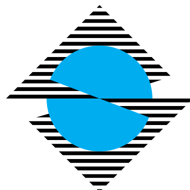


RTOS/Star12 UCOSII
Guide de mise en route

1 Sommaire

1. Sommaire - 1
2. Introduction - 1
3. Système temps réel - 1
   1. Multitâche 1
   2. Ordonnancement 2
   3. Coopération 2
   4. Evénements 3
   5. Préemption 3

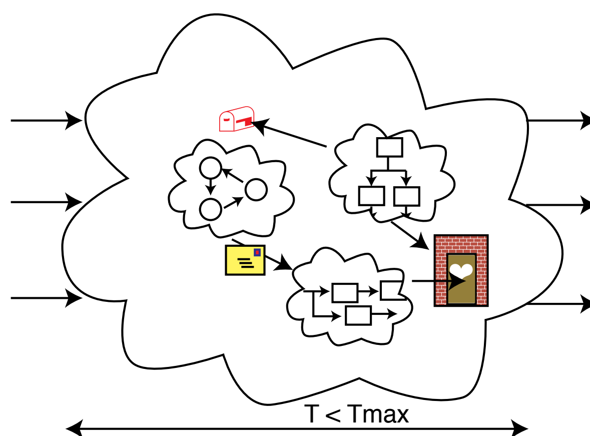
4. UCOSII - 4
1. Surveillance de l'OS 5
2. Démarrage 6
3. Générateur de code automatique 6
5. Services sur STAR12 - - - - - - - - - - - - - - - - - - - 8
1. RS232 8
2. LCD 9
3. Interruptions 10
4. Bus I2C 10
6. Difficultés liées aux RTOS - - - - - - - - - - - - - - - - - 11
1. Interbloquage 11
2. Inversion de priorité 11
7. Erreurs - 11
8. Historique des révisons de ce document - - - - - - - - - 12

2 Introduction

Ce document présente le concept de systèmes temps réel multitâche préemptif et donne les informations permettant de démarrer un développement sur la carte STAR12 avec le noyau temps réel UCOSII.

3 Système temps réel

Un système est dit temps réel ou réactif quand il est caractérisé par une latence (c'est à dire un temps de réponse) bornée. C'est là une définition académique, basée sur les contraintes imposées par les applications, qui pourrait aussi bien s'appliquer au déclenchement d'un airbag qu'aux prévisions météo. Dans un cas comme dans l'autre, si la réponse du système arrive trop tard, elle n'est plus d'aucun intérêt.



3-1 Multitâche

Un système d'exploitation multitâche *(angl. Multitask Operating System)* permet de partager l'exécution du processeur entre différentes tâches *(angl. tasks)*, donnant l'illusion que plusieurs programmes s'effectuent en parallèle. Le système d'exploitation découpe le temps en tranches *(angl. time slices)* distribuées entre chaque tâche. Ces tranches de temps sont suffisamment courtes, de l'ordre de la dizaine de millisecondes, pour qu'il semble à l'utilisateur que toutes les tâches s'exécutent simultanément et continûment.

On distingue deux classes de tâches : les *threads* (dont la traduction littérale en français « fil », n'est jamais employé) sont des tâches qui s'exécutent dans le même espace mémoire. Différents *threads* partagent donc les mêmes variables globales. Les *threads* sont compilés comme un même programme. A l'opposé, les processus *(angl. process)* est une tâche qui s'exécute dans un espace mémoire propre. Un processus est un programme qui tourne. Les processus sont compilés comme des programmes séparés et ne partagent pas leurs variables avec les autres processus. Dans les systèmes multitâches les plus complets, un processus peut contenir plusieurs *threads*.

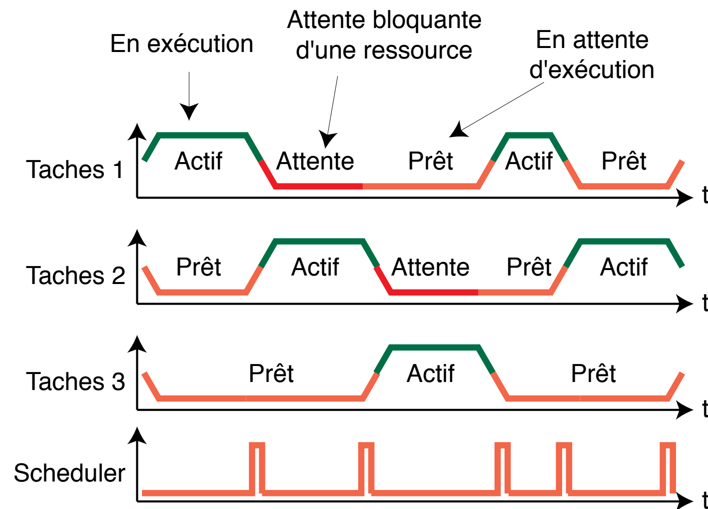
3-2 Ordonnancement

La stratégie employée pour choisir dans quel ordre les tâches vont pouvoir s'exécuter s'appelle l'ordonnancement *(angl. scheduling)*. Dans un système d'exploitation ordinaire, non-temps réel, l'ordonnanceur *(angl. scheduler)* est conçu pour optimiser le débit de calcul, de façon à pouvoir atteindre un taux d'utilisation du processeur proche de 100%.

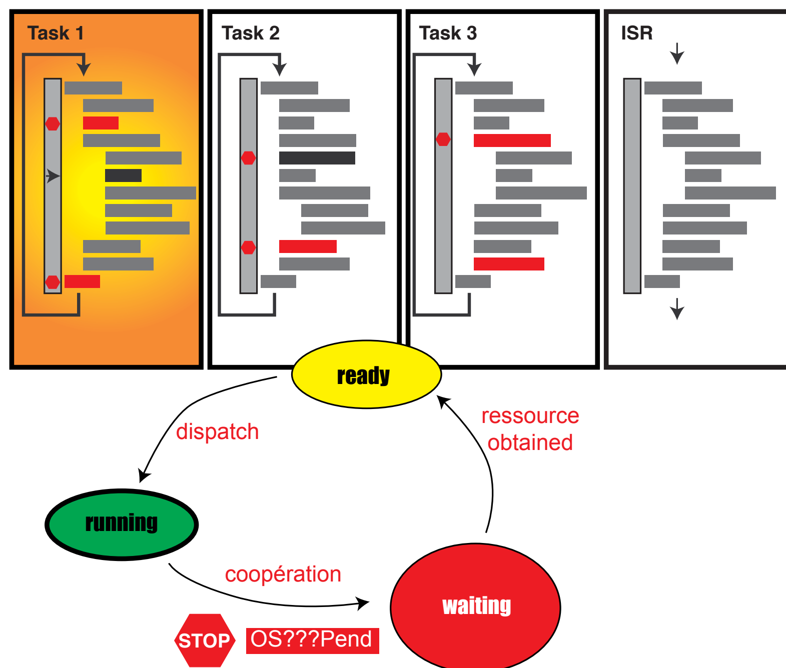
A l'opposé, dans un système d'exploitation temps réel *(angl. Réal Time Operationg System, RTOS)*, l'ordonnanceur est conçu pour minimiser la latence des taches les plus prioritaires et les rendre déterministes, prévisibles. Dans la pratique, cela n'est pas compatible avec un taux d'utilisation du processeur supérieur 70%. Pour qu'un système temps réel fonctionne correctement, il est préférable qu'il passe son temps à attendre des excitations. Pendant qu'il attend un évènement (généralement signalé par une interruption), le processeur exécute la tâche *idle*, correspondant à la priorité la plus basse.

3-3 Coopération

Dans un système embarqué, on peut généralement assimiler chaque tâche à une boucle infinie. **Pour que le partage du temps puisse opérer, les tâches ne doivent jamais exécuter d'opérations bloquantes telles que des scrutations. Au contraire, toutes les tâches doivent absolument faire appel à l'une des primitives du système d'exploitation qui permettent d'attendre un événement sans bloquer le système.** Lors de cet appel, si l'événement attendu est présent, l'exécution se poursuit normalement. Cependant, si l'événement n'est pas encore présent l'ordonnanceur met en attente la tâche en cours pour exécuter la tâche la plus prioritaire en attente d'un événement déjà survenu.



On parle alors d'un changement de contexte *(angl. context switching)* pour décrire le basculement d'une tâche à une autre. On introduit également une notion d'état pour chaque tâche, qui peut être soit en cours d'exécution *(angl. running)*, soit en attente d'un événement *(angl. waiting)*, soit prête à être exécutée *(angl. ready)*. On parle de **coopération** pour décrire le mécanisme qui permet à une tâche en cours d'exécution d'autoriser ainsi le basculement vers une autre tâche prête.

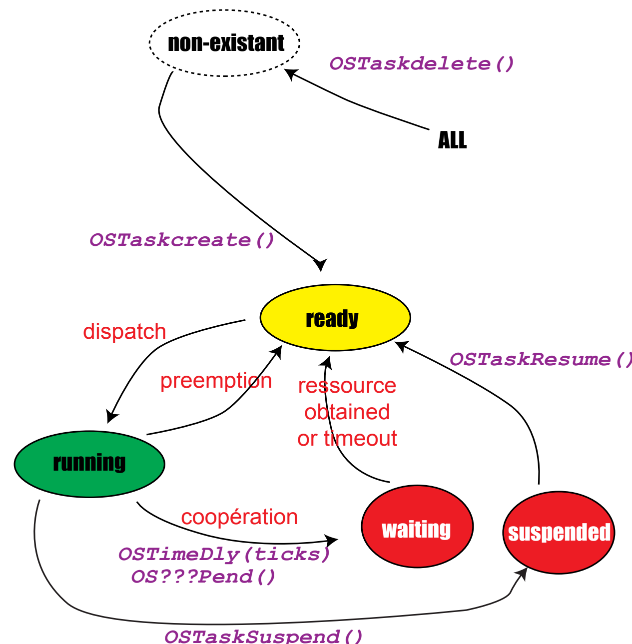


3-4 Evénements

Typiquement, les appels au système d'exploitation qui peuvent provoquer une coopération sont les fonctions de délai, les attentes de sémaphores, de boîte à lettre, de file d'attente de messages ou d'indicateurs binaires *(events)*. Un délai correspond à l'attente d'un certain nombre de *ticks*, unité de temps compté par les interruptions périodiques d'un timer. Un sémaphore est un compteur/décompteur. Une tâche peut attendre qu'un sémaphore devienne positif pour être réveillée. Ils sont souvent utilisés pour synchroniser des tâches (lorsqu'ils sont initialisés à zéro) ou pour garantir l'exclusion mutuelle (lorsqu'ils sont initialisés à un) lors de l'accès à une ressource partagée par plusieurs tâches. Une boîte à lettre est un objet de communication entre tâche. Il contient une donnée - ou plus souvent un pointeur sur une donnée - qu'une tâche peut attendre. La file d'attente de messages est simplement une extension du concept de boîte à lettre pouvant contenir plusieurs messages. Comme son nom l'indique, les messages y sont organisés en FIFO. La file d'attente est nécessaire si l'arrivée des message est sporadique, c'est à dire lorsque les messages sont suceptibles d'arriver par paquet. Les indicateurs *(event flags)* sont généralement attachés à une tache. Chaque tâche peut attendre une conjonction de plusieurs indicateurs, c'est à dire le ET ou le OU de plusieurs indicateurs. C'est généralement le seul moyen pour une tâche de se mettre en attente de plusieurs évènements.

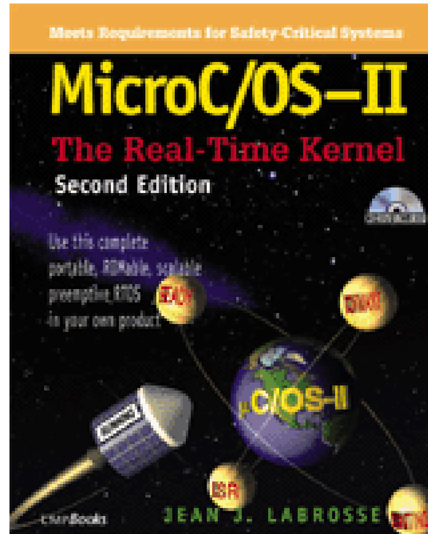
3-5 Préemption

Si le système d'exploitation le supporte, il est possible qu'une tâche prioritaire reprenne la main dès qu'une interruption la rend prête. On parle alors de **préemption**. Un système d'exploitation supportant la préemption est dit **préemptif**. Cela permet d'obtenir une latence beaucoup plus courte pour les tâches de plus hautes priorités. Ainsi, une tâche de haute priorité ne peut pas être bloquée par une tâche de priorité plus faible (sauf cas particuliers, cf § 6-1 Interbloquage page 18 et § 6-2 Inversion de priorité page 18). Notez que dans un OS préemptif, chaque tâche possède nécessairement sa propre pile. Les autres systèmes d'exploitation, dit coopératifs, sont moins réactifs puisqu'il est possible de monopoliser le processeur par des tâches de basses priorités.

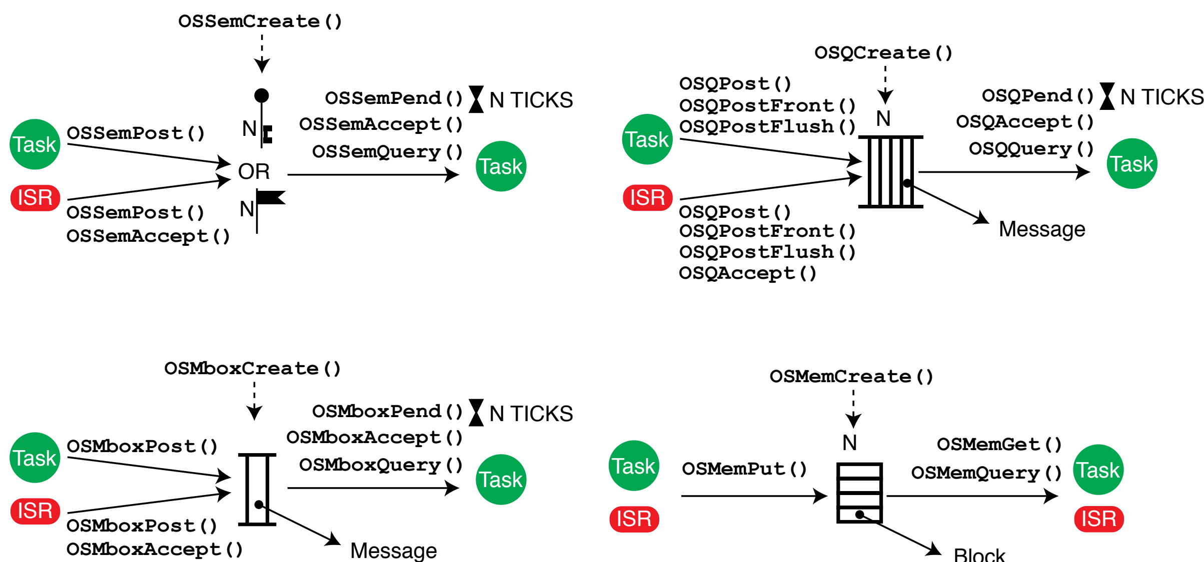


4 UCOSII

UCOSII (micro C OS version 2.0) est un noyau multitâche temps réel préemptif fourni en code source avec le livre le décrivant (cf. www.micrium.com).



Les tâches du noyau temps réel UCOSII sont des *threads*, il n'existe pas de notion de processus. Chaque tâche est identifiée par sa priorité comprise entre 63 (la plus basse) à 0 (la plus haute). Il n'est pas possible de donner la même priorité à deux tâches différentes. Il n'y a donc pas de *time slicing*, et la tâche exécutée est toujours la tâche prête la plus prioritaire. Les sémaphores, les boîtes à lettre et les files de messages sont supportés par UCOSII. Tous trois sont de type `OS_EVENT *`. Les évènements *(event flags)* en revanche ne le sont qu'à partir de la version 2.5 que nous ne possédons pas (il suffirait de racheter une version récente du livre). En outre, il existe des objets appelés *memory partitions*, qui permettent d'allouer dynamiquement des blocs de mémoire de taille fixe. Ils servent à remplacer la fonction `malloc()` qui n'est pas supportée et génèrerait une fragmentation du tas.



Chaque tâche doit contenir dans sa boucle au moins une occurrence de l'une des fonctions suivantes :

`OSTimeDly` (délai), `OSSemPend` (attente de sémaphore), `OSQPend` (attente de message dans une queue de message), `OSMboxPend` (attente d'un message dans une boîte à lettre). A défaut, les tâches de priorités moindres ne seraient jamais exécutées. L'attente peut être sans limite ou limitée par un *timeout* compté en *ticks*. De plus, **une tâche est une fonction qui ne retourne jamais**. Si la fonction associée à une tâche se termine, le retour de fonction pioche dans la pile plus profondément que le fond de la pile, ce qui provoque un plantage. Si la tâche doit être arretée, elle peut être supprimée par `OSTskDelete` ou suspendue `OSTaskSuspend`.

- Les fonctions `OSSemPost`, `OSMboxPost` et `OSQPost` permettent d'émettre vers un sémaphore, une boîte à lettres ou une queue de messages.
- Les fonctions `OSSemAccept`, `OSMboxAccept` et `OSQAccept` permettent d'obtenir, sans l'attendre, un sémaphore ou un message à condition qu'il soit immédiatement présent.
- Les fonctions `OSMemGet` et `OSMemPut` permettent d'obtenir et de rendre un bloc de mémoire.
- Les fonctions de la forme `OSXQuery` permettent d'obtenir une image de l'état d'un objet de UCOSII de type X dans une structure de données structurée de type `OS_X_DATA`.

4-1 Surveillance de l'OS

Deux points sont à surveiller particulièrement lors de l'exécution d'une application : le taux d'utilisation du processeur et l'occupation des différentes piles associées à chaque tâche. Pour réaliser une surveillance de l'état de la pile, la fonction de création de tâche `OSTaskCreateExt` initialise son contenu à zéro si elle est appelée avec `OS_TASK_OPT_STK_CHK|OS_TASK_OPT_STK_CLR` comme dernier argument. Il existe alors une fonction `OSTaskStkChk` qui parcourt la pile jusqu'à trouver une valeur non nulle pour déterminer la zone utilisée et la zone libre de la pile. Il convient de régler la taille des piles de façon à conserver un taux d'utilisation de l'ordre de 50 à 70%.

L'estimation du taux d'utilisation du processeur utilise deux tâches. La première est de toute façon présente, c'est la tâche idle `OSTaskIdle` portant la priorité 63, la plus basse. Elle ne fait rien d'autre que d'incrémenter une variable `OSIdleCtr`. La deuxième tâche, `OSTaskStat`, remet cette variable à zéro toutes les secondes et note au passage la valeur atteinte par le compteur `OSIdleCtrRun`. Au démarrage du système, seuls ces deux tâches s'exécutent pendant 2 secondes. La valeur maximale du compteur `OSIdleCtrMax` est alors relevée au bout d'une seconde. Par la suite, les autres tâches

s'exécutant, la durée d'exécution par seconde de la tâche idle diminue, ce qui se traduit par une diminution de la valeur `OSIdleCtrRun` atteinte par le compteur en une seconde. Le rapport entre ces deux valeurs permet à la tâche de statistique `OSTaskStat` d'estimer le taux d'utilisation du processeur.

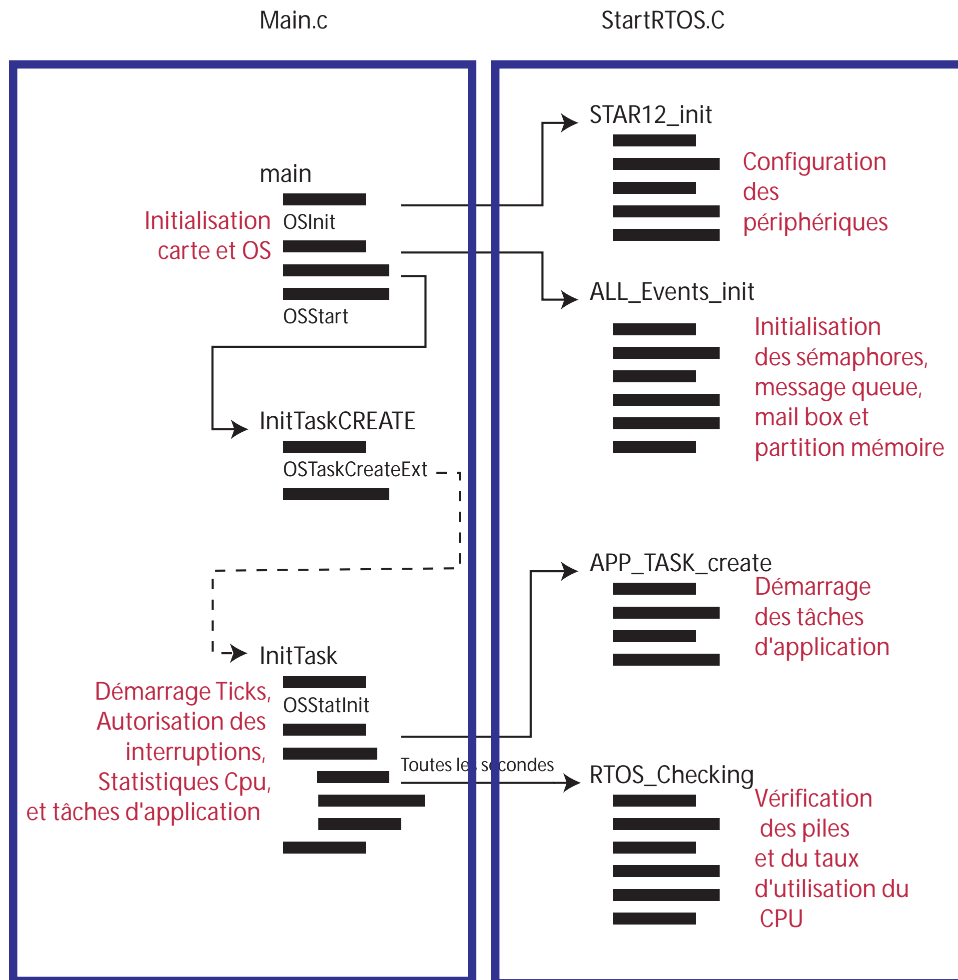
```
OSCPUUsage  = (INT8S) (100L - 100L * OSIdleCtrRun/ OSIdleCtrMax);
```

Il convient de garder un taux d'utilisation du processeur en dessous de 50%. Au-delà, il faut chercher des moyens pour diminuer la charge de calcul. On peut notamment éviter le recours au calcul flottant en utilisant une arithmétique à virgule fixe.

4-2 Démarrage

Une application tournant sous UCOSII démarre classiquement par l'exécution de la routine `startup` (dans le fichier `START12.C`) qui appelle la fonction main. La fonction main (dans le fichier `MAIN.C`) appelle successivement les fonctions

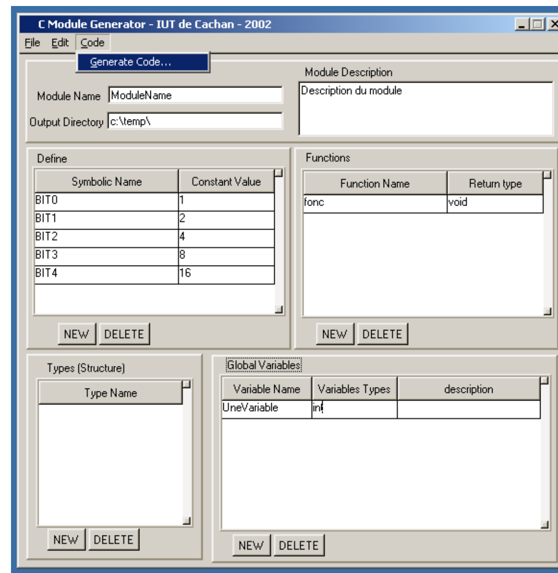
- d'initialisation du matériel `STAR12_init` (dans le fichier `STARTRTOS.C`),
- d'initialisation des tables et variables de l'OS `OSInit`,
- de création des évènements de l'application `ALL_EVENTS_init` (fichier `STARTRTOS.C`)
- de création de la tâche init (`InitTask` dans le fichier `MAIN.C`, priorité 62)
- de démarrage du multitâche par la fonction `OSStart`.



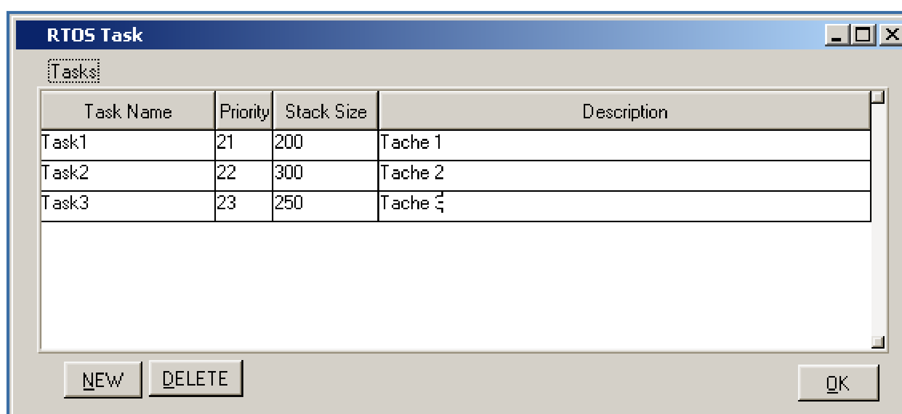
A cet instant, seuls les tâches `OSStart`, `OSTaskIdle` et `OSTaskStat` doivent exister de façon à obtenir un fonctionnement correct de la mesure du taux d'utilisation du processeur. La tâche `OSTaskStat` initialise le timer qui génère les interruptions correspondant aux ticks de l'OS, appelle la fonction `OSStatInit` d'initialisation de la mesure de taux d'utilisation du processeur puis appelle la fonction `APP_TASK_create` dans le fichier `STARTRTOS.C` qui lance enfin les différentes tâches de l'application. Finalement, la tâche init (`InitTask`) appelle toutes les secondes la fonction `RTOS_checking` qui contrôle l'état des piles des différentes tâches.

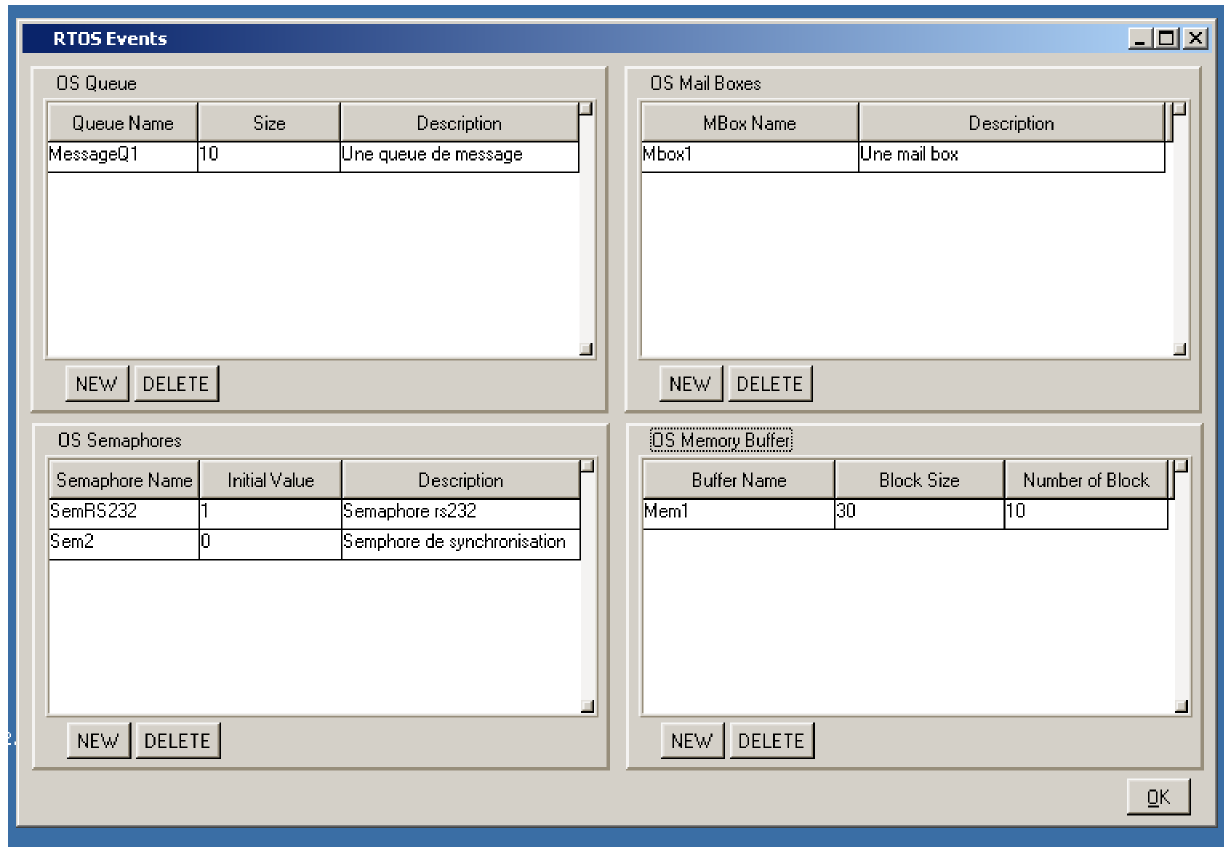
Ainsi, le fichier `MAIN.C` n'a pas à être modifié, seul le fichier `STARTRTOS.C` contient les fonctions à modifier en fonction de l'application.

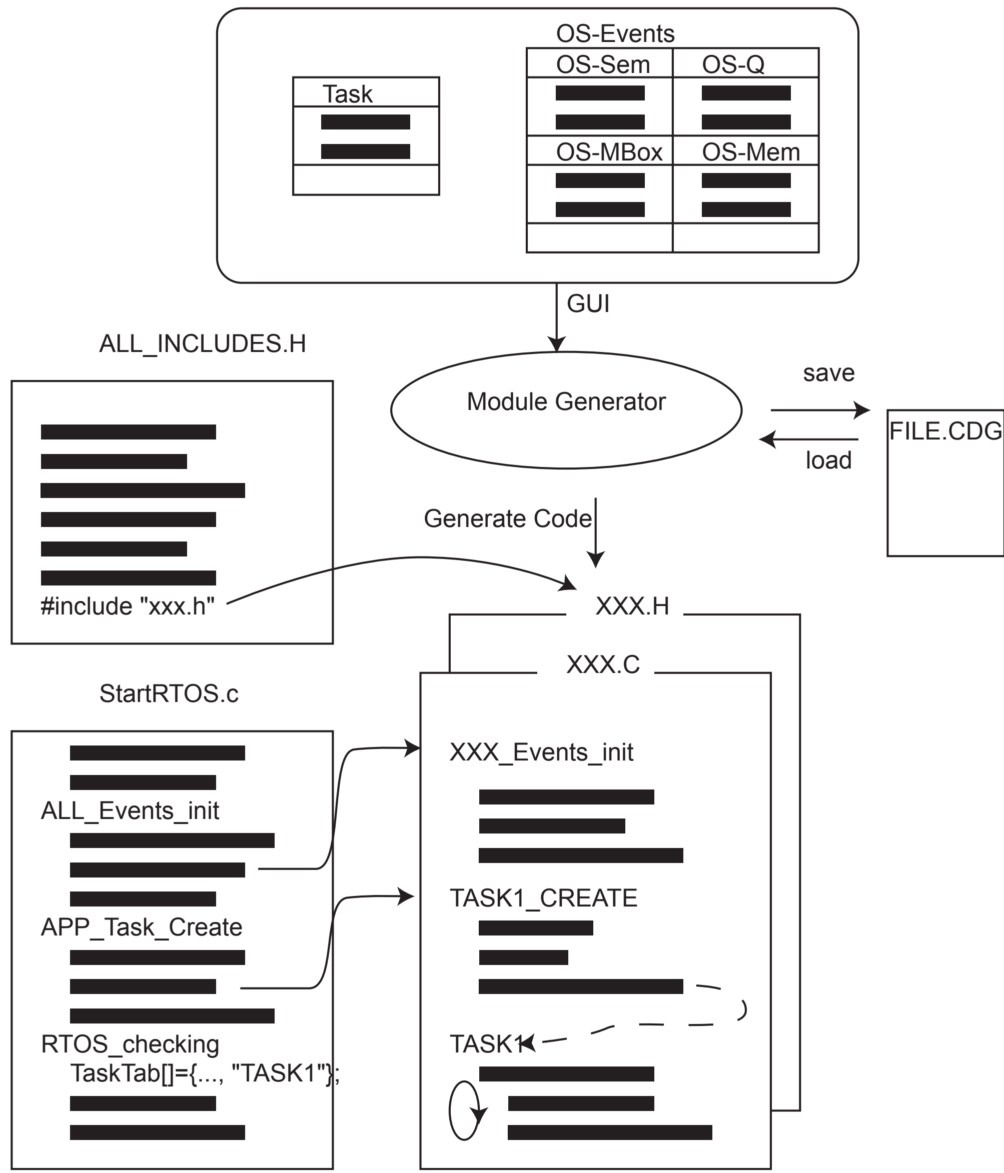
4-3 Générateur de code automatique



L'une des difficultés pour démarrer avec UCOSII consiste à utiliser proprement les fonctions de création et d'initialisation des différents types d'objets (tâches, sémaphore, etc.). J'ai écrit une application Labwindow-CVI qui simplifie cette phase du développement. On y inscrit dans une interface graphique les noms des objets à créer et leurs caractéristiques, puis un générateur de code créé deux fichiers `MODULENAME.C` `MODULENAME.H` contenant les fonctions de création de chaque tâche `taskname_CREATE`, un squelette pour chaque tâche `taskname` et une fonction d'initialisation de tous les évènements `modulename_Events_init`. Les deux fichiers ainsi créés ne sont absolument pas liés à un projet CodeWarrior. Il faut donc les ajouter à un projet, inclure le fichier `MODULENAME.H` dans le fichier `ALL_INCLUDES.H` et ajouter dans le fichier `StartRTOS.C` les appels aux fonctions `modulename_Events_init` dans la fonction `ALL_EVENTS_init` et `taskname_CREATE` dans la fonction `APP_TASK_create`.

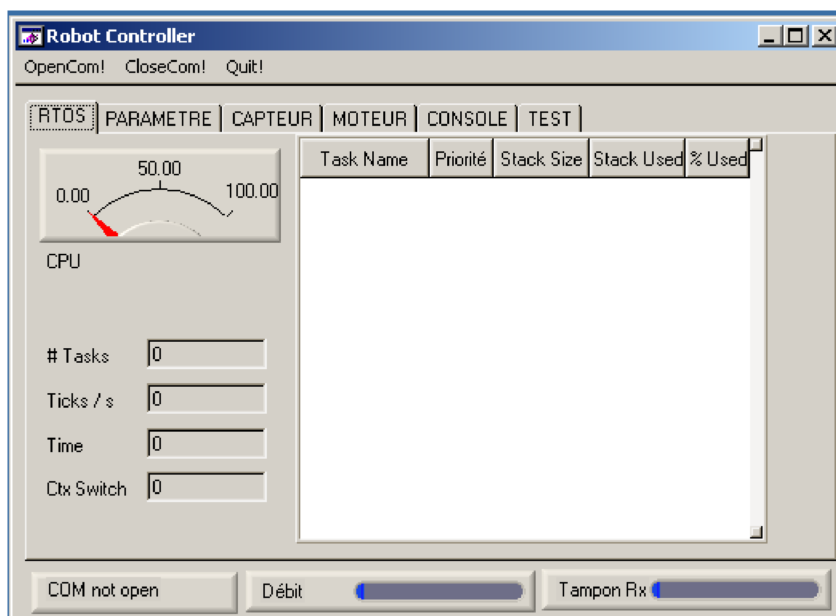



RTOS Events
OS Queue
Queue Name
Size
Description
MessageQ1
10
Une queue de message
NEW
DELETE
OS Mail Boxes
MBox Name
Description
Mbox1
Une mail box
NEW
DELETE
OS Semaphores
Semaphore Name
Initial Value
Description
SemRS232
1
Semaphore rs232
Sem2
0
Semphore de synchronisation
NEW
DELETE
OS Memory Buffer
Buffer Name
Block Size
Number of Block
Mem1
30
10
NEW
DELETE
OK


OS-Events
OS-Sem
OS-Q
Task
OS-MBox
OS-Mem
GUI
ALL_INCLUDES.H
Module Generator
save
load
FILE.CDG
Generate Code
#include "xxx.h"
XXX.H
XXX.C
StartRTOS.c
XXX_Events_init
ALL_Events_init
TASK1_CREATE
APP_Task_Create
TASK1
RTOS_checking
TaskTab[]={..., "TASK1"};

5 Services sur STAR12

5-1 RS232



Une tâche `RS232_BIN_TASK` est destinée à assurer les communications à travers la liaison série vers un programme labwindowsCVI sur un PC pour afficher des informations, notamment numériques, dans un format binaire ou recevoir des paramètres de configuration comme des commandes. La fonction

```
void RS232_BIN_send_frame (unsigned char id,unsigned char * data,unsigned int len)
```

envoie une trame binaire pointée par `data` de longueur `len` octets portant un identificateur de trame `id` vers le PC.

La trame est encodée pour pouvoir être récupéré sur le PC. Des marqueurs de début et de fin de trame sont ajoutés et les données binaires correspondant à ces marqueurs sont remplacées par des codes d'échappement. A la réception, les trames incomplètes sont éliminées et la fonction

```
void call_function (unsigned char id, unsigned char * data)
```

dans lefichier `OnTx.c` du projet labwindowsCVI trie les trames en fonction de leur identificateur `id` pour appeler une fonction d'affichage associée.

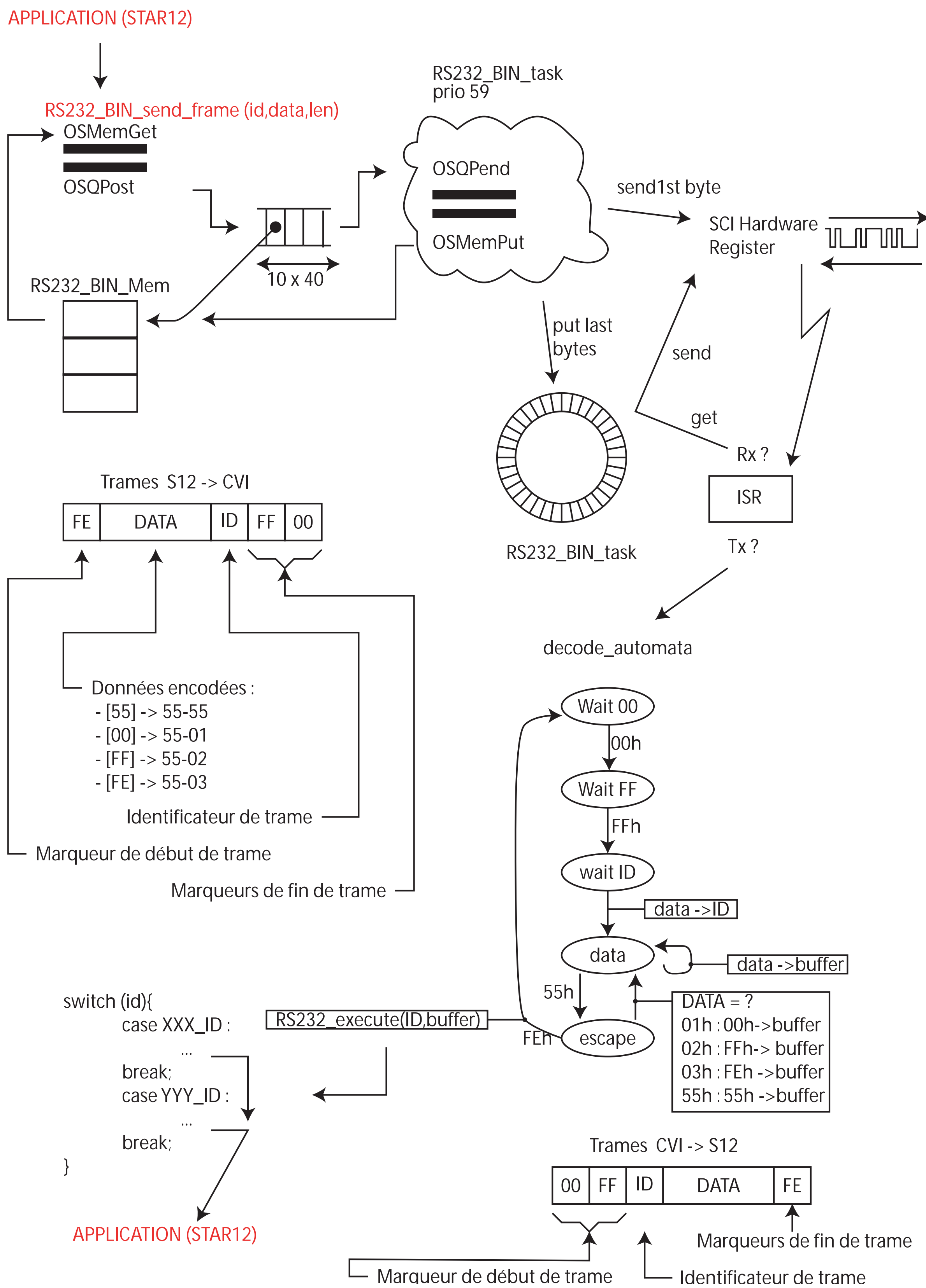
Pour les communications du PC vers la carte STAR12, les fonctions labwindowsCVI du fichier `OnTx.c` :

- `void RS232_bin_send_u8  (U8 address, U8  data)`
- `void RS232_bin_send_u16 (U8 address, U16 data)`
- `void RS232_bin_send_u32 (U8 address, U32 data)`

permettent d'envoyer des trames contenant une donnée codée sur 8, 16 ou 32 bits.

Les trames sont encodées, envoyées, reçues, décodées puis interprétées par la fonction

`void RS232_execute (unsigned char id, unsigned char * ptr)` dans le fichier `hc12_monitor.c`. Cette fonction trie les trames en fonction de leur identificateur `id` et exécute l'action associée. Notez que cette fonction est appelée par la routine d'interruption associée au port série et doit donc conserver un temps d'exécution très court. Typiquement, elle doit seulement mettre à jour une variable avec la valeur reçue.


APPLICATION (STAR12)
RS232_BIN_send_frame (id,data,len)
OSMemGet
OSQPost
RS232_BIN_task
prio 59
OSQPend
OSMemPut
10 x 40
RS232_BIN_Mem
send1st byte
SCI Hardware
Register
put last
bytes
send
get
Rx ?
ISR
Tx ?
RS232_BIN_task
decode_automata
Trames S12 -> CVI
FE
DATA
ID
FF
00
Données encodées :
- [55] -> 55-55
- [00] -> 55-01
- [FF] -> 55-02
- [FE] -> 55-03
Identificateur de trame
Marqueur de début de trame
Marqueurs de fin de trame
Wait 00
00h
Wait FF
FFh
wait ID
data ->ID
data
data ->buffer
55h
DATA = ?
01h : 00h->buffer
02h : FFh-> buffer
03h : FEh ->buffer
55h : 55h ->buffer
escape
FEh
switch (id){
case XXX_ID :
...
break;
case YYY_ID :
...
break;
}
RS232_execute(ID,buffer)
APPLICATION (STAR12)
Trames CVI -> S12
00
FF
ID
DATA
FE
Marqueurs de fin de trame
Marqueur de début de trame
Identificateur de trame

5-2 LCD

Le module LCD présent sur la carte STAR12 nécessite un timing précis pour son fonctionnement. La plupart des délais à insérer entre les instructions pausent un problème dans le contexte d'un RTOS. Ils sont le plus souvent de l'ordre de 100μs ce qui est trop court pour utiliser les ticks de l'OS (typiquement 5 ms) mais reste trop important pour bloquer l'OS avec des délais par scrutation d'un timer. La solution adoptée consiste à utiliser un tampon logiciel `lcd_buffer` qui est périodiquement recopié sur l'écran physique par une tâche de basse priorité (`LCD_task` de priorité 61). Les tâches de l'application n'accèdent qu'au tampon par l'intermédiaire des fonctions `LCD_clear_rtos`, `LCD_puts_yx`, `LCD_printf_yx`, alors que les accès bas niveau sont réservés à la tâche `LCD_task`. Seule la fonction `ErrorIf(condition, message)`, qui rapporte les erreurs, accède directement aux fonctions de bas niveau puisqu'elle arrête complètement le multitâche et rend donc la tâche `LCD_task` inopérante.

- `void LCD_clear_rtos(void)`, Efface le tampon logiciel lcd_buffer.
- `void LCD_puts_yx (y,y,const char * message)`, Copie une chaîne de caractère sur le tampon logiciel lcd_buffer, y représente l'indice de la ligne et x l'indice de colonne.
- `void LCD_printf_yx (y,x, const char * format, ...)`, Affichage formaté sur le tampon logiciel lcd_buffer, y représente l'indice de la ligne et x l'indice de colonne.
- `void ErrorIf(UINT8 condition, const char * ErrorString)`, si la condition est vraie, inhibe les interruptions, affiche le message directement sur le LCD et piège le programme dans une boucle infinie.

Pour fonctionner correctement, le service d'affichage sur l'écran LCD doit être initialisé par la fonction `LCD_init`. Les fonctions de bas niveau utilisent la fonction `TIMER_wait`, qui se sert du canal 0 du timer pour les fonctions de délais. Le timer doit donc être initialisé par la fonction `TIMER_initDefault`. **De plus, ce timer n'est plus disponible pour d'autres tâches et la fonction `TIMER_wait` ne doit pas être utilisée**.

5-3 Interruptions

Les vecteurs d'interruption non utilisés sont programmés dans le fichier `flash.prm` pour pointer sur une routine par défaut qui signale une erreur :

```
1    #pragma TRAP_PROC
2    void Default_isr(void){
3       ErrorIf(TRUE,"IT Vector Uninit");
4    }
```

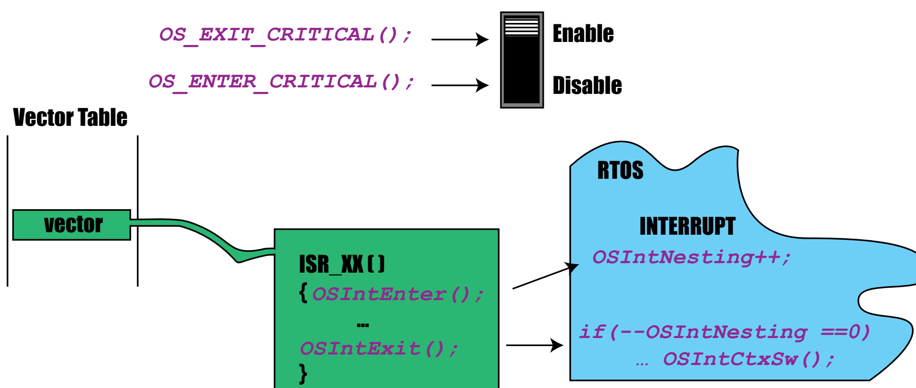
Lorsqu'on ajoute une routine d'interruption, il faut commenter la ligne correspondant dans le fichier `flash.prm` autrement le linker se plaint d'un recouvrement de section.

Les routines d'interruption qui utilisent les primitives du noyau doivent démarrer par la directive

```
OSIntEnter() ;
```

Et se terminer par l'instruction

```
OSIntExit() ;
```



Ce dernier appel peut, si c'est nécessaire, provoquer un changement de contexte. Dans ce cas, le pointeur de pile est manipulé par la routine assembleur `OSIntCtxSw` de façon à ce que l'instruction de retour d'interruption `RTI` ne ramène pas à la tâche interrompue mais à la tâche prête de plus haute priorité. La détermination de la tâche à relancer dans la fonction `OSIntExit` est très rapide. Elle utilise astucieusement un tableau de bits à double niveau où sont marquées les tâches prêtes et une tabulation de la fonction encodeur prioritaire qui évite tous parcours séquentiel de table.

5-4 Bus I2C

Les fonctions d'accès au bus I2C à utiliser dans le contexte d'un RTOS doivent être non bloquantes. Les fonctions `I2C_IT_MasterTx`, `I2C_IT_MasterRx`, `I2C_IT_SlaveTx` et `I2C_IT_SlaveRx` réalisent les communications à l'aide de la routine de service d'interruption I2C_isr, cœur des communications I2C. L'initialisation du contrôleur I2C doit êre effectuée par la fonction `I2C_IT_init`. A ce jour, le fonctionnement en mode maître a été testé, en revanche, le mode esclave ne l'a pas été et requiert a priori des modifications.

6 Difficultés liées aux RTOS

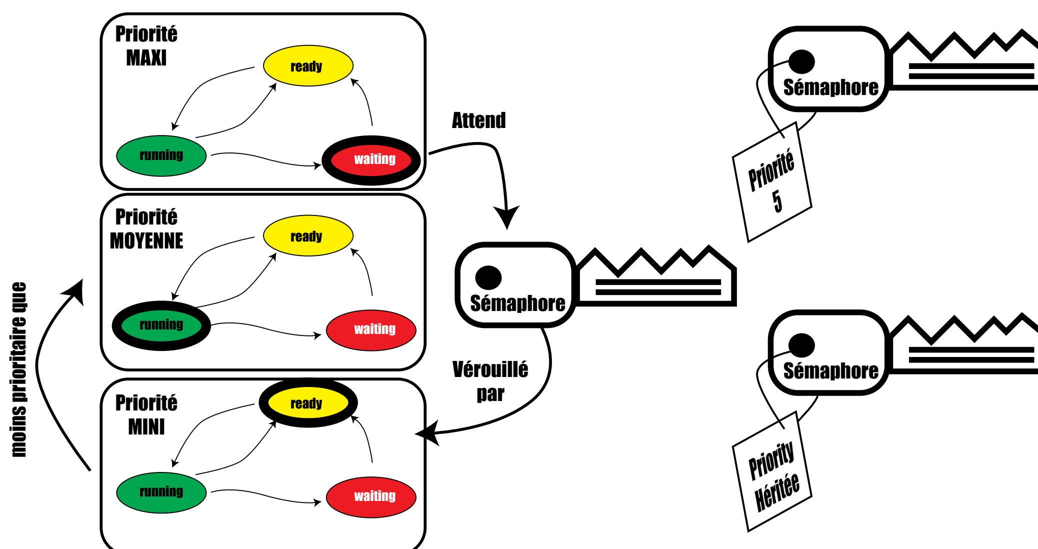
6-1 Interbloquage

Lorsque plusieurs ressources sont partagées par plusieurs tâches, il est utile de protéger l'accès à chaque ressource par un sémaphore de façon à garantir l'exclusion mutuelle. Dans ce cas, il faut définir un ordre pour la demande des sémaphores. A défaut, un interblocage *(angl. deadlock)* peut se produire. La situation est la suivante. Une tâche T1 demande et obtient un sémaphore S1 puis une tâche T2 demande et obtient un sémaphore S2. Si la tâche T1 demande alors le sémaphore S2 elle ne peut pas l'obtenir puisqu'il est occupé par la tâche T2. Si de plus, la tâche T2 demande le sémaphore S1, elle ne peut pas l'obtenir tant qu'il est occupé par la tâche T1. Dans cette situation, les deux tâches sont en attente. Elle ne s'exécute plus et ne peuvent pas libérer leur sémaphore. La situation est donc définitivement bloquée pour l'une comme pour l'autre. On parle parfois d'*étreinte fatale*.

6-2 Inversion de priorité

A rédiger

L'inversion de priorité correspond à une situation dans laquelle une tâche prioritaire est bloquée par une tâche de basse priorité qui détient le sémaphore qu'elle réclame. Si une tâche de niveau intermédiare est prète, c'est elle qui s'exécute, empéchant la tâche de basse proirité de relacher son sémaphore, ce qui bloque la tâche prioritaire. Il existe deux moyen pour éviter cette situation. La première consiste à associer une priorité à chaque sémaphore. Toute tâche qui détient ce sémaphore aura sa priorité au moins égale à celle du sémaphore. Une autre solution consiste à faire hériter à la tâche qui détient le sémaphore la priorité de la tâche la plus prioritaire qui réclame ce sémaphore. On parle alors dd'hitage de priorité. Aucune de ces options n'est disponible dans notre version de UCOSII.



7 Erreurs

- **Interface bus CAN** : A ce jour, La communication à travers le bus CAN sous UCOSII est en cours de développement par le Module 3 RLI.
- **Interface I2C :** La routine d'interption I2C mériterait d'être recodé sous forme d'automate pour être plus claire. A ce jour, le comportement en mode escalve n'a pas été abordée.
- **Section critiques :** Les sections critiques n'utilisent pas la pile et ne peuvent donc pas être imbriquées.

8 Historique des révisons de ce document

Date	Auteur	Modifications
23 mars 2003	J.- O. Klein	Version 0.1 : Document initial.
13 juin 2003	J.- O. Klein	révision 0.2 : Ajoutée figure RS232_BIN, augmentées tailles figures