République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Dr. Yahia Farès de Médéa
Faculté des Sciences et de la Technologie
Département du Génie Electrique et de l'Informatique



Projet de Fin d'Études

Pour l'obtention du diplôme
de Master en Génie Électrique

Option : Systèmes Intelligents et Communicants

Thème

Simulation d'un RTOS sur un DATA LOGGER portatif dédié à l'AQM et à l'analyse des chutes

Présenté par:

⇨ *Hirchi yamena*

Proposé et Dirigé par :

Mr.Chadli Samir

Année Universitaire 2011-2012

Dédicace :

Je dédie Ce travail

A mes très chers parents, ma mère que l'est sacrifiée pour moi et elle n'a jamais cessé de m'encourager, mon père a beaucoup fait pour moi.

À mes très chers sœurs et frères qui m'ont assurés un soutien permanent

Soufiane, Khalida, Mansour, Chaima

A mes très chères amies qui sont vraiment mes sœurs

Houria, Houda, Rakia, Hakima, Hassiba, Khadidja, Meriem, Soumia.z, Hamida, Soumia.m, Noura, Fatima, Zineb, Fouzia, Samira, Hafida.

A tous mes amis.

Et toute ma famille qui n'a pas cessé de me soutenir et de

m'encourager pendant les moments difficiles, que

Dieu les protège

A mes collègues de la section de

SIC

Remerciement :

Je tiens avant tout à remercier mon dieu qui, sans lui, rien de tout cela n'aurait lieu

Je remercie mes parents qui ont été présents dans les hauts et les bas, à mon encadreur Mr.CHADLI qui m'a Suivi jusqu'au bout.

On remercie vivement tout mes chers enseignants.

Je tiens également à remercier tous ceux qui ont contribué de près ou de loin à l'élaboration de ce modeste travail.

Remerciements vont également à tous les membres de jury, qui font l'honneur de juger mon travail.

Introduction générale

Chapitre 1

Généralités

Chapitre 2

Le RTOS: PICos18

Chapitre 3

Réalisation de l'application multitâche

Conclusion

Bibliographie

Bibliographie

[1] : Les microcontrôleurs dans les systèmes embarqués -Elham Firouzi
https://prof.hti.bfh.ch/fileadmin/home/fue1/Microcontroller_in_Embedded_Systems/Script/ManuscrituCESV30a.pdf

[2]: « Le multitâche avec PIC18F452 », H. ZAHDOURH et A. YOUBI, Mémoire D'ingénieur en électronique, Université des Sciences et Techniques d' Oran - 2007.

[3]: « Etude et réalisation d'un instrument portatif pour l'étude de l'équilibre et de la foulée dynamique humaine », R. Dabou et S. YOUBI, Mémoire D'ingénieur en Instrumentation et contrôle, Université des Sciences et de la Technologie Houari Boumediene USTHB- 2006.

[4] : PICos18 Noyau temps réel pour PIC18,
http://www.picos18.com/Download/PICos18_tuto_fr.pdf

[5] : L'ordonnancement temps réel, http://www.infres.enst.fr/~pautet/inf342/supports/rt-sched.pdf

[6] : L'ordonnancement centralisé de tache temps réel
http://cedric.cnam.fr/~bouzefra/cours/cours_LS/ordo_centralise.pdf

Sommaire

Introduction générales ……1

Chapitre 1 : Généralités

I. Les systèmes embarqués temps réels……..……...5

I-1-Les systèmes embarqués …….6

I-2-Les systèmes temps réels.……...……...……..6

I-3- Architecture des Systèmes Temps Réel……..……...7

I-4-Les Systèmes d'Exploitation pour Systèmes Embarqués.…….....…….....7

I-5-Noyaux pour systèmes embarqués temps réel.……8

II. L'AQM (l'analyse quantifieur de la marche)……..9

III. Description de la partie software ..……12

IV .Description des programmes utilisés…….….14

IV-1-L'organigramme du programme principal...……...….14

IV-2-Les organigrammes des sous programmes …….....15

IV-3- La routine d'interruption……...18

V. La nécessité d'un RTOS……..19

Chapitre II: Le RTOS PICos18

I. Les RTOS utilisés avec les microcontrôleurs Pic……....21

II. Présentation et description du noyau……....23

II-1- La norme OSEK/VDX…….....23

II-2- La structure du noyau RTOS PICos18…….....24

II-3- L'organisation du code source de PICos18……..24

II- 4- Les fichiers essentiels du code source de PICos18…….26

II-5- Les Drivers……...31

III. Les outils de développement associés au PICos18……..31

III-1- Le compilateur C18……31

III-2- L'environnement de développement MPLAB……....32

IV. L'ordonnancement temps réels……....32

IV-1- Les Algorithmes d'ordonnancement……..….33

IV-2- Les algorithmes d'ordonnancement des taches périodiques…….…34

Chapitre III: La réalisation de l'application multitâche :

I. Préparation de l'environnement de développement ……40
II. Création du projet App_Mesure ……..40
III. Les des différentes taches du projet App_Mesure ……..41
III-1- La tache tsk_task0.c ……..42
III-2- La tache tsk_task1.c ……..42
III-3- La tache tsk_task2.c ……..43
III-4- La fichier int.c ……...43
III-5- Création des différentes taches……..46
VI. Les drivers ……48

La liste des figures

Figure n° I-1 : La structure générale d'un système embarqué .....……6
Figure n° I-2 : stimulation et détection de l'activité réflexe par EMG …..…….. ...10
Figure n° I-3 : marqueurs rétro-réfléchissant disposés sur le sujet pour l'enregistrement vidéo….....11
Figure n° I-4 : La plate forme de force pour l'analyse de la locomotion humaine……...12
Figure n° I-5 : synoptique du data logger utilisé…..…….....13
Figure n° I-6 : l'organigramme du programme principal……...….14
Figure n° I-7 : l'organigramme du sous programme de mesure des accélérations .....…..….15
Figure n° I-8 : l'organigramme du sous programme de stockage sur carte mémoire SD ...…..16
Figure n° I-9 : l'organigramme du sous programme de gestion des Leds...……...……18
Figure n° I-10 : l'organigramme de la routine d'interruption.....19
Figure n° II-1 : le Principe de gestion des schuduler préemptif et coopératif ……22
Figure n° II-2 : structure de noyau PICos18 …….....24
Figure n° II-3 : l'organisation de noyau PICos18 …….....25
Figure n° II-4 : les différentes états des taches dans PICos18 …….....28
Figure n° II-5 : le schéma général du processus de compilation sous C18…….….31
Figure n° II-6: Le model canonique d'une tache de temps réel …….…….....32
Figure n° II-7: le temps max d'utilisation du processeur pour l'analyse RM…….....34
Figure n° II-8: e chronogramme d'ordonnancement RM …… ...……...35
Figure n° II-9: le chronogramme d'ordonnancement DM …….…..35
Figure n° II-10: le chronogramme d'ordonnancement EDF ……...……....37
Figure n° II-11: le chronogramme d'ordonnancement LLF ……...……....38
Figure n° III-1 : La structure de dossier PICos18…….....41
Figure n° III-2 : les deux types d'interruption des PIC18 … …….…….....43
Figure n° III-3 : la configuration des ports COM hyper terminal ……...…...50
Figure n° III-4 : un exemple de configuration des paramètres …….....50
Figure n° III-5 : la fenêtre principal de l'environnement MPLAB …….....51

ملخص:

في هذا العمل قمنا بدراسة نظام الاستغلال في الوقت الحقيقي : PICos18 ، الذي يبرمج على المتحكمات المصغرة من نوع : PIC18 فيما بعد قمنا ببرمجة هذا النظام على نموذج اختباري لمسجل بيانات يستعمل للتحليل الكمي لحركة المشي و لدراسة حوادث السقوط. هذا الجهاز يعتمد على متحكـــم مصغر من نوع : PIC18F452 . هذه البرمجة سمحت لنا بالوصول إلى تشغيل متعدد المهام لمختلف الأجزاء المكونة للبرنامج المحمول على أساس مفهوم الأفضلية.

كلمات مفتاحية : نظام استغلال، تحليل كمي للمشي، نظام استغلال في الوقت الحقيقي ، PICos18.

Résumé:

Dans ce travail, nous avons étudié le Système d'exploitation temps réel (RTOS) PICos18, par la suite nous l'avons programmé sur un prototype d'un Data logger dédié pour l'analyse quantifié de la marche et l'étude des chutes. Le Data logger est à base d'un microcontrôleur Pic18F452. Cette implémentation nous a permis d'avoir une approche d'exécution multitâche des procédures du programme embarqué à base de la notion de priorité.

Mots clés : SE, AQM, SETR, PICos18.

Abstract:

In this work, we studied the real-time operating system (RTOS) PICos18, then we have programmed a prototype of a dedicated Data logger for walking quantified analysis and fall study. The data logger is based on a PIC18F452 microcontroller. This implementation allows us to have a multitasking approach of embedded program procedures based on the concept of priority.

Keywords: OS, AQM, RTOS, PICos18.

Introduction

Sans aucun doute l'ère actuel est celui des microcontrôleurs, ils sont présents dans un grand nombre de produits de la vie courante, leurs nombre à dépasser depuis plusieurs années celui des PC et compatibles.

Parmi les microcontrôleurs les plus présents sur le marché ceux de la firme Microchip, qui sont les Pics. Ces derniers sont à la base d'un grand nombre de systèmes embarqués, ces dispositifs sont une combinaison de hardware (CAN, USB, RS232, … intégrées) et de software. Les outils de programmation classique avec leur environnement de développement intégré n'arrivent pas dans certaines situations à combler les besoin en matière de programmation car le soft embarqué devient de plus en plus complexe afin de satisfaire des besoins croissants en matière de confort, de rapidité, de sécurité, …etc.

Pour cela une autre approche commence à s'imposer dans la programmation des applications embarquées. Cette approche se base sur l'utilisation des noyaux temps réel qui permettent aux applications temps embarqué d'être développées plus facilement et d'être entretenues d'une manière plus simple.

L'utilisation d'un noyau TR simplifie le processus de conception en divisant le code de l'application en tâches séparées tout en assurant une approche d'exécution multitâches d'un ensemble de procédures qui se partagent les ressources du système embarqué. Ainsi les noyaux TR assure une meilleure exploitation des ressources par l'intermédiaire d'un ensemble de services d'une grande utilité

Dans ce travail nous nous proposons d'implémenter l'RTOS PICos18 de la société Pragmatec qui a l'avantage d'avoir un code libre. Le support matériel de la réalisation est un prototype d'un Data logger réalisé autour d'un Pic18f452, ce Data logger est dédié à l'analyse quantifiée de la marche (AQM) et à l'étude des chutes chez les personnes âgées.

Notre mémoire est organisé en trois chapitres :

- Le chapitre 1 présente un ensemble de généralités sur les systèmes embarqués, les systèmes temps réel, le prototype du Data logger, le soft embarqué, ….etc.
- Le chapitre 2 décrit les éléments qui constituent le RTOS PICos18 de la société Pragmatec qu'on a utilisés au cours de ce projet.
- Le dernier chapitre présente la réalisation d'un projet multitâche en utilisant PICos18 sur le Data logger assurant l'acquisition des mesures via des capteurs et leurs stockage sur une carte mémoire SD.

Chapitre I

Généralités

I. Les systèmes embarqués temps réels :

L'évolution majeure des systèmes d'exploitations (standards ou embarquées), concerne le support de l'exécution concurrente de plusieurs traitements et a donné naissance aux systèmes dits multitâches. Toutefois, les systèmes multitâches même s'ils permettent de partager l'accès au microprocesseur n'offrent que peu de garanties concernant la disponibilité de cette ressource.

Les systèmes temps réels sont donc une réponse au problème de disponibilité de la ressource microprocesseur. Le temps réel est un domaine à part entier de recherche qui est le sujet de nombreuses études multi-domaines (logique temporelle, théorie des files d'attente, systèmes d'exploitation ...).

I-1-Les systèmes embarqués :

Les systèmes embarqués (en anglais «*embedded systems*») sont composés d'une partie matérielle et d'une partie logicielle. Ils sont intégrés ou embarqués dans un produit plus large (un robot, une automobile, un téléphone portable, un agenda électronique, une machine à café, un distributeur automatique de billets, un Data logger, une carte CCP, etc...).

Les systèmes embarqués assurent en générale des procédures de contrôle, de traitement du signal, de surveillance, etc… des différents composants du produit ou du produit dans son ensemble. La fonctionnalité d'un système embarqué se limite à l'exécution d'une ou quelques tâches dédiées.

Quelle que soit la nature et la complexité d'un système embarqué, on peut le décomposer en deux parties essentielles :

- ***Le système contrôlé :*** qui représente l'environnement (le procédé) équipé d'une instrumentation qui réalise l'interface avec le système de contrôle (de commande).

- ***Le système de contrôle :*** regroupe les éléments matériels (microprocesseurs, mémoires, Convertisseurs, …) et logiciels dont la mission est d'agir sur le procédé via les actionneurs en fonction de l'état de ce procédé indiqué par les capteurs de manière à maintenir ou conduire le procédé dans un état donné (Figure n° I-1).

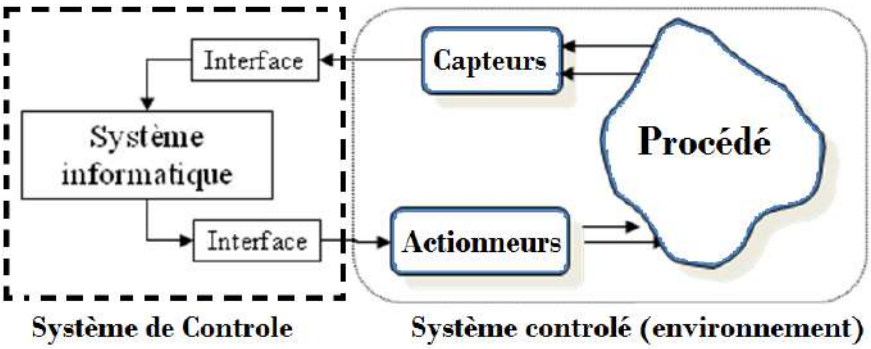


Figure n° I-1 : La structure générale d'un système embarqué

I-2- Les systèmes temps réels :

Il existe plusieurs définitions du temps réel parmi lesquelles nous citons la définition adoptée en 1988 par le Groupe CNRS sur la problématique temps réel, dans laquelle le comportement d'un système informatique est qualifié de « temps réel » lorsqu'il est assujetti à l'évolution d'un procédé qui lui est connecté et qu'il doit piloter ou suivre en réagissant à tous ses changements d'états.

Une autre définition qualifie un système temps réel comme étant un système dont le comportement dépend : de l'exactitude des traitements effectués et du temps où les résultats sont produits.

Le fait de rater une échéance (un retard) par le système conduit à des erreurs de fonctionnement. Un système temps réel n'est pas forcément un système ultra rapide (mauvaise interprétation du temps réel), car l'objectif n'est pas de traiter les données rapidement mais de les traiter à temps. Ainsi un système temps réel doit toujours livrer des réponses correctes dans des délais prédéfinis. Le dépassement de ces délais se traduit par un dysfonctionnement du système. Selon le domaine de l'application le délai (échéance) peut correspondre à quelques microsecondes, millisecondes voire secondes. Il est seulement important que le résultat soit fourni à la limite de ce temps défini.[1]

➢ ***Contraintes temps réel mou/dur :*** Il existe deux types d'échéances :

- ***Échéance dure (hard deadline) :*** dans ce cas l'arrivée après échéance d'un événement attendu déclenchera une exception (tâche de gestion d'erreurs).

- ***Échéance lâche (molle) (soft deadline):*** dans ce cas l'arrivée après échéance d'un événement attendu ne déclenchera pas une exception (la gestion d'erreurs n'est pas obligatoire).

Selon le type d'échéancier toléré par les systèmes temps réel, on peut les classer en deux catégories :

Les systèmes TR dur : dans ce cas l'arrivée après échéance d'un événement attendu ne doit pas se produire (nécessité de gestion d'erreurs), exemple : Le Pilote automatique d'avion, le Freinage ABS des voitures, …etc.

Les systèmes TR mou : dans le cas où l'arrivée exceptionnelle après échéance d'un événement attendu ne mettra pas le système en danger (erreur), exemple : le Système de distribution de son (streaming), distributeur de billet, ….etc.

On parle de contrainte temps réel dur lorsque le système doit respecter toujours les spécifications temporelles, il présente alors un comportement temporel déterministe (Automation industrielle, applications liées à la sécurité, etc…).

I-3-Architecture des Systèmes Temps Réel :

Les systèmes temps réel peuvent être classés selon leur couplage avec des éléments matériels avec lesquels ils interagissent.

- soit directement dans le procédé contrôlé : c'est ce que l'on appelle des systèmes embarqués (*embedded systems*). Le procédé est souvent très spécialisé et fortement dépendant du calculateur. Les exemples de systèmes embarqués sont nombreux : contrôle d'injection automobile, stabilisation d'avion, électroménager, missile. C'est le domaine des systèmes spécifiques intégrant des logiciels sécurisés optimisés en encombrement et en temps de réponse.
- soit le calculateur est détaché du procédé : c'est souvent le cas lorsque le procédé ne peut être physiquement couplé avec le système ou dans le cas général des contrôle/commandes de processus industriels. Dans ce cas, les applications utilisent généralement des calculateurs industriels ou des automates programmables industriels comme dans les chaînes de montage industriels par exemple.

I-4- Les Systèmes d'Exploitation temps réel pour systèmes Embarqués :

Les Systèmes d'Exploitation (SE) servent à rendre disponible les éléments matériels d'un système donné pour les différentes tâches. En fonction du contexte d'exécution, des objectifs du SE, les types de tâches qu'il supporte, leurs architecture, peuvent varier, même si leurs rôle reste toujours globalement identique.

L'usage de Systèmes d'Exploitation est devenu nécessaire dans les systèmes embarqués, du fait de la complexité croissante de ces derniers, de la présence de fortes contraintes en temps réel, de la limitation des ressources disponibles, tant en mémoire qu'en énergie disponible et donc en puissance de calcul. Ces Systèmes d'Exploitations dédiés pour les Systèmes Embarqués, se distinguent par rapport aux SE classiques par leur aptitude à gérer les aspects des applications temps-réel, ces Systèmes d'Exploitation temps réel (RTOS : Real Time Operating Systems) se caractérisent par leur comportement déterministe et prévisible.[2]

I-5- Noyaux pour systèmes embarqués temps réel :

Ce sont des noyaux exécutifs de petites tailles, assez performants et particulièrement appropriés à des systèmes embarqués simples où les besoins sont essentiellement une exécution et un temps de réaction aux événements rapides (alarmes, interruptions...). Ils offrent un ordonnancement par priorités (généralement un ensemble de priorités prédéfinies est offert), une horloge globale, des primitives pour la préemption des tâches, pour le retardement ainsi que des mécanismes de synchronisation, etc. Toutefois, ces noyaux sont inefficaces pour des systèmes complexes ou de taille importante. En effet, dans de pareils systèmes, les contraintes sont plus compliquées et portent à la fois, sur le temps, les ressources, la communication et il est extrêmement difficile de prédire avec de pareils noyaux de base que les contraintes seront vérifiées. La principale cause est due aux délais introduits pour la gestion de ces systèmes. Dans le cas où ces noyaux sont utilisés, on se retrouve aussi avec des phénomènes d'inversion de priorités car les mécanismes d'ordonnancement utilisés dans ces noyaux sont de très bas niveau. Egalement, des situations où les ressources ne sont pas disponibles à temps, où les messages ne sont pas délivrés à temps, etc.[2]

I-5-1- Les Caractéristiques d'un noyau embarqué temps réel :

C'est en fait un ensemble de fonctionnalités, regroupées sous le terme de ***services*** pour la plupart, qui forment le noyau. 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. 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 ***multiprogrammation*** 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 ***multitâches préemptif***. 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 ***multitâches coopératif.***

Le noyau multitâche 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 ***propriétés*** différentes, et doivent pouvoir répondre à un événement dans un temps plus court. 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 ***déterminisme***.[2]

II. L'AQM (l'analyse quantifieur de la marche) :

L'analyse quantifiée de la marche (AQM) est un examen complémentaire permettant d'enregistrer les données cinématiques, cinétiques et électromyographiques. Les variables mesurées sont les suivantes :

1. La vitesse de marche.
2. La fréquence des cycles de marche.
3. La longueur des pas.
4. La symétrie des demi-pas droit et gauche pour avoir l'indice de symétrie globale.
5. La régularité temporo-spatiale des cycles pour exprimer la variabilité de la marche.
6. Le déplacement vertical pour l'estimation des déplacements de la région lombaire entre les points les plus hauts et plus bas.
7. L'activité mécanique selon les 3 axes.
8. Le coût énergétique calculé à partir de la somme des 3 activités mécaniques corrélées à la consommation d'oxygène et rapportées à la vitesse de marche.
9. Les contraintes subies par les membres inférieurs.

A l'heure où les pratiques médicales se doivent de faire la preuve de leur efficacité, les systèmes d'AQM apparaissent comme des outils d'aide à la décision médicale qui permettent aux médecins :

1. L'évaluation des traitements médicaux (maladie de Parkinson, toxine botulique, etc...), évaluation des traitements par rééducation, des programmes d'entraînement thérapeutiques ou

préventifs pour les pathologies neurologiques, et les pathologies dégénératives de l'appareil locomoteur (arthroses, polyarthrites…).

2. L'évaluation et l'amélioration des prothèses ou des orthèses pour les amputés, les personnes atteintes de scoliose, etc...

3. L'évaluation du résultat de traitements chirurgicaux en orthopédie adulte (prothèse de hanche ou de genou, ligament plasties, ostéosynthèse de fracture ou résection tumorale des membres), en orthopédie pédiatrique (chirurgie multi sites de l'infirme moteur, chirurgie des pathologies congénitales: pied bot, varus, équin; luxation congénitale de hanche ou chirurgie de la scoliose…) ; ou en neurochirurgie, (chirurgie de la maladie de Parkinson, rhizotomies…).

4. Evaluation du programme d'entraînement pour les sportifs de haut niveau.

Dans la pratique hospitalière, il existe plusieurs méthodes d'analyse de la marche parmi lesquelles on peut signaler l'analyse cinésiologique, l'analyse cinématique et l'analyse cinétique.[3]

- ✓ ***L'analyse cinésiologique*** fait appel à l'électromyographie, et permet de comprendre les actions musculaires au cours de la marche, Le système d'enregistrement EMG est un représentant majeur de cette méthode d'analyse. Il permet d'enregistrer à l'aide d'électrodes de surface l'activité électrique des muscles ainsi que leur période de contraction rapportée au cycle de marche. La Figure n° I-2 représente les conditions de stimulation et de détection de l'activité réflexe. SE indique la stimulation du nerf sciatique, R l'emplacement des électrodes d'EMG sur le mollet. En B, on a une comparaison des réponses électriques en fonction de l'intensité d'excitation.

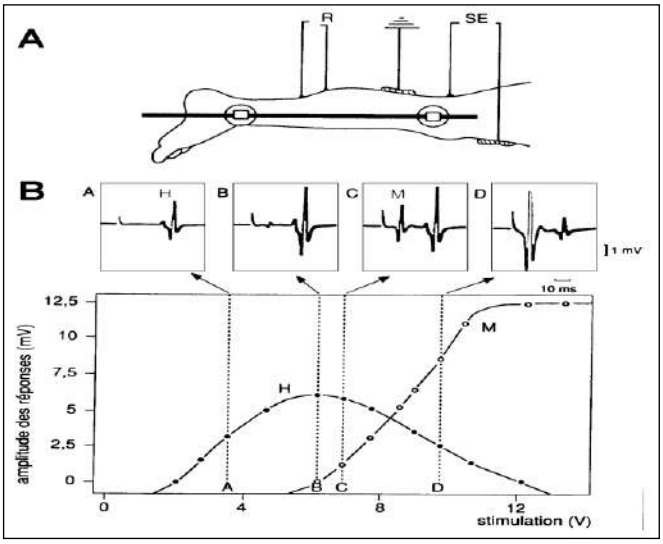


Figure n° I-2 : stimulation et détection de l'activité réflexe par EMG

- ✓ ***L'analyse cinématique*** utilise des enregistrements vidéo et des systèmes optoélectroniques. Ces outils d'analyse du mouvement en 3D permettent l'enregistrement simultané du déplacement de vue de face, de profil et de dessus et donnent la possibilité d'une visualisation du mouvement au ralenti ou l'arrêt sur image. Comme indiqué sur la figure n°I-3, on dispose sur le sujet des marqueurs rétro réfléchissants positionnés sur des repères anatomiques précis. Le déplacement des marqueurs reconstruit en 3 dimensions permet, à l'aide d'un logiciel d'animation, de reproduire le déplacement des différents segments corporels. Cette technique est largement utilisée dans les laboratoires de biomécanique vu son caractère analytique et descriptif, mais sa mise en œuvre reste limitée à des applications cliniques principalement pour des raisons de coûts.

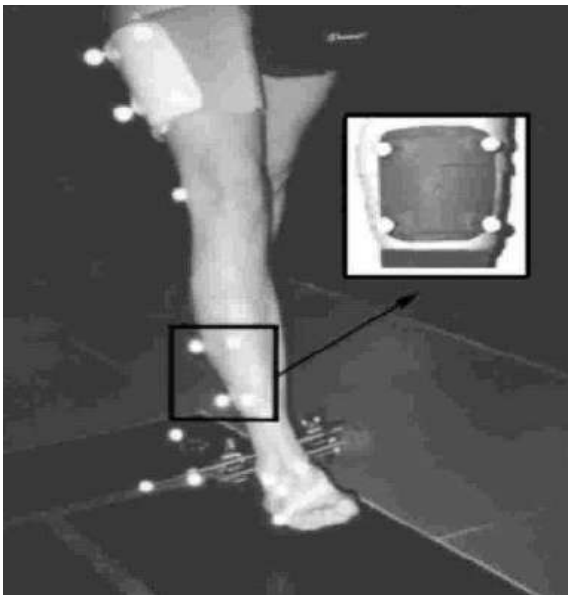


Figure n° I-3 : marqueurs rétro-réfléchissants disposés sur le sujet pour l'enregistrement vidéo

- ✓ ***L'analyse cinétique*** s'intéresse à l'étude des causes du mouvement : forces, moments, accélérations, etc. Il existe plusieurs appareils de mesure qui peuvent être utilisés parmi lesquels on trouve les semelles baro-podométriques, les plates-formes de forces (Figure n° I-4). La méthode accéléromètrique est une méthode d'analyse dynamique qui utilise des capteurs d'accélérations fixés sur le corps. Cette méthode présente de nombreux avantages tels que l'adaptation à la mesure ambulatoire dans des conditions normales d'activité du patient et d'un athlète. Elle peut également s'appliquer à un grand nombre de sujets en raison de sa facilité d'emploi et de sa rapidité de mise en œuvre, elle est accessible à toutes les analyses de la locomotion, qu'elles soient animales ou humaines.

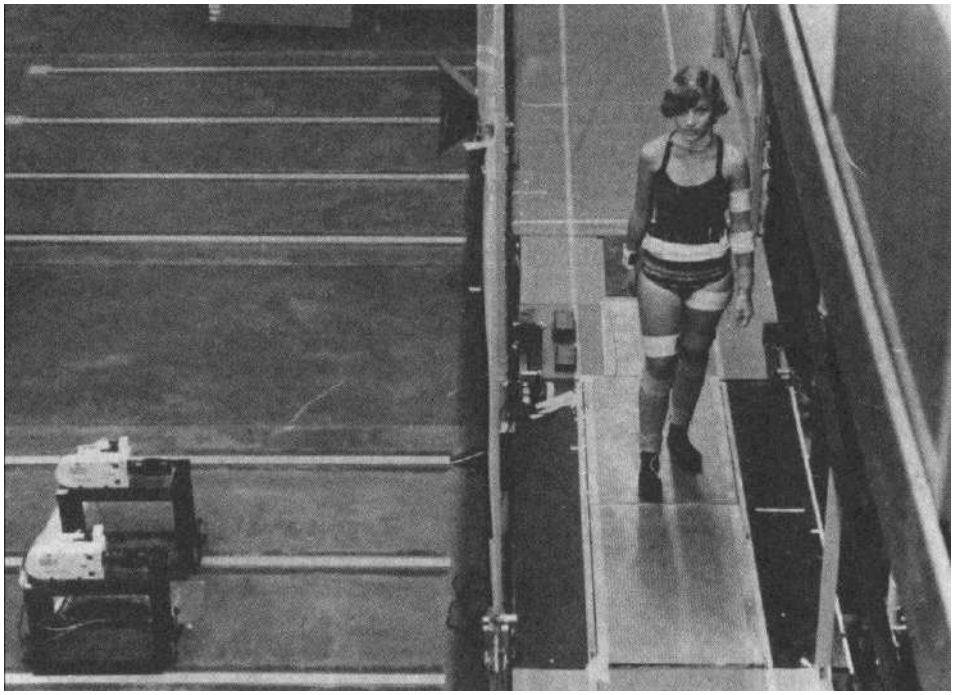


Figure n° I-4 : Plate forme de force pour l'analyse de la locomotion humaine

III. Présentation de la partie hard du système :

La plate forme hardware qui a été utilisé dans ce projet est un prototype d'un Data logger développé au sein du laboratoire d'instrumentation LINS de l'Université des Sciences et de la Technologie Houari Boumediene (USTHB), destiné à *l'analyse cinétique de la marche* et *l'étude des chutes chez les personnes âgés*.

Le Data logger est un système d'enregistrement de données, c'est un dispositif électronique qui enregistre les données au cours du temps via des capteurs intégrés ou par l'intermédiaire de capteurs externes. Ces dispositifs sont généralement basés sur un processeur numérique (ou ordinateur), ayant de petite taille, alimentés par batterie, portables, et équipés d'une mémoire interne pour le stockage de données. Certains Data loggers s'interfacent avec un ordinateur personnel et utilise un logiciel pour visualiser et analyser les données recueillies, tandis que d'autres ont un dispositif local d'interface (clavier, écran) et peuvent être utilisés comme des appareils autonomes.

La figure n° I-5 présente le schéma synoptique du prototype de Data logger utilisé :

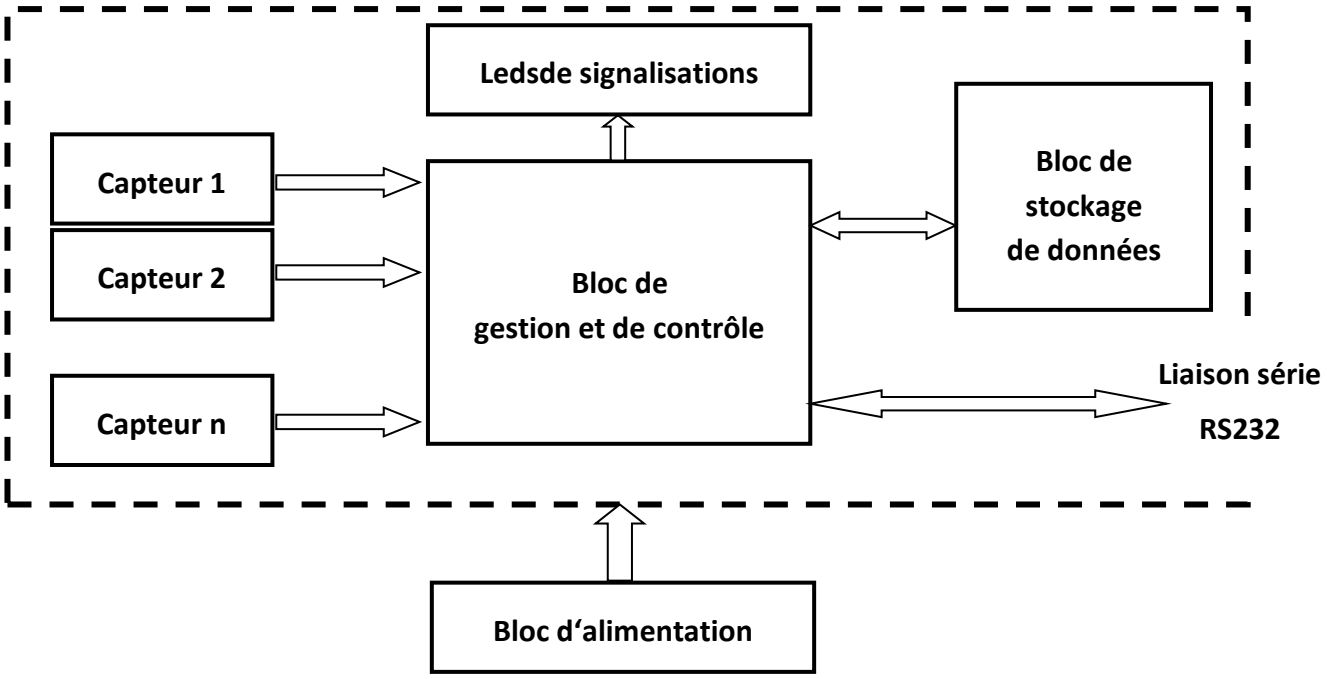


Figure n° I-5 : Synoptique du Data logger utilisé

Ce Data logger est constitué des blocs suivants :

1. ***Un bloc de gestion et de contrôle***, ce dernier est réalisé autour d'un microcontrôleur de la firme de MICROCHIP de la famille High End, le Pic 18F452. Le rôle de ce dernier est d'assurer le contrôle des différents capteurs et du reste des blocs du système, la gestion des mesures et leurs stockage en mémoire, la conversion analogique/numérique des signaux des capteurs analogiques, la communication avec un PC via une liaison série.
2. ***Un bloc de stockage de données,*** qui assure l'enregistrement des mesures réalisés par le système en vue d'un traitement ultérieur. Ce bloc est à base d'une carte mémoire SD « Secure Digital », qui est une carte mémoire amovible de stockage de données numériques. Depuis 2010,ce type de carte mémoire fait office de standard de stockage, elle est largement utilisée pour le stockage de fichiers dans les appareils photo numériques, les caméscopes numériques, les consoles de jeux vidéo, etc…., leur capacité s'échelonne jusqu'à 128 Go.
3. ***Un ensemble de capteurs,*** permettant de quantifier les grandeurs physiques à mesurer (accélération, vitesse, inclinaison, cap, etc…). Le Data logger utilisé peut être interfacé avec des capteurs à sortie analogique ou à sortie numérique (I2C, PWM, SPI, …).
4. ***Un bloc d'alimentation,*** ce dernier fourni deux tensions continues (+5V et +3.3V) nécessaires pour le fonctionnement du microcontrôleur, la mémoire SD et les différents capteurs.
5. ***Deux Leds de signalisation,*** pour fournir une information visuelle sur le fonctionnement du Data logger (phase d'acquisition, alerte de saturation de la mémoire, …).

Le Data logger possède également une liaison série de type RS232 qui permet l'interfaçage avec un PC ou autre dispositif série.

IV. Description de la partie software :

Pour réaliser une *analyse cinétique de la marche,* le Data logger a été muni d'accéléromètres (capteurs d'accélération), configurés pour fonctionner en inclinomètres (capteurs d'inclinaisons) grâce à la mesure de la valeur de la pesanteur terrestre. Sur chaque segment du membre inferieur, droit au nombre de 3 (et également celui du gauche), un accéléromètre triaxiales est fixé permettant la quantification de l'accélération terrestre sur les trois axes (x, y et z), par conséquence en connaissant la valeur exacte de la pesanteur et par l'utilisation des formules trigonométriques on arrive à évaluer les angles d'inclinaisons d'un segment donné par rapport aux trois axes du repère (xyz).

Le suivi des angles d'inclinaisons de l'ensemble des segments des deux membres inférieurs en fonction du temps permet à l'aide d'un traitement approprié de reconstitué le mouvement de la marche d'un sujet d'étude. Ainsi, le Data logger réalise des acquisitions (mesures d'accélérations) durant des périodes programmées et les sauvegardent sur la carte mémoire SD.

IV-1- L'organigramme du programme principal :

La figure n° I-6 présente l'organigramme générale du programme embarqué sur le système Data logger :

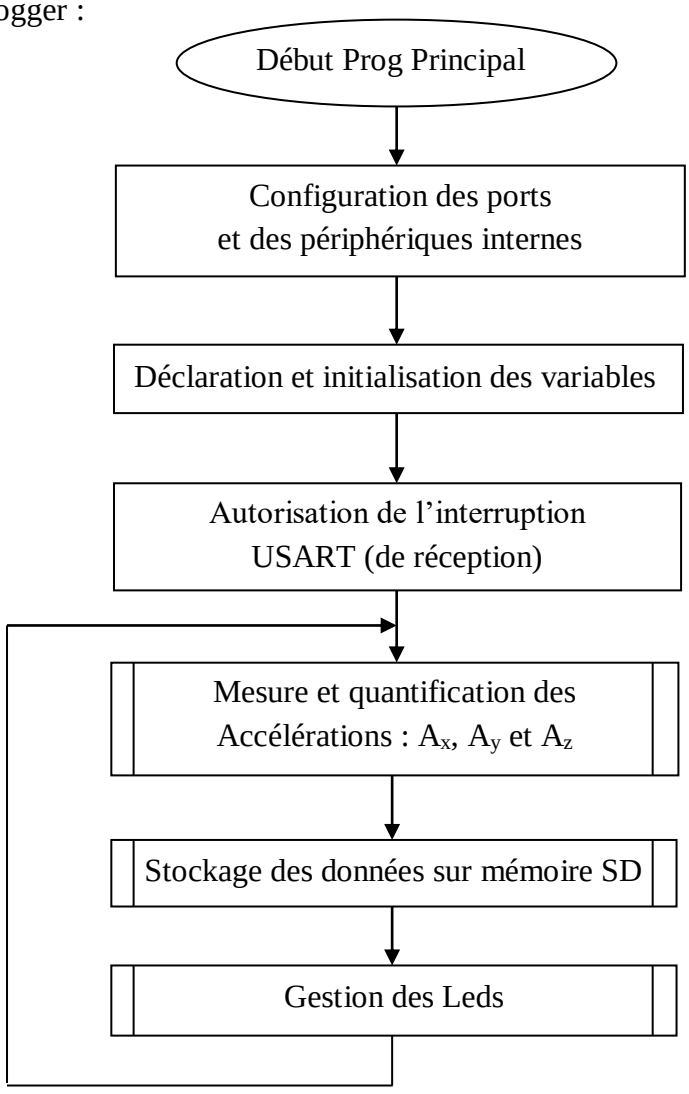


Figure n° I-6 : L'organigramme du programme principal

Le programme principal est constitué des éléments suivants :

- ***Un ensemble d'étapes*** : permettant au début la configuration des ports d'E/S et la configuration du type de signaux en entrées (analogique ou numérique), ainsi que la configuration des différents périphériques internes (Timers, CAN, USART, …), le paramétrage du convertisseur analogique/numérique intégré (tension de référence, résolution, ….). Par la suite le programme passe à la phase de création des variables nécessaires dans la mémoire RAM et leur initialisation. A la fin des étapes d'initialisation le programme active l'interruption de réception liée au module de communication série intégrée (USART) du Pic.
- ***Trois sous programmes*** : le premier permet l'acquisition des tensions analogiques issues des différents capteurs, la numérisation de ces derniers et par la suite la quantification des accélérations A_x, A_y et A_z qui permettront l'évaluation des angles d'inclinaison.

Le deuxième sous-programme a pour tâche le stockage des mesures sur une carte mémoire SD via une liaison de type SPI.

Le troisième sous-programme sert à la gestion de deux Leds (DEL) pour signaler divers informations nécessaires à l'exploitation du Data logger.

- ***Une routine d'interruption*** : qui permet de charger ou de modifier la configuration de l'acquisition des mesures (fréquence d'échantillonnage, nombre de canaux à convertir, résolution, …..).

IV-2- Les organigrammes des sous programmes :

A- *Le sous programme de mesures des accélérations :*

Ce sous-programme prend en charge la conversion analogique/numérique des signaux des différents accéléromètres associés au Data logger, il contient deux boucles, la première sert à scruter les différents entrées analogique à tour de rôle (au nombre de ***n*** selon le choix de l'utilisateur) puisque le Pic ne dispose que d'un seul ADC. La deuxième boucle permet de répéter la mesure (la numérisation) 10 fois, car on a remarqué dans la pratique que les signaux issus des accéléromètres présentent un niveau de bruit assez important malgré la présence de filtres. Ce qui permet de moyenner les valeurs mesurées pour réduire les bruits observés sur ces capteurs.

Par la suite la valeur moyenne est sauvegardée en RAM en attente de la stocker sur la carte mémoire SD par un sous-programme approprié.

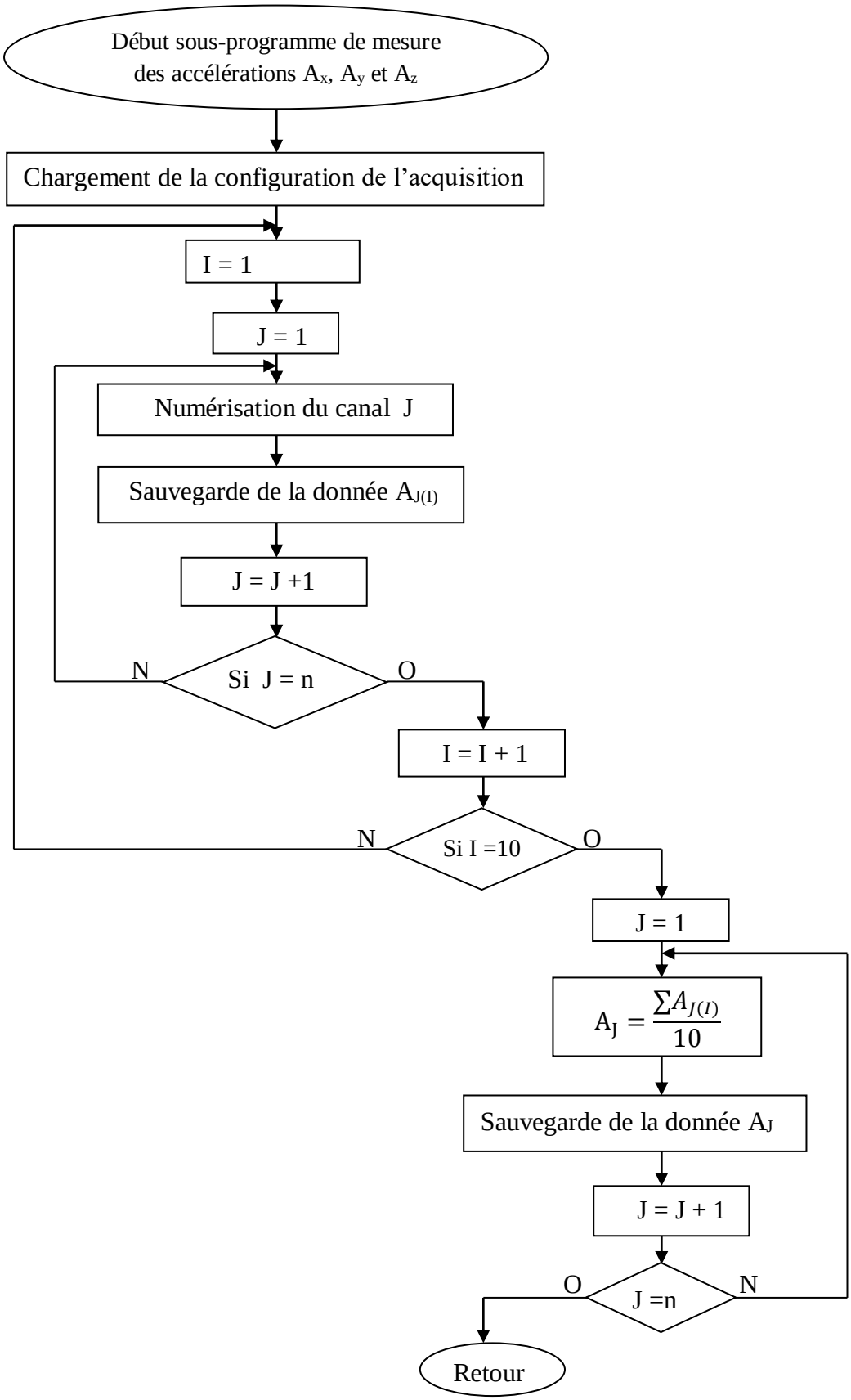


Figure n° I-7 : L'organigramme du sous programme de mesure des accélérations

B- *Le sous programme de stockage des données sur carte mémoire SD :*

Ce sous-programme permet de sauvegarder les valeurs moyennes des accélérations mesurées sur la carte mémoire SD, cette dernière est interfacée au Pic via une liaison série de type SPI, on commence par l'initialisation de cette liaison de la part du microcontrôleur pic qui assure le rôle du maître de la communication, puis on sélectionne la carte SD par une ligne appropriée, car le Pic peut communiquer avec plusieurs périphériques SPI (mais un seul à la fois).

L'étape suivante consiste à créer un fichier de mesures sur la carte SD destinée à recevoir les valeurs des accélérations mesurées des ***n*** capteurs pour une période de mesure donnée. Après chaque valeur mémorisée sur la carte mémoire, un compteur (NCM) est décrémenté pour gérer l'espace de stockage est afin de commander une Led (de couleur rouge) qui alerte l'utilisateur en cas de dépassement de la capacité mémoire.

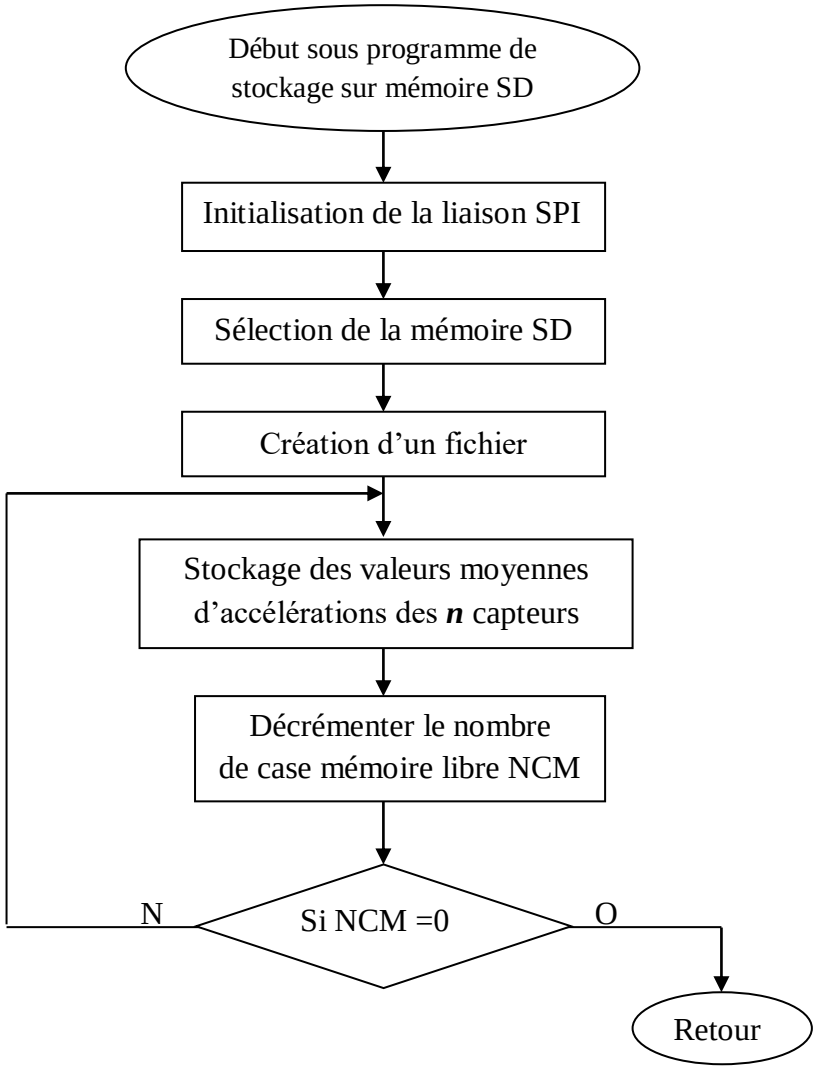


Figure n° I-8 : L'organigramme du sous programme de stockage sur carte mémoire SD

C- *Le sous programme de gestion des Leds :*

Le dernier sous-programme permet d'allumer une Led de couleur Rouge lorsque la carte mémoire SD se sature et de faire clignoter une Led de couleur jaune à une fréquence de 1 Hz lorsque le Data logger est en cour d'acquisition des accélérations.

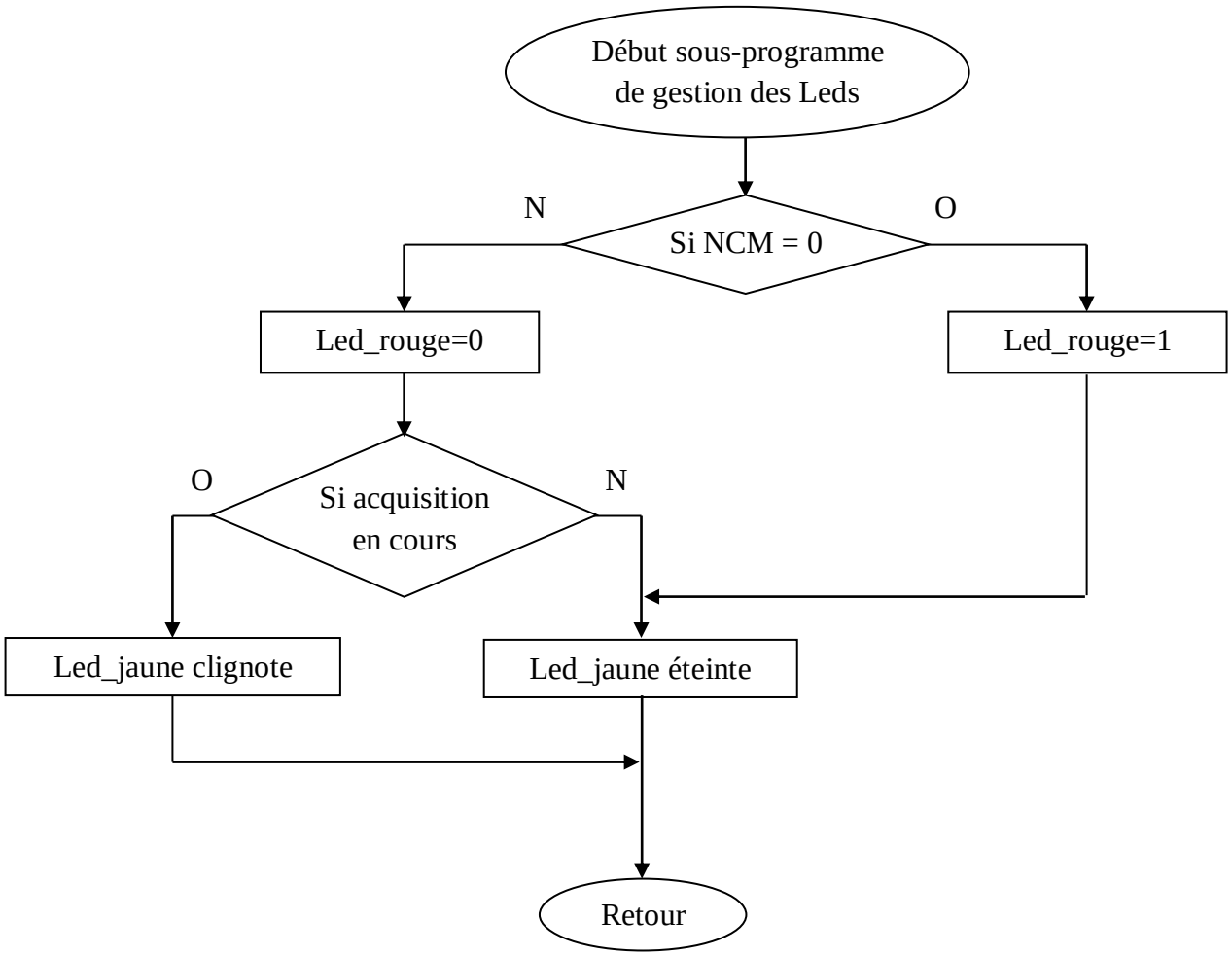


Figure n° I-9 : L'organigramme du sous programme de gestion des Leds

IV-3- La routine d'interruption :

Cette routine permet d'exécuter un petit programme lorsque le module de communication série USART du Pic génère une interruption suite à la réception d'octets via la liaison série. Cette dernière est utilisée pour charger par l'intermédiaire d'un PC à travers la liaison série RS232 les différents paramètres de configuration de l'acquisition des donnés (fréquence d'échantillonnage, résolution de l'ADC, nombre de voies à numériser, durée de mesure, …etc).
On commence par désactiver toutes les sources d'interruption pour qu'on puisse traiter qu'une seule à la fois, ensuite on vérifie le flag d'indication de l'interruption réception USART, si celui-ci est à 1 on charge la nouvelle configuration, par la suite on effectue un reset du flag et on active enfin toutes les sources d'interruption.

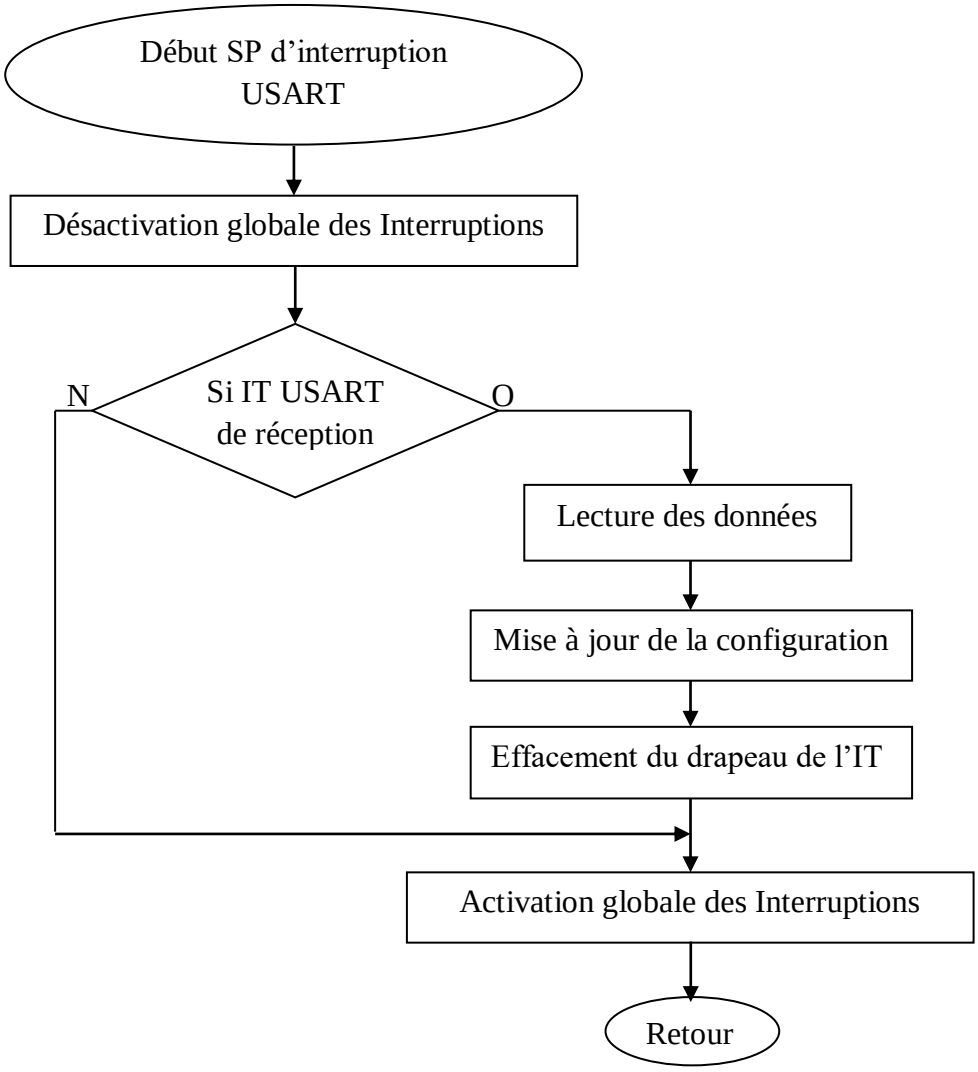


Figure n° I-10 : L'organigramme de la routine d'interruption

V. La nécessité d'un RTOS :

En examinant les différents organigrammes du soft embarqué, nous remarquons qu'il ya plusieurs boucles imbriquées les unes dans les autres, ainsi qu'un nombre important de tests nécessaires pour la prise de décisions. Ces deux types de procédures (boucles et tests) sont difficiles à maintenir et se compliquent de plus en plus lorsqu'on ajoute de nouvelles tâches, en plus ils peuvent engendrer des blocages du système si le code n'est pas optimisé et n'est pas bien testé.

Pour ces raisons, on a opté pour l'utilisation d'un système d'exploitation temps réel (RTOS) qui permettra d'adopter une approche multitâche destinée à gérer d'une manière optimale les différentes tâches du système et facilitera la création et l'ajout de nouvelle tâches au code, ainsi le code devient plus simple et plus court avec une approche de temps réel dans l'exécution des différentes parties du soft.

Chapitre II

Le RTOS PICos18

I. Les RTOS utilisés avec les microcontrôleurs Pic :

Comme nous l'avons déjà décrit dans le premier chapitre, le cœur du Data logger utilisé est un microcontrôleur Pic 18F452, le soft embarqué sur ce dernier suit l'approche séquentielle qui est largement utilisée dans la programmation classique.

Dans ce travail nous nous somme intéressés à un noyau RTOS (Real Time Operating System) qui permet de faire tourner plusieurs tâches de manières indépendantes. Le principe est de découper le code en grands groupes fonctionnels (taches) qui seront exécutés en parallèle. On peut par exemple imaginer dans le cas de notre projet la création des tâches suivantes :

- Gestion des Leds ;
- Gestion de la liaison série SPI avec La carte mémoire SD ;
- Gestion de la mesure des accélérations ;
- Gestion de la liaison série RS232 ; …

Chaque tâche se programme sous forme d'une boucle infinie ayant une structure du type :

```
while (1) {
        attente_evenement();
        actions() ;
        }
```

Il faut noter au passage, qu'il existe deux grandes familles de noyaux temps réels :

- Les RTOS préemptifs ;
- Les RTOS coopératifs.

Dans le mode **préemptif**, le *scheduler* (ordonnanceur) du noyau se charge de faire exécuter les tâches en leurs répartissant le temps de traitement, éventuellement en fonction des priorités que l'on peut affecter à chaque tâche. Une tâche dont l'exécution est longue va donc être interrompue en cours de traitement pour permettre à une autre tâche de se lancer. La tâche interrompue continuera son traitement la prochaine fois que le *scheduler* la réactivera.

Dans le mode **coopératif**, le *scheduler* n'agit pas de manière intrusive. Les tâches ne sont pas interrompues avant leur fin et doivent donc éviter de rester actives trop longtemps, ce qui bloquerait les autres tâches du système.

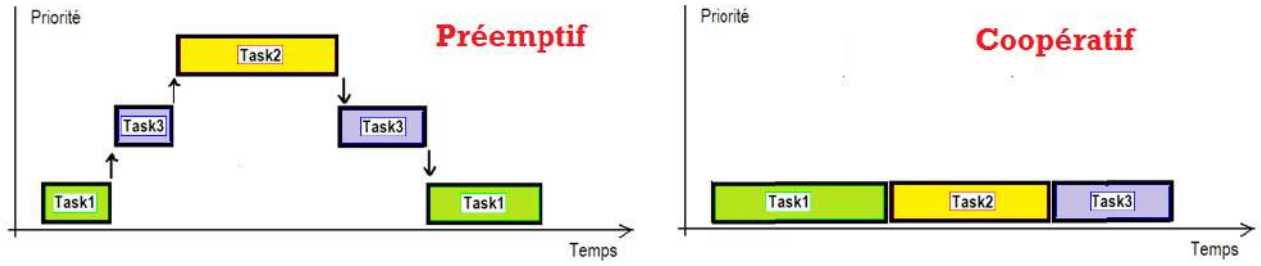


Figure n° II-1 : Le principe de gestion des Scheduler préemptifs et coopératifs

Les RTOS coopératifs sont plus contraignant pour la programmation car il faut s'assurer qu'une fonction ne prend pas trop de temps à s'exécuter et qu'elle "rend bien la main" au noyau de manière régulière afin d'éviter de pénaliser d'autres taches.

Dans la pratique, l'utilisation des RTOS est parfois assez simple à mettre en œuvre. La plus part du temps les tâches sont en attente d'un événement (un résultat de mesure, un message à afficher, une fin de temporisation, etc…), ce qui permet au *scheduler* d'activer d'autres tâches et donc de répartir les traitements. Si on sait qu'une boucle va prendre beaucoup de temps, il suffit d'intercaler des appels au *scheduler* pour activer les autres tâches.

Ce type de RTOS destiné aux systèmes embarqués est plus 'simpliste' est présente habituellement une taille de code pour le noyau le plus faible et une consommation mémoire plus réduite pour ses besoins internes.

Ainsi, nous avons cherché sur le web les noyaux temps réels qui sont dédiés à fonctionner sur les microcontrôleurs Pic18Fxxx. Cette recherche nous à fait retenir les RTOS suivants :

- ***Salvo RTOS*** de Pumpkin (http://www.pumpkininc.com/). C'est un RTOS préemptif d'un bon niveau possédant de nombreuses fonctions de messages et de synchronisations inter-tâches. C'est un RTOS payant et la version gratuite est limitée en nombre de tâches que l'on peut déclarer.

- ***OSA*** **(**http://pic24.ru/doku.php/en/osa/ref/intro**)**. C'est un RTOS de type coopératif qui n'utilise que peu de mémoire et ne rajoute que peu de code pour son noyau. Il possède des fonctions de messages et de synchronisations entre tâches. Il faut cependant bien lire la documentation pour savoir dans quel contexte les fonctions peuvent être appelées.
- ***PICos18*** (http://www.picos18.com/). C'est un RTOS préemptif écrit pour fonctionner sur des microcontrôleurs de type PIC 18Fxxxx. Bien que la version courante semble peu récente, il est intéressant d'une part vu ces caractéristiques acceptables, et d'une autre part vu la disponibilité du code source du RTOS au téléchargement gratuit.

Il faut noter qu'il existe d'autres RTOS plus élaborés (FreeRTOS, µC-OS-III de Micrium, etc…) qui s'adaptent à d'autres types de microcontrôleurs. Pour notre cas nous avons opté pour le RTOS PICos18 de la société Pragamatec pour sa simplicité, la disponibilité de son code source et la nature préemptif de son scheduler.

II. Présentation et description du noyau :

PICos18 est un produit de la société ***Pragmatec*** distribué gratuitement sous la licence GPL (General Public Licence). Celle-ci garantit la libre circulation des sources de ce dernier. PICos18 est un noyau temps réel basé sur la norme automobile OSEK/VDX et destiné aux microcontrôleurs PIC18Fxxx de la société Microchip Technologie Inc.[2]

II-1- La norme OSEK/VDX :

OSEK/VDX est une norme récente d'exécutif pour l'électronique embarquée dans les automobiles. Cette norme est le fruit des travaux de nombreux constructeurs automobiles et équipementiers européens depuis septembre 1995. OSEK a été étudié principalement par les équipes Allemandes et Françaises ; elle est le résultat commun de leurs travaux. L'un des objectifs visés par le consortium est de standardiser les interfaces du système d'exploitation afin de faciliter la fourniture de logiciels par de multiples sources, la portabilité, l'interopérabilité et la réutilisation. Différents travaux ont été menés dans le cadre d'OSEK/VDX :

- ✓ *OSEK/VDX OS* : les services de base du noyau du système d'exploitation.
- ✓ *OSEK/VDX COM* : les services pour la communication entre des nœuds d'un système réparti ou la communication locale.
- ✓ *OSEK/VDX NM (Network Management)* : les services de gestion et de surveillance du réseau.
- ✓ *OSEK/VDX OIL (OSEK Implémentation Langage)* : langage de description pour la mise en œuvre automatisée d'une application.[3]

II-2- La structure du noyau RTOS PICos18 :

Le noyau PICos18 est constitué des services suivants : (Figure n° II-2)



Figure n° II-2 : La structure du noyau PICos18

- **Le cœur du noyau** : composé de trois services (***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 de 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 les compteurs associés 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.[4]

II-3- L'organisation du code source de PICos18 :

Le code source du noyau temps réel PICOS18 est disponible sur le site de la société Pragmatec, il contient un ensemble de fichiers nécessaires pour la mise en ouvre de ce RTOS organisés dans 4 dossiers différents. La figure suivante montre l'organisation des fichiers du code source :

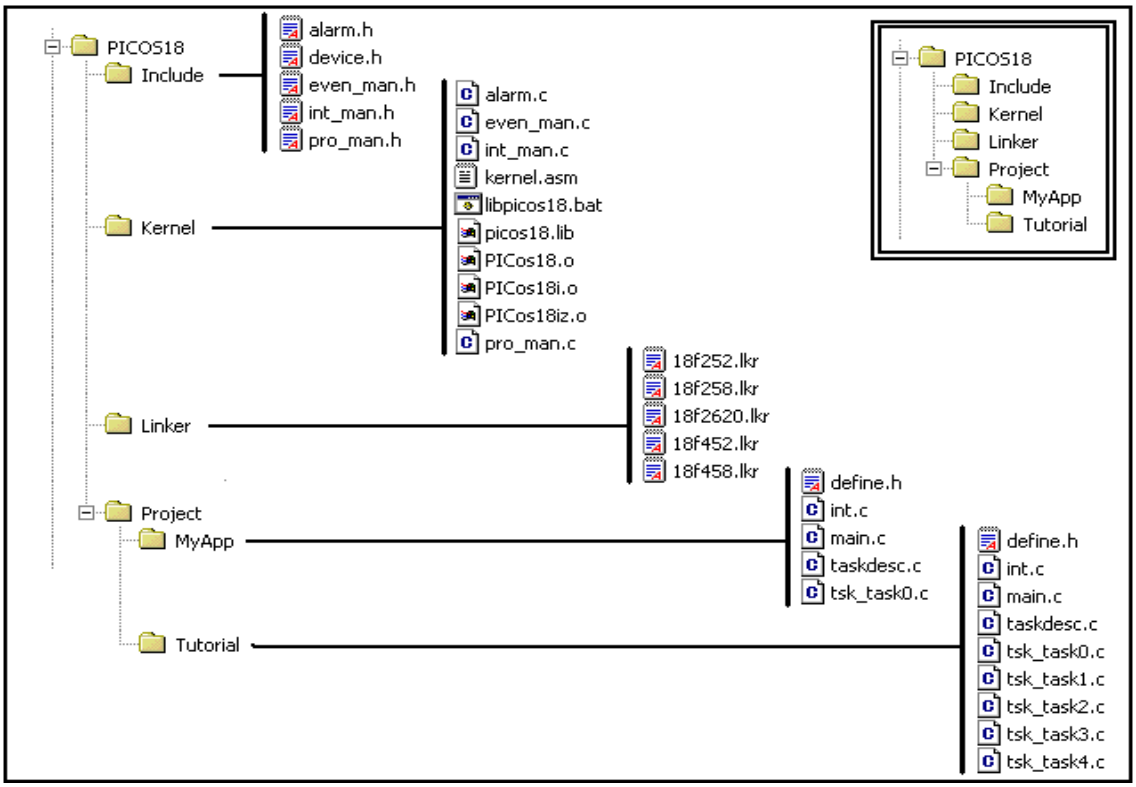


Figure n° II-3 : L'organisation du noyau PICos18

A- Le dossier 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 qui se trouvent dans les répertoires propres aux applications. 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.

B- Le dossier Kernel :

On trouve 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, qui 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 et on peut les modifier en fonction de nos propres besoins. Pour permettre d'utiliser PICos18 plus facilement, le noyau est ajouté sous forme de librairie : ***picos18.lib***. De plus les codes d'amorce du compilateur fournis par Microchip ne sont pas adaptés à PICos18, on va donc les adapter en conséquence. Ils sont fournis sous forme de fichier ***.o*** (comme ***PICos18iz.o***).

C- Le dossier Linker :

Lorsqu'on construit l'application avec PICos18, on écrit le code en langage C. Par la suite il convient de compiler l'application avec le compilateur Microchip ***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 éviter de gérer cette partie délicate de l'étape de compilation. Pour chaque nouveau Pic18 géré par PICos18, un nouveau script de linker se verra ajouter à ce répertoire au fil du temps.

D- Le répertoire Project :

Ce répertoire contient les fichiers nécessaires à la création de tout projet avec PICos18. Comme on pouvait 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 l'application (alarmes, compteurs, caractéristiques des tâches, ...), et un fichier ***C*** par tâche de l'application.[4]

II- 4- Les fichiers essentiels du code source de PICos18 :

a- Le Fichier define.h :

Ce fichier est utilisé pour identifier les événements ainsi que les tâches et leurs priorités :

```
/*********************************************************************
 * ---------------------------- 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
```

Ce fichier contient aussi les valeurs disponibles pour la fréquence interne du noyau :

```
/*********************************************************************
 * ----------------------- Timer settings ---------------------------
 *********************************************************************/
#define _32MHZ     57768
#define _40MHZ     55768
```

b- *Fichier tascdesc.c :*

Ce fichier contient la déclaration et la description des tâches :

```
/*********************************************************************
 * ---------------------------  task 0 ----------------------------
 *********************************************************************/
rom_desc_tsk rom_desc_task0 = {
  TASK0_PRIO,              /* priorité de 0 à 15              */
  stack0,                  /* empiler l'adresse (16 bits)     */
  TASK0,                   /* adresse de départ (16 bits)     */
  READY,                   /* Etat à la phase d'initialisation */
  TASK0_ID,                /* id_tache de 1 à 15              */
  sizeof(stack0)           /* la taille de pile    (16 bits)  */
};
```

- ***Priorité :*** c'est la priorité de la tâche, elle est comprise entre 1 et 15 inclus.
- ***Empiler l'adresse :*** c'est la pile logicielle de la tâche.
- ***Adresse de départ :*** c'est le nom de la tâche, ou le nom de la fonction d'entrée de tâche.
- ***Etat à la phase d'initialisation :*** c'est l'état initial de notre tâche.
- ***id_tache :*** l'identificateur de la tâche.
- ***La taille de pile :*** c'est la taille de la pile logicielle attribuée.

Dans PICos18, au respect de la norme OSEK, une tâche possède 4 états possibles : (Fig n° II-4)

- ***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 le processeur pendant un temps T.

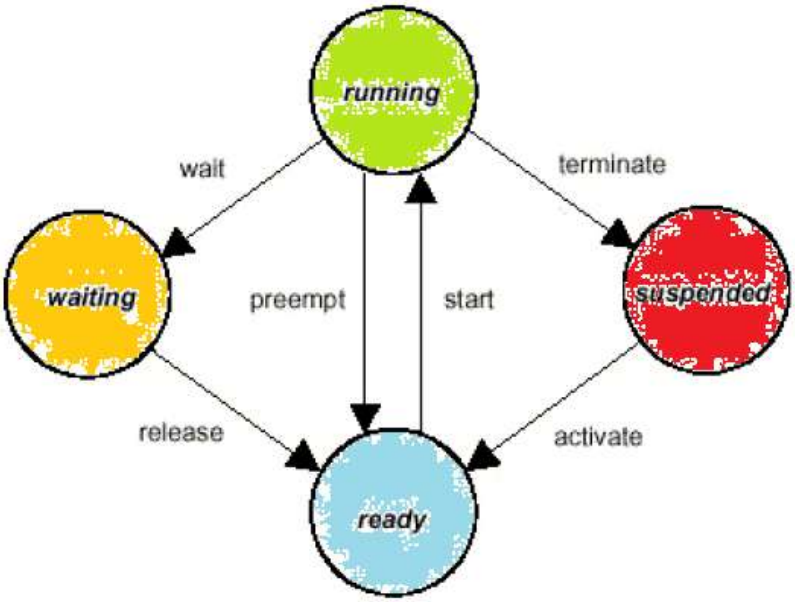


Figure n° II-4 : Les différents états des taches dans PICos18

Dans ce même fichier on trouve une liste des alarmes de l'application :

```
AlarmObject Alarm_list[] =
 {
  /*****************************************************************
   * ------------------------ First task -------------------------
   *****************************************************************/
  {
   OFF,                    /* état                */
   0,                      /* AlarmValue          */
   0,                      /* cycle               */
   &Counter_kernel,        /* ptr_compteur        */
   TASK0_ID,               /* tach_id_activé      */
   ALARM_EVENT,            /* evenement_poster    */
   0                       /* rappeler            */
  },
  };
```

Le tableau ***Alarm_list[]*** 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.

Une alarme est associée à un compteur (***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).

Le fichier ***taskdesc.c*** contient aussi une liste des ressources de l'application :

```
/*************************************************************************
 * ---------------------- COUNTER & ALARM DEFINITION -------------------
 *************************************************************************/
Resource Resource_list[] =
 {
  {
    10,                    /* priorité          */
     0,                    /* tache_priorité    */
     0,                    /*  verrou           */
  }
 };
```

Une ressource est un objet géré par le noyau, elle est généralement un périphérique partagé par plusieurs tâches, elle possède une priorité (***priorité***) et peut être verrouillée (***verrou***) par une tâche (***tache_priorité***).

En plus le fichier contient la liste des piles logicielles de l'application :

Une pile logicielle est une zone mémoire dynamique. Les variables locales (déclarées au sein de *fonctions C*) sont elles compilées par C18 de sorte de se retrouver stocker dans cette zone mémoire (la pile). 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 pour gagner beaucoup de place mémoire.

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.

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

volatile unsigned char stack0[DEFAULT_STACK_SIZE];
```

La taille de la pile (***DEFAULT_STACK_SIZE***) possède trois valeurs possibles, qui sont : 64, 128 et 256.

c- Le Fichier int.c :

Ce fichier contient un certains nombres de définitions, y compris la routine de gestion d'interruption.

```
#pragma  code _INTERRUPT_VECTORL = 0x003000
#pragma interruptlow InterruptVectorL save=section(".tmpdata"), PROD
void InterruptVectorL(void)
{
 EnterISR();
    if (INTCONbits.TMR0IF == 1)
    AddOneTick();
    /*****      ici la prochaine interruption       *****/
    /*if (INTCONbits.RBIF == 1)                      */
    /*  MyOwnISR();                                  */
 LeaveISR();
}
#pragma  code
```

Lorsque l'interruption du TIMER0 se déclenche (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 déclencher des tâches.

d- Le Fichier main.c :

Le fichier ***main.c*** nécessaire au compilateur, il 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 ce fichier on trouve une fonction ***Init()*** qui va nous permettre de paramétrer la fréquence de fonctionnement sans avoir besoin d'ouvrir la datasheet du PIC18 :

```
void Init(void)
{
 FSR0H = 0;
 FSR0L = 0;
 /* Réglage de l'utilisateur: la fréquence réelle du PIC */
 Tmr0.lt = _40MHZ;
 /* Timer OFF - Activée par le noyau */
 T0CON = 0x08;
 TMR0H = Tmr0.bt[1];
 TMR0L = Tmr0.bt[0];
}
```

Tmr0.lt : permet de régler la fréquence interne du PIC18, si on souhaite utiliser une fréquence différente, on va consulter les fréquences disponibles dans le fichier ***define.h***.

II-5- Les Drivers :

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 destinée 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, …etc. Par contre 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 ce dernier.

Dans PICos18, il existe plusieurs drivers parmi lesquels nous citons les drivers : I2C, CAN, RS232, SPI, …etc.

III. Les outils de développement associés au PICos18 :

III-1- Le compilateur C18 : il est fournit par la firme Microchip, il est caractérisé par :

- ✓ Compatible à la norme C ANSI ;
- ✓ Intégrable dans MPLAB pour faciliter la mise au point et la gestion d'un projet ;
- ✓ Génération de modules objet relogeables ;
- ✓ Compatible avec des modules objets générés par MP ASM ;
- ✓ Bibliothèque étendue incluant des modules de gestion de divers périphériques ; PWM, SPI, … ;
- ✓ Contrôle total par l'utilisateur de l'allocation des données et du code en mémoire.

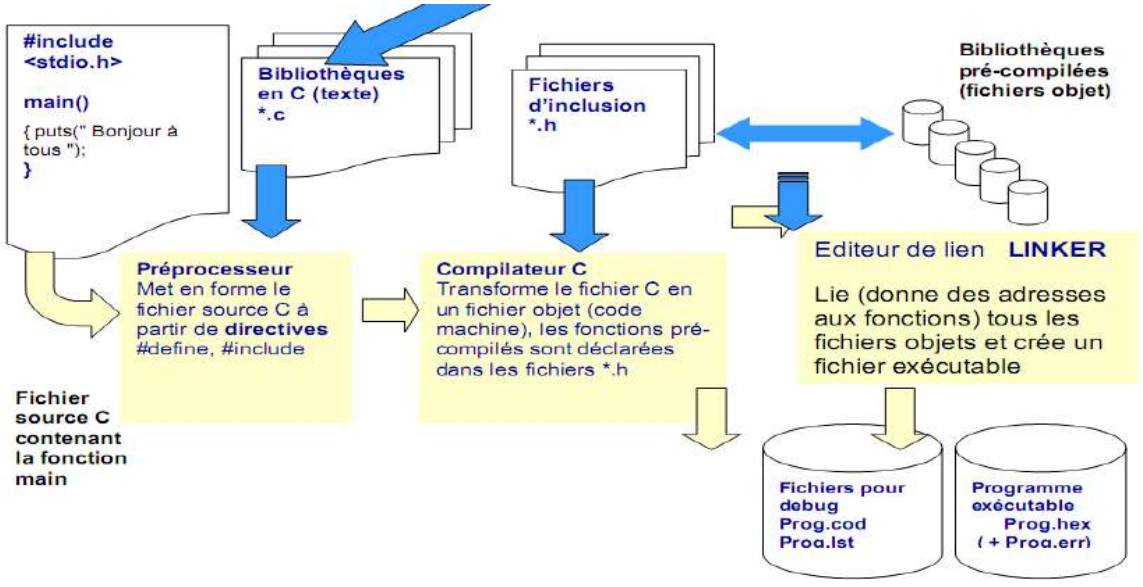


Figure n° II-5 : Le schéma général du processus de compilation sous C18

III-2- L'environnement de développement MPLAB :

MPLAB est un outil de développement spécifique aux microcontrôleurs PIC, conçu et fourni gratuitement par Microchip. Il permet d'écrire, de mettre au point et d'optimiser les programmes, sous Windows. En effet, en plus d'un éditeur et d'un assembleur, il inclut un ensemble d'outils permettant non seulement de fabriquer le code objet d'une application, mais aussi de simuler le programme, c'est à dire de le voir se dérouler à l'écran. Il supporte plusieurs outils externes, parmi lesquels deux modèles de programmateurs (PICSTART PLUS et PROMATE 2) et deux émulateurs (PICMASTER et PICMASTER-CE).

MPLAB est facile à utiliser et à prendre en main. Il offre beaucoup de flexibilité aux développeurs, notamment grâce aux nombreuses fenêtres pouvant être ouvertes à tout moment lors d'une mise au point, permettant de voir le contenu d'un quelconque emplacement mémoire et d'un quelconque registre.

IV. L'ordonnancement temps réels :

La fonction d'ordonnancement gère le partage du processeur entre les différents processus en attente pour s'exécuter, c'est-à-dire entre les différents processus qui sont dans l'état prêt. L'opération d'élection consiste à allouer le processeur à un processus. L'ordonnanceur (scheduler) est un programme système dont le rôle est d'allouer le processeur à un processus prêt.

On distingue trois types de tâches :

- ***Les tâches Périodiques :*** qui doivent être activées à intervalles réguliers ;
- ***Les tâches Sporadiques :*** qui sont irrégulières, mais il existe un temps minimal entre deux activations ;
- ***Les tâches Apériodiques :*** ces dernières sont irrégulières.

Le modèle canonique des tâches temps-réel est représenté par la figure n° II-6:

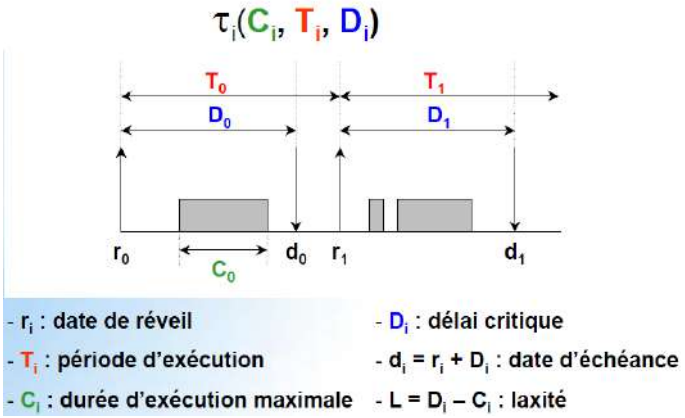


Figure n° II-6 : Le modèle canonique d'une tache temps réel

IV-1- Les Algorithmes d'ordonnancement :

IV-1-1 Les différentes catégories d'algorithmes : On peut classer les différents algorithmes d'ordonnancement en deux catégories de base:

- ***Les algorithmes Off line :*** les décisions d'ordonnancement sont prises avant l'exécution du système, c'est des algorithmes statiques ;
- ***Les algorithmes On line :*** basés sur la réévaluation en ligne de l'ordonnancement en fonction de la modification des critères de choix (prise en compte d'évènements non prévus, …etc), c'est des algorithmes dynamiques.

IV-1-2- Les types d'algorithmes : On distingue en général plusieurs types d'algorithmes :

- Les algorithmes statiques pilotés par table ;
- Les algorithmes statiques préemptifs basés sur les priorités ;
- Les algorithmes dynamiques avec planification à l'exécution ;

A- Statiques pilotés par table:

C'est un algorithme hors ligne, l'application est découpée en séquences élémentaires qui ne seront jamais interrompues (une Séquence est une unité de base ou procédure définie par l'utilisateur).L'ordonnancement des processus (suite ordonnée de séquences) est régi par un calendrier (table) spécifiant la liste des processus à activer.

C'est un ordonnancement cyclique qui n'utilise pas de préemption, les processus sont indépendants les uns des autres et l'ordonnancement des processus est régi par une horloge.

Le découpage en séquences non interruptibles assure la protection des données partagées entre les processus et il est aisé de certifier le comportement de l'application.

B- Statiques préemptifs basés sur les priorités:

C'est un algorithme hors ligne, qui permet d'assigner les priorités aux tâches (au début), une fois en cours d'exécution, le système utilise un ordonnancement préemptif basée sur la notion de priorité fixée apriori.

C- Dynamique avec planification à l'exécution:

C'est un algorithme On line, basé sur une analyse en ligne, le choix de l'ordonnancement et réalisé en fonction de critères d'analyse tels que :

- ***HPF :*** Highest Priority First (la priorité la plus grande) ;
- ***EDF :*** Earliest Deadline First (l'échéance la plus proche) ;
- ***LLF :*** Least Laxity First (la marge la plus courte, EDF + Notion de durée de travail).[5]

NB : La prise de décision est effectuée quand le système est actif et les critères d'analyse peuvent varier durant l'exécution du programme.

IV-3- Les algorithmes d'ordonnancement des taches périodiques :

IV-3-1-L'analyse statique :

A- Analyse Rate Monotonic (RM) : (Liu et Layland 1973)

C'est un algorithme hors ligne; basé sur la notion de priorité statiques, l'affectation de la priorité se fait en fonction de la fréquence des tâches: plus la fréquence est forte, plus la priorité est élevée : ($P_i = 1/T_i$)

Il fournit un ordonnancement optimal si le concept HPF est utilisé, cet algorithme doit satisfaire les critères suivants :

– Tâches périodiques ;

– Échéance sur période ($D=T$) ;

– Préemption possible avec un temps de préemption négligeable ;

– Temps d'ordonnancement négligeable ;

– Tâches indépendantes et la durée maximale de traitement C doit être connue.[6]

Condition suffisante :

Pour toute tâche i ($1 \leq i \leq n$), avec C_i connu, et $T_i = D_i$, on doit vérifier la condition suivante :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\,(\sqrt[n]{2} - 1) \qquad (1)$$

- La partie de gauche représente le taux d'utilisation du processeur par les ***n*** tâches ;
- La partie droite représente le temps CPU allouable au ***max***, en fonction du nombre de tâches ***n***.

Ce temps max allouable est donné par la figure suivante pour ***n*** allant de 1 à 10 :

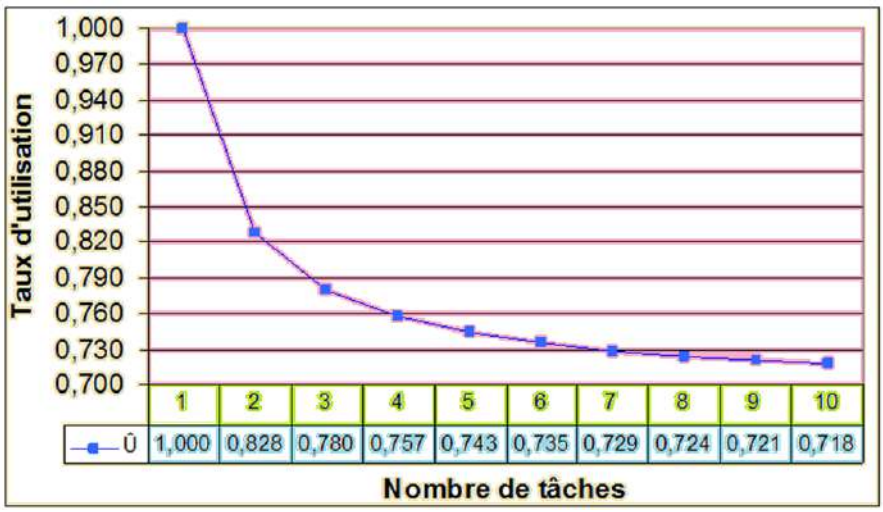


Figure n° II-7 : Le temps max d'utilisation du processeur pour l'analyse RM

Exemple : soient les 03 tâches : τ_i (C_i, T_i, D_i) : τ_1 (3, 20, 20) ;τ_2 (2, 10, 10) et τ_3 (2, 5, 5).

L'utilisation de l'algorithme d'ordonnancement RM donne :

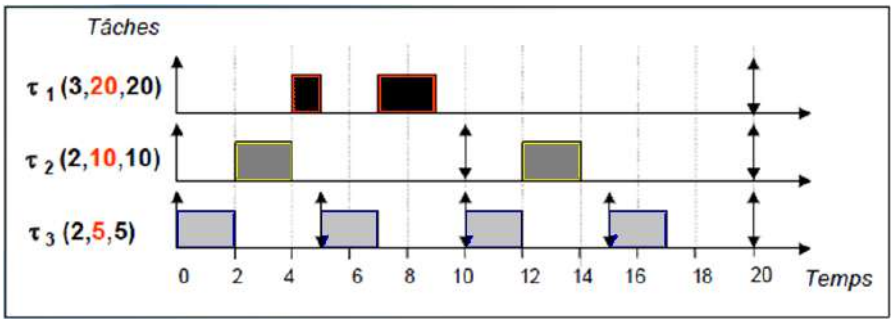


Figure n° II-8 : Le chronogramme d'ordonnancement RM

Les trois tâches respectent leur échéance temporelle et la condition de faisabilité est vérifiée :

$$3/20 + 2/10 + 2/5 \leq 3\ (\sqrt[3]{2} - 1)$$

$$0.75 \leq 0.77 \text{ (test respecté)}$$

NB : L'algorithme RM est optimal dans la classe des algorithmes à priorités fixes pour des tâches périodiques indépendantes à échéances sur requêtes ($\mathbf{D_i = T_i}$).

B- Analyse Deadline Monotonic (DM) :(Leung & Whitehead 1985)

Cet algorithme est dérivé du précédent (RM), il est utilisé avec les tâches ayant une échéance inférieure au égale à la période ($D_i < T_i$) c'est-à-dire à échéance avant requête. Il est caractérisé par l'utilisation des priorités fixes qui sont affectées par ordre inverse des échéances des tâches. Ces priorités sont fixées d'une manière statique avec application du principe HPF.[6]

Condition suffisante : Pour toute tâche i ($1 \leq i \leq n$) on doit vérifier la condition suivante :

$$U_i = \sum_{j=1}^{i} \frac{C_j}{D_j} \leq i\ (\sqrt[i]{2} - 1) \qquad (2)$$

Exemple : soient les trois tâches suivantes :τ_1 (3, 20, 9) ;τ_2 (2, 10, 10) et τ_3 (2, 5, 5).

L'ordonnancement par l'algorithme DM est schématisé par la figure II-9 :

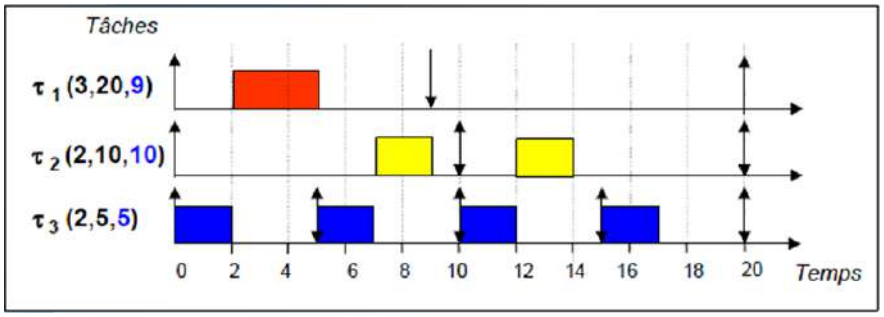


Figure n° II-9 : Le chronogramme d'ordonnancement DM

NB : L'algorithme DM est optimal dans la classe des algorithmes à priorités fixes pour des taches périodiques indépendantes à échéances sur requêtes ($\mathbf{D_i \leq T_i}$).

Théorème de la zone critique :

Les algorithmes présentés plus haut sont basés sur la considération du cas pire, c'est-à-dire, ils évaluent l'ordonnançabilité d'un ensemble de tâches périodiques à un instant critique, l'***instant critique*** étant un instant où le temps de réponse de l'ordonnanceur à la demande d'exécution de toutes les tâches est le plus long. L'instant critique d'une tâche est l'instant où la demande d'exécution d'une tâche arrive simultanément avec la demande d'exécution de toutes les tâches de priorité supérieure. Le cas pire est donc considéré si on suppose que les déphasages entre les tâches sont nulles ; il suffit alors de vérifier si la première demande de chaque tâche est exécutée avant son échéance.

Ainsi on considère un temps de simulation donnée. On réveille toutes les tâches périodiques en même temps à l'instant 0 (au pire des cas). Si toutes les tâches sont ordonnancées par HPF sur le temps de simulation sans qu'aucune échéance ne soit manquée, alors le test est considéré comme validé.

C- Avantages et inconvénients de l'ordonnancement à priorités fixes :

- **Intérêts :**

  - Mécanisme simple ;

  - S'implante naturellement dans les OS du marché.

- **Inconvénients :**

  - Hypothèses restrictives ($T_i \leq D_i$, taches périodiques indépendantes, …).

  - Indépendance des tâches impérative pour l'utilisation des conditions de faisabilité.

  - Borne supérieure pour le facteur d'utilisation du processeur.

IV-3-2-L'analyse Dynamique :

A-Analyse EDF : (Liu et Layland 1973)

Cet algorithme est basé sur la variation des priorités des tâches lorsque le système est en fonctionnement, la priorité la plus élevée est affectée à la tâche qui possède l'échéance la plus proche, en cas d'égalité : la tâche en cours reste en exécution.

Condition de faisabilité :

- Si $D_i = T_i$: Condition nécessaire et suffisante :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \qquad (3)$$

- Si $D_i \leq T_i$: Condition suffisante :

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \qquad (4)$$

Exemple : soient les trois taches suivantes : τ_1 (1, 20, 8) ;τ_2 (4, 10, 10) et τ_3 (2, 5, 4).

L'ordonnancement par l'algorithme EDF est schématisé par la figure n° II-10:

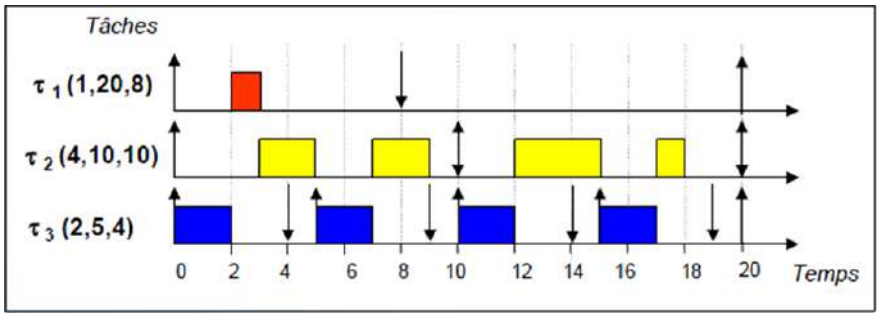


Figure n° II-10 : Le chronogramme d'ordonnancement EDF

NB : L'algorithme EDF est optimal dans la classe des algorithmes préemptifs pour des tâches périodiques indépendantes telles que $D_i \leq T_i$.

B- Analyse LLF : (Sorenson 1974)

Il est basé sur le principe suivant : à chaque instant, la tâche la plus prioritaire est celle qui possède la marge la plus faible, c'est-à-dire dont la laxité : $L(t) = r_i + D_i - (t + C_i(t))$ est la plus petite, c'est un ordonnancement dynamique (variation des priorités des tâches lorsque le système est en fonctionnement).

D'une autre manière : la Marge = échéance – temps de calcul restant.

Condition de faisabilité :

- Si $D_i = T_i$: Condition nécessaire et suffisante :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \qquad (5)$$

- Si $D_i \leq T_i$: Condition suffisante :

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \qquad (6)$$

Exemple : soient les trois tâches suivantes : τ_1 (1, 20, 8) ; τ_2 (4, 10, 10) et τ_3 (2, 5, 4).

L'ordonnancement par l'algorithme LLF est schématisé par la figure n° II-11:

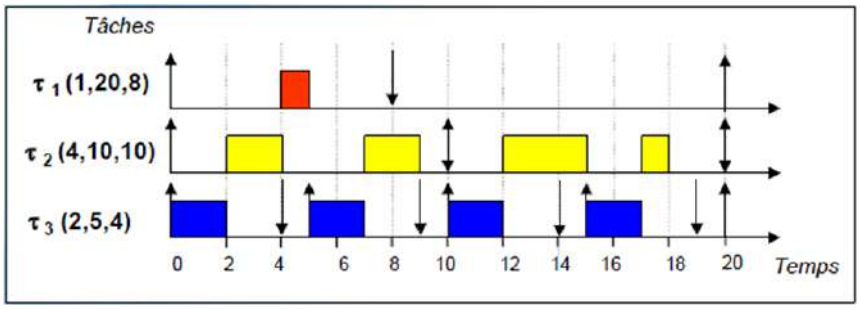


Figure n° II-11 : Le chronogramme d'ordonnancement LLF

NB : L'algorithme LLF est optimal dans la classe des algorithmes préemptifs pour des tâches périodiques indépendantes telles que $D_i \leq T_i$.

C-Avantages et inconvénients de l'ordonnancement à priorités dynamiques :

- **Intérêts :**
  - Optimisation de l'usage des ressources ;
  - Bien adapté aux tâches périodiques à courtes échéances.
- **Inconvénients :**
  - Indépendance des tâches impératives pour l'utilisation des conditions de faisabilité ;
  - Nombreux changements de contexte dans certains cas (LLF) ;
  - Difficilement implantable dans les OS actuels.

Chapitre III

La réalisation de l'application multitâche

I. Préparation de l'environnement de développement :

- Au départ nous avons commencé par télécharger les éléments Soft suivants :
  a- L'environnement de développement MPLAB, nous avons utilisé ***MPLAB IDE v7.31*** ;
  b- Le compilateur C de Microchip ***C18***, nous avons utilisé la version ***v3.40*** ;
  c- Le code source du RTOS ***PICos18***, nous avons utilisé la version ***v2.10***.
- Par la suite, nous avons décompressé l'ensemble des sources directement sous **C:**. Ainsi le répertoire ***PICOS18*** s'est logé bel et bien sous **C:**. Après nous avons installé MPLAB et le compilateur C18 en version gratuite. La version gratuite correspond à une version sans gestion des optimisations par le compilateur, or d'après les développeurs de Pragmatec il ne faut surtout pas utiliser les optimisations de C18 avec PICos18, certaines n'étant pas adaptés dans le cadre d'une utilisation multitâches. Sous MPLAB, nous avons paramétré l'environnement de développement sur ***Microchip C18 Toolsuite***.
- Enfin nous avons créé sous MPLAB un projet intitulé ***App_Mesure***.

II. Création du projet *App_Mesure* :

- Nous avons commencé par la configuration des options du projet créé, sur la boite de dialogue ***Projet/Built Options*** nous avons précisé les chemins d'inclusion et celui des librairies du compilateur comme du noyau. Et par la suite on a coché l'option "***Use alternate settings***" sur l'onglet "***MPLAB C18***" et on a ajouté l'option "***-On-***" dans la ligne de commande. Ceci a pour effet d'éviter l'optimisation des sélections de bank, non fonctionnelle dans le cas de PICos18.
- Puis nous avons ajouté les fichiers suivants : ***int.c***, ***main.c***, ***taskdesc.c*** et ***define.h*** à partir du répertoire Tutorial. De même pour le fichier ***18f452.lkr*** qui se trouve dans le répertoire Linker.

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

Voici à quoi correspond la structure du dossier PICOS18 :

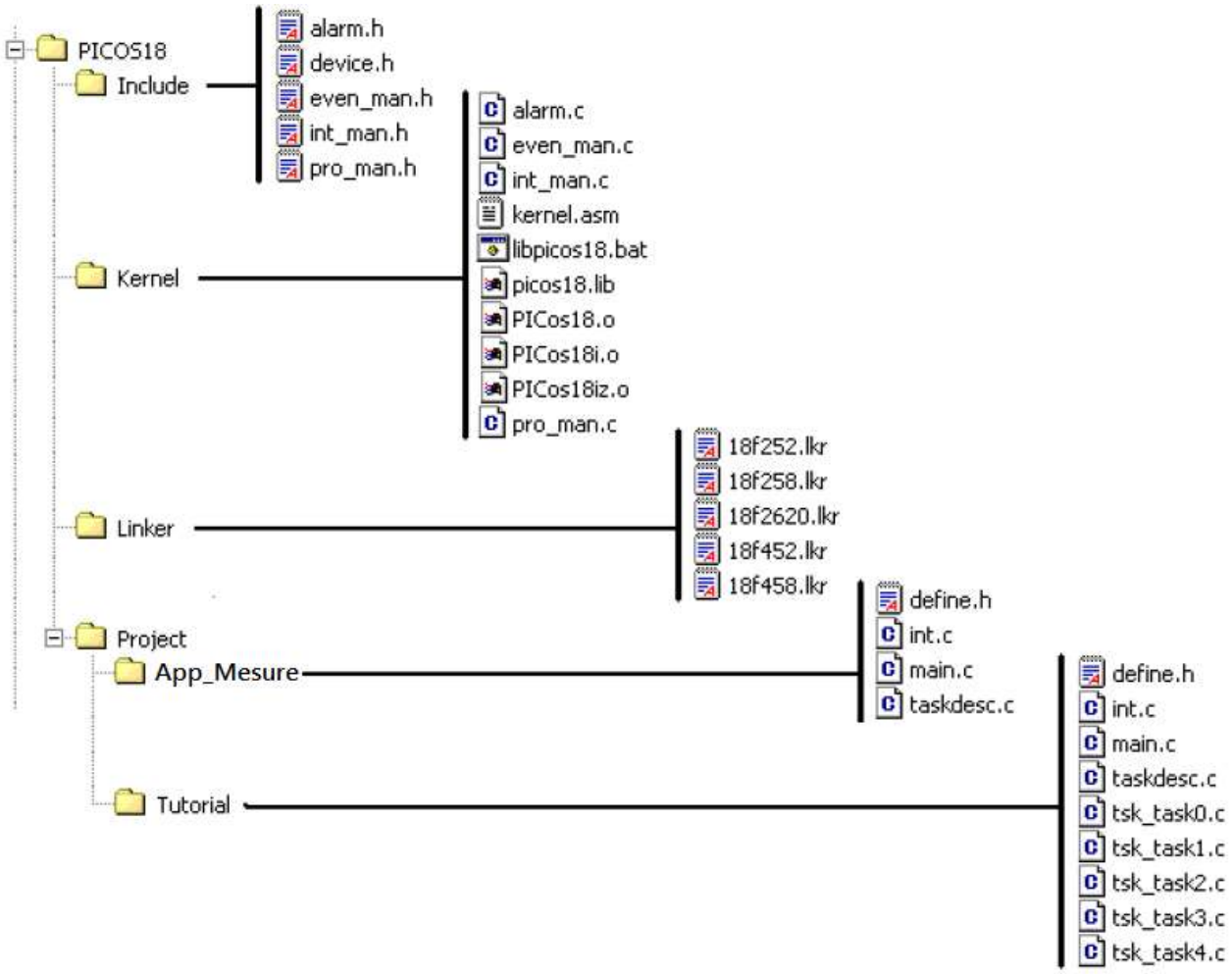


Figure n° III-1 : La structure du dossier PICOS18

III. Les des différentes tâches du projet *App_Mesure* :

Nous avons opté pour l'organisation du programme embarqué sur le Data logger en trois tâches et une routine d'interruption.

Ces tâches sont :

1- La tâche_0 qui assure la mesure de différentes accélérations.

2- La tâche_1 qui assure le stockage des mesures dans la carte mémoire SD.

3- La tâche_2 qui assure gestion des Leds d'indications.

4- Une routine d'interruption pour traiter l'événement de réception de données à travers la liaison série RS232.

Le PICos18 est un RTOS préemptif, nous avons choisi d'utiliser une stratégie d'ordonnancement ***Offline*** (statique) qui s'avère plus simple à mettre en œuvre et plus facile à gérer par ce RTOS, contrairement à l'ordonnancement ***Online*** (dynamique). Ainsi nous avons choisi d'utiliser l'algorithme classique ***RM*** (Rate Monotonic) qui affecte une priorité fixe à chaque tâche avec un niveau de priorité proportionnel à la fréquence de cette dernière.

III-1- La tache *tsk_task0.c* :

- ***Rôle :***

a- Convertir la tension analogique issue d'un accéléromètre donné en valeur digitale, puis traduire cette valeur en accélération (en ***g***, unité de mesure des accélérations, 1 g = 9.8 m/s^2).

b- La valeur de chaque entrée analogique est stockée temporairement dans un vecteur pour pouvoir la moyenner par la suite afin de réduire les bruits.

c- Si aucune alarme ne s'active, alors le programme peut basculer à une autre entrée analogique.

d- Poster l'alarme ***Ready_Event*** vers la tâche ***tsk_task1*** pour lui indiquer la disponibilité d'une mesure à sauvegarder.

e- Poster l'alarme ***End_Event*** vers la tâche ***tsk_task2*** pour lui indiquer la fin d'acquisition.

- ***Priorité :*** la tâche a une priorité de « 10 », car c'est une tâche importante est qui a une grande fréquence.
- ***Mode :*** Exécutée toutes les ***n*** ms (n est fixé par la configuration de la mesure).
- ***Etat lors de l'initialisation :*** l'état de cette tâche est READY (prête à être activée).

III-2- La tache *tsk_task1.c* :

- ***Rôle :***

a- Initialisation de la liaison SPI et sélection de la Carte mémoire SD.

b- Création d'un fichier de mesure.

c- Sauvegarde de la mesure.

d- Décrémenter le compteur ***NCM*** (nombre de cases mémoires libres).

e- Poster une alerte ***Sat_Event*** (pour indiquer la saturation de la carte mémoire SD lorsque NCM = 0).

f- Sauvegarder la mesure suivante s'il n'y a pas d'alerte ***Sat_Event***.

- ***Priorité :*** la tâche a une priorité de « 8 », c'est une tâche d'une grande fréquence moyenne.
- ***Mode :*** Attente sur un événement d'une moyenne de 10 x ***n*** ms.
- ***Etat lors de l'initialisation :*** l'état de cette tâche est SUSPENDED (la tâche n'est pas prise en compte que lorsque l'événement ***Ready_Event*** est reçu de la part de ***tsk_task0***).

III-3- La tache *tsk_task2.c* :

- ***Rôle :***

a- Eteindre la Led jaune lorsque le Data logger n'est pas en cours d'acquisition de mesures, c'est à dire lorsque la tâche reçoit une alerte ***End_Event*** de la part de la tâche ***tsk_task0***.

b- Allumer la Led rouge lorsque la tâche reçoit une alerte ***Sat_Event*** de la part de la tâche ***tsk_task1***.

- ***Priorité :*** la tâche a une priorité de « 2 », c'est une tâche de faible priorité car elle n'est activée que lorsqu'elle reçoit une alerte des tâches : ***tsk_task0*** et ***tsk_task1***.
- ***Mode :*** Attente sur un événement.
- ***Etat lors de l'initialisation:*** l'état de cette tâche est SUSPENDED (la tâche n'est pas prise en compte que lorsque les événements : ***End_Event*** et ***Sat_Event*** sont reçus de la part des tâches : ***tsk_task0*** et ***tsk_task1***).

III-4- La fichier *int.c* :

Afin de pouvoir exploiter au maximum le microcontrôleur PIC18, il est important de pouvoir utiliser les périphériques intégrés. De ce point de vue, les interruptions sont essentielles et leurs utilisations nécessitent l'écriture des routines d'interruption.

Dans notre projet on s'intéresse à l'interruption du module USART en mode réception, tout d'abord nous allons décrire le fichier "***int.c***", ce 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). Or le déterminisme du PICos18 a était évalué de 50µs (ce qu'on appelle le temps de latence), donc il existe une zone d'ombre, un blackout de 50µs pendant laquelle une interruption ne sera pas détectée.

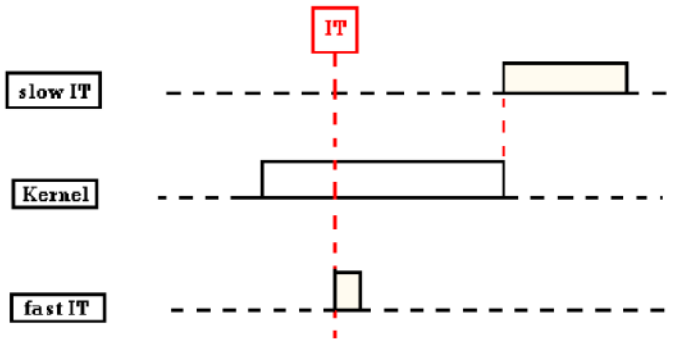


Figure n° III-2 : Les deux types d'interruptions des Pic18

Pour éviter ce problème on utilise les interruptions rapides ("***fast interrupts***") du PIC18, qui utilisent un mécanisme hardware pour la sauvegarde des registres élémentaires (comme WREG, STATUS, ...). Malheureusement 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 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 (PIR3bits.RXB1IF == 1)
```

```
/* Usart reception ISR(); */
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
```

Une première lecture du fichier ***int.c*** peut paraître bien compliqué, mais il est en fait fort simple. Les 2 seules zones qui peuvent être soumises à des modifications sont les fonctions ***InterruptVectorL()*** et ***InterruptVectorH().*** Comme dans 99% des cas, l'interruption lente est suffisante pour les applications développées, nous n'avons modifié que la fonction ***InterruptVectorL*()**.

Comme 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 dé validé, nous avons utilisé INT1 en la reprogrammant en mode "slow interrupt". Il suffit à présent d'ajouter la fonction "***MyOwnISR***" dans n'importe quel fichier C, de préférence une des 3 tâches de fond qui ont été créées précédemment.

Il faut noter qu'on ne doit pas oublier de déclarer la fonction ***MyOwnISR*** au début du fichier, comme le montre la partie du code suivant :

```
#include "define.h"
void MyOwnISR(void);
/*********************************************************************
* Definition dedicated to the local functions.
*********************************************************************/
```

III-5- Création des différentes taches :

Sous MPLAB, nous avons commencé par enregistrer le premier fichier "***tsk_task0.c***" dans ce dernier nous avons édité le programme de traitement de cette tâche en ***langage C***, par la suite les deux tâches "***tsk_task1.c***" et "***tsk_task2.c***" sont créées à partir du fichier de la première tâche et leurs contenue est modifié suivant le rôle de chaque tâche.

Au début nous avons fixé par la partie du code suivante la taille de notre pile à une valeur moyenne de 128 octets grâce au fichier ***taskdesc.c*** pour les trois tâches créées.

```
#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];
```

Par la suite nous avons modifié le fichier ***taskdesc.c*** pour prendre en considération les tâches créées précédemment.

```
/*************************************************************************
* ---------------------- 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) */
SUSPENDED,             /* state at init phase */
TASK1_ID,              /* id_tsk from 1 to 15 */
sizeof(stack1)         /* stack size (16 bits) */
};
/*************************************************************************
* ------------------------------ task 2 ------------------------------
*************************************************************************/
rom_desc_tsk rom_desc_task2 = {
TASK2_PRIO,            /* prioinit from 0 to 15 */
Stack2,                /* stack address (16 bits) */
TASK1,                 /* start address (16 bits) */
SUSPENDED,             /* state at init phase */
TASK2_ID,              /* id_tsk from 1 to 15 */
sizeof(stack2)         /* stack size (16 bits) */
};
```

Dans la partie suivante du fichier header : ***define.h***, nous avons défini les identificateurs des tâches et les priorités assignées précédemment :

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

#define TASK0_PRIO            10
#define TASK1_PRIO            8
#define TASK2_PRIO            2

#endif /* _DEFINE_H */
/* End of File : define.h */
```

Dans le même fichier header : ***define.h***, nous avons défini les différentes alarmes utilisées dans la programmation :

```
/************************************************************************
 * ---------------------------- Events ------------------------------
 ************************************************************************/
#define Ready_EVENT       0x02
#define End_EVENT         0x04
#define Sat_EVENT         0x06
#define Usart_EVENT       0x08
```

IV. Les drivers :

Le RTOS PICos18 est livré avec le noyau seul, mais sur la page DOWNLOAD du site web de la société Pragmatec, nous trouvons aussi des drivers ou des librairies disponibles gratuitement. Le but de ces dernies est de permettre un développement modulaire avec lequel il suffirait d'intégrer des briques logicielles pour réaliser une application.

Nous avons exploité deux drivers qui sont RS232 et SPI. Nous avons commencé par télécharger ces deux drivers, le package de chacun des deux possède un fichier texte (***.TXT***) qui indique comment inclure le driver dans l'application à développer, comment paramétrer le driver (baudrate dans le cas d'un driver RS232) et aussi comment le mettre en œuvre.

Dans ce qui suit nous décrirons l'utilisation du driver RS232, ce dernier est utilisé dans la routine d'interruption, on doit le déclarer au début du code :

```
#include "define.h"
#include "drv_rs.h"
#include <stdio.h>
#include <string.h>
#define Usart_Event
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];
/************************************************************************
* ---------------------------- MyOwnISR ----------------------------
************************************************************************/
```

```
{
SetRelAlarm(Usart_Event, 1000, 1000);
while(1)
{
WaitEvent(Usart_EVENT);
ClearEvent(Usart_EVENT);
Printf("%02d : %02d : %02d\r", (int)SFreq, (int)Resolut, (int)Time);
}
}
/*********************************************************************
* 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_enqMsg(&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. Ensuite il faut déclarer Usart_Event et aussi déclarer comme "extern" les variables externes "***SFreq***" : la fréquence d'échantillonnage en Hz, "***Resolut***" : la résolution de l'ADC (8 ou 10) et "***Time***" m temps d'acquisition en mn, pour indiquer 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 de réception.

Paramétrage de l'utilitaire Hyper terminal :

Afin d'assurer une communication entre le PC et le Data logger dans de bonnes conditions on doit faire attention au paramétrage du port COM série du PC sous HyperTerminal, elle doit être à 11500 bits/sec et sans contrôle de flux, comme le montre la figure n° III-3 :



Figure n° III-3 : La configuration du port COM sous HyperTerminal

L'envoie des données de configurations de l'acquisition sous HyperTerminal par l'utilisateur est simple à réaliser comme le montre la figure n° III-4 :

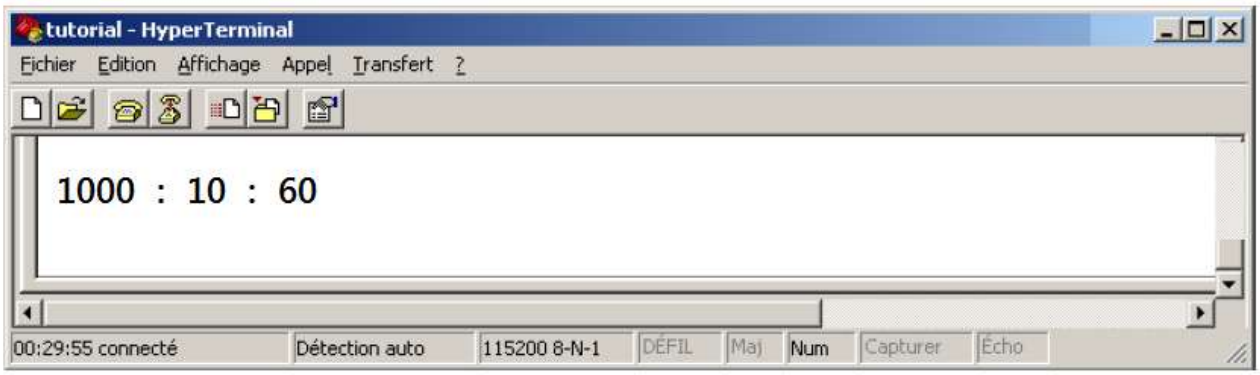


Figure n° III-4 : Un exemple de la configuration des paramètres d'acquisition

- La valeur 1000 indique la fréquence d'échantillonnage en Hz (donc pour cet exemple F = 1 kHz).
- La valeur 10 indique une conversion sur 10 bits.
- La valeur 60 indique une acquisition sur 60 mn (1heure).

L'avant dernière phase de la réalisation du programme consiste à compiler le projet entier ; et à corriger les erreurs indiquées par MPLAB.

Finalement nous avons lancé la simulation en cliquant sur "Debugger/Select Tool/MPLAB SIM", puis sur le bouton F9 (Run). Il faut attendre quelques secondes avant d'appuyer sur le bouton F5 (Stop).

A travers des fenêtres similaires à la figure n° III-5 on arrive à suivre pas à pas l'exécution du programme élaboré et de le corriger au fur et à mesure.

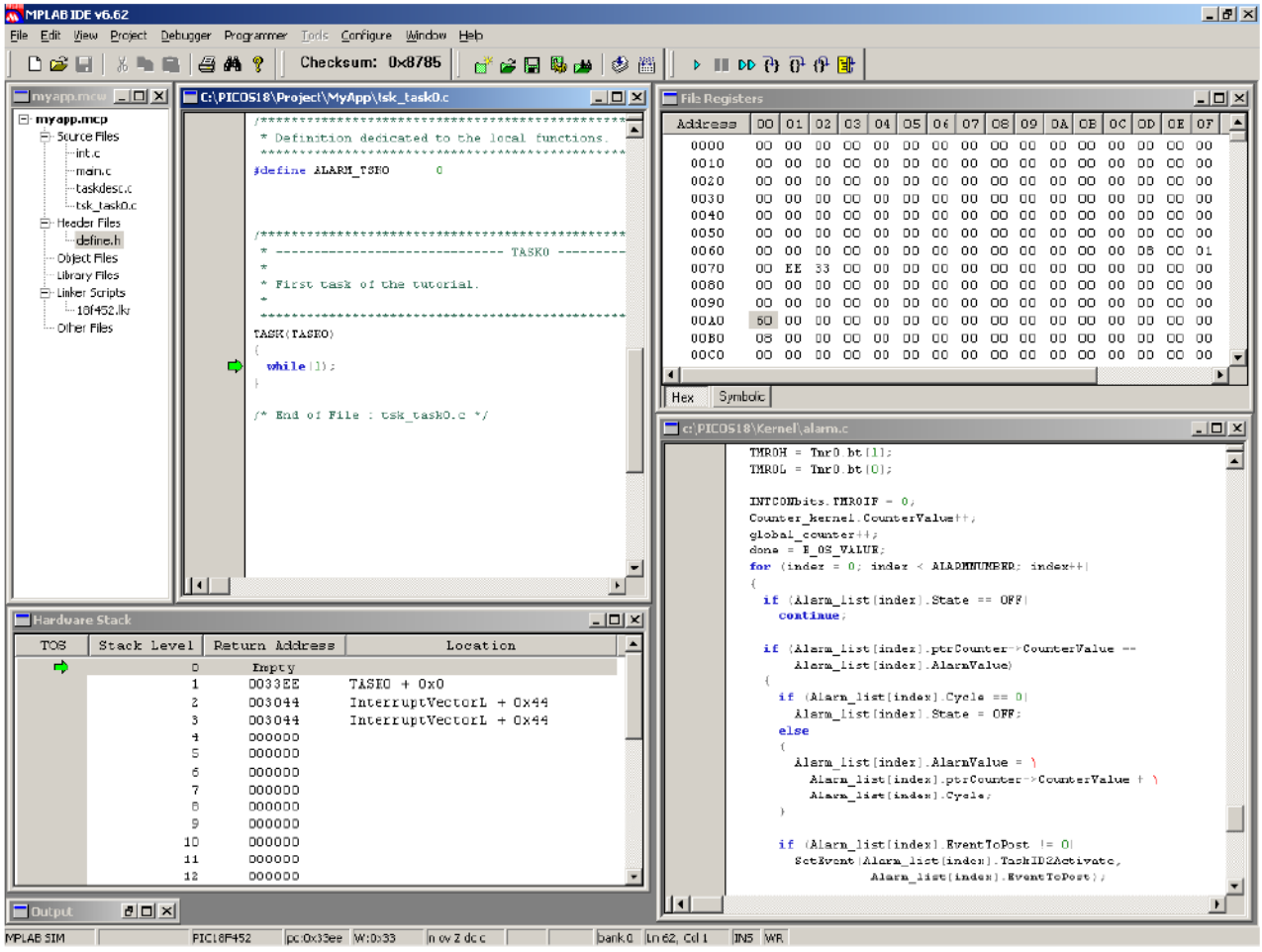


Figure n° III-5 : La fenêtre principale de l'environnement MPLAB

Conclusion

Dans ce travail nous nous somme familiariser à travailler avec le noyau PICos18 qui offre des solutions intéressantes pour un bon nombre d'applications. Ce travail nous a permis d'une part de manipuler des outils de développement pour microcontrôleurs Pic tels que MPLAB et C18 de Microchip. D'autre part, la réalisation nous a mis en face des contraintes de la programmation temps réel ce qui nous a permis de comprendre beaucoup de mécanismes liés à l'exécution des programmes.

Le travail réalisé est une modeste contribution à l'amélioration du code embarqué sur le Data logger afin d'améliorer les caractéristiques de ce dispositif, néanmoins il est nécessaire d'effectuer des séries de test pour s'assurer du bon fonctionnement du code implémenté.

La simulation n'a pas été achevée à 100% pour plusieurs problèmes liés au soft utilisé ainsi qu'a certains bugs du code source du PICos18.