Exercice 1 en bare Métal

Mettre en place 3 clignoteurs indépendants :

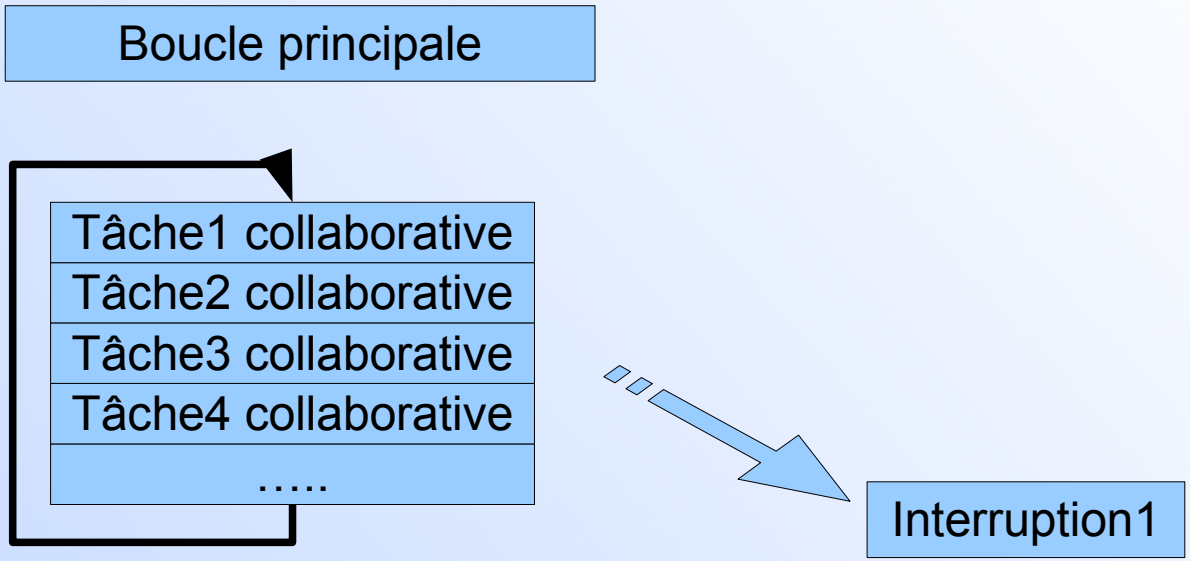
1 Rouge 1Hz (0,3s ON 0,7s OFF)

1 Vert 2,5Hz (0,1s ON 0,3s OFF)

1 Bleu qui commute entre 1, 2, 3, 4 Hz

à chaque appui d'un bouton scruté à 10Hz,

durée allumage 0,1s



aucune tâche n'est bloquante

Conception bare Métal

Le programme principale est une boucle while(1) dans laquelle :

– certaines taches vont être réalisées en interruption pour réduire le temps de réaction

– chaque pseudo tâche est une fonction parfois à exécution conditionnelle

```
while(1)
{  if(cond_lancement_tache1) {tache1();}
 tache2();
 if(semaphore_tache3) {tache3(); semaphore_tache3=0;}
}
```

– Le pire des cas pour le temps de réaction des taches de la boucle, est le plus grand temps de parcours de la boucle avec un maximum d'interruptions

CONTRAINTES d'implémentation:
– Un processeur ne peut faire qu'une chose à la fois, faire plusieurs choses à la fois revient à changer périodiquement de chose à faire sans trop tarder à reprendre.

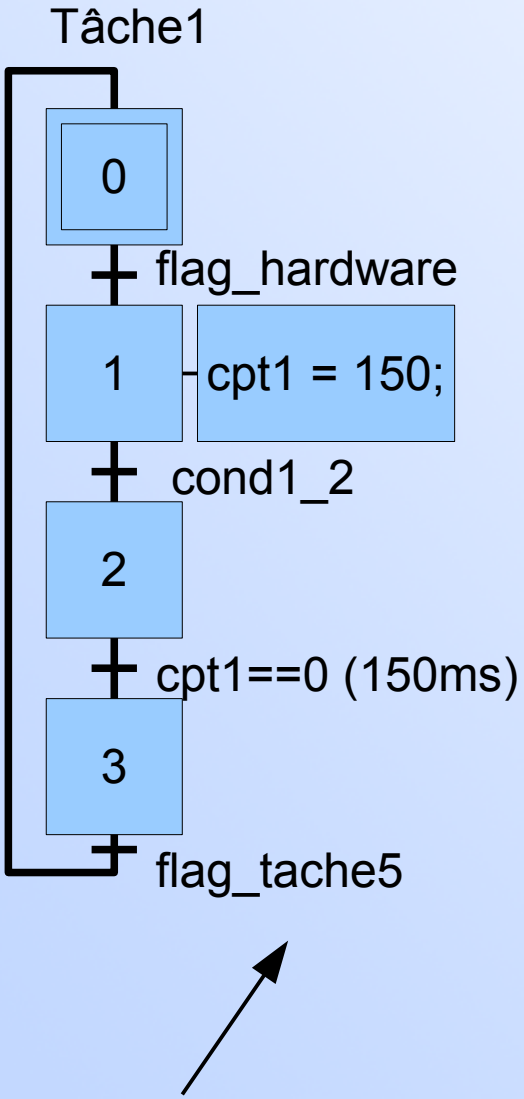
– Attendre étant bloquant, le concept doit être remplacé par :

un test de levée d'obstacle : soit on réalise maintenant soit on fera plus tard

– faire plus tard nécessite :
- mémoriser où on en est (variable d'état) pour pouvoir y revenir
- rendre la main
- avoir la garantie de récupérer la main à un moment
- refaire le test de levée d'obstacle dans des conditions identiques

Outils à mettre en place sur petits systèmes

Machines à états



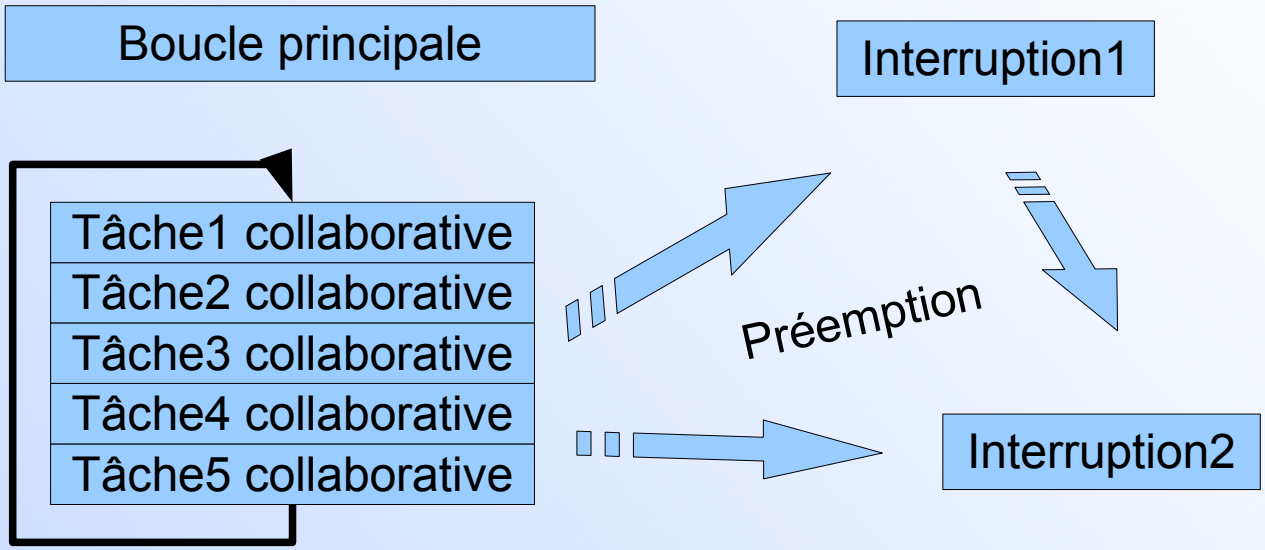
Sémaphores

IT_timer_tick

```
//acq IT
tick ++;
if (cpt1) cpt1--;
if (cpt2) cpt2--;
```

FIFOs

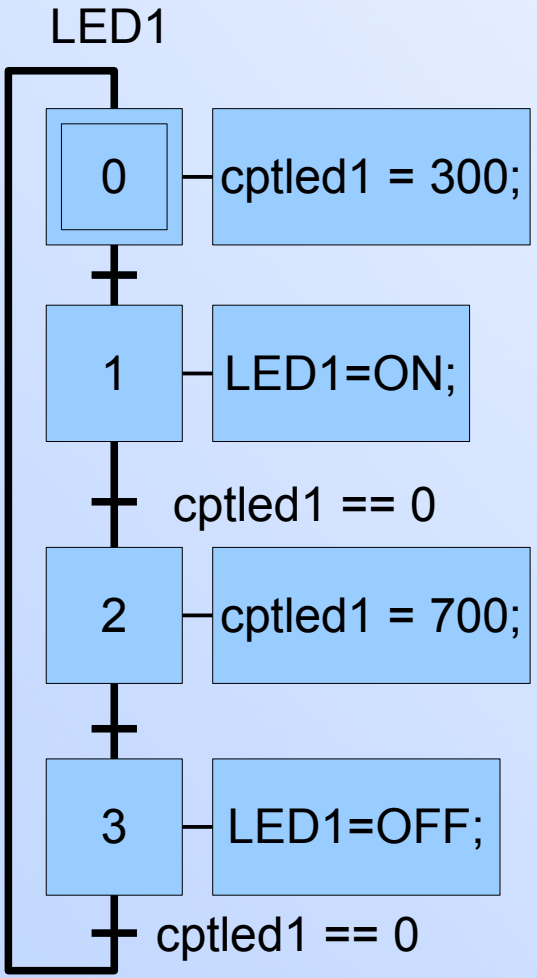
```
typedef unsigned char \
                    uchar ;
uchar FifoSend[16];
uchar pwFifo = 0 ;
uchar prFifo = 0 ;
uchar PlaceFifo = 16 ;
```

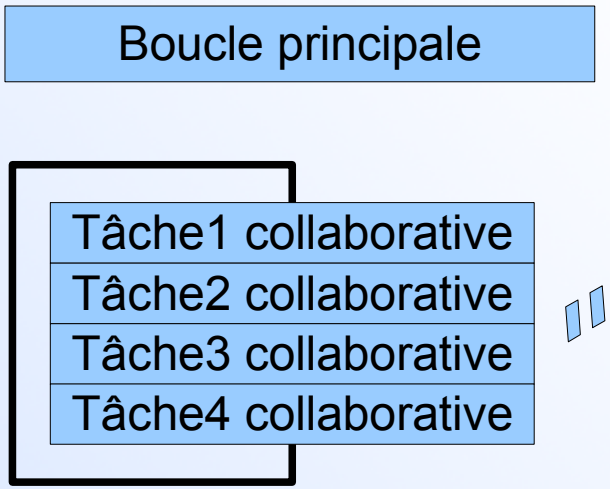


aucune tâche n'est bloquante

animation3leds + lecture périodique bouton

Machines à états

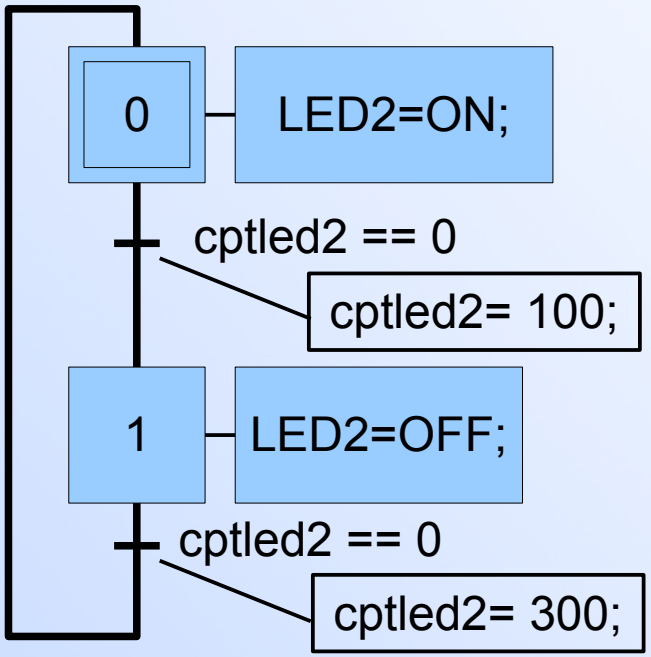




aucune tâche n'est bloquante

IT_timer_tick

```
//acq IT
tick ++;
if (cptled1) cptled1--;
if (cptled2) cptled2--;
if (cptled3) cptled3--;
if (!(--cptbouton))
{ flag_BP=1;
 cptbouton =100; }
```



avec action
au franchissement
de la condition de
changement d'état

code exemple

Chaque tâche est non bloquante:
Pas de délai dans les fonctions, usage d'une IT qui gère le timing pour tout le processeur
on remplacera le concept par un test de fin de délai, ce test est non bloquant :
Il faudra revenir dans la tâche à l'endroit où elle en est : découpage en blocs exécutés d'une traite et changement de bloc à l'aide d'un switch case

```
technique_interdite_gere_led()
{ allume_led();
  delai(300);
  eteint_led();
  delai(700);
}
```

```
int main (void)
 { init_proc();
   SysTick_Config(100);     /* Interruption chaque 1 ms */
   while (1)
    {       tache_cligneR();
            tache_cligneV();
            tache_cligneB();
            if(flag_BP) {flag_BP=0;tache_scrutation_bouton();}
     }
}
```

IT_timer_tick

```
acq_IT(); // it toutes les ms
tick ++;
if (cptledR) cptledR--;
if (cptledV) cptledV--;
if (cptledB) cptledB--;
if (!(--cptbouton))
{ flag_BP=1;
 cptbouton =100; }
```

Chaque tâche doit avoir une mémorisation de là où elle doit reprendre quand elle rend la main pour ne pas bloquer la collectivité: UTILISATION DE MACHINE A ETAT

code exemple

Chaque tâche est non bloquante:
Pas de délai dans la fonction,
on a remplacé le concept par un test de fin de délai,
ce test est non bloquant : Il faut revenir dans la tâche
à différents endroits du code via un switch case

IT_timer_tick

```
acq_IT(); // it toutes les ms
tick ++;
if (cptledR) cptledR--;
if (cptledV) cptledV--;
if (cptledB) cptledB--;
if (!(--cptbouton))
{ flag_BP=1;
 cptbouton =100; }
```

```
unsigned char mae_R = 0;
void tache_cligneR()
{switch(mae_R)
 {case 0:
    if(cptledR==0)  {mae_R=1;cptledR=700;actualise_ledR(0);}
  break;
  case 1:
    if(cptledR==0) {mae_R=0;cptledR=300;actualise_ledR(1);}
   break;
  }
}
```

minimisation du nombre d'états
avec action au franchissement
(quand le compteur atteint 0)

```
void tache_cligneV()
{   if(cptledV==0)
   {cptledV=(etat_ledR() ==0)?100:300;
    actualise_ledR( etat_ledR() ==0)?1:0);
   }
}
```

Ici, on ne semble plus avoir de machine à état,
elle existe pourtant implicitement :
c'est l'état de la led auquel on accède via la fonction
etat_ledR() qui donne l'étape de la tâche

Rappel: l'opérateur ternaire renvoie une valeur différente en fonction d'un test:
(test) ? (valeur_si_test_vrai) : (valeur_si_test_faux)
on peut alors affecter une variable ou un paramètre de fonction

code exemple

Chaque tâche est non bloquante:
Pas de délai dans la fonction,
on a remplacé le concept par un test de fin de délai,
ce test est non bloquant : Il faut revenir dans la tâche
à différents endroits du code via un switch case

IT_timer_tick

```
acq_IT(); // it toutes les ms
tick ++;
if (cptledR) cptledR--;
if (cptledV) cptledV--;
if (cptledB) cptledB--;
if (!(--cptbouton))
{ flag_BP=1;
 cptbouton =100; }
```

```
unsigned char mae_R = 0;
void tache_cligneR()
{switch(mae_R)
 {case 0:
    if(cptledR==0)  {mae_R=1;cptledR=700;actualise_ledR(0);}
  break;
  case 1:
    if(cptledR==0) {mae_R=0;cptledR=300;actualise_ledR(1);}
   break;
  }
}
```

minimisation du nombre d'états
avec action au franchissement
(quand le compteur atteint 0)

```
void tache_cligneV()
{   if(cptledV==0)
   {cptledV=(etat_ledR() ==0)?100:300;
    actualise_ledR( etat_ledR() ==0)?1:0);
   }
}
```

Ici, on ne semble plus avoir de machine à état,
elle existe pourtant implicitement :
c'est l'état de la led auquel on accède via la fonction
etat_ledR() qui donne l'étape de la tâche

Rappel: l'opérateur ternaire renvoie une valeur différente en fonction d'un test:
(test) ? (valeur_si_test_vrai) : (valeur_si_test_faux)
on peut alors affecter une variable ou un paramètre de fonction

code exemple

```
unsigned int reload_B= 900; // variable globale partagée
void tache_cligneB()
{static unsigned char mae_B=0;
  switch(mae_B)
  {case 0:    if(cptledB==0)
                {mae_B=1;cptledB=reload_B;actualise_ledB(0);}
   break;
   case 1:   if(cptledB==0)
                {mae_B=0;cptledB=100;actualise_ledB(1);}
   break;
 } }
```

IT_timer_tick

```
acq_IT(); // it toutes les ms
tick ++;
if (cptledR) cptledR--;
if (cptledV) cptledV--;
if (cptledB) cptledB--;
if (!(--cptbouton))
{ flag_BP=1;
 cptbouton =100; }
```

reload_B est une grandeur de réglage mise à jour dans une autre tâche
Ainsi le code est très simple, on l'écrit en postulant qu'une autre tâche a pour mission de mettre à jour le réglage:
La variable est nécessairement globale

```
void tache_scrutation_bouton()
{static unsigned char old_bouton=0;
 unsigned char bouton;
      bouton = ((LPC_GPIO0->FIOPIN)&(1<<10))?1:0;
  if(old_bouton && !bouton)  // test front descendant
    {switch (reload_B)
     {case 900: reload_B = 400;  break;//1hZ   période 900+100ms
      case 400: reload_B = 233;  break;//2Hz   période 400+100ms
      case 233: reload_B = 150;  break;//3 Hz  période 233 +100ms
      case 150: reload_B = 900;  break;//4Hz   période 150+100ms
    } }
  old_bouton=bouton;
}
```

Ici, on utilise des variables statiques, soit pour garder la mémoire de la machine à état, soit pour mémoriser l'état précédent d'un bouton pour créer un détecteur d'appui de bouton

Dernière optimisation : appeler les tâches seulement si leur attente est finie moins d'appel, et réduction du nombre d'état de MAE

```
unsigned char mae_R = 0;
void tache_ligneR()
{switch(mae_R)
 {case 0: mae_R=1;cptledR=700;actualise_ledR(0);  break;
  case 1: mae_R=0;cptledR=300;actualise_ledR(1);  break;
  }
}

int main (void)
 { init_proc();
   SysTick_Config(100);     /* Interruption chaque 1 ms */
   while (1)
    {        if(!cptledR)  tache_ligneR();
             if(!cptledV)  tache_ligneV();
             if(!cptledB)  tache_ligneB();
             if(flag_BP) {flag_BP=0;tache_scrutation_bouton();}
     }
}
```

IT_timer_tick

```
acq_IT(); // it toutes les ms
tick ++;
if (cptledR) cptledR--;
if (cptledV) cptledV--;
if (cptledB) cptledB--;
if (!(--cptbouton))
{ flag_BP=1;
 cptbouton =100; }
```

Attention : SE RAPPELER QUE LES VARIABLES de type SEMAPHORE (flag_BP) ainsi que les variables gérant les attentes doivent non seulement être globales mais aussi être déclarées en tant que volatile pour provoquer un véritable accès à la mémoire.

BILAN DES COMPETENCES A AVOIR

Décrire sous forme de GRAFCET

GRAphe Fonctionnel Commande Etapes Transitions

Cela fait apparaître naturellement :

– la nécessité de pluralité de tâches (plusieurs jetons nécessaires pour décrire le système)

– les notions d'attente

– les notions de synchronisation (avancer dépend de la position d'un autre jeton)

– les notions de boucles (durée de boucle, nombre de boucle)

– les notions de réglages (information disponible utilisée en lecture seulement et mise à jour en permanence par un autre code)

– une attente se fait dans une étape dédiée, la mémorisation du numéro d'étape permet de reprendre là où on en était

– la gestion du temps est gérée en interruption pour décompter les durées restantes

– Le découpage en étapes permet de repérer des fragments de code à executer d'un seul tenant, on peut les symboliser par une fonction dont le nom est explicite

Découpage en tâches et sous tâches (1)

– repérer les entités de base qui peuvent être réalisées d'un jet et en décrire les séquences:

c'est une sous tâche (case xxx:) d'une machine à état propre à la tâche

– repérer les évènements déclencheurs ou bloquants (attente évènement)

Utiliser un sémaphore(flag) hardware (bit de registre) qui est activé par l'évènement
Espionner un sémaphore sofware (variable globale) qui est activé par une autre tâche

– repérer les attentes temporelles bloquantes … durée que l'on attend cela va nécessiter un mécanisme pour les notions d'attente :

- Créer autant de variables globales qu'il y a d'attentes simultanées cpt_xx
- Utiliser une décrémentation si non nulle en IT : if(cpt_xx) cpt_xx--;
- Configurer la variable cpt_xx pour choisir une durée (jitter = durée d'une IT)

--- Avoir un état d'attente distinct pour chaque attente différente d'une tâche
+- Avoir un premier état de machine à état dédié à l'attente et une variable etat_futur,
case Attente : if (!cpt_xx) {mae_tache=etat_futur;} //etat_futur initialisé avant de basculer en état d'attente
++ Appeler la tâche que si l'attente est terminée, les étapes ne testent plus cpt_xx :
if(!cpt_xx) {tache_xx();} // évite de rentrer/sortir de la tâche pour tester cpt_xx

Découpage en tâches et sous tâches (2)

– Identifier les paramètres de réglage :

Créer une variable globale,
Exploitée en lecture seule dans une tâche sans se préoccuper de quoi que ce soit...
Mettre à jour la variable dans une autre tâche totalement indépendante

– repérer les actions répétitives :

– utiliser une boucle for seulement si la durée totale est compatible avec les latences tolérables des autres tâches
– fragmenter la boucle for en plusieurs morceaux si c'est possible et nécessaire, mettre chaque fragment de boucle dans des cases différents, en créant les variables statiques indispensables à passer des informations intermédiaires d'un bloc de calcul à l'autre :

```
tache_boucle_fractionnee()
{static unsigned char mae=0; static int accumule; int boucle
 switch(mae){
  case 0:  accumule =0;mae++;
    for(boucle=0; boucle < 10;boucle++) {accumule+=calcul(boucle);}
   break;
  case 1:  for(boucle=10; boucle < 20;boucle++) {accumule+=calcul(boucle);}
   transfert = accumule; flag_calcul_dispo =1;mae=0; break;
 }              }
```

Découpage en tâches : Priorités

– **identifier le temps de boucle du While(1) du main :**

quelle tâche possède la contrainte maximale de temps de réaction+ traitement ?

– **repérer les séquences trop longues** par rapport ce temps de boucle et les fragmenter....

pour intercaler plusieurs fois la tâche critique (beurk....)

– **repérer les actions critiques, nécessitant la plus faible latence :**

envisager de les gérer par IT (si temps de réaction faible)
envisager de les faire gérer par un périphérique nécessitant moins d'attention immédiate (ex capture d'un Timer)

Attention en cas d'ITs multiples se poser la question de la priorité entre ITs

– **identifier les flux à gérer et les risques d'écrasement de l'information précédente**

vérifier qu'on a le temps de récupérer, traiter, revenir avant l'arrivée du suivant
envisager d'utiliser un DMA pour du transfert automatique (économie d'IT)

Partage du temps d'exécution ordonnancement sur petite cible

Chaque tâche doit rendre régulièrement la main volontairement :

- Fragmenter une tâche longue en de multiples états d'un "switch () case … :
- Fragmenter une tâche longue en petites tâches qui se déclenchent en cascade
- La boucle du main est l'ordonnanceur des tâches de fond :

ATTENTION aux faux découpages, aux phénomènes de cascades directes
→ penser à ordonner à l'envers pour garantir une seule étape par boucle
AVOIR EN TÊTE le pire des cas pour savoir si toutes les tâches sont réalisables,
au besoin re-intercaler une même tâche dans la boucle.

Le processeur a la capacité de suspendre une tâche
en cours pour passer la main à une Interruption :

METTRE en Interruption les fonctionnalités les plus urgentes,
les plus contraignantes en terme d'échéances.
NE PAS METTRE en Interruption une tâche réalisable par scrutation,
dont l'échéance n'est pas une contrainte au niveau du temps de boucle

Limitation du modèle

Besoin de commutation de tâche

→ Limite à la fragmentation d'une tâche : compilateur nul => switch devient if ...else ...if ...else
Une machine à état comportant de nombreux cas doit être optimisée :
Table de goto avec des Enum consécutifs (compliqué à mettre en place en C)
Distribution par dichotomie (4 tests pour 16 cas, 5 pour 32 cas)
Regrouper les cas d'attente en un seul au début pour rendre la main vite:

```
case ATTENTE :  if(!cpt) {mae= mae_next;}  break;
case XXX : cpt = 10; mae_next= YYY; mae = ATTENTE; break;
case YYY : //la suite
```

Appeler la tâche si attente finie (plus de cas d'attente dans le switch)

→ Limite de consommation de temps CPU à scrupter trop de sémaphores

→ Répétition de code dans la boucle du main() pour gérer une tâche plus urgente

→ Un Os maison n'est pas compatible avec des librairies externes :
Les librairies pour le réseau, l'utilisation d'une mémoire avec FAT
sont souvent dédiées à un OS particulier

→ Périphériques ou developpements consommateurs de ressources et de temps CPU :
Ecran graphique
SDCard avec ou sans FAT
Réseau

→ Limitation des capacités du programmeur à évaluer le pire des cas