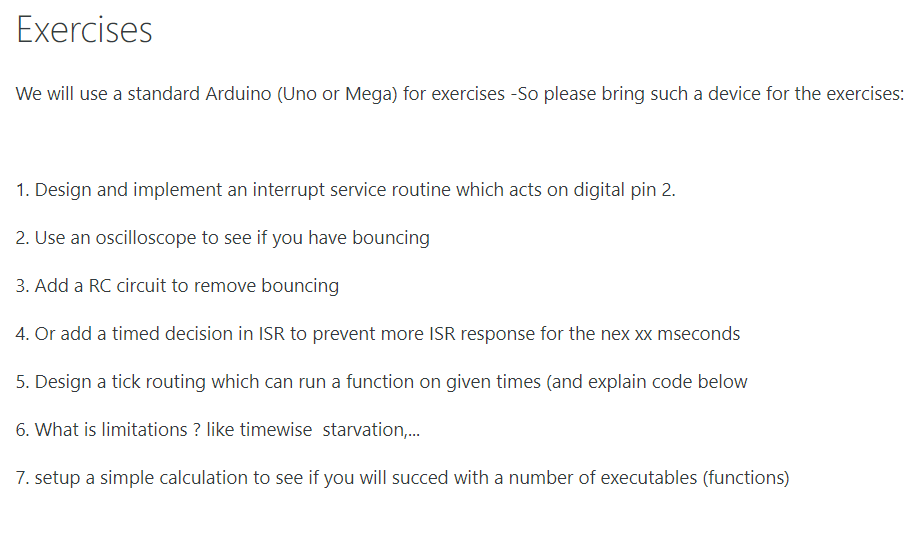
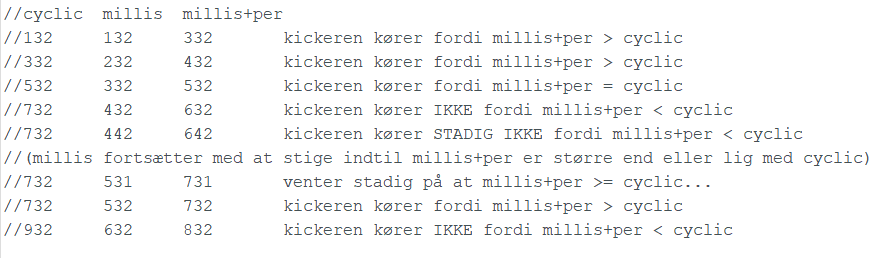
# #1: Introduction to embedded SW

**QUESTION NO 1** : show code and explain for 1,4,5,6 (design and impl ISR, time based debouncing, tick routing, limitations)



1. Vi har oprettet en ISR funktion kaldet **ISR1**, Som er styret af en knap tilsluttet PIN 2. Ved hjælp af Arduinos egen **attachInterrupt()** funktion.
2. Debounce: I vores **ISR1** funktion, gemmes **micros()** I **newTrigger** og derefter tjekkes der om **newTrigger** er 1 sek større end **lastTrigger**. Hvis den er så tændes LED13
3. Ved **setup** bliver **codeCyclic’s** næste starttid sat til “lige nu”. Loopet kører så igennem en funktion (**kicker();**), der sørger for at eksekvere **codeCyclic** hver XX ms (**per**
4. Når koden starter (vha. **Setup)**, bliver **cyclic** sat til det nuværende tidspunkt. **Cyclic** er da på det tidspunkt hvor **codeCyclic** senest skal eksekveres. **codeCylic** kan ikke gå i gang før den forrige deadline er nået, men den forrige **codeCyclic** er nødt til at gå i gang med det samme, hvis der er mindre end XX ms (**per**) tilbage før deadline.  
   Begrænsningen her er, at når koden starter, så er der allerede mindre end XX ms tilbage af perioden, og **codeCyclic** er nødt til at gå i gang med det samme. Efter første eksekvering er koden dog stadig bagud I forhold til deadline, og den er igen nødt til at gå i gang med det samme (timewise starvation princippet). Efter et par iterationer bliver koden dog stabil, da perioden før deadline er betydeligt længere end eksekveringstiden.



# #4: Embedded kernels

**QUESTION NO 2:**

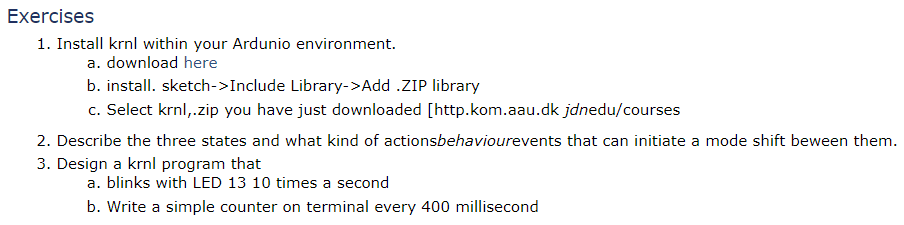
Show LED13 blink code and explain program

Explain interrupt

Lack of control maybe bq of non-interruptable ISRs

Procesmodel: running,ready,blocked

semaphore wait and signal



1. **T1** funktionen oprettes:
   1. Indeholder **k\_sleep(50)** der er en af **krnl.h** funktionerne, som fungerer som et 50 ms **delay** men kun på nuværende **krnl opgave** og pauser IKKE hele koden (active wait).
   2. Derefter bruges **digitalWrite(LED13, HIGH)** til at tænde for onboard-led’en, og efter tændt I 50 ms **digitalWrite(LED13, LOW)** til at slukke den igen.
2. **K\_init(2,0,0)** er sat til at udføre 2 tasks samtidig. De to tasks der udføres er **t1** og **t2** som styres af **k\_crt\_task**, **krnl** funktionen, som opsættes efter (task nr., prioritet(mindst udføres først), stack, stack størrelse)   
   \*(HINT: Hver task SKAL have sin egen **stack**)

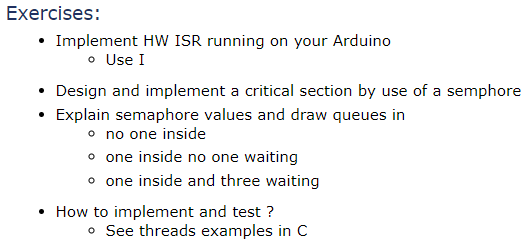
RUNNING: The process that is currently being executed.

READY: A process that is queuing and prepared to execute when given the opportunity.

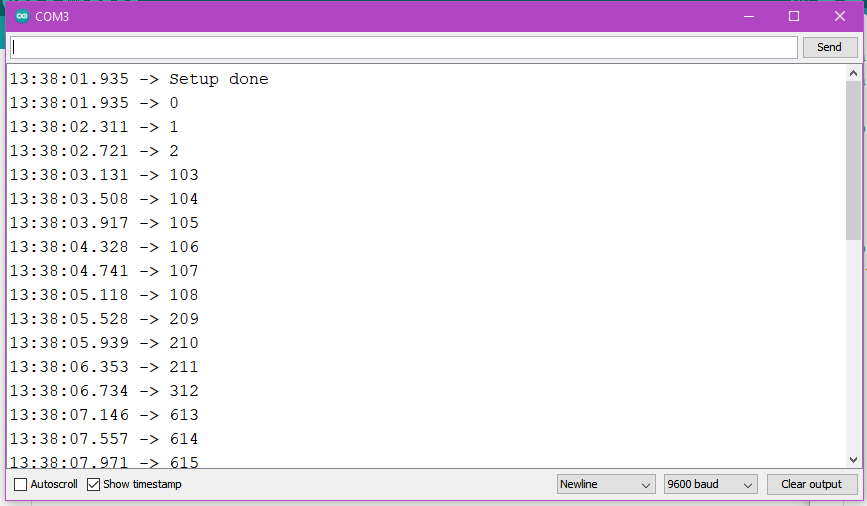
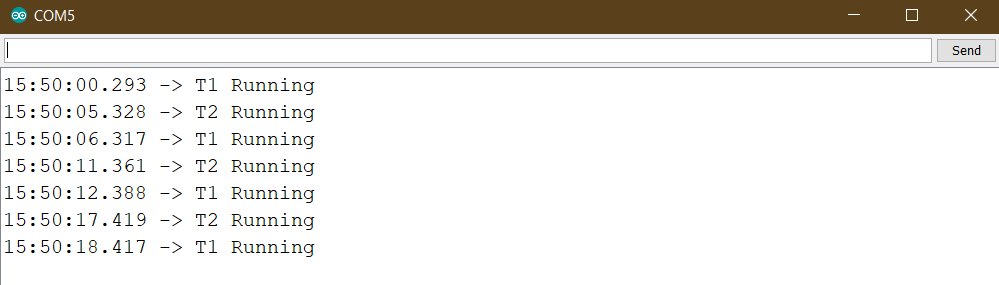
BLOCKED: A process that cannot execute until some event occurs, such as the completion of an I/O operation. (Semaphores as an example)

# #5: Embedded kernels II

**QUESTION 3:** What is a semaphore, show code from above and explaing critical region setup



**Semaphores:** En måde at blokere en “critical section” vha. En **Wait** og **Signal** funktion. Det er en smart måde at holde styr på hvor mange gange en sektion skal kunne tilgås. Semaphorens værdi er det antal der kan tilgå denne sektion, hvor hver tilgang reducerer Semaphorens værdi med 1 indtil koden er eksekveret, der aktiverer **Signal**, og derved lægger 1 til Semaphoren igen.

1. Ex5 & Ex5.3: **ISR** ved knap tryk ligges 100 til **counter** og venter 1 sek.  
   
2. Ex5\_semaMutex\_Jens: Bruger **1 semaphore** token til **t1** og **t2** funktionerne. De har samme prioritet ,men deler den samme token og kører derfor aldrig præcis samtidig . Men **t1** tager 5 sekog aflevere (Signal) token tilbage, **t2** tager 1 sek.

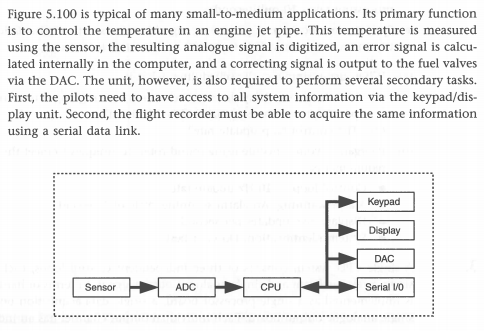
# #6: Embedded kernels III

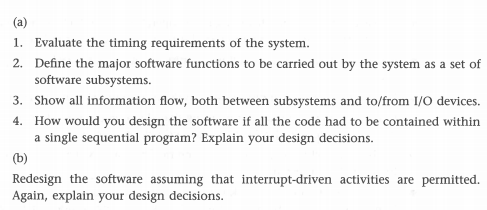
**QUESTION 4:** Show code for and explain message queues as a buffering tool.

1. Message Queue’s er til for at sende variabler fra funktionen **t2** til **t1** via et **BufferArray** på 10 pladser af 2 bytes
2. Der kan skabes over- og underflow ved at ændre k\_wait tiderne for **t1** og **t2**.

# #6.5 Deadlock

QUESTION 5: Exercise 1 in part3 on page295 (Pdf is on moodle)





**A)**

1. **Evaluate the timing requirements of the system.**

For at undgå at temperatur værdier både læses og skrives på samme tid, skal der være mutual exclusion, omkring dette data. Dette kunne evt. laves med en Semaphore.

1. **Define the major software functions to be carried out by the system as a set of software subsystems**

På CPU’en vil følgende software subsystems køre

* ReadTemp()
  1. Aflæser data fra ADC og konvertere det til en tempratur
* CalcError()
  1. Beregner som ud fra temp og brændstofdoseringen
* WriteFuel()
  1. Justere brændstof vha. DAC
* WriteLog()
  1. Skriver det nuværende Temp og brændstof output til display og Serial I/O

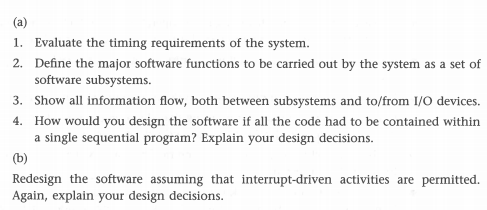
1. **Show all information flow, both between subsystems and to/from I/O devices**

ReadTemp -> ADC

WriteFuel -> DAC

WriteLog -> Keypad og Serial I/O

1. **How would you designe the software if all the code had to be contained within a single sequential program? Explain your design decisions.**
2. ReadTemp();
3. CalcError();
4. WriteFuel();
5. WriteLog();



**B)**

**Redesign the software assuming that interrupt-driven activities are permitted, again explain your design decisions.**

Der vil være nogle perioder/funktioner hvor det vil være nødvendigt at udelukke interrupts. F.eks. hvis der sker interrupt mens “**WriteFuel**” udføres, så kunne man risikere at motoren begynder at køre forkert grundet den har fået en forkert justering. “...*Ikke så smart i en F35 med 1000 km/t (eller ved mach 2).”*

Hvis nogle af disse interrupts vil have adgang til temperaturdata samtidig med “**ReadTemp**”, kan der opstå deadlocks. Man kan risikere at interruptet står og venter på noget data der er låst, dog bliver dette ikke låst op før interruptet er færdig.

Dette kan også ødelægge det beskyttede område, hvis en behandling i gang med det kritiske område, idet der forekommer et interrupt. Dette interrupt kunne have formålet at ændre værdien i det kritiske område, men fordi der var en funktion i gang, der læste værdien, bliver denne nu afbrudt of får en “mærkeligt” resultat. Vi har derfor ødelagt det kritiske områdes beskyttelse.

Mutual exclusion med flag... kan give deadlock

Mutual exclusion er nødvendig på temperatur data, således det ikke kan læses og skrives fra på samme tid. Her antages at dette udføres med flag-metoder.

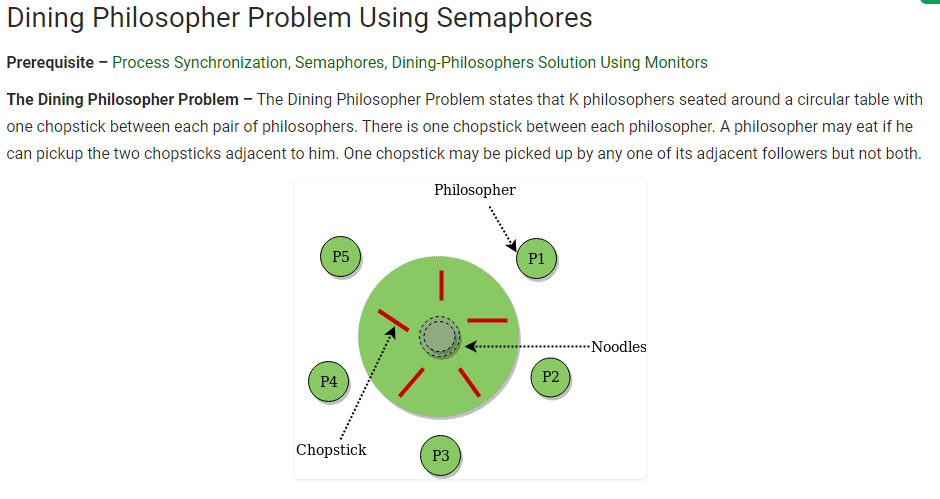
**Singel Flag:** Et enkelt flag der indikerer om resursen er i brug eller ej i brug. -> her kan mutual exclusion gå i stykker, hvis de begge går ind samtidig. Grundet der er et kort delay fra de får ind, til de hæver flaget.

**Turn flag:** Indikere hvis tur det er til at anvende dataen. Ideen er god, men lige så snart man får flere funktioner, man ønsker måske nogle er prioritet, nogle køre måske meget oftere end andre. Så kommer der unødvendig meget ventetid og problemer. Resultat det virker ikke i praksis.

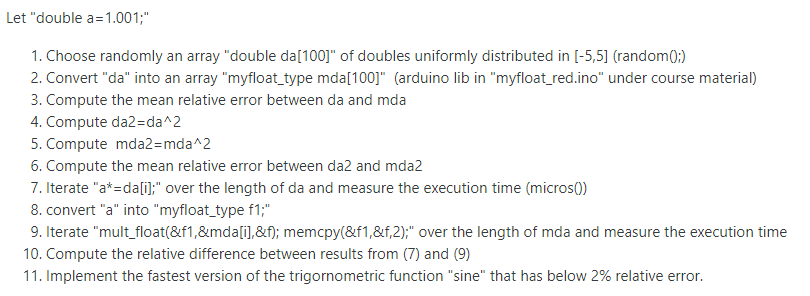
**Two flag:** De har hver deres flag. Et problem med two flags, er hvis 2 brugere, opererer med præcis samme hastighed og på præcist samme tid, så de begge hæver deres flag samtidig. Her ville de komme ud i en wait condition, som ville vente på den anden bruger, er færdig, dermed lede til Deadlock.

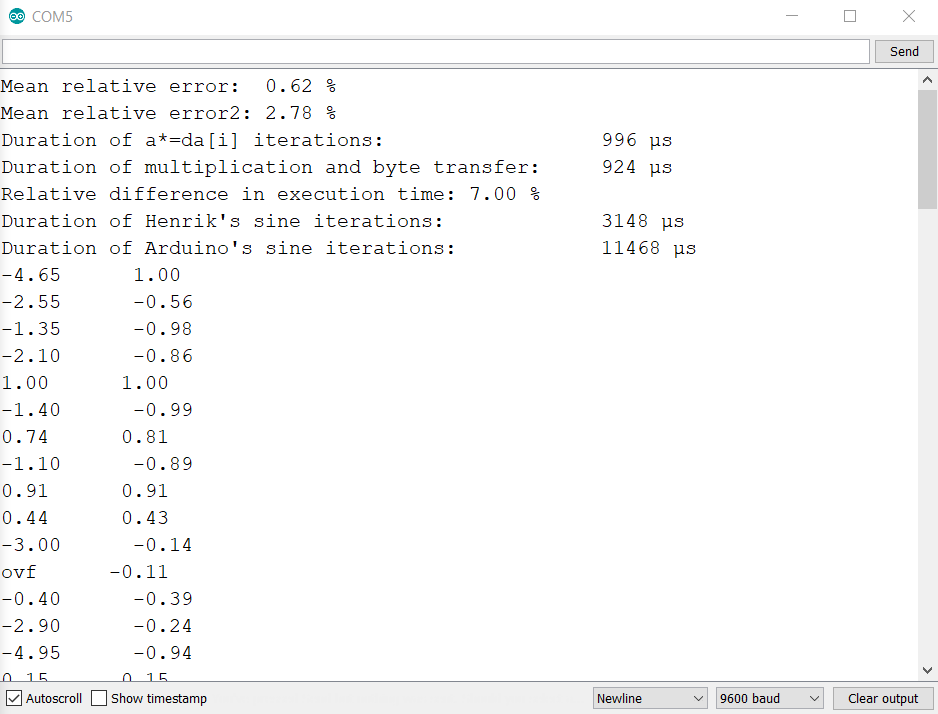
Deadlock:

Deadlock er således, 2 funktioner der afhængige af hinanden, men begge sidder i en uendelig løkke og venter på den anden bliver færdig. Da funktionerne først kan forsætte når den anden er færdig.

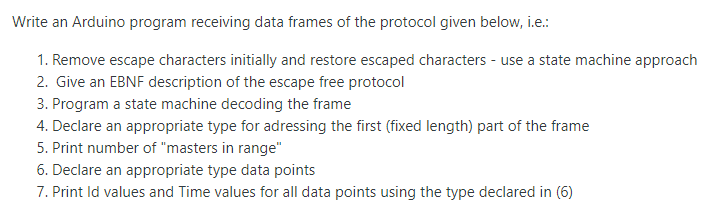


# #2: ex 1-11



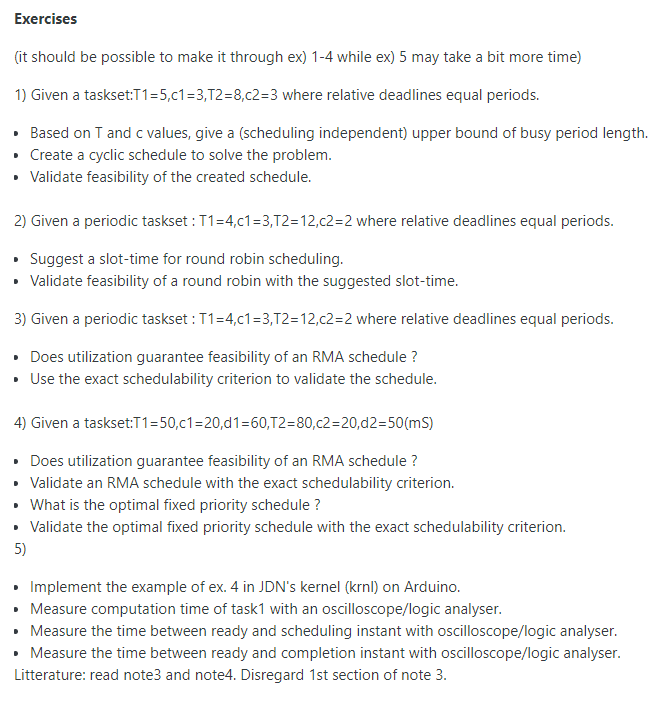


# #3: ex 4-7

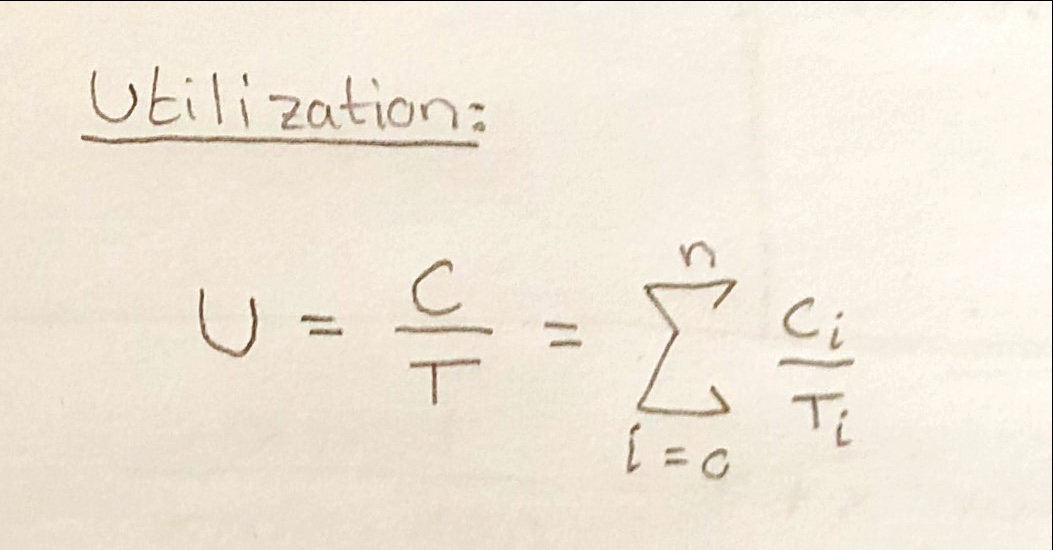


SPRUNGET OVER... Wtf

# #7: 4-5

2

4.1)   
RMA = Rate Monotonic Analysis/Scheduling  
Utilization determines a schedule’s stability, and is calculated using a taskset’s computation time and period.



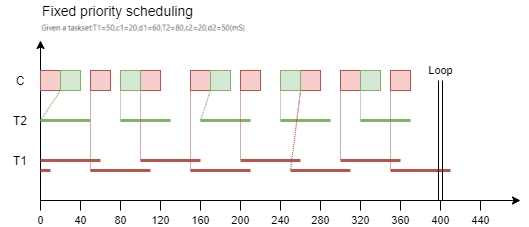
If U < 1 ---> Stable, causes **no** delay in task execution.

If U > 1 ---> Unstable, causes delay in execution.

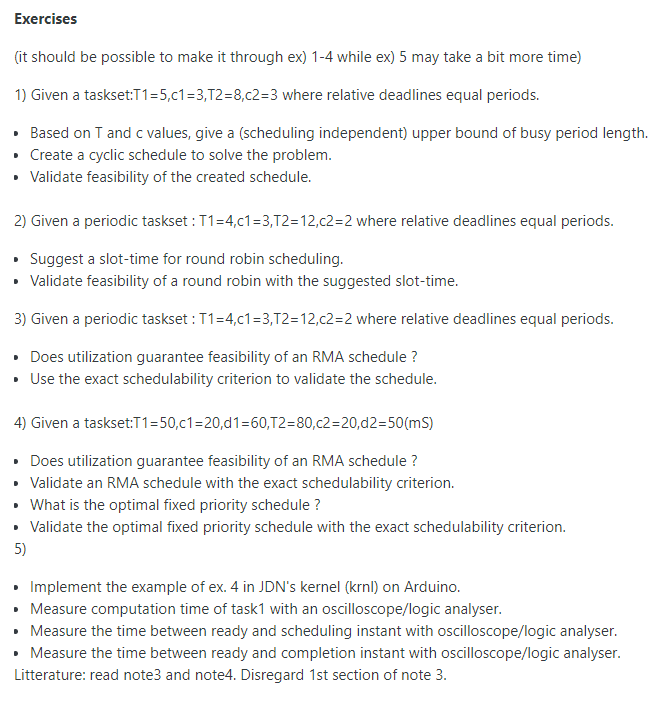
4.2)

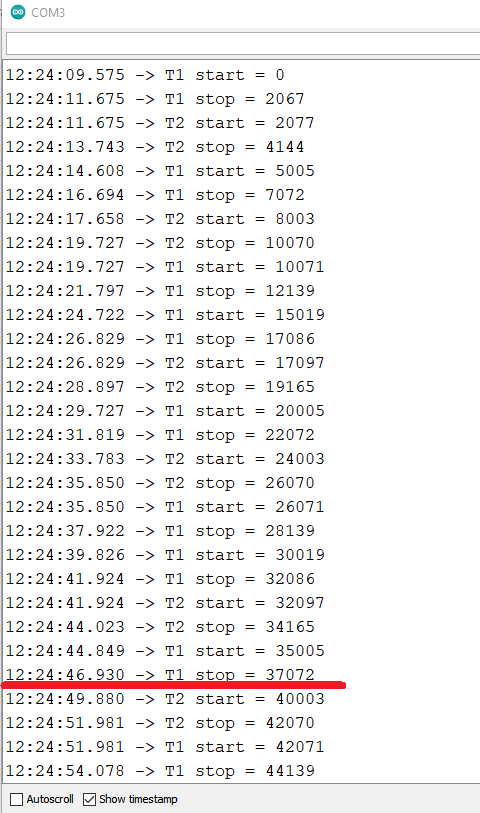


4.3+4)

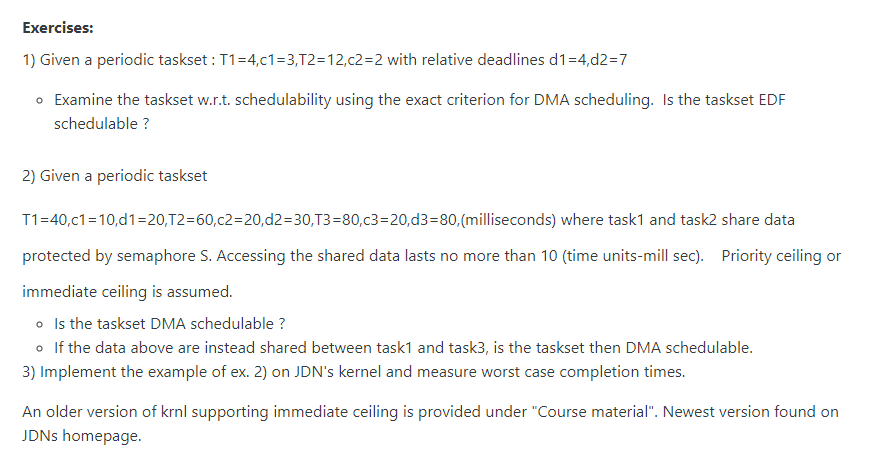


5.0)

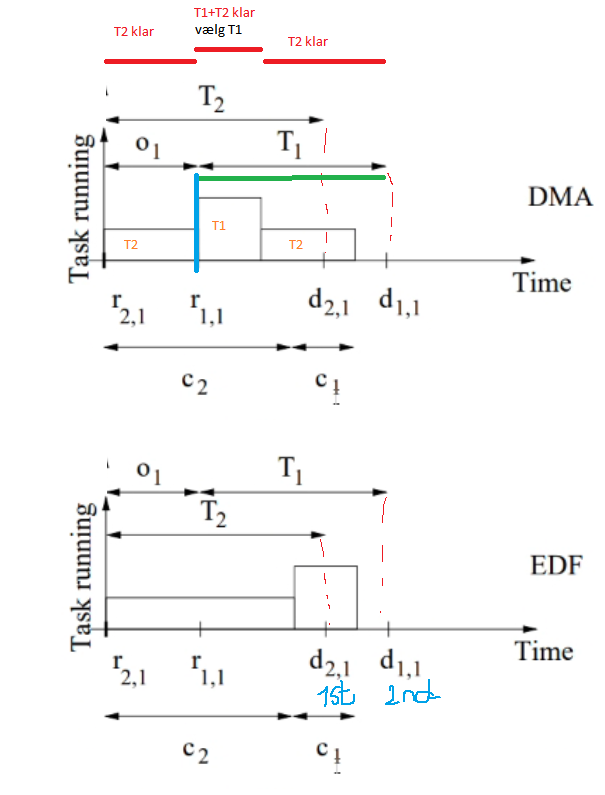




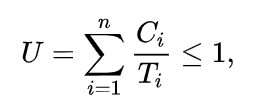
# #8: 1-2



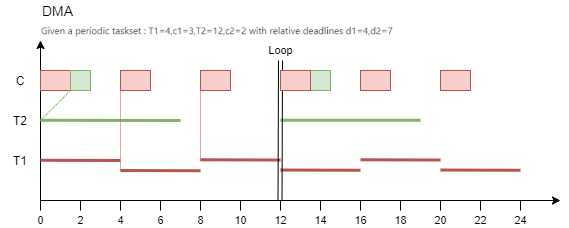
DMA = Deadline-monotonic scheduling



EDF = Earliest Deadline First

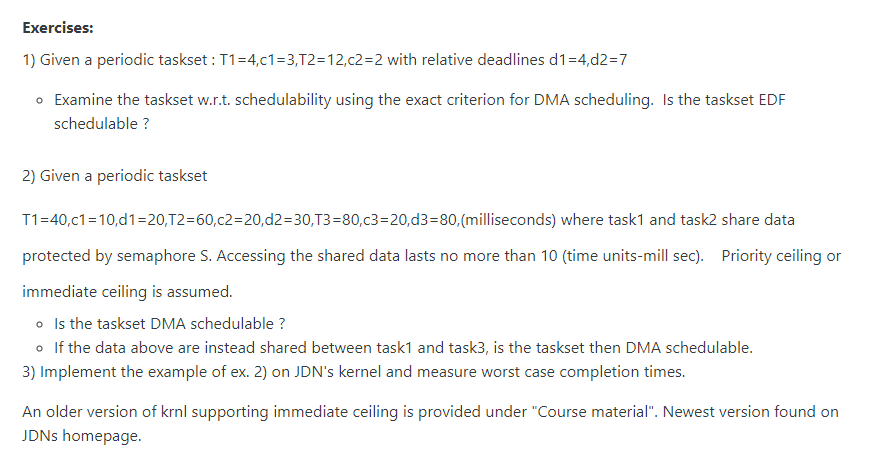


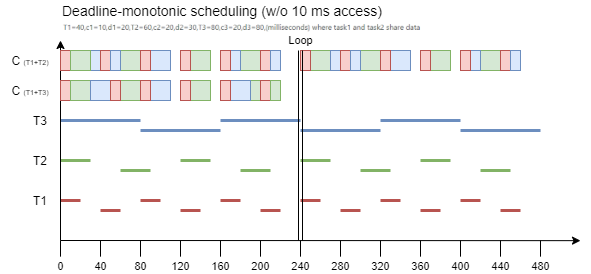
8.1)



* Ja EDF, da T1 har første deadline og korteste, så den er uændret.

8.2)





If the10ms is incl. in **c** value, the utilization calculation shows:



This means the DMA scheduling is possible!

8.2.1)  
Utilization calculated between the 3 tasks, where **task 1** and **task 2** has the added 10ms from accessing the shared data in the critical region.



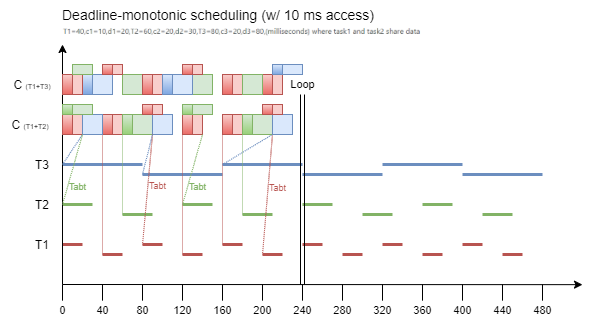
Calculation shows a **significant** instability in the scheduling, which results in delay that causes some tasks to never run at all. Thereby it results in a possible, but unstable DMA schedulable.

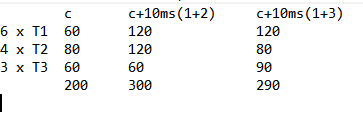
8.2.2)

Utilization calculated between the 3 tasks, where **task 1** and **task 3** has the added 10ms from accessing the shared data in the critical region.



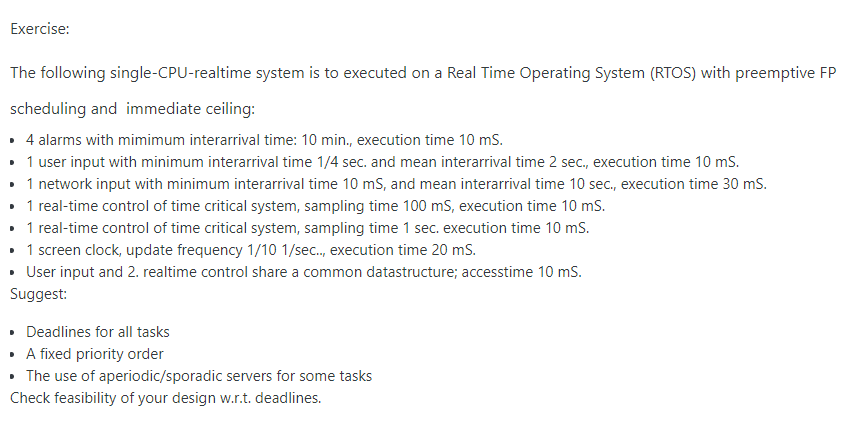
This also shows a significant instability, with the same arguments as the prior situation. The lower utilization, is assumed to be because of task 3 also having the lowest recurrence and therefore has a lower impact on the scheduling.

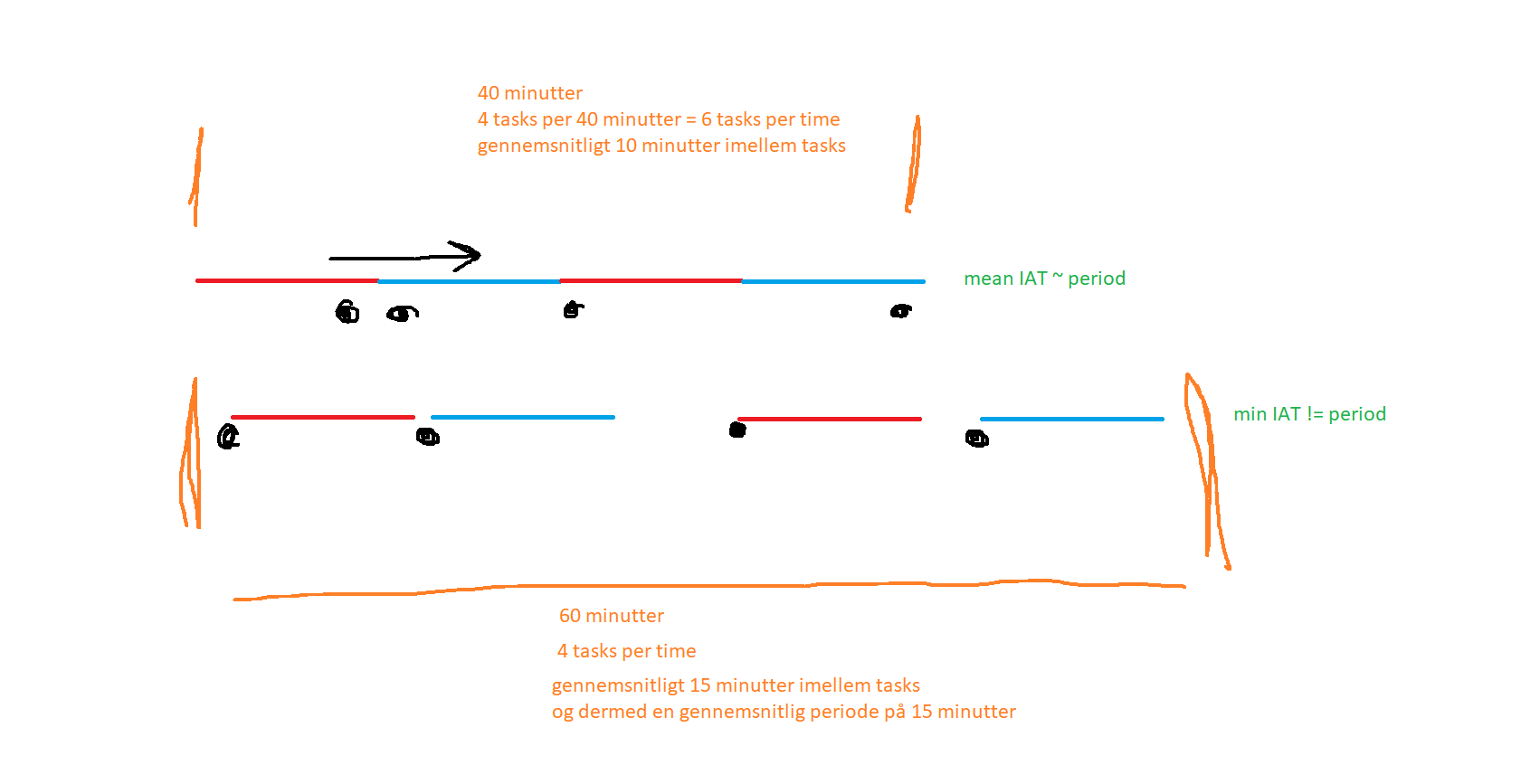




(ud af 240 til rådighed)

# #9: the entire exercise





**SE .PDF FOR GRAFIK**

Sporadic task

Task der har meget længer periode og deadline end computation time (Ti > ci og di > ci)

Interarrival time

Tiden I mellem der forekommer den pågældende hændelse. I en butik, kunne dette beskrive tiden I mellem at der ankommer en kunde.

Minimum:

Den tid, der mindst går I mellem at der kommer en kunde.

Mean:

Den gennemsnitlige tid, der går I mellem en kunde ankommer.