

Rapporttitel

ONDERTITEL RAPPORT

Naam | Cursustitel | Datum

# Week 3

## 3.3

### B

Code:



### C

Omdat de flag verandert moet kunnen worden in de ISR, en de compiler moet deze variabele negeren tijdens het optimaliseren. Als dit niet gebeurt kan de variabele zodanig veranderd worden dat de ISR en niet meer bij kan.

### D

Code:



## 3.4



## 3.5



## 3.6

Different functions for task management:





Task structure with task functions

**enum** {*NOT\_AVAILABLE*, *WAITING*, *READY*};

**void** **Toggle\_Green**(**void**){

GPIOD->ODR ^= 0x1000;

}

**void** **Toggle\_Orange**(**void**){

GPIOD->ODR ^= 0x2000;

}

**void** **Toggle\_Red**(**void**){

GPIOD->ODR ^= 0x4000;

}

**void** **Toggle\_Blue**(**void**){

GPIOD->ODR ^= 0x8000;

}

**struct** tasks{

**void** (\*function)(**void**);

**int** period;

**int** counter;

**int** initial\_delay;

**int** state;

} task;

**struct** tasks task\_list[8];

Main function section for task adding and interrupt waking:

**int** ret = Add\_task(&Toggle\_Green, 200, 100);

**if** (ret == 0){

**while**(1);

}

ret = Add\_task(&Toggle\_Orange, 500, 200);

**if** (ret == 0){

**while**(1);

}

ret = Add\_task(&Toggle\_Red, 750, 300);

**if** (ret == 0){

**while**(1);

}

ret = Add\_task(&Toggle\_Blue, 300, 400);

**if** (ret == 0){

**while**(1);

}

// Do forever:

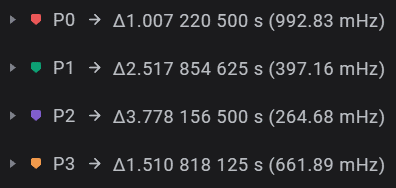
**while** (1)

{

\_\_WFI();

runReadyTasks();

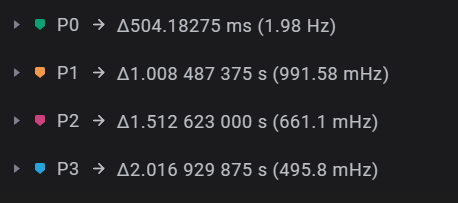
}



Results from logic analyzer

|  |  |
| --- | --- |
| Logic signal | Corresponding function |
| P0 | Toggle\_Green |
| P1 | Toggle\_Orange |
| P2 | Toggle\_Red |
| P3 | Toggle\_Blue |

## 3.7



Results from the logic analyzer including the initial delay ticks. Every tick is 0.005s, therefore 100 ticks is equal to 0.5S

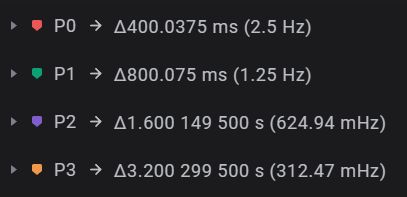
|  |  |  |  |
| --- | --- | --- | --- |
| Logic signal | LED | Set cycles (ticks) | Corresponding time (s) |
| P0 | Green | 100 | 0.5 |
| P1 | Orange | 200 | 1 |
| P2 | Red | 300 | 1.5 |
| P3 | Blue | 400 | 2 |

# Week 4

## 4.1

### D

The times are x4 because every 1ms the tasks switches to the next task. Therefore after 4ms task 1,2,3 and 4 are 1ms further in their own scope. This is an order execution of tasks, without priorities.



|  |  |  |  |
| --- | --- | --- | --- |
| Logic signal | LED | Set cycles (ticks) | Corresponding time (ms) |
| P0 | Green | 100 | 400 |
| P1 | Orange | 200 | 800 |
| P2 | Red | 300 | 1200 |
| P3 | Blue | 400 | 1600 |

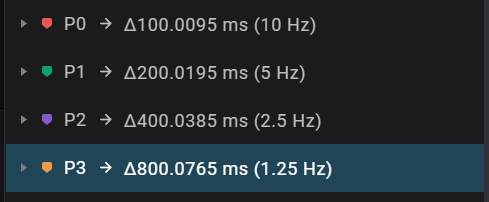
## 4.2

### A

Adjustments in code:



### B



|  |  |  |  |
| --- | --- | --- | --- |
| Logic signal | LED | Set cycles (ticks) | Corresponding time (ms) |
| P0 | Green | 100 | 100 |
| P1 | Orange | 200 | 200 |
| P2 | Red | 300 | 400 |
| P3 | Blue | 400 | 800 |

## 4.3

### A



Code for changing period to 2ms



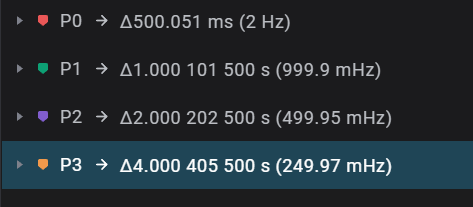
Added a set Control function to set the nPRIV bit high.

## 4.4









With 1 tick being 5ms (x5 from previous assignment) this is the result

|  |  |  |  |
| --- | --- | --- | --- |
| Logic signal | LED | Set cycles (ticks) | Corresponding time (ms) |
| P0 | Green | 100 | 500 |
| P1 | Orange | 200 | 1000 |
| P2 | Red | 400 | 2000 |
| P3 | Blue | 800 | 4000 |

## 4.5

Deze opdracht hebben we niet gemaakt

# Week 5

## 5.1

### A

downloaden van het project

### B

importeren van het project, en checken of deze werkt. Hierbij krijgt de consument een prioriteit van 3 (hoogste), de frikandel producent een prioriteit van 2, en de kroket producent een prioriteit van 1 (laagste). Dit is de output die we krijgen op ons console:



Eerst wordt er gevraagd naar de prioriteiten. Hierbij krijgt de consument een prioriteit van 3 (hoogste), de frikandel producent een prioriteit van 2, en de kroket producent een prioriteit van 1 (laagste). Hierna start het de consumenten thread (0x200015e8) als eerste, want deze heeft de hoogste prioriteit. Omdat de frikandellen producent prioriteit 2 heeft, start deze thread (0x20001b80) hierna. Direct hierna start de consument met het consumeren van de frikandellen dat te zien is aan “FFFFFFFFFF…”. Als de frikandellen allemaal gemaakt zijn, stopt de frikandellen producent. Omdat de consument de hoogste prioriteit heeft, is deze eerder klaar met alle frikandellen consumeren dan dat de frikandel producent thread stopt. Omdat de kroket producent prioriteit 1 heeft, start deze thread (0x20002118) hierna. Direct hierna start de consument met het consumeren van de kroketten dat te zien is aan “KKKKKKK…”. Als de kroketten allemaal gemaakt zijn, stopt de kroketten producent. Omdat de consument de hoogste prioriteit heeft, is deze eerder klaar met alle kroketten consumeren dan dat de kroket producent thread stopt.

Er zal maar één snack in de buffer terecht komen voordat de eerste snack is geconsumeerd, omdat de consument de hoogste prioriteit heeft.

### C

sem\_wait zorgt er voor dat maar één taak tegelijkertijd gedaan kan worden. Normaal zou eerst de get() functie uitgevoerd worden, en daarna de rest van de consument functie. Nu wordt er eerst begonnen aan de consument functie, en deze is nog niet afgemaakt, en kan dus niet naar de get() functie. Hierdoor loopt stalls het programma.

### D

compileer het originele programma met prioriteiten: consument = 1, frikandel producent =2 en kroket producent =3. Dit genereerd de volgende output:



Aangezien de kroket producent (0x20002118) de hoogste prioriteit heeft, start deze eerst. Hierna start de frikandel producent (0x20001b80) meteen. Nadat deze twee zijn gestart, start de consument (0x200015e8) die de laagste prioriteit heeft. De kroketten worden als eerst geconsumeerd. Nadat de kroket producent klaar is, worden er nog kroketten geconsumeerd door de consument, omdat er nog kroketten in de buffer zitten. sem\_Empty = -1. Dit betekend dat de buffer helemaal vol zit +1, en er dus nog 9 kroketten geconsumeerd worden nadat de kroket producent is gestopt. Hierna worden de frikandellen geconsumeerd. Nadat de frikandellen producent is gestopt, worden er nog 9 frikandellen geconsumeerd, omdat de buffer nog vol zit, en sem\_Empty =-1. Hierna stopt ook de consumenten thread, en is het programma klaar.

### E

Ik verwacht dat het programma het zelfde draait als in opdracht D, alleen met de frikandel producer en kroket producer omgedraaid.



Het programma draait inderdaad zoals we hadden verwacht.

### F

Ik verwacht dat het programma de twee producers om en om laat draaien, en dus steeds van producer wisselt. daarom zal de consument ook om en om een kroket en een frikandel krijgen.



Het programma draait zoals we hadden verwacht.

### G

Ik verwacht dat het programma de twee producers om en om laat draaien, en dus steeds van producer wisselt. daarom zal de consument ook om en om een kroket en een frikandel krijgen. We krijgen daarnaast ook weer hetzelfde geval als in opdracht D, waar de buffer nog niet leeg is nadat de producer is gestopt.



Het programma draait zoals we hadden verwacht.

## 5.2

### A

maken van het project, en check of het voorbeeld werkt.

### B vervang het globale variabel buffer en de semaforen semMutialExclusive, semEmpty en semFilled met een message queue.

### 





### C

welke prioriteit zou je de *messages* moeten geven om real-time gedrag te implementeren?

Op het moment dat een bepaald bericht een deadline heeft, die korter is als de ander, zal deze een hogere prioriteit moeten krijgen.

## 5.3

### A

maken van het project, en check of het voorbeeld werkt.

### B

vervang de semafoor *semPrintf* met een mutex.







Week 6 rekenopdracht

# A

**Bepaal voor elke taak i de prioriteit Pi als DMPA (Deadline Monotonic Priority  
Assignment) wordt gebruikt. De hoogste prioriteit die je mag gebruiken is 5 en de  
laagste prioriteit die je mag gebruiken is 1.**

Bij DMPA wordt niet gekeken naar de periodetijd maar naar de deadline. Hierbij is de kleinste deadline de grootste prioriteit. Hierdoor krijg je:

|  |  |
| --- | --- |
| Taak (i) | Prioriteit (P) |
| 1 | 2 |
| 2 | 4 |
| 3 | 5 |
| 4 | 3 |
| 5 | 1 |

# B

**Bereken voor elke taak i de blocking time Bi.**

* Bi = Maximale blocking time voor taak i
* = totaal aantal gebruikte resources
* = Boleaanse functie met de volgende statements:
  + Als een lagere prioriteit resource gebruikt
  + En als taak of een hogere prioriteit resource gebruikt
* Als beide statements waar zijn wordt dit stukje van de functie 1, anders 0.
* = maximale vergrendeltijd van resource

Als je dit invult bij elke taak krijg je de volgende resultaten:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Blocking time | Resource 1 | Resource 2 | Resource 3 | Resource 4 | Totaal blocking time |
|  | 0 | 100 | 0 | 0 | 100 |
|  | 0 | 100 | 50 | 20 | 170 |
|  | 25 | 100 | 0 | 0 | 125 |
|  | 0 | 100 | 50 | 0 | 150 |
|  | 0 | 0 | 0 | 0 | 0 |

# C

**Bereken voor elke taak i of de deadline wordt gehaald en geef, indien de deadline  
wordt gehaald, de response tijd Ri.**

Om antwoord te geven op deze vraag moeten alle response tijden van de taken geanalyseerd en berekent worden. Dit is hieronder te zien van hoogste naar laagste prioriteit:

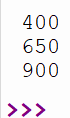
## Prioriteit 5 berekening

De volgende functies van berkening 2 tm 5 worden geïtereerd tot = , hieronder is de code ook te vinden in python per berekening

## Prioriteit 4 berekening



Resultaat:

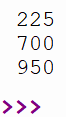
R2 is dus 900.

. Hierdoor zal de deadline behaald worden.

## Prioriteit 3 berekening



Resultaat:

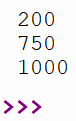
R4 is dus 950

. Hierdoor zal de deadline behaald worden.

## Prioriteit 2 berekening



Resultaat:

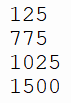
R1 = 1000

. Hierdoor zal de deadline behaald worden.

## Prioriteit 1 berekening



Resultaat:

R5 = 1500

. Hierdoor zal de deadline niet behaald worden.