hogeschool vives

Microcontrollers

docent **L. Espeel**

studiegebied **Technology**

bachelor in **Elektronica-ICT**

campus **Brugge**

academiejaar **2024-2025**

Inhoudsopgave

Oorspronkelijke auteur van deze cursus is R. Buysschaert met aanpassingen en aanvullingen door L. Espeel.

1 Microcontrollers

Om dit vak te kunnen volgen is *strikt gezien* geen speciale voorkennis vereist. Eerdere ervaring met microcontrollersystemen wordt uiteraard aanzien als een pluspunt.

Toch wordt het sterk aanbevolen om al over een aantal basisvaardigheden en kennis te beschikken om de leerstof vlot te kunnen volgen:

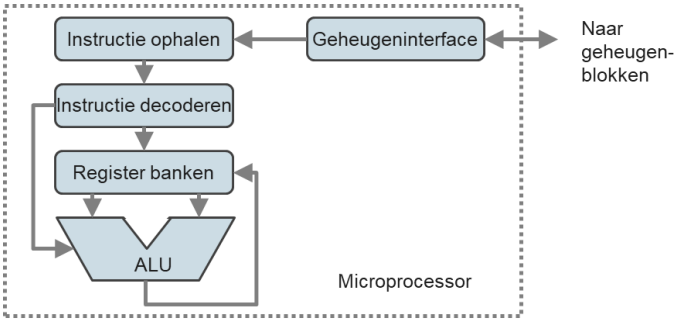
- **Algemene programmeerkennis:** het hebben van een basisinzicht in programmeren is belangrijk. Ervaring met de C-taal die we zullen gebruiken is geen vereiste, maar wordt wel als een pluspunt beschouwd.
  De nodige uitleg over deze taal zal sowieso gegeven worden op het moment dat dit vereist is.
- **Kennis van digitale basislogica:** begrip van logische bouwstenen zoals AND-, OR-, en NOT-poorten. Daarnaast kan kennis van combinatorische en sequentiële logica, zoals flipflops en tellers, nuttig zijn.
- **Binair rekenen:** het kunnen werken met binaire getallen en het begrijpen van de basisbewerkingen zoals binair optellen, aftrekken, verschuiven ("shiften"), … is essentieel.
  Ook vlot kunnen omrekenen van het ene talstelstel naar het andere zoals decimaal, binair en hexadecimaal talstel.
  Dit zal regelmatig aan bod komen tijdens het semester.

Je zal merken dat het werken met microcontrollers de max zijn, ze laten je vaak toe een wild idee om te zetten naar een werkend geheel!

1.1 Definitie

Wat is nu het verschil tussen een microcontroller of microprocessor? Velen kunnen deze vraag niet onmiddellijk beantwoorden, hoewel dit niet zo moeilijk is.

Een **microprocessor** (soms ook **CPU** genaamd, staat voor Central Processing Unit) bevat quasi nooit een interne geheugenmodule met uitzondering van cachegeheugen en registers. Het heeft daarom altijd externe componenten nodig voor opslag en I/O (input/output).



Figuur 1: opbouw microprocessor

De microprocessor bevat een rekeneenheid, de **ALU** (Arithmetic and Logic Unit) die continu rekenkundige elementaire bewerkingen uitvoert zoals optellen, aftrekken, binaire AND, binaire OR enz. De data dient van de externe geheugenmodule te komen en het berekende resultaat moet weer hierin worden geplaatst.
(Opmerking: de geheugenmodule bevat vaak een zone om gewone **data** (variabelen) op te slaan en een aparte zone voor **programmacode**; meer info volgt verder.)

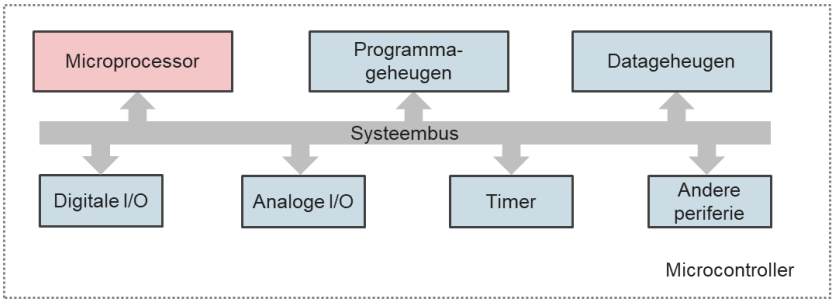
Doordat er een grote kloof is tussen de verwerkingssnelheid van de microprocessor, die vaak vele malen sneller is dan de geheugenmodule, zijn in de loop der tijd verschillende **optimalisaties** doorgevoerd. Hierdoor leest en schrijft de ALU niet rechtstreeks naar en van de geheugenmodule.

Een **eerste optimalisatie** is dat de data uit de geheugenmodule in stappen gekopieerd wordt naar de **registers** in registerbanken, die zich dicht bij de microprocessor bevinden. Deze registers fungeren als tijdelijk geheugen en zijn veel sneller dan de geheugenmodule, maar ze zijn echter beperkt in aantal en grootte. De ALU leest data uit de registerbanken, voert de benodigde bewerkingen uit en schrijft het resultaat terug naar de registerbanken. Daarna wordt het resultaat overgekopieerd naar de geheugenmodule.

Tussen de registers en de geheugenmodule bevindt zich bovendien **cachegeheugen**, een **verdere optimalisatie**. Data die vaak opgevraagd wordt uit de geheugenmodule, wordt opgeslagen in dit tussenliggende cachegeheugen. Dit geheugen is sneller dan de geheugenmodule, maar minder snel dan de registerbanken. Het cachegeheugen is groter dan de registerbanken, maar veel kleiner dan de geheugenmodule.

Enkele voorbeelden van moderne microprocessoren zijn de AMD Ryzen-serie en de Intel Core-serie, die beide gebaseerd zijn op de x86-64-architectuur voor consumentencomputers, de Intel Xeon-serie voor servers, en de AMD EPYC-serie voor datacenters en bedrijfstoepassingen.

Een **microcontroller** (soms ook **MCU** genaamd, staat voor MicroController Unit) is een compacte geïntegreerde schakeling die een microprocessor, intern geheugen en extra voorzieningen (vaak **periferie** of modules genaamd). De verschillende delen zijn verbonden d.m.v. **bussen**, zie verder. Onder periferie hoort: digitale input/output, analoge input/output, timers, seriële interfaces (UART, SPI, I^2C, …).
Een aantal microcontrollers bevatten ook geavanceerdere periferie zoals USB controller, CAN bus controller, Ethernet, …



Figuur 2: opbouw microcontroller

Er zijn verschillende populaire microcontrollerreeksen, denk maar aan de Microchip PIC microcontrollers, Atmel AVR-reeks, ARM microcontrollers, …

Opmerkingen:

- Tegenwoordig komt ook de term **System on Chip (SoC)** vaak voor. Hoewel een SoC, net als een MCU, alle noodzakelijke componenten op één chip bevat, zijn er toch belangrijke verschillen.

  Een SoC omvat veel complexere en krachtigere componenten, zoals een Graphics Processing Unit (GPU), netwerkinterfaces (zoals WiFi en Bluetooth) en bevat vaak meerdere cores in de

microprocessor voor multitasking. Maar het is ook minder low power dan een microcontroller. Dit maakt SoC's geschikt voor toepassingen in smartphones, tablets en ook Raspberry Pi's.

- Daarnaast komt de term **embedded systeem** regelmatig voor. Het is belangrijk te begrijpen dat dit een bredere term is en geen synoniem voor een microcontroller.

  Een embedded systeem verwijst naar een gespecialiseerd computersysteem dat is ontworpen om een specifieke taak of functie uit te voeren binnen een groter systeem.
  Een embedded systeem kan één of meerdere microcontrollers bevatten, maar dit is geen vereiste. Het kan ook andere componenten omvatten zoals een SoC of een FPGA (Field Programmable Gate Array). In sommige gevallen bevat een embedded systeem dus helemaal zelfs geen microcontroller.

1.2 Toepassingen

Microcontrollers zijn niet meer weg te denken uit heel veel hedendaagse toestellen. Enkele voorbeelden staan hieronder.

Thuis

- Centrale verwarming en bijhorende thermostaat
- Inbraakbeveiliging
- Microgolfoven
- Speelgoed
- Digitale uurwerk
- Vaatwasser
- Audioversterker
- Domoticasystemen
- Batterijladers
- GSM
- …

Kantoor

- Printer
- Scanner
- e-ID-lezer
- Toetsenbord
- Telefoontoestellen en -centrale
- …

Auto

- Klimaatcontrole
- Parkeerassistentie
- Remassistentie
- GPS
- Alle systemen aangesloten op de CAN-bus
- …

Kortom: quasi alle toestellen die wat 'intelligentie' bezitten, hebben meestal één of andere vorm van een microcontroller in zich.

1.3 Fabrikanten

Er zijn veel fabrikanten van microcontrollers, hieronder volgt een klein overzicht.

STMicroelectronics

STMicroelectronics heeft twee belangrijke reeksen microcontrollers: de STM8 (8-bit) en de STM32. De STM32-serie is gebaseerd op de ARM Cortex processor cores en heeft veel aandacht gekregen vanwege de combinatie van lage kosten en krachtige prestaties. Dit heeft de 32-bit ARM microcontrollers steeds populairder gemaakt, vooral in de industrie, waar ze vaak de standaard zijn.

Microchip

Microchip is een grote speler en is zeer populair bij de hobbyist en de industrie. Dit heeft onder andere te maken met de kostprijs van de controllers, maar ook dankzij de gratis beschikbare ontwikkelomgeving en de ruime ondersteuning via de websites van de producenten.
Microchip biedt 2 grote reeksen aan: de PIC18 (8-bit) en PIC32 microcontrollerreeks. De PIC32 reeks is gebaseerd op de MIPS M-Class of M4K core.

Een andere fabrikant is **Atmel**, dat opgekocht werd door Microchip voor iets meer dan 3 miljard euro. Atmel leverde ook microcontrollers voor op Arduino-bordjes te plaatsen.

NXP

NXP is de zelfstandig geworden semiconductorafdeling van Philips en heeft nu een hele reeks microcontrollers op basis van verschillende ARM cores.

NXP heeft indertijd de andere fabrikant **Freescale** overgenomen. Freescale had microcontroller-reeksen zoals ColdFire, Power Architecture, en Kinetis (ARM-gebaseerd).

Texas Instruments

Texas Instruments biedt verschillende reeksen microcontrollers aan. Naast twee reeksen die gebouwd zijn rond ARM-processoren, heeft TI ook de MSP430 (16-bit) en C2000 (32-bit) reeksen. De MSP430 is bekend om zijn ultra-low-power eigenschappen en wordt veel gebruikt in toepassingen die energie-efficiëntie vereisen.

Renesas

Renesas is ontstaan uit een consortium van NEC, Hitachi en Mitsubishi. Renesas biedt een breed scala aan microcontrollers, waaronder ultra-low-power 8-bit en 16-bit microcontrollers (RL78, 78K, en R8C) en een 32-bit RISC (V850, SuperH) en CISC (RX) reeks.

Infineon

Infineon biedt een reeks van 8-bit (XC800), 16-bit (XC2200) en 32-bit (TriCore) microcontrollers aan. De TriCore microcontrollers zijn vooral gericht op high-performance toepassingen.

Silicon labs

Silicon Labs heeft een microcontroller-reeks gebaseerd op een gepijplijnde versie van de 8051 (8-bit) microcontroller, en biedt ook andere oplossingen voor IoT en draadloze communicatie.

Bovenstaande lijst is absoluut niet volledig en gevoelig voor verandering. Fabrikanten maken nieuwe reeksen, fusioneren, verlaten een gamma, … De meest recente info is terug te vinden op de respectievelijke websites van de fabrikanten.

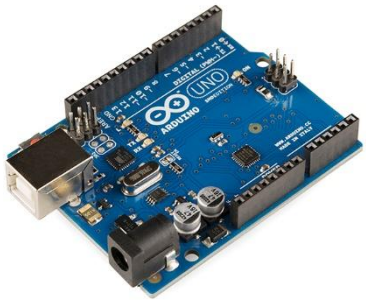
1.4 Keuze van microcontroller

Op de markt zijn er verschillende toegankelijke instapmodellen van microcontrollers met ontwikkelplatformen, zoals het bekende **Arduino**-platform.

Arduino is een open-source platform dat wordt gebruikt om eenvoudige tot geavanceerde elektronische projecten te ontwikkelen. Het bestaat uit twee hoofdcomponenten: de hardware (een Arduino-bord) en de software (de Arduino IDE). Arduino is voornamelijk ontworpen om toegankelijk te zijn voor beginners en hobbyisten.

In feite is een Arduino-bord een standaard microcontrollerbord met daarop een Atmel-microcontroller en een Arduino-bootloader. Deze bootloader is een klein stukje firmware dat ervoor zorgt dat we de microcontroller kunnen programmeren via USB-interface. Hierdoor is er geen externe programmer nodig, wat het voor beginnende programmeurs en elektronicaliefhebbers eenvoudig maakt om snel hun systeem werkend te krijgen, zelfs zonder uitgebreide voorkennis.

Het meest gebruikte Arduino-bord, de Arduino Uno R3, bevat een 8-bit ATmega328P-microcontroller die werkt op een kloksnelheid van 16 MHz.



Figuur 3: Arduino Uno R3 (bron: wikipedia.org)

Voordelen van bovenstaand bord zijn:
- Prijs
- (stacking) headers voorzien zodat we snel kunnen prototypen
- Veel shields beschikbaar
- Programmeerbaar via USB-interface

Nadelen:
- Weinig tot geen direct beschikbare IO (knoppen en LED's). Vooraleer we kunnen starten met programmeren moeten we dus eerst hardware koppelen.
- De print heeft een vaste afmeting die misschien absoluut niet past in het project dat we voor ogen hebben.
- De standaard IDE van Arduino is heel eenvoudig, maar mist enkele handige tools die een professionele gebruiker er graag bij heeft (vooral qua debugging) …
- Moeilijk te gebruiken voor tijd kritische systemen (timers, vaste looptijden, …).

Arduino is leuk, gemakkelijk, we krijgen snel 'leven' in onze projecten, maar we kunnen het niet gebruiken als standaard om embedded systemen te leren programmeren.
Maar op een grondige manier microcontrollers leren programmeren is niet evident. Als we het echt goed willen doen, moeten we tijd investeren en ervaring opdoen. Alleen op die manier kunnen we de gelimiteerde resources van een embedded systeem ten volle benutten.

"Using an Arduino to learn programming is like using MacDonalds to learn cooking; you get your meal, very fast, but you don't get the skills to cook yourself. When you need a quick meal, the Mac can be OK (debatable, but just to make my point), but it's not a cooking class."
(oorspronkelijke bron: http://www.hackvandedam.nl/blog/?p=762)

Andere vaak gehoorde negatieve bevindingen van Arduino zijn:
- Slechte IDE
- Opsplitsen van code wordt niet gestimuleerd in de IDE
- Slecht gekozen benamingen: bijvoorbeeld *analogWrite()*. Het zet eigenlijk geen analoge waarde, maar stelt een PWM-signaal in. Daarenboven kunnen we niet zomaar de PWM-frequentie aanpassen.
- Geen *exact width* variabelen (zie verder)
- De documentatie laat teveel zaken onbesproken voor de professionele gebruiker. Dat kan handig zijn voor de beginner, maar eenmaal we meer willen, komen we een 'black box' tegen.
- Door het black box-principe is het heel moeilijk om firmware te schrijven met een minimale footprint.

Wees dus op je hoede voor het Arduino-beest, het heeft zeker zijn plaats binnen de elektronica-ICT-wereld, maar niet overal! In de industrie wordt Arduino heel weinig gebruikt.
Vandaar dat we het Arduino-bordje niet zullen behandelen in het vak *Microcontrollers* en de vervolgvakken *IoT Devices* en *IoT Systems*.

In dit vak *Microcontrollers* zullen we gebruikmaken van de **STM32F091RC-microcontroller** van **STMicroelectronics**. Dit is een veelzijdige en energie-efficiënte 32-bit ARM microcontroller die geschikt is voor zowel educatieve toepassingen als industriële toepassingen.
De STM32F091RC staat bekend om zijn brede ondersteuning, uitgebreide documentatie en veelgebruikte ontwikkeltools, gecombineerd met een lage kostprijs, wat het een uitstekende keuze maakt voor het leren werken met microcontrollers.

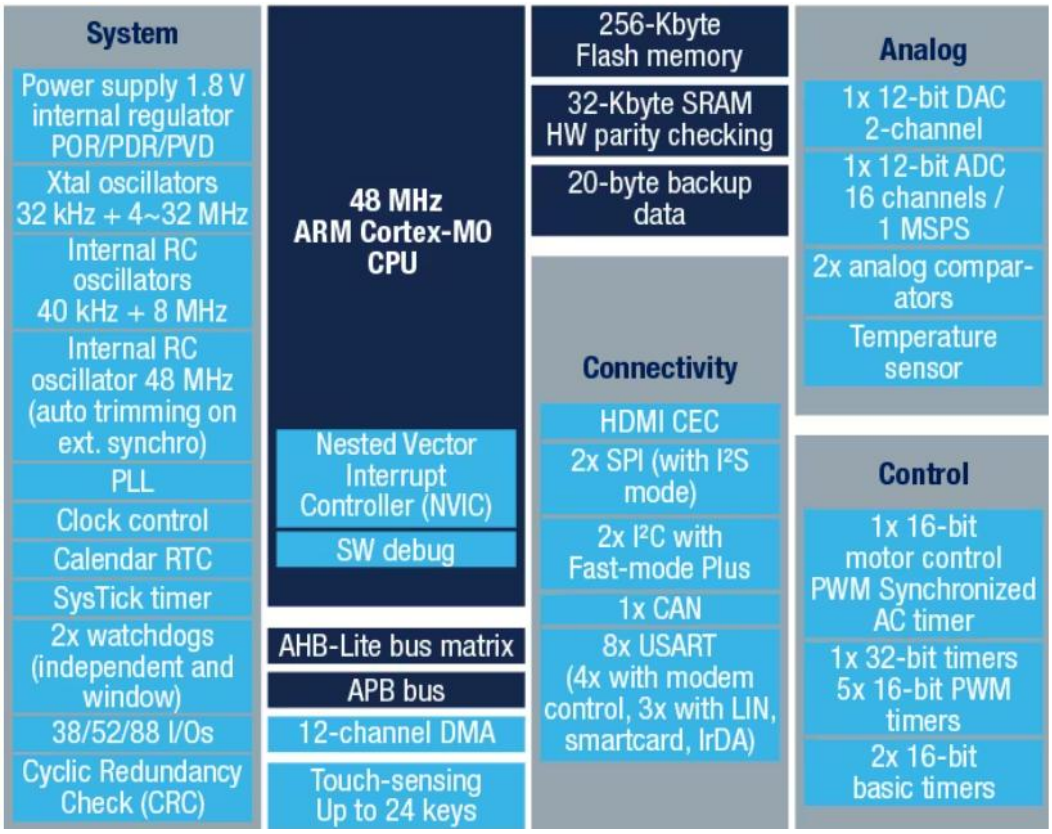
1.5 Onderdelen

Veelvoorkomende onderdelen bij een microcontroller zijn:
- **CPU:** bevat de ALU die instructies uitvoert (logische, aritmetische, lees/schrijf acties).
- **Oscillator:** maakt met behulp van een kristal een klok waarop de logica werkt. Hier wordt dus de kloksnelheid van heel het systeem bepaald.
- **Bussen:** adres-, data- en controlebussen die de verschillende blokken verbinden in de microcontroller.
- **ROM (Read Only Memory)/flash:** geheugen dat niet volatiel is. Dit geheugen wordt gebruikt om de programmacode in op te slaan. Het programma blijft ook opgeslagen als de microcontroller zonder stroom komt te staan.
  - Opmerking: er is een klein verschil tussen ROM- en flashgeheugen. Het programma in ROM wordt tijdens de fabricage of programmering vastgelegd en kan daarna niet of nauwelijks worden gewijzigd, terwijl we bij flash meerdere malen het programma kunnen overschrijven.
- **RAM (Random Access Memory):** geheugen dat volatiel is, maar sneller dan andere types geheugen (ROM of flash).
- **Timers:** hardware modules die tellen en signalen geven wanneer een bepaalde tijd verstreken is.
- **Interrupt controller:** bundelt verschillende interrupts die al dan niet doorgegeven worden aan de processor.

- **IO**, diverse vormen:
  - Parallelle IO: één byte wordt bijvoorbeeld met 8 lijnen doorgestuurd naar een ontvanger.
  - Seriële IO: één byte wordt bit per bit over 1 lijn overgestuurd (bv. via UART, staat voor Universal Asynchronous Receiver-Transmitter).
  - Analoge IO: het voltage toegevoerd aan een analoge input wordt door een ADC (Analog Digital Converter) omgezet tot een binaire representatie. Analoge input kan bijvoorbeeld een microfoon zijn.
    Bij DAC (Digital Analog Converter) wordt vanuit een digitale waarde een voltage gegenereerd. Dit wordt bijvoorbeeld gebruikt bij het genereren van de voltages voor een luidspreker.
- **Watchdog timer:** dit is een component die af en toe een trigger moet krijgen, zo niet wordt de CPU gereset. Dit voorkomt dat de CPU, door bijvoorbeeld een bug, voor altijd in een lus blijft 'vastzitten'.

Bovenstaande lijst is niet compleet. De fabrikant van een microcontroller kan ervoor kiezen om meer of minder van dergelijke periferie te implementeren. Het is aan de ontwerper van het volledige systeem om een verstandige keuze te maken qua microcontrollertype.

Om een duidelijk overzicht te bekomen van de interne modules van onze microcontroller, bekijken we best onderstaand blokschema:



Figuur 4: overzicht van de modules in de STM32F091RC (bron: st.com)

1.6 Classificatie

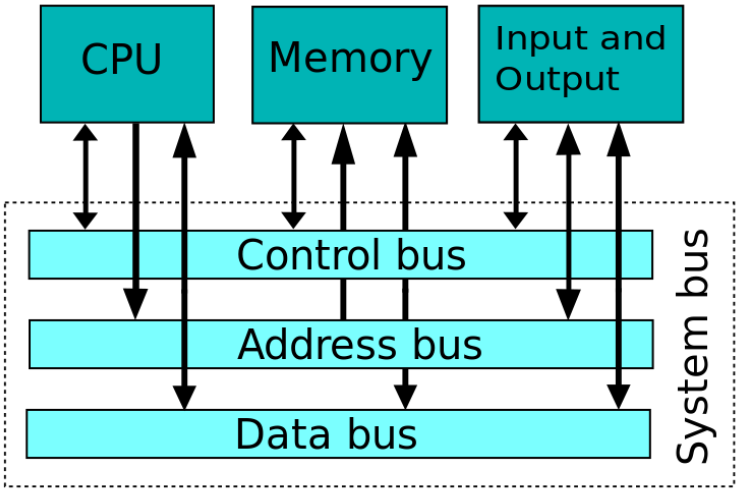
Microcontrollers kunnen onder meer ingedeeld worden volgens:

- Architectuur
- Woordbreedte
- Complexiteit instructieset
- Type instructieset
- Fabrikant en model

1.6.1 Architectuur

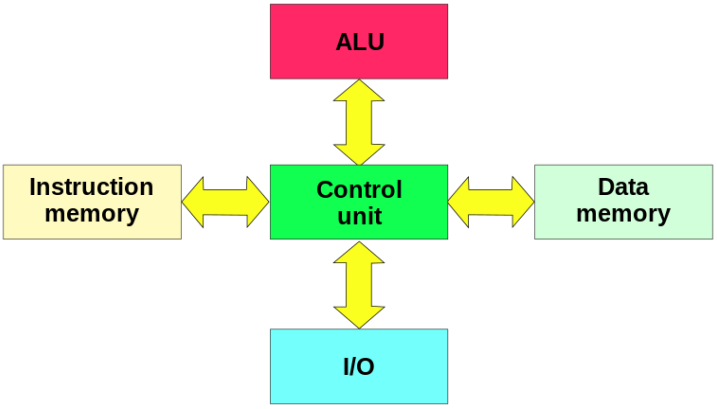
Von Neuman architectuur
Bij de Von Neumann-architectuur maakt de CPU gebruik van **één systeembus** voor zowel het datageheugen (waar de gegevens worden opgeslagen) als het programmageheugen (waar de code/instructies worden bewaard). Dit betekent dat zowel instructies als data over dezelfde bus moeten worden getransporteerd, wat kan leiden tot een 'bottle neck': de CPU kan niet tegelijkertijd instructies en data ophalen, wat de prestaties kan beperken.

Het grootste voordeel van de Von Neumann-architectuur is de eenvoud van het ontwerp, wat het aantrekkelijk maakt voor veel toepassingen. Het nadeel is echter dat de gedeelde bus voor instructies en data de snelheid kan beïnvloeden.
De meeste microprocessoren zijn hoofdzakelijk volgens deze architectuur gemaakt.



Figuur 5: Von Neuman-architectuur (bron: wikipedia.org)

Harvard architectuur
De Harvard-architectuur heeft **gescheiden bussen** voor het datageheugen en het programmageheugen, wat betekent dat de CPU gelijktijdig gegevens en instructies kan ophalen zonder te wachten op de andere bus. Dit kan de prestaties aanzienlijk verbeteren, vooral in systemen waar snelheid cruciaal is.
De meeste microcontrollers zijn met deze architectuur ontworpen.



Figuur 6: Harvard-architectuur (bron: wikipedia.org)

Een ander voordeel van de Harvard-architectuur is dat het datageheugen en het programmageheugen verschillende eigenschappen kunnen hebben, zoals verschillende timing of woordbreedte. Dit maakt het mogelijk om verschillende soorten geheugen te gebruiken voor instructies en data, zoals ROM/flash voor het programma en RAM voor de gegevens.

Bij de **gewijzigde Harvard-architectuur** (Modified Harvard Architecture) zijn de twee geheugens, hoewel gescheiden, na de cache niet volledig geïsoleerd van elkaar. Moderne microprocessoren gebruiken vaak een gescheiden cache voor zowel instructie- als datageheugen, maar na de cache worden de bussen weer samengevoegd op één bus voor verdere communicatie met de CPU. Dit biedt een balans tussen de voordelen van gescheiden geheugens en de kosten van een volledig gescheiden systeem.

In sommige gevallen, zoals bij bepaalde microcontrollers, wordt er gedeeltelijk gebruik gemaakt van de Harvard-architectuur. Bijvoorbeeld, bij de PIC18-reeks blijven strings van het programma opgeslagen in ROM in plaats van te worden gekopieerd naar RAM, en kunnen ze via een speciale pointer in de C-taal direct worden benaderd. Dit bespaart geheugen en maakt efficiënter gebruik van de beschikbare resources.

1.6.2 Woordbreedte

De woordbreedte bepaalt op hoeveel data de processor per klokpuls bewerkingen uitvoert.

Microprocessoren in laptops en computers zijn tegenwoordig bijna allemaal 64-bit. Zelfs voor de kleinste pc's wordt veelal een 64-bit processor gebruikt.

Er zijn momenteel nog veel 8-bit en 16-bit microcontrollers in gebruik. Er is wel duidelijk een trend naar 32-bit microcontrollers. De prijs van dergelijke 32-bitters is de laatste jaren fel gedaald, waardoor het ook in goedkopere systemen mogelijk wordt om een 32-bit microcontroller te gebruiken.

In de praktijk worden de meeste microcontrollers geprogrammeerd in de programmeertaal C.
In deze programmeertaal hebben de **integer types niet altijd dezelfde grootte (breedte)**. Dit stelt natuurlijk problemen bij gebruik van deze integer types. Om deze problemen te vermijden definieert men types die onafhankelijk van de woordbreedte dezelfde grootte hebben.

In ons geval voorzien we deze types:

```
/* exact-width signed integer types */
typedef   signed          char int8_t;
typedef   signed short     int int16_t;
typedef   signed           int int32_t;
typedef   signed       __INT64 int64_t;

/* exact-width unsigned integer types */
typedef unsigned          char uint8_t;
typedef unsigned short     int uint16_t;
typedef unsigned           int uint32_t;
typedef unsigned       __INT64 uint64_t;
```

Figuur 7: exact-width integers (stdint.h)

Als we dus bijvoorbeeld een getal willen voorstellen dat als bereik 0 tot 54205 moet hebben en toch zo efficiënt mogelijk gebruik moet maken van de beschikbare resources, kunnen we het best het type *uint16_t* gebruiken.

Bekijk zelf eens enkele andere mogelijkheden.

1.6.3 Complexiteit instructieset

CISC (Complex Instruction Set Computer)
CISC-processoren zijn ontworpen om complexe taken met één enkele instructie uit te voeren. Een CISC-instructie kan meerdere acties omvatten, zoals het lezen van een waarde uit het geheugen, het uitvoeren van een bewerking en het terugschrijven van het resultaat naar het geheugen.
Bijvoorbeeld:

- Instructie: bereken de som van twee getallen uit het geheugen en sla het resultaat direct op in een register of geheugenlocatie.

CISC-architecturen hebben doorgaans een grote en diverse instructieset, wat de hardwarecomplexiteit verhoogt. Dit maakt ze geschikt voor systemen waar compatibiliteit met bestaande software belangrijk is, zoals bij de Intel x86-architectuur, die veel wordt gebruikt in desktops en servers.

RISC (Reduced Instruction Set Computer)
RISC-processoren zijn ontworpen met eenvoud en efficiëntie in gedachten. In tegenstelling tot CISC zijn de individuele instructies in een RISC-architectuur zeer eenvoudig. Deze eenvoud maakt het mogelijk om een hoge kloksnelheid te bereiken, omdat elke instructie in één enkele kloksnelheidscyclus kan worden uitgevoerd.
De term "reduced" verwijst niet naar het aantal instructies in de set, maar naar de hoeveelheid werk die per instructie wordt uitgevoerd.

Bijvoorbeeld:

- Instructies:
  1. Lees een eerste waarde uit het geheugen naar een register.
  2. Lees een tweede waarde uit het geheugen naar een register.
  3. Bereken de som van de twee waarden in de registers en schrijf het resultaat weg naar een register.
  4. Schrijf het resultaat naar het geheugen.

Een voorbeeld van een veelgebruikte RISC-architectuur is ARM, die niet alleen in onze STM-microcontroller voorkomt, maar ook in tablets en smartphones.

1.6.4 Type instructieset

De bekendste instructieset is wellicht de **x86**. Dit is een instructieset die ontwikkeld werd voor het gebruik op de Intel 8086 processoren. Oorspronkelijk was dit een 16-bit instructieset. Later volgden 32- en 64-bit versies.
Andere instructiesets zijn PowerPC, SPARC, PIC, ARM.

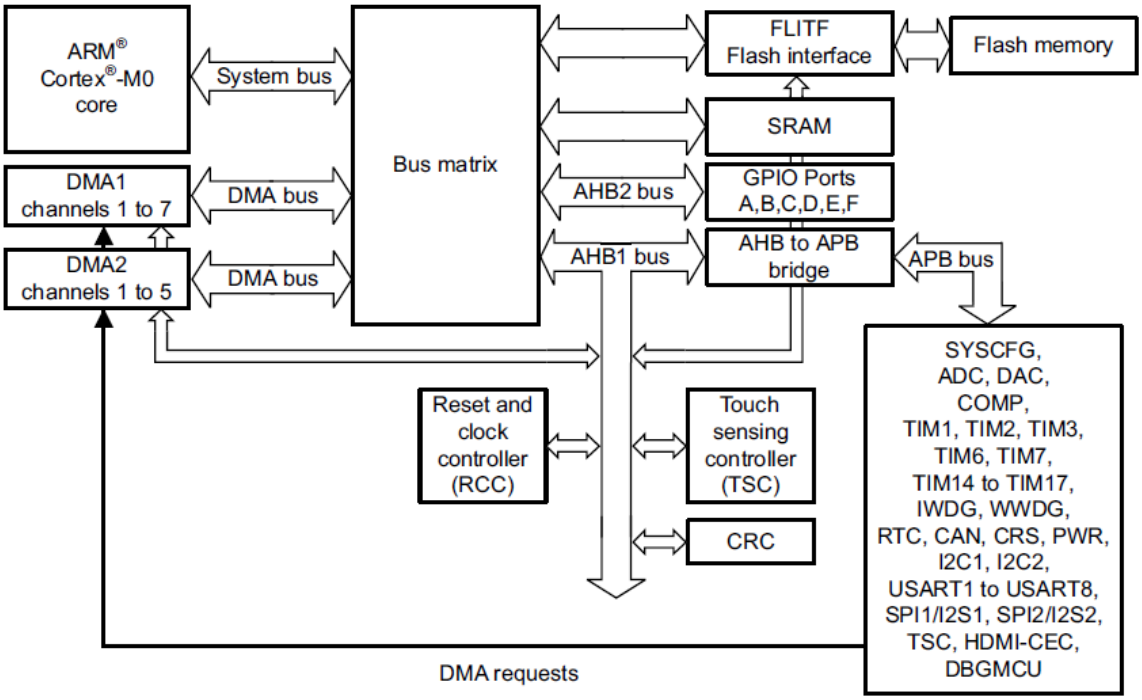
Veelal is een instructieset gekoppeld met een fabrikant. Echter wordt de x86 niet enkel door Intel, maar ook door meerdere fabrikanten gebruikt waaronder AMD.

Van de **ARM instructieset** zijn er ook meerdere fabrikanten. De firma ARM (www.arm.com, weetje: ARM staat trouwens voor Advanced RISC Machines) produceert zelf geen microprocessoren maar verkoopt ontwerpen aan andere firma's zoals NXP en Qualcomm.
Ook van onze microcontroller STM32F091RC is de architectuur ontworpen door het bedrijf ARM, maar in een chip geplaatst door de fabrikant ST (www.st.com).

2 STM32-architectuur (STM32F091RC Cortex M0)

2.1 Blokschema

Onze STM32F091RC ARM microcontroller heeft onderstaande architectuuropbouw:



Figuur 8: STM32F0-architectuur (bron: STM32F0 Reference Manual - DM00031936)

Merk op dat het niet meer zo eenvoudig is om hier nog de opsplitsing te maken tussen Von Neuman en Harvard. De ARM-architecturen zijn eigenlijk specifiek ontworpen met meestal één bepaalde focus voor ogen.

In ons geval is dit bij de Cortex M0-reeks: *"The ARM® Cortex®-M0 processor is the smallest ARM processor available. The exceptionally small silicon area, low power and minimal code footprint of the processor enables developers to achieve 32-bit performance at an 8-bit price point, bypassing the step to 16-bit devices. The ultra-low gate count of the Cortex-M0 processor also enables its deployment in analog and mixed signal devices."*
(bron: http://www.arm.com/products/processors/cortex-m/cortex-m0.php)

2.2 Beschrijving

Bovenstaand blokschema kan logisch ingedeeld worden in Masters, System bus, Slaves en AHB naar APB brug.

Masters
Bovenstaand blokschema heeft faciliteiten voor drie masters. Een master speelt baas in een elektronisch systeem. Andere componenten uit het zelfde systeem moeten gehoorzamen/luisteren naar wat de master hen vraagt. De **drie masters** hier zijn: **ARM Cortex M0 core**, **DMA1** en **DMA2**.

In de STM32F091RC is de ARM Cortex M0 core hét hart van het systeem (de belangrijkste master). Dit is de CPU waar alle berekeningen plaatsvinden. Een Cortex M0 core is een 'entry level' ARM microcontroller. Dat wil zeggen dat het een van de eenvoudigste ARM Cortex cores is die er bestaan. De complexere Cortex cores worden bijvoorbeeld ingezet bij smartphones maar vallen buiten het bestek van deze cursus.

De DMA1 en DMA2 module zorgen voor Direct Memory Access faciliteiten. DMA-gebruik is echter niet voor beginners, daarom volgt meer uitleg in een vervolgcursus.

System bus
De systeembus uit voorgaand blokschema verbindt de CPU met de Bus Matrix. Die Bus Matrix regelt de arbitrage tussen de core en de DMA-bussen, m.a.w. het zorgt dat er geen bits botsen en dergelijke. Het is dus die Bus Matrix die bij een DMA request de nodige stappen zal ondernemen om de CPU te ontheven van zijn heel repetitieve tijdrovende acties en zo de efficiëntie van heel het systeem omhoog te krikken door via DMA-technieken bepaalde acties te voltooien.

Slaves
Als er masters zijn, moeten er ook slaves zijn. In de Cortex M0 core zijn dit er vier:

- **FLITF**: FLash memory InTerFace
- **SRAM**: Static RAM
- **AHB1** (Advanced High-performance Bus 1) met "**AHB naar APB-brug**"
  - APB staat voor Advanced Peripheral Bus
- **AHB2**: Advanced High-performance Bus 2

AHB naar APB brug
De AHB1, bezit een brug (bridge) naar de APB, deze laatste maakt connectie met Timers, UART's, RTC, I^2C, … Doorheen die brug kan de periferie de CPU bereiken. Dit wordt verder in de cursus heel belangrijk.

3 Gebruikte hardware

In dit deel beschrijven we wat we zullen gebruiken van hardware.

3.1 STM32F091RC

De keuze van de microcontroller is bij de aanvang van een embedded project heel belangrijk. Zaken waar men zeker rekening moet mee houden zijn:

- Wat mag de kostprijs zijn?
- Hoe vlot kunnen we de microcontrollers aankopen?
- Hoeveel bordruimte (grootte) is er beschikbaar op de print?
- Hoeveel berekeningen per seconde willen we hebben?
- Op welke spanning willen we werken?
- Welk stroomverbruik is toegelaten?
- Hoeveel IO's hebben we nodig?
- Welke communicatiekanalen willen we gebruiken (UART, SPI, CAN, …)?
- In welke programmeertaal willen we programmeren?
- …

Met bovenstaande in het achterhoofd is het onmogelijk om de perfecte microcontroller te vinden, maar hebben we de STM32F091RC-microcontroller gekozen om volgende redenen:

- 32-bit microcontroller (grote rekenkracht)
- Tot 48 MHz klokfrequentie (tamelijk grote rekenkracht)
- Alle basis communicatiekanalen zijn aanwezig (UART, SPI, I²C, CAN, …)
- Prijs
- Het is een ARM microcontroller en wordt wellicht 'de' standaard de komende jaren.
- Snel beschikbaar via ontwikkelborden
- De ontwikkelborden zijn uitbreidbaar via Arduino compatibele stacking headers en de microcontroller is programmeerbaar via USB-interface
- …

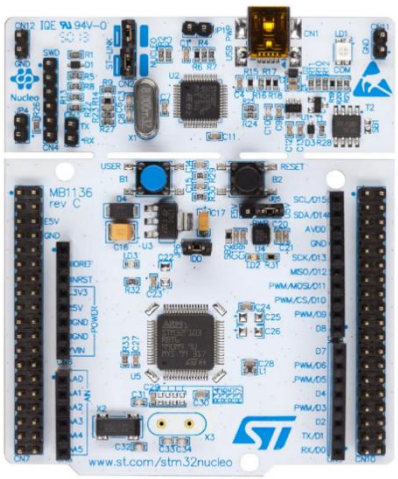
De specifieke specificaties van de microcontroller zijn terug te vinden in Figuur 4.

De microcontroller is beschikbaar in LQFP64, dus het dient om op een printplaat (PCB) te worden geplaatst en is daardoor minder geschikt voor directe plaatsing op bijvoorbeeld een breadboard of gaatjesprint. Dit maakt het lastig om snel te testen en prototypes te bouwen.
Om dit probleem op te lossen, heeft STMicroelectronics een reeks **ontwikkelborden**, de **Nucleo-borden**, ontworpen.

3.2 NUCLEO-F091RC

Een Nucleo-bord maakt het eenvoudig om verbindingen te kunnen leggen met jumperkabels, (Arduino) shields te gebruiken, code te programmeren via USB-interface en de microcontroller te debuggen. Het Nucleo-bord bevat daarnaast een drukknop, een resetknop en een LED.

In deze cursus zullen we gebruikmaken van het **NUCLEO-F091RC-bord** waarbij het typenummer uiteraard verwijst naar de gebruikte microcontroller.



Figuur 9: Nucleo-bord

3.2.1 Schema

Het schema van het Nucleo-bord is gemeenschappelijk met andere Nucleo-borden waarop een 64-pin ARM op gesoldeerd is. Dit geeft onmiddellijk ook aan dat STMicroelectronics de kaart trekt van opschaalbaarheid en compatibiliteit. Als we een systeem hardwarematig ontworpen hebt, de print getekend dus, kunnen we in principe een andere 64-pin uit de M0-reeks op het bord solderen met andere specificaties, maar dezelfde hardware standaard. Dit bevordert uiteraard het hergebruik van hardware.

Het schema van het Nucleo-bord maakt integraal deel uit van de cursus en is terug te vinden op Toledo. Bestudeer het schema en maak het je eigen zodat je tijdens het schrijven van code steeds weet waar je hardwarematig naar toe wil.

Merk op dat er twee soorten uitbreidingsconnectoren aanwezig zijn op het Nucleo-bord (op Toledo zijn er afbeeldingen te vinden van de functionaliteiten per pin):
1. **Arduino** Uno compatibele stacking headers
2. **ST Morpho** headers

Beide zorgen voor relatief eenvoudige plug-and-play uitbreidingsmogelijkheden want het Nucleo-bord is beperkt van het aantal drukknoppen en LED's. We zullen dus bijkomend gebruik maken van een zelfgemaakt shield, de **Nucleo Extension Shield (NES)**, dat ingeplugd wordt via de Arduino headers.

3.2.2 ST-link

Op het Nucleo-bord is er een ST-link programmer voorzien via USB-interface. Met die programmer kunnen we de microcontroller herprogrammeren. Mocht die ST-link niet voorzien zijn, zouden we via een externe (ST-link) programmer ons programma in de microcontroller moeten programmeren.

Figuur 10: externe ST-link programmer (bron: st.com)

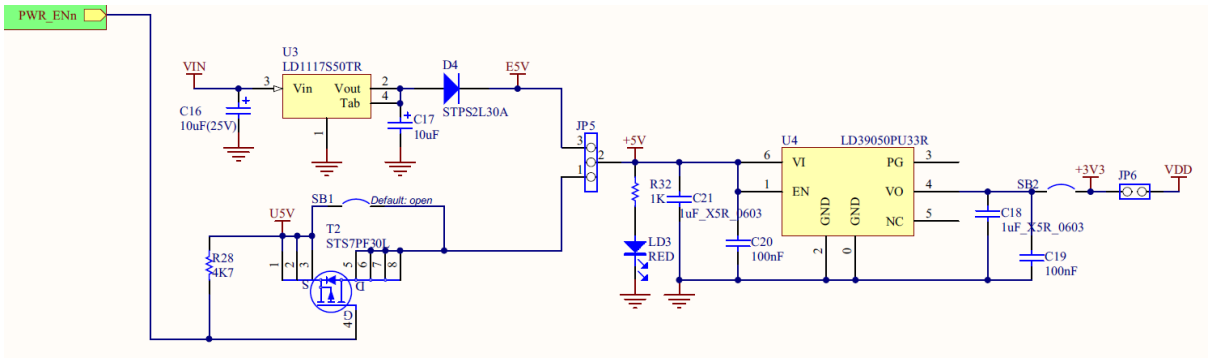
Opmerking: we kunnen ons Nucleo-bord echter ook 'omtoveren' tot een ST-link programmer om ook externe microcontrollers te kunnen programmeren.
Hiervoor moeten we de jumpers op CN2 verwijderen en hebben we de SWD-connector van CN4 ter beschikking.

3.2.3 Externe voeding

Het Nucleo-bord kan gevoed worden via de USB-poort die op een computer is aangesloten. Op die manier kan de totale schakeling 300 mA aan stroom verbruiken. Als we via USB willen voeden zonder dat het bord aan een computer hangt (en dus niet geënumereerd wordt), moeten we jumper JP1 kortsluiten. Dan mag de schakeling wel slechts 100 mA stroom verbruiken. Meer info is terug te vinden in de 'Nucleo user manual' (UM1724 - User manual).

Als we het Nucleo-bord willen voeden met een voedingsbron die niet uit de USB-poort komt van een computer, moet er een aantal zaken aangepast worden.

Bekijk het gedeelte van het voedingscircuit maar eens:



Figuur 11: voedingscircuit van het Nucleo-bord

Hebben we een voedingspanning ter beschikking die nog **niet gestabiliseerd** is op 5 V dc, dan kunnen we best via VIN binnenkomen met de voedingsbron. Via U3 wordt er dan 5 V dc afgeregeld. Daarna maakt U4 er 3V3 van. Om dit te laten werken moet JP5 pin 2 en 3 verbonden worden.

Hebben we echter wel een **gestabiliseerde** 5 V dc bron ter beschikking, kunnen we rechtstreeks op de +5V pin binnenkomen. Ook om dit te laten werken moet JP5 pin 2 en 3 verbonden worden.

Er zijn twee mogelijkheden om een gestabiliseerde 5 V bron aan te sluiten:

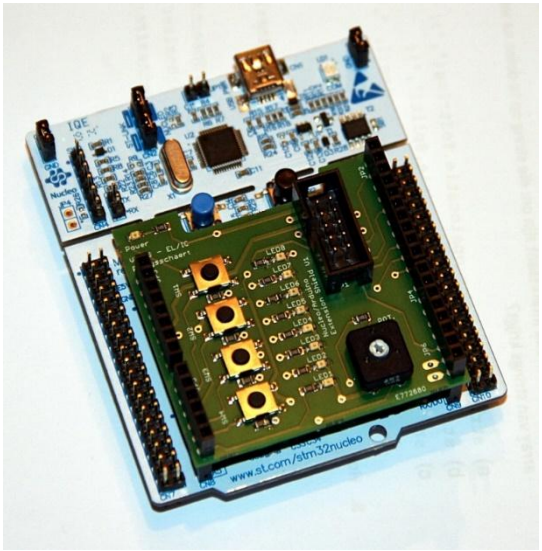
1. De pinnen van connector CN6 (Arduino):
    a. pin 5 = +5 V dc
    b. pin 6 = GND

2. De pinnen van connector CN7 (Morpho):
    a. pin 24 = +5 V dc
    b. pin 22 = GND

Let wel op: foute verbindingen op dit niveau kan het volledige bord onherroepelijk stuk maken!
Zoek zelf eens uit wat de eigenschappen zijn van U3 en U4.
Wat is de bedoeling van PWR_ENn?

3.3 Nucleo Extension Shield

Bovenop dat Nucleo-bord pluggen we een zelfgemaakt extension shield in. De locatie van de stackable headers is zodanig dat die compatibel is met de Nucleo-borden, maar ook met de Arduino Uno R3 borden.
Het shield zorgt voor wat invoer- uitvoermogelijkheden (4 drukknoppen, 8 LED's, 1 trimmer en een 10-pin header voor het gebruik van bijvoorbeeld de UART, I²C, SPI, ….
In een vervolgcursus koppelen we dan verder andere modules en sensoren zodat een breed scala aan oefeningen mogelijk wordt.



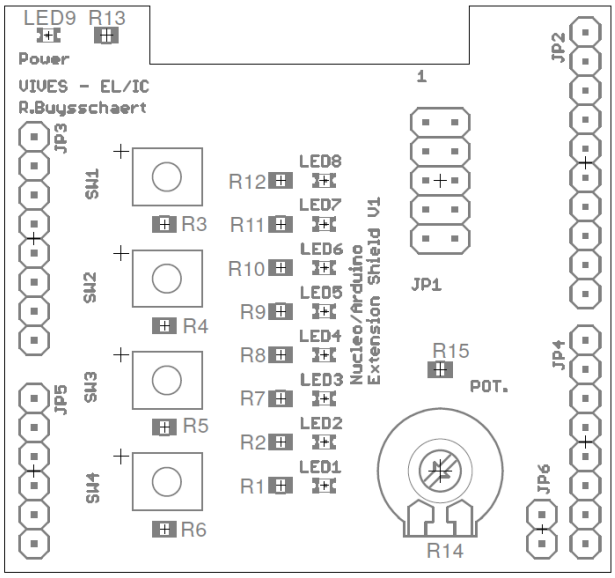
Figuur 12: Nucleo Extension Shield op het NUCLEO-F091RC bord

3.3.1 Schema

Net als het schema van het NUCLEO-F091RC bord maakt het schema van het Nucleo Extension Shield integraal deel uit van de cursus en is terug te vinden op Toledo.
Dit zijn documenten die je steeds bij je moet hebben tijdens de lessen. Dit kan op papier of anders in PDF-formaat op je laptop.

3.3.2 Lay-out

Het bovenaanzicht van de bord lay-out wordt hieronder weergegeven.



Figuur 13: Nucleo Extension Shield - bovenaanzicht

4 Gebruikte software

In dit deel beschrijven we wat we zullen gebruiken van programmeertaal en software.

4.1 Programmeertalen

Er bestaan vandaag de dag veel programmeertalen, maar slechts een beperkt aantal wordt gebruikt voor het programmeren van microcontrollers.

4.1.1 Assembly

Assembly is een low-level programmeertaal. Assembly staat dus heel dicht bij de hardware en is net nog leesbaar voor de mens.

Meestal is het zo dat een assembly-instructie een 1-op-1 overeenkomst heeft met de effectief mogelijk uit te voeren instructies op een microprocessor. De microprocessor begrijpt namelijk enkel de machinetaal, dit is niet anders dan een reeks hexadecimale waarden.

Een voorbeeld van een assembly-programma staat hieronder:

```
    .syntax unified
    .cpu cortex-m0
    .thumb

    .section .text
    .global _start

_start:
    LDR R0, =0x20000000   @ Laad een geheugenadres in register R0
    MOV R1, #0x55         @ Zet waarde 0x55 in register R1
    STR R1, [R0]          @ Sla de waarde van R1 op in het geheugenadres van R0
    B .                   @ Oneindige lus
```

Figuur 14: Assembly-codefragment (voor ARM Cortex M0)

Hierin zien we enkele 'assembly-instructies' zoals:
LDR = Load Register
MOV = Move data naar een register
STR = Store Register
B = Branch (spring naar een adres)

Het voordeel hiervan is dat we exact weten hoeveel tijd ons programma in beslag neemt. Iedere instructie heeft namelijk een duidelijke omschrijving van zijn duurtijd (meestal 1 of 2 klokcycli). Op die manier kunnen we heel tijdkritische programma's ontwerpen.

Het nadeel is vooral de moeilijke leesbaarheid. De afkortingen van de instructies zijn niet zomaar als tekst af te lezen en te begrijpen. We kunnen de code ook minder mooi opdelen in logische blokken. Als we het slecht aanpakken, hebben we heel snel 'spaghetti-code'.

Assembly was aanvankelijk de enige manier om microcontrollers te programmeren. Nu wordt ze minder toegepast, tenzij in enkele heel speciale (tijdkritische) gevallen.

4.1.2 C

C is tegenwoordig de meest gebruikte taal voor het programmeren van embedded systemen. Het is ontworpen om relatief ver van de hardware (vergeleken met assembly althans), de programmeur heel flexibel code te laten schrijven. Het is een programmeertaal die op vele plaatsen inzetbaar is en gestructureerd programmeren ondersteunt. De C-code kan via een compiler omgezet worden naar assembly en uiteindelijk naar de machinetaal voor de processor.

Kernighan en Ritchie schreven de originele C-definitie in het boek: *"The C Programming Language"*.

Op een goede manier C-taal schrijven is echter niet zo evident. Om de basistechnieken aan te leren zijn er vele goede boeken en tutorials. Eén veel gebruikte is "Essential C" door Nick Parlante dat ook terug te vinden is op Toledo. Ten gepaste tijde zal teruggegrepen worden naar dit document om op die manier een goede C-programmeertechniek aan te leren.

4.1.3 C++

C++ is als het ware een opvolger van C en is ook algemeen inzetbaar maar met ondersteuning voor object georiënteerd programmeren. C++ staat dus iets verder van de hardware maar is ook goed uitgerust om systemen met gelimiteerde resources te gaan programmeren. C++ wordt dus ook dikwijls gebruikt als er microcontrollers geprogrammeerd moeten worden.

Opmerking: C++ heeft ook veel invloed uitgeoefend op het ontwikkelen van andere programmeertalen zoals Java en C#.

4.1.4 C#

C# is oorspronkelijk een Microsoft .NET programmeertaal die nog verder t.o.v. de hardware staat dan assembly, C en C++.
Naarmate embedded systemen alsmaar krachtiger werden, werd het ook mogelijk om een microcontroller in bv. C# te gaan programmeren. Het is echter zo dat de microcontroller niet direct de C#-instructies kan verwerken. Er is een compiler en een framework nodig om alles draaiende te krijgen.
Een voorbeeld framework is .NET nanoFramework.

4.1.5 Varia

Naast C# zijn er nog andere programmeertalen voor microcontrollers zoals Python (via MicroPython of CircuitPython), JavaScript (via Espruino), Rust (relatief nieuwe taal die steeds meer terrein wint voor embedded systemen vanwege de focus op veiligheid en prestaties), Ada (taal die specifiek is ontworpen voor betrouwbare systemen en vaak gebruikt wordt in de luchtvaart en defensie), …

Net als C#, worden de meeste van deze talen niet rechtsreeks uitgevoerd op de microprocessor en vereist dit een tussenstap van compilatie of interpretatie.

4.2 Keil IDE

4.2.1 Beschrijving

Zoals de fabrikant zelf zegt: *"Keil MDK (Microcontroller Development Kit) is the complete software development environment for a range of Arm Cortex-M based microcontroller devices. MDK includes Keil Studio, the µVision IDE, and debugger, Arm C/C++ compiler, and essential middleware components. It supports all silicon vendors with more than 10,000 devices and is easy to learn and use."*

Het is dus een ontwikkelomgeving waarin zowel voorbeeldcode, een compiler als een IDE (Integrated Development Environment) aanwezig zijn.

Keil µVision 5 kan gedownload worden van hun website (https://www.keil.com/demo/eval/arm.htm) of het installatiebestand is ook te vinden op Toledo. Standaard staat de gratis versie actief die wel een aantal beperkingen heeft, maar dit vormt voor deze cursus geen belemmering. Ook is Keil op dit moment enkel beschikbaar voor Windows.

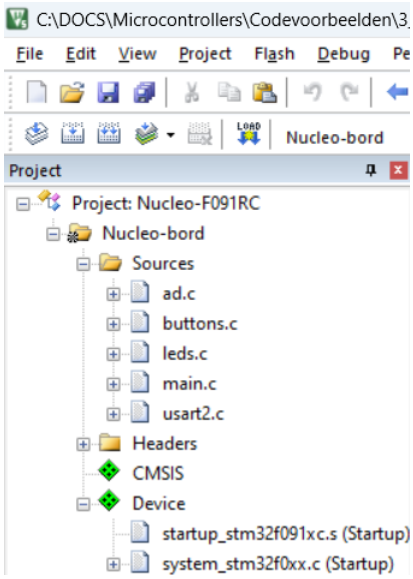
Om met het Nucleo-bord te werken, is het echter niet verplicht om met de Keil software te werken. Het is wel zo dat de andere softwarepakketten niet in deze cursus behandeld zullen worden.

4.2.2 Aanmaken van een project

Om een nieuw project aan te maken in Keil, dient er een heel stappenplan gevolgd te worden. Op Toledo is hiervoor het document *Nieuw project aanmaken in µVision.pdf* beschikbaar.
Het nadeel is dat per project deze stappen opnieuw gevolgd moeten worden, vandaar dat er een aantal sjablonen (zoals *NucleoTemplate*) ter beschikking gesteld zullen worden via Toledo die kunnen dienen als opstart voor de komende projecten.

4.2.3 Compileren en uitvoeren van een project

Open het projectsjabloon door te dubbelklikken op het *.uvprojx*-bestand.



Figuur 15: projectopbouw in Keil µVision

Dubbelklik op het bestand *main.c* en bekijk de C-code in de *main*-functie. Deze code leest de knop op het Nucleo-bord en de vier knoppen op het Nucleo Extension Shield in, en stuurt de acht LED's op het Nucleo Extension Shield aan:

```
29  int main(void)
30  {
31    // Initialisaties.
32    SystemClock_Config();
33    InitIo();
34    InitButtons();
35    InitLeds();
36    InitUsart2(9600);
37    InitAd();
38
39    // Laten weten dat we opgestart zijn, via de USART2 (USB).
40    StringToUsart2("Reboot\r\n");
41
42    // Oneindige lus starten.
43    while (1)
44    {
45      // Knoppen en LED's testen.
46      if(SW1Active() || SW2Active() || SW3Active() || SW4Active() || UserButtonActive())
47        ByteToLeds(255);
48      else
49      {
50        WaitForMs(100);
51        count++;
52        ByteToLeds(count);
53      }
54    }
55  }
```

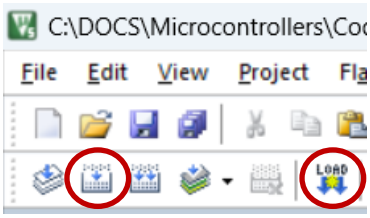
Figuur 16: de main-functie in main.c

Zoals we kunnen zien, worden er heel wat functies aangeroepen.

- Bij initialisatie: *SystemClock_Config()*, *InitIo()*, *InitButtons()*, *InitLeds()*, *InitUsart2()* en *InitAd()*.
- Bij continue uitvoer: *SW1Active()*, *SW2Active()*, *SW3Active()*, *SW4Active()*, *UserButtonActive()*, *ByteToLeds()* en *WaitForMs()*.

Deze functies zijn opgenomen in het sjabloon om het ontwikkelproces te versnellen, maar het zal in deze cursus natuurlijk de bedoeling zijn om de implementatie van deze functies in de komende hoofdstukken grondig te analyseren.

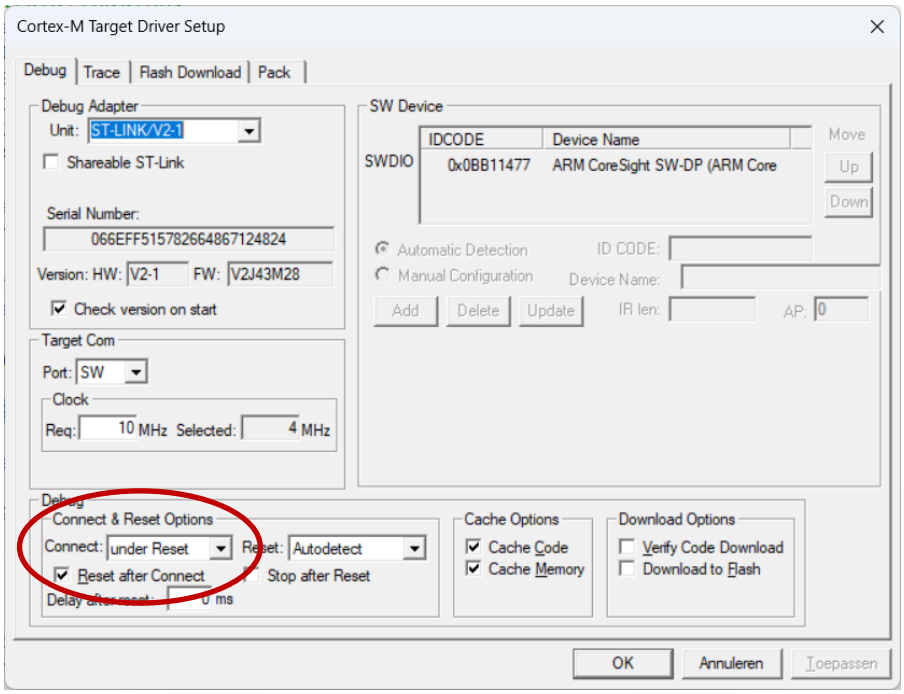
Probeer het programma te uploaden naar het Nucleo-bord met Extension Shield. Klik eerst op compileren. Als er geen fouten in voorkomen, klik vervolgens op de load-knop:



Figuur 17: programma compileren en laden in µVision

Als alles goed gaat, kan je de code uittesten op het bordje.

Als je echter problemen ervaart bij het uploaden, bekijk dan eerst of de drivers correct zijn geïnstalleerd. Als dat het geval is en je ervaart nog 'target-problemen', bekijk dan eens de instellingen van de ST-link debugger:



Figuur 18: instellingen ST-link debugger

4.3 Niveaus van programmeren

Een microcontroller programmeren betekent dat we deze configureren en hardwaremodules gaan aansturen. Dit gebeurt via **peripheral registers** (verwar deze niet met de eerder besproken registers van de microprocessor, dit is totaal iets anders!), dit zijn speciale geheugenlocaties die instellingen opslaan voor IO, timers en andere randapparatuur.
Deze registers bevinden zich niet in het RAM-geheugen, maar in een apart geheugenbereik toegankelijk via memory-mapped IO (MMIO).
Let op: wanneer de microcontroller zonder stroom komt te zitten of gereset wordt, gaan de waarden in deze registers verloren.

In de volgende paragrafen bespreken we de verschillende programmeerniveaus en de mate waarin bibliotheken worden gebruikt.

Figuur 19: lagenstructuur

4.3.1 CMSIS-bibliotheken

In deze cursus zullen we gebruik maken van de **CMSIS** (Cortex Microcontroller Software Interface Standard)-bibliotheken. CMSIS is een door de fabrikant geleverde softwarebibliotheek die helpt bij de aansturing van hardware.
Hiermee kunnen we registers en hardwarefunctionaliteiten efficiënt aanpassen op bitniveau. Dit vereist het includen van het juiste header-bestand:
#include "stm32f091xc.h"

Als we eens naar de implementatie van de functie *ByteToLeds()* kijken die aangeroepen wordt in Figuur 16, dan zien we daar code staan om LED per LED in of uit te schakelen.
Laten we ons eens concentreren op de regel die LED1 zal doen inschakelen:
`GPIOC->ODR = GPIOC->ODR | GPIO_ODR_0;`

De gebruikte benamingen zijn door de fabrikant gedefinieerd en komt grotendeels overeen met de namen van de hoofdstukken en tabellen in de datasheet van de microcontroller.
Het voordeel van CMSIS is de leesbaarheid & structuur en efficiëntie (weinig overhead), maar een nadeel is de afhankelijkheid van het specifieke microcontrollertype. Bij een overstap naar een ander type microcontroller moet de code worden aangepast. Bovendien is de code gevoelig voor logische fouten, zoals verkeerd gezette bits of ontbrekende activeringen.

4.3.2 LL en HAL-bibliotheken

In latere cursussen, zoals *IoT Devices* en *IoT Systems*, maken we gebruik van de **HAL** (Hardware Abstraction Layer)-bibliotheken.
De LL (Low-Level)-bibliotheken slaan we over, deze bevinden zich tussen CMSIS en HAL.

De HAL-bibliotheken bouwen voort op CMSIS en LL en bieden een hoger abstractieniveau. Ze voorzien algemene functies voor het aansturen en inlezen van periferiecomponenten, waardoor de code minder afhankelijk is van het specifieke microcontrollertype.
Dit heeft ook een keerzijde: er is meer overhead en hoger verbruik van resources, grotere kans op bugs en compatibiliteitsproblemen tussen bibliotheken. Verder is er ook minder inzicht in de precieze werking van de microcontroller.

4.3.3 Bare-metal

Bij **bare-metal** programmering werken we zonder CMSIS en andere bibliotheken en schrijven we direct waarden naar geheugenadressen van de peripheral registers. Dit vereist grondige kennis van de microcontroller en zijn datasheet.

Als we deze regel CMSIS-code:
`GPIOC->ODR = GPIOC->ODR | GPIO_ODR_0;`
willen vervangen om geen gebruik meer te maken van een bibliotheek, dan bekomen we dit:

```
#define GPIOC_ODR   (*(volatile uint32_t*)0x48000814)  // Adres van het ODR-register van GPIOC
#define GPIO_ODR_0  (1U << 0)                          // Bitmasker voor pin 0

// Zet pin 0 van GPIOC hoog
GPIOC_ODR = GPIOC_ODR | GPIO_ODR_0;
```

Figuur 20: bare metal codevoorbeeld in C

Dit is de meest directe manier om de microcontroller te programmeren in C.

Opmerking: in de C-taal zijn er twee manieren om **constanten** te definiëren: met **#define** en met **const**.

- ***#define***
  Dit is een preprocessor-directief en is te herkennen aan het #. Ook *#include* is een voorbeeld van een preprocessor-directief. Dit zijn zaken die tijdens de allereerste stap in het compilatieproces wordt uitgevoerd.
  Met *#define* kan er een macro aangemaakt worden. Een macro is een stukje tekst dat tijdens de allereerste stap in het compilatieproces zal worden vervangen door de waarde (de waarde heeft geen datatype!).

  In voorgaand codevoorbeeld betekent dit dat elke keer dat de compiler bv. *GPIOC_ODR* tegenkomt, het automatisch vervangen wordt door *(*(volatile uint32_t*)0x48000814)*.
  Dit zorgt ervoor dat de code makkelijker te lezen is, omdat een "symbolische naam" gebruikt wordt in plaats van een direct geheugenadres.

- ***const***
  De andere manier om constanten in C te definiëren, is door gebruik te maken van *const*. Met *const* maken we een constante variabele, wat betekent dat de waarde van deze variabele niet kan worden gewijzigd na de initiële toewijzing. Dit heeft enkele voordelen ten opzichte van *#define*, vooral wat betreft datatypeveiligheid.

  Equivalente code, nu gebruikmakend van *const*:
  ```
  const volatile uint32_t* GPIOC_ODR = (volatile uint32_t*)0x48000814;
  const uint32_t GPIO_ODR_0 = (1U << 0);
  *GPIOC_ODR = *GPIOC_ODR | GPIO_ODR_0;
  ```

  Een nadeel is dat *const* soms wat meer geheugen kan verbruiken of minder snel is dan *#define*, omdat het een echte variabele is die in het geheugen wordt opgeslagen.

Als we echter de maximale controle willen hebben over de hardware met de hoogste performantie, dan kunnen we Assembly gebruiken:

```
LDR     R0, =0x48000814  // Laad het geheugenadres van GPIOC->ODR in R0
LDR     R1, [R0]         // Lees de huidige waarde van ODR in R1
MOV     R2, #1           // Laad bitmasker voor bit 0 in R2
ORR     R1, R1, R2       // Zet bit 0 in R1
STR     R1, [R0]         // Schrijf nieuwe waarde terug naar ODR-register
```

Figuur 21: bare metal codevoorbeeld in Assembly

Dit is het laagste niveau om de microcontroller te programmeren. Het ontwikkelproces is echter tijdrovender en de code is zeer gevoelig voor fouten.

5 GPIO

In dit hoofdstuk behandelen we GPIO (General-Purpose Input/Output), de eenvoudigste periferie die aangestuurd kan worden.

5.1 Output

5.1.1 Basisinstellingen

Na het resetten van de microcontroller zijn alle pinnen standaard ingesteld als input. Dit voorkomt schade aan de hardware, omdat een input zich als een hoogimpedante verbinding gedraagt en daardoor geen grote belasting vormt voor aangesloten componenten.

Om een pin als output te gebruiken, moeten we enkele instellingen aanpassen. De eerste stap is uiteraard bepalen welke pin we willen gebruiken. Hiervoor raadplegen we, afhankelijk van de toepassing, de **beschikbare documentatie** zoals Nucleo-schema, Nucleo-bordafbeeldingen op Toledo of Nucleo Extension Shield schema.
De pinnen worden steeds aangeduid met de **letter P** gevolgd door de **poortnaam** (A t.e.m. F) en het **pinnummer** (0 t.e.m. 15). **Let op:**

- Er staan in de schema's vaak nog andere pinnamen zoals A0, A1, D1, D2, … . Dit is echter de benaming van de **Arduino-headers** en kunnen we **niet gebruiken** in onze CMSIS-code!
- **Niet alle pinnen** van de microcontroller zijn **extern toegankelijk** via het Nucleo-bord!

Stel dat we LED1 op het Nucleo Extension Shield willen aansturen, zullen we eerst moeten opzoeken op welke pin deze LED is aangesloten om als output in te stellen. Uit het Nucleo Extension Shield schema blijkt dat LED1 op pin PC0 zit.
Vervolgens kan deze pin als output worden ingesteld met de volgende code:

```
// Clock voor GPIOC inschakelen.
RCC->AHBENR = RCC->AHBENR | RCC_AHBENR_GPIOCEN;

// LED1 op output zetten
GPIOC->MODER = (GPIOC->MODER & ~GPIO_MODER_MODER0) | GPIO_MODER_MODER0_0;
```

Figuur 22: codefragment voor het instellen van LED1

De gebruikte benamingen zijn gedefinieerd door de fabrikant in de CMSIS-bibliotheek.
Zo is *AHBENR (AHB peripheral clock enable register)* een register binnen de *RCC (Reset and Clock Control)* periferie en *MODER (GPIO port mode register)* een register binnen de *GPIOC*-periferie (poort C van *GPIO*).
In de code wordt een register bereikt d.m.v. deze constructie: *periferienaam->registernaam*
De betekenis van de -> operator wordt later uitgelegd.

Om de exacte betekenis en mogelijke instellingen van deze registers te achterhalen, dient de datasheet *STM32F0 Reference Manual - DM00031936* geraadpleegd te worden. In deze cursus zullen we dit regelmatig doen.
We navigeren hierin naar het hoofdstuk over de specifieke periferie (RCC en vervolgens GPIO).
Binnen elk hoofdstuk staat in de laatste paragraaf een oplijsting van de registers in tabelvorm met de nodige uitleg.
Elk register is 32-bits breed waarin elke bit een betekenis heeft, maar niet alle bits worden altijd gebruikt (ze worden dan vaak als *reserved* aangeduid).

Merk op dat we in voorgaande code binaire operatoren gebruiken om bitwijzigingen door te voeren.
De binaire operatoren zijn: | & ~ ^ >> <<

In de **eerste stap** zullen we de klok voor poort C gaan inschakelen:
```
RCC->AHBENR = RCC->AHBENR | RCC_AHBENR_GPIOCEN;
```
Hierin wordt de **|** operator (OR) gebruikt om specifieke bits in *AHBENR* in te stellen zonder de andere bits te beïnvloeden.
Dit wordt vaak het **zetten van bits** genoemd.

RCC_AHBENR_GPIOCEN is echter een macro (constante) die in het bestand *stm32f091xc.h*, na wat zoeken, gedefinieerd is als *(1 << 19)*.
De code kan dus eventueel ook geschreven worden als *RCC->AHBENR = RCC->AHBENR | (1 << 19);*
Hierin stelt de **<<** de linker schuifoperator (left shift operator), dit zorgt ervoor dat de waarde '1' gezet wordt gevolgd door negentien keer '0', m.a.w. op bitpositie 19 (te tellen van bit 0) zetten we een 1.
Deze bit betekent volgens de datasheet *"I/O port C clock enabled"*. Als we dit niet zouden inschakelen, zal poort C gewoon niet werken als we hierop één of meerdere pinnen willen gebruiken.

Opmerking: i.p.v. de **decimale notatie met shift operator** *(1 << 19)*, kunnen we ook de **hexadecimale notatie** *0x80000* of gewoon de **decimale notatie** *524288* gebruiken in de C-code. Het nadeel van deze twee laatste notaties is dat het wat moeilijker is om snel te zien welke bits al dan niet gezet zijn geweest in een register, we moeten nl. omrekenen naar binaire waarden.
In de cursus zullen we voornamelijk gebruik maken van de **decimale notatie met shiftoperator** voor het kunnen manipuleren van een **beperkt** aantal bits in een register en de **hexadecimale notatie** voor het kunnen manipuleren van **alle** bits in een register.
(Verder bestaat ook de **binaire notatie** dat in code als *0b10000000000000000000* geschreven kan worden, maar dit is zeer sterk afgeraden! Het wordt al snel omslachtig, want voor sommige waarden zouden we al tot 32-bits moeten neerschrijven. Deze notatie gaan we bijgevolg niet gebruiken!)

In de **tweede stap** zullen we pin PC0 als output zetten:
```
GPIOC->MODER = (GPIOC->MODER & ~GPIO_MODER_MODER0) | GPIO_MODER_MODER0_0;
```

We werken met pin PC0, dus dienen we in *MODER* onder MODER0[1:0] twee bits in te stellen om de pin als output te configureren door de waarde '01' in te stellen.

We vervangen de macro's even door hun werkelijke waarden:
```
GPIOC->MODER = (GPIOC->MODER & ~(3 << 0)) | (1 << 0);
```

Merk op dat *<< 0* weggelaten kan worden, want dit zal niets doen, echter maakt het de code duidelijker dat de waarde 3 (binair '11') vanaf bitpositie 0 wordt geplaatst en de waarde 1 op bitpositie 0.

Het eerste deel van de bewerking *(GPIOC->MODER & ~(3 << 0))* zorgt ervoor dat 3 (= binair dertig keer '0' gevolgd door '11') geïnverteerd wordt d.m.v. de **~**-operator. We verkrijgen dan als resultaat binair dertig keer '1' gevolgd door '00'. (*)
Daarna wordt deze waarde met de **&**-operator toegepast op *GPIOC->MODER*. Dit zorgt ervoor dat de onderste twee bits op '00' worden gezet. Dit proces wordt vaak het **maskeren van bits** genoemd.
Het tweede deel van de bewerking *| (1 << 0)* zal het voorgaand resultaat combineren met een '1'.
Het uiteindelijk resultaat is binair '01' wat we wegschrijven naar *MODER*.

(*) Opmerking: het is in ons codevoorbeeld optioneel om expliciet '00' te gaan wegschrijven in *MODER* vermits '00' al de resetwaarde is. Maar het is een goede gewoonte om dit toch te doen, het kan best zijn dat de pin voorheen in de code eerst een andere modus had. Of, in sommige toepassingen, kan eenzelfde pin hergebruikt worden en wordt er voortdurend geswitcht tussen input en output.

5.1.2 Maximumstroom

Elke IO mag maximum 25 mA leveren (source) of opnemen (sink). De som van alle stromen in de microcontroller mag niet hoger zijn dan 80 mA. Die info kan teruggevonden worden in de datasheet van de STM32F091RC.

5.1.3 Schakelsnelheid

Om het energieverbruik van de microcontroller zoveel mogelijk te beperken kunnen de stijg- en daaltijden van de outputs ingesteld worden. Hoe sneller een output moet schakelen, hoe meer energie daarvoor nodig is. De schakelsnelheid wordt ingesteld via het *OSPEEDR* register.

Vragen:
- Zoek uit hoe dit ingesteld staat na reset en bekijk de maximum schakelsnelheden als we een voedingspanning van 3,3 V gebruiken.

5.1.4 Output aansturen

Vul hieronder aan hoe we LED1 zouden kunnen in- en uitschakelen:

```
GPIOC->ODR =


GPIOC->ODR =
```

Figuur 23: codefragment voor het in- en uitschakelen van LED1

Vragen:
- Analyseer de neergeschreven code en verklaar elk deel.

5.1.5 Output toggelen

Als kort maar krachtig bewijs dat de klokinstellingen (zie later voor meer info) en de outputs van de microcontroller correct zijn ingesteld, kunnen we een knipper-LED maken. Hier nemen we terug LED1 als voorbeeld.

Een **eerste manier** is d.m.v. de LED in te schakelen, even te wachten (delay), de LED terug uit te schakelen, terug even te wachten (delay) en deze cyclus te herhalen.
Een **tweede manier** is via deze code gevolgd door een delay:

```
GPIOC->ODR = GPIOC->ODR ^ GPIO_ODR_0;
```

Figuur 24: codefragment voor het knipperen van LED1

Vragen:
- Wat doet de ^ operator?

Hoe kunnen we nu een vertraging (**delay**) maken zodat we de LED effectief zien knipperen? Er zijn daar verschillende manieren voor, zowel softwarematig als hardwarematig.
Hier bespreken we de softwarematige manier, eigenlijk is dit niet anders dan een lus (of soms zelfs twee geneste lussen) met een hoge tellerwaarde zodat de microprocessor een tijdje zal blijven lussen alvorens verder te gaan met code.
Een eerste aanzet is om bv. deze lus te creëren:

```
// ... code ...

// Een tijdje wachten
for(ticks = 0; ticks < 2000000; ticks++)
{
  __NOP();
}

// ... code ...
```

Figuur 25: codefragment delay

Vragen:

- Wat betekent *__NOP()*? Hoelang duurt het uitvoeren van deze regel?
- Hoelang duurt het uitvoeren van de for-lus?
- Wat is het nadeel van deze delay?
- Welk datatype kiezen we best voor de variabele *ticks*?

5.2 Input

5.2.1 Basisinstellingen

Zoals eerder vermeld, zijn na het resetten van de microcontroller alle pinnen standaard ingesteld als input. We kunnen die dus onmiddellijk gebruiken; het resetten van *MODER* door '00' op de juiste bitpositie weg te schrijven is in feite niet nodig, tenzij de pin voordien een andere modus had.
Wat wel nog steeds nodig is, is het activeren van de klok voor de poort.

Stel dat we SW1 op het Nucleo Extension Shield willen inlezen, bekijken we in het schema op welke pin deze knop is aangesloten. Vervolgens schrijven we de code om de klok te voorzien:

```
// SW1 is standaard input na power on reset, wel nog klok voorzien.
RCC->AHBENR = RCC->AHBENR | RCC_AHBENR_GPIOAEN;
```

Figuur 26: codefragment voor het voorzien van een klok op poort A

5.2.2 Pull-up/pull-down

Om de ontwerper gemakkelijk inputs te laten verwerken, voorziet de microcontroller de mogelijkheid om interne pull-up of pull-down weerstanden te gebruiken. De instelling gebeurt via *PUPDR*.

Vragen:

- Zoek uit wat de instelling is na reset in de datasheet.

Het Nucleo Extension Shield heeft externe pull-up weerstanden *on board*. De interne weerstanden van de STM32F091RC zullen we dus in dit geval niet gebruiken.

5.2.3 Input inlezen

Als basisvoorbeeld van het verwerken van input gaan we de toestand van SW1 inlezen en op LED1 weergeven:

```
if((GPIOA->IDR & GPIO_IDR_1) != GPIO_IDR_1)
    GPIOC->ODR = GPIOC->ODR | GPIO_ODR_0;
else
    GPIOC->ODR = GPIOC->ODR & ~GPIO_ODR_0;
```

Figuur 27: codefragment inlezen van SW1

Vragen:

- Ontleed de code en verklaar elk stuk. Bekijk welke waarde de macro *GPIO_IDR_1* is en waarom het if-statement zo opgebouwd is?
- Hoe kan men weten als een knop een opgaande (*rising edge*) of neergaande flank (*falling edge*) heeft?
    - Maak een extra hulpvariabele aan die de vorige toestand van de knop bijhoudt.

Opmerking: de bovenstaande code wordt voortdurend uitgevoerd en bevindt zich, al dan niet rechtstreeks, in de oneindige while-lus. De if-conditie wordt dus altijd uitgevoerd om na te gaan of de knop is ingedrukt. Dit principe wordt **polling** genoemd.
De code is eenvoudig en vereist niet veel instellingen, maar is niet efficiënt. Een deel van de rekentijd van de microprocessor wordt besteed aan het controleren van dergelijke periferie.
Er bestaan betere technieken, zoals het gebruik van **interrupts**. Zie verder.

6 Basisklokinstellingen

Voordat we verdergaan met bv. andere periferie, is het goed om eerst even stil te staan bij de klokinstellingen.

6.1 Klok

De microcontroller kan alleen zijn werk doen (instructies uitvoeren) als hij wordt gevoed met een kloksignaal. De klok zorgt ervoor dat de microcontroller een instructie start en afhandelt, waarna de volgende instructie volgt, enzovoort.
Ook periferie zoals de ADC, timers en seriële interfaces (UART, I^2C, SPI) zijn hiervan sterk afhankelijk en werken alleen correct als de klok goed is ingesteld.

Hoe sneller de klokpulsen, hoe sneller de microcontroller werkt. Uiteraard is er een bovengrens aan die snelheid. Die bovengrens wordt onder andere bepaald door parasitaire capaciteiten in de printbanen, de imperfecties van de microcontroller-chip ('die'), en het maximaal toegestane energieverbruik.

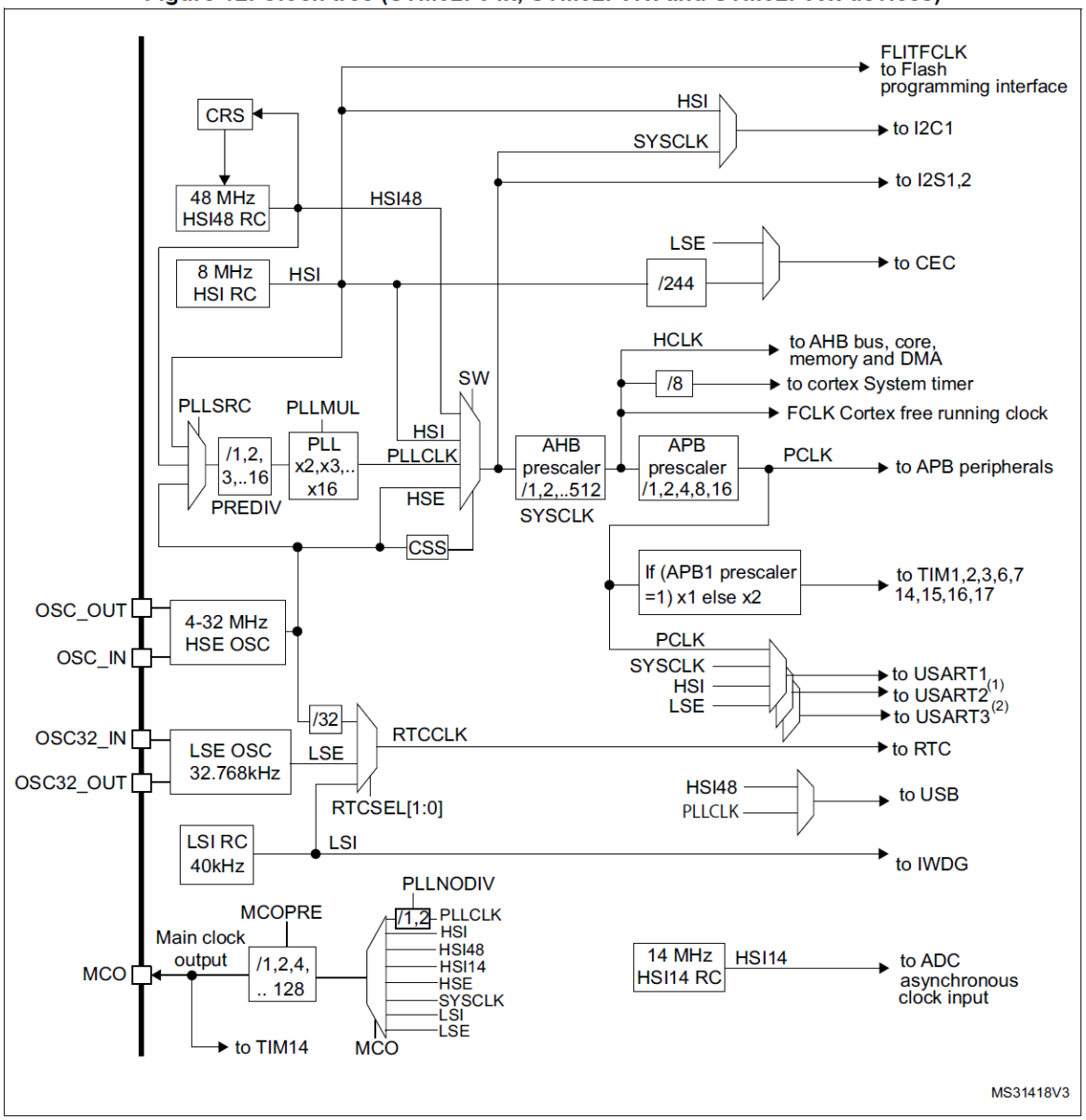
Een microcontroller kan prima functioneren op een klokfrequentie die lager is dan zijn maximale werkbare frequentie. Een hogere frequentie dan toegestaan moet echter wel worden vermeden. Het is aan de ontwerper van het systeem of de software om de juiste klokinstelling te bepalen.

6.2 Kloksysteem

Omdat het instellen van de klok soms vrij ingewikkeld kan zijn, voorzien de meeste fabrikanten een visuele voorstelling van het hele kloksysteem, de zogenaamde **clock tree**.

De *clock tree* op de volgende pagina is heel belangrijk! Zonder een (juiste) klok, geen (correct werkende) microcontrollers…

Figure 12. Clock tree (STM32F04x, STM32F07x and STM32F09x devices)



1. Not available on STM32F04x devices.
2. Not available on STM32F04x and STM32F07x devices

FCLK acts as Cortex®-M0's free-running clock. For more details refer to the *ARM Cortex™-M0 r0p0 technical reference manual (TRM).*

Figuur 28: clock tree STM32F091RC (bron: RM0091)

Vragen:

- Zoek uit langs welk pad we de CPU-kern met **48 MHz** zouden kunnen voeden.
  Let op: gebruik niet de HSI48 RC klok, want die is niet stabiel genoeg en wensen we niet te gebruiken.

6.3 Instellingen

Op het moment dat de microcontroller gereset wordt, wordt gebruik gemaakt van een basisklokinstelling. Onze gebruikte microcontroller heeft intern een HSI (High Speed Internal) RC oscillator van 8 MHz, waardoor een extern kristal niet echt hoeft gebruikt te worden (tenzij we een heel stabiele klok wensen te hebben).

Die HSI RC klok werkt dus op 8 MHz. Die 8 MHz kan via **PLL** vermenigvuldigers en **prescalers** omgezet worden naar 48 MHz, de hoogste snelheid waarop de STM32F091RC kan werken.

Na het opstarten van de microcontroller moet de code voorzien worden om de 48 MHz klok te gaan instellen.
Een mogelijkheid om dit te bekomen wordt hieronder aangegeven:

```
void SystemClock_Config(void)
{
  RCC->CR |= RCC_CR_HSITRIM_4;
  RCC->CR  |= RCC_CR_HSION;
  while((RCC->CR & RCC_CR_HSIRDY) == 0);

  RCC->CFGR &= ~RCC_CFGR_SW;
  while((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_HSI);

  RCC->CR &= ~RCC_CR_PLLON;
  while((RCC->CR & RCC_CR_PLLRDY) != 0);

  RCC->CFGR |= RCC_CFGR_PLLSRC_HSI_PREDIV;
  RCC->CFGR2 |= RCC_CFGR2_PREDIV_DIV2;
  RCC->CFGR |= RCC_CFGR_PLLMUL12;

  FLASH->ACR |= FLASH_ACR_LATENCY;

  RCC->CR |= RCC_CR_PLLON;
  while((RCC->CR & RCC_CR_PLLRDY) == 0);

  RCC->CFGR |= RCC_CFGR_SW_PLL;
  while((RCC->CFGR & RCC_CFGR_SWS) != RCC_CFGR_SWS_PLL);

  RCC->CFGR |= RCC_CFGR_HPRE_DIV1;
  RCC->CFGR |= RCC_CFGR_PPRE_DIV1;

  SystemCoreClockUpdate();
  SysTick_Config(48000);

}
```

Figuur 29: klokinstellingen

Vragen:

- Analyseer deze code goed en zoek uit wat alle registers betekenen, hoe de bitbewerkingen effect hebben, … Leg ook de figuur van de *clock tree* ernaast om dit te volgen.

Ga hier zeker niet licht over! Dergelijke code zal steeds opnieuw voorkomen als we de ARM microcontrollers gaan programmeren. Dit niet begrijpen is geen optie…

7 Interrupts

7.1 Beschrijving

Een interrupt is een verzoek aan de processor om onmiddellijk zijn hoofdprogramma te verlaten en een interrupt routine af te handelen. Deze **interrupt handler** verwerkt de situatie, waarna de processor opnieuw met zijn hoofdprogramma kan verder gaan.

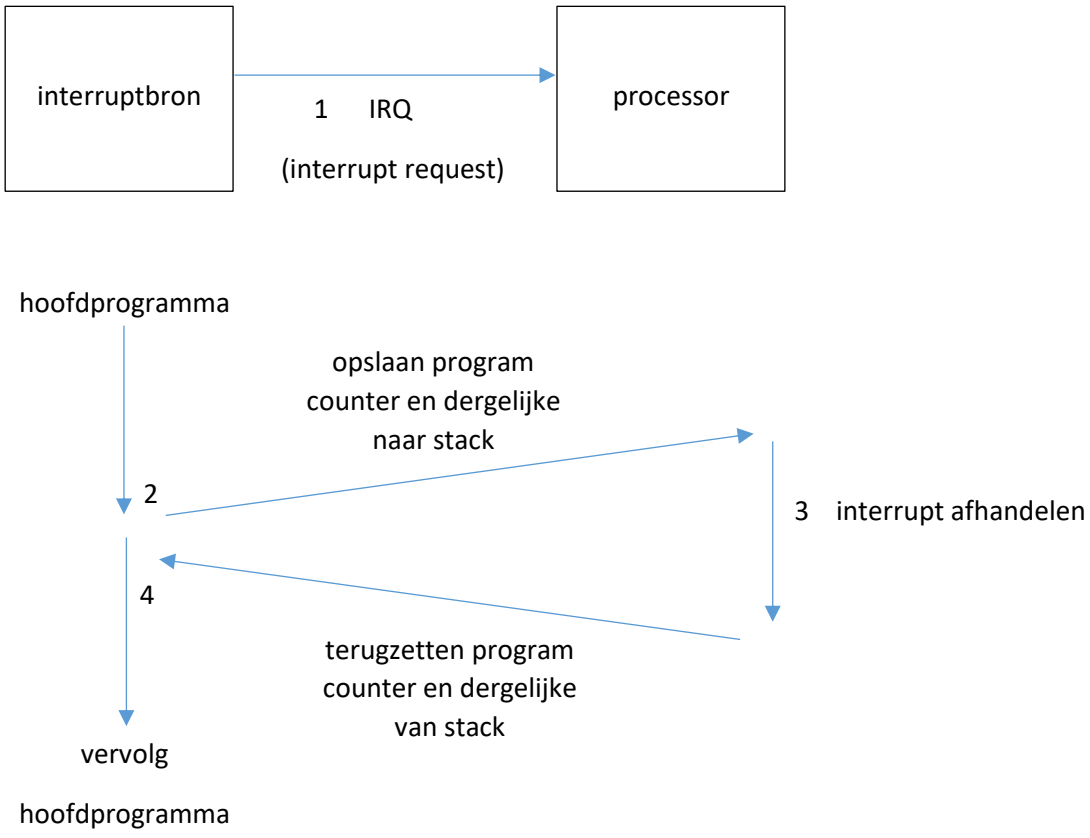
Vlak vóór het verlaten van het hoofdprogramma worden de huidige **program counter** en een aantal andere essentiële processor registers, **weggeschreven naar de stack**. Daarna wordt de program counter ingesteld op het adres van de interrupt (**vector table**) om zo naar de **interrupt handler te verwijzen**.

- Program counter: microprocessor register dat het adres van de volgende instructie bijhoudt.
- Stack: geheugenstructuur die wordt aangemaakt in het RAM-geheugen per uitvoerende/actieve functie in de code om tijdelijk gegevens, zoals registerwaarden en return adressen, op te slaan.

Na het afhandelen van de interrupt worden de gegevens van de stack teruggezet zoals voorheen en kan het **hoofdprogramma** verdergaan zoals dit het geval was vóór de interrupt.

Er zijn verschillende bronnen van interrupts, ze kunnen zowel softwarematig als hardwarematig zijn. Vooral periferie kunnen interrupts veroorzaken.
Het voordeel van het werken met interrupts is dat de aanvrager van de interrupt heel snel op zijn wenken bediend wordt!

Het **interruptconcept** kan als volgt voorgesteld worden:



Figuur 30: visuele voorstelling van het interruptconcept

7.2 NVIC

De **NVIC** (Nested Vectored Interrupt Controller) is de schakel tussen de interruptbron en de CPU en kan 32 interruptkanalen in goede banen leiden. Daarbovenop zijn er nog 16 Cortex M0-interruptlijnen.

7.2.1 Prioriteitsniveaus

Die 32 interruptkanalen kunnen volgens **4 programmeerbare prioriteitenniveaus** (0 tot 3) afgehandeld worden. Als er zich op het zelfde moment twee interrupts voordoen die dezelfde programmeerbare prioriteit hebben meegekregen, wordt eerst de interrupt afgehandeld van diegene met de **laagste vaste prioriteit** in de vector table (zie reference manual datasheet p. 218).

Voorbeeld met *EXTI0_1* en *SPI1*: als er op hetzelfde moment een interrupt binnenkomt van *EXTI0_1* (vaste prioriteit 12 en programmacode adres 0x0000 0054) en *SPI1* (vaste prioriteit 32 en programmacode adres 0x0000 00A4), waarvan de programmeerbare interruptprioriteit beide op 0 stonden, dan wordt eerst de *EXTI0_1* interrupt afgehandeld en daarna de *SPI1* interrupt.

In code wordt dit:

```
NVIC_SetPriority(EXTI0_1_IRQn,0);

NVIC_SetPriority(SPI1_IRQn,0);
```

Figuur 31: programmeerbare interruptprioriteiten gelijk zetten

Opmerking: voorgaande code hoeven we eigenlijk niet expliciet te schrijven vermits de programmeerbare interruptprioriteit standaard steeds op 0 staat.

Als we om programmatorische redenen toch eerst de interrupt van *SPI1* willen afhandelen, kunnen we via de programmeerbare interruptprioriteit de *SPI1* een lagere prioriteit instellen:

```
NVIC_SetPriority(EXTI0_1_IRQn,1);

NVIC_SetPriority(SPI1_IRQn,0);
```

Figuur 32: programmeerbare interruptprioriteiten verschillend maken

7.2.2 Vector mapping

Bij de opstart van de microcontroller moet er gereset worden en naar de *main*-functie gegaan worden. Om dat correct te laten verlopen moet de Vector Mapping goed zijn, ook voor de interrupts wordt die *mapping* belangrijk. Bekijk eens het Assembly-bestand *startup_stm32f091xc.s* in het Keil-project onder *Device*.

```
__Vectors       DCD     __initial_sp               ; Top of Stack
                DCD     Reset_Handler              ; Reset Handler
                DCD     NMI_Handler                ; NMI Handler
                DCD     HardFault_Handler          ; Hard Fault Handler
                DCD     0                          ; Reserved
```

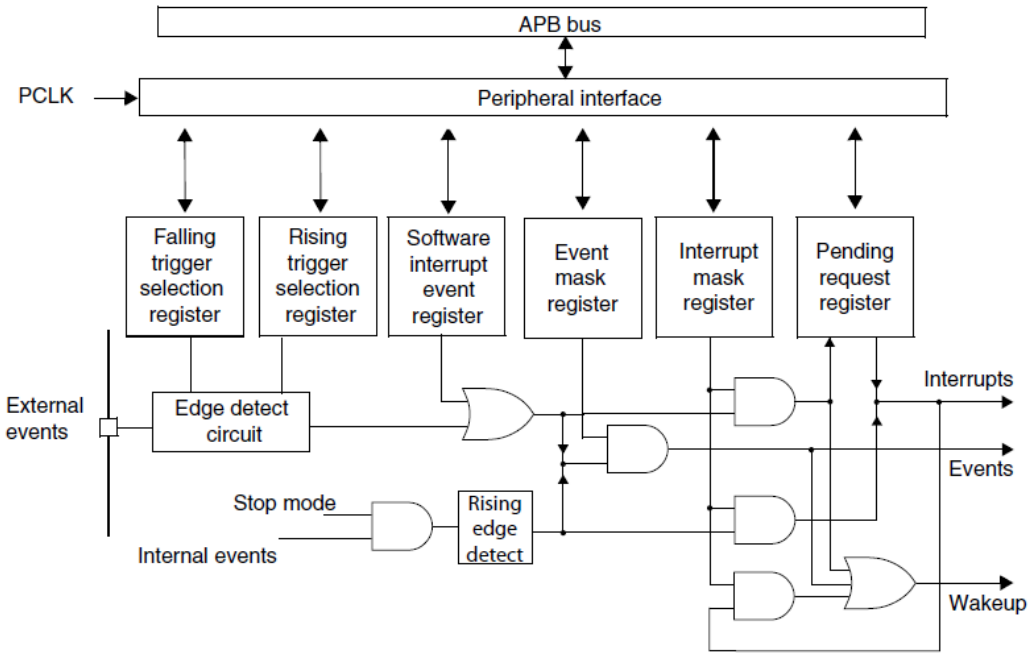
Figuur 33: codefragment startup_stm32f091xc.s

Opmerking: DCD (Define Constant Data) is een Assembly keyword waarmee we één of meerdere 32-bits (4 bytes) woorden in het geheugen kunnen reserveren en initialiseren met specifieke waarden.

7.3 Externe interrupts

Eén type van interrupts die de processor kan verwerken, komt van de EXTernal Interrupt and event controller. Via deze weg kunnen we gemakkelijk interrupts van 32 bronnen (23 externe en 9 interne) tot bij de processor loodsen.

Het principeschema ziet er als volgt uit:



Figuur 34: blokschema externe interrupt en event controller

Vragen:

- Bespreek de verschillende blokken.
- Teken de weg die gevolgd wordt om een stijgende flank op een pin tot bij de core voor een externe interrupt te brengen.

7.4 Andere interruptbronnen

De meeste periferie hebben de mogelijkheid om één of meerdere type interrupts in te stellen. We maken onderscheid tussen **externe** interrupts (zoals stijgende of dalende flank op een GPIO-pin) en **interne** interrupts (zoals aflopende timer, afgeronde AD-conversie, …).
Een aantal hiervan komen verder aan bod in deze cursus.

Telkens wanneer we een interrupt willen gebruiken, moeten alle betrokken registers correct worden ingesteld. Het is raadzaam de reference manual datasheet bij de hand te houden voor de juiste instellingen.

7.5 Instellingen externe interrupts

We kunnen de externe interrupts gebruiken om via een knop een interrupt te genereren. Hieronder staat de code om de interrupt in te stellen voor SW2 op het Nucleo Extension Shield:

```
// SYSCFG clock enable.
RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;

// PA4 (SW2) koppelen aan EXTI 4.
SYSCFG->EXTICR[1] |= SYSCFG_EXTICR2_EXTI4_PA;

// Falling edge detecteren.
EXTI->FTSR = EXTI->FTSR | EXTI_FTSR_TR4;

// Interrupt toelaten.
EXTI->IMR = EXTI->IMR | EXTI_IMR_MR4;

// Eén van de 4 prioriteiten kiezen. Als er twee
// interrupts zijn met dezelfde prioriteit, wordt
// eerst diegene afgehandeld die de laagste
// 'position' heeft (zie vector table NVIC in datasheet).
NVIC_SetPriority(EXTI4_15_IRQn,0);

// Interrupt effectief toelaten.
NVIC_EnableIRQ(EXTI4_15_IRQn);
```

Figuur 35: codefragment instellen externe interrupt voor SW2

De interrupt wordt afgehandeld in deze interrupt handler:

```
// Interrupt handler aanmaken. De definities van
// de handler kan je vinden in startup_stp32f091xc.s.
void EXTI4_15_IRQHandler(void)
{
  // Als het een interrupt is van PA4, de LED's inschakelen.
  if((EXTI->PR & EXTI_PR_PR4) == EXTI_PR_PR4)
  {
    // Interrupt (pending) vlag wissen door
    // een 1 te schrijven...
    EXTI->PR |= EXTI_PR_PR4;

    // LED's inschakelen als bewijs van interrupt.
    GPIOC->ODR = GPIOC->ODR | GPIO_ODR_0;
    GPIOB->ODR = GPIOB->ODR | GPIO_ODR_3;
    GPIOB->ODR = GPIOB->ODR | GPIO_ODR_5;
    GPIOB->ODR = GPIOB->ODR | GPIO_ODR_4;
    GPIOB->ODR = GPIOB->ODR | GPIO_ODR_10;
    GPIOA->ODR = GPIOA->ODR | GPIO_ODR_8;
    GPIOC->ODR = GPIOC->ODR | GPIO_ODR_7;
    GPIOB->ODR = GPIOB->ODR | GPIO_ODR_6;
  }
}
```

Figuur 36: interrupt handler voor SW2

Vragen:

- Neem de reference manual datasheet bij de hand en zoek alle info op van de gebruikte registers in bovenstaande codefragmenten. Verklaar ook wat de C-code precies doet.

8 Strings, structs en pointers in C

In dit hoofdstuk bekijken we hoe we een string kunnen opbouwen in C, wat minder vanzelfsprekend is dan in andere programmeertalen.
Daarnaast gaan we dieper in op de syntax *periferienaam->registernaam* die we al veelvuldig zijn tegengekomen in de voorgaande hoofdstukken zoals *RCC->AHBENR*, *GPIOC->ODR*, *RCC->CR* enz.
Wat betekent die *perifierienaam*, het symbool -> en *registernaam* technisch gezien in C-code?

8.1 Strings

Het is belangrijk om te weten dat C geen ingebouwd **string-datatype** heeft, zoals andere programmeertalen. Strings worden echter voorgesteld als een array van karakters met een afsluitend nulkarakter '\0'.
Als dit laatste karakter wordt vergeten, werken sommige string-bibliotheekfuncties in C niet correct, omdat ze niet weten waar de karakterarray eindigt.

Om een string te maken, dienen we gebruik te maken van een karakterarray. Het declareren en vervolgens initialiseren gebeurt als volgt voor het woord "Test" dat vier karakters telt:

```
char tekst[5];
tekst[0] = 'T';
tekst[1] = 'e';
tekst[2] = 's';
tekst[3] = 't';
tekst[4] = '\0';
```

Figuur 37: declaratie en initialisatie karakterarray in meerdere stappen

De declaratie en initialisatie kan ook in één stap gebeuren, hier mag eventueel de opgegeven arraygrootte tussen [ ] weggelaten worden vermits de compiler dit automatisch zal tellen:

```
char tekst[] = { 'T', 'e', 's', 't', '\0' };
```

Of wat eenvoudiger:

```
char tekst[] = "Test";
```

Figuur 38: declaratie en initialisatie karakterarray in één stap

Opmerking:
In plaats van *char* kunnen we ook *int8_t* of zelfs *uint8_t* gebruiken. Het laatste maakt het mogelijk om ook de uitgebreide ASCII-reeks (128-255) te gebruiken.

8.2 Structs

Een struct is een complex datatype dat meerdere variabelen groepeert onder één naam. De verschillende onderdelen uit een struct kunnen via één overkoepelende naam bereikt worden.

Het voordeel van een struct is vooral dat de programmeur zijn of haar data op een meer logische manier kan structureren. C ondersteunt immers geen klassen en objecten, dus structs zijn de meest geavanceerde datastructuur die beschikbaar is.
Een codevoorbeeld waar we zelf een struct opbouwen:

```
#if !defined(PERSOON_DEFINED)
  #define PERSOON_DEFINED

  struct persoon {
    char voornaam[50];
    char achternaam[50];
    float lengte;
  };
#endif
```

Figuur 39: definitie van een struct

Bovenstaande code kan eventueel via een header-bestand (met extensie .h) toegevoegd worden aan het project. Op die manier blijft alles overzichtelijk en gestructureerd.

Een goede gewoonte is om een header-bestand steeds te laten starten met preprocessor-directief ***#if !defined(NAAM)*** of ***#ifndef NAAM*** waarbij *NAAM* uniek is over het ganse project.
Bv. neem de naam van het bestand en plaats dit in hoofdletters (soms neemt men in de naam ook de h-extensie op, bv. *PERSOON_H*).
Dit zorgt ervoor dat als het header-bestand in meerdere bestanden wordt opgenomen, de compiler niet meerdere keren dezelfde struct definieert, anders kan dit compilatiefouten opleveren.

Het nieuw aangemaakt type *persoon* kan nu gebruikt worden in het hoofdprogramma door bijvoorbeeld de variabele *mezelf* te declareren:

```
struct persoon mezelf;
```

Opmerking: overal waar we een variabele van een struct-type willen aanmaken, moeten we in C steeds *struct* voor dat type plaatsen. Dit kan worden vereenvoudigd d.m.v. *typedef* in de struct-definitie. We mogen structnaam *persoon* eventueel zelfs weglaten (het is dan een anonieme struct).

```
#if !defined(PERSOON_DEFINED)
  #define PERSOON_DEFINED

  typedef struct {
    char voornaam[50];
    char achternaam[50];
    float lengte;
  } Persoon;
#endif
```

Figuur 40: definitie van een struct via een typedef

Nu hoeft er geen *struct* meer te staan bij de declaratie en kunnen we meteen *Persoon* als type gebruiken:

```
Persoon mezelf;
```

Het opvullen van de velden in *Persoon* gebeurt door de variabelenaam *mezelf* op te geven gevolgd door een punt en de veldnaam:

```
//Struct invullen met de gewenste data.
sprintf(mezelf.voornaam, "Ludovic");
sprintf(mezelf.achternaam, "Espeel");
mezelf.lengte = 1.84;
```

Figuur 41: opvullen van de struct-velden

Extra info: *sprintf* is een functie uit de C-bibliotheek *stdio.h* waarmee data kan gekopieerd worden naar een karakterarray. Of we kunnen ook de *strcpy*-functie gebruiken uit de C-bibliotheek *string.h*. Met *sprintf* is het ook mogelijk om variabelen daarin te verwerken, bijvoorbeeld:

```
char buffer[100];
int getal = 42;
sprintf(buffer, "Het getal is %d", getal);
```

Figuur 42: voorbeeld met sprintf-functie

Opmerking: we kunnen de declaratie en initialisatie van een struct ook in één stap uitvoeren:

```
// Attributen worden opgevuld in volgorde (voornaam, achternaam en lengte):
Persoon mezelf = { "Ludovic", "Espeel", 1.84 };

// Attributen worden opgevuld volgens opgegeven attribuutnaam,
// de volgorde speelt geen rol meer:
Persoon mezelf2 = { .achternaam = "Espeel", .voornaam = "Ludovic", .lengte = 1.84 };
```

Figuur 42: declaratie en initialisatie in één stap

8.3 Pointers

Ook in de C-taal worden variabelen opgeslagen op een bepaald adres in het RAM-geheugen. Veronderstel bijvoorbeeld een variabele *getal* dat 8-bit breed is (bv. *uint8_t*), opgeslagen is op adres 12543 en als waarde 138 heeft. Dit kunnen we in het RAM-geheugen min of meer grafisch voorstellen door:

getal (adres = 12543)

138

Het adres van een variabele in C-taal kan opgevraagd worden via het **&**-teken (ampersand). Zo kunnen we stellen dat:
&getal gelijk moet zijn aan 12543.

In de C-taal bestaan er speciale types die **als inhoud een adres bevatten** naar een ander type. Zoiets heet men een **pointervariabele**. In C# kunnen we dit vergelijken met een referentietype.
Om een pointervariabele te maken gebruiken we het ***-teken (asterisk). Als we een pointer naar een unsigned 8-bit getal willen maken, kunnen we dit als volgt doen:
*uint8_t *getalPointer;*

Om daarin dan het adres van een dergelijke 8-bit variabele te plaatsen kunnen we:
getalPointer = &getal;

Als we dit via een grafisch in het RAM-geheugen willen voorstellen:

getalPointer (adres = 14592)

12543

Daarna kunnen we *getalPointer* gebruiken op **twee manieren**:

1. We kunnen de **inhoud van *getalPointer*** opvragen. Die inhoud is dus **het adres van *getal* (12543)**. Dit doen we zo:
   … = getalPointer;

2. We kunnen de **inhoud waarnaar *getalPointer* wijst** opvragen. Die inhoud is de **inhoud van *getal (138)***. Dit noemt men ook vaak **dereferencen van de pointer**.
   *… = *getalPointer;*

Het bovenstaande is wel verwarrend. Waarom? Omdat het ***-teken** dus **twee betekenissen** heeft! We kunnen het gebruiken bij de aanmaak van een variabele, dan creëren we dus een **pointervariabele**.
Maar we kunnen het ook gebruiken bij een bestaande pointervariabele, dan krijgen we de **data waarnaar de pointervariabele verwijst.**

Probeer je dit goed voor te stellen! Maak steeds een tekening van hoe de variabelen zich in het RAM-geheugen gedragen! Enkel dan kan je pointers echt goed begrijpen.

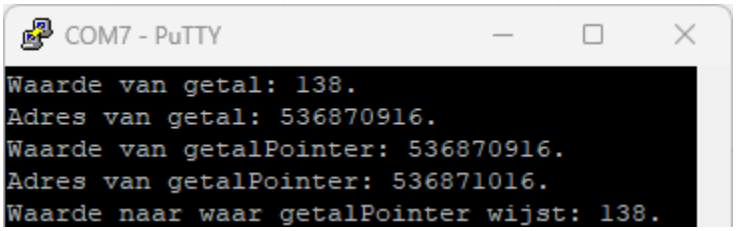
Hieronder volgt een codevoorbeeld met pointers:

```
25  uint8_t getal = 138;
26  uint8_t* getalPointer;

47      sprintf(text, "Waarde van getal: %d.\r\n", getal);
48      StringToUsart2(text);
49
50      sprintf(text, "Adres van getal: %d.\r\n", (uint32_t)&getal);
51      StringToUsart2(text);
52
53      getalPointer = &getal;
54
55      sprintf(text, "Waarde van getalPointer: %d.\r\n", (uint32_t)getalPointer);
56      StringToUsart2(text);
57
58      sprintf(text, "Adres van getalPointer: %d.\r\n", (uint32_t)&getalPointer);
59      StringToUsart2(text);
60
61      sprintf(text, "Waarde naar waar getalPointer wijst: %d.\r\n", *getalPointer);
62      StringToUsart2(text);
63
64      WaitForMs(5000);
65
66      // Pas de waarde van getal aan via getalPointer...
67      *getalPointer = 139;
```

Figuur 43: codevoorbeeld met pointers

Het *-teken kan dus inderdaad twee betekenissen hebben. Ofwel wordt het gebruikt om een pointer naar een bepaald type aan te maken (lijn 26), ofwel voor het opvragen of aanpassen van de data die aan een bepaalde pointervariabele is gekoppeld (lijnen 61 en 67).
Het &-teken wordt gebruikt om van een bepaalde variabele het adres op te vragen (lijnen 50, 53 en 58).

Als we voorgaande code uitvoeren, krijgen we dit te zien op de computer in bv. Putty:



Figuur 44: uitvoer van het codevoorbeeld met pointers

Naslagwerk:
Doorneem zelf nog eens de bijlage *Essential C* voor meer info over strings, structs en pointers. Dit vind je terug in *Section 3: Complex Data Types* op p. 15-23 en in *Section 4: Functions* op p. 26-28.

8.4 CMSIS-structuur

De CMSIS-bibliotheek maakt volop gebruik van structs en pointers om het programmeren gemakkelijker te maken.
Om de **samenwerking tussen structs en pointe**r beter te begrijpen, zullen we eerst onderstaand voorbeeld bekijken: we maken een domoticasysteem voor een huis met twee verdiepingen en op die twee verdiepingen zijn telkens 4 lampen voorzien.

```
16 typedef struct{
17   // Hier stellen de 4 LSB's de 4 lampen voor op het gelijkvloers.
18   // De 4 MSB's zijn 'reserved' en moeten steeds op nul staan.
19   uint8_t verlichtingGelijkvloers;
20
21   // Hier stellen de 4 LSB's de 4 lampen voor op de eerste verdieping.
22   // De 4 MSB's zijn 'reserved' en moeten steeds op nul staan.
23   uint8_t verlichtingEersteVerdieping;
24 }DomoticaSysteem;
25
26
27 DomoticaSysteem huisVanPapaEnMama;    //  ;-)
28 DomoticaSysteem* huisVanDeKindjes;    //  :-p
```

```
39   // Schakel alle vier de lichten op het gelijkvloers aan.
40   huisVanPapaEnMama.verlichtingGelijkvloers = 0x0F;
41   // Schakel alle lichten boven uit.
42   huisVanPapaEnMama.verlichtingEersteVerdieping = 0x00;
43
44   // Omdat de kindjes nog thuis wonen, kan je bijvoorbeeld via
45   // een pointer naar 'huisVanPapaEnMama' ook diezelfde lichten
46   // schakelen... Let op: het gaat dus wel degelijk over dezelfde
47   // lichtpunten! Je schakelt ze gewoon via een andere variabele.
48   huisVanDeKindjes = &huisVanPapaEnMama;
49   // Schakel alle lichten op het gelijkvloers uit.
50   // Merk op dat je nu geen '.' meer mag gebruiken om binnen te
51   // treden in de struct, maar een '->'!!! Verder blijft de
52   // redenering hetzelfde.
53   huisVanDeKindjes->verlichtingGelijkvloers = 0x00;
54   // Schakel 1 licht aan op de eerste verdieping...
55   huisVanDeKindjes->verlichtingEersteVerdieping |= 0x01;
```

Figuur 45: codevoorbeeld met struct en pointers

In het voorgaand codevoorbeeld bevat de pointervariabele *huisVanDeKindjes* het adres van de struct *DomoticaSysteem*.
Om de velden te kunnen lezen of schrijven dienen we **eerst** de inhoud van de pointervariabele op te vragen (*dereferencen*), vandaar dat dit tussen haakjes staat. Vervolgens kunnen we aan het veld:
*(*huisVandeKindjes).verlichtingGelijkvloers = 0x00;*

Vaak vindt men deze syntax wat omslachtig, een elegantere manier is d.m.v. de **-> operator**:
huisVandeKindjes->verlichtingGelijkvloers = 0x00;

Als het voorgaande wat duidelijk is geworden, dan kunnen we zonder twijfel de CMSIS structs en pointers ook begrijpen. Bekijk eens in Keil de inhoud van het bestand *stm32f091xc.h* en speur eens rond naar o.a. deze stukken code:

```
683 #define RCC_BASE              (AHBPERIPH_BASE + 0x00001000)

763 #define RCC                  ((RCC_TypeDef *) RCC_BASE)

463 typedef struct
464 {
465   __IO uint32_t CR;
466   __IO uint32_t CFGR;
467   __IO uint32_t CIR;
468   __IO uint32_t APB2RSTR;
469   __IO uint32_t APB1RSTR;
470   __IO uint32_t AHBENR;
471   __IO uint32_t APB2ENR;
472   __IO uint32_t APB1ENR;
473   __IO uint32_t BDCR;
474   __IO uint32_t CSR;
475   __IO uint32_t AHBRSTR;
476   __IO uint32_t CFGR2;
477   __IO uint32_t CFGR3;
478   __IO uint32_t CR2;
479 } RCC_TypeDef;
```

```
// Clock voor GPIOA inschakelen (on board LED).
RCC->AHBENR = RCC->AHBENR | RCC_AHBENR_GPIOAEN;
```

Figuur 46: codefragmenten CMSIS-structs en pointers

Opmerkingen:

- Het kan zijn dat de regelnummers hier niet (meer) overeenkomen met ons project in Keil, er gebeuren soms updates aan de CMSIS-bibliotheken…
- Op lijn 763 gaan we een 32-bitwaarde *casten* (omzetten) naar een *RCC_TypeDef* pointer en definiëren als *RCC*.

Vragen:

- Bekijk eens hoe bv. *GPIOC* is opgebouwd in het bestand *stm32f091xc.h*.

9 ADC

9.1 Beschrijving

ADC staat voor Analog-to-Digital Converter en met deze module kan dus een willekeurige analoge spanning vertaald worden naar een evenwaardig digitaal getal. Een DAC (Digital-to-Analog Converter) doet het omgekeerde.
Bekijk het document *ADC-DAC.pdf* op Toledo dat dieper ingaat op de werking, dit maakt ook deel uit van de leerstof.

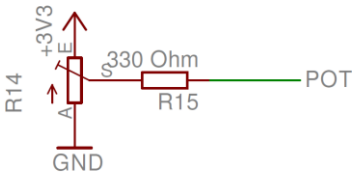
Hou er rekening mee dat niet alle pinnen ondersteuning bieden voor ADC of DAC.
De microcontroller beschikt slechts over twee DAC-kanalen, die elk aan een vaste pin zijn toegewezen. De DAC behandelen we in deze cursus voorlopig niet.

De ADC heeft daarentegen 16 kanalen, die eveneens elk aan een vaste pin zijn toegewezen. Raadpleeg de tabel in de datasheet (p. 34 – 40) om te zien welk kanaal bij welke pin hoort. Je herkent dit aan de benaming "ADC_INx", waarbij x het kanaalnummer voorstelt.

9.2 Werking

De spanning op de pinnen van de microcontroller mag uiteraard niet boven zijn 'absolute maximum ratings' gaan.

Om onze ADC gemakkelijk te kunnen testen staat er op het Nucleo Extension Shield een trimmer geplaatst. Die trimmer kunnen we gebruiken om spanningen tussen 0 en 3V3 te bekomen. De uitgang van de trimmer is aangesloten op GPIOA pin 0.



Figuur 47: trimmer op het Nucleo Extension Shield

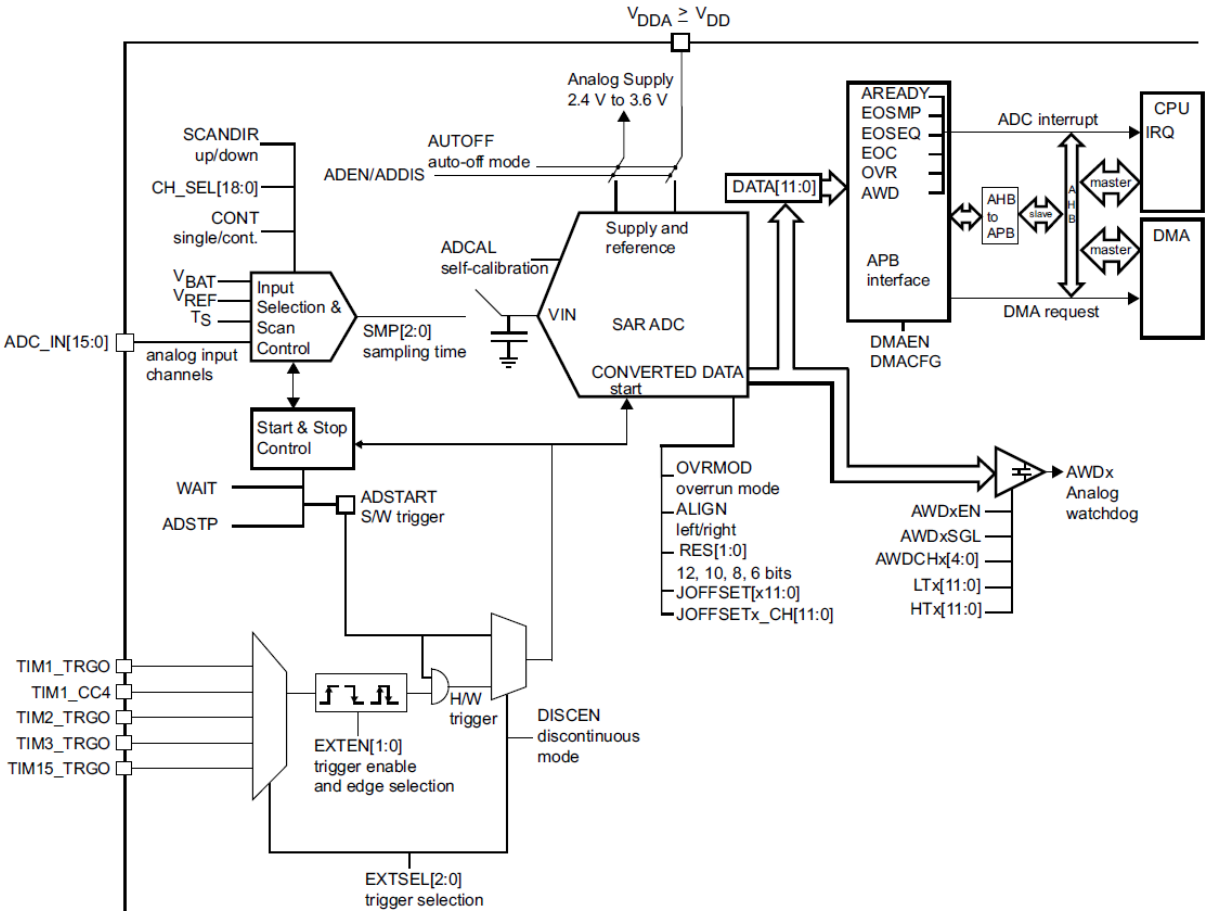
De AD-converter van de STM32F091RC is een **12-bit** converter. De spanningen tussen 0 en 3V3 worden dus opgesplitst in 4096 gelijke stukjes.

Vragen:

- Hoeveel Volt stelt één bitwaarde dan voor?

  1 bit = …… mV

Het blokschema van de ADC ziet er als volgt uit:



Figuur 48: blokschema ADC (bron: RM0091)

Vragen:

- Kan je het pad volgen van de input tot op de databus?

9.3 Instellingen

Om de ADC te kunnen gebruiken moeten een aantal instellingen correct gebeuren. Een aantal heel belangrijke zijn:

1. Plaats pin A0 op analoge mode
2. Voorzie een klok voor de ADC
3. Kies de uitlijning
4. Schakel de ADC in
5. Kies een kanaal om te bemeten
6. Kies eventueel een sample frequentie

Optioneel kunnen er ook nog andere zaken ingesteld worden:

1. Kalibratie uitvoeren
2. Auto turn off
3. Interrupts genereren
4. …

In code wordt dit:

```
// AD-module initialiseren
void InitAd(void)
{
  // Poort A 0 als analoge ingang (potentiometer op het Nucleo Extension Shield).
  GPIOA->MODER |= GPIO_MODER_MODER0;

  // AD-module van klok voorzien.
  RCC->APB2ENR |= RCC_APB2ENR_ADC1EN;

  // Right alignment (optioneel).
  ADC1->CFGR1 &= ~ADC_CFGR1_ALIGN;

  // AD enable'n.
  ADC1->CR |= ADC_CR_ADEN;
  while ((ADC1->ISR & ADC_ISR_ADRDY) == 0);

  // AD-kanaal 0 kiezen (verbonden met de connector voor analog input).
  ADC1->CHSELR = ADC_CHSELR_CHSEL0;

  // Tragere sample rate kiezen (voor hoogimpedante analoge bronnen).
  ADC1->SMPR |= ADC_SMPR_SMP_0 | ADC_SMPR_SMP_1 | ADC_SMPR_SMP_2;
}
```

Figuur 49: ADC-initialisatie

De ADC starten en het resultaat uitlezen kan zo:

```
// De blokkerende versie van de AD-conversie (later met interrupts).
uint16_t GetAdValue(void)
{
  uint32_t temp = 0;

  ADC1->CR |= ADC_CR_ADSTART;                 // Start de AD-omzetting.
  while ((ADC1->ISR & ADC_ISR_EOSEQ) == 0);   // Wachten op einde sequentie
                                              // (sequentie bestaat hier uit één conversie).
  temp = ADC1->DR;

  return (uint16_t)temp;
}
```

Figuur 50: ADC-resultaat uitlezen

Vragen:

- Bekijk bovenstaande code samen met de reference manual datasheet.
- Hoe zou je deze 12-bit waarde van de ADC het best kunnen tonen op de 8 LED's van het Nucleo Extension Shield?

10 Timers

10.1 Beschrijving

Een hardware timer is een digitale teller die hardwarematig (met flipflops) ingebed zit in de microcontroller. Het voordeel van een hardware timer is dat die timer automatisch kan tellen, zonder dat hij daarvoor de tussenkomst van de CPU nodig heeft.
Dit zorgt er dan weer voor dat we met hardware timers heel precies tijden kan gaan meten of instellen en de CPU nuttigere zaken kunnen laten doen. De enige bepalende factor qua nauwkeurigheid is de stabiliteit van de oscillator waarmee de timer gevoed wordt.

Timers in microcontrollers hebben verschillende toepassingen zoals:

- **Delay routines:** vertragingen in de code creëren zonder dat de CPU constant hoeft te blokkeren. De CPU kan andere taken uitvoeren terwijl de timer wacht.
- **Interrupts:** een timer kan een interrupt genereren na een bepaalde tijd. Deze interrupt kan worden gebruikt om periodieke taken uit te voeren.
- **Input Capture (*):** een timer kan gekoppeld worden aan een input pin om gebeurtenissen zoals een opgaande of neergaande flank van een signaal te detecteren. Dit kan handig zijn om bijvoorbeeld de tijd tussen deze twee flanken te meten ofwel de flanken te tellen.
- **Output Compare (*):** een timer kan ingesteld worden om een output pin te activeren wanneer de timer een bepaalde waarde bereikt.
- **PWM-signaal genereren (*):** een timer kan gebruikt worden om een periodiek blokgolfsignaal te genereren met een variabele pulsbreedte.

(*) Opmerking: hier moeten we de pinnen configureren als speciale pinnen die gekoppeld zijn aan een timer-module, zodat ze niet (meer) als gewone GPIO-pinnen functioneren.
Houd er ook rekening mee dat niet alle pinnen deze mogelijkheid bieden!

Onze STM32F091RC beschikt over 12 verschillende timers. Deze timers variëren in complexiteit en functionaliteit: sommige zijn geavanceerd en geschikt voor gecompliceerde toepassingen, terwijl andere eenvoudiger zijn en dienen voor basisfunctionaliteiten zoals timing en signaalgeneratie.
De meeste timers tellen oplopend, maar sommige tellen aflopend, en bij bepaalde timers is dit zelfs instelbaar. Hieronder volgt een overzicht van de verschillende timers:

Table 7. Timer feature comparison

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary outputs
Advanced control	TIM1	16-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	3
General purpose	TIM2	32-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	-
	TIM3	16-bit	Up, down, up/down	integer from 1 to 65536	Yes	4	-
	TIM14	16-bit	Up	integer from 1 to 65536	No	1	-
	TIM15	16-bit	Up	integer from 1 to 65536	Yes	2	1
	TIM16 TIM17	16-bit	Up	integer from 1 to 65536	Yes	1	1
Basic	TIM6 TIM7	16-bit	Up	integer from 1 to 65536	Yes	-	-

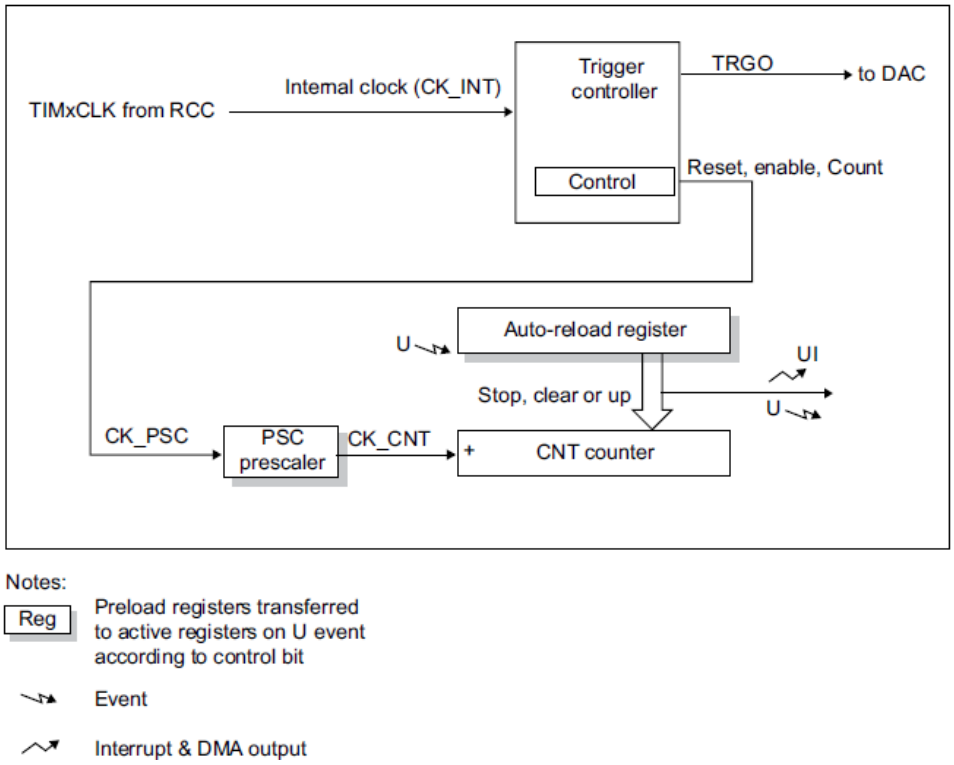
Figuur 51: overzicht timers (bron: datasheet STM32F091RC)

Daarnaast zijn er drie timers die gerelateerd zijn aan het systeem: de **SysTick**-timer, de **Independent Watchdog (IWDG)** en de **System Window Watchdog (WWDG)**.

Meer info over al deze timers vind je terug in de datasheet p. 21 en voor de specifieke instellingen in de reference manual datasheet vanaf p. 303.
Een aantal van deze timers zal verder aan bod komen in deze cursus.

10.2 Werking

We nemen als voorbeeld de eenvoudigste timers: Timer 6 en Timer 7. Deze hebben niet veel instellingen en hebben een gelijkaardige opbouw. Het zijn **16-bit auto-reload** timers.
Het principeschema is hieronder te vinden:



Figuur 52: blokschema Timer 6/7 (bron: RM0091)

Vragen:

- Zoek de werking uit hoe een Update Interrupt (UI) gegenereerd kan worden vertrekkende van TIMxCLK?

10.3 Toepassingen

10.3.1 Timer met interrupt

Hieronder staat de code van Timer 6 om een interrupt te genereren om de 125 ms. Die interrupt wordt gebruikt om LED7 en LED8 te laten knipperen.

```
InitTimer6();

// Oneindige lus starten.
while (1)
{
}

void InitTimer6(void)
{
  RCC->APB1ENR |= RCC_APB1ENR_TIM6EN; // clock voorzien voor de timer

  TIM6->PSC = 47999;                  // Als je een klok wil op 1000 Hz:
                                      // 48 000 000 / (47 999 + 1) = 1000 Hz
  TIM6->ARR = 125;                    // Als je iedere 125 ms een interrupt wil...
                                      // 125 stappen van 1 ms.

  TIM6->DIER |= TIM_DIER_UIE;         // Interrupt enable voor timer 6
  TIM6->CR1 |= TIM_CR1_CEN;           // counter enable

  //TIM6->CR1 |= TIM_CR1_OPM;         // one pulse mode kiezen,
                                      // dan stopt de timer na één overflow.

  NVIC_SetPriority(TIM6_IRQn, 0);
  NVIC_EnableIRQ(TIM6_IRQn);
}

// Interrupt van Timer 6 opvangen.
void TIM6_IRQHandler(void)
{
  if((TIM6->SR & TIM_SR_UIF) == TIM_SR_UIF)
  {
    // Interruptvlag resetten
    TIM6->SR &= ~TIM_SR_UIF;

    ToggleLed(7); // TODO: vervangen door variabelen
    ToggleLed(8);
  }
}
```

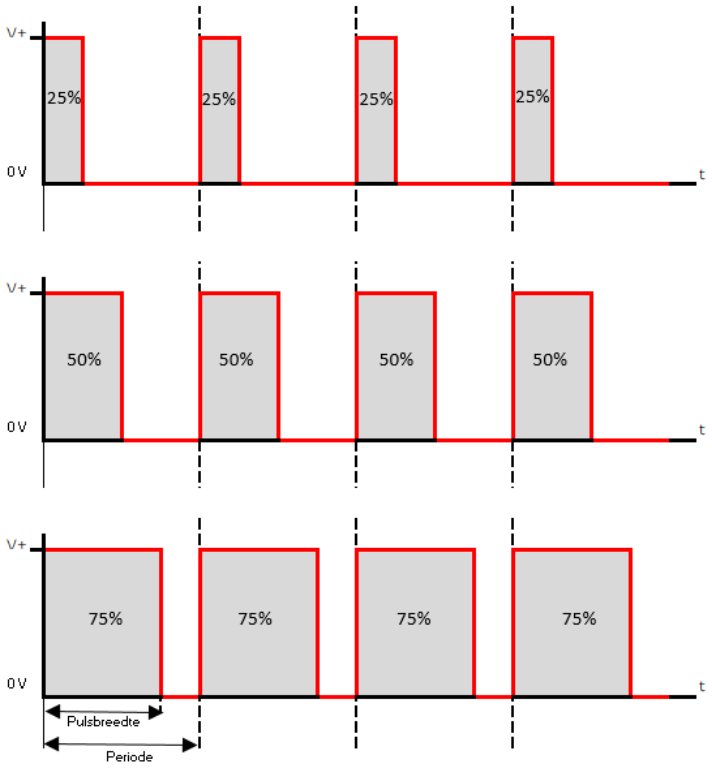
Figuur 53: instellen Timer 6 met interrupt handler

Vragen:

- Ga de betekenis van de code na door de reference manual datasheet te raadplegen.

10.3.2 Timer met PWM

PWM (Pulse Width Modulation) is een techniek waarbij een digitaal signaal wordt gebruikt om een analoog effect te simuleren. Dit gebeurt door snel een signaal aan en uit te schakelen, waarbij de verhouding tussen de aan-tijd (pulsbreedte) en de totale periode wordt aangepast. Deze verhouding wordt de **duty cycle** of **duty ratio** genoemd en wordt meestal in procenten uitgedrukt.



Figuur 54: voorstelling PWM duty cycles

Een PWM-signaal kan gebruikt worden om een LED of een motor aan te sturen. Hoe hoger de duty cycle, hoe feller de LED zal branden of hoe sneller de motor zal draaien.

Daarnaast kan PWM ook gebruikt worden om een modelbouwservomotor aan te sturen en de positie te regelen.
De totale periodetijd van het PWM-signaal moet ongeveer 20 ms zijn.
Een puls van ongeveer 1 ms zet de servo in de minimumpositie, terwijl een puls van ongeveer 2 ms de servo in de maximumpositie plaatst. Een puls tussen 1 ms en 2 ms stelt de servo in een tussenliggende positie.

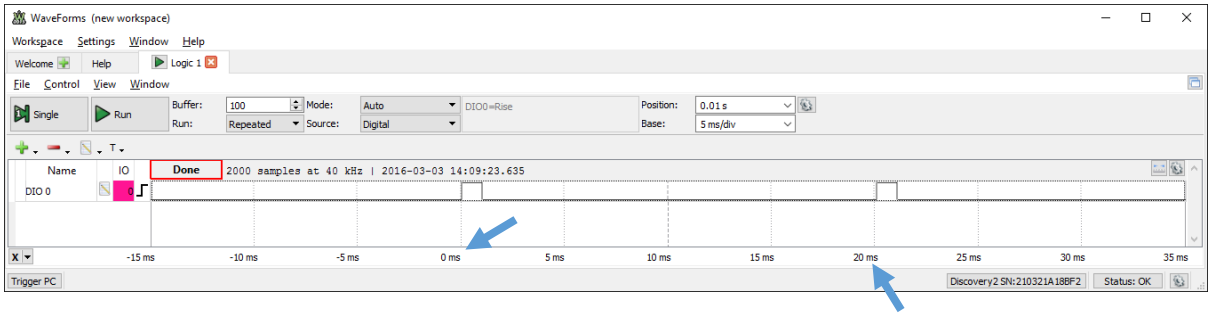
Figuur 55: modelbouwservomotor

Een dergelijk PWM-signaal kan perfect worden gemaakt met een timer-module. Op Toledo is onder de map *Losse bestanden* een **codevoorbeeld** te vinden: *main_met_pwm.c*, *pwm.h* en *pwm.c*.

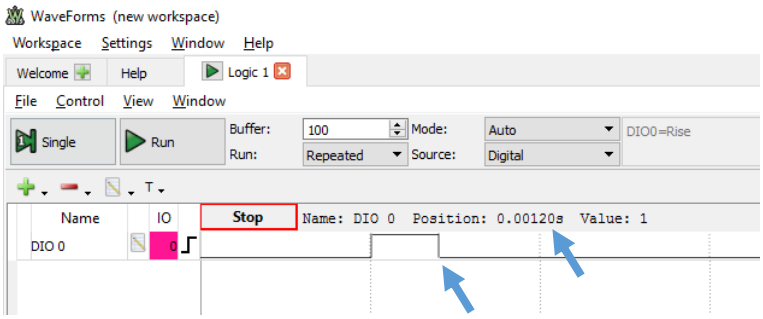
In het codevoorbeeld gebruiken we de knoppen SW1, SW2 en SW3, waarbij de duty cycle per knop verandert. Hierdoor verandert de positie van de servomotor en de lichtintensiteit van LED6.
De servomotor (op PA9) en LED6 (op PA8) worden via PWM aangestuurd met Timer 1, aangezien Timer 6 geen PWM-signalen kan genereren.
Om deze functionaliteit te gebruiken, moeten de pinnen, die standaard als GPIO geconfigureerd zijn, ook worden ingesteld als PWM-pinnen via *alternate functions* (zie later). De PWM-pinnen zijn: TIM1_CH1 (PA8) en TIM1_CH2 (PA9).

Bekijk zelf eens in detail het codevoorbeeld …

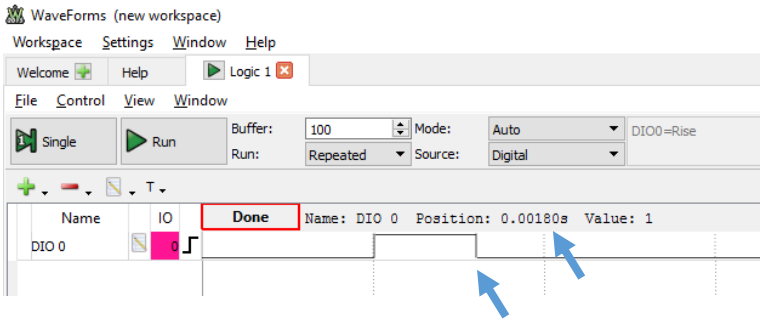
Als we van het PWM-signaal oscilloscoopbeelden maken, krijgen we onderstaande resultaten:



Figuur 56: scoopbeeld PWM-signaal met periodetijd 20 ms



Figuur 57: scoopbeeld PWM-signaal met een aan-tijd van 1,2 ms



Figuur 58: scoopbeeld PWM-signaal met een aan-tijd van 1,8 ms

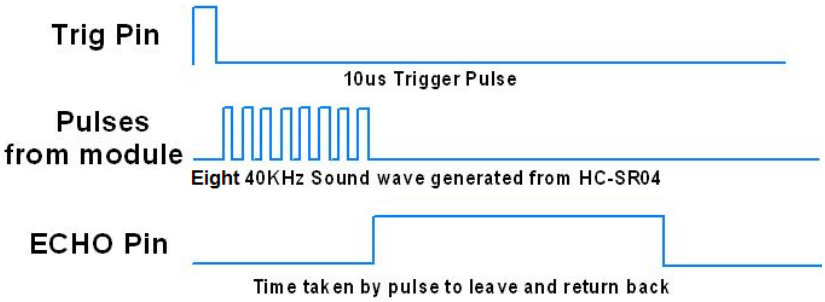
10.3.3 Timer met capture

Met capture kunnen we nauwkeurige tijdmetingen uitvoeren zoals het bepalen van de tijd tussen twee signalen.

Als voorbeeld nemen we hier de HC-SR04, een module met een ultrasone afstandssensor die de afstand tot een object meet met geluidsgolven. De module bestaat uit een zender (transmitter) en een ontvanger (receiver).

Figuur 59: HC-SR04-afstandssensor

Deze module sturen we aan door een puls van minimaal 10 µs naar de trigger-pin te sturen via een GPIO output pin op de microcontroller. De sensor zendt vervolgens een ultrasone geluidspuls van 40 kHz uit, waarna de echo-pin hoog wordt gezet. De sensor wacht op de weerkaatste geluidspuls. Zodra de geluidspuls is ontvangen, wordt de echo-pin weer laag gezet.
We meten de tijd tussen de opgaande en neergaande flank van de echo-pin, dit gebeurt via een inputpin op de microcontroller.
Uit deze tijd kunnen we dan de afstand berekenen op basis van de geluidssnelheid.



Figuur 60: tijdsdiagram trigger en echo

Vragen:

- De berekende afstand moeten we uiteindelijk door 2 delen. Waarom?
- Er zijn verschillende manieren om de tijd van de echopuls te meten via de inputpin van onze microcontroller.
  Een eerste manier is via een GPIO input pin. Hoe zou je, met de kennis die je tot nu toe hebt, dit kunnen realiseren?

Een tweede manier is door gebruik te maken van Input Capture, hiermee kunnen we nog iets vlotter de tijd opmeten.
Op Toledo is onder de map *Losse bestanden* een **codevoorbeeld** te vinden: *main_met_capture.c*, *capture.h* en *capture.c*. De puls naar trigger wordt verstuurd via GPIO output pin PC2 en de echo wordt ingelezen via pin PC8. Deze pin is ingesteld als Input Capture (TIM3_CH3) via *alternate functions* (zie later) en is gekoppeld aan Timer 3, aangezien Timer 6 deze functionaliteit niet heeft.

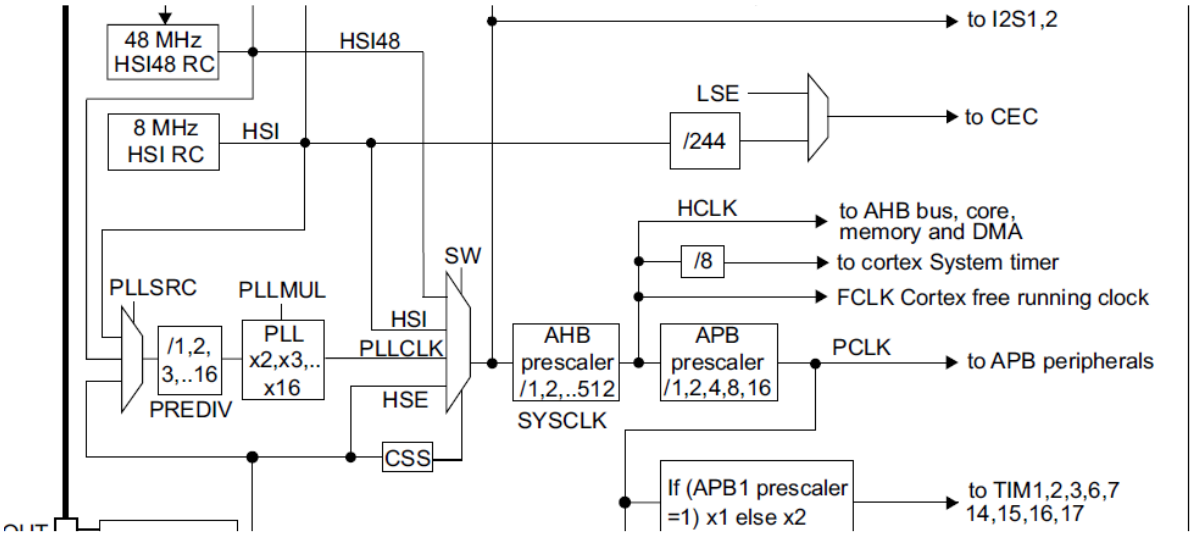
Bekijk zelf eens in detail het codevoorbeeld …

11 SysTick

11.1 Beschijving

De SysTick-timer is een timer die veelvuldig wordt gebruikt in RTOS-systemen (Real-Time Operating System). We kunnen instellen na hoeveel tijd er een 'tick' wordt gegenereerd, waardoor het RTOS kan schakelen tussen verschillende taken.
De SysTick-timer kan echter ook handig worden gebruikt wanneer er geen RTOS draait, maar een 'gewoon' stuk embedded software.

De SysTick-timer is een 24-bit afteller die wordt aangestuurd op het ritme van HCLK of HCLK/8 (afhankelijk van het type processor). Er wordt een interrupt gegenereerd wanneer de waarde 0 wordt bereikt.



Figuur 61: gedeelte van de clock tree (bron: RM0091)

11.2 Instellingen

We stellen de functie *SystemClock_Config* zo in dat er elke 20 ms een tick wordt gegenereerd:

```
SystemCoreClockUpdate();  // Nieuwe waarde van de core frequentie opslaan
                          // in SystemCoreClock variabele.

SysTick_Config(960000);   // Interrupt genereren. Zie core_cm0.h, om na iedere
                          // 20 ms een interrupt te hebben op SysTick_Handler().
                          // 48 000 000 / 960 000 = 50 keer per seconde => 20 ms.
                          // Let op: SysTick is 24-bit breed, dus maximumwaarde
                          // is 16 777 215.
```

Figuur 62: codefragment instellen van een tick

In de interrupt handler kunnen we dan bijvoorbeeld een LED doen knipperen:

```
// Handler die iedere 20 ms afloopt.
// Ingesteld met SystemCoreClockUpdate() en SysTick_Config().
void SysTick_Handler(void)
{
  ToggleLed(1);
}
```

Figuur 63: SysTick interrupt handler

11.3 Toepassingen

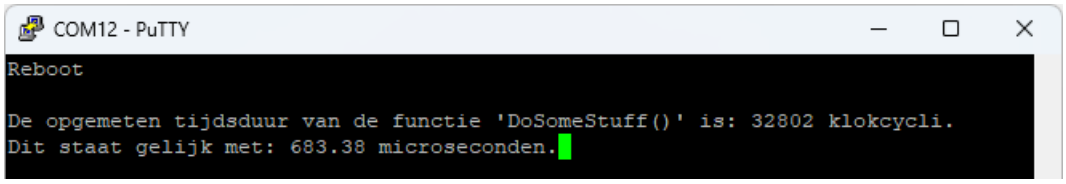
11.3.1 Tijd meten

In sommige gevallen, bijvoorbeeld in regeltechnische systemen, is het van groot belang om rekening te houden met de tijdsduur die een bepaalde actie in beslag neemt. Een SysTick-timer kan daarbij zeer handig zijn.
Het onderstaande voorbeeld meet met behulp van de SysTick-timer hoe lang de functie *DoSomeStuff()* duurt. Het resultaat wordt weergegeven op de seriële poort.

```
// Initialisatie

// SysTick uitschakelen
SysTick->CTRL = 0;

// 24-bit 'reload' getal op maximum zetten zodat we
// aftellen van maximumwaarde na bereiken van 0.
SysTick->LOAD = 0xFFFFFF;

// Huidige waarde op nul zetten zodat straks onmiddellijk
// gereset wordt op 0xFFFFFF.
SysTick->VAL = 0;

// HCLK selecteren en SysTick Timer enable (geen interrupt toelaten).
SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk | SysTick_CTRL_ENABLE_Msk;

// Wachten om herladen van de teller (met 0xFFFFFF)
while (SysTick->VAL == 0);
```

```
// ...
// Opmeten

// Begintijd vaststellen
startTijd = SysTick->VAL;

// Beetje uitrusten
DoSomeStuff();

// Eindtijd vaststellen
eindTijd = SysTick->VAL;

// Controleren of er geen underflow/roll over geweest is (onder nul gezakt).
if ((SysTick->CTRL & 0x10000) == 0)
{
    tijdsduur = startTijd - eindTijd;
    tijdsduurMs = (float)tijdsduur / 48;
    sprintf(info, "\r\nDe opgemeten tijdsduur van 'DoSomeStuff()' is: %d klokcycli.", tijdsduur);
    StringToUsart2(info);
    sprintf(info, "\r\nDit staat gelijk met: %.2f microseconden.", tijdsduurMs);
    StringToUsart2(info);
}
else
{
    tijdsduur = 0;
    sprintf(info, "\r\nDe opgemeten tijdsduur van 'DoSomeStuff()' is onbekend (SysTick underflow).");
    StringToUsart2(info);
}
```

Figuur 64: codefragment opmeten uitvoeringstijd DoSomeStuff() met de SysTick-timer



Figuur 65: voorbeelduitvoer

11.3.2 Delay routine

In de functie *WaitForMs()* van de Nucleo template maken we gebruik van de ticks van de SysTick timer. Dit is al iets nauwkeuriger dan wat we hebben gezien in Figuur 25 (lus met grote tellerwaarde).

```
// Handler die iedere 1 ms afloopt. Ingesteld met SystemCoreClockUpdate() en SysTick_Config(48000).
void SysTick_Handler(void)
{
  ticks++;
}

// Wachtfunctie via de SysTick.
void WaitForMs(uint32_t timespan)
{
  uint32_t startTime = ticks;

  while(ticks < startTime + timespan);
}
```

Figuur 66: functie WaitForMs()

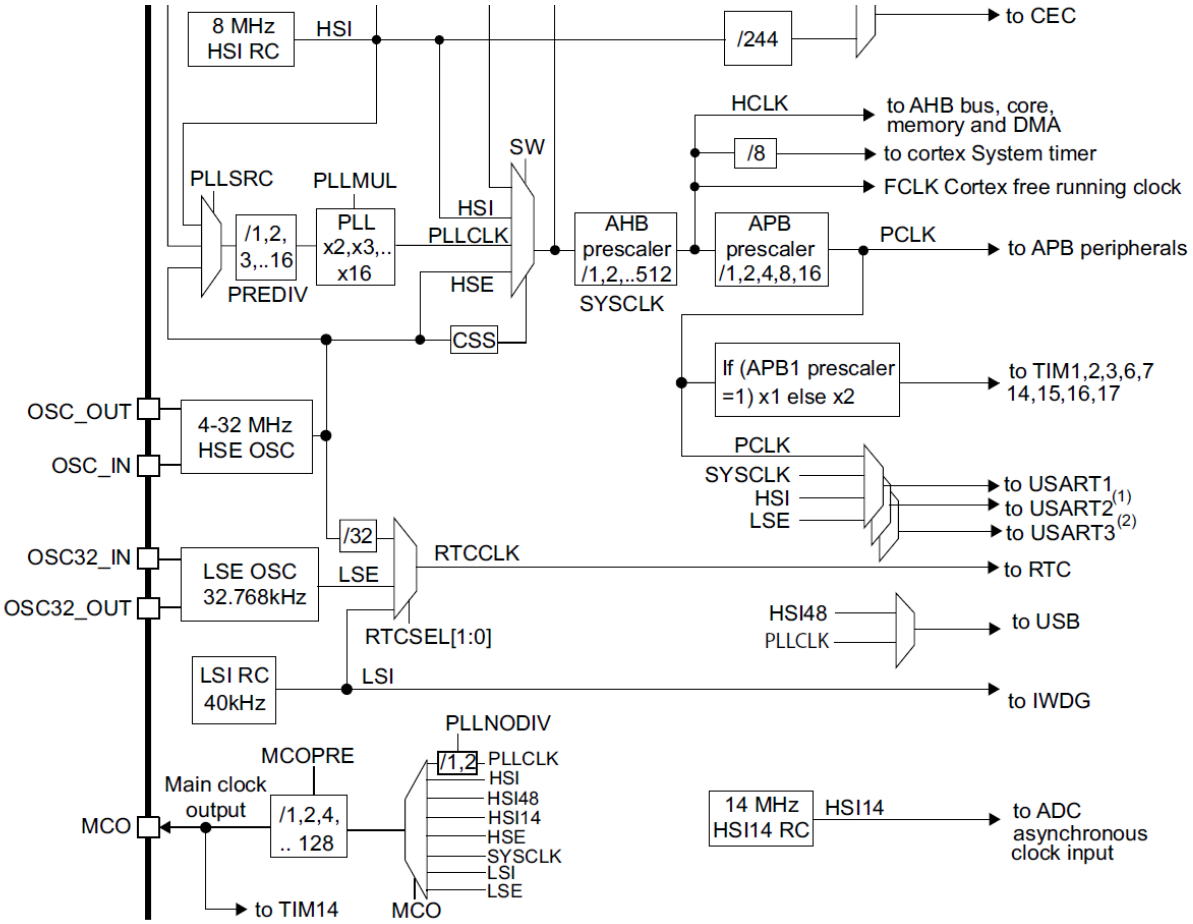
12 Watchdog timer

12.1 Beschrijving

De meeste microcontrollers worden ingezet bij toepassingen die weinig of zelfs geen user interface hebben. Daarenboven staan ze dicht bij de hardware en zijn ze essentieel in een groter systeem. Om vastgelopen software van een microcontroller automatisch te resetten kunnen we gebruik maken van een watchdog timer.

Een watchdog is een soort 'waakhond'. Als de timer verbonden met de watchdog, niet tijdig gereset wordt, gaat die timer zelf het initiatief nemen om heel de microcontroller te resetten in de hoop dat de herstart het systeem weer 'op de rails' krijgt.

Ook de STM32F091RC heeft zo'n watchdog timer, de Independant WatchDoG (IWDG), die gevoed wordt met een aparte oscillator op 40 kHz.



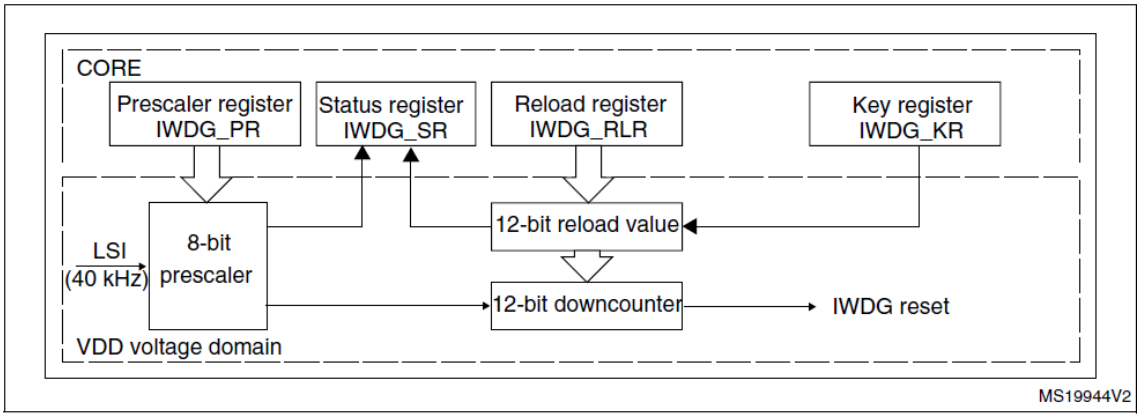
Figuur 67: gedeelte van de clock tree (bron: RM0091)

Vragen:

- Duid de juiste locatie aan van de oscillator tot de watchdog timer.

Het voordeel van een aparte klok is dat een foute klokinstelling voor de core, geen invloed heeft op de watchdog timer.

Het blokschema van de watchdog timer:



The watchdog function is implemented in the CORE voltage domain that is still functional in Stop and Standby modes.

Figuur 68: independant watchdog blokschema (bron: RM0091)

De watchdog kan worden gestart/geactiveerd door 0x0000CCCC naar het Key-register te schrijven. Om vervolgens toegang te krijgen tot het Reload-register en het Prescaler-register, moet 0x00005555 naar het Key-register worden geschreven.
De waarde van het Reload-register wordt pas in de 12-bit downcounter geladen wanneer 0x0000AAAA naar het Key-register wordt geschreven.
Als dit regelmatig in de code wordt uitgevoerd, wordt voorkomen dat de 12-bit downcounter nul bereikt en de microcontroller wordt gereset.

12.2 Instellingen

Als de watchdog de microcontroller moet resetten na iets meer dan 6 seconden, kunnen we de volgende code gebruiken:

```
void InitWatchdog(void)
{
  IWDG->KR = 0x0000CCCC;
  IWDG->KR = 0x00005555;
  IWDG->PR = IWDG_PR_PR;
  IWDG->RLR = 1000;
  while(IWDG->SR != 0);
  IWDG->KR = 0x0000AAAA;
}

void ResetWatchdog(void)
{
  IWDG->KR = 0x0000AAAA;
}
```

Figuur 69: codefragment watchdog timer

Vragen:

- Zoek zelf uit wat al die code exact doet op basis van de reference manual datasheet.
- Kan je dit voorbeeld verwerken in een werkend programma (bv. in combinatie met SW1)?

13 Alternate functions

13.1 Beschrijving

Alle I/O pinnen gedragen zich in principe als één van de onderstaande mogelijkheden die ingesteld worden via het **GPIOx_MODER** register (pin mode register).

MODERy[1:0]: Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Figuur 70: pinconfiguratie bits in het mode register (bron: RM0091)

Eerder in deze cursus hebben we reeds kennis kunnen maken met digitale in- en uitgangen en analoge ingangen. Het enige wat nog rest is een pin een alternatieve functie geven.

Neem er de **datasheet** van de STM32F091RC bij en zoek in tabel 13 (p. 40) het volgende op voor pin 8 van poort B (PB8):

A3	95	B3	61	A6	45	PB8	I/O	FTf		I2C1_SCL, CEC, TIM16_CH1, TSC_SYNC, CAN_RX	-

Figuur 71: pinnen (bron: datasheet STM32F091RC)

Daarop zien we dat de I/O geklasseerd is als 5V-tolerante pin (FT). Daarnaast kan die pin zich ook nog op een 'alternatieve manier' gedragen namelijk:

- Als klok van I²C-bus 1 (I2C1_SCL)
- Als pin voor de CEC-communicatie (CEC)
- Als pin gekoppeld met channel 1 van timer 16 (TIM16_CH1)
- Als Touch Sensing Controller pin (TSC_SYNC)
- Als CAN-bus receiver pin (CAN_RX)

13.2 Instellingen

Als we op een pin het gedrag willen instellen op de alternatieve functie, moeten we een aantal stappen doorlopen.
Stel na het voorzien van een klok voor de poort, eerst het *MODER* register in, daarna de *AFR* registers (Alternate Function Register).
Per poort (A, B, C, …) is er een *GPIOx_AFRL* en *GPIOx_AFRH* register (zie reference manual datasheet p. 169).
Deze registers worden in de CMSIS-code, voor bijvoorbeeld poort B, respectievelijk genoteerd als *GPIOB->AFR[0]* en *GPIOB->AFR[1]*.

Vragen:

- Duid op de volgende pagina aan waar de 4 bits zich bevinden die de alternate function instellingen bevatten voor PB8.

9.4.9 GPIO alternate function low register (GPIOx_AFRL) (x = A..F)

Address offset: 0x20

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFR7[3:0]				AFR6[3:0]				AFR5[3:0]				AFR4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFR3[3:0]				AFR2[3:0]				AFR1[3:0]				AFR0[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFRy[3:0]:** Alternate function selection for port x pin y (y = 0..7)
These bits are written by software to configure alternate function I/Os

AFRy selection:
0000: AF0
0001: AF1
0010: AF2
0011: AF3
0100: AF4
0101: AF5
0110: AF6
0111: AF7
1000: Reserved
1001: Reserved
1010: Reserved
1011: Reserved
1100: Reserved
1101: Reserved
1110: Reserved
1111: Reserved

9.4.10 GPIO alternate function high register (GPIOx_AFRH) (x = A..F)

Address offset: 0x24

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AFR15[3:0]				AFR14[3:0]				AFR13[3:0]				AFR12[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
AFR11[3:0]				AFR10[3:0]				AFR9[3:0]				AFR8[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **AFRy[3:0]:** Alternate function selection for port x pin y (y = 8..15)
These bits are written by software to configure alternate function I/Os

AFRy selection:
0000: AF0
0001: AF1
0010: AF2
0011: AF3
0100: AF4
0101: AF5
0110: AF6
0111: AF7
1000: Reserved
1001: Reserved
1010: Reserved
1011: Reserved
1100: Reserved
1101: Reserved
1110: Reserved
1111: Reserved

Figuur 72: alternate function registers (bron: RM0091)

In de alternate function registers komen dus 4 bits per pin. Met die 4 bits kunnen we een bepaalde functie selecteren op die pin. De beschrijving voor de alternate functions kan in de datasheet gevonden worden. Voor poort B wordt dit:

Table 15. Alternate functions selected through GPIOB_AFR registers for port B

Pin name	AF0	AF1	AF2	AF3	AF4	AF5
PB0	EVENTOUT	TIM3_CH3	TIM1_CH2N	TSC_G3_IO2	USART3_CK	-
PB1	TIM14_CH1	TIM3_CH4	TIM1_CH3N	TSC_G3_IO3	USART3_RTS	-
PB2	-	-	-	TSC_G3_IO4	-	-
PB3	SPI1_SCK, I2S1_CK	EVENTOUT	TIM2_CH2	TSC_G5_IO1	USART5_TX	-
PB4	SPI1_MISO, I2S1_MCK	TIM3_CH1	EVENTOUT	TSC_G5_IO2	USART5_RX	TIM17_BKIN
PB5	SPI1_MOSI, I2S1_SD	TIM3_CH2	TIM16_BKIN	I2C1_SMBA	USART5_CK_RTS	-
PB6	USART1_TX	I2C1_SCL	TIM16_CH1N	TSC_G5_IO3	-	-
PB7	USART1_RX	I2C1_SDA	TIM17_CH1N	TSC_G5_IO4	USART4_CTS	-
PB8	CEC	I2C1_SCL	TIM16_CH1	TSC_SYNC	CAN_RX	-
PB9	IR_OUT	I2C1_SDA	TIM17_CH1	EVENTOUT	CAN_TX	SPI2_NSS, I2S2_WS
PB10	CEC	I2C2_SCL	TIM2_CH3	TSC_SYNC	USART3_TX	SPI2_SCK, I2S2_CK
PB11	EVENTOUT	I2C2_SDA	TIM2_CH4	TSC_G6_IO1	USART3_RX	-
PB12	SPI2_NSS, I2S2_WS	EVENTOUT	TIM1_BKIN	TSC_G6_IO2	USART3_CK	TIM15_BKIN
PB13	SPI2_SCK, I2S2_CK	-	TIM1_CH1N	TSC_G6_IO3	USART3_CTS	I2C2_SCL
PB14	SPI2_MISO, I2S2_MCK	TIM15_CH1	TIM1_CH2N	TSC_G6_IO4	USART3_RTS	I2C2_SDA
PB15	SPI2_MOSI, I2S2_SD	TIM15_CH2	TIM1_CH3N	TIM15_CH1N	-	-

Figuur 73: alternate functions voor poort B (bron: datasheet STM32F091RC)

Vragen:

- Vind je de aanduiding waarbij je op pin 8 van poort B de I²C klok kan instellen?
- Welke waarde zal je nu moeten invullen op de vorige pagina in het alternate function register?

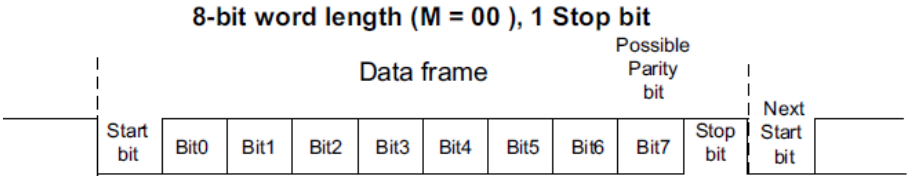
14 UART

14.1 Beschrijving

UART is de afkorting van Universal Asynchronous Receiver and Transmitter. Deze module kan gebruikt worden om full duplex te communiceren tussen twee devices zonder een kloksignaal (vandaar asynchroon).
UART is wellicht beter gekend onder de vorm van RS-232, RS-485, RS-422, … De UART voorziet de correcte bitstroom, andere IC's kunnen dan de spanningsniveaus aanpassen zodat we kunnen spreken van bijvoorbeeld RS-232.

De UART-module in de STM32F091RC is een 'speciaal geval'. We kunnen er naast asynchrone communicatie ook synchrone communicatie mee verzorgen. Dan moeten we wel een klok voorzien. Daarom spreekt men van USART (Universal Synchronous and Asynchronous Receiver and Transmitter). De synchrone versie valt echter buiten het bestek van deze cursus. We bekijken later wel nog andere synchrone bussen zoals I²C en SPI.

De UART verstuurt standaard een byte (8 bits) over één lijn. Eén zo'n byte/data frame ziet er zo uit:



Figuur 74: UART data frame (bron: RM0091)

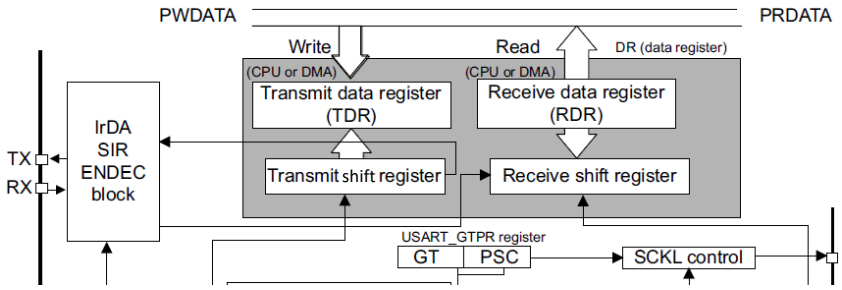
Om een byte te versturen hebben we standaard 10 'tijdsloten' van een bit nodig, er is namelijk bijkomend een start- en een stopbit nodig.
De start- en stopbit hebben als functie het synchroon laten lopen van de zender en ontvanger. Omdat we immers geen klok meesturen, moet er een mogelijkheid zijn voor de ontvanger om op een correcte manier de verzonden bits te ontvangen. Dat kan dus door gebruik te maken van twee principes:

1. Zender en ontvanger moeten op voorhand een **datatransfertsnelheid** afspreken. Hoe snel zal de data aankomen? Dat wordt typisch uitgedrukt in een aantal **baud** (genoemd naar Émile Baudot, een telegraaf-ingenieur, die leefde in Frankrijk in de tweede helft van de 19de eeuw).
   Baud drukt het **aantal symbolen per seconde** uit die we kunnen versturen. Een symbool kan één of meerdere bits bevatten. Op veel systemen, zoals de UART op onze microcontroller, is één symbool gelijk aan één bit. Enkele voorbeelden van baudsnelheden: 4800, 9600, 115200, …
2. Via een **start- en stopbit** kan de ontvanger het begin en einde van een ontvangen byte detecteren.

Opmerking: de lengte van het data frame kan afwijken van 10 bits wanneer er gekozen wordt voor één of meer van de volgende opties: het aantal databits wordt ingesteld op 7 of 9 in plaats van 8, er wordt een extra pariteitsbit toegevoegd voor foutdetectie, het aanpassen van de stopbitlengte.

Het Nucleo bordje beschikt over een on board ST-Link programmer die verbonden is met pinnen PA2 en PA3 van de STM32F091RC. Die pinnen kunnen gekoppeld worden met UART2 en dus via USB-

interface. Daarmee kunnen we eenvoudig via een virtuele COM-poort communiceren met de computer, natuurlijk dient hiervoor de ST-link driver geïnstalleerd te zijn.
Een gedeelte van het blokschema van de UART van de STM32F091RC:



Figuur 75: gedeeltelijk blokschema UART (bron: RM0091)

Het register TDR wordt gebruikt om data te verzenden, RDR om data te ontvangen.

14.2 Instellingen

Als we UART willen gebruiken moeten we verschillende stappen doorlopen:
1. De gebruikte poort van klok voorzien
2. De gebruikte pinnen op alternate function zetten
3. De alternate function kiezen
4. De UART initialiseren met de gewenste parameters (baud rate, aantal stopbits, …)
5. De transmitter en/of receiver activeren

```
// Clock voor GPIOA inschakelen.
RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

// Pinnen van de USART2 instellen als alternate function.
// PA2 => TX-pin
GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODER2) | GPIO_MODER_MODER2_1;
GPIOA->AFR[0] |= 0x00000100;

// PA3 => RX-pin
GPIOA->MODER = (GPIOA->MODER & ~GPIO_MODER_MODER3) | GPIO_MODER_MODER3_1;
GPIOA->AFR[0] |= 0x00001000;

// Usart module van een klok voorzien.
RCC->APB1ENR |= RCC_APB1ENR_USART2EN;

// Baudrate zetten.
USART2->BRR = 48000000/baudRate;

// USART inschakelen.
USART2->CR1 |= USART_CR1_UE;

// Transmitter inschakelen.
USART2->CR1 |= USART_CR1_TE;

// Receiver inschakelen.
USART2->CR1 |= USART_CR1_RE;
```

Figuur 76: codefragment UART2-instellingen

Zowel voor het versturen als voor het ontvangen van data via UART hebben we de keuze tussen het werken via:
1. Polling
2. Interrupt-gestuurd
3. DMA

Dit staat in volgorde van complexiteit. Zo is polling de eenvoudigste vorm met als nadeel dat de CPU dikwijls staat te wachten op feedback van de UART-module. Het is zelfs mogelijk dat er data verloren gaat! DMA is dan weer de meest efficiënte manier van werken, maar is relatief complex.

14.3 Toepassingen

14.3.1 Virtuele COM-poort via polling

Hieronder staat het stuk code dat data verzendt en ontvangt via UART2 d.m.v. polling. De data kunnen we bekijken en ingeven via een console-applicatie zoals Putty.

```
while (1)
{
  // Tekst verzenden.
  StringToUsart2("Hallo, ik ben de STM32F091RC!\r\n");

  // Eén byte ontvangen.
  if((USART2->ISR & USART_ISR_RXNE) == USART_ISR_RXNE)
  {
    uint8_t ontvangenByte = (uint8_t)USART2->RDR;
    ByteToLeds(ontvangenByte);
  }

  // Even wachten.
  WaitForMs(2000);
}
```

Figuur 77: communicatie met virtuele COM-poort via polling

Deze code is heel eenvoudig. Hét grote probleem echter is het ontvangen van data. De ontvangende microcontroller zal slechts één maal per lusdoorgang controleren of er een byte ontvangen is.
Is dat het geval, komt de waarde van die byte op de LED's. Als er meer dan één byte aangekomen is tijdens deze laatste lusdoorgang, zijn we die voorgaande bytes kwijt. Enkel de laatste byte blijft in het ontvangstregister (RDR) en kan opgevraagd worden.

14.3.2 Virtuele COM-poort via interrupt

De voorgaande code kan aangepast worden zodat het ontvangen via een interrupt gebeurt, in de initialisatie voegen we deze code toe:

```
// Interrupt voor de receiver inschakelen (receiver buffer not empty interrupt enable).
USART2->CR1 |= USART_CR1_RXNEIE;

// Koppeling van interrupt maken met de NVIC.
NVIC_SetPriority(USART2_IRQn, 0);
NVIC_EnableIRQ(USART2_IRQn);
```

Figuur 78: codefragment activeren interrupt voor ontvangst

En we verplaatsen de code, die instaat voor het ontvangen, naar de interrupt handler:

```
// Interrupt handler van USART2.
void USART2_IRQHandler(void)
{
  // Byte ontvangen?
  if((USART2->ISR & USART_ISR_RXNE) == USART_ISR_RXNE)
  {
    // Byte ontvangen, lees hem om alle vlaggen te wissen.
    uint8_t ontvangenByte = (uint8_t)USART2->RDR;
    ByteToLeds(ontvangenByte);
  }
}
```

Figuur 79: communicatie met virtuele COM-poort via interrupt

Door interrupt-gestuurd te werken, is de kans dat inkomende bytes aan data verloren wordt bijzonder gering geworden.

14.3.3 Varia

De microcontroller beschikt meestal niet over ingebouwde modules zoals GPS, Bluetooth of WiFi. Dergelijke modules worden vaak extern aangesloten en via UART aangestuurd door de microcontroller. Omdat deze modules doorgaans veel data verzenden en ontvangen, is een point-to-point full-duplex verbinding zoals UART geschikt om een maximale performantie te garanderen.

15 I²C

15.1 Beschrijving

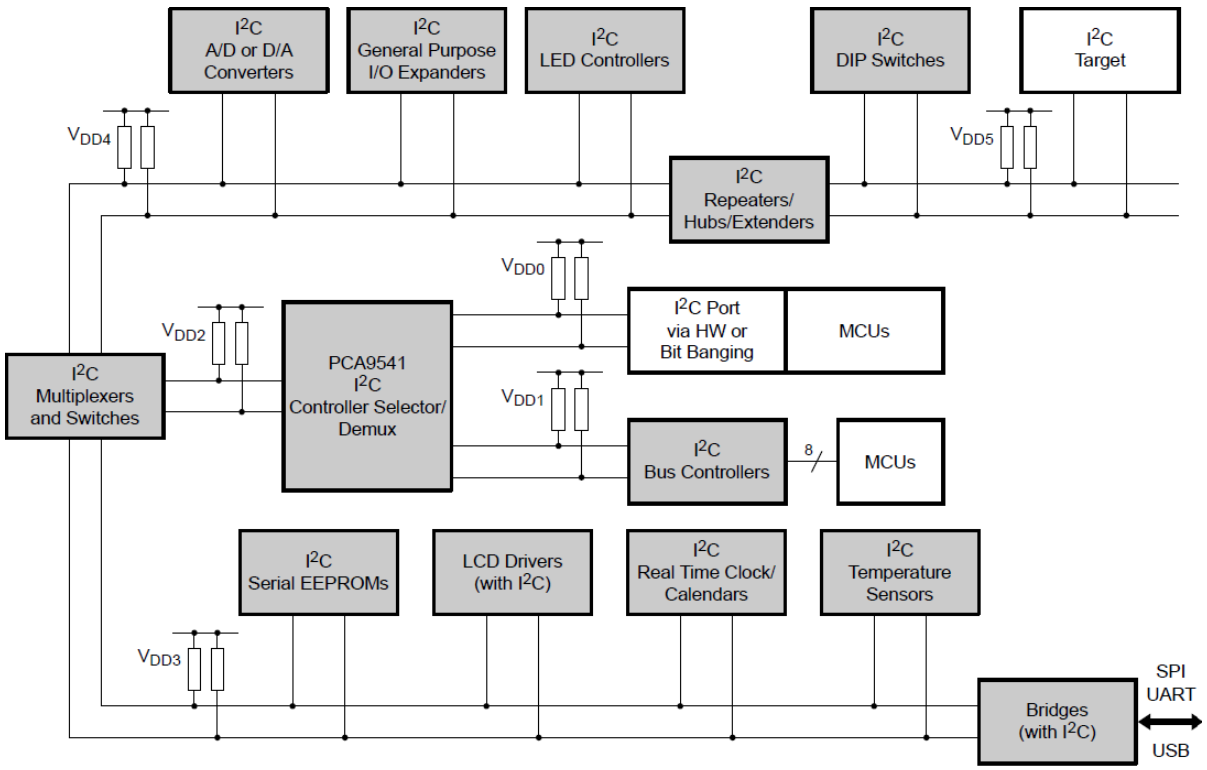
I²C is de afkorting van Inter Integrated Circuit. Het is een digitale **multi-master multi-slave synchrone communicatiebus** die half duplex werkt.
Ze kan gebruikt worden om te communiceren tussen twee IC's. Dat kan op één en hetzelfde printbord zijn, maar ook op een apart bord. Om het even in welke situatie is I²C niet geschikt voor grote afstanden (best < 1 m).

Om te communiceren zijn er twee draden nodig: **SDA** (Serial Data/Address) en **SCL** (Serial Clock). Deze signalen moeten via **pull-up** weerstanden naar de voedingspanning getrokken worden, veelgebruikte pull-up weerstandswaardes zijn 4k7 ohm.
Hierbij moet uiteraard rekening gehouden worden met de voedingspanning van alle devices en moeten alle devices een zelfde referentiespanning hebben (massa).

Vragen:
- Wat zou de reden zijn dat er met pull-up weerstanden gewerkt wordt?

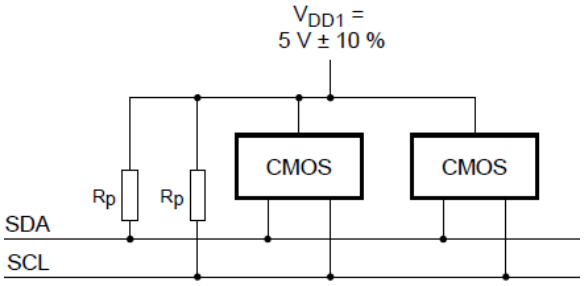
Het maximum aantal devices is beperkt door de adres- en maximale buscapaciteit (400 pF). De standaard gebruikte (laagste) klokfrequentie is 100 kHz.

Een typische opbouw van een I²C-systeem is hieronder te zien, het is een volledig overzicht van wat mogelijk is:



Figuur 80: I²C-bustopologie (bron: Philips I²C bus specifications)

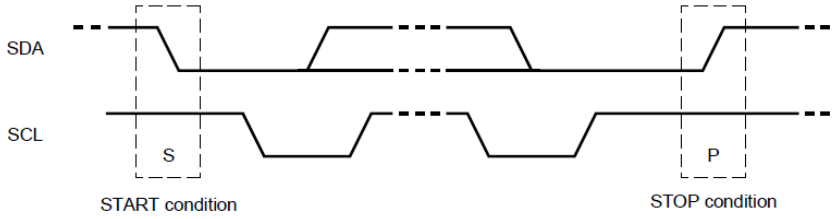
In de meeste gevallen kan het schema eenvoudiger, bv. als we maar 2 devices hebben met hun pull-up weerstanden:



Figuur 81: eenvoudige I²C-opstelling (bron: Philips I²C bus specifications)

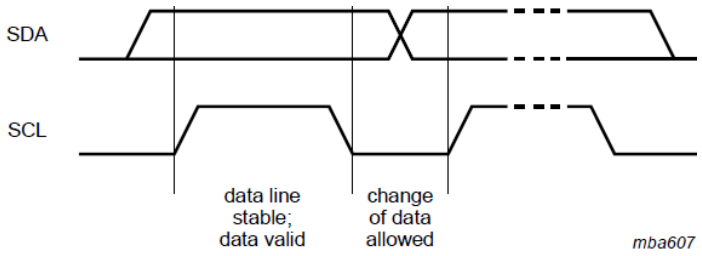
Eén device fungeert als **master**, de andere als **slave**. De master is vaak een microcontroller die gegevens nodig heeft van een bepaalde module, zoals een sensor (deze fungeert steeds als slave). Zoals eerder vermeld, is het in I²C mogelijk om meerdere masters op de bus aan te sluiten (bijvoorbeeld meerdere microcontrollers), maar tijdens de communicatie kan slechts één master actief zijn; de overige devices gedragen zich dan als slaves.

Omdat I²C een half duplex bus is, moeten er afspraken gemaakt worden over de manier van versturen van data. Eén van die afspraken is het werken met **start- en stopcondities** die door de master wordt gegeven. Ook zorgt de master voor het genereren van de **klok** op de bus:



Figuur 82: start- en stopcondities bij I²C (bron: Philips I²C bus specifications)

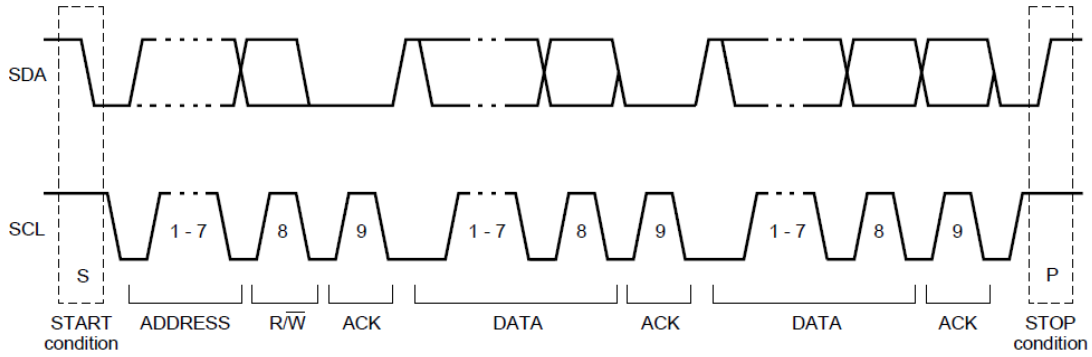
Daarmee samenhangend is er ook een afspraak rond de periode waarin data stabiel moet zijn of kan variëren:



Figuur 83: datacondities bij I²C (bron: Philips I²C bus specifications)

De I²C-module op onze microcontroller zal al deze timing voorwaarden voor zijn rekening nemen (eenmaal de correcte instellingen gedaan zijn).

Tussen de start- en stopconditie wordt dan de data verstuurd. Een volledig stuk I²C-communicatie kan er als volgt uitzien:



Figuur 84: voorbeeld I²C-communicatie (bron: Philips I²C bus specifications)

Na de startconditie stuurt de master altijd het adres van de slave (dit staat vermeld in de datasheet van de slave), gevolgd door een R/W-bit (read/write) op de SDA-lijn. Als er een slave gevonden is op de bus, antwoordt hij met een ACK (acknowledge).

Afhankelijk van het R/W-bit, dat de richting van de communicatie bepaalt, wordt de data ofwel van de slave gelezen (bit = 1), ofwel naar de slave geschreven (bit = 0). De ACK's worden daarbij gegeven door de master bij het lezen van data, en door de slave bij het schrijven.

15.2 Instellingen

Om pinnen als I²C-pinnen te gebruiken, moeten een aantal instellingen gedaan worden. Hieronder volgt een summier overzicht:

1. Pinnen instellen als alternate function.
2. I²C-module van klok voorzien.
3. Timing instellen (100 kHz, rise and fall times, …).
4. I²C-module inschakelen: de STM32F091RC heeft twee I²C-modules.

Bij het versturen van data (byte(s)), moeten nog andere instellingen gebeuren:

1. Slave address goed zetten.
2. Kiezen tussen lezen en schrijven.
3. Aantal te versturen/ontvangen bytes instellen.
4. Startconditie verzenden.
5. Byte klaarzetten (indien we willen verzenden).
6. Stopconditie verzenden.

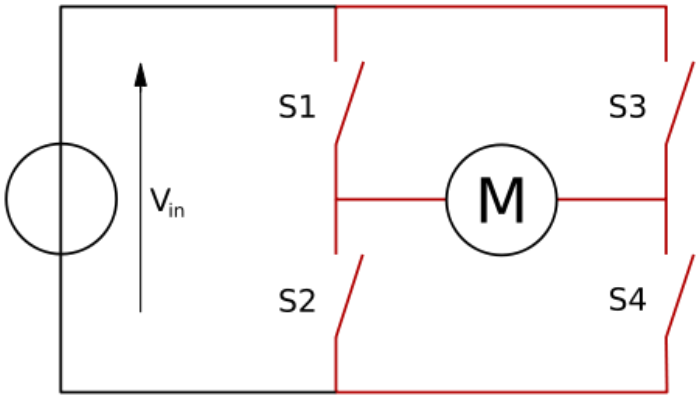
15.3 Toepassingen

15.3.1 H-brug

Als toepassingsvoorbeeld het aansturen van een DC-motor met een I²C H-brug. De H-brug kan, door zijn specifieke opstelling, de stroom door een DC-motor van richting laten veranderen. Hierdoor zal de motor zowel links als rechts kunnen draaien.

Vragen:

- Kan je verklaren hoe het links en rechts draaien werkt?



Figuur 85: principe van een H-brug (bron: wikipedia.org)

De gebruikte H-brug kan werken op spanningen tot 16 V en stromen verwerken tot 10 A. Ze verwacht via I²C twee bytes aan data: direction en speed. Het adres van de module is 0x41 (karakter 'A').

Een mogelijke initialisatie van de I²C-module verloopt als volgt:

```
// Pinnen van de I²C1 instellen als alternate function.
// PB8 => SCL-pin (5V tolerant)
GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODER8) | GPIO_MODER_MODER8_1;
GPIOB->AFR[1] |= 0x00000001;

// PB9 => SDA-pin (5V tolerant)
GPIOB->MODER = (GPIOB->MODER & ~GPIO_MODER_MODER9) | GPIO_MODER_MODER9_1;
GPIOB->AFR[1] |= 0x00000010;

// I²C1-module van een klok voorzien
RCC->APB1ENR |= RCC_APB1ENR_I2C1EN;

// Timing correct instellen (via AN4235 en Excel bestand van ST)
// 100kHz vertrekkende vanuit 8MHz (HSI)
I2C1->TIMINGR = 0x00201D2B;

// I²C1-module enable
I2C1->CR1 |= I2C_CR1_PE;
```

Figuur 86: initialisatie I²C module 1

Het versturen van een nieuwe waarde kan dan als volgt:

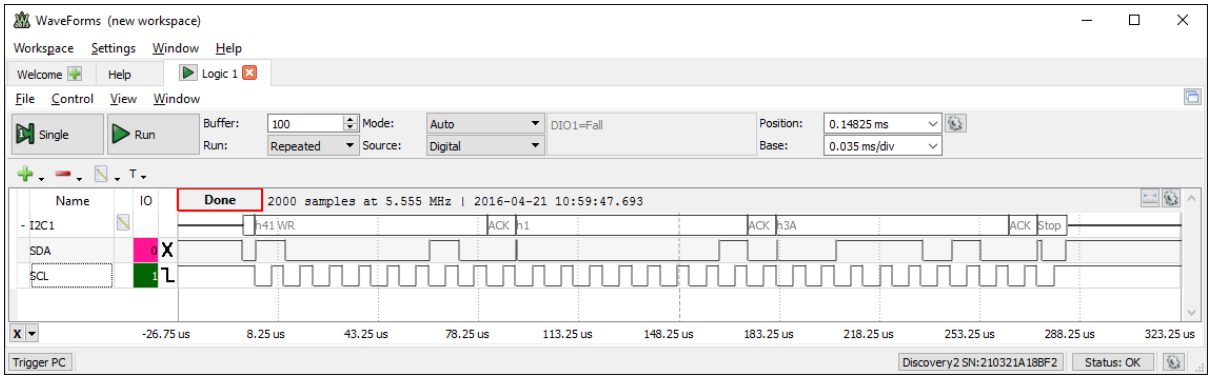
```
// Nieuwe waarde versturen via I²C.
// Schrijfactie.
I2C1->CR2 &= ~I2C_CR2_RD_WRN;

// Slave address waarmee gecommuniceerd moet worden invullen.
I2C1->CR2 &= 0xFFFFFF00;                        // slave address resetten
I2C1->CR2 |= (SLAVE_ADDRESS << 1);              // adres klaarzetten

// Aantal te versturen bytes, waarna een stop gegenereerd moet worden (TC vlag wordt gezet).
I2C1->CR2 &= ~I2C_CR2_NBYTES;                   // NBYTES wissen
I2C1->CR2 |= (2 << 16);                         // 2 bytes te versturen, dus 2 in NBYTES plaatsen.

// Start conditie.
I2C1->CR2 |= I2C_CR2_START;

// Bytes versturen.
while((I2C1->ISR & I2C_ISR_TXE) != I2C_ISR_TXE);  // Is verzendbuffer leeg?
I2C1->TXDR = direction;                           // direction versturen
while((I2C1->ISR & I2C_ISR_TXE) != I2C_ISR_TXE);  // Is verzendbuffer leeg?
I2C1->TXDR = speed;                               // speed versturen

// Stop conditie.
while((I2C1->ISR & I2C_ISR_TC) != I2C_ISR_TC);    // Wacht op transmission complete.
I2C1->CR2 |= I2C_CR2_STOP;
```

Figuur 87: codefragment data versturen via I²C

Vragen:

- Kan je zelf een doordachte signatuur bedenken voor een functie die data moet kunnen versturen op deze I²C-bus?

De communicatie kan ook gemonitord worden met een oscilloscoop:



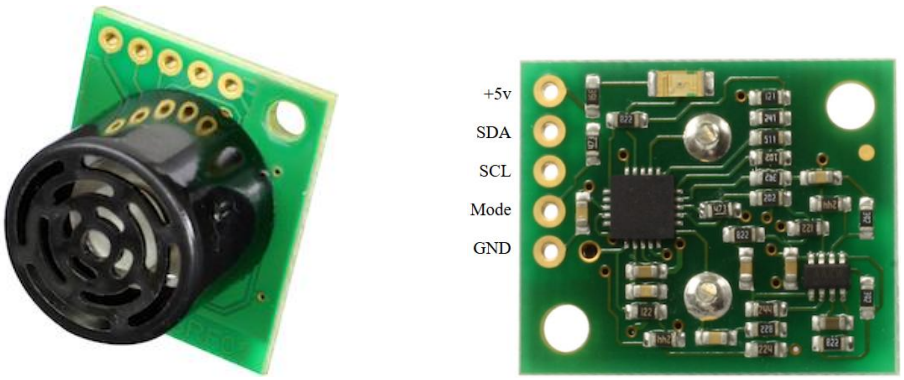
Figuur 88: scoopbeeld I²C-communicatie H-brug

Vragen:

- Wat is er in bovenstaand voorbeeld verstuurd geweest? Wat verwacht je dat de H-brug doet?

15.3.2 Ultrasone sensor (SRF-02)

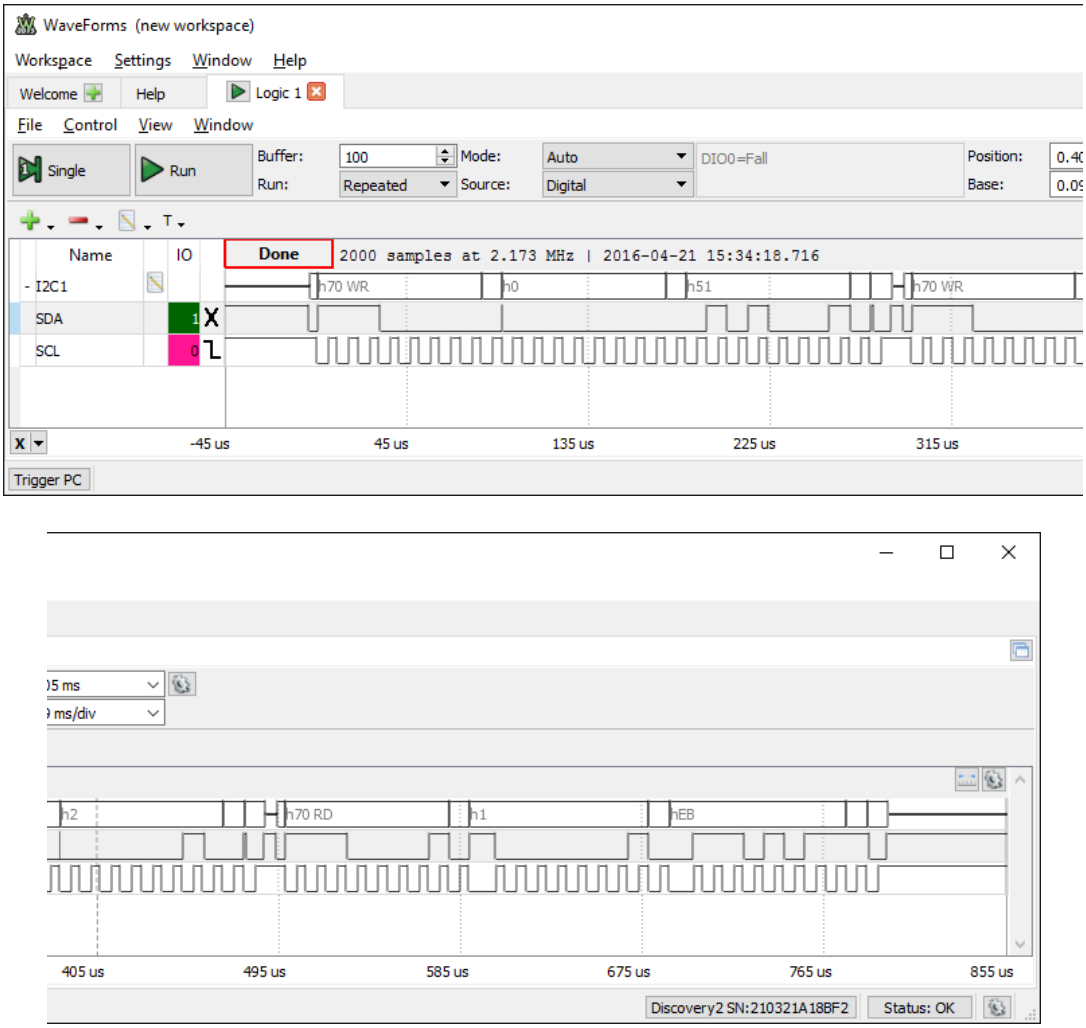
De SRF-02 module is een ultrasone sensor dat aangestuurd wordt via I²C.



Figuur 89: http://www.robot-electronics.co.uk/htm/srf02techI2C.htm

Vragen:

- Bekijk het oscilloscoopbeeld en zoek uit op welke afstand de sensor iets 'gezien' heeft.



Figuur 90: scoopbeeld I²C-communicatie met SRF-02

16 SPI

16.1 Beschrijving

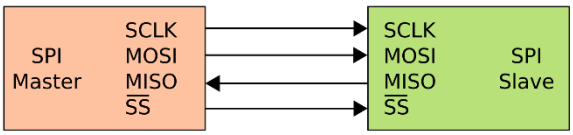
SPI is de afkorting van Serial Peripheral Interface. Het is een digitale **single-master multi-slave synchrone communicatiebus** die full duplex werkt.
De SPI-bus wordt heel dikwijls gebruikt om op eenzelfde print (PCB) verschillende IC's met elkaar te koppelen. Zo kunnen we onze microcontroller koppelen met een IO-expander, een sensor, een actuator, een LCD display, een geheugenmodule, …

Bij SPI kan de kloksnelheid (en dus ook de datatransfertsnelheid) ingesteld worden. Hierbij is de maximumgrens onder andere bepaald door de slew rate van alle componenten op de bus, parasitaire capaciteiten, mogelijkheid tot instralen en opnemen van storingen, …
Kloksnelheden van enkele megahertzen zijn niet ongewoon.

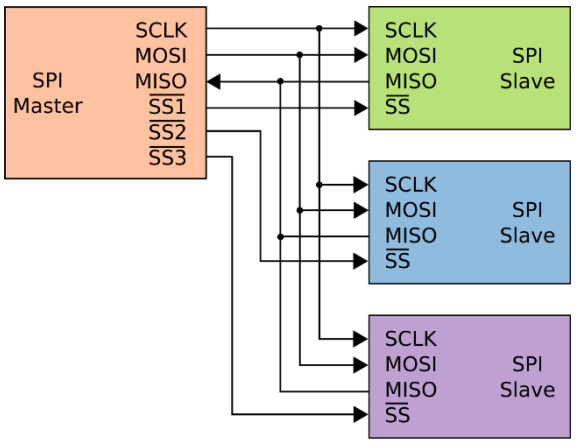
Om met de SPI-bus te werken worden meestal 2 tot 4 pinnen ingesteld:
1. **SCK**: serial clock, bepaalt de kloksnelheid.
2. **MOSI**: Master Output Slave Input, data uitgang van de master en ingang van de slave.
3. **MISO**: Master Input Slave Output, data ingang van de master en uitgang van de slave.
4. $\overline{\mathbf{SS}}$: Slave Select, pin die actief laag gemaakt moet worden om de slave te activeren.

Pinnen SCK en MOSI zijn essentieel, MISO en $\overline{\text{SS}}$ zijn optioneel.

Er zijn verschillende manieren om een SPI-bus op te bouwen. Het kan zijn dat we slechts twee modules willen koppelen, maar ook het koppelen van meer dan twee modules is mogelijk:

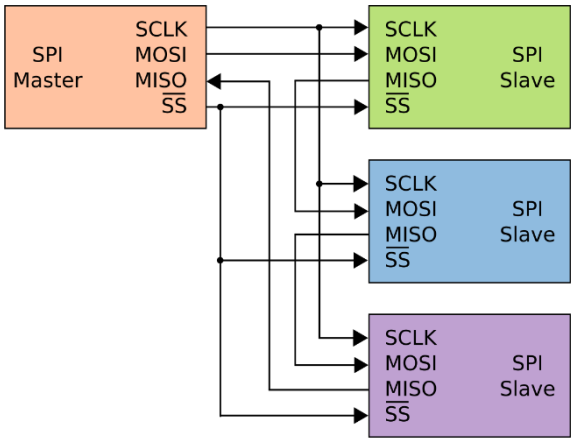


Figuur 91: single-master single-slave configuratie (bron: wikipedia.org)



Figuur 92: single-master multi-slave configuratie (bron: wikipedia.org)

Bij SPI werken we echter niet met slave-adressen en ACK's, wat het protocol eenvoudiger maakt. Een slave wordt door de master geactiveerd door zijn SS-lijn actief laag te maken.
De standaardconfiguratie heeft wel als nadeel dat er per slave een aparte SS-lijn voorzien moet worden.
Maar er bestaat echter ook een zogenaamde daisy chain-configuratie, waarbij slechts één SS-lijn nodig is. De MOSI van de master is verbonden met de eerste slave, waarvan de MISO vervolgens wordt gekoppeld aan de MOSI van de volgende slave. De MISO van de laatste slave is verbonden met de master. Zo creëren we als het ware een lus.
Het nadeel van deze configuratie is dat de software complexer wordt, omdat de data zodanig gestructureerd moet worden dat elke slave zijn eigen commando ontvangt op basis van zijn positie in de keten.



Figuur 93: single master/multiple slave via daisy chaining (bron: wikipedia.org)

16.2 Instellingen

Om met de SPI-module te kunnen werken in de STM32F091RC moeten volgende stappen doorlopen worden:
1. Gebruikte poorten van klok voorzien.
2. Gebruikte SPI-module van klok voorzien.
3. Alternate functions instellen voor de SPI-pinnen.
4. Gewenste kloksnelheid instellen.
5. Software slave management inschakelen indien we geen slave select pin gebruikt.
6. Master mode instellen.
7. FIFO reception threshold op ¼ te zetten (8-bit trigger).
8. SPI inschakelen.

Optioneel zijn:
1. Klokpolarisatie instellen.
2. Klokfase instellen.
3. Bidirectionele mode instellen.
4. LSB of MSB first instellen.
5. Internal slave select instellen.
6. Aantal databits instellen.
7. …

```
void InitSpi1(void)
{
    // Poort A van klok voorzien (mocht dit nog niet gebeurd zijn).
    RCC->AHBENR = RCC->AHBENR | RCC_AHBENR_GPIOAEN;

    // RCC_APBxENR registers instellen zodat de SPI-modules een klok krijgt.
    RCC->APB2ENR = RCC->APB2ENR | RCC_APB2ENR_SPI1EN;       //SPI1 enable

    // SPI1 moet op de alternate function pinnen komen
    GPIOA->MODER = (GPIOA->MODER & ~(GPIO_MODER_MODER5)) | (GPIO_MODER_MODER5_1);
    GPIOA->MODER = (GPIOA->MODER & ~(GPIO_MODER_MODER6)) | (GPIO_MODER_MODER6_1);
    GPIOA->MODER = (GPIOA->MODER & ~(GPIO_MODER_MODER7)) | (GPIO_MODER_MODER7_1);
    GPIOA->AFR[0] &= 0x11011111;
    GPIOA->AFR[0] &= 0x10111111;
    GPIOA->AFR[0] &= 0x01111111;

    // SPI1 mode instellen
    SPI1->CR1 = SPI1->CR1 | SPI_CR1_BR_2 | SPI_CR1_BR_1 | SPI_CR1_BR_0;
    SPI1->CR1 = SPI1->CR1 & ~SPI_CR1_CPOL;
    SPI1->CR1 = SPI1->CR1 & ~SPI_CR1_CPHA;
    SPI1->CR1 = SPI1->CR1 & ~SPI_CR1_BIDIMODE;
    SPI1->CR1 = SPI1->CR1 & ~SPI_CR1_LSBFIRST;
    SPI1->CR1 |= SPI_CR1_SSM;
    SPI1->CR1 |= SPI_CR1_SSI;
    SPI1->CR1 |=  SPI_CR1_MSTR;
    SPI1->CR2 = (SPI1->CR2 & ~(SPI_CR2_DS)) | SPI_CR2_DS_0 | SPI_CR2_DS_1 | SPI_CR2_DS_2;
    SPI1->CR2 = SPI1->CR2 & ~SPI_CR2_FRF;
    SPI1->CR2 |= SPI_CR2_FRXTH;
    SPI1->CR1 |= SPI_CR1_SPE;
}
```

Figuur 94: codefragment instellen SPI module 1

```
void ByteToSpi1(uint8_t data)
{
    // SPI1 data verzenden
    *(uint8_t *)&(SPI1->DR) = data;
    while((SPI1->SR & SPI_SR_BSY) == SPI_SR_BSY);
}
```

Figuur 95: initialisatie en byte versturen via SPI

Vragen:

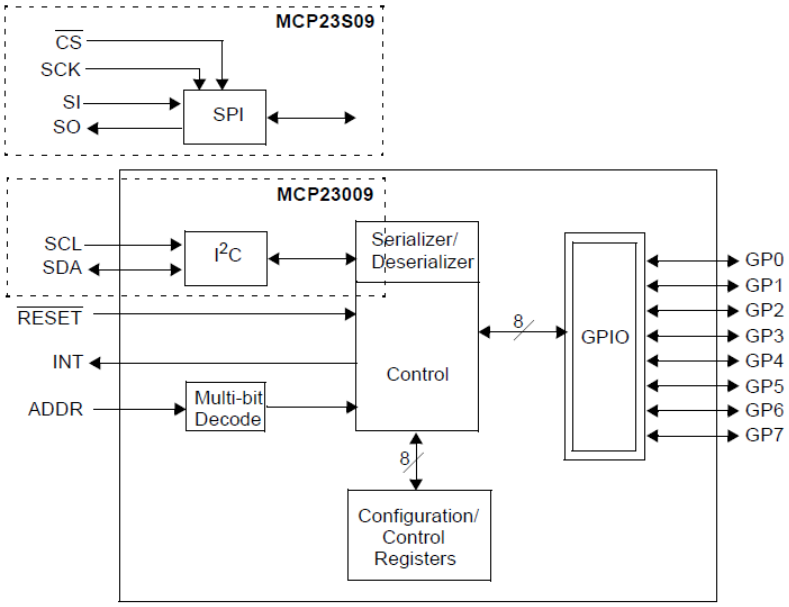
- Bekijk zelf eens hoe de pointerconversies in elkaar steekt.
  De reden waarom we dit doen, is om ervoor te zorgen dat alleen de onderste 8 bits van het DR-register worden geschreven. Dit voorkomt dat onbedoeld extra bits worden verzonden wanneer de SPI in 8-bit modus werkt, aangezien het DR-register hardwarematig als 16-bit gedefinieerd is. Als we direct naar het DR-register zouden schrijven, zouden de bovenste 8 bits onbedoeld worden verzonden. Het gevolg hiervan is dat er junkdata wordt verstuurd, er extra klokpulsen optreden, en de slave de data mogelijk verkeerd interpreteert.
- Waarom staat de while-lus er?

16.3 Toepassingen

16.3.1 IO-expander

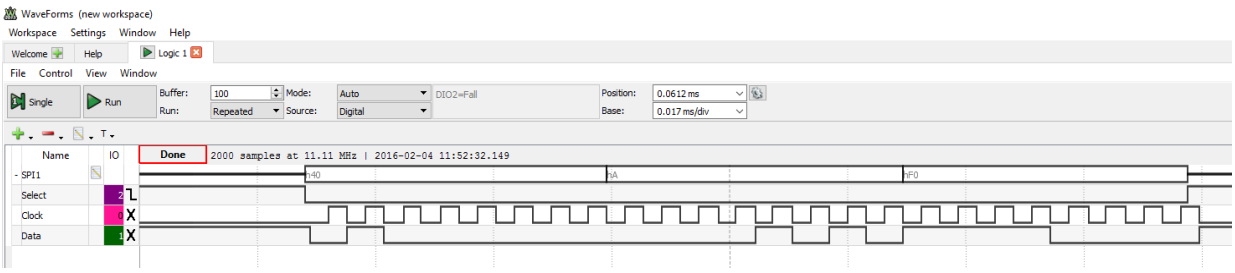
Om meer inputs en outputs te voorzien op een bestaand systeem (microcontroller), kan er gebruik gemaakt worden van een IO-expander. Zo'n IC kan via SPI (of I²C) de toestand van 8 pinnen instellen (output) of uitlezen (input).

Microchip is één fabrikant van dergelijk IC, namelijk de MCP23S09.



Figuur 96: blokschema MCP23S09

Hieronder een beeld van een meting met de oscilloscoop op de SPI-bus gekoppeld met een IO-expander.



Figuur 97: scoopbeeld SPI-communicatie IO-expander

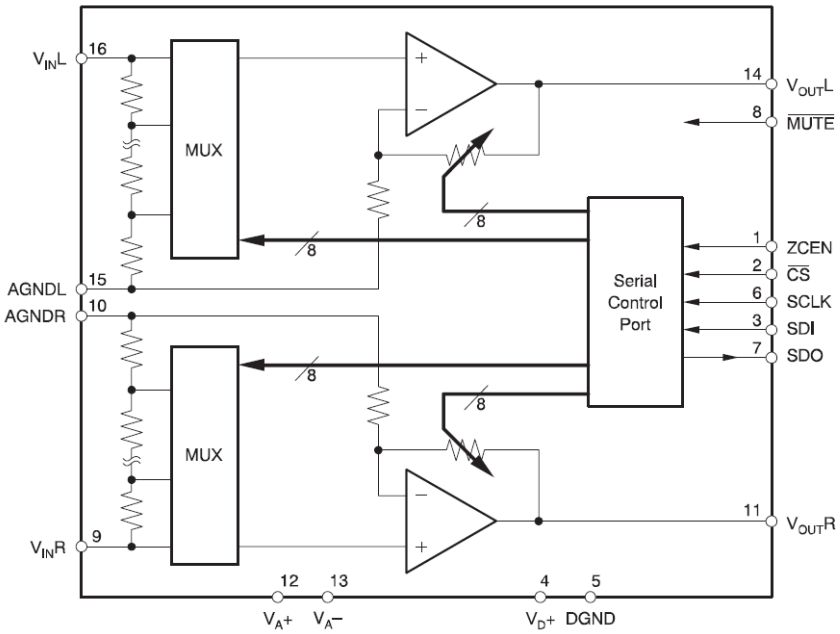
Vragen:

- Kan je uitzoeken wat de toestand van de LED's is?
- Waarom worden 24 bits verstuurd?

16.3.2 Volumeregeling (PGA2311)

Ook volumes van audiosignalen kunnen versterkt of verzwakt worden via SPI-modules. Eén voorbeeld hiervan is de PGA2311.

Bekijk het blokschema. Kan je de SPI-interface terugvinden?

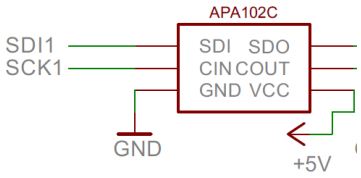


Figuur 98: blokschema PGA2311 (bron: Burr Brown datasheet PGA2311).

16.3.3 Adresseerbare LED (APA102C)

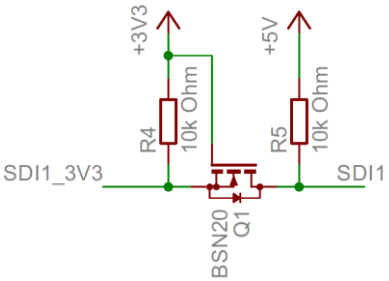
Meer en meer toepassingen met LED-strips verwachten adresseerbare LED's. Een recente ontwikkeling daarbij is het gebruik van RGB-LED's die via SPI aangestuurd kunnen worden. Bijvoorbeeld de APA102C, de zogenaamde 'super LED'.

De aansluiting is als volgt:



Figuur 99: aansluitingen APA102C

De APA102C LED's zijn 5V gevoed. Ook hun SPI-bus moet dus 5V niveau's behandelen. Omdat onze microcontroller op 3V3 werkt, moet dus een **level converter** gebruikt worden. Een eenvoudige bidirectionele level converter kan als volgt gemaakt worden:



Figuur 100: bidirectionele level converter met 1 MOSFET

Vragen:

- Kan je de werking van bovenstaande schakeling verklaren?