Bâtiment EARHART
ZAC Grenoble Air Parc
38590 St Etienne de St Geoirs - France
www.pragmatec.net

Pragmatec

Produits et services dédiés aux systèmes embarqués temps-réel

PICos18

Noyau temps réel pour PIC18

Interface de programmation
API du noyau v2.xx

30 April 2006 – Rev 1.02

TO ANY PICos18 USER

Distribution:

PICos18 is free software; you can redistribute it and/or modify it under the terms of the GNU General License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

PICos18 is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with PICOS18; see the file COPYING.txt. If not, write to the Free Software Foundation, 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.

PICos18 est un produit de la société PRAGMATEC S.A.R.L. distribué gratuitement sous licence GPL. Celle-ci garantie la libre circulation des sources de PICos18.

PICos18 est un noyau temps réel basé sur la norme automobile OSEK/VDX™ et destiné aux microcontrôleurs PIC18 de la société Microchip Technology Inc.

Ce tutorial est la propriété de PRAGMATEC S.A.R.L. Il est destiné à la compréhension et la prise en main du logiciel PICos18. Pour des raisons pratiques et techniques, il a été réalisé sous MPLAB® et compilé avec le compilateur C18 pour cible PIC18F452.

TABLE DES MATIERES

1. PREAMBULE 6

UN NOYAU MULTI-TACHES TEMPS REEL 6
PICOS18 : UN NOYAU MULTI-TACHES TEMPS REEL POUR PIC18 7
LA NORME OSEK/VDX™ 7
LES CARACTERISTIQUES DE PICOS18 8
LA LICENCE GPL 9
LA CHAINE DE COMPILATION MICROCHIP 10
LA SOCIETE PRAGMATEC 11

2. API DE PICOS18 : PRESENTATION 12

GESTION DES TACHES 12
GESTION DES EVENEMENTS 12
GESTION DES ALARMES 12
CONTROLE DU NOYAU 12
GESTION DES INTERRUPTIONS 13
GESTION DES RESSOURCES 13
HOOK ROUTINES 13

3. GESTION DES TACHES 14

DECLARETASK 15
ACTIVATETASK 16
TERMINATETASK 17
CHAINTASK 18
SCHEDULE 19
GETTASKID 20
GETTASKSTATE 21

4. GESTION DES ÉVÉNEMENTS 22

DECLAREEVENT 23
SETEVENT 24
CLEAREVENT 25
GETEVENT 26
WAITEVENT 27

5. GESTION DES ALARMES 28

DECLAREALARM 29
GETALARMBASE 30
GETALARM 31
SETRELALARM 32
SETABSALARM 33
CANCELALARM 34

6. CONTROLE DU NOYAU 35

GETACTIVEAPPLICATIONMODE 36
START OS 37
SHUTDOWNOS 38

7. GESTION DES INTERRUPTIONS 39

RESUMEALLINTERRUPTS 40
SUSPENDALLINTERRUPTS 41
ENABLEALLINTERRUPTS 42
DISABLEALLINTERRUPTS 43
RESUMEOSINTERRUPTS 44
SUSPENDOSINTERRUPTS 45

8. GESTION DES RESSOURCES 46

LE PRINCIPE DU « CEILING PROTOCOL » EST SIMPLE : 46
DECLARERESOURCE 47
GETRESOURCE 48
RELEASERESOURCE 49

9. HOOK ROUTINES 50

ERRORHOOK 51
PRETASKHOOK 52
POSTTASKHOOK 53
STARTUPHOOK 54
SHUTDOWNHOOK 55

1. Préambule

Un noyau multi-tâches temps réel

On entend trop souvent parler de noyau temps réel dans le monde de l'embarqué sans trop savoir pour autant de quoi il en retourne. En fait pour comprendre ce qu'apporte un noyau multi-tâches temps-réel, il faut définir 3 aspects :

Un noyau...

C'est en fait un ensemble de fonctionnalités, regroupées sous le terme de **SERVICES** pour la plupart, qui forment le noyau. Dans le cas de Linux, par exemple, le noyau est constitué de la gestion des différents programmes, de l'accès au hardware, de la gestion des systèmes de fichiers... Le shell, quant à lui, est un programme, et n'est donc pas inclu dans le noyau. La fonction malloc, qui alloue dynamiquement de la mémoire à un programme, fait appel à un service du noyau, car c'est bel et bien le noyau qui est responsable de la gestion des ressources.
Une des fonctionnalié les plus connue des noyaux (sans être pour autant un service) est le **SCHEDULER**, responsable de la cohabitation des différents programmes pendant leur exécution.

... multi-tâches ...

C'est donc le noyau qui contrôle les ressources et permet leur utilisation de façon sûre et efficace au travers de **SERVICES**. Puisque le noyau garantit la stabilité du système, plusieurs programmes indépendants peuvent se tenir prêts à fonctionner, au bon vouloir du noyau. Cette **MULTI-PROGRAMMATION** permet aux développeurs de créer des programmes sans se soucier de savoir s'il existe d'autres programmes dans le système. Chacun a l'impression que le système lui est dédié.
Si le noyau le permet, il est même possible de faire fonctionner ces programmes en parallèle tout en donnant l'impression à chacun d'être seul à fonctionner (au prix d'une perte de vitesse bien entendu).
Lorsque le noyau joue parfaitement son rôle de chef d'orchestre (le fonctionnement en parallèle des tâches incombent au noyau seul) on dit que le noyau est **MULTI-TACHES PREEMPTIF**. Si le noyau n'est pas capable de faire une telle chose, alors c'est aux tâches de "rendre la main" au noyau de temps en temps. On dit que le noyau est **MULTI-TACHES COOPERATIF**.

PICos18 est un noyau temps réel préemptif.

... temps réel

Le noyau multi-tâche peut simplement gérer le parallélisme en découpant le temps en parts égales pour chaque tâche en mémoire. Le problème, c'est que les tâches en fonctionnement ont rarement les mêmes besoins, et certaines, rarement actives, requièrent toute la puissance du système lorsqu'elles se réveillent.
Les tâches ont donc des **PRIORITES** différentes, et doivent pouvoir répondre à un événement dans un temps le plus court possible. Plutôt que d'assurer un temps de réactivité quasi-nul (ce qui est impossible), le noyau se doit de garantir un **TEMPS DE LATENCE** constant : c'est le **DETERMINISME**.

Le temps de latence de PICos18 est de 50 us.

PICos18 : un noyau multi-tâches temps réel pour PIC18

Avant l'arrivée des PIC18, il n'était pas possible de développer un tel noyau sur les PICmicro. En effet la caractéristique première d'un noyau multi-tâches est de faire cohabiter des tâches ensemble, ce qui implique de contrôler la pile des appels de fonctions [*cf datasheet du PIC18F452 DS39564A, page 37, chapitre 4.2 « Return Address Stack »*]. Imaginez ce qui se passerait si toutes les tâches en fonctionnement partageaient la même pile : le noyau interromprait le fonctionnement d'une tâche pour activer une autre tâche, et à la prochaine instruction RETURN rencontrée, le retour se ferait dans la première tâche, au niveau du dernier appel de fonction !

Les PIC18 permettent la manipulation de la pile des appels de fonctions (instructions PUSH et POP ajoutée au jeu d'instructions, et déplacement du pointeur de pile), si bien qu'il est désormais possible de mettre la pile dans un état correct avant l'activation de la prochaine tâche.

Il restait alors à définir la liste des services du noyau à développer et sa gestion interne des tâches et des ressources. Plutôt que de partir dans une solution propriétaire, la société Pragmatec, à l'origine de PICos18, a choisi de se baser sur une norme : la norme OSEK/VDX™.

La norme OSEK/VDXÔ

La norme retenue est la norme OSEK-VDX™ (www.osek-vdx.org).
OSEK-VDX™ est un vaste projet de l'industrie automobile, commun aux plus grands constructeurs et équipementiers allemands et français. L'objectif de ce projet est de définir un standard pour le contrôle et la supervision des architectures distribuées embarquées dans les véhicules. En effet, les véhicules actuels peuvent embarquer jusqu'à 30 calculateurs (contrôle moteur, habitacle, blocs de portière, ABS, ESP…) qui communiquent entre eux, par liaisons filaires ou multiplexées (bus CAN, VAN, LIN, MOST…).

Normaliser les relations entre ces calculateurs, c'est :

- homogénéiser les spécifications des développements, les développements eux-mêmes, les validations systèmes ;
- avoir un langage commun entre constructeurs, équipementiers et fournisseurs d'outils de développement ;
- fournir une base commune pour tous pour le développement, les validations et intégrations.

Le terme OSEK signifie « Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug » (Systèmes ouverts et les interfaces correspondantes pour l'électronique automobile »). Le terme VDX signifie « Vehicle Distributed eXecutive ».

La fonctionnalité de l'OS OSEK a été combiné avec VDX.

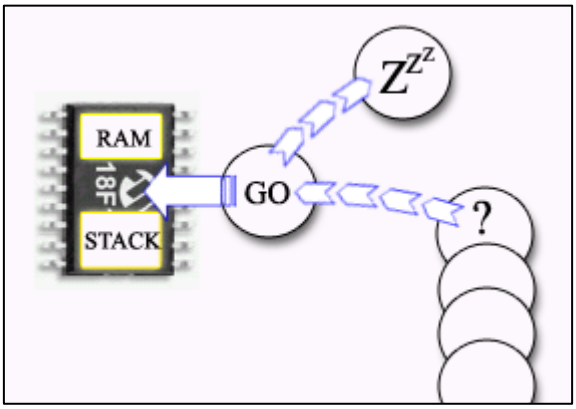
Aujourd'hui utilisée dans l'automobile et la robotique, cette norme définie le rôle du noyau autour de 3 axes : Operating System (OS), Communication (COM) et Network Management (NM). Pour l'instant seule la partie OS a été implémentée dans PICos18.

La norme OSEK-VDX™ est une norme parfaitement adaptée aux PIC18. Une application sous

PICos18 se compose d'un ensemble de tâches symbolisées par des cercles sur le schéma ci-dessous.

Le principe de fonctionnement veut qu'une seule tâche à la fois peut avoir accès au PIC18 (plus exactement au processeur, à la mémoire RAM et à la pile matérielle).

Afin de déterminer quelle tâche mérite d'être exécutée à un instant T, le noyau PICos18 examine l'ensemble des tâches de l'application et choisi la tâche prête la plus prioritaire.



Lors de son exécution celle-ci peut se mettre en attente d'un événement et ainsi s'endormir, laissant la place à une autre tâche moins prioritaire. Lorsque l'événement attendue sera détecté par le noyau, ce dernier s'empressera de réveiller la tâche précédemment endormie.

Les différents états d'une tâche de PICos18 peuvent être : **READY**, **SUSPENDED**, **WAITING** et **RUNNING**.

Les caractéristiques de PICos18

Le noyau PICos18 possède les caractéristiques suivantes :

- ✓ Le **cœur du noyau** (Init + Scheduler + Task Manager) qui a la responsabilité de gérer les tâches de l'application et donc de déterminer la prochaine tâche active en fonction de l'état et la priorité de chaque tâche.
- ✓ Le **gestionnaire d'alarmes et de compteurs** (Alarm Manager). Proche du cœur du noyau, il répond à l'interruption du TIMER0 afin de mettre à jour périodiquement les alarmes et compteurs associées aux tâches.
- ✓ Les **Hook routines** sont proches du coeur du noyau et permettent à l'utilisateur de dérouter le déroulement normal du noyau de façon à prendre temporairement le contrôle du système.
- ✓ Le **gestionnaire de tâches** (Process Manager) est un service du noyau, dont le rôle est d'offrir à l'application les fonctions nécessaires à la gestion des états (changer l'état d'une tâche, chaîner des tâches, activer une tâche...).
- ✓ Le **gestionnaire d'évènement** (Event Manager) est un service du noyau dont le rôle est d'offrir à l'application les fonctions nécessaires à la gestion des évènements d'une tâche (mise en attente sur un évènement, effacer un évènement...).
- ✓ Le **gestionnaire d'interruption** (INT Manager) offre à l'application les fonctions nécessaires à l'activation et la désactivation des interruptions du système.

PICos18 est un noyau modulaire dans le sens où les accès spécifiques aux ressources (drivers, file system manager, etc.) peuvent être réalisés par des tâches dissociées du noyau.

De cette façon PICos18 offre la possibilité d'intégrer des modules sous forme de **briques logicielles** afin de constituer un projet, telles que les extensions proposées par la société PRAGMATEC.

La licence GPL

Le noyau PICos18 est en *open-source* et est distribué sous licence GPL *(General Public Licence)*. Cela signifie que toutes les sources du noyau en C et en assembleur sont disponibles. Cela signifie également qu'il est totalement gratuit et n'implique le paiement d'aucune royaltie à ses auteurs.

En en-tête de chaque fichier source de PICos18, vous retrouverez donc le même texte, à conserver dans les fichiers quelque soit votre utilisation ou votre projet avec PICos18 :

```
/************************************************************************/
/*                                                                      */
/* File name: le nom du fichier source                                  */
/*                                                                      */
/* Since:     date de création                                          */
/*                                                                      */
/* Version:   2.xx  (version courante du noyau PICos18)                 */
/*                                                                      */
/* Author:    Designed by Pragmatec S.A.R.L.        www.pragmatec.net */
/*            MONTAGNE Xavier [XM]      xavier.montagne@pragmatec.net */
/*            FAMILLE Prenom  [xx]                                      */
/*                                                                      */
/* Purpose:   une explication sur le rôle du fichier                    */
/*                                                                      */
/*  Et enfin, l'enregistrement de PICos18 auprès de la fondation        */
/*  du logiciel libre à Boston aux Etats Unis, la « Free Software       */
/*  Foundation »                                                        */
/* Distribution: This file is part of PICos18.                          */
/*            PICos18 is free software; you can redistribute it         */
/*            and/or modify it under the terms of the GNU General       */
/*            Public License as published by the Free Software          */
/*            Foundation; either version 2, or (at your option)         */
/*            any later version.                                        */
/*                                                                      */
/*            PICos18 is distributed in the hope that it will be        */
/*            useful, but WITHOUT ANY WARRANTY; without even the        */
/*            implied warranty of MERCHANTABILITY or FITNESS FOR A      */
/*            PARTICULAR PURPOSE.  See the GNU General Public           */
/*            License for more details.                                 */
/*                                                                      */
/*            You should have received a copy of the GNU General        */
/*            Public License along with gpsim; see the file             */
/*            COPYING.txt. If not, write to the Free Software           */
/*            Foundation, 59 Temple Place - Suite 330,                  */
/*            Boston, MA 02111-1307, USA.                               */
/*                                                                      */
/*          > A special exception to the GPL can be applied should      */
/*            you wish to distribute a combined work that includes      */
/*            PICos18, without being obliged to provide the source      */
/*            code for any proprietary components.                      */
/*                                                                      */
/* History:   Les modifications successives apportées à ce fichier      */
/* 2004/09/20 [XM]  Create this file.                                   */
/*                                                                      */
/************************************************************************/
```

La licence GPL garantie la libre circulation des sources de PICos18, sans restrictions ou protections imposées par une quelconque société ou organisation En revanche, la société PRAGMATEC, à

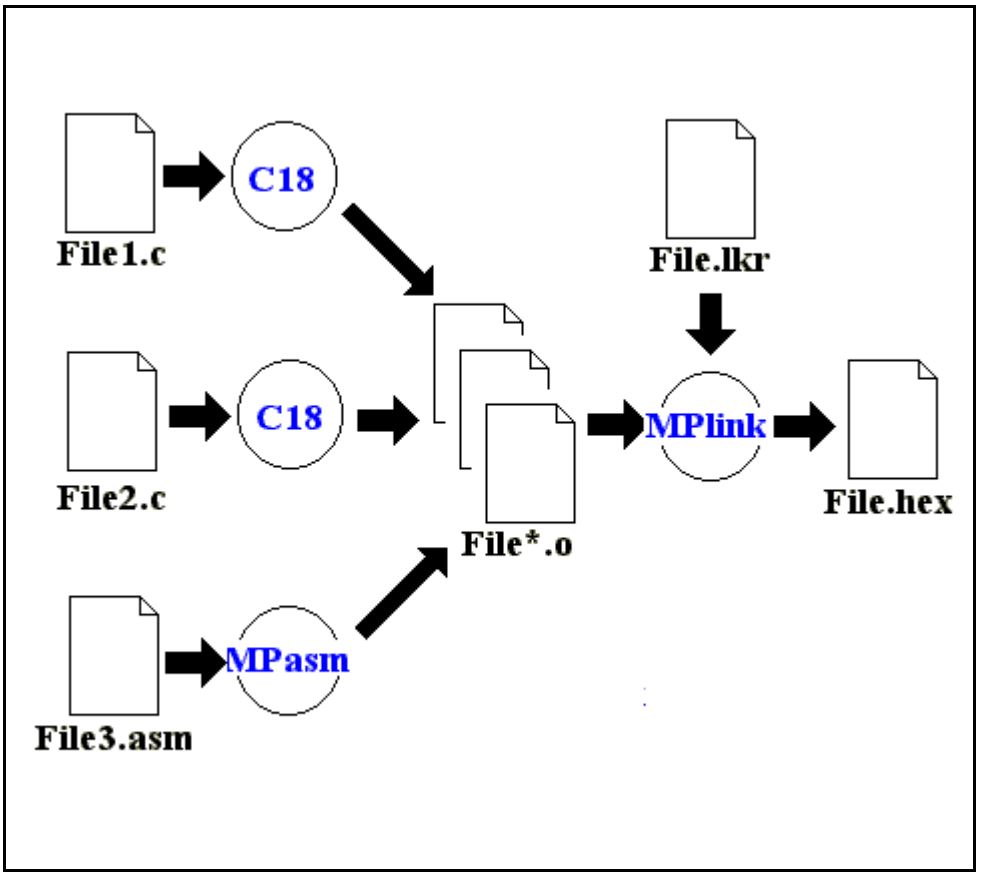
l'origine de PICos18, s'engage à maintenir le noyau et à le faire évoluer, tout en respectant les règles de la licence GPL. Bien entendu, quiconque souhaite apporter sa contribution à l'édifice peut le faire. Son nom apparaîtra alors au côté de celui des auteurs dans l'en-tête des fichiers.

Une mention spéciale a été rajouté à la licence GPL et ceci dans chaque entête de fichier. Cette mention écrite en anglais précise qu'il est possible d'associer à PICos18 une application sans pour autant être tenu de respecter la licence GPL sur cette partie, c'est-à-dire sans devoir fournir le code source de l'application. Par contre vous devrez être en mesure de fournir à quiconque le code source du noyau PICos18.

La chaîne de compilation Microchip

Vous allez programmer votre application en C sous PICos18. Toutefois le noyau lui-même est composé de fichiers en C pour les services par exemple et de fichiers écrits en assembleur comme le fichier kernel.asm.

Ces différents fichiers devront être compilés ou assemblés puis liés afin d'obtenir un seul fichier au final : le fichier HEX qui pourra être chargé dans le PIC18.



La chaîne de compilation Microchip se décompose en 3 éléments :

- l'assembleur MPASM qui permet de transformer un fichier ASM en fichier O

- le compilateur MCC18 qui permet de transformer un fichier C en un fichier O
- le linkeur MPLINK qui permet de fusionner tous les fichiers O en un unique fichier HEX

Le fichier dit “script du linkeur” (File.lkr sur le schéma ci-dessus) est essentiel pour permettre de générer le fichier HEX. En effet le contenu des fichiers O ne permet pas encore de savoir où va être logé le code en mémoire. Par exemple la fonction main() a été traduite en un langage compréhensible par le PIC18 mais pas encore positionné à un endroit précis de la mémoire.

C’est le rôle du script du linkeur que de préciser les emplacements de chaque portion de code en ROM et chaque portion de variable en RAM. PICos18 est fourni avec des scripts de linker pré-établi pour les PIC les plus utilisés de la famille PICos18. Vous pouvez vous en inspirer pour réaliser votre propre script ou bien pour l’adapter à un nouveau PIC18.

Il existe d’autres types de fichiers générés pendant la compilation et l’assemblage des fichiers du projet (*.map, *.lst, *.cod). Référez vous aux documents Microchip pour de plus amples informations.

Ce tutorial est destiné à la compréhension et la prise en main du logiciel PICos18. Pour des raisons techniques, il a été réalisé sous MPLAB® et compilé avec le compilateur C18 pour cible PIC18F452.

Nous tenons à préciser que ce tutorial a été entièrement réalisé et testé sous le système d’exploitation Microsoft Windows™ 98/NT/2K/XP, ainsi qu’avec l’environnement de développement MPLAB® v7.00 et le compilateur C18 v2.40 de Microchip (version gratuite).

Ces différents logiciels peuvent être téléchargés depuis le site web de Microchip Technology Inc. : www.microchip.com.

La société Pragmatec

PICos18 est un produit de la société PRAGMATEC distribué gratuitement sous licence GPL.

La société PRAGMATEC développe et distribue des extensions de PICos18 afin de permettre aux développeurs de réaliser leurs applications à l’aide briques logicielles (driver RS232, driver bus CAN pour utiliser le contrôleur de protocole CAN du PIC18F458, USB, I2C, etc…).

PRAGMATEC propose aussi un ensemble de logiciels de plus haut niveau pour la mise en œuvre de PICos18, permettant par exemple le contrôle et le debug des tâches à distance, via RS232 ou CAN…

2. API de PICOS18 : présentation

L'objectif de ce document est de présenter l'interface de programmation de PICos18. La norme OSEK-VDX™ définit non seulement les fonctionnalités du noyau, mais aussi la façon dont il doit être interfacé avec les programmes utilisateurs. PICos18 ne respecte pas encore toutes ces spécifications, et nous présentons ici l'ensemble des interfaces du noyau ainsi que des exemples se rapportant à ces interfaces :

Gestion des tâches

Ce chapitre comprend toutes les fonctions se rapportant à la manipulation des tâches comme l'activation d'une tâche ou bien la lecture de son état ou de son ID. Avec PICos18 les tâches sont déclarées de façon statiques, c'est-à-dire qu'elles sont déclarées au noyau lors à la compilation, et ceci une fois pour toute. Les fonctions se rapportant à la « gestion des tâches » font donc référence à la manipulation des tâches en cours de fonctionnement.

Gestion des événements

Les événements OSEK/VDX™ sont en quelque sorte les sémaphores de la norme POSIX : un moyen de communication inter-process. Les événements ont un rôle essentiel car ils vont permettre à une tâche de se mettre en attente sur un événement. Il est aussi possible pour une tâche ou une interruption de poster des événements afin de réveiller des tâches ou se synchroniser des actions entre plusieurs tâches.

Gestion des alarmes

La synchronisation des tâches entre elles est un élément essentiel des systèmes multi-tâches, mais ce n'est pas le seul. En effet certaines tâches nécessitent de se synchroniser sur une base de temps telles que les alarmes logicielles. En utilisant une alarme une tâche va pouvoir être réveillée à des intervalles fixes et pour un nombre de cycles fixes ou infinis. On pourra ainsi facilement créer une tâche qui fera clignoter une LED toutes les 300 ms.

Contrôle du noyau

Le noyau ne démarre pas automatiquement au démarrage selon la norme OSEK/VDX™. En effet c'est tout d'abord votre application qui prend la main durant la phase de boot en exécutant la fonction main(). Par la suite il vous incombe d'appeler le noyau pour permettre la gestion de votre application multi-tâches.

Gestion des interruptions

Pourquoi utiliser un microcontrôleur si l'on ne peut utiliser ses périphériques ? PICos18 est conçu pour vous permettre de bénéficier des interruptions, qui peuvent même s'interfacer avec vos tâches en leur postant des évènements. Les routines d'interruptions, appelées ISR, peuvent elles aussi appeler les services du noyau et ainsi faire office de pré-process pour les tâches associées.

Gestion des ressources

PICos18 autorise le libre accès à toutes les ressources du PIC18 à toutes les tâches de vos applications. Ceci permet de vous simplifier l'accès aux périphériques mais peut engendrer quelques soucis de droit daccès lorsque ce même périphérique est utilisé par plusieurs tâches en même temps. Ceci est résolu par la gestion des ressources en mode «ceiling protocol » de la norme OSEK/VDX™, désormais supporté par la version v2.xx de PICos18.

Hook routines

Durant l'exécution de votre application il est toujours possible de contrôler son bon déroulement grâce à des « sondes logicielles », les Hook routines, véritables capteurs liés à vos tâches.

3. Gestion des tâches

Ce chapitre comprend toutes les fonctions se rapportant à la manipulation des tâches comme l'activation d'une tâche ou bien la lecture de son état ou de son ID. Avec PICos18 les tâches sont déclarées de façon statiques, c'est-à-dire qu'elles sont déclarées au noyau lors à la compilation, et ceci une fois pour toute. Les fonctions se rapportant à la « gestion des tâches » font donc référence à la manipulation des tâches en cours de fonctionnement.

Types utilisés pour la gestion des tâches :

TaskType	Permet de renseigner l'ID d'une tâche. Valeur 8 bits signée (char) comprise entre 0 et 7 inclus.
TaskRefType	Permet d'obtenir d'ID d'une tâche. Référence sur une valeur 8 bits signée (char*) comprise entre 0 et 7 inclus.
TaskStateType	Contient la valeur de l'état d'une tâche : ✓ SUSPENDED 0x00 ✓ READY 0x20 ✓ RUNNING 0x40 ✓ WAITING 0x80 Ce type n'est pas utilisé par l'API OSEK-VDX™.
TaskStateRefType	Permet d'obtenir l'état d'une tâche. Référence sur une valeur 8 bits signées (char*) qui vaut : ✓ SUSPENDED 0x00 ✓ READY 0x20 ✓ RUNNING 0x40 ✓ WAITING 0x80

DeclareTask

Prototype	**`DeclareTask(TaskIdentifier)`**
Description	Permet de déclarer le nom de la tâche à la chaîne de compilation. Est utilisé par le linker au moment de l'édition de lien.
Paramètres [in]	*TaskIdentifier* : nom de la tâche.
Valeur de retour	Aucune
Remarques	Cette fonction de l'API OSEK-VDX™ n'est pas utilisée avec PICos18. Le nom de la tâche n'est pas traitée par le linker MPLINK de Microchip. Pour des soucis de compatibilités, la fonction est tout de même implémentée sous forme de macro : *#define DeclareTask(TaskIdentifier) extern TASK(TaskIdentifier)*
Fichier	pro_man.h
Exemple *Le fichier tscdesc.c a pour fonction de décrire l'application PICos18. Afin que le noyau puisse connaître l'amplacement de vos tâches vous devez toutes les déclarer dans le fichier tascdesc.c comme indiqué dans l'exemple.*	``` ... #define DEFAULT_STACK_SIZE 128 DeclareTask(TASK0); DeclareTask(HD4478_DRV); volatile unsigned char stack0[DEFAULT_STACK_SIZE]; volatile unsigned char stack_hd4478[DEFAULT_STACK_SIZE]; ... ```

ActivateTask

Prototype	`StatusType ActivateTask (TaskType TaskID)`
Description	Permet de faire passer l'état d'une tâche de SUSPENDED à READY. Si la tâche ainsi activée est la tâche prête la plus prioritaire elle prend immédiatement la main sur la tâche en cours.
Paramètres [in]	*TaskID* : ID de la tâche à activer.
Valeur de retour	Aucune si l'ID correspond à une tâche de l'application. `E_OS_ID` dans le cas contraire.
Remarques	Si l'ID passé en paramètre correspond à une tâche de l'application, la fonction ActivateTask appelle le scheduler afin de déterminer la prochaine tâche active. Les événements précédemment postés pour la tâche activées ne sont pas remis à zéro. Le noyau garantit l'exécution de la tâche activée depuis la première ligne de code de la tâche. Vous pouvez activer plusieurs fois une tâche avant qu'elle ne se termine. On dit qu'il y a activation multiple de la tâche. Lorsqu'une tâche se termine, elle est en mesure de s'activer à nouveau tant que son compteur d'activation n'est pas nulle (valeur maximale de 7).
Fichier	pro_man.c
Exemple *La tâche MY_TASK attend ici l'événement KEY_PRESSED et active une tâche de gestion de LCD dès la détection de la touche appuyée.* *ActivateTask ne change pas l'état de la tâche appelante. Le scheduler est automatiquement appelé afin de déterminer si la priorité de la nouvelle tâche est suffisante pour permettre son activation immédiate.*	

```
TASK(MY_TASK)
{
  unsigned char Key;

  while (1)
  {
    WaitEvent(KEY_PRESSED);
    ClearEvent(KEY_PRESSED);

    KeyDetection(&Key);
    ActivateTask(LCD_TASK_ID);

    /* ... */
  }
}
```

TerminateTask

Prototype	`StatusType TerminateTask (void)`
Description	Permet de faire passer l'état d'une tâche de RUNNING à SYSPENDED. La tâche n'est plus alors considérée par le noyau. Pour l'exécuter de nouveau il est nécessaire de l'activer à l'aide de la fonction *`ActivateTask.`*
Paramètres [in]	Aucun
Valeur de retour	Aucune
Remarques	La fonction TerminateTask appelle systématiquement le scheduler afin de déterminer la prochaine tâche active. Une tâche peut choisir de se terminer elle-même mais ne peut pas choisir de terminer une autre tâche. La fonction TerminateTask ne retourne pas de valeur. Ne jamais l'appeler depuis une fonction. La tâche se terminant ne doit en aucun cas occuper des ressources tels qu'un périphérique ou une alarme. Il n'est donc pas possible de créer des alarmes au sein d'une tâche qui appellerait TerminateTask. Dans ce cas précis il est nécessaire de déclarer et de paramétrer l'alarme à l'extérieur de la tâche (dans la fonction main par exemple). Si le compteur d'activation n'est pas nulle, la fonction TerminateTask redémarre immédiatement la tâche concernée (voir ActivateTask).
Fichier	pro_man.c
Exemple *TerminateTask permet de placer une tâche à l'état SUSPENDED de façon à la désactiver.*	`TASK(MY_TASK)` `{` `  My_Task_Init();` `  while (1)` `  {` `  WaitEvent(RS232_EVENT | TIMEOUT);` `  GetEvent(my_task_ID, &my_task_event);` `  if (my_task_event & TIMEOUT)` `  {` `  ClearEvent(TIMEOUT);` `  **TerminateTask();**` `  }` `  /* ... */` `  }` `}`

ChainTask

Prototype	**`StatusType ChainTask (TaskType TaskID)`**
Description	Permet de faire passer l'état de tâche courante de RUNNING à SUSPENDED. La tâche dont l'ID est TaskID passe à l 'état READY quelque soit son état actuel. Si la tâche ainsi activée est la tâche prête la plus prioritaire elle prend immédiatement la main sur la tâche en cours.
Paramètres [in]	*TaskID* : ID de la tâche chaînée.
Valeur de retour	Aucune
Remarques	La fonction ChainTask appelle systématiquement le scheduler afin de déterminer la prochaine tâche active. Une tâche peut choisir de se chaîner avec elle-même ce qui a pour effet de la redémarrer. La fonction ChainTask ne retourne pas de valeur. Ne jamais l'appeler depuis une fonction. La tâche se terminant ne doit en aucun cas occuper des ressources tels qu'un périphérique ou une alarme. Il n'est donc pas possible de créer des alarmes au sein d'une tâche qui appellerait ChainTask. Dans ce cas précis il est nécessaire de déclarer et paramétrer l'alarme à l'extérieur de la tâche (dans la fonction main par exemple). La fonction ChainTask termine effectivement la tâche en cours, quelque soit sa valeur de compteur d'activation, et ceci afin de garantir le chaînage des 2 tâches. Pour conserver la notion d'activation multiple, préféré l'emploi de la fonction ActivateTask(TASK_ID) suivi de TerminateTask() (voir la fonction ActivateTask).
Fichier	pro_man.c
Exemple *ChainTaks permet non seulement d'activer une prochaine tâche mais aussi de terminer la tâche courante. Attention tout code suivant l'appel à ChainTask ne sera pas exécuté !*	`TASK(MY_TASK)` `{` `  TaskStateType  the_task_state;` `  My_Task_Init();` `  While (1)` `  {` `  WaitEvent(ALARM_EVENT);` `  ClearEvent(ALARM_EVENT);` `  GetTaskState(TARGET_TASK_ID, &the_task_state);` `  If (the_task_state == SUSPENDED)` `  {` `  **ChainTask(TARGET_TASK_ID);**` `  /* The current task is suspended now ...*/` `  }` `  }` `}`

Schedule

Prototype	`StatusType Schedule(void)`
Description	Appel le scheduler du noyau afin de déterminer la prochaine tâche active. Si la tâche appelante est la tâche à l'état READY la plus prioritaire, l'exécution se poursuit immédiatement après l'appel à la fonction Schedule.
Paramètres [in]	/
Valeur de retour	Aucune.
Remarques	Le noyau PICos18 est un noyau préemptif et non pas coopératif, il n'est donc pas nécessaire d'appeler Schedule pour forcer le ré-ordonnancement des tâches, celui-ci étant à la charge complète du noyau. L'appel à la fonction Schedule peut être utilisé pour sortir d'une région critique, c'est-à-dire d'une portion de code durant laquelle toutes les interruptions sont dévalidées. Ce cas précis est le seul où le noyau ne pourra pas reprendre la main si nécessaire, il est donc de la responsabilité du développeur de faire le ré-ordonnancement des tâches.
Fichier	pro_man.c
Exemple *Certains accès à des périphériques internes ou externes nécessitent de rentrer en région critique, c'est-à-dire de dévalider toutes les interruptions. L'appel à Schedule permet de forcer le ré-ordonnancement des autres tâches si besoin. Attention : Schedule repositionne les interruptions donc il est nécessaire de les dévalider pour rester en région critique (d'où le second « GIE=0 » après l'appel à Schedule).*	``` TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ /* Enter critical region */ INTCONbit.GIEL = 0; For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); Schedule(); /* GIEL = 1 when coming back from Schedule */ INTCONbit.GIEL = 0; } INTCONbit.GIEL = 1; /* Leave critical region */ /* ... */ } } ```

GetTaskID

Prototype	**`StatusType GetTaskID (TaskRefType TaskID)`**
Description	Retourne l'ID de la tâche en cours.
Paramètres [out]	*TaskID* : ID de la tâche en cours.
Valeur de retour	*E_OK* systématiquement.
Remarques	Cette fonction est destinée à être utilisée par les Hook Routines et les ISR. Appelée depuis la fonction main, elle ne peut apporter aucune information utile et retourne la valeur INVALID_TASK (ID hors limites). Appelée depuis une tâche, cette fonction renvoie l'ID de la tâche. Depuis une ISR elle renvoie l'ID de la tâche interrompue. Depuis la fonction main, elle retourne INVALID_TASK.
Fichier	pro_man.c
Exemple *GetTaskID est utilisé ici pour éviter d'utiliser des constantes représentant l'ID de la tâche (id_tsk_run). Cet appel system est aussi fort utile au sein d'une ISR pour connaître l'ID de la tâche interrompue.*	`TASK(MY_TASK)` `{` `  TaskType my_task_ID;` `  while (1)` `  {` `  WaitEvent(ALARM_EVENT | AN_EVENT);` `  **GetTaskID(&my_task_ID);**` `  GetEvent(my_task_ID, &my_task_event);` `  if (my_task_event & AN_EVENT)` `  ClearEvent(ALARM_EVENT);` `  /* ... */` `  }` `}`

GetTaskState

Prototype	`StatusType GetTaskState (TaskType TaskID, TaskStateRefType State)`
Description	Retourne l'état de la tâche interrogée.
Paramètres [in] [out]	*TaskID* : ID de la tâche interrogée. *State* : état de la tâche interrogée.
Valeur de retour	*E_OK* si l'ID de la tâche interrogée existe. *E_OS_ID* dans le cas contraire.
Remarques	GetTaskState renvoie la valeur exacte de l'état de la tâche interrogée au moment de l'appel. Si l'appel se fait au sein d'une tâche, l'état retourné est RUNNING si la tâche cherche à connaître son propre état (seule la tâche en cours est dans l'état RUNNING). Cette fonction peut être appelée depuis une ISR ou une Hook Routines. Appelée depuis la fonction main elle renseigne l'état SUSPENDED (aucune tâche n'est encore en fonctionnement).
Fichier	pro_man.c
Exemple *Il est possible de connaître l'état courant de n'importe quelle tâche de l'application à l'aide de la fonction GetTaskState. Appliquée sur elle-même une tâche obtient 0x6x (READY + RUNNING) .* *SUSPENDED = 0x00* *READY = 0x20* *RUNNING = 0x40* *WAITING = 0x80*	`TASK(MY_TASK)` `{` `  TaskStateType  the_task_state;` `  while (1)` `  {` `  WaitEvent(ALARM_EVENT);` `  ClearEvent(ALARM_EVENT);` `  GetTaskState(TARGET_TASK_ID, &the_task_state);` `  if (the_task_state == SUSPENDED)` `  {` `  ActivateTask(TARGET_TASK_ID);` `  }` `  /* ... */` `  }` `}`

4. Gestion des événements

Les événements OSEK/VDX™ sont en quelque sorte les sémaphores de la norme POSIX : un moyen de communication inter-process. Les événements ont un rôle essentiel car ils vont permettre à une tâche de se mettre en attente sur un événement. Il est aussi possible pour une tâche ou une interruption de poster des événements afin de réveiller des tâches ou se synchroniser des actions entre plusieurs tâches.

Types utilisés pour la gestion des événements :

EventMaskType	Désigne un ensemble d'événements attendus ou postés d'une tâche. Valeur 8 bits signée (char) comprise entre 0 et 128 inclus. Cette valeur est un multiple de 2 (0, 1, 2, 4, 8, 16, 32, 64 et 128).
EventMaskRefType	Permet d'obtenir les événements postés à une tâche. Référence sur une valeur 8 bits signée (char*) comprise entre 0 et 128 inclus. Cette valeur est un multiple de 2 (0, 1, 2, 4, 8, 16, 32, 64 et 128).

DeclareEvent

Prototype	`DeclareEvent (EventIdentifier)`
Description	Permet de déclarer le nom de l'événement à la chaîne de compilation. Est utilisé par le linker au moment de l'édition de lien.
Paramètres [in]	*EventIdentifier* : nom de l'événement.
Valeur de retour	/
Remarques	Cette fonction de l'API OSEK-VDX™ n'est pas utilisée dans PICos18. Le nom de l'événement n'est pas traitée par le linker MPLINK de Microchip. Cette fonction n'est pas implémentée dans PICos18.
Fichier	/
Exemple	/

SetEvent

Prototype	`StatusType SetEvent (TaskType TaskID, EventMaskType Mask)`
Description	Permet de poster un événement à la tâche dont l'identificateur est ID et selon le masque d'événements Mask.
Paramètres [in] **[in]**	*TaskID* : ID de la tâche concernée. *Mask* : masque de l'événement posté.
Valeur de retour	*E_OS_STATE* si la tâche concernée est dans l'état SUSPENDED. *E_OK* si la tâche concernée n'attend pas l'événement posté. Aucune si la tâche concernée attend l'événement posté.
Remarques	Cette fonction sert à poster un événement à une tâche depuis une autre tâche ou depuis une ISR. Elle ne doit pas être utilisée depuis une Hook Routine. Si la tâche concernée est dans l'état WAITING sur l'événement posté, un ré-ordonnancement est forcé : si la tâche concernée est plus prioritaire elle prend la main sur la tâche en cours, cette dernière passant à l'état READY (cas d'une préemption sur événement). L'événement posté doit toujours être un multiple de 2. Il n'est donc possible de poster que 8 types d'événements différents à une tâche.
Fichier	even_man.c
Exemple *Lors de l'écriture d'un driver, la fonction ISR en charge de la détection du flag d'interruption poste un événement à la tâche « driver » à l'aide de la fonction SetEvent si le flag d'interruption est positionné*	`/* CAN driver ISR */` `Void CAN_INT(void)` `{` `  if (PIR3bits.RX0IF)` `  {` `  PIR3bits.RX0IF = 0;` `  SetEvent(CAN_DRV_ID, CAN_RX_EVENT);` `  };` `  /* ... */` `}`

ClearEvent

Prototype	`StatusType ClearEvent (EventMaskType Mask)`
Description	Permet de supprimer un événement reçu par la tâche en cours selon le masque d'événements Mask.
Paramètres [in]	*Mask* : masque de l'événement supprimé.
Valeur de retour	*E_OK* dans tous les cas.
Remarques	Cette fonction sert à supprimer un événement reçu par une tâche. La tâche en attente d'événements à en effet la responsabilité de supprimer ces événements une fois reçus. Si la tâche réveillée ne supprime pas l'événement reçu, elle risque de boucler à l'infini, réveillé par un événement toujours présent. Elle ne doit pas être utilisée depuis une Hook Routine, une ISR ou la fonction main. L'événement supprimé doit toujours être un multiple de 2. Il n'est donc possible de supprimer que 8 types d'événements différents.
Fichier	even_man.c
Exemple *Après avoir reçu un événement, la tâche doit l'effacer par l'appel à la fonction ClearEvent.*	`TASK(MY_TASK)` `{` `  My_Task_Init();` `  while (1)` `  {` `  WaitEvent(ALARM_EVENT | AN_EVENT);` `  GetEvent(MY_TASK_ID, &my_task_event);` `  if (my_task_event & ALARM_EVENT)` `  {` `  **ClearEvent(ALARM_EVENT);**` `  LATEbits.LATE2 = ~LATEbits.LATE2;` `  }` `  /* ... */` `  }` `}`

GetEvent

Prototype	**`StatusType GetEvent (TaskType TaskID, EventMaskRefType Event)`**
Description	Permet d'obtenir les événements reçus par une tâche dont l'identificateur est ID.
Paramètres [in] [out]	*TaskID* : ID de la tâche concernée. *Event* : masque des événements reçus.
Valeur de retour	*`E_OS_STATE`* si la tâche concernée est dans l'état SUSPENDED. *`E_OS_ID`* si l'ID ne correspond pas à une tâche de l'application. *`E_OK`* dans le cas contraire.
Remarques	Cette fonction permet de connaître les événements reçus par une tâche, et non pas les événements en attente. Elle peut être utilisée depuis une Hook Routine, une ISR ou la fonction main. La valeur Event est composé de plusieurs événements, chacun étant un multiple de 2. Lorsque la tâche réveillée attend plusieurs événements, il est nécessaire de filtrer la valeur Event afin de déterminer l'événement qui a réveillé la tâche.
Fichier	even_man.c
Exemple *Lorsqu'une tâche attend plusieurs événements et qu'elle est réveillée par l'un de ces événements, elle peut connaître le ou les événements postés à l'aide de la fonction GetEvent.*	`TASK(MY_TASK)` `{` `  My_Task_Init();` `  while (1)` `  {` `  WaitEvent(ALARM_EVENT | AN_EVENT);` **`  GetEvent(MY_TASK_ID, &my_task_event);`** `  if (my_task_event & ALARM_EVENT)` `  {` `  ClearEvent(ALARM_EVENT);` `  LATEbits.LATE2 = ~LATEbits.LATE2;` `  }` `  /* ... */` `  }` `}`

WaitEvent

Prototype	`StatusType WaitEvent (EventMaskType Mask)`
Description	Fait passer l'état de la tâche en cours de RUNNING à WAITING. La valeur Mask désigne l'ensemble des événements attendus par la tâche.
Paramètres [in]	*Mask* : ensemble des événements attendus.
Valeur de retour	*E_OK* si aucun des événements attendus n'a déjà été posté. *E_OS_ID* si l'ID ne correspond pas à une tâche de l'application. Aucune si un des événement attendu est présent.
Remarques	Si un des événements attendus a déjà été préalablement posté alors un ré-ordonnancement est forcé : si la tâche en cours est la tâche la plus prioritaire elle poursuit son exécution. Dans le cas contraire c'est la tâche la plus prioritaire qui prend la main. Cette fonction ne peut être appelée depuis une Hook Routine, une ISR ou la fonction main. La valeur Mask est composée de plusieurs événements, chacun étant un multiple de 2. Pour mettre la tâche courante en attente sur plusieurs événements en même temps il est nécessaire de composer la valeur Mask à l'aide d'un masque d'événements. Un seul des événements attendus peut suffire à réveiller la tâche en attente.
Fichier	even_man.c
Exemple *Afin de positionner une tâche en attente d'un événement il est recommandé d'utiliser l'appel système WaitEvent à l'intérieur d'une boucle while.*	

```
TASK(MY_TASK)
{
  My_Task_Init();

  while (1)
  {
    WaitEvent(ALARM_EVENT | AN_EVENT);
    GetEvent(MY_TASK_ID, &my_task_event);

    if (my_task_event & ALARM_EVENT)
    {
      ClearEvent(ALARM_EVENT);
      LATEbits.LATE2 = ~LATEbits.LATE2;
    }
    /* ... */
  }
}
```

5. Gestion des alarmes

La synchronisation des tâches entre elles est un élément essentiel des systèmes multi-tâches, mais ce n'est pas le seul. En effet certaines tâches nécessitent de se synchroniser sur une base de temps telles que les alarmes logicielles. En utilisant une alarme, une tâche va pouvoir être réveillée à des intervalles fixes et pour un nombre de cycles fixes ou infinis. On pourra ainsi facilement créer une tâche qui fera clignoter une LED toutes les 300 ms.

Types utilisés pour la gestion des alarmes :

TickType	Unité de temps de comptage de l'alarme. Valeur 16 bits signée (int *) comprise entre 0 et 65535 inclus. Correspond à 1ms sur un PIC18 cadencé à 40 MHz (10 MHz avec PLLx4).
TickRefType	Permet d'obtenir une valeur de tick d'une alarme. Référence sur une valeur 16 bits signée (int *) comprise entre 0 et 65535 inclus.
AlarmBaseType	Paramètres internes d'une alarme. Ce type n'est pas utilisé dans PICos18.
AlarmBaseRefType	Référence sur les paramètres internes d'une alarme. Ce type n'est pas utilisé dans PICos18.
AlarmType	Identifiant de l'alarme : contient l'Id de l'alarme et l'ID de la tâche qui a créé cette alarme. Valeur 8 bits signée (char) comprise entre 1 et 255 inclus.
AlarmObject	Objet alarme qui possède un compteur de type TickType interne. Une fois que ce compteur atteint une valeur de consigne, l'alarme se déclenche soit en postant un événement soit en activant une tâche. Une alarme n'est pas forcément basée sur un Tick de 1ms. Elle peut être gérée comme un simple compteur qu'il est possible d'incrémenter ou de décrémenter depuis une tâche.
AlarmRefObject	Référence sur une alarme.

Afin de permettre à PICos18 de faire correspondre un tick à 1ms, il faut lui préciser la fréquence à laquelle vous allez faire fonctionner votre PIC18.

Pour cela ouvrez le fichier main.c et modifier la fonction Init() pour y spécifier la valeur de la fréquence interne du PIC18 :

Tmr0.It = _32MHZ;

Vous trouverez les différentres valeurs possible dans le fichier device.h. Ne jamais utiliser PICos18 avec une fréquence interne de moins de 8MHz (quartz 4MHZ sans PLL par exemple).

DeclareAlarm

Prototype	`StatusType DeclareAlarm (AlarmIdentifier)`
Prototype	`DeclareAlarm (EventIdentifier)`
Description	Permet de déclarer le nom d'une alarme à la chaîne de compilation. Est utilisé par le linker au moment de l'édition de lien.
Paramètres [in]	*AlarmIdentifier* : nom de l'événement.
Valeur de retour	/
Remarques	Cette fonction de l'API OSEK-VDX™ n'est pas utilisée dans PICos18. Le nom de l'événement n'est pas traitée par le linker MPLINK de Microchip. Cette fonction n'est pas implémentée dans PICos18.
Fichier	/

GetAlarmBase

Prototype	`StatusType GetAlarmBase (AlarmType AlarmID, AlarmBaseRefType Info)`
Description	Retourne l'ID de la tâche en cours. Appelée depuis une tâche, cette fonction renvoie l'ID de la tâche. Appelée depuis la fonction main, elle retourne INVALID_TASK.
Paramètres [in] **[out]**	*AlarmID* : ID de l'alarme. Index du tableau Alarm_list dans le fichier tascdesc.c. *Info* : Référence sur la structure d'information.
Valeur de retour	*E_OS_ID* si l'ID de l'alarme n'existe pas. *E_OK* dans le cas contraire.
Remarques	Cette fonction renseigne sur les paramètres internes de l'alarme. La structure des paramètres internes d'une alarme existe dans PICos18 pour des raisons de compatibilités avec la norme OSEK-VDX™ mais n'est pas utilisée par le noyau.
Fichier	alarm.c
Exemple	/

GetAlarm

Prototype	`StatusType GetAlarm (AlarmType AlarmID, TickRefType Tick)`
Description	Renseigne le nombre de ticks restant avant que l'alarme ne se déclenche.
Paramètres [in] [out]	*TaskID* : ID de l'alarme interrogée. *Tick* : nombre de ticks restant avant déclenchement de l'alarme.
Valeur de retour	*E_OS_NOFUNC* si l'alarme n'est pas en cours de fonctionnement. *E_OS_ID* si AlarmID ne correspond à aucune alarme. *E_OK* dans le cas contraire.
Remarques	Cette fonction permet par exemple d'annuler le déclenchement d'une alarme par l'appel de CancelAlarm si cela s'avère nécessaire.
Fichier	alarm.c
Exemple *Lorsqu'une alarme se déclenche elle ne s'arrête pas pour autant : elle continue de fonctionner pendant le déroulement de votre application.* *Parfois il peut être intéressant de connaître l'état d'avancement d'une alarme et détecter ainsi si elle risque de se déclencher rapidement. Dans ce cas il est possible de dévalider l'alarme pour la reprogrammer.*	

```
AlarmObject Alarm_list[] =
  {
   {
     OFF,              /* State                    */
     0,                /* AlarmValue               */
     0,                /* Cycle                    */
     &Counter_kernel,  /* ptrCounter               */
     MY_TASK,          /* TaskID2Activate          */
     ALARM_EVENT,      /* EventToPost              */
     0                 /* CallBack                 */
   }
 };
...

#define ALARM_TSK0      0

TASK(MY_TASK)
{
  SetRelAlarm(ALARM_TSK0, 1000, 0);

  while (1)
  {
    WaitEvent(ALARM_EVENT);
    ClearEvent(ALARM_EVENT);

    LATEbits.LATE2 = ~LATEbits.LATE2;
    /* ... */

    GetAlarm(ALARM_TSK0, &Tick);
    if (Tick > 400)
    {
      CancelAlarm(GetAlarmID(TheTimer));
      SetRelAlarm(GetAlarmID(TheTimer), 500, 0);
    }

    /* ... */
  }
}
```

SetRelAlarm

Prototype	`StatusType SetRelAlarm (AlarmType AlarmID, TickType increment, TickType cycle)`
Description	Programme une alarme en lui donnant une période de déclenchement. La programmation est relative ce qui signifie qu'elle se fait par rapport à la valeur courante du nombre de ticks de l'alarme.
Paramètres **[in]** **[in]** **[in]**	*AlarmID* : ID de l'alarme programmée. Index du tableau Alarm_list dans le fichier tascdesc.c. *Increment* : délais du premmier déclenchement. *Cycle* : délais des déclenchements suivants.
Valeur de retour	*E_OS_STATE* si l'alarme est en cours de fonctionnement. *E_OS_ID* si AlarmID ne correspond à aucune alarme. *E_OK* dans le cas contraire.
Remarques	Cette fonction permet de réveiller une tâche de façon cyclique, pour allumer une LED à fréquence fixe par exemple. Si vous souhaitez que l'alarme s'active quelque fois seulement, utilisez une variable locale pour compter le nombre de déclenchements d'alarme et la fonction CancelAlarm pour arrêter l'alarme. L'alarme ainsi programmée ne doit pas être en cours de fonctionnement. Dans le cas contraire appelez la fonction CancelAlarm avant de la reprogrammer.
Fichier	alarm.c
Exemple *SetRelAlarm permet de programmer les délais de déclenchements d'une alarme.* *Dans cet exemple l'alarme est programmée pour se déclencher toutes les 500 ms et ceci au bout de 10 ms.* *La programmation est relative à la valeur courante de l'alarme.*	`AlarmObject Alarm_list[] = { ... };` `...` `#define ALARM_TSK0 0` `TASK(MY_TASK)` `{` `  SetRelAlarm(ALARM_TSK0, 10, 500);` `  while (1)` `  {` `  WaitEvent(ALARM_EVENT);` `  ClearEvent(ALARM_EVENT);` `  LATEbits.LATE2 = ~LATEbits.LATE2;` `  /* ... */` `  }` `}`

SetAbsAlarm

Prototype	`StatusType SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle)`
Description	Programme une alarme en lui donnant une période de déclenchement. La programmation est absolue ce qui signifie qu'elle se fait par rapport à la valeur 0, valeur d'origine de l'alarme au moment de sa création.
Paramètres **[in]** **[in]** **[in]**	*AlarmID* : ID de l'alarme programmée. Index du tableau Alarm_list dans le fichier tascdesc.c. *Start* : valeur absolue de déclenchement. *Cycle* : nombre de déclenchements.
Valeur de retour	*E_OS_STATE* si l'alarme est en cours de fonctionnement. *E_OS_ID* si AlarmID ne correspond à aucune alarme. *E_OK* dans le cas contraire.
Remarques	Cette fonction permet de réveiller une tâche de façon absolue, de façon très précise par rapport au temps du système. Après déclenchement l'alarme continue d'incrémenter son nombre de ticks (de 0 à 65535). Pour utiliser une période plus petite, préférez la fonction SetRelAlarm. Si vous souhaitez que l'alarme s'active quelque fois seulement, utilisez une variable locale pour compter le nombre de déclenchements d'alarme et la fonction CancelAlarm pour arrêter l'alarme. L'alarme ainsi programmée ne doit pas être en cours de fonctionnement. Dans le cas contraire appelez la fonction CancelAlarm avant de la reprogrammer.
Fichier	alarm.c
Exemple *SetRelAlarm permet de programmer les délais de déclenchements d'une alarme par rapport à l'horloge absolue du système.* *Dans cet exemple l'alarme est programmée pour se déclencher lorsque l'horloge système vaut 60000 ms puis s'arrête.* *La programmation est relative à la valeur absolue de l'horloge système.*	`AlarmObject Alarm_list[] = { ... }; ...` `#define ALARM_TSK0 0` `TASK(MY_TASK) { SetAbsAlarm(ALARM_TSK0, 60000, 0); while (1) { WaitEvent(ALARM_EVENT); ClearEvent(ALARM_EVENT); LATEbits.LATE2 = ~LATEbits.LATE2; /* ... */ } }`

CancelAlarm

Prototype	`StatusType CancelAlarm (AlarmType AlarmID)`
Description	Permet de désactiver une alarme. Ceci ne signifie pas que l'alarme ne continue pas de compter, mais plutôt qu'il n'y aura pas de déclenchement lorsque le nombre de ticks atteindra la valeur programmée.
Paramètres [in]	*AlarmID* : ID de l'alarme programmée. Index du tableau Alarm_list dans le fichier tascdesc.c.
Valeur de retour	*E_OS_NOFUNC* si l'alarme est déjà désactivée. *E_OS_ID* si AlarmID ne correspond à aucune alarme. *E_OK* dans le cas contraire.
Remarques	Cette fonction stoppe la capacité de déclenchement d'une alarme. Elle ne remet pas à zéro la valeur de ticks courant de l'alarme. Doit être appelée de préférence avant l'utilisation des autres fonctions de gestion d'alarme (SetAbsAlarm, SetRelAlarm,…) afin d'être sûr que l'alarme ne soit pas en fonctionnement pendant une manipulation.
Fichier	alarm.c
Exemple *CancelAlarm permet de stopper une alarme à tout moment.* *Le nombre de Ticks courant n'est pas remis à zéro, l'alarme est en quelque sorte gelée.* *Pour redémarrer l'alarme il est nécessaire de la reprogrammer à l'aide des fonctions SetRel ou SetAbsAlarm (précédé d'un CancelAlarm pour autoriser la reprogrammation).*	`AlarmObject Alarm_list[] =` `  {` `  ...` `  };` `...` `#define ALARM_TSK0  0` `TASK(MY_TASK)` `{` `  SetRelAlarm(ALARM_TSK0, 500, 500);` `  while (1)` `  {` `  WaitEvent(ALARM_EVENT);` `  ClearEvent(ALARM_EVENT);` `  LATEbits.LATE2 = ~LATEbits.LATE2;` `  /* ... */` `  GetAlarm(ALARM_TSK0, &Tick);` `  if (Tick > 400)` `  {` `  **CancelAlarm(ALARM_TSK0);**` `  SetRelAlarm(ALARM_TSK0, 500, 500);` `  }` `  /* ... */` `  }` `}`

6. Contrôle du noyau

Le noyau ne démarre pas automatiquement au démarrage selon la norme OSEK/VDX™. En effet c'est tout d'abord votre application qui prend la main durant la phase de boot en exécutant la fonction main(). Par la suite il vous incombe d'appeler le noyau pour permettre la gestion de votre application multi-tâches.

Types utilisés pour la gestion du noyau :

AppModeType	Renseigne sur le mode dans lequel l'application fonctionne. Le sens de cette valeur est spécifique à l'application. Valeur 8 bits signée (char) comprise entre 0 et 255 inclus.

GetActiveApplicationMode

Prototype	`AppModeType GetActiveApplicationMode (void)`
Description	La fonction retourne le mode courant de l'application. Cette valeur est stockée dans la variable globale `appMode` du fichier pro_man.c.
Paramètres [in]	/
Valeur de retour	La valeur de `appMode`.
Remarques	Cette fonction permet d'écrire une application sensible à différents mode de fonctionnement comme un mode R&D, VALIDATION, DELIVERY,… Les tâches seraient alors à même de réagir différemment en fonction de la valeur `appMode`.
Fichier	pro_man.c
Exemple	/

StartOS

Prototype	`void StartOS (AppModeType Mode)`
Description	Sauvegarde l'adresse de retour dans la fonction main(). Appel la fonction d'initialisation du noyau PICos18 en affectant la valeur `Mode` à la variable globale `appMode`.
Paramètres [in]	*Mode* : mode fonctionnement de l'application.
Valeur de retour	/
Remarques	Cette fonction est nécessaire au démarrage du noyau PICos18. La valeur `Mode` sert à conserver un indicateur sur le mode fonctionnement de l'application. Ceci permet d'écrire une application sensible à différents mode de fonctionnement comme un mode R&D, VALIDATION, DELIVERY,… Les tâches seraient alors à même de réagir différemment en fonction de la valeur `appMode`.
Fichier	pro_man.c
Exemple	/

ShutdownOS

Prototype	`void ShutdownOS (StatusType Error)`
Description	Permet de retourner à la fonction main() en cas d'arrêt d'urgence.
Paramètres [in]	*Error* : erreur à l'origine de l'instabilité.
Valeur de retour	/
Remarques	En cas de détection d'erreur majeure nécessitant un redémarrage du système, une tâche peut appeler ShutdownOS afin de voir le noyau s'arrêter et retourner à la fonction main(). Le reste des opérations est spécifique à l'application mais de façon générale il est préférable que la fonction main() redémarre l'application complète.
Fichier	pro_man.c
Exemple	/

7. Gestion des interruptions

Pourquoi utiliser un microcontrôleur si l'on ne peut utiliser ses périphériques ? PICos18 est conçu pour vous permettre de bénéficier des interruptions, qui peuvent même s'interfacer avec vos tâches en leur postant des évènements. Les routines d'interruptions, appelées ISR, peuvent elles aussi appeler les services du noyau et ainsi faire office de pre-process pour les tâches associées.

ResumeAllInterrupts

Prototype	`StatusType ResumeAllInterrupts (void)`
Description	Permet de valider les interruptions générales du PIC18. Ce service fonctionne avec le service DisableAllInterrupts.
Paramètres [in]	/
Valeur de retour	/
Remarques	Le PIC18 possède un bit de validation générale des interruptions (GIEL pour Global Interrupt Enable for Low Interrupts et GIEH pour High Interrupts). Lorsque GIEx est positionné, toutes les interruptions de cette catégories sont activées. Lorsque GIEx n'est pas positionné, toutes les interruptions de cette catégories sont désactivées. ResumeAllInterrupts permet de positionner GIEL et GIEH si bien que la totalité des interruptions ne sont plus masquées.
Fichier	int_man.c
Exemple *Certains accès à des périphériques internes ou externes nécessitent de rentrer en région critique, c'est-à-dire de dévalider toutes les interruptions. Pour cela on peut soit faire appel aux services ResumeAllInterrupts et SuspendAllInterrupts.* *Attention Schedule repositionne les interruptions donc il est nécessaire de les dévalider pour rester en région critique.*	`TASK(MY_TASK)` `{` `  unsigned char table[10];` `  Init_Table(&table[0], 10);` `  while (1)` `  {` `  /* ... */` `  /* Enter critical region */` `  DisableAllInterrupts();` `  For (i = 0 ; i < 10 ; i++)` `  {` `  SendEEPROMAddr(i);` `  WriteData(table[i]);` `  Schedule();` `  DisableAllInterrupts();` `  }` `  **ResumeAllInterrupts();**` `  /* Leave critical region */` `  /* ... */` `  }` `}`

SuspendAllInterrupts

Prototype	`StatusType SuspendAllInterrupts (void)`
Description	Permet de dévalider les interruptions générales du PIC18. Ce service fonctionne avec le service EnableAllInterrupts.
Paramètres [in]	/
Valeur de retour	/
Remarques	Le PIC18 possède un bit de validation générale des interruptions (GIEL pour Global Interrupt Enable for Low Interrupts et GIEH pour High Interrupts). Lorsque GIEx est positionné, toutes les interruptions de cette catégories sont activées. Lorsque GIEx n'est pas positionné, toutes les interruptions de cette catégories sont désactivées. SuspendAllInterrupts permet de dévalider GIEL et GIEH si bien que la totalité des interruptions sont désormais masquées.
Fichier	int_man.c
Exemple *Certains accès à des périphériques internes ou externes nécessitent de rentrer en région critique, c'est-à-dire de dévalider toutes les interruptions. Pour cela on peut soit faire appel aux services ResumeAllInterrupts et SuspendAllInterrupts.* *Attention Schedule repositionne les interruptions donc il est nécessaire de les dévalider pour rester en région critique.*	`TASK(MY_TASK)` `{` `  unsigned char table[10];` `  Init_Table(&table[0], 10);` `  while (1)` `  {` `  /* ... */` `  /* Enter critical region */` `  SuspendAllInterrupts ();` `  For (i = 0 ; i < 10 ; i++)` `  {` `  SendEEPROMAddr(i);` `  WriteData(table[i]);` `  Schedule();` `  SuspendAllInterrupts ();` `  }` `  EnableAllInterrupts();` `  /* Leave critical region */` `  /* ... */` `  }` `}`

EnableAllInterrupts

Prototype	`void EnableAllInterrupts (void)`
Description	Permet de valider les interruptions générales du PIC18. Ce service fonctionne avec le service DisableAllInterrupts.
Paramètres [in]	/
Valeur de retour	/
Remarques	Le PIC18 possède un bit de validation générale des interruptions (GIEL pour Global Interrupt Enable for Low Interrupts et GIEH pour High Interrupts). Lorsque GIEx est positionné, toutes les interruptions de cette catégories sont activées. Lorsque GIEx n'est pas positionné, toutes les interruptions de cette catégories sont désactivées. EnableAllInterrupts permet de valider GIEL et GIEH si bien que la totalité des interruptions ne sont pas masquées.
Fichier	int_man.c
Exemple *Certains accès à des périphériques internes ou externes nécessitent de rentrer en région critique, c'est-à-dire de dévalider toutes les interruptions. Pour cela on peut soit faire appel aux services ResumeAllInterrupts et SuspendAllInterrupts.* *Attention Schedule repositionne les interruptions donc il est nécessaire de les dévalider pour rester en région critique.*	

```
TASK(MY_TASK)
{
  unsigned char table[10];
  Init_Table(&table[0], 10);

  while (1)
  {
    /* ... */

    /* Enter critical region */
    DisableAllInterrupts();
    For (i = 0 ; i < 10 ; i++)
    {
      SendEEPROMAddr(i);
      WriteData(table[i]);
      Schedule();
      DisableAllInterrupts();
    }
    EnableAllInterrupts();
    /* Leave critical region */

    /* ... */
  }
}
```

DisableAllInterrupts

Prototype	`void DisableAllInterrupts (void)`
Description	Permet de dévalider les interruptions générales du PIC18. Ce service fonctionne avec le service EnableAllInterrupts.
Paramètres [in]	/
Valeur de retour	/
Remarques	Le PIC18 possède un bit de validation générale des interruptions (GIEL pour Global Interrupt Enable for Low Interrupts et GIEH pour High Interrupts). Lorsque GIEx est positionné, toutes les interruptions de cette catégories sont activées. Lorsque GIEx n'est pas positionné, toutes les interruptions de cette catégories sont désactivées. DisableAllInterrupts permet de dévalider GIEL et GIEH si bien que la totalité des interruptions sont désormais masquées.
Fichier	int_man.c
Exemple *Certains accès à des périphériques internes ou externes nécessitent de rentrer en région critique, c'est-à-dire de dévalider toutes les interruptions. Pour cela on peut soit faire appel aux services ResumeAllInterrupts et SuspendAllInterrupts.* *Attention Schedule repositionne les interruptions donc il est nécessaire de les dévalider pour rester en région critique.*	`TASK(MY_TASK)` `{` `  unsigned char table[10];` `  Init_Table(&table[0], 10);` `  while (1)` `  {` `  /* ... */` `  /* Enter critical region */` `  **DisableAllInterrupts();**` `  For (i = 0 ; i < 10 ; i++)` `  {` `  SendEEPROMAddr(i);` `  WriteData(table[i]);` `  Schedule();` `  **DisableAllInterrupts();**` `  }` `  EnableAllInterrupts();` `  /* Leave critical region */` `  /* ... */` `  }` `}`

ResumeOSInterrupts

Prototype	`void ResumeOSInterrupts (void)`
Description	Permet de dévalider les interruptions basses du PIC18. Ce service fonctionne avec le service SuspendOSInterrupts.
Paramètres [in]	/
Valeur de retour	/
Remarques	Le PIC18 possède un bit de validation générale des interruptions (GIEL pour Global Interrupt Enable for Low Interrupts et GIEH pour High Interrupts). Lorsque GIEx est positionné, toutes les interruptions de cette catégories sont activées. Lorsque GIEx n'est pas positionné, toutes les interruptions de cette catégories sont désactivées. ResumeOSInterrupts permet de valider les interruptions basses du PIC18 après qu'elles aient été dévalidées.
Fichier	int_man.c
Exemple *Les interruptions basses ne peuvent pas interrompre le noyau PICos18 par contre elles peuvent interrompre n'importe quelle tâche.* *Parfois on souhaite pouvoir être interrompu dans la tâche mais pas lorsque l'on effectue des opérations délicates dans le noyau, comme le passage d'une tâche à l'autre. On peut alors appeler le noyau par l'appel « Schedule » et être sûr ainsi de ne pas être interrompu par une IT haute (dite fast interrupt).*	

```
TASK(MY_TASK)
{
  unsigned char table[10];
  Init_Table(&table[0], 10);

  while (1)
  {
    /* ... */

    For (i = 0 ; i < 10 ; i++)
    {
      SendEEPROMAddr(i);
      WriteData(table[i]);
      /* Disable fast interrupts only */
      SuspendOSInterrupts();
      Schedule();
      ResumeOSInterrupts();
    }
    /* Leave critical region */

    /* ... */
  }
}
```

SuspendOSInterrupts

Prototype	`void SuspendOSInterrupts (void)`
Description	Permet de dévalider les interruptions rapides du PIC18. Ce service fonctionne avec le service ResumeOSInterrupts.
Paramètres [in]	/
Valeur de retour	/
Remarques	Le PIC18 possède un bit de validation générale des interruptions (GIEL pour Global Interrupt Enable for Low Interrupts et GIEH pour High Interrupts). Lorsque GIEx est positionné, toutes les interruptions de cette catégories sont activées. Lorsque GIEx n'est pas positionné, toutes les interruptions de cette catégories sont désactivées. SuspendOSInterrupts permet de dévalider les interruptions hautes (rapides) du PIC18.
Fichier	int_man.c
Exemple *Les interruptions basses ne peuvent pas interrompre le noyau PICos18 par contre elles peuvent interrompent n'importe quelle tâche.* *Parfois on souhaite pouvoir être interrompu dans la tâche mais pas lorsque l'on effectue des opérations délicates dans le noyau, comme le passage d'une tâche à l'autre. On peut alors appeler le noyau par l'appel « Schedule » et être sûr ainsi de ne pas être interrompu par une IT haute (dite fast interrupt).*	``` TASK(MY_TASK) { unsigned char table[10]; Init_Table(&table[0], 10); while (1) { /* ... */ For (i = 0 ; i < 10 ; i++) { SendEEPROMAddr(i); WriteData(table[i]); /* Disable fast interrupts only */ SuspendOSInterrupts(); Schedule(); ResumeOSInterrupts(); } /* Leave critical region */ /* ... */ } } ```

8. Gestion des ressources

PICos18 autorise le libre accès à toutes les ressources du PIC18xxx à toutes les tâches de vos applications. Ceci permet de vous simplifier l'accès aux périphériques mais peut engendrer quelques soucis de droit d'accès lorsque ce même périphérique est utilisé par plusieurs tâches en même temps. Ceci est résolu par la gestion des ressources en mode «ceiling protocole » de la norme OSEK/VDX™.

Types utilisés pour la gestion des ressources :

ResourceType	Permet de renseigner les paramètres d'une ressource : `- priority : priorité de la ressource concernée` `- Taskprio : priorité de la tâche accédant à la ressource` `- lock : semaphore d'exclusion mutuelle`

Le principe du « ceiling protocol » est simple :

Une ressource possède une certaine priorité. Supposons que 2 tâches cherchent à accéder à l'EEPROM interne du PIC18. Pour éviter des accès concurrents du fait que l'application est multi-tâches, on attribue à la ressource une priorité supérieure aux 2 autres tâches.

Lorsque l'une des tâches accède à la ressource, sa priorité est modifiée pour prendre la valeur de la priorité de la ressource. Sa priorité est ainsi assurée d'être la plus élevée d'entre les 2 tâches. La ressource ne pourra donc être prise par la tâche de moindre priorité car celle-ci n'aura pas la main, du fait du jeux des priorités.

Toutefois la tâche qui a accès à la ressource peut aussi être ponctuellement suspendue, en appelant par exemple le service WaitEvent. Pour éviter que la tâche de moindre priorité n'en profite pour accéder à la ressource déjà réservée, un champ «lock » a été ajouté afin de permettre afin de vérifier si la ressource n'est pas déjà réservée.

Consultez les tâches du tutorial livrées avec la release de PICos18, vous y trouverez un bon exemple de gestion des ressources au travers des tâches 0 et 1.

DeclareResource

Prototype	`DeclareResource (ResourceIdentifier)`
Description	Permet de déclarer le nom de la ressource à la chaîne de compilation.
Paramètres [in]	*ResourceIdentifier* : nom de la ressource.
Valeur de retour	/
Remarques	**Cette fonction n'est pas implémentée dans PICos18.**
Fichier	/
Exemple	/

GetResource

Prototype	`StatusType GetResource (ResourceType ResID)`
Description	Permet de réserver une ressource par `CeilingProtocol` (OSEK-VDX™).
Paramètres [in]	*ResID* : ID de la ressource. Index de la ressource dans le tableau Resource_list de tascdesc.c.
Valeur de retour	*E_OS_ACCESS* si la ressource est déjà réservée. *E_OS_ID* si la ressource n'existe pas. *E_OK* dans le cas contraire.
Remarques	Cette fonction permet de réserver une ressource.
Fichier	pro_man.c
Exemple *La tâche TASK0 réserve la ressource dont l'index est 0. Désormais sa priorité est élevée à 10 de sorte que les autres tâches ne peuvent y accéder à leur tour.* *Toutefois une autre tâche ayant réservée la ressource pourrait avoir été suspendue. Il est donc nécessaire de vérifier si la ressource est verrouillée ou non avant d'y accéder réellement.*	`Resource Resource_list[] =` `  {` `   {` `  10,  /* priority  */` `  0,  /* Task prio  */` `  0,  /* lock  */` `   }` `  };` `...` `#define ALARM_TSK0  0` `TASK(TASK0)` `{` `  unsigned char value;` `  SetRelAlarm(ALARM_TSK0, 20, 20);` `  while(1)` `  {` `  WaitEvent(ALARM_EVENT);` `  ClearEvent(ALARM_EVENT);` `  if (GetResource(0) == E_OK)` `  {` `  value++;` `  ReleaseResource(0);` `  }` `  }` `}`

ReleaseResource

Prototype	`StatusType ReleaseResource (ResourceType ResID)`
Description	Permet de libérer une ressource par `CeilingProtocol` (OSEK-VDX™).
Paramètres [in]	*ResID* : ID de la ressource. Index de la ressource dans le tableau Resource_list de tascdesc.c.
Valeur de retour	*E_OS_ACCESS* si la ressource est déjà réservée. *E_OS_ID* si la ressource n'existe pas. *E_OK* dans le cas contraire.
Remarques	Cette fonction permet de réserver une ressource.
Fichier	pro_man.c
Exemple *La tâche TASK0 réserve la ressource dont l'index est 0. Désormais sa priorité est élevée à 10 de sorte que les autres tâches ne peuvent y accéder à leur tour.* *Toutefois une autre tâche ayant réservée la ressource pourrait avoir été suspendue. Il est donc nécessaire de vérifier si la ressource est verrouillée ou non avant d'y accéder réellement.*	

```
Resource Resource_list[] =
  {
   {
      10,             /* priority           */
       0,             /* Task prio          */
       0,             /* lock               */
   }
  };
...

#define ALARM_TSK0      0

TASK(TASK0)
{
  unsigned char value;
  SetRelAlarm(ALARM_TSK0, 20, 20);

  while(1)
  {
    WaitEvent(ALARM_EVENT);
    ClearEvent(ALARM_EVENT);
    if (GetResource(0) == E_OK)
    {
      value++;
      ReleaseResource(0);
    }
  }
}
```

9. Hook routines

Durant l'exécution de votre application il est toujours possible de contrôler son bon déroulement grâce à des « sondes logicielles », les Hook routines, véritables capteurs liés à vos tâches.

Types utilisés pour la gestion des Hook routines :

OSServiceIdType	Permet de connaître la dernier service traité par le noyau. **Ce type n'est pas implémentée dans PICos18.**

ErrorHook

Prototype	`void ErrorHook (StatusType Error)`
Description	Cette Hook routine est appelée systématiquement par le noyau lorsqu'un service retourne un code d'erreur (valeur de retour différente de E_OK).
Paramètres [in]	Erreur retournée par le service en défaut.
Valeur de retour	/
Remarques	**Cette fonction n'est pas implémentée dans PICos18.**
Fichier	main.c
Exemple	/

PreTaskHook

Prototype	`void PreTaskHook (void)`
Description	Cette Hook routine est appelée avant de sauter dans la prochaine tâche active.
Paramètres [in]	/
Valeur de retour	/
Remarques	Cette fonction permet l'exécution d'un certain nombre d'instructions avant de rentrer dans la prochaine tâche active. Vous pouvez ainsi détecter la rentrée dans une tâche en particulier ou bien sauvegarder certains paramètres avant qu'ils ne soient modifiés par la tâche. La fonction PreTaskHook n'est à utiliser que pendant la phase de mise au point de votre code, elle n'est pas destinée à demeurer dans la version finale de votre application. Pour activer la fonction PreTaskHook, compilez votre projet avec l'option « PRETASKHOOK ». Pour cela rajoutez « -DPRETASKHOOK » dans la ligne de commande du panneau des options de compilation. Consultez la documentation de C18 pour obtenir plus d'information sur la ligne de commande du compilateur.
Fichier	main.c
Exemple *Ici nous utilisons la fonction PreTaskHook pour vérifier le fait que l'on rentre bien dans la tâche TASK_COUNTER.* *La fonction PreTaskHook sera appelée par le noyau à chaque fois qu'une nouvelle tâche sera activée. Il suffit alors d'obtenir l'ID de la tâche activée et de faire clignoter une LED dans le cas où cette tâche est TASK_COUNTER.*	`/*******************************************************/` `/* From the main.c file                                 */` `/*******************************************************/` `...` `/*******************************************************` `* Hook routine called just before entering in a task.` `*` `* @return void` `*******************************************************/` `void PreTaskHook(void)` `{` `  unsigned char task_ID;` `  task_ID = GetTaskID();` `  if (task_ID == TASK_COUNTER)` `  {` `  LATEbits.RE1 = ~LATEbits.RE1 ;` `  }` `  return ;` `}`

PostTaskHook

Prototype	`void PostTaskHook (void)`
Description	Cette Hook routine est appelée juste après avoir quitté la dernière tâche active.
Paramètres [in]	/
Valeur de retour	/
Remarques	Cette fonction permet l'exécution d'un certain nombre d'instructions avant de rentrer dans le noyau. Vous pouvez ainsi détecter la sortie d'une tâche en particulier ou bien sauvegarder les valeurs de certaines variables au moment de quitter la tâche. La fonction PostTaskHook n'est à utiliser que pendant la phase de mise au point de votre code, elle n'est pas destinée à demeurer dans la version finale de votre application. Pour activer la fonction PostTaskHook, compilez votre projet avec l'option « POSTTASKHOOK ». Pour cela rajoutez « -DPOSTTASKHOOK » dans la ligne de commande du panneau des options de compilation. Consultez la documentation de C18 pour obtenir plus d'information sur la ligne de commande du compilateur.
Fichier	main.c
Exemple *Ici nous utilisons la fonction PostTaskHook pour vérifier le fait que l'on quitte bien la tâche TASK_COUNTER.* *La fonction PreTaskHook allume une LED sur RE1 lorsque l'on rentre dans TASK_COUNTER. La fonction PostTaskHook éteint la même LED lorsque l'on quitte la tâche.*	

```
/*******************************************************/
/* From the main.c file                                */
/*******************************************************/
...

void PreTaskHook(void)
{
  unsigned char task_ID;

  task_ID = GetTaskID();
  if (task_ID == TASK_COUNTER)
    LATEbits.RE1 = 1;
}

/*******************************************************
 * Hook routine called just after leaving a task.
 *
 * @return void
 *******************************************************/
void PostTaskHook(void)
{
  unsigned char task_ID;

  task_ID = GetTaskID();
  if (task_ID == TASK_COUNTER)
    LATEbits.RE1 = 0;
}
```

StartupHook

Prototype	`void StartupHook (void)`
Description	Cette Hook routine est appelée juste après terminé la phase d'initialisation du noyau. Cette fonction ne sera traitée qu'une seule fois, au moment de l'initialisation du noyau.
Paramètres [in]	/
Valeur de retour	/
Remarques	Cette fonction permet l'exécution d'un certain nombre d'instructions avant de rentrer dans le noyau. Vous pouvez par exemple initialiser des variables de votre application à une certaine valeur. La fonction StartUpHook n'est à utiliser que pendant la phase de mise au point de votre code, elle n'est pas destinée à demeurer dans la version finale de votre application. Pour activer la fonction StartUpHook, compilez votre projet avec l'option « STARTUPHOOK ». Pour cela rajoutez « -DSTARTUPHOOK » dans la ligne de commande du panneau des options de compilation. Consultez la documentation de C18 pour obtenir plus d'information sur la ligne de commande du compilateur.
Fichier	main.c
Exemple *Dans cet exemple StartUpHook est utilisée pour initialiser un tableau de nombres pairs.* *Il est aussi possible d'appeler des services du noyau depuis la fonction StartUpHook. Toutefois, comme les tâches ne sont pas encore activées par le noyau, la plupart des services n'auront aucun effet.*	

```
/*******************************************************/
/* From the main.c file                                */
/*******************************************************/
...

extern unsigned char even_tab[8];

...

/*******************************************************
 * Hook routine called before entering into the kernel
 *
 * @return void
 *******************************************************/
void StartUpHook(void)
{
  unsigned char index;

  for (index = 0; index < 8; index++)
  {
    even_tab[index] = index * 2;
  }
}
```

ShutdownHook

Prototype	`void ShutdownHook (StatusType Error)`
Description	Cette Hook routine est avant le traitement de la fonction ShutDownOS. Elle permet notamment de connaître l'origine de la panne qui a provoquée l'arrêt du système.
Paramètres [in]	Erreur passée en paramètre à la fonction ShutDownOS.
Valeur de retour	/
Remarques	Cette fonction permet l'exécution d'un certain nombre d'instructions avant de rentrer dans la fonction ShutDownOS. La fonction ShutDownOS reçoit une erreur en paramètre, cette erreur est retransmise à la fonction ShutdownHook afin de lui permettre d'identifier l'origine de la panne. Pour activer la fonction ShutdownHook, compilez votre projet avec l'option « SHUTDOWNHOOK ». Pour cela rajoutez « -DSHUTDOWNHOOK » dans la ligne de commande du panneau des options de compilation. Consultez la documentation de c18 pour obtenir plus d'information sur la ligne de commande du compilateur.
Fichier	main.c
Exemple *Ci-contre la fonction ShutdownHook sert enregistrer la dernière erreur reçue dans un tableau d'erreur, puis d'enregistrer en EEPROM le contenu du tableau.* *Il faut envisager la routine ShutdownHook comme la dernière fonction de l'application traitée avant le redémarrage du système. Toutes les variables seront donc réinitialisées par la suite.*	

```
/*******************************************************/
/* From the main.c file                                */
/*******************************************************/
...

extern unsigned char error_tab[10];
extern unsigned char error_index;

...

/*******************************************************
 * Hook routine called just after leaving the kernel.
 *
 * @return void
 *******************************************************/
void ShutdownHook(StatusType error)
{
  even_tab[error_index] = error;

  WriteInEEPROM(error_tab, error_index);
}
```