Arduino Programming

Theorie Sessie 5

ARTESIS PLANTIJN
HOGESCHOOL ANTWERPEN

AVR-C Toolchain
Preprocessor commando’s
Zelf libraries maken
Arduino OOP

Toepassing

ARTESIS PLANTIJN
HOGESCHOOL ANTWERPEN

AVR-C Toolchain

AVR-C Toolchain

Inleiding

Je staat er misschien niet direct bij stil,
maar er zijn heel wat zaken voor nodig
om de code die je als tekst intikt in de Arduino IDE
uiteindelijk als uitvoerbare bits en bytes
in de AT-Mega328p microcontroler
op de Arduino hardware te kijgen.

Na een klein overzicht gaan we
deze elementen één voor één bespreken

AVR-C Toolchain

Overzicht

de “AVR-C toolchain collection”
bestaat o.a. uit volgende onderdelen:

- Compiler,
- Assembler,
- Linker,
- Standard C library,
- Math libraries,
- Simulator
- Debugger

AVR-C Toolchain

Compiler

Definitie:

Een compiler is een computer programma
dat computer code geschreven in één taal
(de bron taal of “source language”)
vertaalt in een andere computer taal
(de bestemming of “target language”)

AVR-C Toolchain

Compiler

Gebruikelijk is dit een vertaling van een *hogere programmeertaal* naar een *lagere programmeertaal*

- met een hogere programmeertaal bedoelen we een taal die dichter bij de mens staat
- een lagere programmeertaal daarintegen is een taal die dichter ligt bij wat een computer werkelijk begrijpt (binaire code).

AVR-C Toolchain

Compiler

De taal die een computer of microcontroler
begrijpt is primair afhankelijk van zijn CPU

Elke CPU heeft een ingebakken set van instructies
die wanneer het er op aan komt
de énige instructies zijn die werkelijk door de
hardware begrepen kunnen worden.

AVR-C Toolchain

Compiler

Willen we code in een hogere programmeertaal
doen werken op een bepaalde hardware,

dan moet deze code ultiem vertaald worden
naar deze énige instructies
die de CPU van deze hardware kan begrijpen…

AVR-C Toolchain

Compiler

Ook al is de CPU het belangrijkste element,
ook andere hardware onderdelen spelen een rol.

Dit geheel wordt meestal aangeduid met de term

Hardware Architectuur

AVR-C Toolchain

Compiler

Een compiler vertaalt normaal gesproken de code van een hogere taal naar de lagere taal specifiek de architectuur waarop de compiler geïnstalleerd is

Het is ook mogelijk dat een compiler een vertaling doet naar de lagere taal specifiek *een andere hardware architectuur dan deze waar de compiler zelf op geïnstalleerd is.*

In dit geval spreekt men van een ***cross-compiler***

AVR-C Toolchain

Compiler

De AVR-C compiler staat als onderdeel van de Arduino IDE op onze laptop of desktop geïnstalleerd, doch heeft als doel de door ons geschreven code te vertalen naar een lagere taal specifiek aan de Atmel ATMega Microcontroller

de AVR-C compiler is dus een cross-compiler

AVR-C Toolchain

Compiler

Compilers zullen echter zelden de code
van een hogere programmeertaal direct vertalen
naar de eentjes en nullen die de hardware begrijpt.

Compilers vertalen naar assembler code.

Assembler code bestaat uit de specifieke instructies
die de architectuur begrijpt, maar in de vorm van ‘n,
wel héél moeilijk te begrijpen, des-al-niet-te-min
voor de mens nog steeds leesbare tekst

AVR-C Toolchain

Assembler

De volgende element van
de ‘toolchain’ is de *Assembler*.

→ *Het woord ‘chain’ of ketting is goed gekozen hier,*
want het gaat hier effectief over een
opeenvolgende ketting van onderdelen

De Assembler zal nu vervolgens deze assembler-code omzetten naar *binaire code*, dus de reeks ééntjes en nullen die de hardware wél begrijpt.

AVR-C Toolchain

Assembler

Die reeks enen en nullen wordt om alsnog een beetje menselijkheid er aan te geven gebruikelijk neergeschreven in de vorm van hexadecimale-code

Het resultaat van de Assembler is echter géén pure reeks van énen en nullen !!

Hier en daar tussen deze enen en nullen staan er enkele gewone voor de mens leesbare woorden ...

AVR-C Toolchain

Assembler

Het resultaat van de Assembler,
namelijk deze mix van Hex-code en ASCII-tekst,
noemen we **object-code**

Wat zijn nu die woorden leesbare ASCII tekst... ?

AVR-C Toolchain

Assembler

Dit heeft te maken met het feit dat de Compiler
van de oorspronkelijke hogere taal *niet alles*
zal vertalen in de instructies van de Architectuur

maar naar specifieke Assembler commando's die
eigenlijk een soort "verwijzings-teksten" zijn.

Een verwijzing dat op deze plaats,
op dit punt in de code,
een *gekende standard functie* wordt gebruikt.

AVR-C Toolchain

Assembler

In de object-code die de Assembler vervolgens zal produceren zullen deze verwijzingen opnieuw in een bepaalde vorm *behouden blijven*.

Ultiem zijn dit verwijzingen naar de “Standard C library” functies.

AVR-C Toolchain

Standard C Library

Ook al is de volgorde in het overzicht anders, vóór we kunnen begrijpen wat de linker doet moet men eerst weten wat de “*Standard C Library*” juist bevat:

Een C-functie kan men eigenlijk bezien als een appart soort programmaatje.

Een programma die een zekere *input* neemt, en een zekere *output* produceert.

AVR-C Toolchain

Standard C Library

Omdat *alleen de input en de output* van deze code
zal veranderen in de hogere programmeertaal
en *niet de code van deze functies zelf*

zal wanneer de compiler de code
van deze functie moet vertalen
het resultaat *telkens weer* opnieuw
hetzelfde stukje Assembler code zijn...

AVR-C Toolchain

Standard C Library

En als de Assembler vervolgens
dit stukje code omzet naar énen en nullen,
voor ons dus de HEX-code,
dan zal ook dat télkens opnieuw
hetzelfde stukje HEX-code opleveren !

AVR-C Toolchain

Standard C Library

Omdat Compileren en vervolgens Assembling
een *zéér arbeids-intensieve* taak is

en het dubbel zot zou zijn deze zware taak
telkens opnieuw te doen voor dezelfde code

worden *belangrijke functies* van een hogere
programmeertaal voor-gecompileerd
en als een reeks HEX-code bestanden
opgeslagen in een bibliotheek.

AVR-C Toolchain

Linker

De *linker* zal nu de object code
van de Assembler binnen nemen

en overal waar deze een verwijzing ziet
naar een bestaande standard functie
deze verwijzing vervangen door de HEX-code
die de linker kon vinden in de Standard C Library.

Het resultaat van de Linker
is gebruikelijk een “**executable**”

AVR-C Toolchain

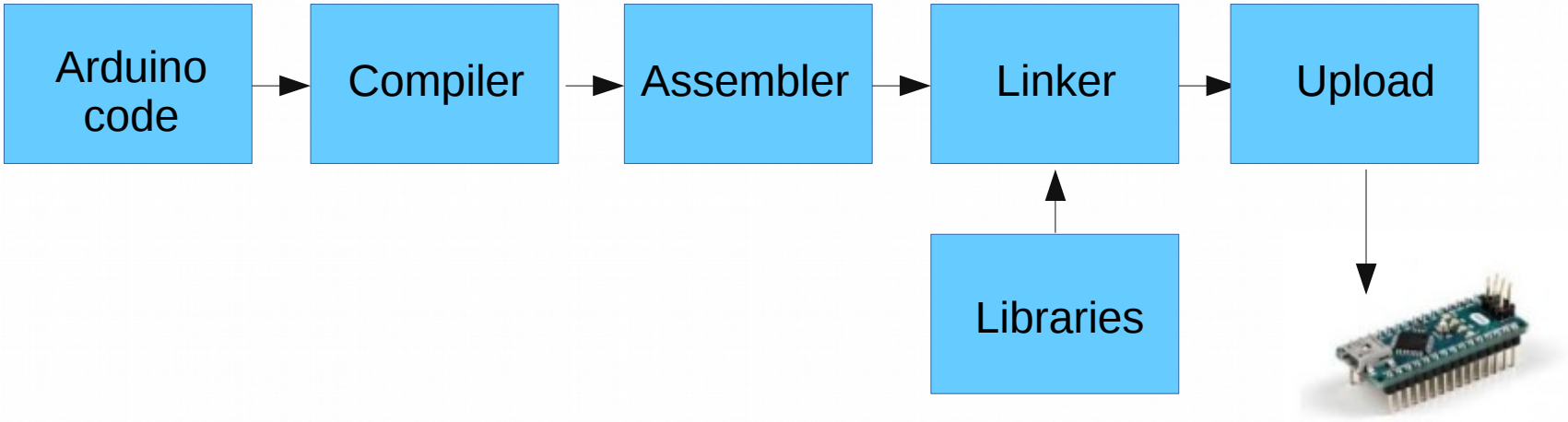
Linker

Een executable of executable-file

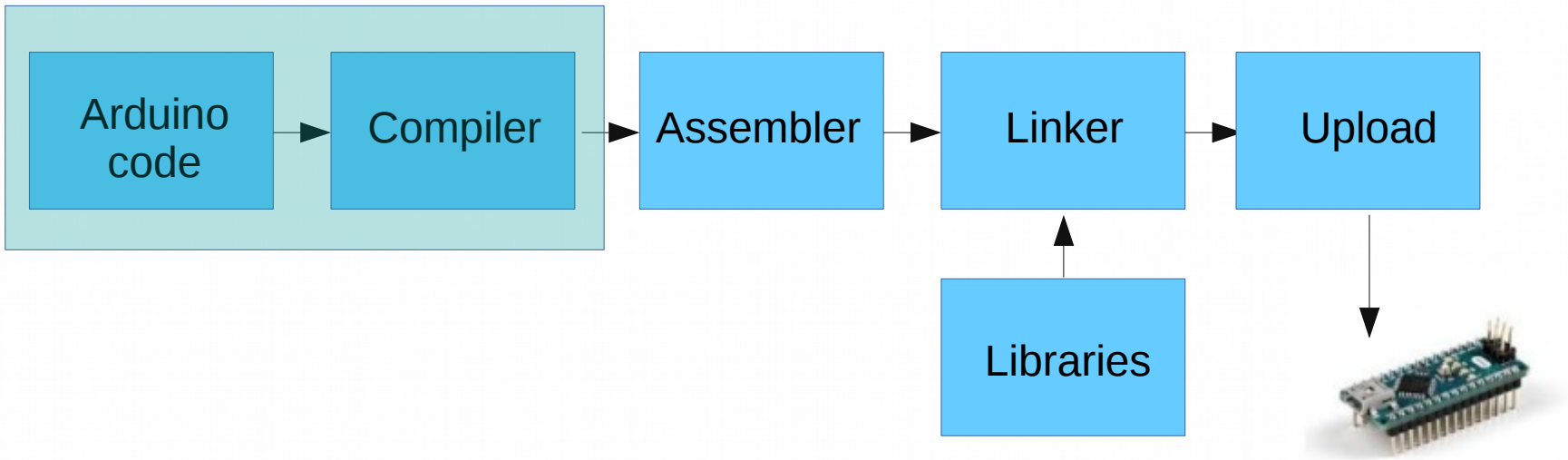
is een bestand die énkel nog HEX-code bevat
en die *rechtstreeks* uitgevoerd kan worden
door de Architectuur waarvoor het bedoeld is.

(Wanneer op de Architectuur ook een
Operating System aanwezig is zal gebruikelijk
de *loader* van het OS de initiatie van
deze uitvoering afhandelen)

AVR-C Toolchain

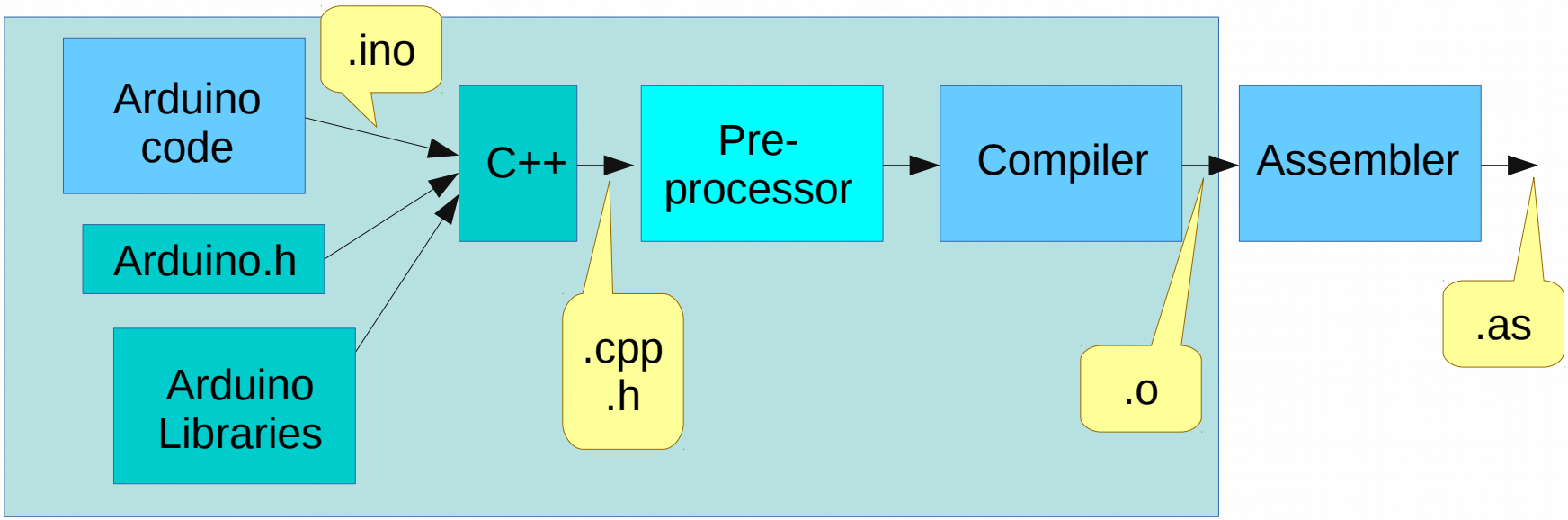


AVR-C Toolchain



AVR-C Toolchain

Compiler Setup



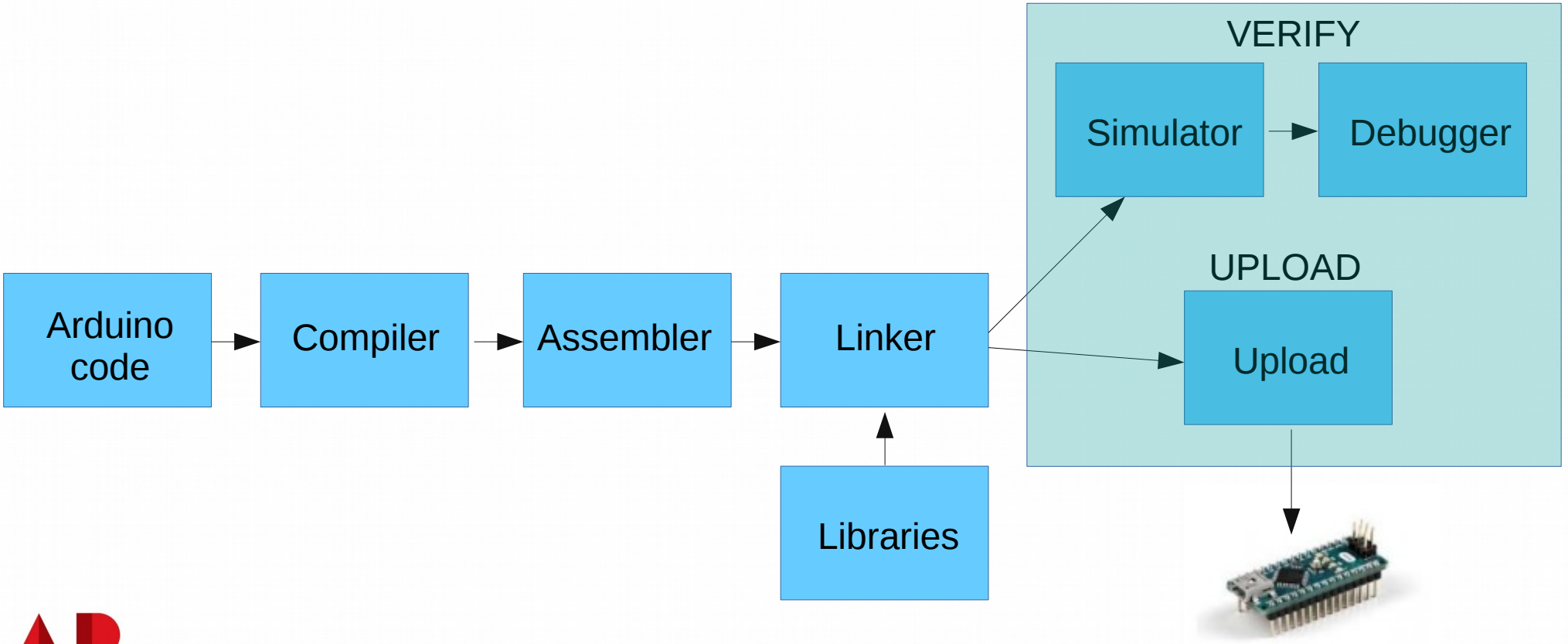
AVR-C Toolchain

Pre-Processor

De compiler *pre-processor* biedt de mogelijkheid om:

- *Header-files* toe te voegen aan de code
- Uitvoeren van *Macro Expansies*
- Aan *Conditionele compilatie* te doen

AVR-C Toolchain



AVR-C Toolchain

Simulator & Debugger

Wanneer we “*Verify*” drukken in de Arduino IDE dan kunnen we onze code controleren op fouten.

Misschien is het reeds opgevallen dat dit lukt ook *zonder* dat de Arduino via USB is aangesloten...

Dit is dankzij de *Simulator*

AVR-C Toolchain

Simulator & Debugger

De ***Simulator*** is een stukje programma code
dat de volledige werking van de microcontroller
Op de hardware *softwarematig gaat imiteren*

Hiermee kan men de werking van de code testen.

De ***Debugger*** zorgt er dan weer voor ons
meer informatie over de fouten te bezorgen
in het *debug venster van de IDE*
alsook hulpmiddelen om fouten te analyseren.

Preprocessor commando’s

Preprocessor Commando's

Algemeen

Preprocessor commando's zijn commando's die aan de gewone programmeercode kunnen toegevoegd worden om bepaalde acties te ondernemen ***vóór*** de code gecompileerd wordt.

Preprocessor commando's starten met een '#' en hebben, in tegenstelling tot andere lijnen van de code, *géén puntkomma* nodig op het einde.

Preprocessor Commando's

#include

Het meest gebruikte pre-processor commando
is zonder twijfel *#include*

Met #include kan men code uit een ander bestand
toevoegen aan de eigen programma code

De pre-processor zal bij het zien van #include
de code uit de vermelde file ophalen
en deze code op de plaats waar
het #include commando staat toevoegen
en dan pas het resultaat naar de compiler sturen.

Preprocessor Commando's

#define

Define is gemakkelijk te verwarren met constante variabelen maar er is een belangrijk verschil:

Met define kan men *labels* maken
die een bepaalde waarde meekrijgen
nét zoals men zou doen met een variabele.

Het verschil is echter dat de pre-processor vóór het de code naar de compiler stuurt overal in de programma tekst deze labels zal vervangen door de opgegeven waarde

Preprocessor Commando’s

#define

Dat wil zeggen dat deze labels,
in tegenstelling tot variable-namen
niet in de uiteindelijk code terecht komen.

Dat wil ook zeggen dat deze labels
gemaakt met #define *géén scope* hebben

Vanuit het standpunt van de compiler is er dus
géén verschil merkbaar met hardcoded waardes...

Preprocessor Commando's

#define

#define combineert dus de voordelen van variabelen
met de voordelen van hardcoded waarden

Het nadeel is dus dat de compiler
géén fouten kan ontdekken
indien er fout gebruik van gemaakt wordt...

Gebruik:

```
#define  OUTPUT   7
#define  BOARD   "nano"
```

Preprocessor Commando’s

#ifdef #ifndef #endif

Een mogelijk probleem met preprocessor commando’s is dat men door het toevoegen van andere files (header-files) met #include meermalig hetzelfde preprocessor commando gaat toevoegen, dit in het slechtste geval ook nog eens met andere waardes ...

Even een voorbeeld om dit te illustreren :

Preprocessor Commando’s

#ifdef #ifndef #endif

Stel je hebt .ino code die start met:

#define OUTPUT 3

vervolgens voegt men een header file toe:

#include <outputs.h>

Preprocessor Commando’s

#ifdef #ifndef #endif

de éérste lijn code van deze header file is echter:

#define OUTPUT 7

.. dan is er een dubbele declaratie van de label
en kan het gecompileerde resultaat wel eens
iets anders zijn dan wat we verwachten...

Preprocessor Commando's

#ifdef #ifndef #endif

Om dit te voorkomen kunnen we een
conditionele preprocessor instructie toevoegen:

#ifdef → wil zeggen "*if defind = true*"
#ifndef → wil zeggen "*if not defind = true*"

Beiden moeten steeds eindigen met een ***#endif***

De preprocessor zal hiermee vóór het uitvoeren
éérst *kijken of er niet reeds éérder* een
instructie was die hetzelfde doel had.

Preprocessor Commando’s

#ifdef #ifndef #endif

We kunnen hiermee in ons voorbeeld
de code van de header file herschrijven als:

```
#ifndef OUTPUT
#define  OUTPUT  7
#endif
```

Preprocessor Commando’s

#ifdef #ifndef #endif

In dit voorbeeld zal de waarde van
OUTPUT 3 blijven ook ná de #define

Wissen we de #define uit de .ino code
dan wordt OUTPUT alsnog 7

Preprocessor Commando's

#ifdef #ifndef #endif

De instructies #ifdef en #ifndef controleren
niet enkel op wat met #define gedefinieerd wordt,
maar om een ruimere interpretatie van "gedefinieerd"

#ifdef en #ifndef werkt bevoorbeeld ook
bij wat gedefinieerd wordt met #include

→ *Het gebruik van #ifndef .. #endif*
bij een #include noemen we een
include guard

Preprocessor Commando’s

Macro’s

Er zijn verder nog een hele reeks preprocessor commando’s:

#define	*Substitutes a preprocessor macro.*
#include	*Inserts a particular header from another le.*
#undef	*Unde nes a preprocessor macro.*
#ifdef	*Returns true if this macro is de ned.*
#ifndef	*Returns true if this macro is not de ned.*
#if	*Tests if a compile time condition is true.*
#else	*The alternative for #if.*
#elif	*#else and #if in one statement.*
#endif	*Ends preprocessor conditional.*
#error	*Prints error message on stderr.*

Preprocessor Commando’s

Macro’s

#error tokens
#warning tokens
#message tokens
#pragma overlap option
#pragma error instruction
#pragma warning instruction

en nog een hele hoop meer ...

Preprocessor Commando’s

Macro’s

Er bestaan genoeg preprocessor commando’s
opdat men er volledige stukjes code
in kan programmeren...

→ dit soort pre-processor code
noemen we een ***macro***

Preprocessor Commando’s

Macro’s

Belangrijk is te zien dat Macro’s
slechts *éénmalig* uitgevoerd worden

dit *vóór* de finale code gecompileerd
wordt en dus deze “macro-code” nooit
in de uiteindelijke executable terecht komt !

Zelf libraries maken

Zelf libraries maken

Inleiding

Deszover hebben we in het labo enkel bestaande libraries gebruikt. Het is echter ook mogelijk om zelf zo'n library te maken.

We zagen in een vorige theorieles reeds het belang hierbij van " " versus < > bij het gebruik van #include, met name in welke directory op disk deze library zal gezocht worden

Zelf libraries maken

Inleiding

Gebruikelijk zullen we zo’n library opsplitsen
in 3 verschillende documenten:

- Een header bestand
- Een code bestand
- Een keyword bestand

Zelf libraries maken

Headerfile (.h)

In het header bestand zullen we al onze preprocessor macro's, globale variabelen, constanten en de "forward declaration" van alle gebruikte functies, klasses en methodes van onze zelf gemaakte library concenteren.

Een "forward declaration" is de definiëring van een functie of klasse met z'n methodes zonder verdere code van deze functie of methodes zelf.

Zelf libraries maken

Headerfile (.h)

Door deze “forward declaration”
weet de preprocessor dat de code
van die functies of klasses
op komst zijn in een volgend bestand

Zelf libraries maken

Codefile (.cpp)

Dit bestand bevat de bulk van de
programma code van onze eigen library.

Belangrijk te weten is dat dit *géén .ino bestand* is

Willen we de typische arduino commando's
gebruiken in de .cpp file van onze library dan moeten
we hier éérst de *Arduino.h library* toevoegen:

#include “Arduino.h”

Zelf libraries maken

Keywordfile (.txt)

Met het keyword bestand kunnen we de Arduino IDE onze klasses en methodes volgens *hetzelfde kleurschema* laten weergeven zoals we dat van de gewone Arduino commando's gewoon zijn.

Dit bestand is niet verplicht
maar helpt uiteraard wel
de gebruikers van onze library...

Zelf libraries maken

Keywordfile (.txt)

Zo'n keyword bestand is een gewoon tekstbestand,
waarin we volgens volgend eenvoudig formaat
de woorden die we een bepaalde kleur
willen geven gaan oplijsten:

klassenaam KEYWORD1
methodenaam KEYWORD2

Dit bestand moet *altijd* de naam "***keywords.txt"***
meekrijgen en in dezelfde map staan als .cpp / .h

Zelf libraries maken

Keywordfile (.txt)

Waar na de beschrijving “*KEYWORD1*” staat
zal deze tekst in de IDE *oranje* worden

Waar “*KEYWORD2*” staat wordt de tekst *bruin*

Om tenslotte de IDE het kleurschema van dit
keyword bestand te laten herkennen
moet de IDE evenwel *herstart* worden.

Arduino OOP

Arduino OOP

Inleiding

De bedoeling hier is niet om een volledig inzicht in alle details van C++ Classes & OOP te geven, maar eerder de verschillen met C# bekijken zodat je ook met klasses in je Arduino code aan de slag kan.

Verder gaan we er hier dan ook vanuit dat je de algemene begrippen van OOP, zoals klasses, methodes, inheritance, polymorphism, …, reeds onder de knie hebt

Arduino OOP

C++ Classes

Het definiëren van een klasse verloopt vrij gelijkaardig met wat je leerde bij C# namelijk:

```
class mijnCppClass {
    …
    …
    …
}
```

Arduino OOP

C++ Classes

Het verschil met C# is dat hier het keyword “public” of “private” *vóór* het keyword “class” ontbreekt...

In C++ kan een klasse *niet in zijn geheel* public of private kan gemaakt worden door het gebruik van dergelijke keywoorden.

Men kan het bereik van klassen evenwel beperken door *de plaats* waar men de klasse gaat definiëren.

Arduino OOP

C++ Classes

Door een definitie ervan in de *header-file*
te plaatsen zal de klasse *publiek* zijn,

definieert men een klasse echter
in de *.cpp file* dan zal de klasse
énkel daar *lokaal* beschikbaar zijn.

Men kan *wél* gewoon de inhoud van de klasse
volledig public, private of protected maken...

Arduino OOP

C++ Classes bereik

Van de members van een klasse
kan men dus het bereik bepalen:

```
class mijnCppClass {
    public:
        int a;
    private:
        int b;
    protected:
        int c;
}
```

Arduino OOP

C++ Classes bereik

Hierbij is de scoop van

public, private en *protected*

identiek aan wat deze bereiken
ook betekenen in C#

Arduino OOP

Constructor - Destructor

Constructors en Destructors worden
op dezelfde manier geschreven als in C#

```
class mijnCppClass {
    public:
        mijnConstructor();
        ~mijnDestructor();
}
```

Destructors werken echter enigsinds anders...

Arduino OOP

Constructor - Destructor

in C++ is het mogelijk om *expliciet zélf* de destructor aan te roepen

daar waar dit in C# enkel zal gebeuren wanneer de *garbage collector* het object vernietigt.

C++ heeft géén garbage collector

dit is een belangrijk verschil tussen C# en C++ !!

Toepassing

Toepassing

Voorbeeldje v/e eigen Library

Om een en ander van de gezien concepten wat duidelijker te maken beschrijven we hier een voorbeeld van een zelf gemaakte library.

De bedoeling is het maken een library
die een LED kan doen knipperen
aan twee verschillende snelheden

Toepassing

Voorbeeldje v/e eigen Library

We starten met de header-file van onze library. We noemen deze file flash.h

We starten dit header bestand met enkele preprocessor commando's:

```
#ifndef flash_h
#define flash_h

#include "Arduino.h"
```

Toepassing

Voorbeeldje v/e eigen Library

Vervolgens beschrijven we de klasse ‘Flash’:

constructor

destructor

```
class Flash
{
  public:
    Flash(int pinnr);
    void fast();
    void slow();
    ~Flash();
  private:
    int _pinnr;
};
```

Toepassing

Voorbeeldje v/e eigen Library

En mogen we niet vergeten het preprocessor commando ‘ifdef’ af te sluiten:

#endif

Toepassing

Voorbeeldje v/e eigen Library

Vervolgens creëren we het .cpp bestand
met de hoofdbrok van onze code.

```
#include "Arduino.h"
#include "Flash.h"

Flash::Flash(int pinnr)
{
      _pinnr = pinnr;
      pinMode(_pinnr, OUTPUT);
}
```

constructor

Toepassing

Voorbeeldje v/e eigen Library

```
void Flash::slow()
{
  digitalWrite(_pinnr, HIGH);
  delay(500);
  digitalWrite(_pinnr, LOW);
  delay(500);
}
```

Toepassing

Voorbeeldje v/e eigen Library

```
void Flash::fast()
{
  digitalWrite(_pinnr, HIGH);
  delay(100);
  digitalWrite(_pinnr, LOW);
  delay(100);
}
```

destructor

```
Flash::~Flash()
{
  digitalWrite(_pinnr, LOW);
}
```

Toepassing

Voorbeeldje v/e eigen Library

Om onze library de echte Arduino look te geven creëren we ook het *keywords.txt* bestand:

Flash	*KEYWORD1*
slow	*KEYWORD2*
fast	*KEYWORD2*

Toepassing

Voorbeeldje v/e eigen Library

Nu zijn we klaar met alle bestanden van de library.

Belangrijk is dat we al deze bestanden
in een map plaatsen met de naam “Flash”.

Vervolgens plaatsen we deze in de
Map libraries onder de Arduino folder

(behalve bij het gebruik van #include <>)

Toepassing

Voorbeeldje v/e eigen Library

Finaal kunnen we onze library als volgt gebruiken in onze arduino sketch:

```
#include "Flash.h"

Flash knipper(4);

void setup() {

 }
```

Toepassing

Voorbeeldje v/e eigen Library

Finaal kunnen we onze library als volgt gebruiken in onze arduino sketch:

```
#include "Flash.h"

Flash knipperLED(4);

void setup() {

 }
```

De setup is hier leeg want pinMode staat reeds in de constructor

Toepassing

Voorbeeldje v/e eigen Library

```
void loop() {

    for ( int n=0; n<3; n++) {
      knipperLED.fast();
    }

    for ( int n=0; n<3; n++) {
      knipperLED.slow();
    }
}
```

Arduino Library Manager

Properties File

Willen we nu onze library beschikbaar maken
in de IDE en/of aan de Arduino gemeenschap

dan moeten de “Arduino Library Manager”
informeren over het bestaan van onze library.

Dit doen we via *meta-data* over onze library
te schrijven in een ***library.properties*** bestand

Arduino Library Manager

Properties File

voorbeeld van een “library.properties” file:

name=WebServer
version=1.0.0
author=Cristian Maglie <c.maglie@example.com>
maintainer=Cristian Maglie <c.maglie@example.com>
sentence=A library that makes coding a Webserver a breeze.
paragraph=Supports HTTP1.1 both GET and POST.
category=Communication
url=http://example.com/
architectures=avr
includes=WebServer.h

Arduino Library Manager

Properties File

Zo’n properties bestand te schrijven moet volgens een bepaald formaat met heel wat criteria.

Deze criteria staan bekend als de:

Arduino IDE Library Specifications

De details hiervan zullen we niet bespreken doch kunnen geraadpleegd worden via *github.com* bij de *Arduino wiki pagina’s*