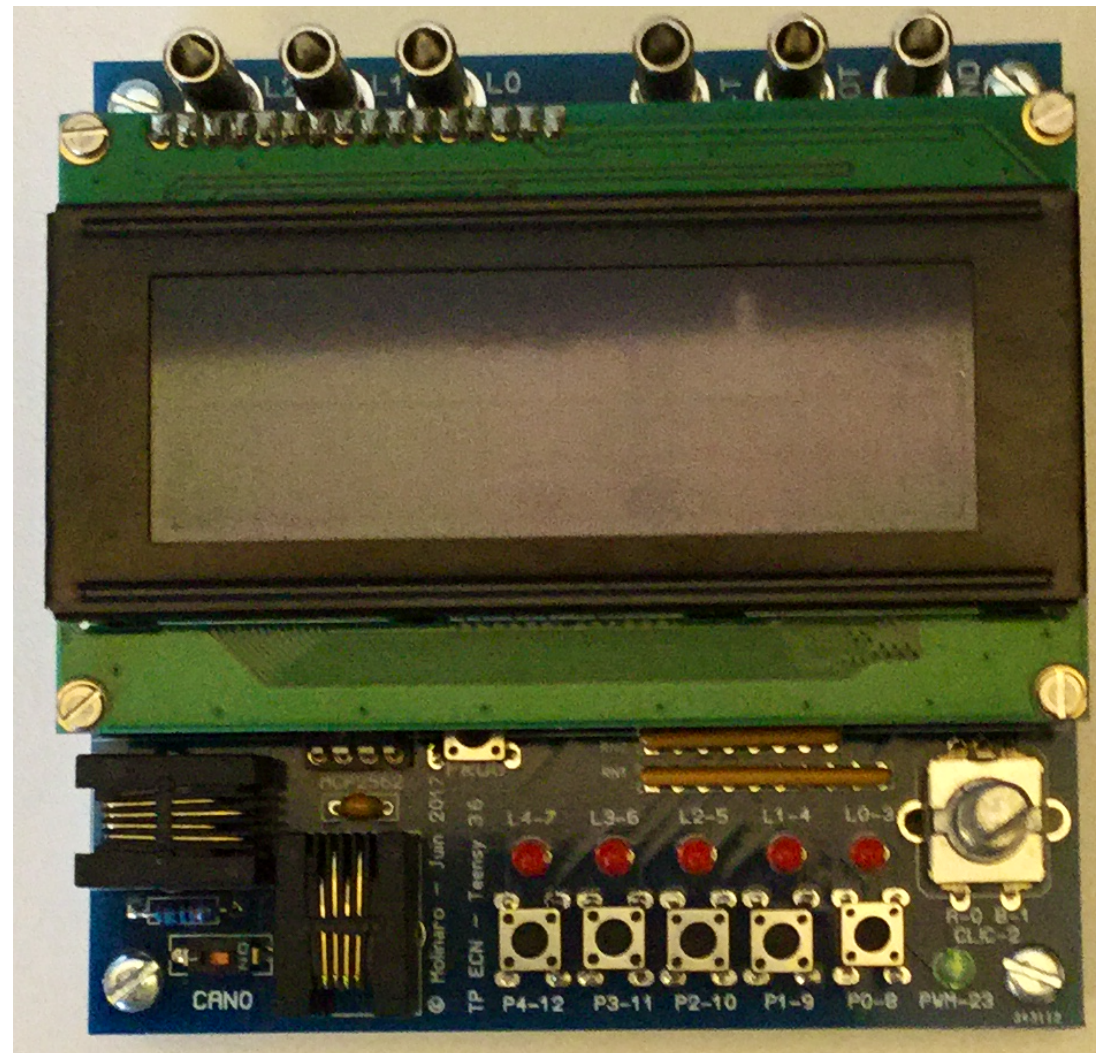


Temps Réel



Programme *16-synchronization-tools--wait-until*

Description de cette étape

Nous allons étendre les outils de synchronisation avec la possibilité d'effectuer une attente jusqu'à une échéance. C'est une première extension, moins générale que les *commandes gardées*.

Ceci impose de faire des modifications importantes dans l'exécutif, que nous allons détailler dans les pages qui suivent.

Dupliquer le projet de l'étape précédente et renommez-le par exemple **16-synchronization+wait-until**.

Primitive P_until

C'est un exemple typique de ce que l'on veut écrire. La primitive **P_until** effectue une attente jusqu'à une échéance. C'est une méthode qui sera rajoutée à la classe **Semaphore**, et dont l'appel s'écrit :

```
bool r = s.P_until (MODE_ échéance) ;
```

Le fonctionnement est le suivant :

- si, au moment de l'appel le sémaphore est strictement positif, le sémaphore est décrémenté, et la primitive retourne immédiatement avec la valeur **true** ;
- si, au moment de l'appel le sémaphore est nul, et l'échéance est atteinte ou dépassée, et la primitive retourne immédiatement avec la valeur **false** ;
- sinon, la primitive est bloquante, la tâche est en attente sur le sémaphore et l'échéance, jusqu'à ce que :
 - ▶ la tâche est débloquée suite à l'invocation de **V** par une autre tâche ; la valeur **true** est retournée ;
 - ▶ la tâche est débloquée parce que l'échéance est atteinte ; la valeur **false** est retournée.

Dans tous les cas, la valeur de retour indique l'évènement qui a permis de passer la primitive :

- **true** si c'est le sémaphore ;
- **false** si c'est l'échéance.

Modifications à apporter à l'exécutif

Modifications à apporter à l'exécutif

Maintenant, une tâche peut se retrouver bloquée dans deux listes :

- la liste des tâches bloquées de l'outil de synchronisation ;
- la liste des tâches bloquées sur échéance.

Lors du déblocage par l'outil de synchronisation, il faut :

- enlever la tâche la liste des tâches bloquées sur échéance ;
- retourner la valeur **true**.

Lors du déblocage par l'atteinte de l'échéance, il faut :

- enlever la tâche la liste des tâches bloquées sur l'outil de synchronisation ;
- retourner la valeur **false**.

La valeur booléenne retournée est stockée temporairement dans un nouveau champ du descripteur de tâche, **mUserResult**.

Descripteur de tâches

Ajouter les champs **mBlockingList** et **mUserResult** au descripteur de tâche (fichier **xtr.cpp**) :

```
typedef struct TaskControlBlock {  
    //--- Context buffer  
    TaskContext mTaskContext ; // SHOULD BE THE FIRST FIELD  
    //--- This field is used for deadline  
    uint32_t mDeadline ;  
    //--- Task blocking list (nullptr if task is not blocked)  
    TaskList * mBlockingList ;  
    //--- Task index  
    uint8_t mTaskIndex ;  
    //--- User result  
    bool mUserResult ;  
} TaskControlBlock ;
```

Par défaut, ces champs sont initialisés par des zéros binaires, ce qui correspond aux valeurs **nullptr** et **false**.

Fonctions kernel_setUserResult et getUserResult

Ces deux fonctions sont à ajoutées dans le fichier **xtr.cpp** :

```
void kernel_setUserResult (KERNEL_MODE_ const bool inUserResult) {  
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;  
    gRunningTaskControlBlockPtr->mUserResult = inUserResult ;  
}  
  
bool getUserResult (USER_MODE) {  
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;  
    return gRunningTaskControlBlockPtr->mUserResult ;  
}
```

Ajouter aussi la déclaration du prototype de ces fonctions dans **xtr.h** :

```
void kernel_setUserResult (KERNEL_MODE_ const bool inUserResult) ;  
  
bool getUserResult (USER_MODE) asm ("get.user.result") ;
```

L'annotation **asm** suggère que la seconde est appelée par des routines assembleur.

Fonction kernel_makeTaskReady

Ajouter l'argument **inUserResult** à la fonction **kernel_makeTaskReady** (fichier **xtr.cpp**) :

```
static void kernel_makeTaskReady (IRQ_MODE_
                                TaskControlBlock * inTaskPtr,
                                const bool inUserResult) {
    XTR_ASSERT_NON_NULL_POINTER (inTaskPtr) ;
    gReadyTaskList.enterTask (MODE_ inTaskPtr) ;
    inTaskPtr->mUserResult = inUserResult ;
}
```

Ensuite, pour tous les appels de cette fonction, ajouter comme nouvel argument la valeur **true**, sauf pour l'appel situé dans **irq_makeTasksReadyFromCurrentDate** où il faut ajouter **false**.

Fonction `irq_makeTasksReadyFromCurrentDate`

La fonction `irq_makeTasksReadyFromCurrentDate` (fichier `xtr.cpp`) est appelée à chaque interruption temps-réel ; si une tâche est débloquée, il faut la retirer de la liste des tâches bloquées d'un outil de synchronisation ; le champ `mBlockingList` du descripteur de tâche désigne cette liste, ou vaut `nullptr` si la tâche n'est pas bloquée sur un outil de synchronisation :

```
static void irq_makeTasksReadyFromCurrentDate (IRQ_MODE_
                                              const uint32_t inCurrentDate) {
    TaskList::Iterator iterator (MODE_ gDeadlineWaitingTaskList) ;
    TaskControlBlock * taskPtr ;
    while ((taskPtr = iterator.nextTask (MODE))) {
        if (inCurrentDate >= taskPtr->mDeadline) {
            //--- Remove task from blocking list
            if (nullptr != taskPtr->mBlockingList) {
                taskPtr->mBlockingList->removeTask (MODE_ taskPtr) ;
                taskPtr->mBlockingList = nullptr ;
            }
            //--- Remove task from deadline list
            gDeadlineWaitingTaskList.removeTask (MODE_ taskPtr) ;
            //--- Make task ready
            kernel_makeTaskReady (MODE_ taskPtr, false) ;
        }
    }
}
```

Fonction `irq_makeTaskReadyFromList`

La fonction **`kernel_blockRunningTaskInList`** (fichier **`xtr.cpp`**) rend prête la tâche bloquée la plus prioritaire dans la liste passée en argument. Il faut maintenant retirer la tâche de la liste des tâches bloquées sur échéance (la fonction **`removeTask`** n'a pas d'effet si la tâche n'est pas bloquée sur échéance):

```
bool irq_makeTaskReadyFromList (IRQ_MODE_ TaskList & ioWaitingList) {
    TaskControlBlock * taskPtr = ioWaitingList.removeFirstTask (MODE) ;
    const bool found = taskPtr != nullptr ;
    if (found) {
        taskPtr->mBlockingList = nullptr ;
        //--- Remove from deadline list
        gDeadlineWaitingTaskList.removeTask (MODE_ taskPtr) ;
        //--- Make task ready
        kernel_makeTaskReady (MODE_ taskPtr, true) ;
    }
    return found ;
}
```

Fonction `kernel_blockRunningTaskInListAndDeadline`

C'est une nouvelle fonction à ajouter (fichier **xtr.cpp**) : elle effectue le blocage de la tâche appelante sur un outil de synchronisation et une échéance :

```
void kernel_blockRunningTaskInListAndDeadline (KERNEL_MODE_  
                                              TaskList & ioWaitingList,  
                                              const uint32_t inDeadline) {  
    XTR_ASSERT_NON_NULL_POINTER (gRunningTaskControlBlockPtr) ;  
    //--- Insert in task list  
    ioWaitingList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;  
    gRunningTaskControlBlockPtr->mBlockingList = & ioWaitingList ;  
    //--- Insert in deadline list  
    gRunningTaskControlBlockPtr->mDeadline = inDeadline ;  
    gDeadlineWaitingTaskList.enterTask (MODE_ gRunningTaskControlBlockPtr) ;  
    //--- Block task  
    kernel_makeNoTaskRunning (MODE) ;  
}
```

Ajouter aussi la déclaration du prototype de cette fonction dans **xtr.h** :

```
void kernel_blockRunningTaskInListAndDeadline (KERNEL_MODE_  
                                              TaskList & ioWaitingList,  
                                              const uint32_t inDeadline) ;
```

Modifications à apporter aux outils de synchronisation

Le sémaphore de Dijkstra : déclaration de `P_until`

Pas de modification des primitives existantes.

On va ajouter la primitive **`P_until`**. D'abord, sa déclaration dans la classe **`Semaphore`** (fichier **`Semaphore.h`**) :

```
//$bool-service semaphore.P_until

public: bool P_until (USER_MODE_
                    const uint32_t inDeadline) asm ("semaphore.P_until") ;

public: void sys_P_until (KERNEL_MODE_
                        const uint32_t inDeadline) asm ("service.semaphore.P_until") ;
```

Deux points sont à souligner :

- une nouvelle annotation de service : `//$bool-service` ; celle-ci signifie que la primitive va renvoyer une valeur booléenne — **`P_until`** est déclarée renvoyant **`bool`** — qui est en fait la valeur du champ **`mUserResult`** ;
- la fonction d'implémentation du service **`sys_P_until`** ne renvoie pas de valeur booléenne, elle déclarée avec **`void`**.

Le sémaphore de Dijkstra : implémentation de sys_P_until

Ajouter l'implémentation de la primitive **sys_P_until** dans le fichier **Semaphore.cpp** :

```
void Semaphore::sys_P_until (KERNEL_MODE_ const uint32_t inDeadline) {
    const bool userResult = mValue > 0 ;
    kernel_setUserResult (MODE_ userResult) ; // SOULD BE CALLED BEFORE TASK BLOCKING
    if (userResult) {
        mValue -= 1 ;
    } else if (inDeadline > millis (MODE)) {
        kernel_blockRunningTaskInListAndDeadline (MODE_ mWaitingTaskList, inDeadline) ;
    }
}
```

Rappelons le fonctionnement de la primitive **P_until** décrit au début de ce document :

- si, au moment de l'appel le sémaphore est strictement positif, le sémaphore est décrémenté, et la primitive retourne immédiatement avec la valeur **true** ; **[kernel_setUserResult est appelée avec la valeur true]**
- si, au moment de l'appel le sémaphore est nul, et l'échéance est atteinte ou dépassée, et la primitive retourne immédiatement avec la valeur **false** ; **[kernel_setUserResult est appelée avec la valeur false]**
- sinon, la primitive est bloquante **[kernel_setUserResult est appelée avec une valeur sans importance]**, la tâche est en attente sur le sémaphore et l'échéance, jusqu'à ce que :
 - ▶ la tâche est débloquée suite à l'invocation de **V** par une autre tâche **[irq_makeTaskReadyFromList est appelé, ce qui provoque l'appel de kernel_setUserResult avec la valeur true]** ; la valeur **true** est retournée ;
 - ▶ la tâche est débloquée parce que l'échéance est atteinte **[irq_makeTasksReadyFromCurrentDate est appelé, ce qui provoque l'appel de kernel_setUserResult avec la valeur false]** ; la valeur **false** est retournée.

Le sémaphore de Dijkstra : appels svc engendrés

Après compilation, on examine le fichier engendré **zSOURCES/interrupt-handlers.s** (les numéros attribués aux appels **svc** peuvent être différents) :

```
semaphore.P:  
    .fnstart  
    svc #7  
    bx  lr  
  
.....  
  
semaphore.P_until:  
    .fnstart  
    svc #8  
    b    get.user.result
```

Pour la primitive **P**, l'annotation `//$service` engendre un appel qui se termine par un retour (instruction **bx lr**).

Pour la primitive **P_until**, l'annotation `//$bool-service` engendre un appel qui se termine par un branchement à la fonction **get.user.result**.

Travail à faire

Imaginer un programme qui met en évidence le fonctionnement de la primitive **P_until**.

Pour ceux qui sont en avance : implémenter les primitives d'attente temporisées sur les autres outils de synchronisation présentés à l'étape précédente.