data Visualiser is een plot tool waarmee je die inkomende seriële data kunt visualiseren als een bewegende real time grafiek. Een voorbeeld hiervan is zichtbaar in figuur 37.

De UART verbinding is een serieel protocol waarmee we characters versturen. Je stuurt dus een char (8-bits) door een combinatie van 1'en en 0'en op de lijn te zetten. Het asynchrone stuk houdt in dat we hier geen rekening houden met een klok. Er is niet een kloklijn aanwezig die de uitlezing van het signaal synchroniseert. Dat betekent dat we van te voren moeten afspreken hoe snel we met elkaar praten, zodat beide kanten weten hoe snel ze moeten luisteren. Dit noemen we de BAUD-rate. Een voordeel van asynchroon is dat we geen kloklijn nodig hebben en dat het daarmee makkelijker te realiseren is, een nadeel is, dat wanneer de CPU met iets anders bezig is, we gemakkelijk een bericht kunnen missen. Let op dat jij zelf bepaalt wat je stuurt op de UART. Wanneer je geen bericht verstuurd, zal er ook niets versturen. We kunnen dit bericht maken in software, of door peripherals laten genereren (maar dat gaan we niet doen in dit practicum). Het kan daardoor zijn dat je ondanks een hele hoge Baud-rate, geen hoge communicatiesnelheid hebt omdat je bijvoorbeeld niet zoveel berichten verstuurt.

## 5.2 Practicum

We gaan beginnen met het aanmaken van een nieuw project. We noemen dit week5\_adc. We gaan nu stuk voor stuk verschillende onderdelen configureren zodat we de potentiometer kunnen uitlezen op onze pc. In principe gaat ons systeem er uitzien zoals aangegeven in figuur 36.

Nadat je je project hebt aangemaakt en MCC geopend hebt, moeten we een aantal verschillende peripherals configureren. Dit zijn de ISR, de ADC, de RTC het event system en de UART-verbinding.

### 5.2.1 ISR

We beginnen met het aanzetten van de globale interrupts.

240. Ga naar de Interrupt manager en zet de global interrupt enable aan.

### 5.2.2 ADC

We voegen de ADC als peripheral toe aan ons project.

241. Ga naar project resources en voeg daar de ADC0 toe.

242. Wanneer dit nog niet gedaan is, enable je de ADC.

243. In het configuratiescherm klik je op start event input enable

244. Scroll naar beneden naar interrupt settings en zet de result ready interrupt enable aan.

Start event input enable maakt het mogelijk om de ADC via het event system te laten starten (na een trigger vanuit de RTC). Result ready interrupt enable zal een interrupt genereren wanneer de ADC klaar is met samplen. Deze interrupt moet je dus straks gaan afhandelen!

We willen in eerste instantie een uitlezing van de potentiometer doen. De potmeter is een handig component waarmee we verschillende spanningen kunnen genereren (variabele spanningsdeler) door er simpelweg aan te draaien. Om de potmeter aan te sluiten op de ADC dienen we hiervoor de juiste pin te selecteren (onder component signals).

245. Kijk op het elektronisch schema van de SMU en kies de juiste pin zodat we de potmeter straks kunnen uitlezen met de ADC.

De ADC heeft 22 verschillende channels die met GPIO pinnen verbonden zijn. Deze 22 pinnen kunnen dus analoog uitgelezen worden. PD0 is gekoppeld aan AIN0, PD1 aan AIN1 en er wordt doorgeteld t/m PF5 (AIN21) etc. Deze koppeling is vindbaar in het datasheet en het is ook zichtbaar in de pin grid view.

246. Open de pin grid view en zet de juiste pin als analoge pin



247. Ga terug naar de ADC0 configuratie en selecteer onder positive input channel de correcte channel. Deze wordt niet automatisch goed ingesteld!

Wanneer je het niet zeker weet, vraag na bij je buurman/vrouw of je docent.

### 5.2.3 VREF

We stellen de VREF in zodat we tussen een geschikte spanning en 0V kunnen meten. Navigeer hiervoor naar de VREF-peripheral. Deze is automatisch toegevoegd aan het systeemoverzicht wanneer je de ADC hebt toegevoegd.

248. Ga nu Project Resources en klik op VREF.
249. Stel VREF voor de ADC in op de supply voltage (VDD)

### 5.2.4 RTC

Om onze ADC periodiek te laten samplen gaan we de Real Time Clock (RTC) gebruiken. We kunnen de RTC als peripheral toevoegen binnen MCC.

250. Voeg een RTC instantie toe aan je project (zie Device Resources - Timer)

De RTC is standaard verbonden met het 32,768 kHz kristal. We kunnen de klok langzamer laten lopen door een prescaler toe te passen. We gaan nu eerst naar de clock configuration kijken.

251. Enable de RTC
252. Welke prescaler zou je moeten toepassen wanneer je een frequentie van 1024 Hz wil? Stel de prescaler op deze waarde in.
253. Stel als laatste de clock period op 0.5 in.
254. Wat is nu de samplerate (Hz)?
255. Sluit de RTC configuratie weer af.

### 5.2.5 Event System

We gaan nu bezig met het configureren van het event system. We kunnen het event system weer als peripheral toevoegen binnen MCC.

256. Voeg een instantie van het event system (EVSYS) toe aan je project.
257. Dubbelklik op de nu geopende EVSYS tab, zodat deze naar het volledige scherm gaat.

We zien nu een interface met 3 kolommen. Generators, channels en users.

258. Ga voor jezelf na dat je begrijpt wat een event generator is, event channel en event user.

We gaan nu de RTC een event laten genereren waarmee de ADC getriggerd kan worden. Tip: het verbinden doe je door te klikken en te slepen.

259. Verbind de output van de RTC\_OVF (Real Time Counter overflow) met de ingang van Event channel 0.

260. Verbind de uitgang van Event channel 0 door met de ingang van ADC0START (Users kolom).

De ADC0 is nu geabonneerd op event channel 0 wat weer doorverbonden is met de real time counter overflow. Je kunt dit in het vervolg ook voor andere peripherals doen. In principe geldt er: er is 1 event generator verbonden met een event channel. Op een event channel kunnen meerdere event users geabonneerd zijn. Zoals je misschien al ziet heb je 10 event channels tot je beschikking. Meer dan genoeg om gave dingen te kunnen doen! Mocht je dit interessant vinden, dan kun je eens kijken welke peripherals er allemaal event triggers kunnen genereren, en welke peripherals event user zijn.

261. Kijk eens wat rond, en sluit daarna de EVSYS configuratie weer.

### 5.2.6 UART

Nu we het event system geconfigureerd hebben gaan we de UART (USART) configureren.

262. Voeg de USART/UART3 component toe.

We gaan de USART in asynchronous mode gebruiken. Ook gebruiken we alleen de transmitlijn. Dat betekent dat onze uC alleen data verstuurd, en niet ontvangt. Wanneer je ook data wil ontvangen dienen we de receiver ook aan te zetten. Dit gaan we pas in week 6 gebruiken. Wanneer we de UART met de computer willen verbinden voor de AVR128DB48 (curiosity nano) te gebruiken configureer je hem als volgt: (voor andere bordjes/chips kan dit anders zijn).

263. Configureer de USART als volgt:

(a) UART3

(b) Baud rate: 115200. Let op: dit moeten we ook als baudrate in de data visualiser gebruiken.

De UART3 zit verbonden met RXD: PB1 en TXD: PB0.

264. Sluit de UART configuratie af.

265. Klik op generate zodat het project gegenereerd wordt.

### 5.2.7 Uitlezing van de ADC

We gaan nu eerst ons project bouwen.

266. Klik op build.
267. Navigeer naar MCC generated files → adc → src → adc0.c
268. Kijk eens rond in de verschillende functies die je hier ziet en probeer voor tenminste 3 functies te bepalen wat er gedaan wordt.
269. Welke ISR functie wordt aangeroepen wanneer een ADC resultaat klaar is?

Omdat de focus van dit practicum op de ADC ligt, krijg je de code voor de UART cadeau:

Listing 4: Incomplete code voor het uitlezen van de ADC

```
#include "mcc_generated_files/system/system.h"
#include <avr/sleep.h>
#include <util/delay.h>

volatile uint16_t adc_res = 0;
volatile uint8_t sendflag = 0;

void ADC0_Interrupt_handler(){
    // TODO: Voeg hier de juiste ISR callback functie toe.
    // Kopieer de ADC uitlezing naar adc_res
    // Tip: gebruik ADC0.RES of de juiste functie
    // Maak sendflag 1.
}

/* Zet een char klaar op de transmitlijn van UART3.
Deze wordt hardwarematig verstuurd */

void usart_put_char(uint8_t data){
    while(!(USART3.STATUS & USART_DREIF_bm));
    USART3.TXDATA = data;
}

int main(void)
{
    SYSTEM_Initialize();

    // Koppel bovenstaande interrupthandlerfunctie aan ADC0 ISR.

    ADC0_ConversionDoneCallbackRegister(ADC0_Interrupt_handler);
```

```

// sendflag bepaalt of er een nieuw meting is
// We versturen de 8 meest significante bits (schuif 4 opzij)
// We reseten sendflag

while(1){

    if (sendflag){
        usart_put_char(adc_res >>4);
        sendflag = 0;
    }

}

```

De ADC0 is zo geconfigureerd dat deze een interrupt genereert wanneer hij klaar is.

270. Implementeer de juiste interrupthandlerfunctie voor de ADC0.

- (a) We willen in deze ISR het resultaat opslaan in de variabele `adc_res`.
- (b) Maak `sendflag` gelijk aan 1 (`sendflag = 1;`) zodat we in onze `while` weten dat we een ADC uitlezing paraat hebben.
- (c) MCC zet automatisch de interrupt flag voor ons uit. Waar staat de code hiervoor?

Wanneer je dit correct gedaan hebt, gaan we de code uploaden naar onze uC. Daarna willen we bekijken of de uC daadwerkelijk data verstuurd naar je laptop. Hiervoor gaan we aan de slag met de Data Visualiser.

271. Klik op run main project.

272. Open nu Data Visualiser (Zeshoek met DV erin)

273. Klik op het tandwielletje bij COMX (de juiste compoort kan verschillend zijn per laptop).

274. Zet ook hier de baudrate op 115200 (dit wordt vaak vergeten en zorgt dikwijls voor problemen).

275. voeg een raw plot line toe o.b.v. die compoort.

Wanneer alles goed gaat moet er in het grafiekje rechts een lijntje komen te staan wat elke halve seconde geüpdatet wordt. Is dit niet het geval, dan is er ergens iets misgegaan. Probeer dan systematisch te controleren wat er aan de hand is, of vraag dit na bij je docent.

276. Wanneer het lijntje zichtbaar is, draai eens aan de potmeter.

277. Constateer of de grafiek op en neer beweegt.

Wanneer de grafiek volledig van 0 t/m 255 op en neer gaat als we de potmeter van volledig de ene kant naar de andere kant draaien, hebben we alles goed ingesteld. Wanneer dit niet het geval is, controleer wat er aan de hand is of vraag dit na bij je docent.

### 5.2.8 Energiezuinige ADC

Onze ADC uitlezing is nog niet heel energiezuinig. We kunnen dit verbeteren door de processor te laten slapen wanneer we niets aan het doen zijn.

278. Heropen MCC

279. Voeg de Sleep Control toe. Device Resources → System → SLPCTRL.

280. Enable de Sleepmode in het SLPCTRL configuratiescherm.

281. Klik opnieuw op generate.

282. Plaats boven het if statement het command om in de sleep mode te raken (zie week 4).

We hebben nu de basis gelegd voor onze analoge meting. Je kunt dit vanaf nu gebruiken om allerlei verschillende analoge sensoren uit te lezen. We gaan volgende week de wittelij (CNY70) sensor erop aansluiten.

Twee keer samplen per seconde is nogal traag. Verhoog dit eens naar 50x per seconde.

283. Pas de instellingen van de RTC zo aan dat deze 50x per seconde een overflow heeft en daarmee de ADC0 kan triggeren.

De volgende opdracht kun je alvast uitvoeren als je tijd hebt. Je bereid je hiermee beter voor op practicum Hardware Design van Drive Exchange.

### 5.2.9 LED dimmen met Potmeter

Vorige week hebben we geleerd hoe we een PWM konden generen door een timer output door te koppelen aan een I/O poort. In deze opdracht gaan we de duty cycle actief veranderen op basis van de analoge waarde die we uitlezen vanuit onze ADC. De PWM zelf sluiten we in eerste instantie aan op de LED. We creëren hiermee een dimbare LED.

284. Voeg een Timer toe aan je project in MCC.

285. Configureer de timer zodat er op pin PB3 een output komt. Zorg ervoor dat je timer op split mode staat en controleer dat de juiste WO (waveform output) aan staat.

Wanneer je in eerste instantie je project genereert en runt, zal de LED volledig branden. We hebben namelijk een duty cycle van 0%. De compare waarde kunnen we actief aanpassen. Hiermee variëren we de duty cycle. We kunnen 1-op-1 de 8 meest significante bits van onze ADC uitlezing doorzetten naar het compare register.

286. Voeg een regel code toe waarmee je de PWM in de while lus aanpast naar de analoge waarde van de potmeter. Tip: gebruik hier hetzelfde voor als wat je verstuurd via de UART verbinding, incl. verschuiving.

## 6 State Machine

Het doel van dit practicum is het leren maken van een state machine en deze om te zetten in code. Hiervoor gaan we leren wat een state machine is en waarom dit een handige manier zou kunnen zijn om je programma mee te structureren.

### 6.1 Voorbereiding

Tot nu toe heb je een paar eenvoudige programmaatjes geschreven. Zo heb je bij programmeren basis in periode 1 verschillende problemen en oplossingen vertaald naar code. Wat je toen hebt gemaakt noemen we een algoritme. Dit is ook wel een stappenplan waarmee je een complex probleem stap voor stap op kunt lossen. Denk maar eens aan het voorbeeld toen je gevraagd werd om de kwadraten van de getallen 1 t/m 100 uit te rekenen en te printen op de console. Dit is een eindig probleem. Het probleem is opgelost wanneer we 1,4,9, ..., 10000 hebben geprint. Ons programma sluit dan af, en we zijn klaar. De meeste problemen die we in de dagelijkse praktijk willen oplossen zijn niet eindig. Denk eens na over de robot die je gaat ontwikkelen voor Dx.C.

287. Zouden we voor de robot een eindig programma kunnen schrijven?

288. Wat is het grote nadeel als we onze robot op basis van 1 algoritme gaan programmeren?

We hopen dat je tot de conclusie komt dat het niet heel makkelijk is om één algoritme te bedenken wat de robot van punt A naar punt B volgens een ongedefinieerd pad rijdt. Of een station ontwikkelt wat ongeacht wat er in de buitenwereld gebeurt precies van te voren weet wanneer hij een flesje moet oppakken of afleveren. De echte wereld is onvoorspelbaar en we weten ook niet precies wanneer we klaar zijn. Heel vaak zijn we nooit echt klaar. We willen dat onze systemen blijven werken. We zouden er niets aan hebben als onze theromstaat uitvalt omdat het programma afgelopen is, of dat de koelkast ermee ophoudt omdat hij geen instructies meer krijgt. De oplossing hiervoor is een ander programmeerparadigma, de finite state machine (eindige toestandsmachine). Waar we in algoritmisch programmeren kijken naar het oplossen van simpele taakjes (aan/uit, if/else, berekeningen) kijken we in de statemachine naar de zogenaamde state en statetransitielogica om te bepalen wat het gewenste gedrag is van ons apparaat.

### 6.1.1 State Machine - Introductie

De statemachine is een simpel idee dat we in elke state uniek gedrag vertonen. Gedrag kan het verwarmen van een ruimte zijn (thermostaat), het navigeren over een lijn (AGV), het wachten/idling van een station op een sensorinput (begin- en eindstation) of het wassen volgens een bepaald wasprogramma (wasmachine). Let er hierin op dat we heel algemeen beschrijven wat ons systeem doet in elke state. Er is nergens een functie die voor ons een wasprogramma kan uitvoeren. Het gedrag wat in de state plaatsvindt moeten we natuurlijk wel algoritmisch programmeren. Toch heeft het conceptueel opdelen van jouw systeem in dit soort states grote voordelen. Wanneer we in een bepaalde state zitten, is het niet nodig dat we hoeven te controleren op inputs die alleen in een andere state relevant zijn. Wanneer mijn televisie uitstaat, hoeft hij alleen maar gevoelig te zijn voor de aanknop op de afstandsbediening, niet op alle andere knopjes. Dit maakt het programma simpeler en beter gestructureerd, en maakt de kans op fouten veel kleiner. Daarnaast kun je in het geval van fouten deze veel eenvoudiger vinden, omdat deze dan bijna altijd plaatsvinden in een specifieke state. De hoeveelheid states is eindig (zoals finite/eindig al impliceerde) en dat betekent dat je van te voren goed moet nadenken over welke states je nodig hebt, en welke dus relevant zijn. Een typische FSM (finite state machine) heeft tussen de 2 en 8 states, waarbij elke extra state voor aanzienlijk meer complexiteit in het systeem zorgt. De truc is dus, om te zorgen dat je het juiste aantal states kiest, om ervoor te zorgen dat je een goed passende structuur krijgt.

Om een beeld te krijgen van hoe zo'n statemachine er nou typisch uit ziet, gaan we kijken naar een simpel voorbeeld. Dit is het voorbeeld uit figuur 38 waarbij we een drukknop (SW0) gebruiken om LED0 aan of uit te zetten. In week 1 of 2 heb je dit ook al eens geprogrammeerd, maar het illustreert heel goed wat de kern van statemachinelogica is. De beide states beschrijven gedrag van de LED (al zij het vrij simpel gedrag). De knop zorgt ervoor dat we van de een naar de andere state kunnen. Wanneer we in S\_OFF zitten vindt er een transitie plaats  $S\_OFF \rightarrow S\_ON$ , wanneer we in S\_ON zitten vindt er een transitie plaats van  $S\_ON \rightarrow S\_OFF$ . Het is heel belangrijk om te beseffen dat de transitie, of nog simpeler gezegd, de bestemming van die transitie, afhangt van onze huidige state. Wanneer we in de ene state zitten gaan we naar de ander en vice versa.

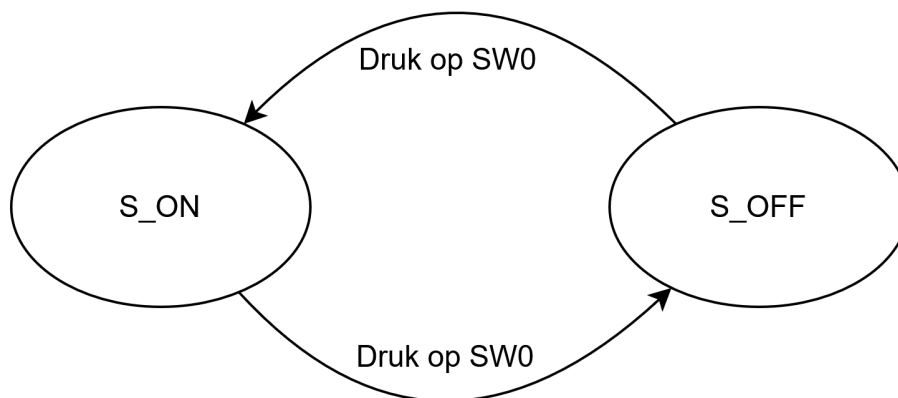


Figure 38: Een statemachine met twee states en twee transities. Er is een state waarin de LED aan staat (S\_ON) en een state waarin de LED uit staat (S\_OFF). Beide states hebben een transitiemogelijkheid naar de andere state. Deze transities worden op dezelfde manier getriggered, namelijk door op de knop SW0 te drukken. Dit voorbeeld lijkt een beetje suf, maar het is in weze precies hetzelfde als jij op je aan/uit knopt drukt op de afstandsbediening en daarmee de TV aan/uitzet.

Deze belangrijke tweede eigenschap van statemachines noemen we de statetransitielogica (of statelogica). In een statetransitie gaan we van de een specifieke state naar een specifieke andere toe. Een statetransitie vindt alleen plaats op een concrete trigger. Voorbeelden van triggers zijn typisch het indrukken van knopjes, timeouts, het binnenkomen van berichten of het afronden van een taak.

Al deze triggers kunnen met behulp van het interrupt en/of eventsysteem geregeld worden. We zitten eerst in state A, er vindt een trigger plaats die hoort bij een specifieke transitie, en dan gaan we 1x die transitie door naar state B. Hier hebben we weer andere mogelijke statetransities. Vaak is het niet wenselijk om elk willekeurige transitie mogelijk te maken.

Het zou kunnen zijn dat je van state A naar state B mag gaan, maar niet meer direct terug kunt. Door het gedrag binnen states, en de statetransitieloga kunnen we zeer complexe problemen oplossen. In weze zijn de meeste problemen op te breken in kleine stukjes die we heel handig aan elkaar kunnen koppelen in de vorm van een statemachine. Om dit te illustreren willen we dat je nadenkt over het volgende probleem: de snoepautomaat



### 6.1.2 State Machine - Voorbeeld Snoepautomaat

Stel je wordt gevraagd om een snoepautomaat te ontwikkelen met 16 snoeprepen en een betaalsysteem. De gebruiker moet een keuze kunnen maken, moeten kunnen betalen en de reep moet uitgegeven worden. Daarnaast moet het niet mogelijk zijn om zonder te betalen een reep te krijgen, mag er maximaal 1 reep per keer uitgegeven worden, en mag de gebruiker ook niet twee keer hoeven te betalen. Je werkt in een groep van 3 mensen, die samen dit project moeten uitvoeren.

289. Probeer voordat je de onderstaande tekst leest, eerst zelf te bedenken hoe je dit probleem zou aanpakken.

De snoepautomaat is een typisch voorbeeld van een ontwikkelprobleem. De gebruiker kan verschillende acties ondernemen, maar die acties hebben niet altijd dezelfde betekenis. Je moet een systeem bedenken wat duidelijk voorspelbaar gedrag vertoont op basis van de input van gebruikers. De volgorde van deze acties is heel belangrijk, want we willen dat de gebruiker eerst betaalt voordat hij zijn reep krijgt. Anders zou de gebruiker daar misbruik van kunnen maken, en weg kunnen lopen zonder te betalen. Ook zijn er 16 verschillende repen, waarvoor je echt niet 16 keer een apart systeem wil ontwikkelen. Sterker nog, we willen niet eens een aparte state per reep hebben, want dat zou ons systeem niet schaalbaar maken. Het zou een crime zijn om zo'n systeem te debuggen, want elke state zou ze eigen randgevallen en foutsituaties kunnen hebben. Wanneer we een variant van 32 repen hebben, zouden we onze systeemsarchitectuur totaal moeten omgooien.

De oplossing is de statemachine, waarbij we de stateloga dus lostrekken van de uitvoering van acties. Hiermee maken we het niet alleen conceptueel duidelijk (d.m.v.) een handig schema, ook kunnen we kijken of we alle randgevallen hebben meegenomen. Kijk maar eens naar figuur 39.

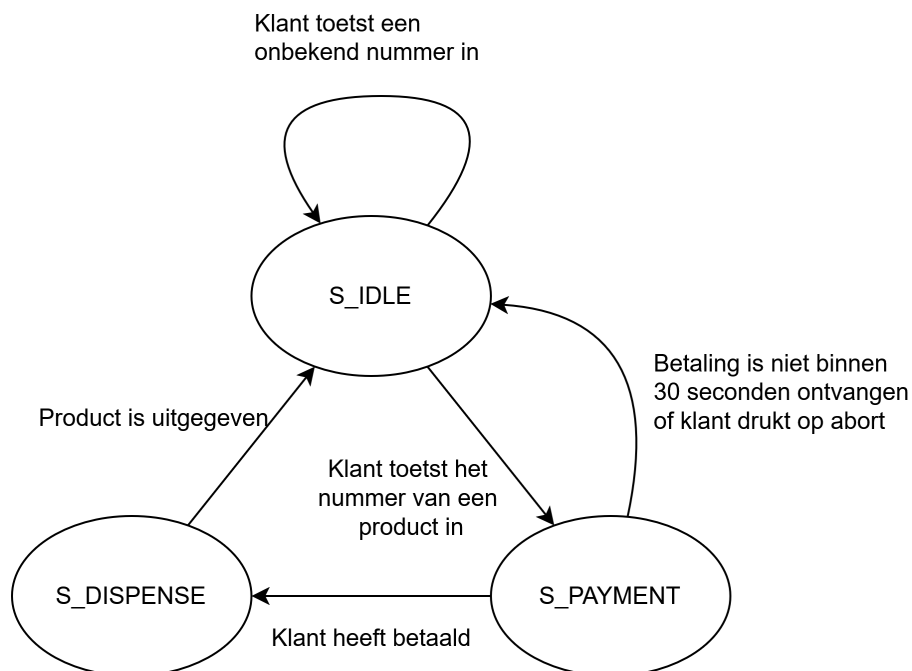


Figure 39: Een statemachine van een snoepautomaat. Er zijn 3 states, en 5 mogelijke transitie's. Sommige transitie's leiden terug naar dezelfde state, en niet elke transitie is mogelijk.

We zien in dit voorbeeld 3 states, S\_IDLE: de automaat doet hier niets en wacht op input van de gebruiker, S\_PAYMENT: de automaat wacht op een betaling door de gebruiker en S\_DISPENSE: De

automaat geeft de juiste reep uit (wat de juiste reep is wordt algoritmisch bepaald, en hoort dus niet thuis in de statel logica). Daarnaast zien we een wat complexere transitie logica. Zo kunnen we op 3 manieren in S\_IDLE komen, wanneer een product is uitgegeven (vanuit S\_DISPENSE), wanneer we een onbekend nummer in toetsen (vanuit S\_IDLE) of wanneer we te lang hebben gewacht met een betaling (vanuit S\_PAYMENT). De enige manier om in S\_PAYMENT te komen is vanuit S\_IDLE door een correct nummer in te voeren, en de enige manier om een product uit te geven is nadat een betaling heeft plaatsgevonden vanuit S\_PAYMENT. Hiermee hebben we er automatisch voor gezorgd dat we nooit een onbetaalde reep uitgeven, of dat we per ongeluk de gebruiker 2 keer laten betalen.

Een typische gang van zaken door deze statemachine zou zijn: De automaat staat standaard op S\_IDLE. Wanneer wij het nummer van een chocoladereep hebben ingetoetst, komt de machine in de S\_PAYMENT terecht. We kunnen nu kiezen om het proces afbreken of het systeem kan een timeout krijgen. In beide gevallen gaan we terug naar de idle state. Wanneer we in plaats daarvan onze betaalpas tegen de automaat aanhouden tijdens de betalingsstate, zal er een betaling plaatsvinden, en gaan we onomkeerbaar naar de dispense (uitgifte) state. Dit zorgt er voor dat de machine de reep uitgeeft, en dat wij de transactie niet meer kunnen afbreken, en het geld niet terugkrijgen. Na de uitgifte gaat de machine weer naar de idelstate. We kunnen in de dispense state niet direct terug naar de betaling state. Dit is natuurlijk ook onwenselijk, want we willen niet 2x moeten betalen, en de fabrikant wil ook niet dat jij je geld terug zou krijgen wanneer je teruggaat, terwijl je wel je reep gekregen hebt.

Dit klinkt natuurlijk triviaal, maar je moet je goed beseffen dat het systeem altijd correct moet werken! Dit is dus niet een systeem met een technicus ernaast in het lab hoeft te werken, maar een systeem wat op elke willekeurige plek moet blijven functioneren. Als ons systeem een bug bevat, of foutief gedrag vertoont, betekent het dat we in het gunstigste geval een dure monteur langs moeten sturen. Veel erger is dat we klanten, geld of aanzien verliezen, omdat ons systeem onbetrouwbaar is. Je kunt vast wel de momenten herinneren waarop de koffieautomaat het niet deed, maar denk eens na over een online betaling of een satteliet. Bij die voorbeelden is het nog veel duidelijker wat de impact is van een foutje. De tolerantie voor fouten is dan zo goed als nul. Om die tolerantie op nul te krijgen, moet je je systeem zo ontwikkelen dat het zo goed als onmogelijk wordt om een foutieve volgorde van handelingen te krijgen.

Omdat we ons probleem zo helder geformuleerd hebben kunnen we ervoor zorgen dat we er 100% zeker van zijn dat we het gedrag van de machine altijd weten. Deze scenario's kunnen van tevoren grondig ontworpen worden, en daarna 1-op-1 in code omgezet worden. De kans op fouten in je programma neemt hiermee aanzienlijk af.

Daarnaast is het mogelijk om de algoritmische code op te splitsen zodat alledrie de projectgenoten zelfstandig hun eigen taak kunnen uitvoeren. Zo kan er een iemand het keuzemenu maken (in S\_IDLE), een het betaalsysteem ontwikkelen (in S\_PAYMENT) en een het uitgiftesysteem (in S\_DISPENSE). Niet alleen hebben we ons probleem versimpeld, de kans op fouten geminimaliseerd, ook hebben we voor iedereen een duidelijke en uitvoerbare taak gecreëerd. Doordat deze taken parallel uitgevoerd kunnen worden, wordt je team vele malen efficiënter en effectiever.

Let op! Een statemachine is dus geen software flowchart. Een software flowchart wordt gebruikt voor algoritmes, een statemachine wordt gebruikt voor gedrag en systeem logica. Een software-flowchart heeft duidelijke acties en stappen die zo snel mogelijk in één richting worden uitgevoerd. Een flowchart heeft een begin en een einde. Een statemachine kan veel flexibeler gedrag vertonen, kan heel lang of heel kort in een bepaalde state blijven zitten, en heeft typisch een onbepaalde tijd dat het functioneert. Het heeft soms een begin (wanneer ik mijn systeem voor het eerst aanzet),

maar nooit een einde. Het heeft in de context van een statemachine geen zin om heel snel acties uit te voeren, omdat de statemachine juist continu moet blijven functioneren. In de context van een flowchart is het juist wel handig om zo snel mogelijk te zijn, omdat dat je programma verbetert en je sneller een antwoord hebt. Om het verschil tussen flowcharts en statemachines helder te houden hanteren we rechthoekige blokken en 1 pijl per blok in een flowchart, en hanteren we ovalen en een wisselende hoeveelheid pijlen per state.

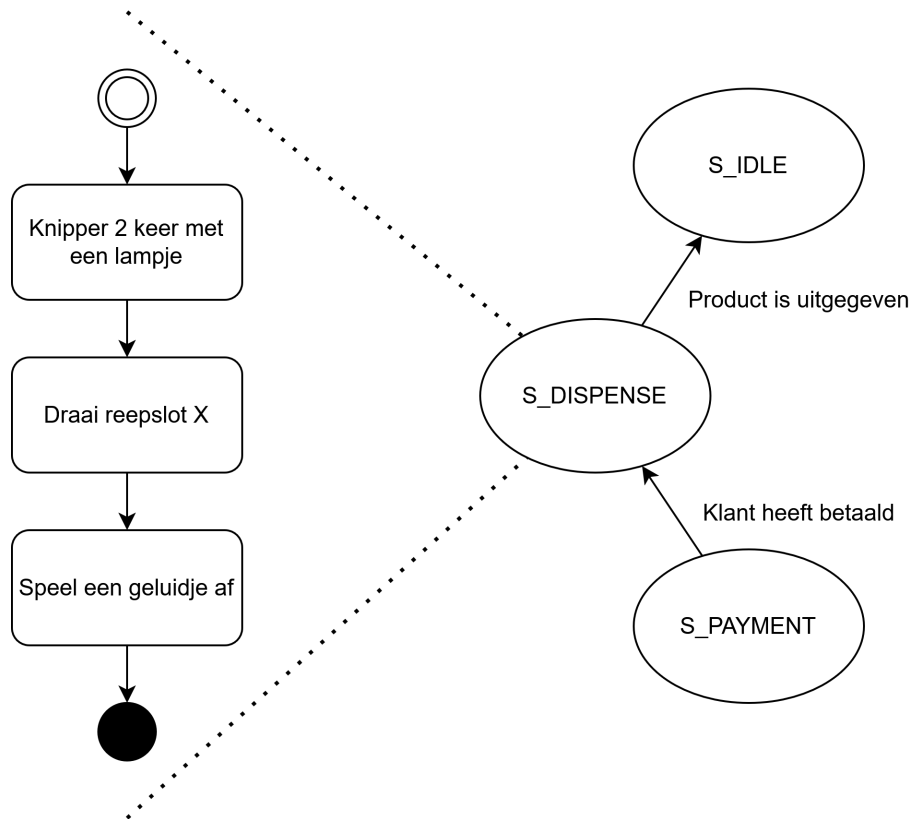


Figure 40: We zien links een flowchart (rechthoeken) en rechts een gedeelte van een Statemachine (ovalen). Het flowchart, wat laat zien hoe een product uitgegeven wordt (modelleert de acties binnen S\_DISPENSE), laat stap voor stap zien wat de automaat moet doen. Er is een duidelijk begin en een duidelijk einde. De statemachine laat de systeemsarchitectuur zien. De State Machine wacht op een trigger om weer verder te kunnen naar S\_IDLE. Deze komt pas wanneer de flowchart klaar is.

Het kan overigens wel zijn dat je voor een bepaalde state een flowchart maakt. Dit is geïllustreerd in figuur 40. Wanneer je bijvoorbeeld in de dispense state zit, is het logisch dat je o.b.v. een aantal parameters zo snel mogelijk de juiste reep uitgeeft en kenbaar maakt aan de klant dat een reep uitgegeven wordt. Wanneer ik dat met een interne statemachine zou gaan modelleren, zou dat het probleem juist weer complexer kunnen maken. Wanneer je een statemachine of een flowchart toepast hangt dus af van de mate van detail waarin je programmeert. Wanneer je op systeemniveau programmeert, hanteer je typisch een statemachine. Wanneer je de detailimplementatie maakt hanteer je typisch een algoritmische aanpak (flowchart). Let er op dat dit richtlijnen zijn, en dat het dus van project tot project kan verschillen waar deze grens precies ligt.

290. Kijk nog eens naar figuur 12 (flowchart) en figuur 38 (statemachine). Deze twee programma's lijken erg op elkaar, maar er is een cruciaal verschil. Op welke manier zijn deze twee systemen anders?

Nu we een aardig beeld hebben van statemachines gaan we nog wat meer de diepte in.

291. Lees de volgende websites door:

- (a) [State Machine Tutorial 1](#), [State Machine Tutorial 2](#), [State Machine Tutorial 3](#).
- (b) Bekijk wat filmpjes/ lees wat websites door over state machines.

292. Wat is een state? Probeer dit eens zo nauwkeurig mogelijk te beschrijven.

293. Hoe zou je van een bepaalde state naar een andere state kunnen geraken?

294. Stel je voor dat we een koffiezetapparaat hebben met 1 knop. Het koffiezet apparaat heeft 3 states: Koffie\_Zetten, Warm\_houden, Uit. Geef voor de volgende begrippen een voorbeeld voor deze state machine. Licht je voorbeeld toe.

- (a) Transition
- (b) Event
- (c) Activity

295. Leg in je eigen woorden uit wat het verschil tussen een state machine diagram en een flowchart is.

296. Zou je voor de volgende problemen beter een flowchart of een statediagram kunnen maken? Licht toe.

- (a) Het berekenen van de eerste 100 priemgetallen
- (b) Programma van een koelkastbesturing
- (c) Een weerstation wat elk uur een meting verstuurt
- (d) Programmeren van een inschrijfformulier voor een muziekvereniging

Stel je eens voor hoe de robot het parcours af zou kunnen leggen.

297. Bedenk welke states de robot zich in kan bevinden. Tip: Denk hierbij aan de handeling die de robot op dat moment moet uitvoeren. Twijfel je? Kijk dan nog eens goed in de handleiding van Drive Exchange.

298. Welke state transitions heb je allemaal?

299. Wat voor events heb je?

300. Wat voor activiteiten zijn er?

301. Teken een state diagram van je DxC station. Schrijf de naam van elke state duidelijk op, teken de transities als pijlen en geef aan per transitie wanneer deze optreedt.

In het practicum gaan we aan de slag met het omzetten van ons statemachine model naar code. Wellicht heb je daar al een idee bij.

302. Hoe zou je een state diagram in code kunnen maken?

Vaak gebruiken we enums om de states, transitions en events mee te programmeren.

303. Leg in je eigen woorden uit waarom de enum hiervoor een geschikte variabele is.

## 6.2 Practicum

### 6.2.1 Statemachine in Code

Bij uC-programmeren heb je al verschillende keren een LED geprogrammeerd. We gaan zien hoe een statemachine het programma daarvoor nog beter kan maken. In deze opdracht gaan we een iets ingewikkeldere state-machine maken met 3 states en 3 overgangen. Deze ziet er als volgt uit.

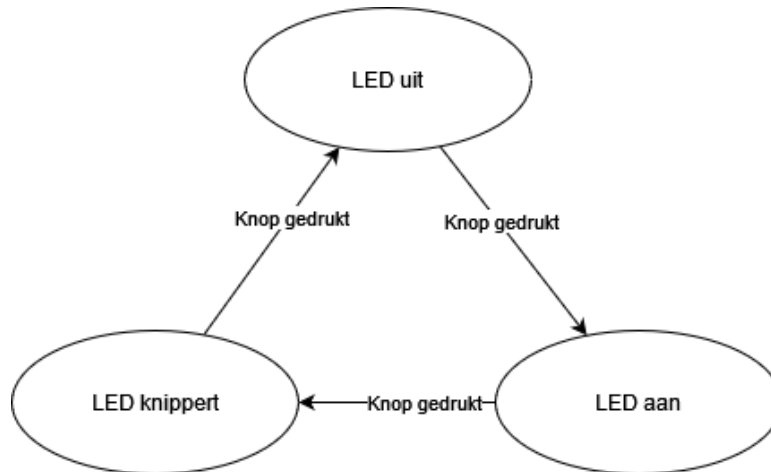


Figure 41: Een Statemachine met drie states. Elke state heeft een overgang naar een andere state

We beginnen met de standaarddingen.

304. Maak een nieuw project aan genaamd week6\_statemachine\_LED en configureer de LED0 en SW0 (pull en interrupt).
305. Zet ook de global interrupt enable aan.
306. Genereer je project.

Om van een state machine diagram code te maken, gaan we een aantal stappen doorlopen. We beginnen met het coderen van states en overgangen in enums. Dit zijn handige variabelen om een beperkte hoeveelheid discrete mogelijkheden weer te geven. Enums werken vergelijkbaar met integers.

Listing 5: Enums voor states en events

```
typedef enum{S_OFF, S_ON, S_BLINK} states;
typedef enum{E_PRESS, E_NOPRESS} events;

states currentState = S_OFF;
events currentEvent = E_NOPRESS;
```

307. Neem onderstaande code over in main.c en plaats deze boven je main functie.

Om een statemachine correct te implementeren moeten we twee dingen doen. We moeten bijhouden wat onze huidige state is (currentState) en we moeten bijhouden wat ons huidig event is

(currentEvent). De state zal voor zich spreken. De event is de trigger die op dat moment plaatsgevonden heeft. Aangezien we 1 mogelijke trigger hebben (druk op knop) hebben we twee mogelijke events. Om praktische redenen is het heel handig om ook een default event te hebben waarin er niets gebeurt is.

308. Waarom heb je een default event nodig?

Om nu de statemachine op te bouwen gaan we gebruik maken van de switch case. Dit is een heel handige manier om een aantal verschillende mogelijkheden tegen elkaar af te wegen. Ideaal voor een statemachine. Herinner je van Programmeren Basis dat een if else vaak gebruikt wordt om bereiken (groter dan, kleiner dan) te bepalen. Een switch case kan alleen maar discrete mogelijkheden bepalen (=1, =2, =3). Dit is ideaal voor de finite state machine, omdat deze alleen maar discrete states heeft. Daarnaast is het CPU-technisch efficiënter om een switch case te gebruiken in plaats van een if else. We gebruiken voor onze stateselectie dus een switch case en geen if else. Dit is een belangrijke structuurbepaling en voorkomt veel fouten.

309. Maak binnen je while loop een switch case aan op currentState en voeg een case toe voor elke mogelijke state. Let er op dat je elke case afsluit met een break;.

Deze switch case kijkt dus elke iteratie van de while lus naar de waarde van currentState. Afhankelijk van die waarde kunnen we straks de juiste acties uitvoeren. Nu we de basisstructuur van onze statemachine hebben moeten we gaan nadenken over de transities. Die doen we op basis van een trigger (currentEvent). Hiervoor gebruiken we opnieuw een switch case, die we nu in elke case van de state machine plaatsen. De reden hiervoor is simpel, in elke state hebben we een aantal mogelijke overgangen naar een andere state. We controleren dus elke keer dat we door de state machine heen gaan, of we een bepaalde trigger (event) gehad hebben, waarna we een eventuele transitie gaan maken. Hiervoor hebben we wel een extra state variabele nodig: nextState.

310. Maak een nieuwe globale variabele aan nextState van het type states. Deze heeft de default waarde S\_OFF (dus in eerste instantie hetzelfde als currentState).

Nu is de state transitie heel simpel. Wanneer currentState en nextState gelijk aan elkaar zijn, blijven we lekker in de currentState. Wanneer nextState anders is, moeten we blijkbaar naar een andere state toe. Hiervoor plaatsen we onder de statemachine een update statement wat onze currentState verandert naar dezelfde state als nextState.

311. Plaats onder de statemachine (maar nog wel in de while lus) een statement wat currentState de waarde toekent van nextState.

De truc is nu om tijdens elke state (case) te kijken of we de volgende iteratie naar een andere state toe moeten. We doen dit door met behulp van de geneste switch case op currentEvent te controleren of dit gebeurt. We moeten dit per state (case) opnieuw doen, want elke state heeft unieke transities. Zo kunnen we van S\_ON alleen naar S\_Blink, en van S\_Blink alleen naar S\_Off.

312. Maak per case een geneste switch case aan op currentEvent. Vergeet niet elke case af te sluiten met een break;

313. Programmeer binnen de elke geneste `currentEvent` switch case de juiste `nextState` toekenning. We willen dus dat de juiste waarde van `nextState` toegekend wordt, zodat we de volgende iteratie naar de juiste volgende state gaan.

Nu we de basisstructuur van onze statemachine klaar is. Moeten we nog een paar dingen doen. We moeten ons `currentEvent` nog bepalen. En we moeten nog het gedrag per state implementeren. Maak nu de volgende functies aan:

Listing 6: Te programmeren functies voor de statemachine

```
void setEvent(events newEvent); // Set currentEvent naar newEvent
void resetEvent(); // Reset currentEvent naar E_NOPRESS.
void doLED_OFF(); // Voert het algoritme uit van LED_OFF
void doLED_ON(); // Voert het algoritme uit van LED_ON
void doLED_BLINK(); // Voert het algoritme uit van LED_BLINK
void SW0InterruptHandler(); // Aangeropen bij indrukken knop
```

314. Voeg onderaan de statemachine (nadat je `currentState` updatet) een delay statement toe waar je 100 ms wacht (vergeet niet de `avr/delay.h` te includen).
315. Implementeer `setEvent`.
316. Implementeer `resetEvent`.
317. Plaats deze aanroep van `resetEvent` op de juiste plaatsen binnen je statemachine. Tip: Deze functie wordt elke keer aangeroepen, nadat de `nextState` geüpdatet is.
318. Implementeer `doLED_OFF()`, `doLED_ON()` Tip: roep binnen deze functies de functies aan die MCC voor je gemaakt heeft.
319. Implementeer `doLED_BLINK()`. Hiervoor mag je een toggle gebruiken.
320. Plaats de aanroep van deze functies op de juiste plaatsen binnen je statemachine. Tip: deze mogen als eerste aangeroepen worden wanneer je binnen een `currentState` case komt.
321. Implementeer `SW0InterruptHandler`. Deze functie roept op zijn beurt `setEvent` aan met attribute `E_PRESS`.
322. Plaats boven de While lus van je main de juiste functie aanroep waarmee je de defaultinterrupthandler van SW0 vervangt door bovenstaande functie. Dit hebben we in week 3 ook gedaan.

Wanneer je al deze functies geïmplementeerd hebt, en de aanroep daarvan correct gedaan hebt, is je statemachine af. Deze structuur kun je in de toekomst als template gaan gebruiken om verschillende andere statemachines mee te bouwen. Sla deze dus op een handige manier op (wanneer je zeker weet dat hij ook werkt!).

323. Run je code eens om te kijken of er wat gebeurt. Wanneer je statemachine nog niet naar behoren werkt, ga je debuggen.
324. Voeg een globale debugvariabele toe, `volatile unsigned int a = 0;`.

325. Plaats `a++`; onder elke `doLED_OFF()`, `doLED_ON()` en `doLED_BLINK()` aanroep. Plaats break-points op de `a++`; statements.
326. Plaats een `watchexpression` op `currentState`, `nextState` en `currentEvent`.
327. Run je code in debugmode en debug deze net zolang tot hij werkt.
328. Demonstreer de werking van je state machine aan de docent.

### 6.2.2 Statemachine met de SMU

Wanneer je dat nog niet gedaan hebt, maak je een kopie van je werkende statemachine van de vorige opdrachten.

We gaan nu een complexere state-machine maken met meerdere overgangen per state. Hiervoor gebruiken we de extra knopjes en LEDs op het SMU bordje.

329. Teken op basis van de volgende tabel een statemachine diagram

Current State	Activity	Event	Next State
S0	LED0 uit	PRESS_SW0	S0
	LED1 uit	PRESS_SW1	S1
S1	LED0 aan	PRESS_SW0	S0
	LED1 uit	PRESS_SW1	S2
S2	LED0 aan	PRESS_SW0	S2
	LED1 aan	PRESS_SW1	S3
S3	LED0 aan	PRESS_SW0	S4
	LED1 knipper	PRESS_SW1	S0
S4	Disco	PRESS_SW0	S1
		PRESS_SW1	S0

Table 1: Een Statemachine tabel met 5 states (S0 t/m S5) en 3 events (NO\_EVENT, PRESS\_SW0 en PRESS\_S1).

330. Maak op basis van een kopie van je eerdere project een nieuwe statemachine.
331. Voeg de extra knopjes en LEDS toe in MCC en regenerate je project.
332. Voeg de extra states en events toe aan je state en events enums.
333. Voeg de extra do functies toe die in elke nieuwe state aangeroepen moeten worden.
334. Voeg de extra interrupthandler functie toe voor SW1 en zorg ervoor dat deze meegegeven wordt met de juiste `setinterrupthandler` functie.
335. Zorg dat alle state cases en bijbehorende event cases afgehandeld worden. Tip: in sommige states heb je maar 1 event waarvoor je gevoelig bent. In plaats van dat je alle event cases daar programmeert, kun je makkelijker een default case toevoegen die in een klap alle niet relevante events afdekt.
336. Demonstreer de werking van je code aan de docent.



Wat even goed is om te weten, is dat de statemachine ook werkt wanneer we zouden gaan slapen (`sleep_mode()` functie onderaan de state machine). Dit komt doordat we wakker gemaakt worden op het moment dat we er een interrupt plaatsvindt. Op die manier weet je dus altijd of er een event trigger geweest is. Welke dat precies is wanneer je meerdere triggers krijgt, wordt natuurlijk in een interrupthandler functie bepaald. Het enige wat overigens niet werkt als we slapen is onze knipper. Dat is ook niet zo gek want we gebruiken een niet zo chique manier om de LED te laten knipperen. Als je dit beter wil oplossen moet je gebruik maken van entry en exit actions i.c.m. een timer of PWM om de LED hardwarematig te laten knipperen, in plaats van activiteiten waarin we de LED softwarematig laten knipperen. Dit gaan we in de volgende opdrachten proberen te verbeteren.

### 6.2.3 Statemachine met entry/exit actions

In voorgaande voorbeelden zagen we dat er in een state continu de waarde van een LED wordt overschreven. Dit is natuurlijk niet heel efficiënt. Het zou mooier zijn als we, wanneer we naar een state toe gaan (entry) maar één keer de LED aan zetten, en wanneer we uit die state gaan (exit) de LED weer netjes uitzetten. Dit is dus anders dan een activity waarbij we continu een actie uitvoeren. Wanneer we de CPU laten slapen, kunnen we softwarematig gezien helemaal geen activiteiten continu laten uitvoeren, onze CPU doet immers niets, dat forceert ons om na te denken wat echt nodig is. Het knipperen zou potentieel softwarematig gedaan kunnen worden als activiteit, maar dit is natuurlijk niet heel slim. Ook dat kun je hardwarematig doen.

337. Maak een kopie van het project bij de vorige opdracht.

338. Voeg de volgende functies toe aan je code:

Hiervoor hebben we een nieuwe statetabel gemaakt met entry en exit acties. Kijk deze eens door.

Current State	Activity	Entry	Exit	Event	Next State
S0				PRESS_SW0	S0
				PRESS_SW1	S1
S1		LED0 aan	LED0 uit	PRESS_SW0	S0
				PRESS_SW1	S2
S2		LED0 aan	LED0 uit	PRESS_SW0	S2
		LED1 aan	LED1 uit	PRESS_SW1	S3
S3	LED1 knipper		LED0 uit	PRESS_SW0	S4
			LED1 uit	PRESS_SW1	S0
S4	Disco		LED0 uit	PRESS_SW0	S1
			LED1 uit	PRESS_SW1	S0

Table 2: De statemachine heeft nu entry en exit actions.

Op basis van bovenstaande tabel moet je voor elke entry, exit en activity (do) een nieuwe functie maken. Zie de lijst hieronder.

Listing 7: Nieuwe functies o.b.v. entry en exit acties.

```
void entry_S1 (); // LED0 aan
void entry_S2 (); // LED0 aan, LED1 aan
void exit_S1 (); // LED0 uit
void exit_S2 (); // LED0 uit, LED1 uit
```

```

void exit_S3 (); // LED0 uit , LED1 uit
void exit_S4 (); // LED0 uit , LED1 uit
void do_S3 ();    // Knipper LED1
void do_S4 ();    // Disco op alle LEDS

```

339. Implementeer deze functies. Tip: Roep binnen elke functie de juiste MCC functies van LED0 en LED1 aan.
340. Voeg een globale enum toe genaamd flow. Deze enum kan de waarde F\_ENTRY, F\_ACTIVITY of F\_EXIT hebben.
341. Maak een instantie aan van deze enum en noem deze currentFlow en geef deze standaard-waarde mee F\_ENTRY.

Omdat de code nu erg lang wordt, geven we je het template voor een goed geïmplementeerd voorbeeld case weg. In dit voorbeeld in listing 8 staat een nieuwe switch case genest in de state switch case. Deze staat onderaan, en wordt nadat de event switch case uitgevoerd is, uitgevoerd.

Listing 8: S\_OFF van oorspronkelijke statemachine met de stateflow.

```

switch(currentState){

    // Geen activity meer hier , staat nu onder case F_ACTIVITY.

    case S_OFF:
        switch(currentEvent){
            case E_PRESS:
                nextState = S_ON;
                resetEvent();
                break;
            case E_NOPRESS:
                break;
        }

        switch(currentFlow){
            case F_ENTRY:
                ENTRY_S_OFF();
                // Entry Action , 1x uitgevoerd
                currentFlow = F_ACTIVITY;
                // Geen Break , we voeren activity direct uit.

            case F_ACTIVITY:
                DO_S_OFF();
                // State activity , elke iteratie uitgevoerd
                if(nextState == currentState){
                    break;
                } else {
                    currentFlow = F_EXIT;
                    // geen break , ga gelijk door naar F_EXIT.
                }
        }
    }

```

```

        case F_EXIT :
            EXIT_S_OFF ();
            // Exit activity , 1x uitgevoerd
            currentFlow = F_ENTRY; // Reset state flow.
            break;
    }

    break;

```

342. Bestudeer bovenstaande code.
343. Benoem de functies die uitgevoerd worden bij entry, activity en exit.
344. Hoe weten we dat we maar 1x een entry en 1x een exit actie uitvoeren?
345. Wordt activity elke iteratie uitgevoerd?
346. Is het uitvoeren van een entry of exit actie afhankelijk van hoe, dus vanuit welke voorgaande state, we deze state binnenkomen?
347. Kopieer bovenstaande code voor elke state case, en pas deze zo aan dat deze per state de juiste functies aanroept. Wanneer er geen entry of exit acties zijn, hoeft je op die plaats geen functie aan te roepen. Vergeet ook niet de juiste events/state overgangen die je bij de vorige opdracht gemaakt had mee te nemen.

#### 6.2.4 Eigen Statemachine

Wanneer het je gelukt is om deze statemachine te programmeren kun je aan de slag gaan met de statemachine die je voor project DxC hebt gemaakt. Misschien kun je deze 1-op-1 in code overnemen, of misschien moet je nog wat aanpassingen doen.

Tips voor je eigen statemachine: Denk nog eens goed na over je states en events. States gaan dus over gedrag, events over triggers. Wanneer je een goede trigger wil gebruik je vaak interrupts. Een analoge sensor kan ook een trigger veroorzaken, maar vaak willen we dan een duidelijk omslagpunt. Hiervoor gebruiken we een AC (analog compare) met Schmitt-trigger. De compare bekijkt of de waarde over een bepaalde grens heen gaat. De Schmitt-trigger zorgt ervoor dat we niet gaan pendelen. Dat gebeurt wanneer we rond de grenswaarde zitten en het voor kan komen dat we meerdere keren erover heen gaan. De Schmitt-trigger lost dit probleem op door een tweede threshold toe te voegen. Sommige sensoren hebben intern al een Schmitt-trigger, soms moet je deze zelf maken of bouwen.

348. Ga op zoek naar de werking van de Schmitt-trigger, en bedenk hoe die in software en/of hardware gemaakt kan worden.
349. Programmeer je eigen state-machine. Je kunt de events eventueel eerst met knopjes laten werken, en later met een sensor.

### 6.2.5 Bonus: LED Dimmer

We kunnen de dutycycle van de PWM tijdens runtime aanpassen. Dit zorgt ervoor dat we bijvoorbeeld de LED feller en minder fel kunnen maken afhankelijk van een input van de gebruiker. Bijvoorbeeld wanneer de gebruiker op een knopje drukt kan de LED feller worden. Dit principe kun je ook toepassen op een PWM aansturing van een motor. Zo kunnen we de motor harder laten draaien wanneer we op een knopje drukken. Nog mooier zou zijn, wanneer we de snelheid van de motor kunnen laten afhangen van een analoge waarde, bijvoorbeeld de waarde van de lijnvolgsensor. Hiervoor hebben we de ADC nodig, die je volgende week krijgt. In deze opdracht bouwen we een state machine met de volgende eigenschappen:

Activity (wat gebeurt er)	Event (ga naar de volgende rij)
LED staat uit	ISR op PB2
LED brandt op 20%	ISR op PB2
LED brandt op 40%	ISR op PB2
LED brandt op 60%	ISR op PB2
LED brandt op 80%	ISR op PB2
LED brandt op 100%	ISR op PB2 (terug naar begin)

Table 3: De LED dimmer

Het idee is dat de LED met een bepaalde intensiteit brandt. Wanneer we op de knop PB2 drukken, gaan we naar een hogere intensiteit. Daar blijven we totdat er opnieuw op de knop gedrukt wordt. Eigenlijk kun je hier het beste een statemachine voor gebruiken. Deze gaan we pas in week 5 behandelen.

We willen dat onze uC dit zo efficiënt mogelijk doet. Ook willen we dat onze code gestructureerd is en gebruik maakt van interrupts en functies. Zorg er daarom voor dat je aan de volgende criteria voldoet:

- De uC slaapt en is alleen wakker wanneer er een interrupt optreedt.
- De event wordt vastgesteld door middel van de ISR
- De State Machine is geprogrammeerd in een eigen functie
- De State Machine zet als entry actie het HCMP0 register op de juiste waarde
- De State Machine wordt doorlopen en gaat naar de volgende state toe wanneer het knopje op PB2 ingedrukt wordt (falling edge)

Laat je werkende dimmer zien aan de docent.

# A Digitale sensors & Communicationprotocols

## A.1 Voorbereiding

Je heb er vast al verschillende keren meegewerkt, ze zitten in heel veel moderne apparatuur, sensoren. Maar wat zijn dat nou eigenlijk sensoren? Wat doen ze? Welke verschillende soorten heb je allemaal? Als we het over sensoren hebben bedoelen we een component wat een bepaalde fysieke grootheid kan aflezen en omzetten in een bruikbare/uitleesbare andere grootheid. Voor de opkomst van moderne elektronica had je sensoren die bijvoorbeeld een meting omzetten in een mechanische beweging. Denk bijvoorbeeld aan een barometer, die de druk kan meten en met behulp van een wijzertje aangeeft wat de druk is.

Als we het tegenwoordig over sensoren hebben, dan hebben we het bijna altijd over elektronische sensoren. Dat betekent dat die gemeten grootheid wordt omgezet in een elektrisch signaal. De reden hiervoor is dat we deze sensoren bijna altijd automatisch willen uitlezen door bijvoorbeeld een (micro) controller of PLC.

Dit elektrische signaal kan verschillende vormen aannemen. In het practicum van deze week gaan we verschillende digitale en volgende week analoge sensoren behandelen. Deze sensoren kun je allemaal gebruiken om de functionaliteit van je robot te verbeteren.

1. Gegeven zijn de volgende grootheden. Zoek voor elke grootheid 3 verschillende sensoren op het internet op.

TODO sensoren wellicht aanpassen.

Type grootheid	Eenheid	Sensor 1	Sensor 2	Sensor 3
Spanning				
Temperatuur				
Magnetisme				
Acceleratie				
Afstand				
Licht				
Radioactiviteit				
Geluid				

Wellicht zul je nog veel meer verschillende sensoren vinden dan 3. Er zijn tal van uitvoeringen, afmetingen en fabrikanten. Op basis waarvan kies je dan een geschikte sensor? Dit hangt naast technische eigenschappen zoals nauwkeurigheid, stabiliteit, snelheid, grootte, gewicht, lineairiteit en meetbereik ook af van niet technische eigenschappen zoals kosten, eenvoudigheid in gebruik, leverbaarheid. We gaan een deel van deze eigenschappen behandelen in dit practicum, maar je zult al begrijpen dat het snel zeer complex/ uitgebreid kan worden. Wanneer je een sensor kiest is het van belang dat je deze eigenschappen meeneemt in je afweging. Wanneer je hier meer over wilt lezen, kijk dan in Snack 1 – Componentenkeuze.

### A.1.1 Sensorkeuze

Vul ter voorbereiding alvast volgende tabellen in:

## Druksensor

Grootheid	Eenheid	Interlink 18mm	Taiwan alpha			
Range						
Voedingsspanning						
Nominale stroom						
Communicatie						
Prijs						
Sample rate (meetfrequentie)						
Kwaliteit documentatie						
Temperatuursafhankelijkheid						
Lichtafhankelijkheid						
Field of View						

Grootheid	Eenheid	3144 Hall	AS5600	OH49E		
Range						
Voedingsspanning						
Nominale stroom						
Communicatie						
Prijs						
Sample rate (meetfrequentie)						
Kwaliteit documentatie						
Temperatuursafhankelijkheid						
Lichtafhankelijkheid						
Field of View						

Grootheid	Eenheid	CNY70	TCRT5000	Module met CNY/ TCRT	Camera	
Range						
Voedingsspanning						
Nominale stroom						
Communicatie						
Prijs						
Sample rate (meetfrequentie)						
Kwaliteit documentatie						
Temperatuursafhankelijkheid						
Lichtafhankelijkheid						
Field of View						

### A.1.2 Ultrasoonsensor

De ultrasoonsensor is een afstandssensor die met behulp van een geluidspuls de afstand kan bepalen.

2. Leg in je eigen woorden uit hoe de ultrasoonsensor afstand kan bepalen op basis van geluid. Onderbouw je uitleg met een formule en een schets/tekening.

De module die we in het practicum gaan gebruiken is de US-100. Deze module meet niet continu, maar start pas een meting nadat we een triggersignaal gegeven hebben. Dit in tegenstelling tot veel analoge sensoren.



Figure 42: US-100

3. Leg uit waarom het gunstig kan zijn om niet continu te meten.
4. Wat is het voordeel van een meting die start o.b.v. een triggersignaal?

In onderstaande figuur zien we een trigger waar na een bepaalde periode een response (echo) ontstaat. De breedte van deze echo puls (tijd) heeft een lineair verband met de afstand die gemeten is. Hoe langer de puls is, hoe verder het gemeten object afstaat van de ultrasoonsensor.

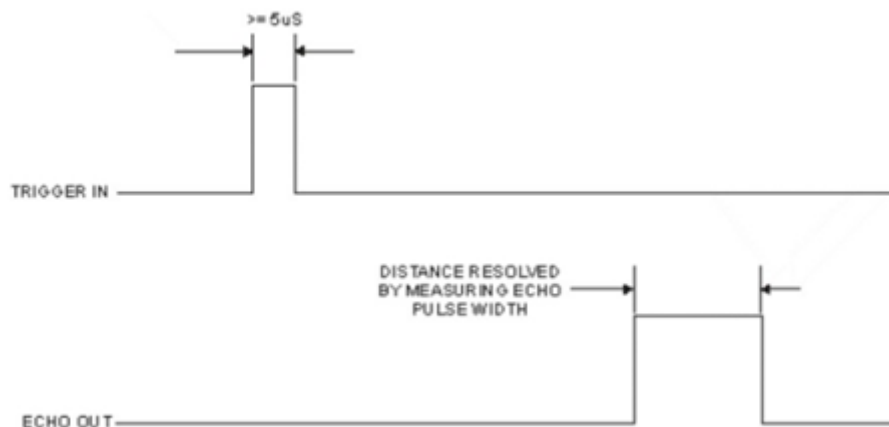


Figure 43: Trigger- en Echosignaal van de US-100

De werking van de Echo kun je enigszins vergelijken met de stopwatch. Op het moment dat de puls verstuurd wordt, zal de echo omhoog gaan. Wanneer de geluidspuls weer opgevangen wordt, zal de echo weer omlaag gaan.

Wanneer we de ultrasoon sensor willen uitlezen met de uC moeten we een capture gebruiken met timer B. In het practicum van uC-programmeren besteden we aandacht aan hoe je deze timer kunt gebruiken.

We leggen in deze sectie kort uit hoe we verschillende communicatieprotocollen werken.

Afhankelijk van de gekozen component kunnen we data op verschillende manieren versturen en ontvangen. Zoals de titel van dit hoofdstuk aangeeft, focussen we ons deze week op digitale protocollen. Digitale protocollen zijn bedacht om een standaard manier te zijn van communicatie. Veel gebruikte protocollen zijn UART, I2C en SPI.

Grootheid	Eenheid	HC-SR04	VL5310x	GP2Y0A 21YK0F	US-100	M5STACK
Range						
Voedingsspanning						
Nominale stroom						
Communicatie						
Prijs						
Sample rate (meetfrequentie)						
Kwaliteit documentatie						
Temperatuursafhankelijkheid						
Lichtafhankelijkheid						
Field of View						

Grootheid	Eenheid	AM2320	AHT21B	BME280	BME280	
Range						
Voedingsspanning						
Nominale stroom						
Communicatie						
Prijs						
Sample rate (meetfrequentie)						
Kwaliteit documentatie						
Temperatuursafhankelijkheid						
Lichtafhankelijkheid						
Field of View						

Grootheid	Eenheid	ADXL345	ADXL335	MPU6050		
Range						
Voedingsspanning						
Nominale stroom						
Communicatie						
Prijs						
Sample rate (meetfrequentie)						
Kwaliteit documentatie						
Temperatuursafhankelijkheid						
Lichtafhankelijkheid						
Field of View						



### A.1.3 UART

#### Leestip: Analog-Devices – UART

De UART of USART verbinding staat voor universal (a)synchronous (serial) receiver transmitter. Dit protocol wordt gebruikt om eenvoudig een verbinding tot stand te brengen. We hebben dit laten zien in het vak uC-programmeren waarbij we verbinding maakten met de data-visualiser.

UART heeft 2 verbindingen een transmitter TX en een receiver RX. Normaalgesproken koppelen we in UART de TX van device A met de RX van device B en vice-versa.

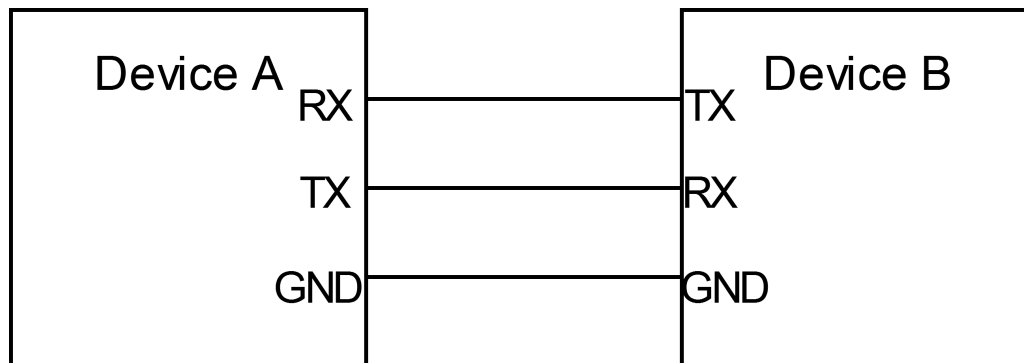


Figure 44: UART connectie tussen 2 devices

Een voordeel van UART is dat de verbinding zeer eenvoudig en voorzichtzelfsprekend is en dat we geen kloksignaal nodig hebben. Het nadeel is dat we maar 2 devices aan elkaar kunnen koppelen met een verbinding. Ook moeten we van te voren een bepaalde baudrate vaststellen. Omdat het signaal asynchroon binnenkomt betekent het dat je bitjes kunt missen wanneer je te traag bent.

UART data wordt overgezet in pakketjes

Start Bit 1	Data Frame 5-9 bits	Parity Bit 0 of 1	Stop Bits 1 of 2
----------------	------------------------	----------------------	---------------------

Figure 45: UART dataframe

In figuur 45 zien we waaruit een datapakketje bestaat. Een UART bericht begint met een startbit, gevolgd door een aantal databits (het bericht). Het eindigt met een parity bit wat gebruikt wordt ter controle van de bitreeks en sluit af met 1 of 2 stopbits om aan te geven dat het bericht voltooid is.

Er zijn verschillende manieren om een UART bericht te construeren en uitlezen. Dit kan hardwarematig gedaan worden, of softwarematig. Wat we bij uC-programmeren hebben gezien is een softwarematige aansturing. Wanneer we via software een UART bericht uitlezen doen we dat m.b.v. libraries/functies. Zie voor verdere uitleg uC-programmeren. Wanneer we met hardware een UART uitlezen/versturen, moeten we ervoor zorgen dat wanneer we uitlezen de component de signalen samplet. Dat wil zeggen dat hij uitleest of de bits hoog of laag zijn. De hardwarecomponent reconstrueert daarmee het bericht. Wanneer we een bericht versturen is dat precies andersom.

Wanneer we willen gaan communiceren moeten we weten wat voor berichten we kunnen versturen en wat voor berichten we kunnen ontvangen. Sensor ... werkt met een UART protocol

5. Ga op het internet op zoek naar hoe UART werkt. (zie leestip).
6. Zoek uit wat alle commando's en berichten zijn die deze device kan ontvangen.
7. Pak de uC-programmeren handleiding erbij en bekijk hoe we UART hebben verstuurd. Probeer dit in je eigen woorden uit te leggen

## A.2 Practicum

Tijdens dit practicum kun je verschillende sensoren gaan doormeten. Het is de bedoeling dat je minimaal 2 kiest. Dit doe je afhankelijk van het type sensor wat je nodig hebt voor je DxC-robot.

### A.2.1 Ultrasoonsensor

Benodigdheden

- US-100
- Labvoeding
- Oscilloscoop
- Functiegenerator
- Breadbord
- Aansluitkabels

In dit practicum gaan we de ultrasoonsensor uitlezen met behulp van de oscilloscoop. Let op, de US-100 werkt op 5V en kan kapot gaan bij een te hoge stroom. Zorg ervoor dat je de stroom en spanning gepast begrenst.

1. Prik de US-100 in je breadbord.

De US-100 dient gevoed te worden met een spanning van 5V, stroombegrenzing 0,05A. We willen nu de US-100 laten triggeren op een korte puls die de functiegenerator genereert. Deze puls willen we ook zichtbaar maken op de oscilloscoop. Daarna willen we de echo van de US-100 zichtbaar maken op de oscilloscoop.

2. Teken op basis van bovenstaande beschrijving een aansluitschema. Laat dit schema eerst zien aan de docent, voordat je verder gaat.
3. Stel de functiegenerator juist in zodat er een puls van minimaal 5  $\mu$ s periodiek gegeneerd wordt. Tip: gebruik een blokgolf en stel de duty cycle laag in. Op die manier houd je genoeg tijd over om de echo af te laten lopen.
4. Laat in eerste instantie de voeding en functiegenerator uitstaan. Sluit nu alles aan volgens het schema.
5. Stel de oscilloscoop zo in, dat je zowel de triggerpuls als de echo zichtbaar kunt krijgen.
6. Zet de voeding en functiegenerator aan.

## B Losse chip en gebruik programmer

In deze opdracht gaan we een losstaande AVR128DB28 chip programmeren. Hiervoor gebruiken we een aparte programmerchip en uploader.

1. Haal een losse AVR128DB28 chip (throughhole) en een programmer (FTDI basic breakout board) op bij de labtechnici
2. Haal een  $470\Omega$  weerstand en een 1N4148 diode op
3. Sluit de programmer chip aan op de volgende manier
  - (a) De GND moet met de GND van de uC verbonden zijn.
  - (b) De uC mag niet gevoed worden vanuit de programmer (dus 3.3V niet doorverbinden)
  - (c) De UDPI pin is een speciale pin op je uC. Zoek deze op in de datasheet
  - (d) Sluit de programmerchip aan met een usb-kabel
4. Zoek in de datasheet ook op op welke pin je de uC moet voeden en sluit een 3,3V voeding aan.
5. Download de SERIAL UPDI Programmer van Brightspace
6. Maak een nieuw ATMEL Start project aan voor de AVR128DB28
7. Programmeer een programma. Je kunt als test bijvoorbeeld een LED aansluiten en aansturen op de losse chip
8. Genereer een hex bestand en upload dat via de programmer
9. Controleer of de code werkt.

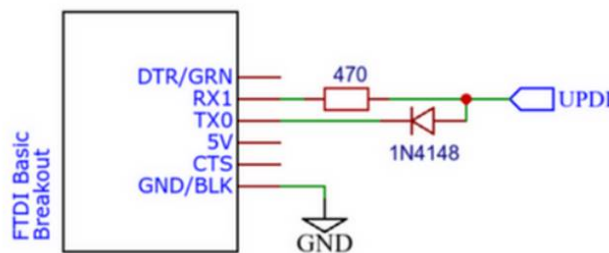


Figure 46: FTDI Interface met de AVR128DB28