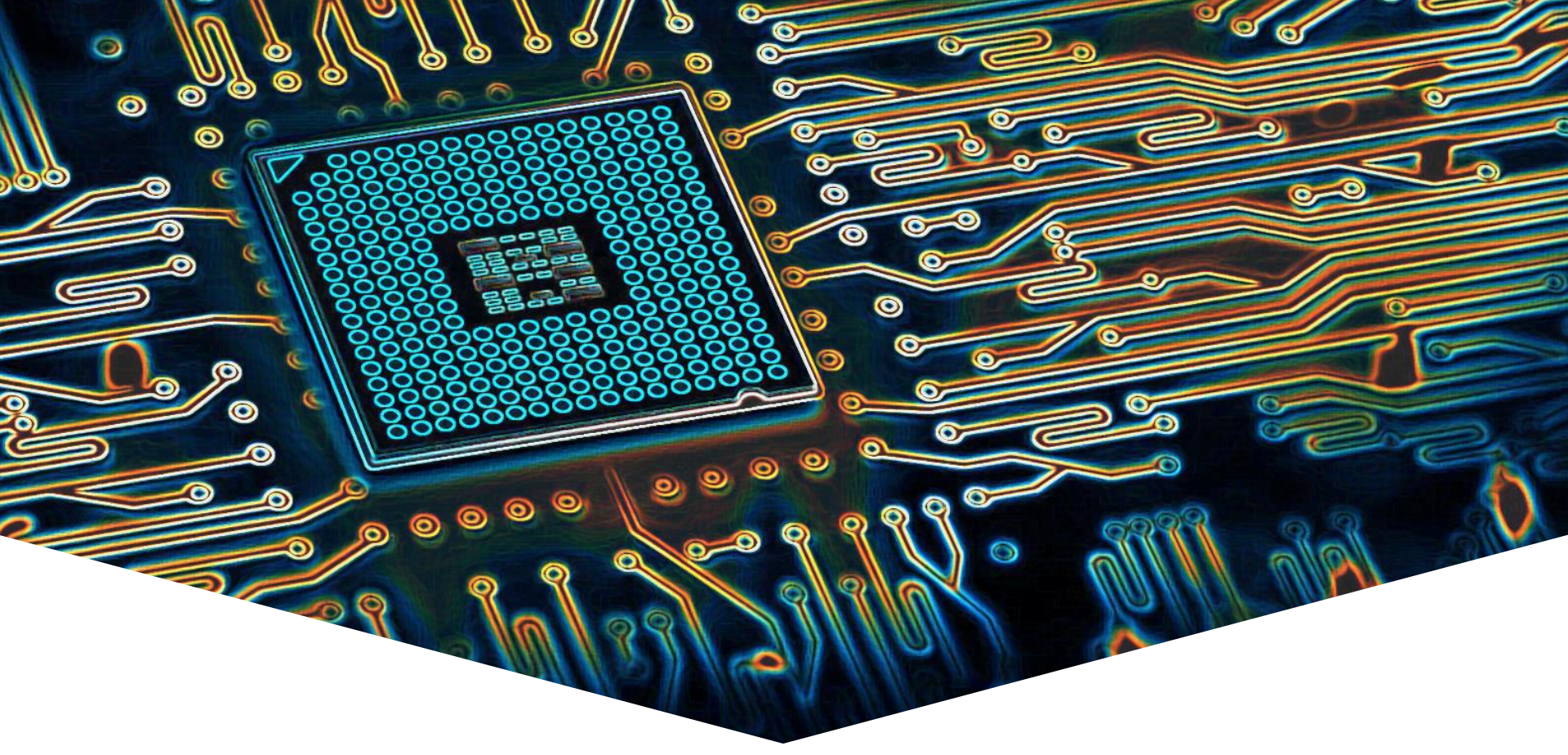


EMS20 Week 4 Lab 1

Leerdoelen Week 4 Lab 1

Leerdoelen week 4 lab 1. Je leert:

- wat een Operating System (OS) voor je kan doen;
- wat het verschil is tussen een General Purpose OS (GPOS) en een Real-Time OS (RTOS);
- waarom het handig is om taken concurrent uit te voeren;
- hoe je taken met behulp van TI-RTOS threads concurrent kan uitvoeren;

Wat doet een OS?

Wat is de belangrijkste functie van een OS (zoals Windows 10 / Linux / Android / iOS)?

- M.b.v. een OS kun je meerdere taken concurrent uitvoeren op één machine.
  - Concurrent is “gelijktijdig”.
    - Op een multicore machine echt gelijktijdig, op een single-core machine schijnbaar gelijktijdig.
  - Het OS verdeelt de beschikbare resources (bronnen) CPU, memory, I/O, enzovoort over de taken.

Zie je boek: hoofdstuk 12

RTOS versus GPOS

GPOS

- OS beschermt taken (applicaties) t.o.v. elkaar.
- OS is de baas (bepaalt prioriteiten van de taken).
- Optimaliseert throughput.

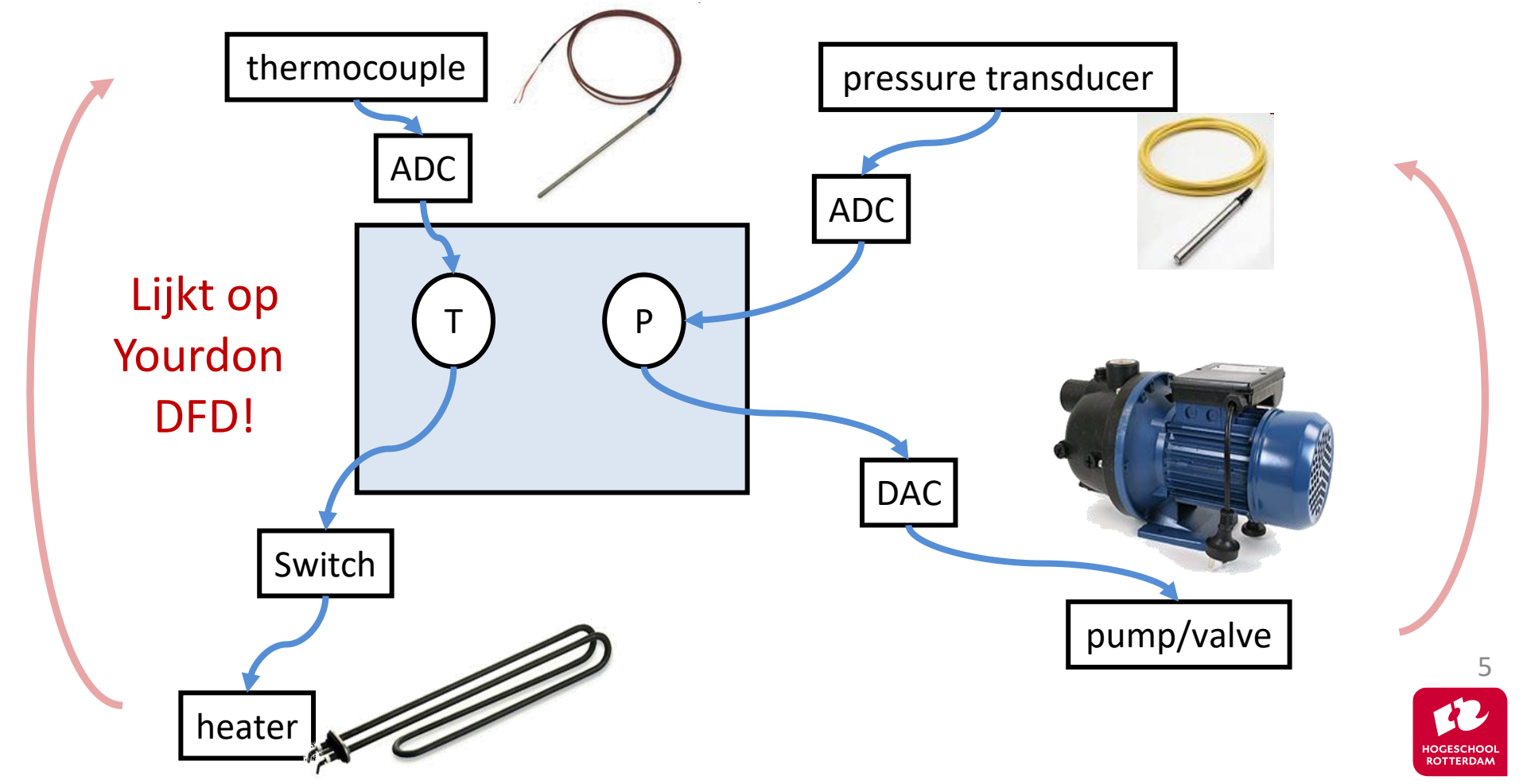
Geschikt voor pc en telefoon.

RTOS

- Taken werken samen.
- Programmeur is de baas (bepaalt prioriteiten van de taken).
- Responsetijden zijn voorspelbaar.

Geschikt voor embedded system.

Voorbeeld embedded systeem



Beschikbare functies


```
double readTemp(void);
void writeSwitch(bool i);
double readPres(void);
void writeDAC(double d);
bool tempControl(double temp);
double presControl(double pres);
```

Oplossing zonder OS


```
int main(void) {
    double temp, pres, dac;
    bool switch_;
    while (1) {
        temp = readTemp();
        switch_ = tempControl(temp);
        writeSwitch(switch_);
        pres = readPres();
        dac = presControl(pres);
        writeDAC(dac);
        sleep(1);
    }
    return 0;
}
```

Problemen van deze oplossing

- Sample rate van temperatuur en druk is gelijk.
  - Kan wel wat aan worden gedaan met tellers maar: Wat doe je als `pressureControl` langer duurt dan gewenste sample rate van temperatuur?
- Als `readTemperature` vastloopt (blijft pollen) dan loopt ook de drukregeling vast.

De temperatuurregeling en de drukregeling zijn twee afzonderlijke “processen”. Maar in het sequentiële programma zitten ze verweven!

Concurrent oplossing met OS

Definieer twee taken:

```
void tempThread(void) {
    double temp;
    int switch_;
    while (1) {
        temp = readTemp();
        switch_ = tempControl(temp);
        writeSwitch(switch_);
        sleep(3);
    }
}
```

Concurrent oplossing met OS


```
void presThread(void) {
    double pres, dac;
    while (1) {
        pres = readPres();
        dac = presControl(pres);
        writeDAC(dac);
        sleep(1);
    }
}
```

Vraag het OS om deze taken concurrent uit te voeren.
(code is OS specifiek)

Dit lab: POSIX

POSIX

- Portable Operating System Interface (POSIX) is een standaard API voor OSen.
  - Veel OSen conformen zich (gedeeltelijk) aan deze standaard. Bijvoorbeeld: Linux, Android, OSX, VxWorks, QNX Neurtino, TI-RTOS enz.
- Taken worden dynamisch aangemaakt d.m.v. API calls.
- Message Queues (en andere resources) worden ook dynamisch aangemaakt.

POSIX maakt gebruik van features uit C die je nog niet (goed) kent:

- Pointers naar functies
- `void` pointers

Pointers naar functies

In C kun je een pointer naar een functie definiëren.

- De waarde van de pointer is het beginadres (van de code) van de functie.

```c
#include <stdio.h>

int kwadraat(int c) {
    return c * c;
}

int dubbel(int c) {
    return c + c;
}
```

`pnf` is een pointer naar een functie met een `int` als parameter en een `int` returnwaarde

```c
int main(void) {
    int a = 7, b;
    int (*pnf)(int);
    pnf = &dubbel;
    b = (*pnf)(a);
```

Waarom haakjes?

Pointers naar functies

In C kun je een pointer naar een functie definiëren.

- De waarde van de pointer is het beginadres (van de code) van de functie.

```c
#include <stdio.h>

int kwadraat(int c) {
    return c * c;
}

int dubbel(int c) {
    return c + c;
}
```

pnf wijst naar de functie dubbel (pnf wordt gelijk aan het adres van de functie dubbel)

```c
int main(void) {
    int a = 7, b;
    int (*pnf)(int);
    pnf = &dubbel;
    b = (*pnf)(a);
```

Pointers naar functies

In C kun je een pointer naar een functie definiëren.

- De waarde van de pointer is het beginadres (van de code) van de functie.

```c
#include <stdio.h>

int kwadraat(int c) {
    return c * c;
}

int dubbel(int c) {
    return c + c;
}
```

De functie waar pnf naar wijst wordt aangeroepen met de waarde van a als argument

```c
int main(void) {
    int a = 7, b;
    int (*pnf)(int);
    pnf = &dubbel;
    b = (*pnf)(a);
```

Waarom haakjes?

source

Pointers naar functies

Verkorte schrijfwijze.

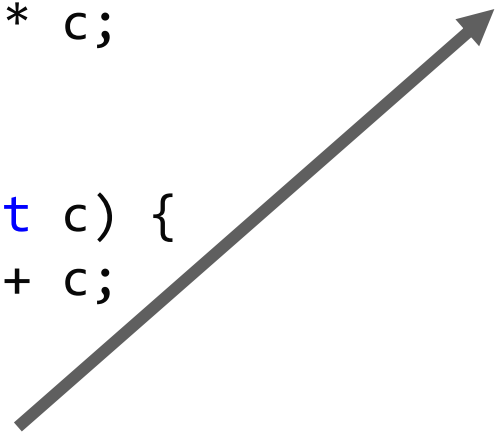
- Naam van een functie ⟷ beginadres (van de code) van de functie.

```
#include <stdio.h>

int kwadraat(int c) {
    return c * c;
}

int dubbel(int c) {
    return c + c;
}
```

```
int main(void) {
    int a = 7, b;
    int (*pnf)(int);
    pnf = dubbel;
    b = pnf(a);
```



source

Pointers naar functies

Wat is het nut?

- Functie als parameter.

```
#include <stdio.h>
/* … */
void printTabel(int (*p)(int), int van, int tot, int stap) {
    for (int x = van; x < tot; x += stap) {
        printf("%10d %10d\n", x, (*p)(x));
    }
}
int main(void) {
    printf("De kwadraten van 1 t/m 10\n");
    printTabel(&kwadraat, 1, 11, 1);
    printf("De dubbelen van de drievouden van 0 t/m 30\n");
    printTabel(&dubbel, 0, 31, 3);
```

source

Uitvoer


```
De kwadraten van 1 t/m 10
          1           1
          2           4
          3           9
          4          16
          5          25
          6          36
          7          49
          8          64
          9          81
         10         100
De dubbelen van de drievouden van 0 t/m 30
          0           0
          3           6
          6          12
          9          18
         12          24
         15          30
         18          36
         21          42
         24          48
         27          54
         30          60
```

void *

- Een `void *` kan wijzen naar elk type.
- Als je de waarde wilt ophalen waar een `void *` naar wijst dan moeten we de pointer casten naar het juiste type.

```
int main(void) {
    int i = 3;
    double d = 4.3;

    void* vp = &i;
    printf("%d\n", *(int*)vp);
    vp = &d;
    printf("%lf\n", *(double*)vp);

    return 0;
}
```

source

void *

- Ook structs!

```
typedef struct {
        int waarde1;
        double waarde2;
} mijnStruct;

void mijnFunc(void * par)
{
        mijnStruct lokaal = *(mijnStruct *)par;
        printf( “Waarde1: %d, Waarde2: %f”,
                lokaal.waarde1,
                lokaal.waarde2);
}
int main()
{
        mijnStruct test = {1, 2.123};
        mijnFunc(&test)
}
```

Aan de slag!

Aan de slag met Opdrachten_Week_4_Lab_1.pdf