

Pragmatec

Produits et services dédiés aux systèmes embarqués temps-réel

PICos18

Noyau temps réel pour PIC18

Tutorial & Manuel du Développeur

Page laissée intentionnellement blanche.

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 garantit 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	5
UN NOYAU MULTI-TACHES TEMPS REEL	5
PICos18 : UN NOYAU MULTI-TACHES TEMPS REEL POUR PIC18	6
LA NORME OSEK/VDX™	6
LES CARACTERISTIQUES DE PICos18	7
LA LICENCE GPL	8
LA CHAINE DE COMPILEATION MICROCHIP	9
LA SOCIETE PRAGMATEC	10
2. PICOS18 : TUTORIAL	11
L'ENVIRONNEMENT DE DEVELOPPEMENT	11
CREATION D'UNE PREMIERE TACHE	11
LA PREEMPTION	11
LE MULTI-TACHES	11
LES INTERRUPTIONS	11
UTILISATION DES DRIVERS	12
EXEMPLE D'APPLICATION	12
3. L'ENVIRONNEMENT DE DEVELOPPEMENT	13
4. CREATION D'UNE PREMIERE TACHE	18
5. LA PREEMPTION	24
6. LE MULTI-TACHES	31
7. LES INTERRUPTIONS	35
8. UTILISATION DES DRIVERS	42
9. EXEMPLE D'APPLICATION	47
10. BIBLIOGRAPHIE ET REFERENCES	52
11. GLOSSAIRE	53

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 fonctionnalités les plus connues 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/VDX

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 ».

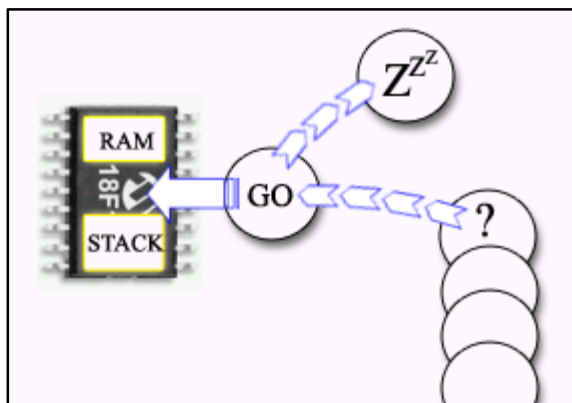
La fonctionnalité de l'OS OSEK a été combiné avec VDX.

Aujourd'hui utilisée dans l'automobile et la robotique, cette norme définit 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 cœur 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 royauté 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.
/*
/*
/* **** */
```

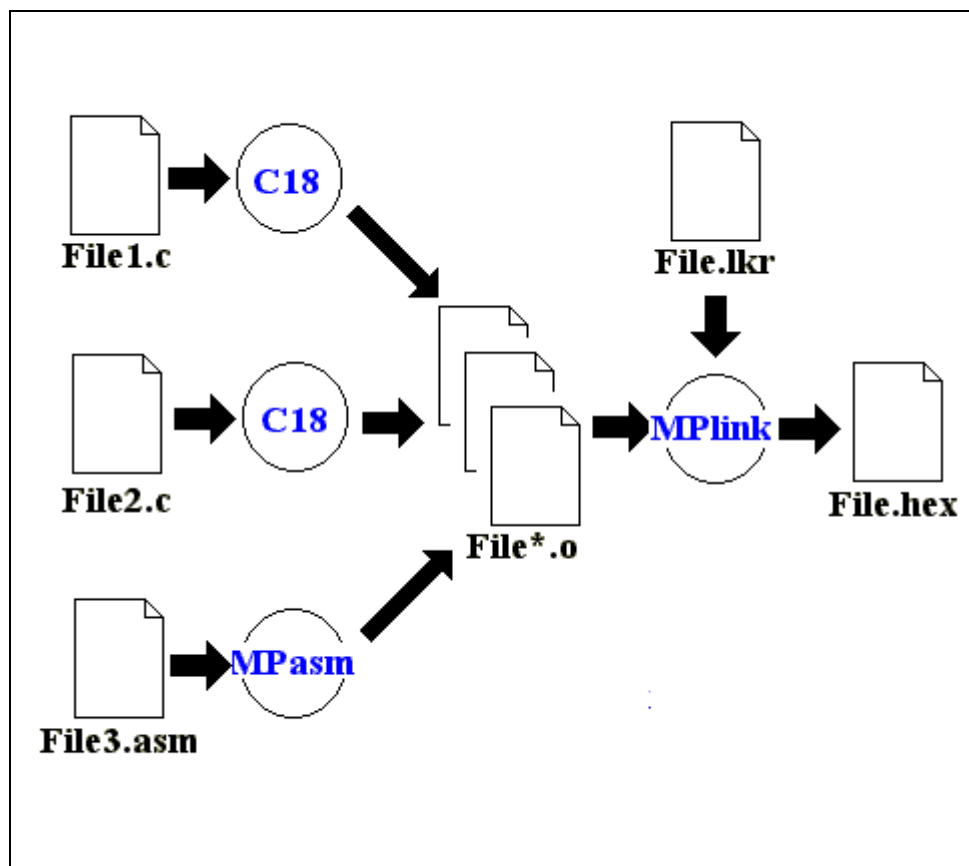

La licence GPL garantie la libre circulation des sources de PICos18, sans restrictions ou protections imposées par une quelconque société ou organisation. En revanche, la société PRAGMATEC, à l'origine de PICos18, s'engage à maintenir le noyau et à le faire évoluer, tout en respectant les règles de la 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 linqueur MPLINK qui permet de fusionner tous les fichiers O en un unique fichier HEX

Le fichier dit "script du linqueur" (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 linqueur 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

Ce tutorial est la propriété de PRAGMATEC S.A.R.L.

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. PICOS18 : Tutorial

L'objectif de ce tutorial est de montrer comment simplement programmer avec PICos18. La norme OSEK définit non seulement les fonctionnalités du noyau, mais aussi la façon dont il doit être interfacé; avec les programmes utilisateurs. Suivez ce tutorial pour apprendre à configurer MPLAB® à utiliser PICos18 et à programmer des applications multi-tâches sur PIC18.



[L'environnement de développement](#)

Dans ce premier chapitre vous mettrez au point un projet basé sur PICos18. Pour cela vous manipulerez l'environnement de développement MPLAB® de Microchip ainsi que le compilateur C18.



[Création d'une première tâche](#)

Maintenant que le décor est planté il est temps de créer et de simuler une première tâche. Vous apprendrez comment l'écrire et comment la déclarer au noyau. De plus vous découvrirez les alarmes et le tick système à 1ms.



[La préemption](#)

L'application mono-tâche que vous venez de réaliser n'a que très peu de raison d'être... En ajoutant à votre application une seconde tâche vous découvrirez les mécanismes de préemption d'une tâche sur l'autre.



[Le multi-tâches](#)

Ce chapitre présente les mécanismes de synchronisation entre tâches de PICos18. Une troisième tâche viendra compléter l'application et postera un événement à une autre tâche afin de l'activer. Vous apprendrez à utiliser les événements et à partager les ressources.



[Les interruptions](#)

Afin de pouvoir tirer parti du microcontrôleur PIC18, il est important de pouvoir utiliser les périphériques. De ce point de vue, les interruptions sont essentielles. Vous verrez comment les interruptions du PIC18 sont mises en œuvre sous PICos18 et comment écrire simplement vos propres routines d'interruption.



Utilisation des drivers

Vous souhaitez faire communiquer votre application au travers du port série ou bien du port CAN ? A présent que vous savez écrire des interruptions et des tâches, et que vous maîtrisez la gestion des événements et des alarmes, il vous sera facile d'utiliser ou d'écrire un véritable driver pour PICos18.



Exemple d'application

Le tutorial se termine ici par un exemple complet d'application. Les sources de cet exemple sont fournis dans le répertoire /Project/Tutorial de PICos18, ceci vous permettant d'apprécier à quel point il est aisé d'écrire une application pour PIC18 avec PICos18.

3. L'environnement de développement

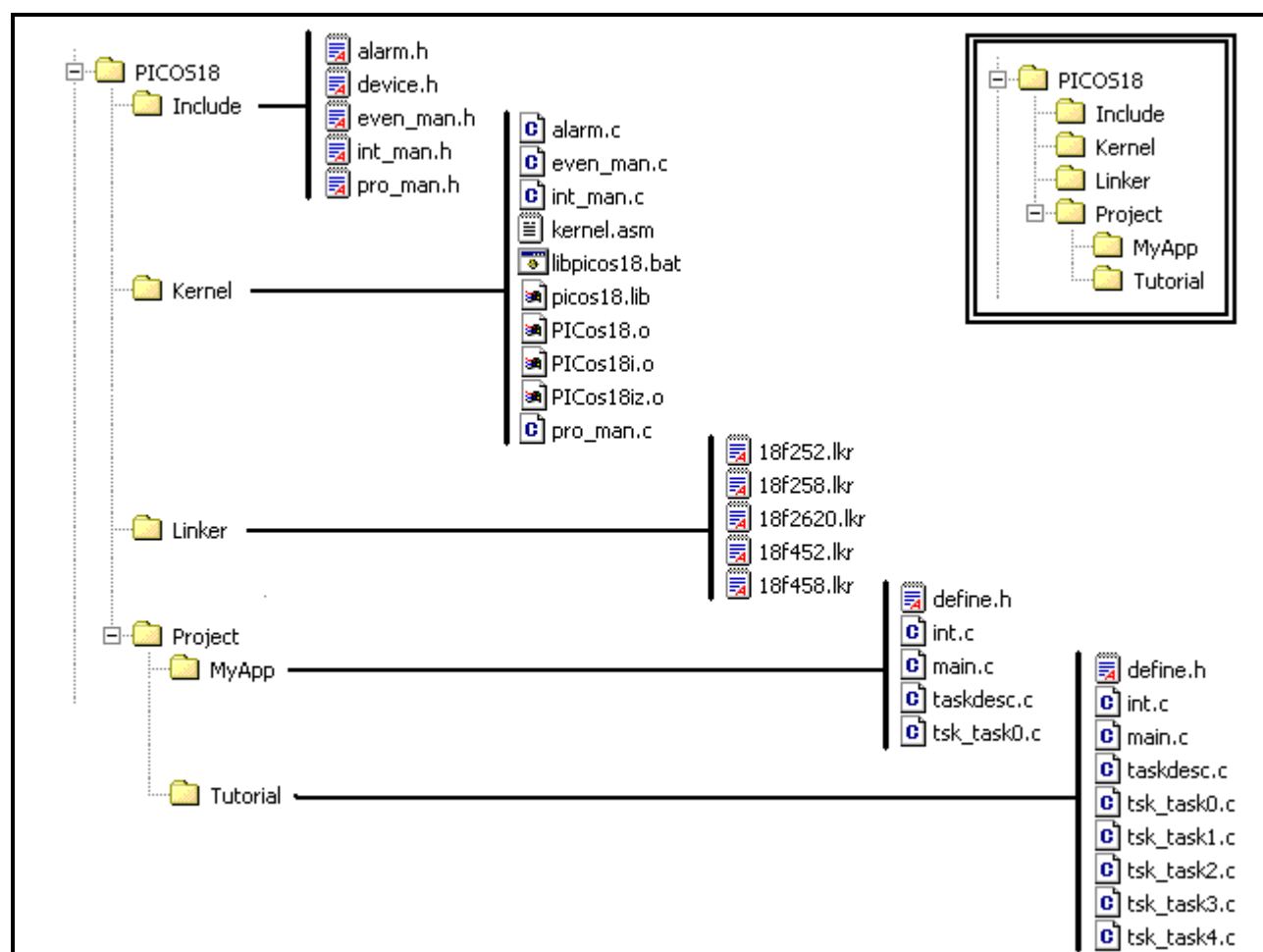


Dans ce premier chapitre vous mettrez au point un projet basé sur PICos18. Pour cela vous manipulerez l'environnement de développement MPLAB® de Microchip ainsi que le compilateur C18.

> Les sources du projet

Avant de lancer MPLAB® téléchargez les sources du tutorial depuis www.picos18.com (section DOWNLOAD), comprenant les fichiers sources du noyau et de l'application.

Voici à quoi correspond chacun de ces fichiers :



Include	Comme son nom l'indique ce répertoire contient l'ensemble des headers du noyau PICos18. Les headers propres à l'application et aux drivers se trouveront dans les répertoires propres aux applications (sous Project). Il existe un header spécifique pour chaque sous-ensemble du noyau : alarmes, événements, interruption et processus. Le fichier device.h contient les définitions génériques du noyau comme les définitions des valeurs de retour des fonctions du noyau.
Kernel	Vous trouverez sous ce répertoire toutes les sources du noyau. Les fichiers C correspondent aux services du noyau (l'API), et le fichier kernel.asm, seul fichier assembleur de PICos18, réunit l'ensemble des fonctions propres au noyau, c'est-à-dire les algorithmes de scheduler, l'ordonnancement des tâches et le gestionnaire d'erreurs. Les sources sont fournies à titre d'information ou si vous souhaitez les modifier pour vos besoins propres. Pour vous permettre d'utiliser PICos18 plus facilement, nous y avons ajouter le noyau sous forme de librairie : picos18.lib. De plus les codes d'amorce du compilateur (runtime) fournis par Microchip ne sont pas adaptés à PICos18, nous avons donc choisi de les adapter en conséquence. Ils sont fournis sous forme de fichier .o (comme PICos18iz.o). Vous n'avez pas à vous soucier de la librairie picos18.lib et des fichiers d'amorce, ils sont automatiquement associés à votre projet par PICos18 !
Linker	Lorsque vous construisez votre application avec PICos18, vous écrivez votre code en langage C. Par la suite il convient de compiler votre application avec C18 (créer chaque fichier .o correspondant à chaque fichier .c) puis de les lier entre eux à l'aide du linker. La façon de lier les fichiers .o est paramétrable, et elle est décrite dans les fichiers .lkr de ce répertoire. Pour chaque type de processeurs remarquables dans la famille PIC18, il est fourni un fichier lkr pour vous éviter de gérer cette partie délicate de l'étape de compilation. Pour chaque nouveau PIC18 géré par un PICos18, un nouveau script de linker se verra ajouter à ce répertoire au fil du temps.
Project/MyApp	Ce répertoire contient les fichiers nécessaires à la création de tout projet avec PICos18. Comme vous pouvez le constater il n'y a que très peu de fichiers : le main.c qui est le premier fichier applicatif exécuté par PICos18, le fichier int.c qui rassemble les routines d'interruptions et d'interruptions rapides, le fichier taskdesc.c qui décrit précisément tous les éléments de votre application (alarmes, compteurs, caractéristiques de vos tâches, ...), et un fichier C par tâche de l'application.
Project/Tutorial	Le tutorial s'appuie sur la réalisation d'une application type sous PICos18. Tout au long des différents chapitres vous allez découvrir comment programmer sous PICos18 et comment construire vos applications. Les fichiers présents sous ce répertoire sont ceux de l'application finale, une fois terminée. Ceci apporte un support concret à ce tutorial en vous fournissant le code source de l'application de test.

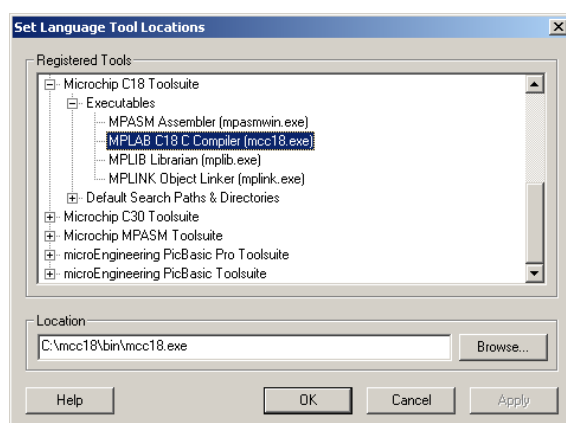
Dé zippez l'ensemble des sources directement sous C:\. Le reste du tutorial suppose que le répertoire PICos18 se trouve bel et bien sous C:\. Dans le cas contraire, effectuez par vous même les rectifications qui s'imposent.

Ce tutorial est pour le moment destiné à un développement sous MPLAB® / Windows™ et a été validé sous MPLAB® v6.62 et v7.00, et le compilateur C de Microchip en version 2.40. Vous pouvez télécharger les versions gratuites de ces outils depuis le site de Microchip. La version gratuite de C18 correspond à une version sans gestion des optimisation par le compilateur, or il ne faut surtout pas utiliser les optimisations de C18 avec PICos18, certaines n'étant pas adaptés dans le cadre d'une utilisation multi-tâches.

C18 en version gratuite est donc un compilateur qui convient parfaitement pour PICos18 et les PIC18, le code assembleur généré étant parfaitement optimisé pour les composants PIC18.

Le composant cible est un PIC18F452 (n'oubliez pas de sélectionner le composant sous MPLAB®).

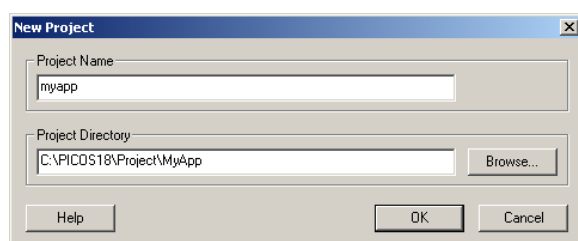
> Compilateur C18



Commencez par installer MPLAB® et le compilateur Microchip pour PIC18 en version gratuite. Par défaut nous supposons que le chemin d'installation est "C:\mcc18\".

Sous MPLAB®, cliquez sur "Project / Set Language Tool Location..." et paramétrez MPASM, MPLAB C18 et MPLINK.

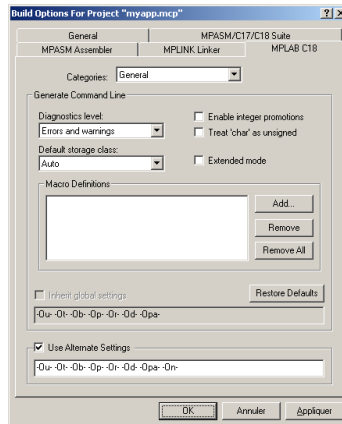
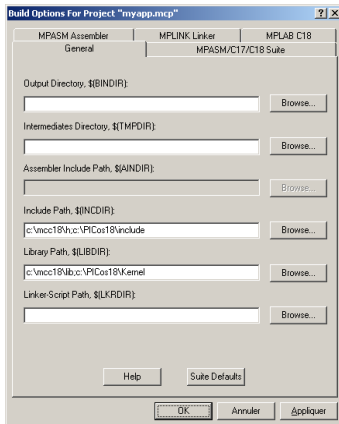
> Création de projet



Puis créer un nouveau projet toujours sous MPLAB®. Cliquez sur "Project / New..." et nommez votre projet avec un nom de moins de 8 caractères. Attention le chemin du répertoire du projet ne doit pas contenir d'espace !!

Puis cliquez sur "Project / Select Language Toolsuite..." et sélectionnez "Microchip C18 Toolsuite".

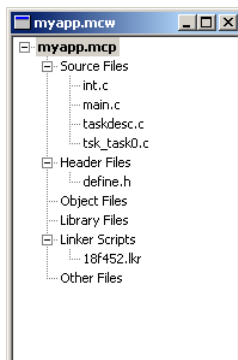
> Les options du projet



Cliquez sur "Project / Build Options..." et précisez les chemins d'inclusion et celui des bibliothèques du compilateur comme du noyau. Attention à ne pas mettre d'espace entre chaque chemin !

Ensuite sélectionnez l'onglet "MPLAB C18" et cochez l'option "Use alternate settings" et ajoutez l'option "-On-" dans la ligne de commande. Ceci à pour effet d'éviter l'optimisation des sélections de bank, non fonctionnelle dans le cas de PICos18.

> Compléter le projet

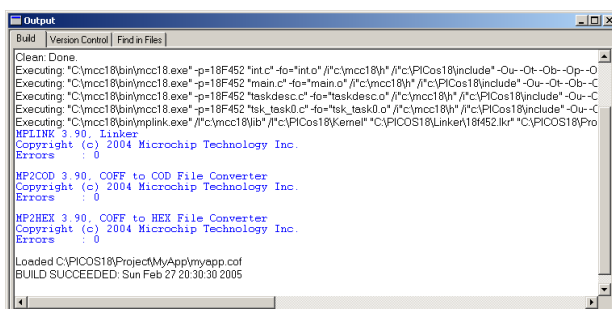


Puis à l'aide du bouton droit de la souris, ajoutez les fichiers suivants :

- **int.c**, **main.c**, **taskdesc.c** et **tsk_task0.c** présents sous MyApp
- **define.h** présent sous MyApp
- **18f452.lkr** présent sous le répertoire Linker

Les fichiers picos18.lib et PICos18iz.o seront trouvés par la chaîne de compilation automatiquement.

> Compilation du projet



Compiler le projet en appuyant sur F10.

Le compilateur va compiler un à un les fichiers C de votre application. Il ne compilera pas les fichiers du noyau PICos18, la bibliothèque étant fournie.

Enfin le linker associe tous les fichiers .o du projet pour créer les fichiers nécessaires à la simulation et la programmation.

> Dévalider le Watchdog

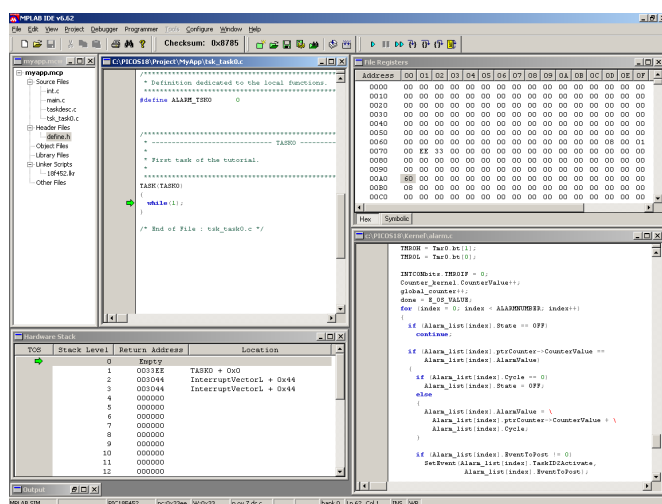
Address	Value	Category	Setting
300001	26	Oscillator	HS-Pll Enabled
		Osc. Switch Enable	Disabled
300002	0D	Power Up Timer	Disabled
		Brown Out Detect	Disabled
		Brown Out Voltage	2.5V
300003	0E	Watchdog Timer	Disabled-Controlled by SWDTEN bit
		Watchdog Postscaler	1:128
300005	01	CCP2 Mux	RC1
300006	80	Stack Overflow Reset	Disabled
		Low Voltage Program	Disabled

La simulation sous MPLAB® est basée sur les paramètres des **bits de configuration** du PIC18.

Paramétrez votre simulation comme indiqué sur la capture d'écran. **Le watchdog doit être obligatoirement dévalidé.**

> Simulation du projet

Finalement lancez la simulation en cliquant sur "Debugger / Select Tool / MPLAB® SIM", puis **F9** (Run). Attendez quelques secondes avant d'appuyez sur **F5** (Stop).



La **simulation** a du s'arrêter sur l'instruction "while(1)" de la tâche 1 comme le montre l'image de droite. Pour autant cette instruction ne bloque pas le PIC18 dans une boucle infinie, car si besoin le noyau peut fonctionner, celui-ci ayant toujours la main sur l'application : essayez plusieurs tentatives pour vous arrêter dans une partie du noyau.

Le fichier source du noyau dans lequel vous êtes arrêté va apparaître à l'écran. Pourtant les sources de PICos18 n'ont jamais été ajoutées au projet MyApp !

En réalité, ceci est du au fait que vous avez positionné PICos18 sous C:\ et que la librairie a été compilée selon ce chemin précis...

4. Création d'une première tâche



Maintenant que le décor est planté il est temps de créer et de simuler une première tâche. Vous apprendrez comment l'écrire et comment la déclarer au noyau. De plus vous découvrirez les alarmes et le tick système à 1ms.

> Pré-requis

Tout d'abord il est nécessaire d'avoir créé un projet sous MPLAB®, projet qui contient les **sources de PICos18**.

Ensuite il faudra avoir installé le compilateur **C18** de Microchip, ainsi que l'avoir paramétré (chapitre précédent).

> Etat d'une tâche

Dans l'état actuel notre projet ne comporte qu'une seule tâche définie dans le fichier "tsk_led.c" :

```
TASK(TASK0)
{
    while(1);
}
```

On comprend facilement que le rôle de la tâche est de créer une boucle infinie.

Sous MPLAB®, placer un point d'arrêt (breakpoint) sur la ligne en face de l'instruction "while(1);", puis lancer la simulation (F9).

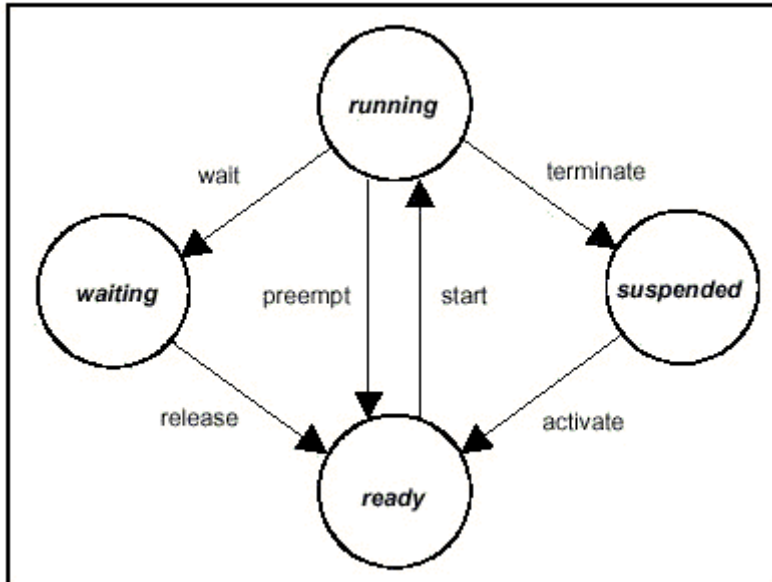
Vous constaterez que la simulation s'arrête sur votre breakpoint. Mais comment fait le noyau pour connaître l'existence de votre tâche, exécuter son code, s'arrêter sur la boucle infinie ... et pour autant ne pas être bloquée par cette instruction ?

En fait la tâche a été déclarée auprès du noyau dans le fichier "**tascdesc.c**".

```
/* *****
 * ----- task 0 -----
 * *****
rom_desc_tsk rom_desc_task0 = {
    TASK0_PRIO,          /* prioint from 0 to 15      */
    stack0,              /* stack address (16 bits)   */
    TASK0,               /* start address (16 bits)   */
    READY,               /* state at init phase      */
    TASK0_ID,            /* id_tsk from 1 to 15      */
    sizeof(stack0),      /* stack size (16 bits)     */
};
```

Sur la ligne en gras vous pouvez indiquer l'état de la tâche lors de son initialisation. En effet même si une tâche existe dans le projet (elle a été compilée par le compilateur), elle n'est pas forcément active.

Dans PICos18, au respect de la norme OSEK, une tâche possède 4 états possibles :



SUSPENDED : la tâche est présente dans le projet, mais n'est pas prise en compte par le noyau.

READY : la tâche est prête à être activée par le noyau, qui désormais la prend en compte.

WAITING : la tâche est en sommeil, elle est temporairement SUSPENDED et sera READY dès qu'un évènement viendra la réveiller.

RUNNING : parmi toutes les tâches prêtes, c'est celle-ci qui occupe la processeur pendant un temps T.

En ce qui concerne votre tâche, elle se trouve en fait dans l'état READY. Tant qu'elle y restera, le noyau pourra y accéder...

Remplacer **READY** par **SUSPENDED** dans le fichier tascdesc.c, recompilez et relancez la simulation: vous constaterez que la simulation ne s'arrête plus sur le breakpoint.

> Description des tâches

Tout comme l'état de la tâche, de nombreuses choses sont décrites dans le fichier tascdesc.c. En fait tout ce qui concerne votre application se trouve dans ce fichier, comme par exemple : l'identifiant de la tâche, sa priorité, sa pile logicielle, les alarmes nécessaires au projet, ...

On trouve tout d'abord la liste des alarmes de l'application:

```

AlarmObject Alarm_list[] =
{
    /* ----- First task ----- */
    {
        OFF, /* State */
        0, /* AlarmValue */
        0, /* Cycle */
        &Counter_kernel, /* ptrCounter */
        TASK0_ID, /* TaskID2Activate */
        ALARM_EVENT, /* EventToPost */
        0 /* CallBack */
    },
};

```

Ce tableau constitue la liste des alarmes, chaque alarme étant représentée par une structure. Une alarme est une sorte d'objet TIMER géré de façon logicielle par le noyau. En fait PICos18 possède une référence de temps de 1ms, générée de façon hardware et logicielle. De cette façon il est possible de créer une alarme logicielle basée sur cette base de temps de 1ms : les alarmes.

Une alarme est associé à un compteur (ici Counter_kernel qui est la référence de 1ms) et une tâche (dont l'identifiant est TASK0_ID). Lorsque l'alarme atteint une valeur de seuil elle peut poster un événement à cette tâche (ALARM_EVENT dans cet exemple) ou bien appeler une fonction en C (mécanisme de Callback).

On trouve ensuite la liste des ressources de l'application:

```

/*****
* ----- COUNTER & ALARM DEFINITION -----
*****/
Resource Resource_list[] =
{
{
    10,                /* priority      */
    0,                 /* Task prio    */
    0,                 /* lock         */
}
};

```

Une ressource est un autre objet géré par le noyau. Une ressource est généralement un périphérique partagé par plusieurs tâches, par exemple un port 8 bits du PIC18. Elle possède une priorité (priority) et peut être verrouillée (lock) par une tâche (Task prio). Ainsi lorsqu'une tâche veut avoir accès au port 8 bits, elle y accède via la ressource définie et empêche ainsi pendant un cours instant à toute autre tâche d'y avoir accès. Les ressources seront vu dans le chapitre lié au multi-tâches.

On trouve ensuite la liste des piles logicielles de l'application:

Il existe 2 types de variables : les variables statiques et les variables automatiques...de façon plus compréhensible nous parlerons de variables globales et de variables locales, même si la correspondance n'est pas tout à fait exacte. Les variables globales (c'est-à-dire placées en dehors de toutes fonctions C) sont placées en RAM à une adresse absolue, adresse qui ne change jamais.

Les variables locales (déclarées au sein de fonctions C) sont elles compilées par C18 de sorte de se retrouver stocker dans une zone mémoire dite pile logicielle. Une pile logicielle est une zone mémoire dynamique, vivante en quelques sortes. Lorsque l'on rentre dans une fonction C, on commence par empiler sur cette zone les variables locales, et lorsque l'on quitte la fonction C, on dépile l'espace réservé aux variables locales libérant ainsi de la mémoire. Ce type de fonctionnement permet de gagner beaucoup de place mémoire lorsqu'il existe de nombreuses fonctions écrites en C, au prix d'un surplus de code minimal.

PICos18 est compatible avec la gestion de la pile logicielle faite par C18. Chaque tâche possède alors sa propre pile logicielle, si bien que chacune des tâches de l'application peut travailler de façon autonome dans son propre espace de travail, sans perturber les autres tâches. De plus PICos18 possède un mécanisme de détection de débordement mémoire, ce qui arrive lorsqu'une pile logicielle commence à déborder sur celle d'une autre tâche (kernelpanic).

```

/*****
* ----- TASK & STACK DEFINITION -----
*****/
#define DEFAULT_STACK_SIZE      128
DeclareTask(TASK0);

volatile unsigned char stack0[DEFAULT_STACK_SIZE];

```

Chaque fois que vous ajoutez une tâche à votre application, pensez à ajouter une pile en copiant la ligne ci-dessus en en changeant le nom de la nouvelle pile (par stack1 par exemple).

- Une taille de pile minimale sous PICos18 est 64.
- Une taille de pile maximale est 256.
- Une taille de pile raisonnable est 128.

Seules ces 3 valeurs sont possibles, n'essayez pas d'utiliser une taille de valeur intermédiaire, la compilateur C18 ne pourrait pas la gérer. De plus une pile doit toujours se trouver sur une même bank, et pas à cheval sur 2 banks. En cas de doutes, laissez toutes vos tailles de pile à 128.

On trouve ensuite la liste des tâches de l'application:

```

/*****
* ----- task 0 -----
*****/
rom_desc_tsk rom_desc_task0 = {
    TASK0_PRIO,          /* prioinit from 0 to 15      */
    stack0,              /* stack address (16 bits)    */
    TASK0,               /* start address (16 bits)    */
    READY,               /* state at init phase        */
    TASK0_ID,            /* id_tsk from 1 to 15        */
    sizeof(stack0)       /* stack size (16 bits)       */
};

```

Nous avons déjà abordé le descripteur de tâche, voici plus en détail le sens de chaque champ :

- **prioinit** : c'est la priorité de la tâche comprise entre 1 et 15 inclus. 1 est la priorité la plus faible et 15 la plus forte. La priorité 0 est réservée au noyau lui-même.
- **stack address** : c'est la la pile logicielle de la tâche. Chaque tâche possède une pile logicielle sur laquelle seront stockées les variables locales au cours de l'exécution, mais aussi tout le contexte du PIC18 (registres, pile hardware) lorsque la tâche n'est pas en cours d'exécution.
- **start address** : c'est le nom de votre tâche, ou encore le nom de la fonction d'entrée de votre tâche. Le compilateur le traduit en une adresse, utilisée par le noyau pour accéder la première fois à votre tâche.
- **state init** : c'est l'état initial de votre tâche. Dans notre cas il vaut READY ce qui signifie que la tâche est opérationnelle au démarrage de l'application.
- **stack size** : c'est la taille de la pile logicielle attribuée. Cela sert au noyau afin de vérifier à tout moment qu'il n'y a aucun débordement mémoire d'une pile sur l'autre.

> Interruptions à 1ms

Même si votre application ne nécessite pas le besoin de gérer des interruptions, vous aurez certainement besoin de l'horloge logicielle interne de 1 ms. Pour cela le fichier "int.c" contient un certains nombres de définition, y compris la routine de gestion d'interruption.

Elle est en fait très simple et vous dispense de toute la gestion délicate des ITs sur PIC18 :

```
#pragma      code _INTERRUPT_VECTORL = 0x003000
#pragma interruptlow InterruptVectorL save=section(".tmpdata"), PROD
void InterruptVectorL(void)
{
    EnterISR();

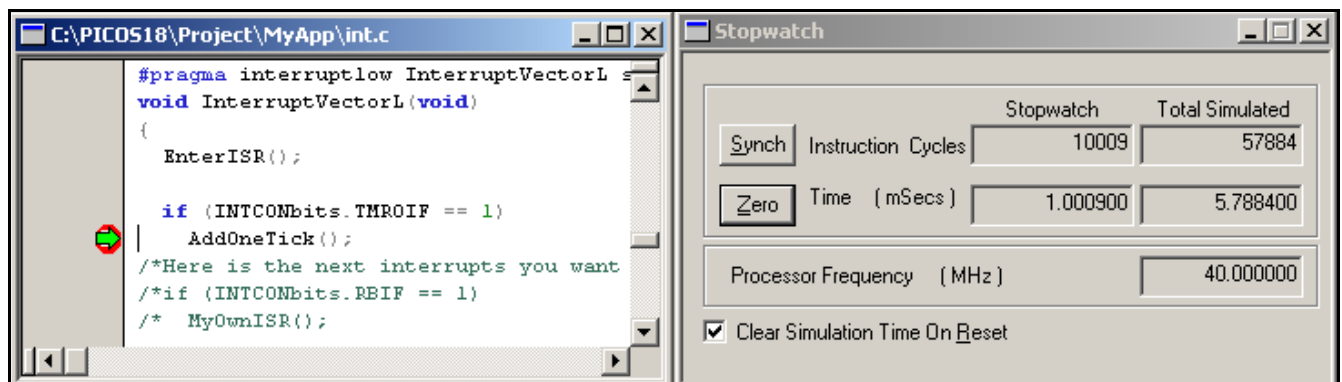
    if (INTCONbits.TMR0IF == 1)
        AddOneTick();
    /*Here is the next interrupts you want to manage */
    /*if (INTCONbits.RBIF == 1)                        */
    /*  MyOwnISR();                                    */

    LeaveISR();
}
#pragma      code
```

Comme vous pouvez le comprendre aisément, lorsque l'interruption du TIMER0 se lève (flag TMR0IF) on appelle la fonction AddOneTick qui ajouter 1 ms à toutes les alarmes de l'application. Lorsque certaines ont atteint leur seuil critique, elle déclenche des actions auprès des tâches associées en transitant par le noyau, seule entité du système à être autorisée à déclenchée des tâches.

Vous voyez donc que le noyau est toujours actif et qu'aucune tâche ne peut le stopper, même avec une boucle infinie du type "while(1)".

Pour le vérifier, paramétrez votre simulation à 40MHZ (Debugger/Settings..), placez un breakpoint en face de l'instruction AddOneTick(), et affichez la stopwatch de MPLAB (Debbuger/Stopwatch) :



> Démarrage de l'application

Enfin votre application possède aussi un fichier "main.c" nécessaire au compilateur. Selon la norme OSEK ce fichier main ne fait pas parti de l'OS mais bel et bien de la partie applicative, et c'est à lui que revient le rôle de lancer le noyau (car il est en effet possible de stopper et de lancer le noyau depuis le fichier main).

Dans le fichier main vous trouverez une fonction Init() qui va vous permettre de paramétrer votre fréquence de fonctionnement sans avoir besoin d'ouvrir la datasheet du PIC18 :

```
void Init(void)
{
    FSR0H = 0;
    FSR0L = 0;

    /* User setting : actual PIC frequency */
    Tmr0.lt = _40MHZ;

    /* Timer OFF - Enabled by Kernel */
    T0CON = 0x08;
    TMR0H = Tmr0.bt[1];
    TMR0L = Tmr0.bt[0];
}
```

La ligne en gras vous permet de régler la fréquence INTERNE du PIC18. En effet celui-ci possède une PLL par 4, que vous pouvez activer ou non. Une utilisation classique des PIC18 consiste à les équiper d'un quartz externe de 10 MHz et de 2 capacités de 15 pF, puis d'activer la PLL interne afin d'obtenir une fréquence interne de **40 MHz**.

Le pipeline interne du processeur étant de niveau 4, le pire cas de fonctionnement du PIC18 est de 10 MIPS, soit 10 millions d'instructions par secondes (voir la datasheet du composant). Cela en fait un microcontrôleur 8 bits faible cout puissant et bien armé en périphérique.

Si vous souhaitez utiliser une fréquence différente, consulter les fréquences disponibles dans le fichier "**define.h**".



5. La préemption

L'application mono-tâche que vous venez de réaliser n'a que très peu de raison d'être... En ajoutant à votre application une seconde tâche vous découvrirez les mécanismes de préemption d'une tâche sur l'autre.

> Réveil périodique d'une tâche

Il est encore difficile de se rendre compte de l'intérêt d'un noyau lorsqu'on utilise qu'une seule tâche pour... boucler à l'infini !

Nous allons donc commencer à rendre un peu plus intéressante notre tâche en lui faisant clignoter une LED à une certaine fréquence.

Modifier le code de la tâche comme suit :

```
TASK(TASK0)
{
    TRISBbits.TRISB4 = 0;
    LATBbits.LATB4   = 0;

    SetRelAlarm(ALARM_TSK0, 1000, 200);

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

        LATBbits.LATB4 = ~LATBbits.LATB4;
    }
}
```

Notre tâche est ici composée de 2 séquences : une qui précède le "while(1)" qui correspond à la phase d'initialisation de la tâche, et une autre qui se trouve dans le corps du while.

Les 2 premières instructions vous permettent de manipuler les ports d'entrées/sorties du PIC18. Le mot clef "TRISx" permet de mettre un port en entrée ou en sortie, alors que le mot clef "TRISxbits" qui est une structure permet de modifier l'état d'un bit de port. Ici nous avons donc mis le port RB4 en sortie (d'où le "0", un "1" signifiant "entrée") et sa valeur de sortie à "0" (mot clef LATxbits).

Ensuite la fonction SetRelAlarm est utilisée pour programmer l'alarme de la tâche. Dans le chapitre précédent, nous avons vu ce qu'était une alarme : un objet TIMER basé sur une horloge logique à 1ms.

Dans le fichier "tascdesc.c" nous avons vu que l'alarme dont l'ID vaut 0 (ALARM_TSK0) a été associée à la tâche TASK0 :

```
AlarmObject Alarm_list[] =
{
    /***** First task *****/
    {
        OFF,                /* State */
        0,                  /* AlarmValue */
        0,                  /* Cycle */
        &Counter_kernel,    /* ptrCounter */
        TASK0_ID,           /* TaskID2Activate */
        ALARM_EVENT,        /* EventToPost */
        0,                  /* CallBack */
    },
};
```

La fonction SetRelAlarm possède 3 champs :

- **Alarm ID** : indice de l'alarme dans le tableau Alarm_list
- **TIMER 1**: nombre de millisecondes avant le premier événement posté
- **TIMER 2**: nombre de millisecondes entre 2 événements successifs

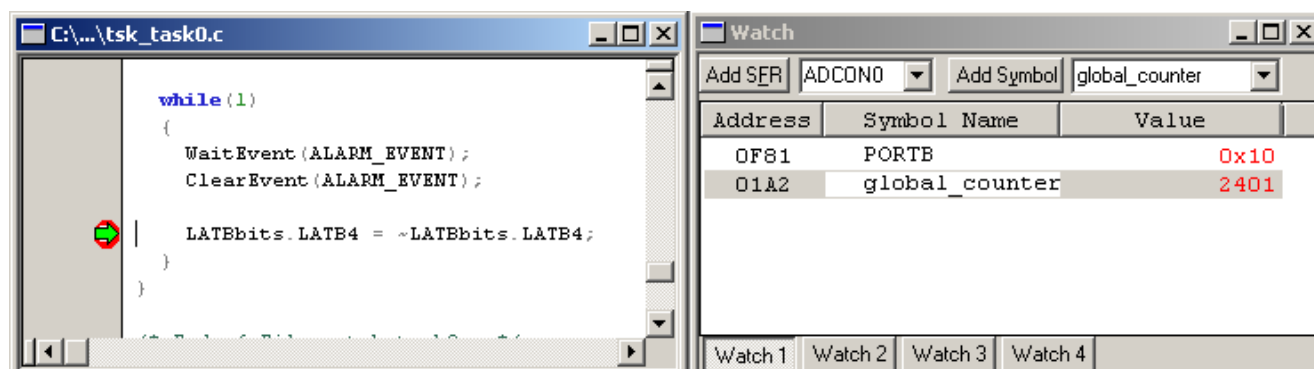
Dans notre cas cela signifie que l'alarme 0 va attendre 1000ms avant de poster l'événement ALARM_EVENT, puis le postera périodiquement toutes les 200 ms.

Une fois l'alarme programmée, le code de la tâche peut continuer à s'exécuter. Dans le "while(1)", la première fonction appelée est la fonction WaitEvent, qui permet de mettre en sommeil la tâche jusqu'à l'arrivée de l'événement attendu, ici **ALARM_EVENT**.

Vous pouvez mettre votre tâche en attente sur n'importe quel événement, l'important étant de le définir comme une puissance de 2 dans le fichier "**define.h**". Les valeurs autorisées sont donc : 0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02 et 0x01.

> Simulation de la tâche 0

Pour vérifier que nous avons bel et bien réalisé une tâche de clignotement à une période de 200ms, il est possible de simuler l'application en plaçant un breakpoint en face de la ligne de pilotage du port RB4:



De plus vous pouvez espionner les périphériques du PIC18 ou bien les variables globales de PICos18, en particulier "global_counter" qui est un compteur décimal du nombre de millisecondes écoulées. Pour obtenir une affichage en décimal de la valeur cliquer à l'aide du bouton droit de la souris sur la variable de votre choix et choisissez un affichage "Dec".

En faisant des RUN successif (F9), vous verrez le **PORTB passer de 0x00 à 0x10**, et la valeur de global_counter s'incrémenter de 200 en 200, en commençant par la valeur 1001 (le noyau met 1ms à booter, d'où la valeur de 1001 au lieu de 1000 attendue au premier abord).

Cela signifie que le port RB4 passe à 1 au bout de 1 seconde puis oscille périodiquement toutes les 200ms. Modifiez à présent les valeurs des paramètres de temps de la fonction SetRelAlarm pour connaître leur effet. Mettez par exemple 0 dans le dernier champ, vous verrez alors que le port RB4 reste à 1 et qu'on ne rentre plus jamais dans la tâche.

Consultez le **PDF relatif à l'API du noyau PICos18** pour tout connaître des fonctions utilisées par la tâche.

> Création d'une seconde tâche

Maintenant que nous avons réalisé une tâche périodique, nous allons créer une seconde tâche et les synchroniser.

- 1) Sous MPLAB®, enregistrez le fichier "tsk_task0.c" sous le nom "**tsk_task1.c**"
- 2) Modifiez les références à la nouvelle tâche dans le fichier taskdesc.c, comme suit :

```
#define DEFAULT_STACK_SIZE      128
DeclareTask(TASK0);
DeclareTask(TASK1);

volatile unsigned char stack0[DEFAULT_STACK_SIZE];
volatile unsigned char stack1[DEFAULT_STACK_SIZE];

/*****
 * ----- TASK DESCRIPTOR SECTION -----
 *****/
#pragma          romdata          DESC_ROM
const rom unsigned int descromarea;
/*****
 * ----- task 0 -----
 *****/
rom_desc_tsk rom_desc_task0 = {
    TASK0_PRIO,          /* prioinit from 0 to 15      */
    stack0,              /* stack address (16 bits)    */
    TASK0,               /* start address (16 bits)    */
    READY,               /* state at init phase       */
    TASK0_ID,            /* id_tsk from 1 to 15       */
    sizeof(stack0)       /* stack size (16 bits)      */
};
/*****
 * ----- task 1 -----
 *****/
rom_desc_tsk rom_desc_task1 = {
    TASK1_PRIO,          /* prioinit from 0 to 15      */
    stack1,              /* stack address (16 bits)    */
    TASK1,               /* start address (16 bits)    */
    READY,               /* state at init phase       */
    TASK1_ID,            /* id_tsk from 1 to 15       */
    sizeof(stack1)       /* stack size (16 bits)      */
};
```

3) Puis changez le corps de la fonction de la tâche 1 par le code suivant :

```
unsigned char hour, min, sec;

/*****
 * ----- TASK1 -----
 *
 * Second task of the tutorial.
 *
 *****/
TASK(TASK1)
{
    hour = min = sec = 0;

    while(1)
    {
        WaitEvent(TASK1_EVENT);
        ClearEvent(TASK1_EVENT);

        sec++;
        if (sec == 60)
        {
            sec = 0;
            min++;
            if (min == 60)
            {
                min = 0;
                hour++;
            }
        }
    }
}
```

Comme vous l'avez très certainement compris cette tâche permet de créer un chronomètre en heure/minute/seconde : à chaque fois que la tâche est réveillée, la variable des secondes est incrémentée de 1, et les variables min et hour sont mises à jour en conséquence.

4) Il paraît donc évident que la tâche doit être appelée toutes les 1000ms. Modifiez la tâche 0 pour quelle poste un événement toutes les secondes à notre nouvelle tâche :

```
SetRelAlarm(ALARM_TSK0, 1000, 1000);

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

    LATBbits.LATB4 = ~LATBbits.LATB4;
    SetEvent(TASK1_ID, TASK1_EVENT);
}
```

5) Il faut renseigner le compilateur sur les nouveau symboles que nous avons ajoutés à l'aide du fichier "define.h".

```

/*****
 * ----- Events -----
 *****/
#define ALARM_EVENT      0x80
#define TASK1_EVENT      0x10

/*****
 * ----- Task ID -----
 *****/
#define TASK0_ID          1
#define TASK1_ID          2

#define TASK0_PRIO        7
#define TASK1_PRIO        6

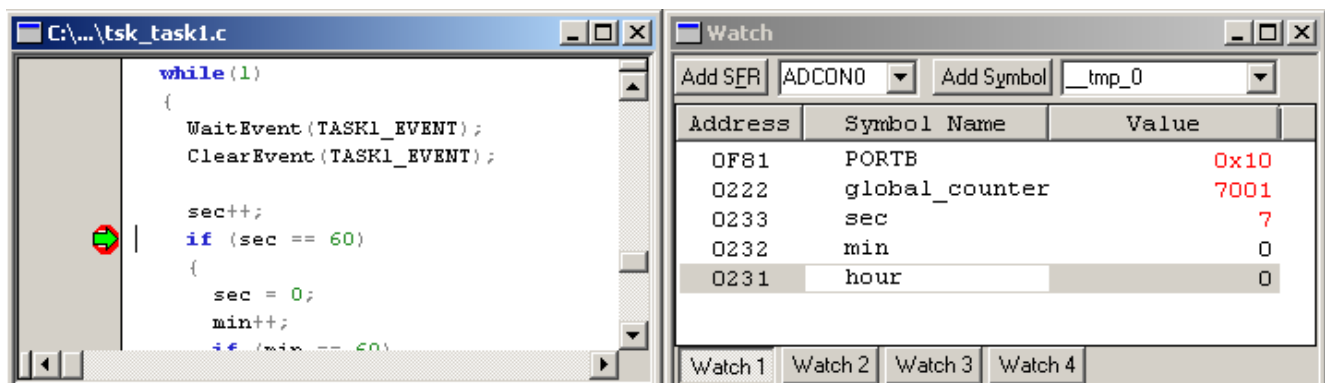
```

6) Enfin il faut ajouter la nouvelle tâche au projet ("Project/Add files to Project...").

Vous pouvez désormais recompiler le projet (F10).

> Simulation de la tâche 1

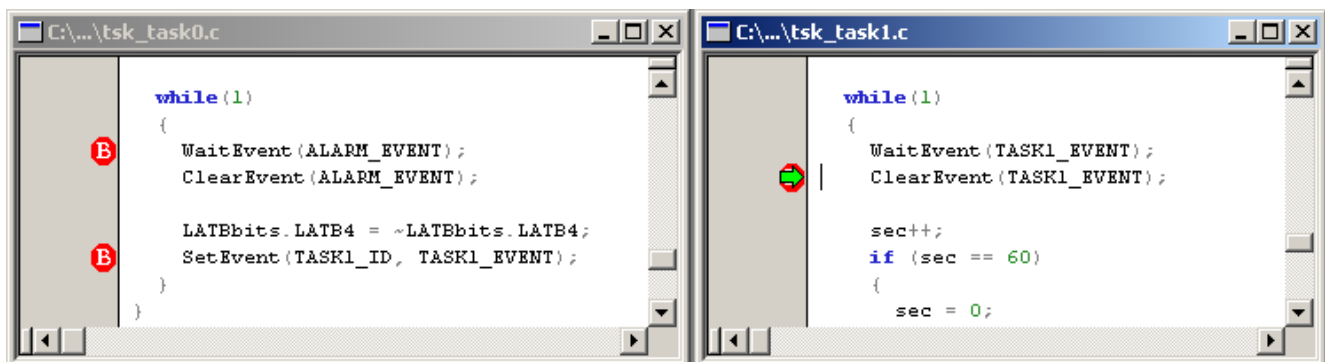
Afin de vérifier que les variables hour, min et sec sont bien mise à jour sur une base de 1 seconde, lancez la simulation avec les breakpoints suivants :



> La préemption

Votre application est désormais composée de 2 tâches, elle est donc à proprement parler multi-tâches. Pourtant cela ne veut pas dire que les 2 tâches fonctionnent en même temps, c'est parfaitement impossible pour le processeur du PIC18 qui ne peut exécuter qu'un seul code à la fois. Tout au plus les tâches fonctionnent en parallèle, c'est-à-dire l'une après l'autre en fonction d'un certain nombre de règles définies par le noyau, notamment les priorités.

Pour bien comprendre les notions de préemption, de multi-tâches et de temps-réel, placez des breakpoints comme ci-dessous et lancer la simulation :



Le pointeur parcourt les breakpoints dans l'ordre suivant :

- **WaitEvent dans la tâche 0**
- **SetEvent dans la tâche 0**
- **WaitEvent dans la tâche 0**
- **ClearEvent dans la tâche 1**

En fait cela s'explique par le fait que la tâche 0 est plus prioritaire que la tâche 1, donc lorsqu'elle aura posté un événement à la tâche 1, elle continuera à s'exécuter jusqu'à ce qu'elle soit mise en sommeil par le WaitEvent. Du coup la tâche 1 a le champ libre pour pouvoir s'exécuter et le pointeur s'arrête sur le breakpoint du ClearEvent.

Maintenant modifiez la priorité de la tâche 1 dans le fichier "define.h" pour qu'elle soit la plus prioritaire :

```

/*****
 * ----- Task ID -----
 *****/
#define TASK0_ID          1
#define TASK1_ID          2

#define TASK0_PRIO        7
#define TASK1_PRIO        10

```

Recompilez et relancez la simulation, désormais l'ordre de passage des breakpoints sera le suivant :

- **WaitEvent dans la tâche 0**
- **SetEvent dans la tâche 0**
- **ClearEvent dans la tâche 1**
- **WaitEvent dans la tâche 0**

Comme la tâche 1 est désormais plus prioritaire, lorsqu'un événement lui est assigné, la tâche en cours est immédiatement suspendue pour permettre à la tâche de plus forte priorité de s'exécuter : on dit qu'il y a eu **préemption** (qui signifie prendre la main).

La préemption est une des caractéristiques fondamentales des systèmes multi-tâches temps réels. En effet dans un système non préemptif, la tâche 0 aurait continué de s'exécuter jusqu'à se mettre en sommeil pour permettre à la tâche 1 de fonctionner. On dit alors qu'un tel système est coopératif, c'est-à-dire que c'est à la tâche en cours de fonctionnement de décider quand rendre la main au reste de l'application. Le système SALVO qui existe pour PIC18 est un bon exemple de système **coopératif**.

Certes dans notre cas, cela n'a pas trop de conséquence, l'application semble fonctionner de façon identique. Toutefois lorsqu'il s'agit de gérer des protocoles de communication complexes ou bien lorsque l'application comporte de nombreuses tâches, un système préemptif garantit le bon fonctionnement de l'application, quelque soit la taille du code.

Pourtant la préemption n'est un critère suffisant pour qualifier un système temps réel. Il est nécessaire d'y ajouter le **déterminisme**, c'est-à-dire la capacité à garantir que le temps nécessaire pour passer d'une tâche à l'autre est une constante. Dans PICos18 elle vaut 50 µs, ce qui veut dire qu'une fois que la tâche 0 poste l'événement à la tâche 1, le passage de la tâche 0 à la tâche 1 prend 50 µs.

PICos18 est donc un système préemptif, multi-tâches et temps réel qui garantit qu'une tâche de plus forte priorité s'exécutera immédiatement si un événement lui est associé, et ceci dans un délai de 50 µs quelque soit l'application. Vous comprendrez donc qu'une boucle infinie "while(1);" dans la tâche la plus prioritaire de votre application bloquera toute votre application. Essayez en simulation !



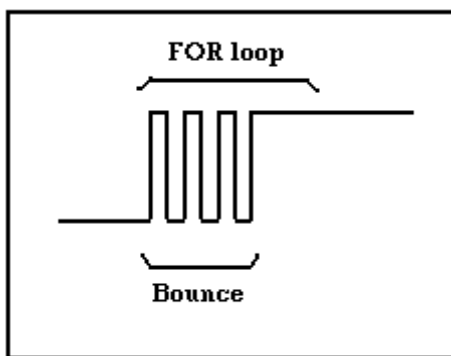
6. Le multi-tâches

Ce chapitre présente les mécanismes de synchronisation entre tâches de PICos18. Une troisième tâche viendra compléter l'application et postera un événement à une autre tâche afin de l'activer. Vous apprendrez à utiliser les événements et à partager les ressources.

> Créations d'une tâche de fond

Dans les chapitres précédents nous avons créé un système composé de 2 tâches en charge de la gestion d'une horloge au travers des variables hour, min et sec.

Le multi-tâches de PICos18 ne permet pas seulement de découper un projet en plusieurs tâches distinctes afin d'en réduire la complexité mais aussi de coder des sous-ensembles complètement indépendant. Nous allons par exemple créer 2 nouvelles tâches qui auront pour fonction **d'inspecter en permanence les ports RB0 et RB1**, correspondant à 2 boutons poussoirs.



Un bouton poussoir n'est pas un signal numérique qui passe de l'état 0 à l'état 1 et y reste jusqu'au prochain changement d'état.

Lorsqu'on appuie sur un bouton poussoir (élément mécanique) il se produit des micro-rebonds, de quelques millisecondes, avant de stabiliser le signal à 1. Il importe donc de réaliser un code qui détecte l'état haut stable pour pouvoir vraiment déterminer si

Nous allons donc commencer par créer une première tâche qui va servir à inspecter le port RB0 sur lequel doit se trouver le bouton BTN0.

1) Sous MPLAB®, enregistrez le fichier "tsk_task0.c" sous le nom "**tsk_task2.c**"

2) Modifiez les références à la nouvelle tâche dans le fichier taskdesc.c, comme suit :

```
#define DEFAULT_STACK_SIZE      128
DeclareTask(TASK0);
DeclareTask(TASK1);
DeclareTask(TASK2);

volatile unsigned char stack0[DEFAULT_STACK_SIZE];
volatile unsigned char stack1[DEFAULT_STACK_SIZE];
volatile unsigned char stack2[DEFAULT_STACK_SIZE];

/*****
 * ----- TASK DESCRIPTOR SECTION -----
 *****/
#pragma          romdata          DESC_ROM
. . .
```

```

/*****
* ----- task 1 -----
*****/
rom_desc_tsk rom_desc_task2 = {
    TASK2_PRIO,          /* prioinit from 0 to 15      */
    stack2,              /* stack address (16 bits)    */
    TASK2,               /* start address (16 bits)    */
    READY,               /* state at init phase        */
    TASK2_ID,            /* id_tsk from 1 to 15        */
    sizeof(stack2)       /* stack size (16 bits)       */
};

```

3) Puis changez le corps de la fonction de la tâche 2 par le code suivant :

```

extern unsigned char hour;

/*****
* ----- TASK2 -----
*
* Third task of the tutorial.
*
*****/
TASK(TASK2)
{
    unsigned int i;

    while(1)
    {
        while(PORTBbits.RB0 == 0);
        for (i = 0; i < 10000; i++)
            Nop();
        if (PORTBbits.RB0 == 1)
            hour++;
    }
}

```

La tâche possède une boucle infinie sans aucune attente à l'aide d'une fonction WaitEvent. Une telle tâche risque donc de bloquer le mécanisme de multi-tâches pour les tâches moins prioritaire... Elle doit donc posséder la priorité la plus faible : on dit que c'est **la tâche de fond**.

Elle commence par inspecter le port RB0 et reste dans une boucle infinie tant que la bouton BTN0 n'est pas enfoncé. Dès que le bouton est enfoncé la tâche attend à l'aide d'une boucle FOR de façon à éviter la phase de rebond. Ensuite si il s'avère que le port RB0 vaut bien 1 alors on incrémente de 1 la variable "hour". Ceci permet par exemple de venir paramétrer l'horloge.

4) Il faut renseigner le compilateur sur les nouveau symboles que nous avons ajoutés à l'aide du fichier "define.h".

```

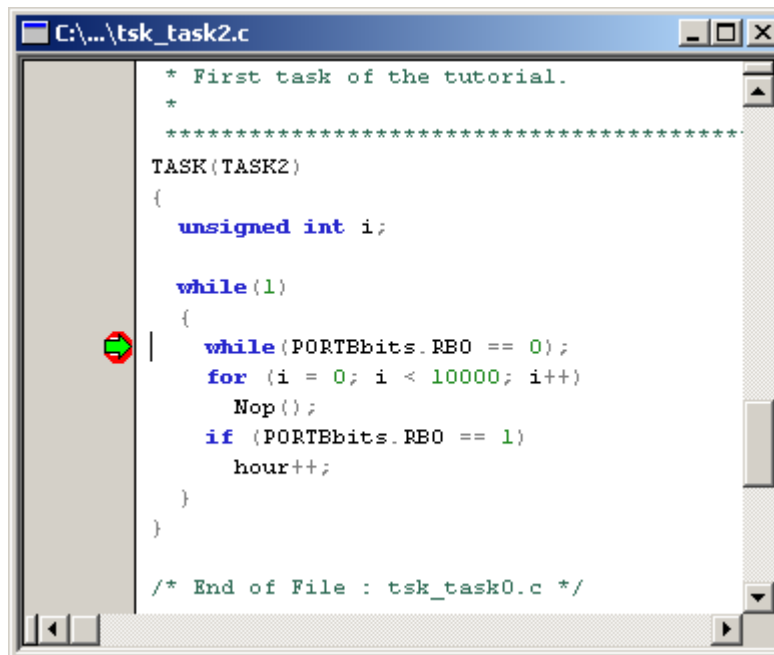
/*****
* ----- Task ID -----
*****/
#define TASK0_ID          1
#define TASK1_ID          2
#define TASK2_ID          3

#define TASK0_PRIO        7
#define TASK1_PRIO        10
#define TASK2_PRIO        1

```


5) Enfin il faut ajouter la nouvelle tâche au projet ("Project/Add files to Project...").

Vous pouvez désormais recompiler le projet (**F10**).



```

C:\...\tsk_task2.c

* First task of the tutorial.
*
*****
TASK(TASK2)
{
    unsigned int i;

    while(1)
    {
        while(PORTBbits.RB0 == 0);
        for (i = 0; i < 10000; i++)
            Nop();
        if (PORTBbits.RB0 == 1)
            hour++;
    }
}

/* End of File : tsk_task0.c */

```

> Round robin

Nous allons maintenant créer une seconde tâche de fond. En effet PICos18 permet de créer plusieurs tâches possédant le même niveau de priorité. Dans ce cas on peut bien se demander laquelle des 2 peut bien prendre la main sur la seconde pour s'exécuter. En fait dans ce cas précis, le scheduler de PICos18 utilise un algorithme spécifique dit 'Round Robin'.

Avec un tel algorithme, les tâches de fond vont se **partager le processeur du PIC18 durant une tranche de temps de 1ms** dans notre cas. Comme ces tâches sont de faibles priorités, elles s'exécuteront uniquement si aucune autre tâche n'a besoin de fonctionner et pendant un temps maximum de 1 ms. Ensuite ce sera à une autre tâche de même priorité de s'exécuter.

Créez donc une tâche 3 qui possède le même code (à l'exception prêt qu'il s'agit d'inspecter le port RB1) et possédant la même priorité :

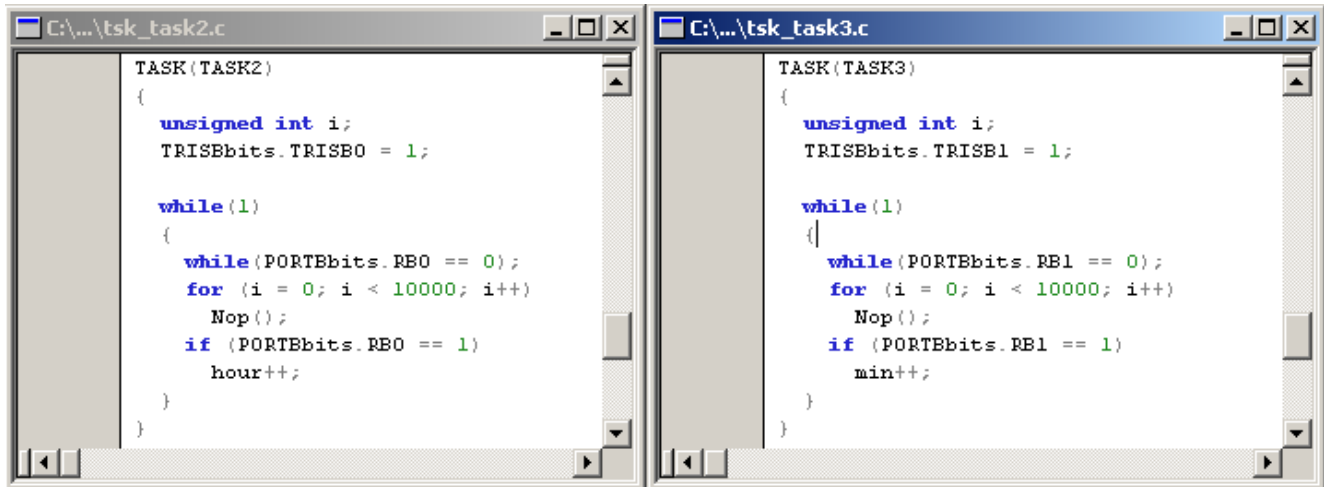
```

/*****
* ----- Task ID -----
*****/
#define TASK0_ID      1
#define TASK1_ID      2
#define TASK2_ID      3
#define TASK3_ID      4

#define TASK0_PRIO    7
#define TASK1_PRIO    10
#define TASK2_PRIO    1
#define TASK3_PRIO    1

```

N'oubliez pas d'ajouter le nouveau fichier au projet, puis recompilez (F10).

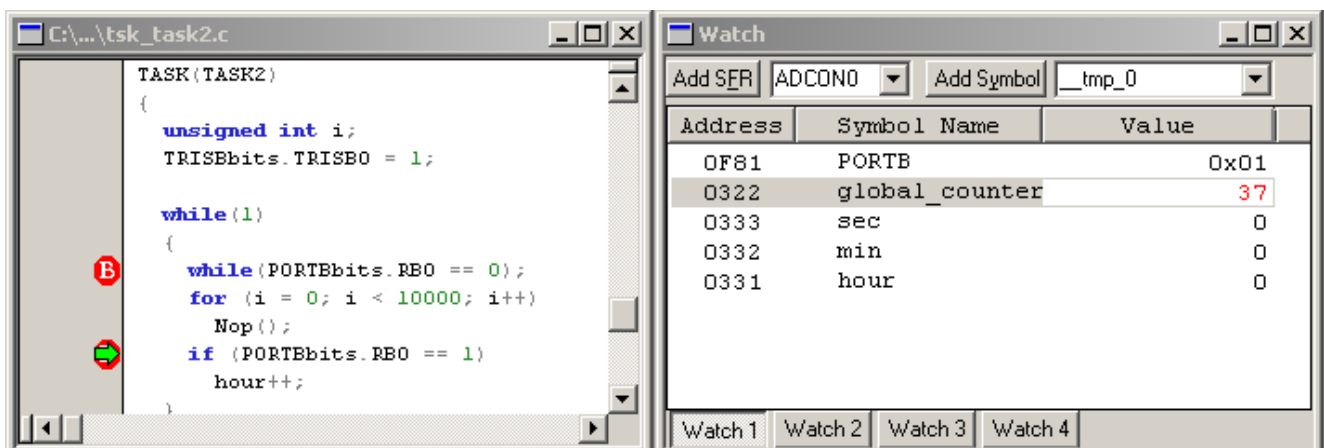


Placez des breakpoints devant chaque instruction "**while(PORTBbits.RBx == 0);**", puis lancez la simulation. Vous verrez tout d'abord le pointeur s'arrêter dans la tâche 2. Dévalidez alors le breakpoint et poursuivez la simulation, le pointeur s'arrêtera alors dans la tâche 3 exactement 1ms plus tard...

Si maintenant vous mettez des breakpoints dans les tâches 0 et 1, vous verrez qu'au bout de 1 seconde, le pointeur s'arrêtera dans une de ces 2 tâches, il y a eu préemption du fait des priorités de ces 2 tâches.

> Simulation

Pour simuler l'appuie sur le bouton BTN0, vous pouvez utiliser la fenêtre de Watch :



Lancez la simulation jusqu'au premier breakpoint de la tâche 2. Cliquez dans la fenêtre de Watch pour passer la valeur du PORTB à 0x01. Puis lancez la simulation en vérifiant que le breakpoint de la tâche 3 est toujours actif.

Vous verrez alors le pointeur s'arrêter d'abord dans la tâche 3 avant d'atteindre le second breakpoint de la tâche 2. En effet au delà de 1ms, le scheduler a jugé bon de donner la main à l'autre tâche...



7. Les interruptions

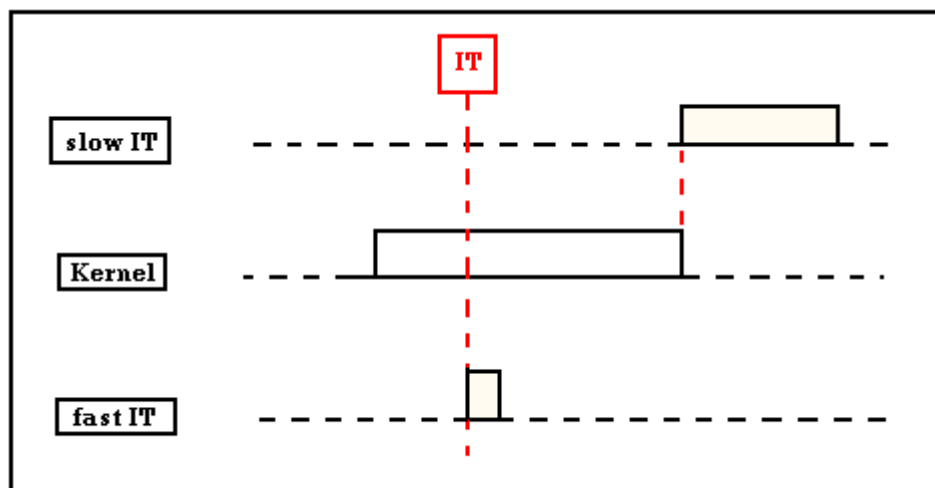
Afin de pouvoir tirer parti du microcontrôleur PIC18, il est important de pouvoir utiliser les périphériques. De ce point de vue, les interruptions sont essentielles. Vous verrez comment les interruptions du PIC18 sont mises en œuvre sous PICos18 et comment écrire simplement vos propres routines d'interruption.

> Présentation du fichier int.c

Nous allons à présent brancher nos 2 boutons sur les broches **d'interruptions externes du PIC18**, ce qui nous permettra de ne traiter les boutons que lorsqu'ils sont utilisés.

Tout d'abord nous allons voir de quoi est constitué le fichier "int.c", fichier qui prend en charge la gestion de la totalité des interruptions. Il faut savoir que le PIC18 gère 2 types d'interruptions : les interruptions de basses priorités (IT_vector_low) et celles de hautes priorités (IT_vector_high). A quoi cela peut-il bien servir....?!

En fait il faut considérer que le noyau PICos18 effectue parfois des opérations délicates sur les tâches et qu'il n'est pas souhaitable qu'il soit interrompu par un IT pendant son traitement (par exemple pendant la préemption d'une nouvelle tâche sur la tâche courant). Du coup pour faire simple on a choisi de rendre le noyau non-interruptible. Or on a vu que le déterminisme de PICos18 était de 50µs (ce qu'on appelle le temps de latence), donc il existe une zone d'ombre, un black-out de 50 µs pendant laquelle une IT ne sera pas détectée.



Pour éviter cela on utilise les interruptions rapides ("fast interrupts") du PIC18, qui utilise un mécanisme hardware pour la sauvegarde des registres élémentaires (comme WREG, STATUS, ...). Hélas pour utiliser les interruptions rapides du PIC18, il faut écrire un code C extrêmement léger, presque trivial. Si l'on a besoin d'un code plus complexe, qui appelle par exemple d'autres fonctions C, il faut impérativement utiliser les interruptions lentes... avec toutefois le risque d'être mis en attente pendant 50µs !

```

/*****
 * Function you want to call when an IT occurs.
 *****/
extern void AddOneTick(void);
/*extern void MyOwnISR(void); */
void InterruptVectorL(void);
void InterruptVectorH(void);

/*****
 * General interrupt vector. Do not modify.
 *****/
#pragma code IT_vector_low=0x18
void Interrupt_low_vec(void)
{
    _asm goto InterruptVectorL _endasm
}
#pragma code

#pragma code IT_vector_high=0x08
void Interrupt_high_vec(void)
{
    _asm goto InterruptVectorH _endasm
}
#pragma code

/*****
 * General ISR router. Complete the function core with the if or switch
 * case you need to jump to the function dedicated to the occurring IT.
 *****/
#pragma code _INTERRUPT_VECTORL = 0x003000
#pragma interruptlow InterruptVectorL save=section(".tmpdata"), PROD
void InterruptVectorL(void)
{
    EnterISR();

    if (INTCONbits.TMR0IF == 1)
        AddOneTick();
    /*Here is the next interrupts you want to manage */
    /*if (INTCONbits.RBIF == 1) */
    /* MyOwnISR(); */

    LeaveISR();
}
#pragma code

/* BE CARREFULL : ONLY BSR, WREG AND STATUS REGISTERS ARE SAVED */
/* DO NOT CALL ANY FUNCTION AND USE PLEASE VERY SIMPLE CODE LIKE */
/* VARIABLE OR FLAG SETTINGS. CHECK THE ASM CODE PRODUCED BY C18 */
/* IN THE LST FILE. */
#pragma code _INTERRUPT_VECTORH = 0x003300
#pragma interrupt InterruptVectorH
void InterruptVectorH(void)
{
    if (INTCONbits.INT0IF == 1)
        INTCONbits.INT0IF = 0;
}
#pragma code

```

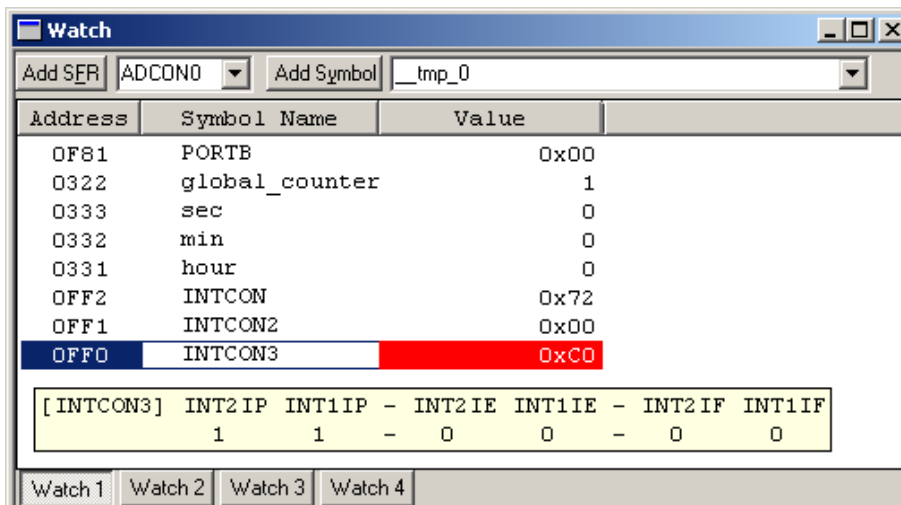
Au premier abord le fichier "int.c" peut paraître bien compliqué, mais il est en fait fort simple. Les 2 seules zones qui peuvent être soumises à modifications sont les fonctions InterruptVectorL() et InterruptVectorH(). Comme dans 99% des cas l'interruption lente sera suffisante pour vos applications, vous n'aurez qu'à intervenir dans la fonction InterruptVectorL().

> Le mode "fast interrupt"

La fonction du mode "fast interrupt" se présente comme suit :

```
/* BE CAREFULL : ONLY BSR, WREG AND STATUS REGISTERS ARE SAVED */
/* DO NOT CALL ANY FUNCTION AND USE PLEASE VERY SIMPLE CODE LIKE */
/* VARIABLE OR FLAG SETTINGS. CHECK THE ASM CODE PRODUCED BY C18 */
/* IN THE LST FILE. */
#pragma code _INTERRUPT_VECTORH = 0x003300
#pragma interrupt InterruptVectorH
void InterruptVectorH(void)
{
    if (INTCONbits.INT0IF == 1)
        INTCONbits.INT0IF = 0;
}
#pragma code
```

Les interruptions externes **INT0 (broche RB0)**, **INT1 (broche RB1)** et **INT2 (broche RB2)** sont par défaut en mode fast interrupt, c'est-à-dire que lorsque l'une de ces IT se produit, elle interrompt le code et exécute la fonction InterruptVectorH().



L'interruption INT0 n'est pas paramétrable, c'est-à-dire qu'elle appelle toujours la fonction InterruptVectorH().

Par contre INT1 et INT2 peuvent être utilisées pour appeler la fonction InterruptVectorL().

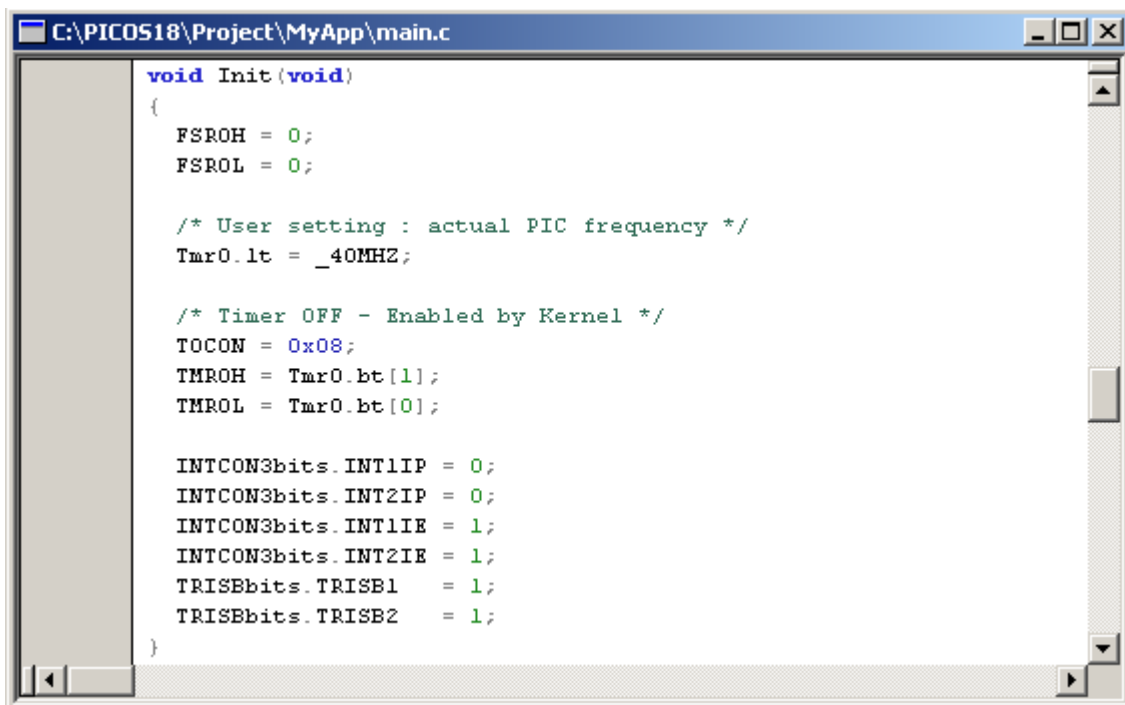
Pour cela il faut modifier le registre **INTCON3** qui permet ce paramétrage au travers des bits INT2IP

Si nous souhaitons à présent brancher nos 2 boutons sur INT1 et INT2, nous allons devoir traiter le mécanisme d'anti-rebond au sein de la fonction InterruptVectorH(). Or nous avons vu que nous avons pas droit à un code trop compliqué, comme à l'utilisation de pointeur, aux appels de fonctions, aux boucles FOR et WHILE,... Bref, tout au plus nous avons droit d'incrémenter ou décrémenter des variables globales...!

La raison en est simple : il ne faut pas modifier d'autres registres du PIC18 que les registres WREG, STATUS et BSR. Si vous hésitez entre "low interrupt" et "fast interrupt", inspecter le code généré par le compilateur pour vérifier qu'il ne modifie pas d'autres registres. Comme nous n'avons pas besoin de traiter l'appuie de nos boutons avec une précision meilleure que 50µs, nous allons donc utiliser le mode "slow interrupt".

> Le mode "slow interrupt"

Dans la plupart des traitements d'interruption, le mode "slow interrupt" est largement suffisant. Les 3 seules interruptions qui sont par défaut en mode "fast interrupt" sont INT0, INT1 et INT2. Comme INT0 ne peut être dévalidé, nous allons utiliser INT1 et INT2 en les reprogrammant en mode "slow interrupt" :



```

C:\PICOS18\Project\MyApp\main.c

void Init(void)
{
    FSROH = 0;
    FSROL = 0;

    /* User setting : actual PIC frequency */
    Tmr0.lt = _40MHZ;

    /* Timer OFF - Enabled by Kernel */
    TOCON = 0x08;
    TMR0H = Tmr0.bt[1];
    TMR0L = Tmr0.bt[0];

    INTCON3bits.INT1IP = 0;
    INTCON3bits.INT2IP = 0;
    INTCON3bits.INT1IE = 1;
    INTCON3bits.INT2IE = 1;
    TRISBbits.TRISB1 = 1;
    TRISBbits.TRISB2 = 1;
}
    
```

Pour utiliser INT1 et INT2 en mode "slow interrupt", modifiez le registre INTCON3 comme indiqué ci-dessus dans la phase d'init du main.

Puis complétez le fichier int.c pour intercepter les interruptions INT1 et INT2 :

```

/*****
 * Function you want to call when an IT occurs.
 *****/
extern void AddOneTick(void);
extern void MyOwnISR(void);
void InterruptVectorL(void);
void InterruptVectorH(void);

...

/*****
 * General ISR router. Complete the function core with the if or switch
 * case you need to jump to the function dedicated to the occuring IT.
 *****/
#pragma code _INTERRUPT_VECTORL = 0x003000
#pragma interruptlow InterruptVectorL save=section(".tmpdata"), PROD
void InterruptVectorL(void)
{
    EnterISR();

    if (INTCONbits.TMR0IF == 1)
        AddOneTick();
}
    
```

```
/*Here is the next interrupts you want to manage */
if (INTCON3bits.INT1IF || INTCON3bits.INT2IF)
    MyOwnISR();

    LeaveISR();
}
#pragma      code
```

Il suffit à présent d'ajouter la fonction "MyOwnISR" dans n'importe quel fichier C, de préférence une des 2 tâches de fond qui sont associés aux boutons. Par exemple nous ajoutons la fonction dans le fichier "tsk_task2.c" :

```
void MyOwnISR(void)
{
    if (INTCON3bits.INT1IF)
    {
        INTCON3bits.INT1IF = 0;
        /* ... */
    }
    if (INTCON3bits.INT2IF)
    {
        INTCON3bits.INT2IF = 0;
        /* ... */
    }
}
```

Et n'oubliez pas de déclarer la fonction au début du fichier :

```
#include "define.h"

void MyOwnISR(void);

/*****
 * Definition dedicated to the local functions.
 *****/
#define ALARM_TSK0      0
```

Vous pouvez à présent recompiler le projet (F10).

> Association aux tâches de fond

Pour le moment la fonction "MyOwnISR" ne permet de faire passer l'information aux tâches de fond qu'un des boutons a été enfoncé. De plus les tâches de fond continuent à fonctionner en permanence au travers d'une boucle infinie. Il faudrait pouvoir mettre en sommeil les 2 tâches de fond et les réveiller sur interruptions. Pour cela nous allons modifier les 2 tâches de fond pour qu'elles soient des tâches BASICS. Modifier "tsk-task2.c" comme suit :

```
TASK(TASK2)
{
    unsigned int i;

    for (i = 0; i < 10000; i++)
        Nop();
    if (PORTBbits.RB1 == 1)
        hour++;

    TerminateTask();
}
```

Comme vous pouvez le constater la tâche 2 ne possède plus de "while(1)" et de plus elle se termine par la fonction "TerminateTask()".

C'est ce qui caractérise une tâche **BASIC** :

- pas d'utilisation de la fonction "WaitEvent()",
- pas de boucle "while(1)"
- et l'appel à la fonction "TerminateTask()" impérativement à la fin.

L'objectif est que la tâche soit à l'état **SUSPENDED** par défaut puis activée à l'aide de la fonction "ActivateTask()".

Modifiez donc le fichier "tascdesc.c" pour mettre les tâches 2 et 3 à SUSPENDED par défaut :

```

/*****
 * ----- task 2 -----
 *****/
rom_desc_tsk rom_desc_task2 = {
    TASK2_PRIO,          /* prioinit from 0 to 15      */
    stack2,              /* stack address (16 bits)    */
    TASK2,               /* start address (16 bits)    */
    SUSPENDED,           /* state at init phase       */
    TASK2_ID,            /* id_tsk from 1 to 15       */
    sizeof(stack2)       /* stack size (16 bits)      */
};
/*****
 * ----- task 3 -----
 *****/
rom_desc_tsk rom_desc_task3 = {
    TASK3_PRIO,          /* prioinit from 0 to 15      */
    stack3,              /* stack address (16 bits)    */
    TASK3,               /* start address (16 bits)    */
    SUSPENDED,           /* state at init phase       */
    TASK3_ID,            /* id_tsk from 1 to 15       */
    sizeof(stack2)       /* stack size (16 bits)      */
};

```

Modifiez le code de la tâche 3 pour qu'elle soit aussi une tâche BASIC :

```

TASK(TASK3)
{
    unsigned int i;

    for (i = 0; i < 10000; i++)
        Nop();
    if (PORTBbits.RB2 == 1)
        min++;

    TerminateTask();
}

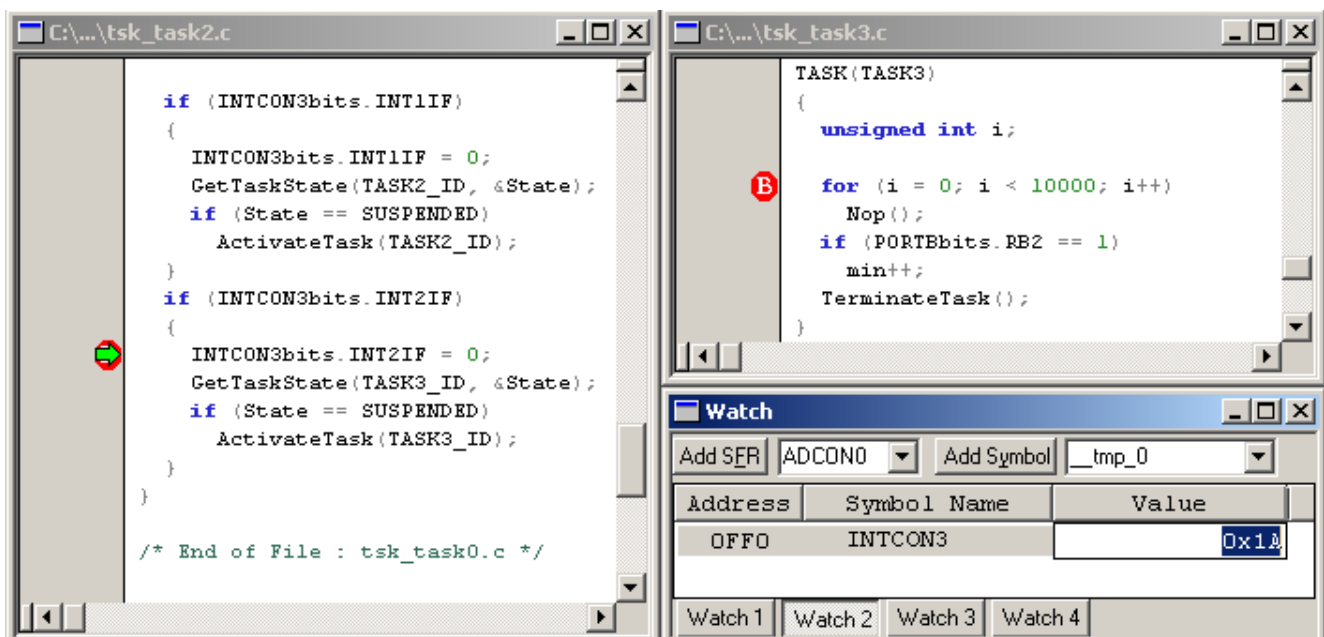
```


Et enfin modifiez la fonction "MyOwnISR" pour activer la tâche nécessaire au traitement du bouton :

```
void MyOwnISR(void)
{
    TaskStateType State;

    if (INTCON3bits.INT1IF)
    {
        INTCON3bits.INT1IF = 0;
        GetTaskState(TASK2_ID, &State);
        if (State == SUSPENDED)
            ActivateTask(TASK2_ID);
    }
    if (INTCON3bits.INT2IF)
    {
        INTCON3bits.INT2IF = 0;
        GetTaskState(TASK3_ID, &State);
        if (State == SUSPENDED)
            ActivateTask(TASK3_ID);
    }
}
```

Recompilez (F10) et relancez la simulation (F9)



Placez un breakpoint dans la tâche 3 et un dans la fonction "MyOwnISR". Lancez la simulation puis stoppez là à n'importe quel moment (F5). Modifiez INTCON3 pour activez une interruption (par exemple mettre 0x1A pour l'INT2) : vous verrez que le pointeur passe par "MyOwnISR" puis par la tâche 3.

Si vous poursuivez la simulation vous verrez que le pointeur ne passe plus par la tâche 3 avant la prochaine interruption : **la tâche a été suspendue**.

8. Utilisation des drivers

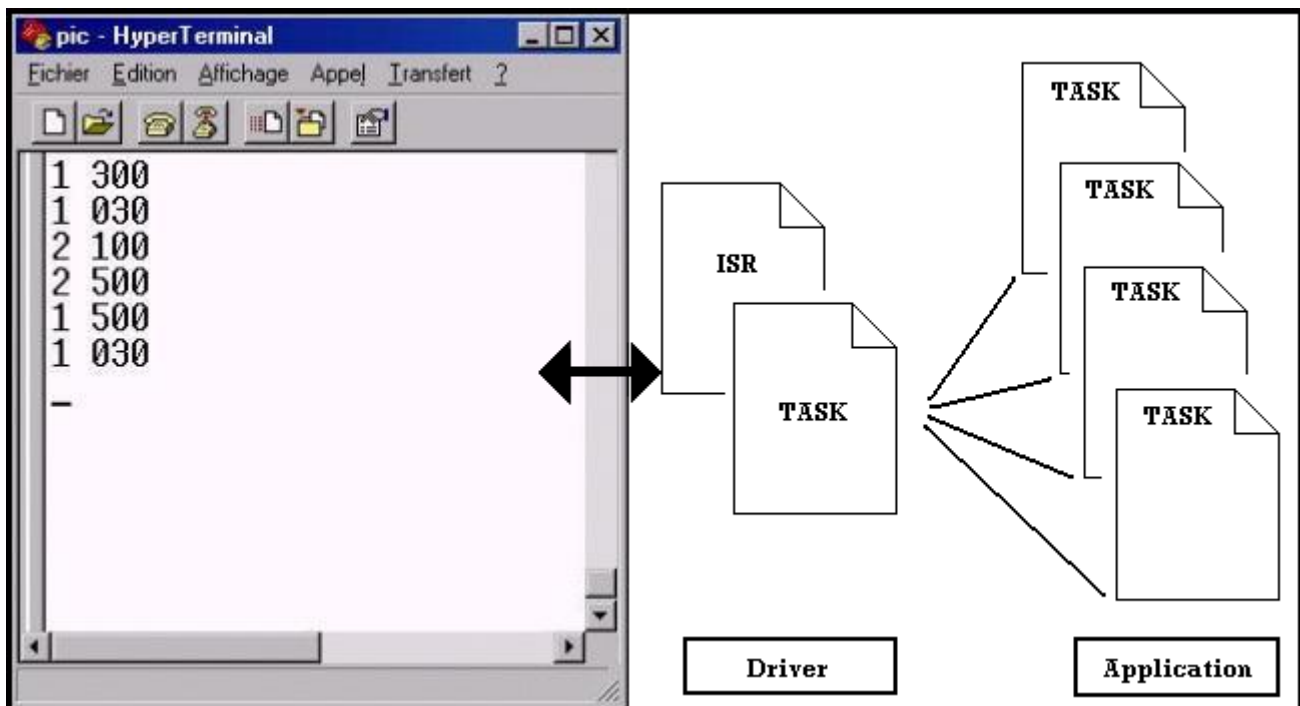


Vous souhaitez faire communiquer votre application au travers du port série ou bien du port CAN ? A présent que vous savez écrire des interruptions et des tâches, et que vous maîtrisez la gestion des évènements et des alarmes, il vous sera facile d'utiliser ou d'écrire un véritable driver pour PICos18.

> Principe d'un driver

Un driver est souvent comparé à une librairie, ce sont pourtant 2 choses bien distinctes. Une librairie est un ensemble de fonctions écrites dans un langage (comme le langage C par exemple) et destiné à simplifier la vie du développeur. Dans le cas d'une communication RS232, une librairie proposerait par exemple des fonctions d'ouverture et de fermeture du port série, de réception et de transmission de données...

Un driver est une couche logicielle destinée à la prise en charge complète d'un périphérique. Son but n'est donc pas d'offrir une série de fonctions propres à utiliser ce périphérique puisque c'est le driver lui seul qui peut manipuler le périphérique...



Un driver est en fait constitué d'une ISR (Interrupt Service Routine) et d'une tâche. Nous avons vu les ISR dans le chapitre précédent, c'est-à-dire les fonctions ajoutées à la fonction InterruptVectorL() ou InterruptVectorH().

L'ISR est en charge d'intercepter les interruptions (IT) et d'informer la tâche du driver que cette interruption vient de se produire. C'est donc à la tâche que revient d'effectuer les traitements nécessaire à la gestion de l'IT.

> Utilisation des briques logicielles

PICos18 est livré avec le noyau seul ainsi que ce tutorial. Toutefois sur la page DOWNLOAD du site web vous trouverez aussi des drivers ou des librairies disponibles gratuitement. Le but est de permettre un **développement modulaire** avec lequel il suffirait d'intégrer des **briques logicielles** pour réaliser une application.

Nous allons voir ici comment utiliser une brique logicielle. Plus précisément nous allons utiliser le driver RS232 de PICos18 pour pouvoir afficher notre horloge avec les variables "hour", "min", et "sec".

Télécharger le driver RS232 sur la page DOWNLOAD de PICos18 :

> Drivers			
	<i>Version</i>	<i>Document</i>	<i>Sources</i>
LCD HD4478	v1.01	<u>TXT</u>	
Librairie HD4478	en cours...		
Driver I2C (master)	v1.03	<u>TXT</u>	
Driver I2C (slave)	en cours...		
Driver CAN	v1.00	<u>TXT</u>	
DS1337 (RTC)	en cours...		
DS1624 (T°)	en cours...		
Driver RS232	v1.02	<u>TXT</u>	

Le package possède un fichier texte (lien TXT) qui indique comment inclure le driver dans votre application, comment paramétrer le driver (baudrate dans le cas d'un driver RS232) et aussi comment le mettre en oeuvre.

```
>
>                                     RS232 driver v1.02
>                                     for
>                                     PICos18 release 2.00 (beta 10 or upper)
>
>                                     PICos18 - Real-time kernel for PIC18 family
>
>
> www.picos18.com                                     www.pragmatec.net
>
...

```

Maintenant que vous savez insérer des tâches au projet, que vous savez écrire des ISR, paramétrer votre application dans le fichier "tascdesc.c", suivez pas à pas les indications du fichier texte pour insérer le driver dans votre application.

Vous n'avez pas besoin d'ajouter les références à la tâche "Example" comme l'alarme de la structure Alarm_list (paragraphe IV du fichier texte) ou encore EXAMPLE_ID du fichier define.h. Ne dépassez pas le paragraphe VI, celui-ci est abordé spécifiquement ci-après.

> Utilisation du driver

Afin d'utiliser le driver créez une nouvelle tâche "TASK4" en lieu et place de la tâche "Example" du fichier texte du driver RS232. N'oubliez pas d'ajouter l'alarme ALARM_TASK4 dans le fichier "tascdesc.c".

Voici à quoi doit ressembler la nouvelle tâche :

```
#include "define.h"
#include "drv_rs.h"
#include <stdio.h>
#include <string.h>

#define ALARM_TSK4      1

int Printf (const rom char *fmt, ...);
extern unsigned char hour;
extern unsigned char min;
extern unsigned char sec;

/*****
 * Definition dedicated to the local functions.
 *****/
RS_message_t RS_msg;
unsigned char  buffer[80];

/*****
 * ----- TASK4 -----
 *
 * Fifth task of the tutorial.
 *
 *****/
TASK(TASK4)
{
    SetRelAlarm(ALARM_TSK4, 1000, 1000);
}
```

```

Printf(" _____ \r\n");
Printf("> _____ <\r\n");
Printf("> PICos18 Tutorial <\r\n");
Printf("> _____ <\r\n");
Printf("> _____ <\r\n");
Printf("> PICos18 - Real-time kernel for PIC18 family <\r\n");
Printf("> _____ <\r\n");
Printf("> _____ <\r\n");
Printf("> www.picos18.com www.pragmatec.net <\r\n");
Printf("> _____ <\r\n");
Printf(" \r\n");
Printf(" \r\n");

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

    Printf("%02d : %02d : %02d\r", (int)hour, (int)min, (int)sec);
}

/*****
 * Function in charge of structure registration and buffer transmission.
 *
 * @param string IN const string send to the USART port
 * @return void
 *****/
int Printf (const rom char *fmt, ...)
{
    va_list ap;
    int n;
    RS_enqueueMsg(&RS_msg, buffer, 50);
    va_start (ap, fmt);
    n = vfprintf (_H_USER, fmt, ap);
    va_end (ap);
    SetEvent(RS_DRV_ID, RS_NEW_MSG);
    WaitEvent(RS_QUEUE_EMPTY); ClearEvent(RS_QUEUE_EMPTY);
    return n;
}

```

La première partie du code correspond aux inclusions des headers propres à l'utilisation du driver RS232 et aux fonctions "printf". Ensuite il faut déclarer ALARM_TASK4 et aussi déclarer comme "extern" les variables externes "hour", "min" et "sec" pour signifier au compilateur C18 que les définitions de ces variables ne se trouvent pas dans ce fichier.

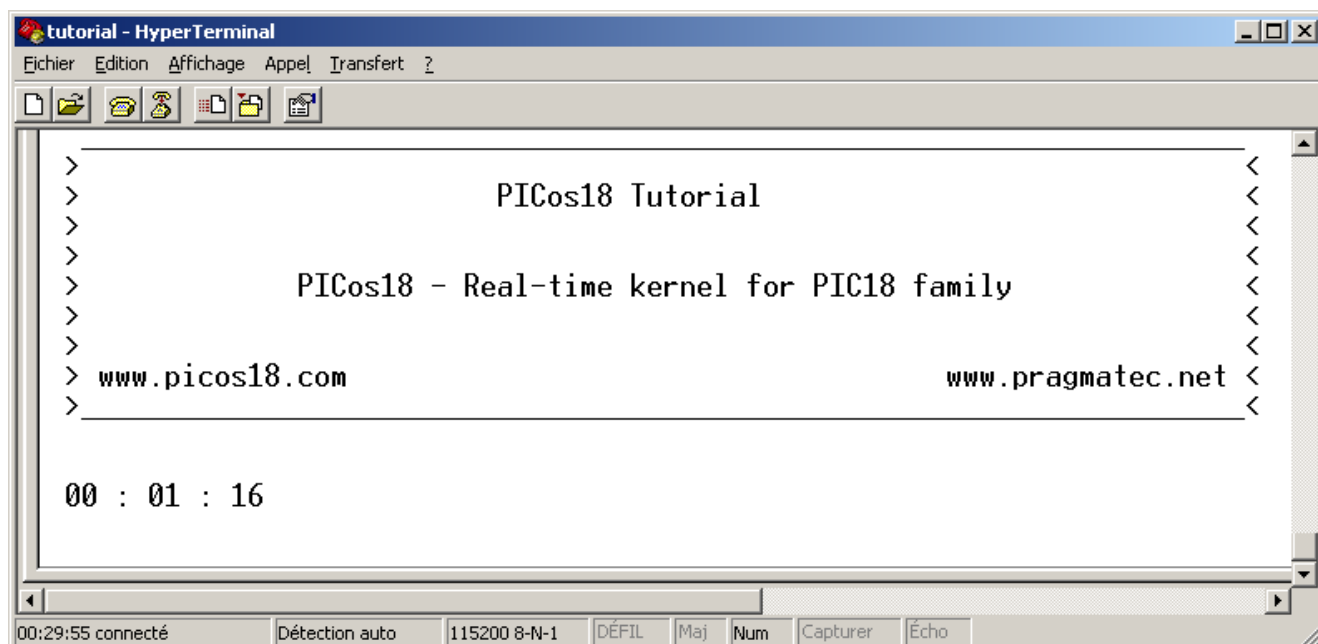
En RAM on déclare 2 types de variables :

- **RS_message_t RS_msg** : cette structure est obligatoire pour émettre un buffer avec le driver RS232
- **unsigned char buffer[80]** : il s'agit justement du buffer à transmettre.

Enfin vous découvrez le coeur de la tâche TASK4, qui arme une alarme qui réagit toutes les 1000ms soit toutes les secondes. La fonction "Printf" est appelée pour pouvoir poster des chaînes constantes, convertir des variables,... Tous les formats de variables sont gérés par C18 sauf les floatants...! La fonction "Printf" nécessite "\r\n" en fin de chaîne si vous souhaitez un retour chariot (chargement d'une nouvelle ligne), le "\r" servant au retour en début de ligne, et le "\n" au chargement d'une nouvelle ligne à proprement parler.

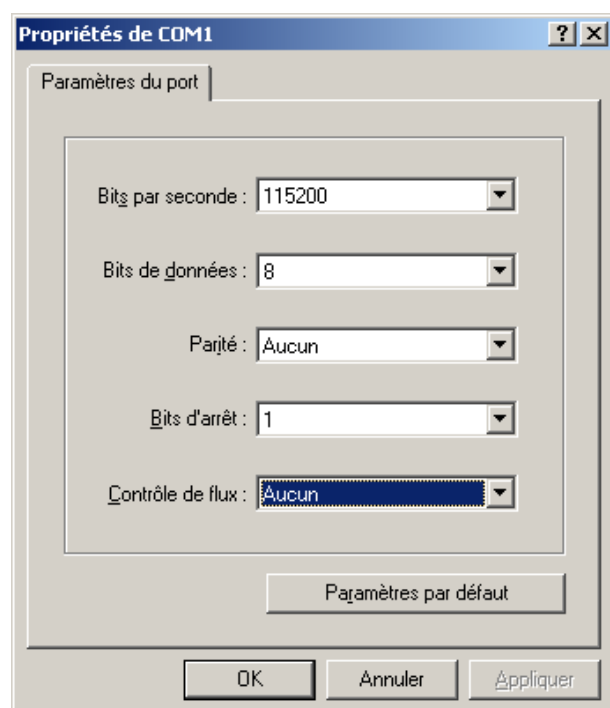
Pour afficher le banner nous utilisons donc "\r\n" et pour afficher l'heure en surimpression, nous utilisons juste "\r".

Au final vous devez obtenir l'affichage suivant sous HyperTerminal :



> Paramétrage d'HyperTerminal

Attention au paramétrage de la COM série sous HyperTerminal, elle doit être à 11500 bits/sec et sans contrôle de flux :



9. Exemple d'application

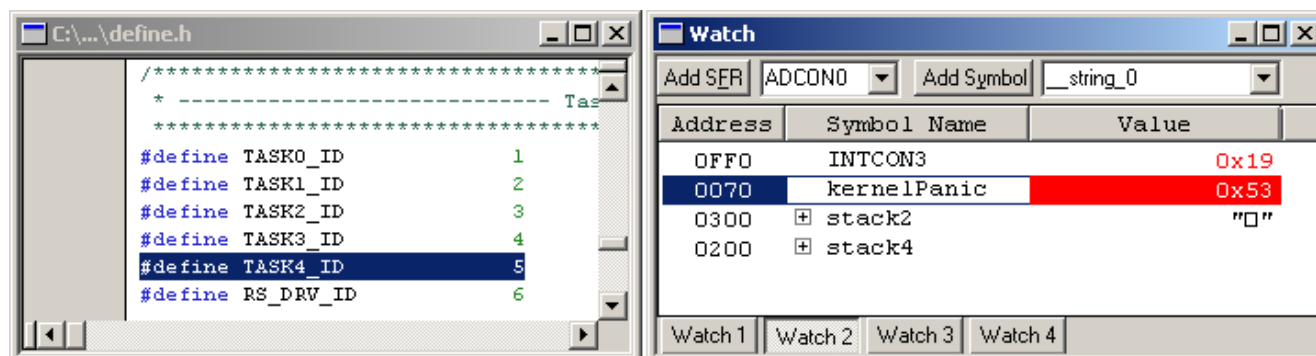


Le tutorial se termine ici par un exemple complet d'application. Les sources de cet exemple sont fournis dans le répertoire /Project/Tutorial de PICos18, ceci vous permettant d'apprécier à quel point il est aisé d'écrire une application pour PIC18 avec PICos18.

> Interruptions multiples

Lorsqu'on exécute l'application du tutorial et qu'on appuie sur les boutons on s'aperçoit vite qu'il existe un problème : le PIC18 semble ne plus bien fonctionner après un instant, surtout si l'on appuie "sauvagement" sur les boutons !

Stopper alors l'application en mode debug si vous utilisez un ICD2 de Microchip ou bien votre simulation si vous travaillez sous le simulateur de MPLAB, et inspectez les variables suivantes :



Address	Symbol Name	Value
0FF0	INTCON3	0x19
0070	kernelPanic	0x53
0300	stack2	"□"
0200	stack4	"□"

Comme vous pouvez le constater la variable "**kernelPanic**" n'est pas nulle. Dans notre cas elle vaut 0x53.

En fait cette variable nous renseigne sur d'éventuel débordement de pile logicielle. Lorsqu'une pile déborde sur une autre, le noyau détecte automatiquement le problème et stoppe l'hémorragie en suspendant la tâche en cause. Ceci permet de limiter la casse, mais surtout de déboguer alors que PICos18 fonctionne toujours.

kernelPanic vaut 0x53.

La partie de droite ("3") signifie que la tâche dont l'identifiant est 3 a vu sa pile déborder. La partie de gauche ("5") signifie que la tâche dont l'identifiant est 5 a vu sa pile se faire écraser par une autre.

Le noyau a donc décidé de geler les 2 tâches car leur fonctionnement n'est plus assuré. Ceci permet au PIC18 de rester fonctionnel malgré le bug, et donc vous autorise à trouver l'origine du bug.

La tâche 2 en charge de l'appuie d'un bouton a fonctionné de façon anormale et sa pile logicielle a débordé sur la pile suivante, celle de la tâche 4.

Mais comment cela se peut-il ?

Et bien 2 cas sont possibles :

- votre code est important et vous avez sous-dimensionné vos besoins en terme de pile logiciel
=> augmentez la taille de la pile à l'origine du problème et recommencez
- une interruption sans fin force une tâche à sauvegarder son contexte (l'ensemble des registres du PIC18 ainsi que sa pile hardware) en boucle ce qui provoque le débordement.

Or nous utilisons des boutons poussoirs, et nous avons vu précédemment que ceux-ci rebondissaient lorsqu'ils étaient pressés. Du coup plusieurs IT en chaîne se produisent et saturent inutilement le PIC18. A chaque IT, on sauvegarde le contexte de la tâche courante, mais les interruptions sont si proches qu'on tourne en boucle dans InterruptVectorL() jusqu'à faire exploser la pile logicielle courante....

Ce phénomène est courant, qu'on utilise ou pas un noyau temps-réel. La seule différence ici, c'est que PICos18 a surveillé l'application en "temps-réel" à l'affût de ce genre de problème et préserve ainsi le reste de l'application. Sans OS, le PIC18 se serait planté... En fait il ne s'agit pas d'un problème d'architecture mais bel et bien d'un problème de code, que nous allons aussitôt corriger.

> Correction de l'application

Le problème vient du fait que nous autorisons toujours les interruptions sur les ports RB1 et RB2 alors qu'on a pas encore commencé à traiter la première interruption. Après tout il ne sert à rien de savoir que le bouton a généré 50 rebonds avant de se stabiliser : il faut juste mémoriser le fait qu'il ait été enfoncé.

Modifier les tâches TASK2 et TASK3 comme suit :

```
TASK(TASK2)
{
    unsigned int i;

    hour++;
    if (hour == 24)
        hour = 0;
    INTCON3bits.INT1IF = 0;
    INTCON3bits.INT1IE = 1;
    TerminateTask();
}

...

TASK(TASK3)
{
    unsigned int i;

    min++;
    if (min == 60)
        min = 0;
    INTCON3bits.INT2IF = 0;
    INTCON3bits.INT2IE = 1;
    TerminateTask();
}
```


Et surtout il faut modifier la gestion des interruptions :

```
void MyOwnISR(void)
{
    TaskStateType State;

    if (INTCON3bits.INT1IF)
    {
        INTCON3bits.INT1IE = 0;
        GetTaskState(TASK2_ID, &State);
        if (State == SUSPENDED)
            ActivateTask(TASK2_ID);
    }
    if (INTCON3bits.INT2IF)
    {
        INTCON3bits.INT2IE = 0;
        GetTaskState(TASK3_ID, &State);
        if (State == SUSPENDED)
            ActivateTask(TASK3_ID);
    }
}

...

void InterruptVectorL(void)
{
    EnterISR();

    if (INTCONbits.TMR0IF == 1)
        AddOneTick();
    /*Here is the next interrupts you want to manage */
    if ((INTCON3bits.INT1IF & INTCON3bits.INT1IE) ||
        (INTCON3bits.INT2IF & INTCON3bits.INT2IE))
        MyOwnISR();
    if ((PIR1bits.RCIF)&(PIE1bits.RCIE))
        RS_RX_INT();
    if ((PIR1bits.TXIF)&(PIE1bits.TXIE))
        RS_TX_INT();

    LeaveISR();
}
```

Comme vous pouvez le constater, MyOwnISR() dévalide les interruptions (INTxIE = 0 ce qui signifie Interrupt Enable à 0) et c'est aux tâches de gérer l'autorisation, une fois seulement que le traitement est réalisé.

Vous pouvez maintenant programmer votre PIC18 et tester le programme en réel, vous verrez qu'en appuyant sur les boutons vous pouvez mettre à jour l'horloge. Toutefois il apparaît un problème : vous avez beau appuyer plusieurs fois de suite sur un bouton, la nouvelle valeur n'est prise en compte que **toutes les secondes**, au moment où l'affichage est rafraîchi. Pour corriger ça nous allons utiliser les capacités temps-réel de PICos18 !

> Modification en temps réel

Pour pouvoir modifier l'affichage à chaque appuie de bouton il faudrait pouvoir informer la tâche d'affichage qu'il est nécessaire de rafraîchir l'affichage avec les nouvelles valeurs.... mais ceci sans perturber l'affichage déjà existant à chaque seconde, c'est-à-dire sans perturber la bon fonctionnement de l'horloge.

Pour cela nous allons poster un événement à la tâche TASK4 :

```
TASK(TASK2)
{
    unsigned int i;

    hour++;
    if (hour == 24)
        hour = 0;
    SetEvent(TASK4_ID, UPDATE_EVENT);
    INTCON3bits.INT1IF = 0;
    INTCON3bits.INT1IE = 1;
    TerminateTask();
}

...

TASK(TASK3)
{
    unsigned int i;

    min++;
    if (min == 60)
        min = 0;
    SetEvent(TASK4_ID, UPDATE_EVENT);
    INTCON3bits.INT2IF = 0;
    INTCON3bits.INT2IE = 1;
    TerminateTask();
}
```

Et dans la tâche TASK4 nous allons nous mettre en attente aussi sur ce type d'événements :

```
...

while(1)
{
    WaitEvent(ALARM_EVENT | UPDATE_EVENT);
    GetEvent(id_tsk_run, &Time_event);

    if (Time_event & ALARM_EVENT)
        ClearEvent(ALARM_EVENT);
    if (Time_event & UPDATE_EVENT)
        ClearEvent(UPDATE_EVENT);

    Printf("%02d : %02d : %02d\r", (int)hour, (int)min, (int)sec);
}

...
```

Aussi n'oubliez pas de rajouter la variable "EventMaskType Time_event" en tant que variable globale ou variable locale à la tâche.

La définition de **UPDATE_EVENT** n'est pas connu de l'application aussi rajoutez là dans le fichier define.h :

```
...

/*****
 * ----- Events -----
 *****/

#define ALARM_EVENT      0x80
#define TASK1_EVENT      0x10
#define UPDATE_EVENT     0x02

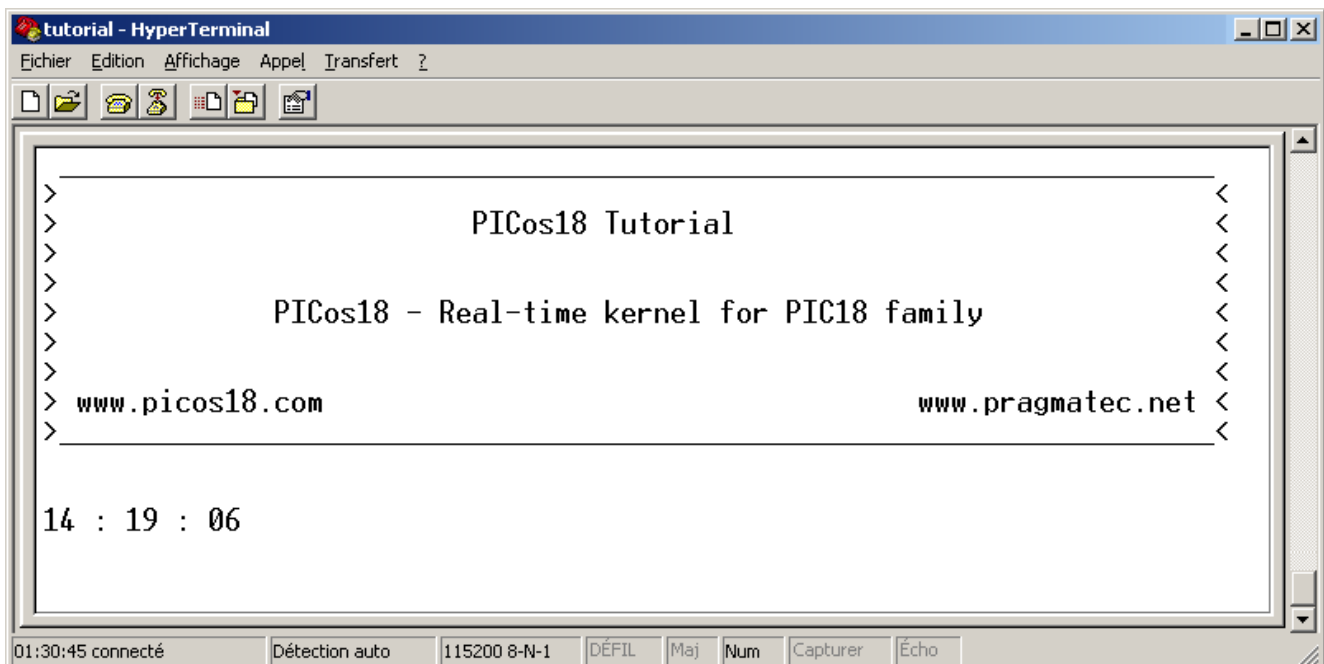
#define RS_NEW_MSG       0x10
#define RS_RCV_MSG       0x20
#define RS_QUEUE_EMPTY   0x10
#define RS_QUEUE_FULL    0x20

...

```

> L'application finale

Voilà, vous pouvez désormais recompilez votre application, la charger dans un PIC18F452 et la tester, vous verrez alors votre horloge s'afficher sous HyperTerminal et vous réussirez à la modifier en temps-réel à l'aide vos boutons poussoirs.

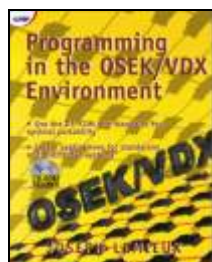


Bien entendu vous pouvez largement simplifier cette application. Ceci n'a pas été fait pour permettre une meilleure compréhension de PICos18. Vous pouvez aussi la compléter, avec votre propre code ou bien en rajouter des drivers de PICos18 : vous pourriez par exemple utiliser un RTC (Real Time Clock) sur bus I2C et la mettre à l'heure directement depuis PICos18 et HyperTerminal...

10. Bibliographie et références

Pour en savoir un peu plus sur la norme OSEK ainsi que sur l'emploi des PIC18 au travers de PIC18, nous vous présentons ici quelques références à explorer.

Bibliographie



Le livre de **Joseph LEMIEUX** des éditions CMPbooks.
Ce livre détaille la réalisation d'un jeu de cartes sur le noyau OsekWorks
de la société WindRiver.

Liens Internet utiles

- www.osek-vdx.org : pour obtenir les normes OSEK/VDX™ au format PDF.
- www.picos18.com : pour obtenir les dernières versions du noyau PICos18.
- www.pragmatec.net : pour obtenir des renseignements concernant les produits développés autour de PICos18.
- www.microchip.com : le site officiel de MICROCHIP, indispensable pour obtenir :
- les data sheets des microcontrôleurs PIC18xxx
- la dernière version de MPLAB®
- la dernière version du compilateur C18 en commande
- www.fsf.org : pour obtenir des renseignements la licence GPL.

11. Glossaire

Banner :	Commentaires situés en tête de fichiers destinés à renseigner tout utilisateur sur la nature du fichier, les fonctionnalités codées, le ou les auteurs, les dates de modifications ainsi qu'une référence à la licence GPL
Breakpoint :	Point d'arrêt en français. Il s'agit d'une fonctionnalité offerte par le simulateur MPLAB : lorsque le programme atteint le point d'arrêt la simulation stoppe immédiatement. Placer des points d'arrêt dans les différentes tâches de votre application permet de suivre l'évolution de votre programme.
Compilateur :	Programme destiné à convertir un code source (par exemple file.c) en un fichier objet (par exemple file.o). Le compilateur C18 de Microchip permet de convertir le code de votre application en une série d'instructions compréhensibles par le PIC18. Le code généré n'est pourtant pas le code final car il contient encore de nombreux symboles (noms de variables, noms de fonctions, ...).
Linker :	Programme dont la fonction est de résoudre les symboles contenus dans les différents fichiers objets (par exemple file.o). Dans le cas d'une tâche faisant appel à un service du noyau, le linker va pouvoir déterminer à quelle adresse faire le saut d'appel de fonction, connaissant la position en mémoire des services du noyau. Pour déterminer la position du code en mémoire le linker s'appuie sur le script du linker (par exemple pic18f452.lkr).
Mapping :	Le fichier de mapping (par exemple projet.map) est un fichier créé par le linker. Son but est de renseigner le développeur sur la position des variables et des fonctions en mémoire, après que le linker est délibérément choisi des emplacements.
GPL :	Désigne la « General Public License », une licence libre dite « publique » à opposer à une licence privée. Cette licence empêche toute personne physique ou morale de s'appropriier le code protégé par la licence GPL, en déposant un brevet par exemple. La licence GPL garantit la disponibilité du code source originale ainsi que ces diverses évolutions.
Open source :	Un projet est dit « open-source » lorsque celui-ci fait l'objet d'une libre divulgation de son code source. Cette divulgation est spécifiée par la licence GPL qui précise que le ou les responsables doivent pouvoir fournir les sources sous une quelconque forme en cas de demande. Concernant PICos18, les sources sont mises à disposition sur le site www.picos18.com .

OSEK/VDXÔ :	C'est la norme sur laquelle repose PICos18. Créée à l'origine pour le milieu automobile, elle est de plus en plus utilisée en robotique et en automatisme. Elle convient en effet parfaitement aux systèmes à base de microcontrôleurs 8 bits bénéficiant de peu de mémoire.
Pile des appels de fonctions :	Lorsque l'on écrit un code en langage C on cherche à factoriser le code le plus possible sous forme de fonction. A chaque appel de fonction, le microcontrôleur PIC18 mémorise l'adresse du code appelant dans une zone appelée la pile des appels de fonctions ou « pile hardware ». Lorsque la fonction en C se terminera par l'instruction « return », le PIC18 utilisera l'adresse mémorisée dans la pile pour revenir au code précédent. Plus exactement c'est l'adresse suivante qui est mémorisée pour permettre la bonne continuation du programme.
Scheduler :	Le scheduler est la partie essentielle d'un noyau. En français on le trouve parfois sous le terme « moniteur ». Le scheduler a pour fonction de déterminer la prochaine tâche active. Dans le cas de PICos18 le scheduler choisit d'activer la tâche à l'état READY la plus prioritaire. Si aucune tâche n'est prête le noyau entre dans un état « idle » qui lui permet d'attendre l'arrivée d'un nouvel événement. Le scheduler est codé en assembleur dans le fichier kernel.asm de PICos18.
Stack :	Nom anglais désignant la pile des appels de fonctions. Dans le cas de PICos18 cette « stack » est sauvegardée à chaque fois qu'il y a préemption, c'est-à-dire passage d'une tâche à une autre. En effet l'appel à des fonctions en C peut corrompre le contenu de la « stack » si bien qu'il est nécessaire de la sauvegarder.
Tick :	Un tick est une unité de temps. Avec un microcontrôleur PIC18xxx cadencé à 40 MHz (quartz à 10 MHz et PLLx4 activée) la valeur d'un tick est calibré à 1 ms. La mise à jour des alarmes se fait sur cette base. Consulter le paragraphe « recommandations » si vous souhaitez modifier cette valeur.
Watchdog :	Le Watchdog est un dispositif électronique intégré aux PIC18xxx qui permet de redémarrer le microcontrôleur lorsque celui-ci ne répond plus. Pour éviter ce redémarrage intempestif, il est nécessaire de mettre le Watchdog à zéro fréquemment. Consulter la documentation Microchip pour de plus amples informations.