REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur
Et de la Recherche Scientifique

Université Dr. YAHIA FARES* de *Médéa



Mémoire de Projet de Fin d'Etude pour
L'Obtention du Diplôme de Master (LMD)
En Génie Electrique
Option: Systèmes Intelligents Et Communicants

Thème :

Etude et portage du RTOS µC/OS-II sur un dsPIC33FJ32MC204

Proposé et dirigé par :

Mr : CHADLI .S.

Réalisé par :

M^elle^: Hadjer BENLAKEHAL

M^elle^: Meriem KACI

Promotion 2012-2013

Les sciences sont des serrures dont la clé est l'étude

La science consiste à oublier ce qu'on croit savoir, et la sagesse à ne pas s'en soucier

Remerciement

Nous tenons tout d'abord à remercier Dieu le tout puissant et miséricordieux, qui nous a donné la force et la patience d'accomplir ce Modeste travail.

En second lieu, nous tenons à remercier notre encadreur Mr : Samir CHADLI, son précieux conseil et son aide durant toute la période du travail.

Nos vifs remerciements vont également aux membres du jury pour l'intérêt qu'ils ont porté à notre recherche en acceptant d'examiner notre travail Et de l'enrichir par leurs propositions.

Enfin, nous tenons également à remercier toutes les personnes qui ont participé de près ou de loin à la réalisation de ce travail.

dedicace

A ma très chère mère

A celle qui s'est toujours dévouée et sacrifiée pour moi ; celle qui m'a aidée du mieuxqu'elle pouvait pour réussir ; celle qui m'a accompagnée tout au long de ce parcourspérilleux ; celle qui a toujours été là dans mes moments de détresse.

A mon très cher père

A celui qui m'a toujours encouragée et soutenue moralement,

A ma très chère sœurAkila, son mari Hamza et à mes très chers frères Ali et Abdelhak

qui m'onténormément aidée et à qui je témoigne mon affection et ma profonde reconnaissance.

A mes chères amies

Souhila ,Sihem , Kamelia,Mazoura,Ihcene .

A filles de mes tentes

Zola et meriem

Merci pour ton amour, tonamitié. Tu étais toujours là pour me soutenir, m'aider et m''écouter.

Que dieu te protège et te procure joie et bonheur et que notre amitié reste à jamais.

A mon binôme

Meriem Kaci

A tous mes ami(e)s, tous ceux que j'aime, tous ceux qui m'aiment et tous ceux qui me sont chers.

Hadjer

dedicace

A ma très chère mère

A celle qui s'est toujours dévouée et sacrifiée pour moi ; celle qui m'a aidée du mieux qu'elle pouvait pour réussir ; celle qui m'a accompagnée tout au long de ce parcours périlleux..

A mon très cher père

A celui qui m'a toujours encouragée et soutenue moralement,

A mes très chers frères rafik, Azzedine, Abdelmalek, et moncef

Qui m'ont énormément aidée

A mon amie intime

Selma

A mes amies

Sihem, Souhila, Soumia, Amina et Khadidja

A mes cousines paternelles et maternelles

Merci pour votre amour, votre amitié. Vous étiez toujours là pour me soutenir, m'aider et m''écouter.

A mon binôme

Hadjer Benlakehal

A tous mes ami(e)s, tous ceux que j'aime, tous ceux qui m'aiment et tous ceux qui me sont chers.

Meriem

Résumé

ملخص

في هذا العمل قمنا بدراسة نظام التشغيل في الوقت الحقيقي µC/OS-II (RTOS)وحدات تحكم . بعد ذلك ركزنا على منصة على أساس متحكم 16بت من رقاقة شركة هو .dsPic33FJ32MC204.

وقد تم البناء تحت MPLABبيئة التطوير المتكاملة من رقاقة واستخدام مترجم .C30.

الكلمات المفتاحية : تحميل **.MPLAB, µC/OS-II, RTOS, OS,**

Résumé

Dans ce travail nous avons étudié le système d'exploitation temps réel (RTOS) µC/OS-II. Par la suite nous l'avons porté sur une plate forme à base d'un microcontrôleur 16 bits de la firme Microchip qui est le dsPic33FJ32MC204.

Le portage a été réalisé sous l'environnement de développement intégré MPLAB de Microchip et on utilisant le compilateur C30.

Mots clés : MPLAB, µC/OS-II, SETR, SE, Portage.

Abstract

In this project we have studied real-time operating system (RTOS) µC/OS-II. After that, we have ported it on a target which is based on a microcontroller dspic33FJ32MC204 of The Microchip Firm

The porting has been realised with the integrated development environment MPLAB and using the compiler C30.

Key Words : MPLAB, µC/OS-II, RTOS, OS, Porting.

sommaire

Sommaire

Introduction générale…… 02

Chapitre I : Généralités sur les RTOS et le temps réel

1. Introduction…… 05
2. Systèmes embarqués…… 05
3. Systèmes temps réels…… 06
4. Architecture des systèmes temps réel…… 08
5. Concepts liés aux softwares des systèmes temps réels…… 08
6. Systèmes d'exploitation temps réels (RTOS)…… 10
7. RTOS pour systèmes embarqués…… 11
8. Noyaux temps réel pour systèmes embarqués…… 12
   - 8.1. Caractéristiques d'un noyau embarqué temps réel…… 13
   - 8.2. Exemple des noyaux…… 14
9. Nécessité d'un RTOS…… 14
10. Difficultés liées aux RTOS…… 15
    - 10.1. Inter-blocage…… 15
    - 10.2. Inversion de priorité…… 15
11. Conclusion…… 16

Chapitre II : Etude du RTOS µC/OS-II de Micrium

1. Introduction…… 18
2. Présentation du noyau…… 18
   - 2.1. Qu'est-ce que µC/OS-II?…… 18
   - 2.2. Fonctionnement de µC/OS-II…… 18
   - 2.3. Caractéristiques du RTOS µC/OS-II…… 19
   - 2.4. Structure du noyau µC/OS-II…… 12
3. Comparaison de dispositifs de µC/OS, et µC/OS-II…… 24

4. Plates formes hardwares prise en charge……. 25
5. Services fournis par µC/OS-II…….. 26
6. Conclusion…….…….…..... 27

Chapitre III : Portage du RTOS µC/OS-II au dsPic33FJ32MC204

1. Introduction……..... 29
2. Microcontrôleur de la famille dsPIC33…….. 29
3. Outils de développement utilisés dans le portage…….…….... 32
   3.1. Compilateur C30…….…… 32
   3.2. Environnement de développement MPLAB…….……..……..... 33
4. Organisation des répertoires et des fichiers du portage…….…..……..... 33
5. Portage du RTOS µC/OS-II au dsPIC33FJ32MC204…….…….. 36
   5.1. Préparation des outils de développement…….…… 36
   5.2. Création de projet **PortageMC**……. 36
   5.3. Description des fichiers du portage…….... 39
   5.4. Gestions des interruptions ……..…….... 47
   5.5. Code d'application…….…….... 47
   5.6. Package BSP…….…….…….... 51
   5.7. Compilation du portage sous MPLAB…….... 52
6. Conclusion…….…….... 53

Conclusion générale…….…….... 55

Références bibliographiques…….…….... 57

Annexe…….…….... 59

Liste des figures

Figure I.1	La structure générale d'un système embarqué…….……..	06
Figure I.2	La notion de contrainte temporelle "échéance"……	07
Figure I.3	Exemples de l'ordre de grandeur des échéances pour divers environnements..	07
Figure I.4	La coopération entre plusieurs taches……	09
Figure I.5	Le principe de gestion des Scheduler préemptifs et coopératifs……...	12
Figure I.6	L'inversion des priorités en utilisant les sémaphores…….....	15
Figure II.1	Structure du noyau µC/OS-II…….……...	21
Figure III.1	Le modèle des registres des dsPIC33…….……..	30
Figure III.3	Structure de compilateur C30…….…..	32
Figure III.3	Les options de configuration MPLAB…….…..……	37
Figure III.4	La structure du dossier…….……	38
Figure III.5	Rapport de compilation du portage de µC/OS-II au dsPic33FJ32MC204…..	52

Liste des tableaux

Tableau II.1 Quelques fichiers de µc/os-II avec leurs propres fonctions..... 22

Tableau II.2 Comparaison des versions de µC/OS..... 24

Liste d'abréviations

Liste d'abréviations

- ***ABS*** Anti Blockier System
- ***ADC*** Analog Digital Converter
- ***ARM*** Advanced RISC Machines
- ***ASCII*** American Standard Code for Information Interchange
- ***CCP*** Compte Chèque Postal
- ***CPU*** Central Processing Unit
- ***DMA*** Direct Memory Access
- ***DSP*** Digital Signal Processor
- ***FIFO*** Fist In First Out
- ***FPGA*** Field Programmable Gate Arrays
- ***IDE*** Integrated Development Environment
- ***IP*** Internet Protocol
- ***ISR*** Interrupt Service Register
- ***MCU*** Multipoint Control Unit
- ***MIPS*** Million Instructions Per Second
- ***OS*** Operating System
- ***OSA*** Operating System Account
- ***RAM*** Random Access Memory
- ***RISC*** Reduced Instruction Set Computer
- ***ROM*** ReadOnly Memory
- ***RTOS*** Real Time Operationg System
- ***TCP*** Transmission Control Protocol
- ***UART*** Universal Asynchronous Receiver Transmitter

Introduction générale

Introduction Générale

Sans aucun doute l'ère actuel est celui des microcontrôleurs, ils sont présents dans un grand nombre de produits de notre 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 Pic et dsPic. 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 programmations classiques 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 de fonctionnement, …etc.

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

Dans ce travail nous nous proposons d'étudier le RTOS µC/OS-II de la société Micrium en premier lieu et de l'implémenter (porter) par la suite sur une plate forme matérielle à base d'un microcontrôleur 16 bits de la firme Microchip qui est le dsPIC33FJ32MC204.

Notre mémoire est organisé de la manière suivante :

- Le premier chapitre présente un ensemble de généralités liées aux : systèmes embarqués, systèmes temps réel, RTOS et noyaux temps réel...etc.
- Le deuxième chapitre présente le noyau du RTOS (µC/OS-II), son architecture et ses services les plus connues.
- Le troisième chapitre décrit le portage du RTOS µC/OS-II sur le dsPic33FJ32MC204.

Enfin nous clôturerons ce document par une conclusion générale.

Chapitre I généralités sur les RTOS et le temps réel

Généralités sur les RTOS et le temps réel

I. Introduction

Aujourd'hui, les applications électroniques embarquées et les applications temps réel sont de plus en plus variées et elles prendront de plus en plus de place dans notre vie quotidienne. Réservés, il y a quelques années, aux applications industrielles, ces systèmes font leur apparition dans beaucoup d'autres secteurs de la vie courante tels que le transport, le multimédia, les consoles de jeux, ...etc.

En termes de complexité, les systèmes embarqués et temps réel couvrent un large spectre allant du simple microcontrôleur (pour le contrôle de la fermeture/ouverture d'une vanne, par-exemple), jusqu'aux systèmes répartis (pour le contrôle du trafic aérien, par exemple). Les enjeux économiques et les intérêts scientifiques liés aux systèmes temps réel et embarqués sont multiples. C'est la raison pour laquelle on assiste, depuis les années soixante-dix, à une profusion de langages, de méthodes, d'algorithmes, de protocoles de communication,…etc, pour le temps réel et l'embarqué.

II. 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, 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 plusieurs tâches dédiées [1]. 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, ports de communication,…etc) 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 I.1).

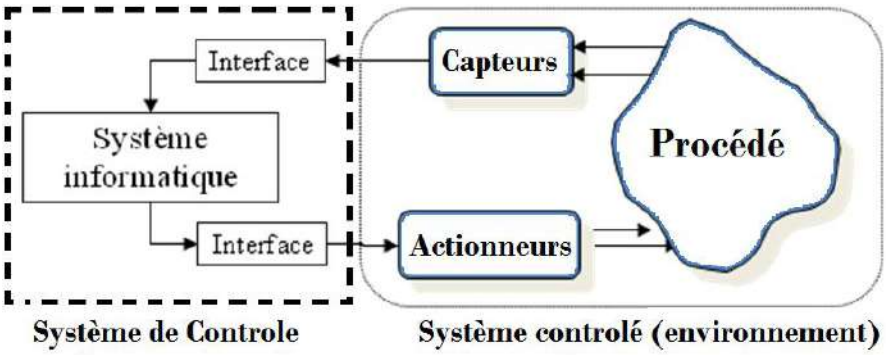


Figure I.1 : La structure générale d'un système embarqué

Un système électronique embarqué est généralement :

- ***Autonome :*** une fois enfouis dans une application ils ne sont (le plus souvent) plus accessibles.
- ***Réactif :*** ils doivent réagir à l'arrivée d'informations extérieures non prévues.
- ***Contraints :*** leurs ressources (calcul, batterie, mémoire, volume…) sont souvent limitées selon l'application considérée.
- ***Sûreté de fonctionnement :*** par fois systèmes dits «critiques» qui ne doivent jamais faillir !
- ***Temps réel :*** (très souvent) les temps de réponses de ces systèmes sont aussi importants que l'exactitude des résultats.

III. Systèmes temps réels

Le monde du temps réel est si vaste que ses technologies s'appliquent à divers domaines tels que le contrôle industriel, le transport, les télécommunications, les systèmes de sécurité, la domotique, ...etc. Ainsi le métro sans conducteur, le pilote automatique d'un avion, le système de freinage ABS de nos voitures sont des exemples très courants des systèmes temps réel qui font aujourd'hui partie de notre vie quotidienne.

On parle d'un **système temps réel** lorsque ce système est capable de contrôler (ou piloter) un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé.

Les systèmes temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles (échéance) dont le respect est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés. (Figure I.2).

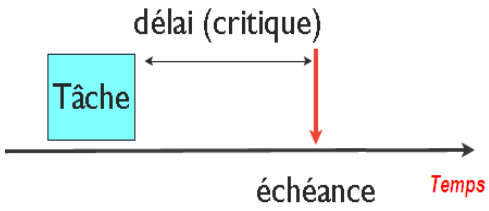


Figure I.2 : La notion de contrainte temporelle "échéance".

On appelle échéance une contrainte temps à laquelle doit au plus tard se produire un événement (deadline). L'échéance est définie généralement par la vitesse d'évolution de l'environnement réel au-delà de laquelle les résultats ne sont plus valides.

Il faut noter que :

- Le fait de rater une échéance 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.
- L'échéance temporelle est définie par la dynamique de l'environnement de l'évolution du système temps réel. Il peut être très court de 100 ns à 1 ms pour les systèmes de mesures comme il peut être très long d'une heure à quelques heures pour le contrôle des réactions chimiques. (Figure I.3).

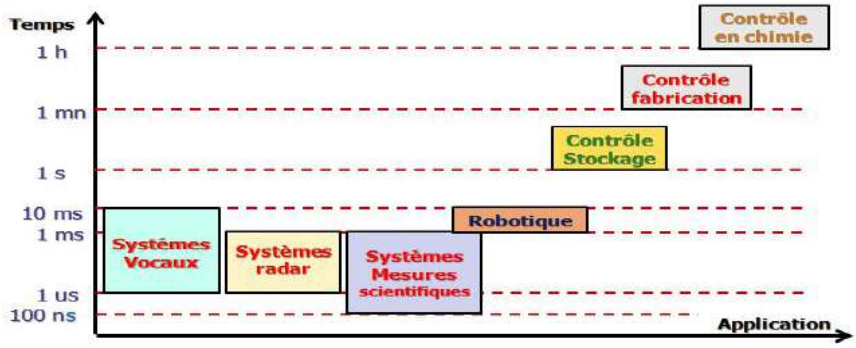


Figure I.3 : Exemples de l'ordre de grandeur des échéances pour divers environnements

IV. Architecture des systèmes temps réel

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

- ***Soit directement dans le procédé contrôlé :*** c'est le cas des systèmes embarqués (*embedded systems*). Le procédé est souvent très spécialisé et fortement dépendant du calculateur. Les exemples de ces systèmes sont nombreux : contrôle d'injection automobile, stabilisation d'avion, électroménager, missile, ...etc. 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.[2]

V. Concepts liés aux softwares des systèmes temps réels

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

On distingue deux classes de tâches : les threads (dont la traduction littérale en français « fil », n'est jamais employé) sont des tâches qui s'exécutent dans le même espace mémoire. Différents threads partagent donc les mêmes variables globales. Les threads sont compilés comme un même programme.

A l'opposé, les processus (process) est une tâche qui s'exécute dans un espace mémoire propre. Un processus est un programme qui tourne. Les processus sont compilés comme des programmes séparés et ne partagent pas leurs variables avec les autres processus. Dans les systèmes multitâches les plus complets, un processus peut contenir plusieurs threads.[3].

B- L'ordonnancement : La stratégie employée pour choisir dans quel ordre les tâches vont pouvoir s'exécuter s'appelle l'ordonnancement (scheduling).

Dans un système d'exploitation ordinaire, non-temps réel, l'ordonnanceur (scheduler) est conçu pour optimiser le débit de calcul, de façon à pouvoir atteindre un taux d'utilisation du processeur proche de 100%. A l'opposé, dans un système d'exploitation temps réel (RTOS), l'ordonnanceur est conçu pour minimiser la latence des taches les plus prioritaires et les rendre déterministes et prévisibles. Dans la pratique, cela n'est pas compatible avec un taux d'utilisation du processeur supérieur à 70%.

Pour qu'un système temps réel fonctionne correctement, il est préférable qu'il passe son temps à attendre des excitations. Pendant qu'il attend un évènement (généralement signalé par une interruption), le processeur exécute la tâche idle, correspondant à la priorité la plus basse.[3]

C- La Coopération: Dans un système embarqué, on peut généralement assimiler chaque tâche à une boucle infinie. Pour que le partage du temps puisse opérer, les tâches ne doivent jamais exécuter d'opérations bloquantes telles que les scrutations.

Au contraire, toutes les tâches doivent absolument faire appel à l'une des primitives du système d'exploitation qui permettent d'attendre un événement sans bloquer le système.

Lors de cet appel, si l'événement attendu est présent, l'exécution se poursuit normalement. Cependant, si l'événement n'est pas encore présent l'ordonnanceur met en attente la tâche en cours pour exécuter la tâche la plus prioritaire en attente d'un événement déjà survenu.[3].

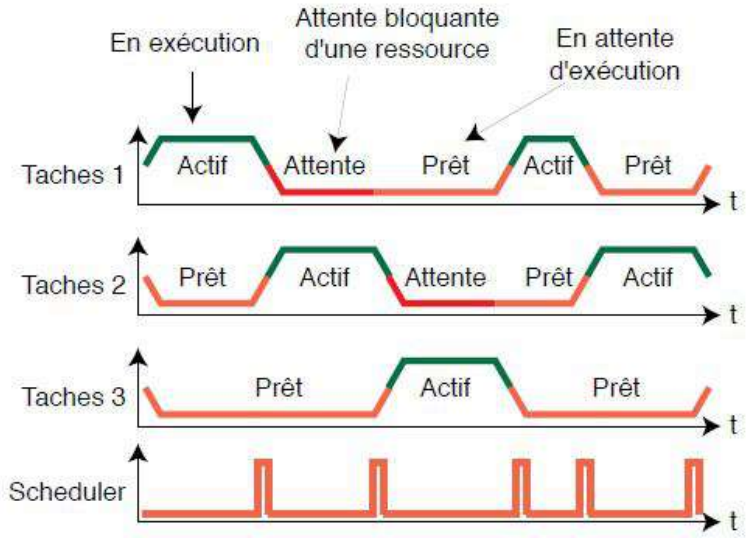


Figure I.4 : La coopération entre plusieurs taches

D- Les Evénements: Typiquement, les appels au système d'exploitation qui peuvent provoquer une coopération sont : les fonctions de délai, les attentes de sémaphores, de boîte à lettre, de file d'attente de messages ou d'indicateurs binaires (events).

- Un délai correspond à l'attente d'une période de temps calculée en unité de temps (compté par les interruptions périodiques d'un timer).
- Une tâche peut attendre qu'un sémaphore (élément binaire ou compteur/décompteur) devienne positif pour être réveillée. Ils sont souvent utilisés pour synchroniser des tâches (lorsqu'ils sont initialisés à zéro) ou pour garantir l'exclusion mutuelle (lorsqu'ils sont initialisés à un) lors de l'accès à une ressource partagée par plusieurs tâches.
- Une boîte à lettre est un objet de communication entre tâche. Il contient une donnée - ou plus souvent un pointeur sur une donnée - qu'une tâche peut attendre.
- La file d'attente de messages est simplement une extension du concept de boîte à lettre pouvant contenir plusieurs messages. Comme son nom l'indique, les messages y sont organisés en FIFO. La file d'attente est nécessaire si l'arrivée des messages est sporadique, c'est à dire lorsque les messages sont susceptibles d'arriver par paquet. [3].

E- La Préemption : Si le système d'exploitation le supporte, il est possible qu'une tâche prioritaire reprenne la main dès qu'une interruption la rend prête. On parle alors de **préemption**. Un système d'exploitation supportant la préemption est dit **préemptif**. Cela permet d'obtenir une latence beaucoup plus courte pour les tâches de plus hautes priorités. Ainsi, une tâche de haute priorité ne peut pas être bloquée par une tâche de priorité plus faible. Notez que dans un OS préemptif, chaque tâche possède nécessairement sa propre pile.

Les autres systèmes d'exploitation, dits coopératifs, sont moins réactifs puisqu'il est possible de monopoliser le processeur par des tâches de basses priorités.[3].

VI. Systèmes d'exploitation temps réels (RTOS)

L'évolution majeure des systèmes d'exploitations (standards ou embarquées), concerne le support de l'exécution concurrente de plusieurs traitements (taches) a donné naissance aux systèmes dits multitâches. Toutefois, les systèmes multitâches classiques 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 la disponibilité de la ressource microprocesseur afin de prendre en compte des contraintes temporelles. Un système d'exploitation conçu pour prendre en compte ces contraintes est appelé système d'exploitation temps réel [4] (en Anglais Real Time Operating System "RTOS").

VII. RTOS 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) se caractérisent par leur comportement déterministe et prévisible [4]**.** Il faut noter au passage, qu'il existe deux grandes familles de noyaux temps réels : les RTOS préemptifs et 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.[5]

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 I.3).[5]

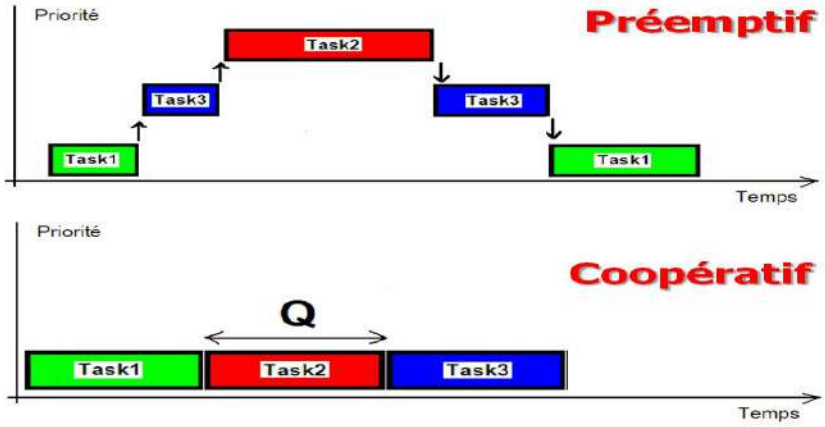


Figure I.5 : Le principe de gestion des Scheduler préemptifs et coopératifs

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, la fin d'une 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.[5].

VIII. Noyaux temps réel pour systèmes embarqués

Ce sont des noyaux exécutifs de petites tailles (par rapport aux RTOS), 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].

VIII.1. 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].

VIII.2. Exemple des noyaux

1- **VxWorks :** Aujourd'hui c'est le noyau temps réel le plus utilisé dans l'industrie. Il est développé par la société Wind River. VxWorks inclut en natif un support TCP/IP et une interface de programmation intégrant les *sockets*. Le gros inconvénient de ces deux systèmes est le coût important des licences. Il est également nécessaire d'utiliser un environnement de compilation croisé.[6]

2- **µC/OS (micro-C OS) et µC/OS II** *:* développé par le Canadien Jean J. Labrosse, est destiné à des environnements de très petite taille comme les microcontrôleurs de type 68HC11. Il est maintenant disponible sur un grand nombre de processeurs et peut intégrer des protocoles standards comme TCP/IP (µC/IP). Il est utilisable gratuitement pour l'enseignement et la recherche.[7]

3- **OSA :** 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.[8]

4- **Salvo RTOS de Pumpkin :** c'est un RTOS préemptif d'un bon niveau possédant de nombreuses fonctions de messages et de synchronisations inter-tâches. Il est payant et la version gratuite est limitée en nombre de tâches que l'on peut déclarer.[9]

5- **LynxOS:** développé par la société LynuxWorks qui a récemment modifié son nom en raison de son virage vers Linux avec le développement de BlueCat. C'est un système temps réel conforme à la norme Posix.[10]

IX. Nécessité d'un RTOS

En examinant le code des systèmes embarqués, nous remarquons en général 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 opte souvent pour l'utilisation d'un système d'exploitation temps réel (RTOS) ou d'un noyau temps réel 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 court et plus simple à gérer avec une approche de temps réel dans l'exécution des différentes parties du soft.[11].

X. Difficultés liées aux RTOS

1. L'inter-blocage : Lorsque plusieurs ressources sont partagées par plusieurs tâches, il est utile de protéger l'accès à chaque ressource par un sémaphore de façon à garantir l'exclusion mutuelle. Dans ce cas, il faut définir un ordre pour la demande des sémaphores. A défaut, un inter-blocage peut se produire.

La situation est la suivante. Une tâche T1 demande et obtient un sémaphore S1 puis une tâche T2 demande et obtient un sémaphore S2. Si la tâche T1 demande alors le sémaphore S2 elle ne peut pas l'obtenir puisqu'il est occupé par la tâche T2. Si de plus, la tâche T2 demande le sémaphore S1, elle ne peut pas l'obtenir tant qu'il est occupé par la tâche T1. Dans cette situation, les deux tâches sont en attente. Elle ne s'exécute plus et ne peuvent pas libérer leur sémaphore. La situation est donc définitivement bloquée pour l'une comme pour l'autre. On parle parfois d'une étreinte fatale.[3].

2. L'inversion de priorité : elle correspond à une situation dans laquelle une tâche prioritaire est bloquée par une tâche de basse priorité qui détient le sémaphore qu'elle réclame. Si une tâche de niveau intermédiaire est prête, c'est elle qui s'exécute, empêchant la tâche de basse priorité de relâcher son sémaphore, ce qui bloque la tâche prioritaire. Il existe deux moyens pour éviter cette situation. La première consiste à associer une priorité à chaque sémaphore. Toute tâche qui détient ce sémaphore aura sa priorité au moins égale à celle du sémaphore. Une autre solution consiste à faire hériter à la tâche qui détient le sémaphore la priorité de la tâche la plus prioritaire qui réclame ce sémaphore (Figure I.4).[3]

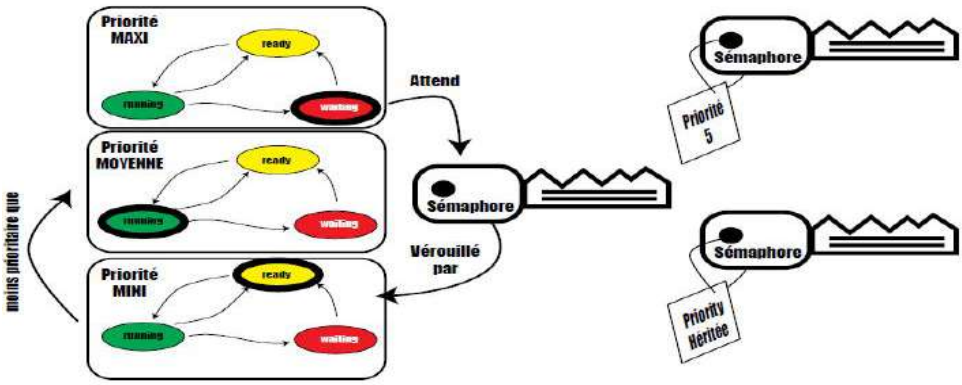


Figure I.6 : L'inversion des priorités en utilisant les sémaphores

XI. Conclusion

Un système embarqué temps réel est un système complexe qui intègre du logiciel et du matériel conçus ensemble afin de fournir des fonctionnalités données. Il contient généralement un ou plusieurs microprocesseurs destinés à exécuter un ensemble de programmes définis lors de la conception et stockés dans des mémoires. Le système matériel et l'application (logiciel) sont intimement liés et immergés dans le matériel et ne sont pas aussi facilement discernables comme dans un environnement de travail classique de type ordinateur de bureau PC (Personal Computer).

Chapitre II

RTOS µC/OS-II

Etude du RTOS µC/OS-II de Micrium

I. Introduction

Dans ce chapitre nous allons présenter le système d'exploitation temps-réel µC/OS-II l'objet de notre étude; nous commençant par la description du noyau de cet RTOS, nous présentons par la suite la structure de ce dernier et enfin nous terminons par une comparaison entre les deux premières versions de cet RTOS.

II. Présentation du noyau du RTOS

II.1. Qu'est-ce que µC/OS-II?

µC/OS-II est un système d'exploitation de la société Micrium développé par le canadien Jean. J. Labrosse basé sur le noyau temps-réel µC/OS (première version), son code source est écrit en langage C et contient une petite partie en langage assembleur. Il est doté d'une documentation assez riche en langue anglaise. µC/OS-II fonctionne sur plusieurs architectures différentes de processeurs avec un ensemble de ports disponibles pour le téléchargement du site Web de la société Micrium.

Il peut intégrer des protocoles standards comme TCP/IP (µC/TCP) pour assurer une connectivité IP sur une liaison série. C'est un noyau entièrement portable, ROMable, évolutif et préemptif, déterministe en temps réel, multitâches pour les microprocesseurs, microcontrôleurs et DSP. [12].

II.2. Fonctionnement de µC/OS-II

µC/OS-II peut gérer jusqu'à 255 tâches d'application, il comprend divers services tels que: les sémaphores, les drapeaux d'événement; les sémaphores d'exclusion mutuelle qui éliminent les inversions de priorité non bornés; les boîtes aux lettres des messages et des files d'attente; la gestion du temps et de la minuterie, et la gestion de blocs de mémoire de taille fixe. L'empreinte

mémoire du RTOS µC/OS-II peut être redimensionné selon les services intégrés (entre 6 à 26 Ko) pour contenir les caractéristiques requises pour une application spécifique.

Le temps d'exécution pour la plupart des services fournis par µC/OS-II est à la fois constant et déterministe; ce temps d'exécution ne dépend pas du nombre de tâches en cours d'exécution dans l'application. [12]

µC/OS-II est un RTOS largement utilisé dans divers domaines de l'industrie et de la vie courante :

- Avionics : utilisé sur le robot mobile" Mars Curiosity Rover";
- Equipement / dispositifs médicaux ;
- Équipement de communications et téléphones portables ;
- Equipements électroménager;
- Commandes industrielles ;
- Ainsi q'un large éventail d'autres applications embarquées critiques. [12]
-

II.3. Caractéristiques du RTOS µC/OS-II

- *Code source* : il est fourni en langage C, environ 5500 lignes, bien organisé.
- *Portable* : µC/OS-II peut être implémenter (porter) sur à un grands nombre d'architecture de CPU. Il peut fonctionner sur la plupart des microprocesseurs 8, 16, 32 et 64 bits ou sur des microcontrôleurs et processeurs DSP.
- *ROMable* : µC/OS-II a été conçu aux applications embarquées, ce qui signifie que si nous disposons d'une chaîne d'outil approprié (compilateur C, assembleur et éditeur de liens), nous pouvons réellement l'intégrer dans la mémoire d'un système donné.
- *Évolutif*: le µC/OS-II a été conçu de sorte que nous pouvons utiliser que les services dont nous avons besoin pour notre application, ce qui signifie qu'un produit peut utiliser quelques services de µC/OS-II, tandis que l'autre produit peut bénéficier de l'ensemble des services. Cette caractéristique nous permet de réduire la quantité de mémoire de base (l'empreinte mémoire) requis par cet RTOS dans un produit.
- *Préemptif* : µC/OS-II est un noyau temps réel entièrement préemptif, ce qui signifie qu'il exécute toujours la tâche la plus prioritaire qui est prête.
- *Multitâche*: µC/OS-II peut gérer jusqu'à 255 tâches. Chaque tâche a une priorité unique attribué à elle, ce qui signifie que µC/OS-II ne peuvent pas faire l'ordonnancement round-robin. Il ya donc 255 niveaux de priorité.

- D*éterministe*: Le temps d'exécution de la plupart des fonctions et des services µC/OS-II sont déterministes, ce qui signifie que nous pouvons toujours savoir combien de temps µC/OS-II nécessitera pour exécuter une fonction.
- *Pile de Tache:* Chaque tâche nécessite sa propre pile, mais µC/OS-II permet à chaque tâche d'avoir une taille de pile différente, ce qui permet de réduire la quantité de RAM nécessaire à notre application. Grâce à la fonction de vérification de pile nous pouvons déterminer exactement combien d'espace de pile chaque tâche nécessite réellement.
- *Services:* µC/OS-II fournit un certain nombre de services système, tels que les sémaphores, l'exclusion mutuelle, drapeaux d'événement, les boîtes de messages, les files d'attente de messages, les partitions de mémoire de taille fixe, gestion des tâches, des fonctions de gestion du temps,... etc.
- *Robuste et fiable:* µC/OS-II est basé sur le noyau de la première version (µC/OS), qui a été utilisé dans des centaines d'applications commerciaux depuis 1992. Utilisant le même noyau et la plupart des fonctions du µC/OS, il offre en plus beaucoup de nouvelles fonctionnalités.

Il faut noter qu'en Juillet 2000, µC/OS-II a été certifiée dans un produit avionique, ou chaque dispositif, fonction et ligne du code de µC/OS-II ont été examinés et testés pour démontrer qu'il est sûr et assez robuste pour être utilisé dans les systèmes critiques nécessitant un grand niveau de sécurité de fonctionnement. [13]

II.4. Structure du noyau µC/OS-II

La Figure II.1 illustre l'architecture du RTOS µC/OS-II et sa relation avec le matériel, le timer (temporisateur) et d'autres dispositifs tels que : UARTs, ADCs…etc.

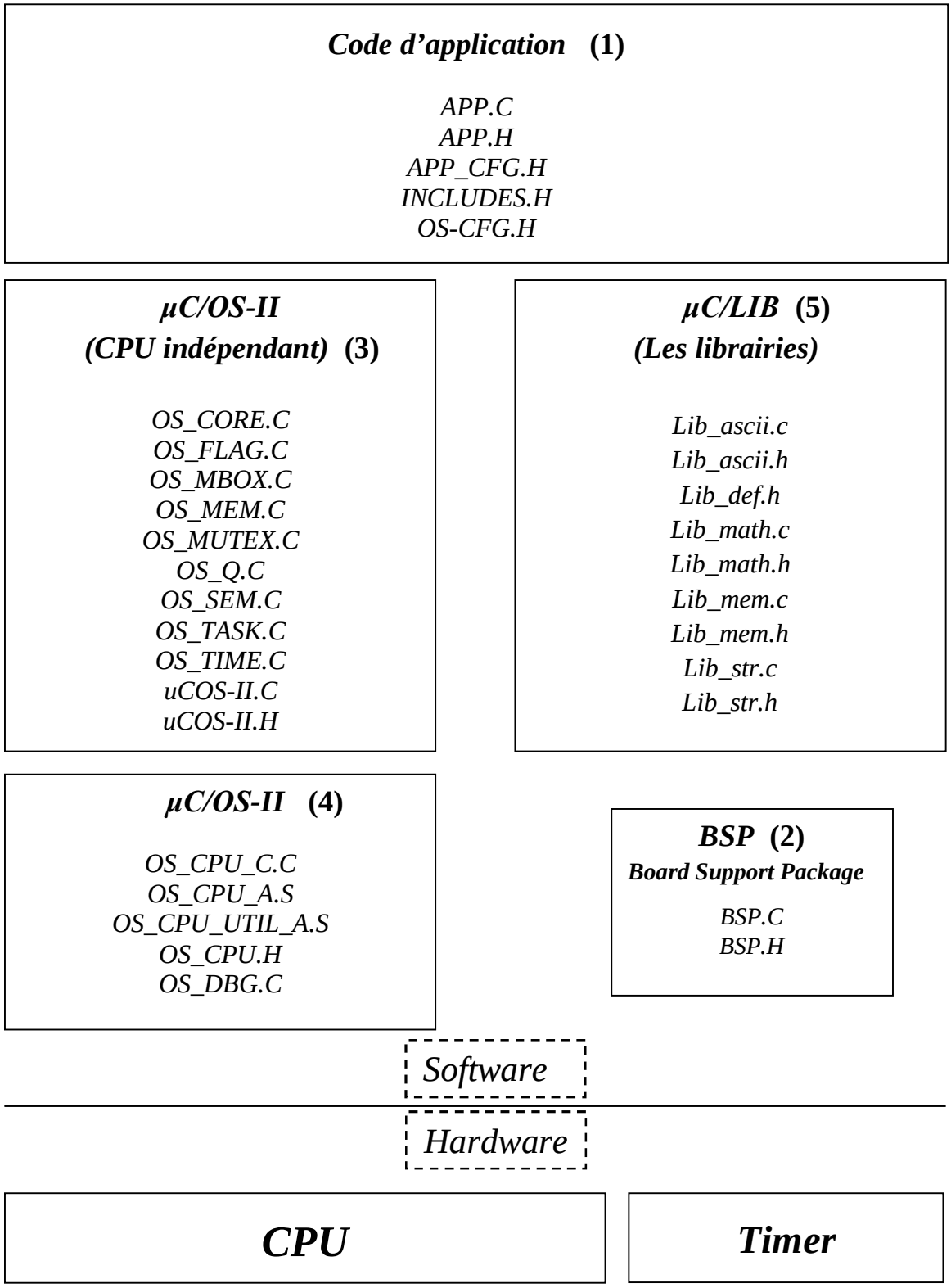


Figure II.1 : Structure du noyau µC/OS-II

L'architecture du RTOS µC/OS-II est composée de deux parties :

- ✓ La partie hardware.
- ✓ La partie software.

A. *Partie hardware :*

Cette partie est constituée de deux composants essentiels : Le CPU et le Timer (temporisateur).

B. *Partie software :* peut être est décomposée en 5 parties :

1. *Le Code d'application :* Il se compose des fichiers du projet ou du produit, qui sont simplement appelés **APP.C** et **APP.H**, une application peut contenir plusieurs fichiers**.** Le code d'application est typiquement celui qui contient la fonction main ().
2. *BSP* : c'est un code qui est typiquement écrit pour interfacer des périphériques sur un carte cible. Par exemple un code qui peut allumer et éteindre les diodes électroluminescentes (les LED), ou un code pour lire des commutateurs, capteurs de température, ... d'une carte cible.
3. *µC/OS-II (*CPU Independent) : C'est le code indépendant de processor de µC/OS-II. Ce code est écrit selon la norme C ASCII.

Le tableau suivant décrit les fichiers essentiels de cette partie du RTOS et leurs propres fonctions :

Fichiers	***Fonctions***
OS_CORE.C	****OSInt(void)** : pour initialiser µC/OS-II* ****OSIntEnter()** : notifie* uC/OS-II que l'ISR est en cours de traitement, ce qui permet de garder une trace de l'interruption. OSIntEnter () est utilisée en conjonction avec OSIntExit (). ***OSStart ()** :* commence le traitement multitâche sous µC/OS-II. Cette fonction est généralement appelée à partir de notre code de démarrage mais après avoir appelé OSInit ().
OS_FLAG.C	***OSFlagAccept () :*** nous permet de vérifier le statut d'une combinaison de bits à chaque mise ou effacement d'un groupe de drapeau de l'événement. ***OSFlagCreate () :*** est utilisée pour créer et initialiser un groupe de drapeau d'événement. ***OSOSFlagDel () :*** est utilisée pour supprimer un groupe de drapeaux d'événements.
OS_MBOX.C	***OSMboxAccept () :*** nous permet de voir si un message est disponible dans une boîte aux lettres donnée. ***OSMboxCreate () :*** crée et initialise une boîte aux lettres. ***OSMboxDel () :*** est utilisée pour supprimer un message d'une boîte aux lettres.

Fichiers	Fonctions
OS_MEM.C	***OSMemCreate () :** crée et initialise une partition de mémoire. ***OSMemGet () :** reçoit un bloc de mémoire à partir d'une partition de mémoire. ***OSMemPut () :** renvoie un bloc de mémoire à une partition de mémoire.
OS_MUTEX.C	***OSMutexAccept () :** nous permet de vérifier si une ressource est disponible. Et il ne suspend pas la tâche appelante si la ressource n'est pas disponible. ***OSMutexCreate () :** est utilisée pour créer et initialiser un mutex. Un mutex est utilisé pour obtenir l'accès exclusif à une ressource. ***OSMutexDel () :** est utilisée pour supprimer un mutex.
OS_Q.C	***OSQAccept () :** vérifie si un message est disponible dans la file d'attente de message et il ne suspend pas la tâche appelante si un message n'est pas disponible. ***OSQCreate () :** crée une file d'attente de messages. ***OSQDel () :** est utilisée pour effacer une file d'attente de messages.
OS_SEM.C	***OSSemAccept () :** vérifie si une ressource est disponible ou un événement s'est produit. ***OSSemCreate () :** crée et initialise un sémaphore. ***OSSemDel () :** est utilisée pour supprimer un sémaphore.
OS_TASK.C	***OSTaskChangePrio ()** : modifie la priorité d'une tâche. ***OSTaskCreate () :** crée une tâche pour qu'elle puisse être géré par µC/OS ***OSTaskCreateExt () :** crée une tâche qui doit être géré par µC/OS-II. Cette fonction sert pour le même but que OSTaskCreate(), sauf qu'elle nous permet de spécifier des informations supplémentaires sur notre tâche à µC/OS-II.
OS_TIME.C	***OSTimeDly () :** permet à une tâche de se retarder à un nombre entier d'impulsions d'horloge. ***OSTimeDlyHMSM () :** permet à une tâche de se retarder pour un montant spécifié par l'utilisateur en heures, en minutes, en secondes et en millisecondes. ***OSTimeDlyResume () :** reprend une tâche qui a été retardée par un appel soit à OSTimeDly () ou OSTimeDlyHMSM ().

Tableau II.1 : Quelques fichiers de µc/os-II avec leurs propres fonctions

4. *µC/OS-II (CPU Spécifique) :* C'est le code de µC/OS-II qui est adapté à une architecture spécifique d'un CPU et s'appelle un portage.

5. *µC/LIB (Les librairies) :* c'est un ensemble de fichiers source permettant de fournir des fonctions communes telles que : la copie de mémoire, les fonctions liées à la norme C ASCII [13].

III. Comparaison de dispositifs de µC/OS, et µC/OS-II

Le tableau suivant montre l'évolution du RTOS µC/OS vers la version µC/OS-II et comparant quelques caractéristiques des deux versions: [12]

Dispositifs	*µC/OS*	*µC/OS-II*
Date d'introduction	1992	1998-présent
La disponibilité du Code source	Oui	Oui
Nombre maximum de tâches	64	255
Nombre de tâches à chaque niveau de priorité	1	1
Multitâche préemptif	Oui	Oui
Ordonnancement Round robin	Non	Non
Sémaphores	Oui	Oui
Sémaphores d'exclusion mutuelle	Non	Oui
Drapeaux d'événement	Non	Oui
Messages des boîtes aux lettres	Oui	Oui
Timers Logiciels	Non	Oui
Empreinte mémoire Code	3k-8k	6k-26k
Empreinte mémoire Data	1k+	1k+
Temporisateurs	Non	Oui
Evolutif	Oui	Oui
Configuration d'exécution	Non	Non
Nombre de service	~20	~90
Mesure des performances intégrées	Non	Limitées
Conforme à la norme C ANSI	Non	Oui

Tableau II.2: Comparaison des versions de µC/OS

IV. Plates formes hardwares prise en charge

µC/OS-II peut être porté sur plusieurs architectures des processeurs et des microcontrôleurs, parmi ces derniers nous citons à titre indicatif et non restrictif :

1- **Nios II :**

C'est un Processeur softcore propriétaire d'Altera. Il est basé sur un cœur RISC 32 bits implémenté dans le FPGA à partir de Quartus.

C'est le plus rapide, il possède une cache instruction ce qui le rend très rapide. Il est possible de le tester gratuitement, mais l'ordinateur doit toujours être connecté au FPGA, sinon le programme s'arrête.[14]

2- **PIC32:**

Sortis en novembre 2007, les PIC32 sont des microcontrôleurs 32 bits. Ils sont basés sur le cœur MIPS32 M4K (Architecture MIPS). L'architecture interne des PIC32 est basée sur le Bus Matrix (128 bits) permettant d'avoir jusqu'à 4 transactions simultanées. Les canaux de DMA peuvent donc fonctionner en parallèle de l'exécution des instructions. Ce cœur comporte des instructions 32 bits et aussi 16 bits (MIPS16e) qui peuvent être mélangées à volonté pour réduire la taille du code.[15]

3- **Fujitsu FR:**

C'est processeur de la famille RISC 32 bits. Nouvelles variantes comprennent une unité à virgule flottante. Il est soutenu par Softune, compilateur GNU et d'autres environnements de développement intégrés.[16]

4- **ARM Cortex-M :**

ARM Cortex-M est une famille de processeur RISC 32-bits d'architecture ARM servant à la fois de microprocesseur et de microcontrôleur à destination de l'embarqué. Elle implémente d'abord le jeu d'instructions ARMv6-M (Cortex-M0, Cortex-M0+ et Cortex-M1) puis le ARMv7-M (Cortex-M3 et Cortex-M4). Cette famille a été développé par ARM Ltd en parallèle à la famille Cortex-A (ARMv7-A) destinée au marché des Smartphones et tablettes tactiles et à la famille Cortex-R (ARMv7-R) destinée au temps réel. La carte en matériel libre Arduino Due est équipée d'un Cortex-M3.[17]

5- **MicroBlaze :**

Le MicroBlaze est un cœur de processeur softcore RISC d'architecture Harvard, entièrement 32 bits de la société Xilinx. Il est conçu pour être implémenté sur les FPGA de ce fabricant.[18]

En effet, il possède :

- ✓ des registres internes de 32 bits.
- ✓ Un bus d'instructions interne.
- ✓ Un bus de données interne.
- ✓ un bus d'instructions externe.
- ✓ un bus de données externe.

V. Services fournis par µC/OS-II

V.1. Gestions des sémaphores

µC/OS-II utilise des sémaphores constitués de deux éléments: un entier non signé sur 8 bits utilisé pour maintenir le compteur du sémaphore (0 à 255) et une liste des tâches en attente pour compter le sémaphore soit supérieure à 0. µC/OS-II fournit six fonctions pour accéder aux sémaphores: OSSemAccept (), OSSemCreate (), OSSemDel (), OSSemPend, OSSemPost () et OSSemQuery ().

Pour activer les services de sémaphores, nous devons définir les constantes de configuration dans le fichier OS_CFG.H. Il faut noter qu'aucun des services de sémaphores sont activés lorsque OS_SEM_EN est mis à 0. Pour activer la fonction (service), il suffit de régler la constante à 1 dans la configuration. [13].

V.2. Sémaphores d'exclusion mutuelle (mutex)

Ils sont utilisés par des tâches pour obtenir un accès exclusif aux ressources. Les mutex sont des sémaphores binaires qui présentent des caractéristiques supplémentaires au-delà du mécanisme des sémaphores normaux prévus par µC/OS-II.[13]

V.3. Gestion de drapeau d'événement

Le drapeau d'événement se composent de deux éléments: une série de bits (soit 8, 16, ou 32) servant à maintenir l'état actuel des événements dans le groupe et une liste des tâches en attente d'une combinaison de ces bits pour l'activé (1) ou l'éffacé (0).

µC/OS-II fournit six fonctions pour gérer les drapeaux d'évenements : OSFlagAccept () OSFlagCreate (), OSFlagDel (), OSFlagPend (), OSFlagPost (), et OSFlagQuery ().

Pour activer les services événement-drapeau dans uC/OS-II, nous devons définir les constantes de configuration dans le fichier OS_CFG.H. En effet, pour activer la fonction il suffit de régler la constante à 1 dans la configuration. [13].

V.4.Gestion des messages de boite aux lettres

Une boîte de message (ou plus simplement une boîte aux lettres) est un objet de µC/OS-II, qui permet à une tâche ou une ISR pour envoyer une variable de pointeur à une autre tâche.

Le pointeur est généralement initialisé pour pointer vers une structure de données spécifique à l'application contenant un message. µC/OS-II fournit six services pour accéder à des boîtes aux lettres: OSMboxCreate () OSMboxPend (), OSMboxPost (), OSMboxPostOpt (), OSMboxAccept (), et OSMboxQuery (). Il faut noter qu'aucune des fonctions de boîte de message sont activées lorsque OS_MBOX_EN est réglé sur 0. [13].

V.5. Scheduler

Le scheduler, appelé aussi le répartiteur, est la partie du noyau chargé de déterminer la tâche qui prend la ressource CPU. La plupart des noyaux temps réel sont basés sur la priorité. Pour chaque tâche on attribue une priorité en fonction de son importance. La priorité pour chaque tâche est spécifique à l'application.

Dans un noyau à base de priorité, le contrôle de CPU est toujours donné à la tâche de priorité la plus élevée prêt à s'exécuter. [13].

VI. Conclusion

Après avoir présenté les différents aspects liés aux RTOS µC/OS-II, nous passons au chapitre suivant qui présente le portage de ce RTOS sur un microcontrôleur 16 bits de la firme Microchip qui est le dsPic33FJ32MC204.

Chapitre III
Portage du RTOS µC/OS-II au dsPic33FJ32MC204

Portage du RTOS µC/OS-II au dsPic33FJ32MC204

I. Introduction

Dans ce chapitre nous allons présenter le portage du RTOS µC-OS-II sur le microcontrôleur dsPic33FJ32MC204 de la firme Microchip, nous commençant par décrire quelque caractéristique de l'architecture interne des microcontrôleurs de la famille dsPIC33. Par la suite nous allons présenter les outils de développement pour réaliser le portage du RTOS. Enfin nous décrirons comment le RTOS µC/OS-II à été porté au microcontrôleur dsPIC33FJ32MC204.

II. Microcontrôleur de la famille dsPIC33

La famille dsPIC33 est l'une des architectures microcontrôleurs 16 bits de la firme Microchip; elle est conçue pour les applications de traitement de signal. Voici quelques caractéristiques de la famille des cœurs de processeurs dsPIC33:

- 16 Registres de 16 bits à usage général;
- Double accumulateurs de 40 bits avec un multiplieur câblé 16x16 bits;
- Mots d'instruction de longueur variable de 24 bits, compteur de programme de 23 bits;
- Une Ram DMA dédié de 1 Ko;
- Traitement d'interruption déterministe de haute vitesse; ....etc.

Dans l'architecture des dsPIC33 l'octet de poids faible d'une information (instruction ou donnée) est placé à un emplacement de mémoire inférieur à son octet le plus significatif.

La figure III.1 montre le modèle des registres du dsPIC33 qui se compose de 16 registres à usage général, deux accumulateurs, un registre de Tableau de page de données, un registre d'adresse de page de visibilité de l'espace de programme, un compteur de boucle de répétition,

trois registres dédiés à la boucle DO, un registre de configuration du noyau, et un registre de statut de CPU.[19]

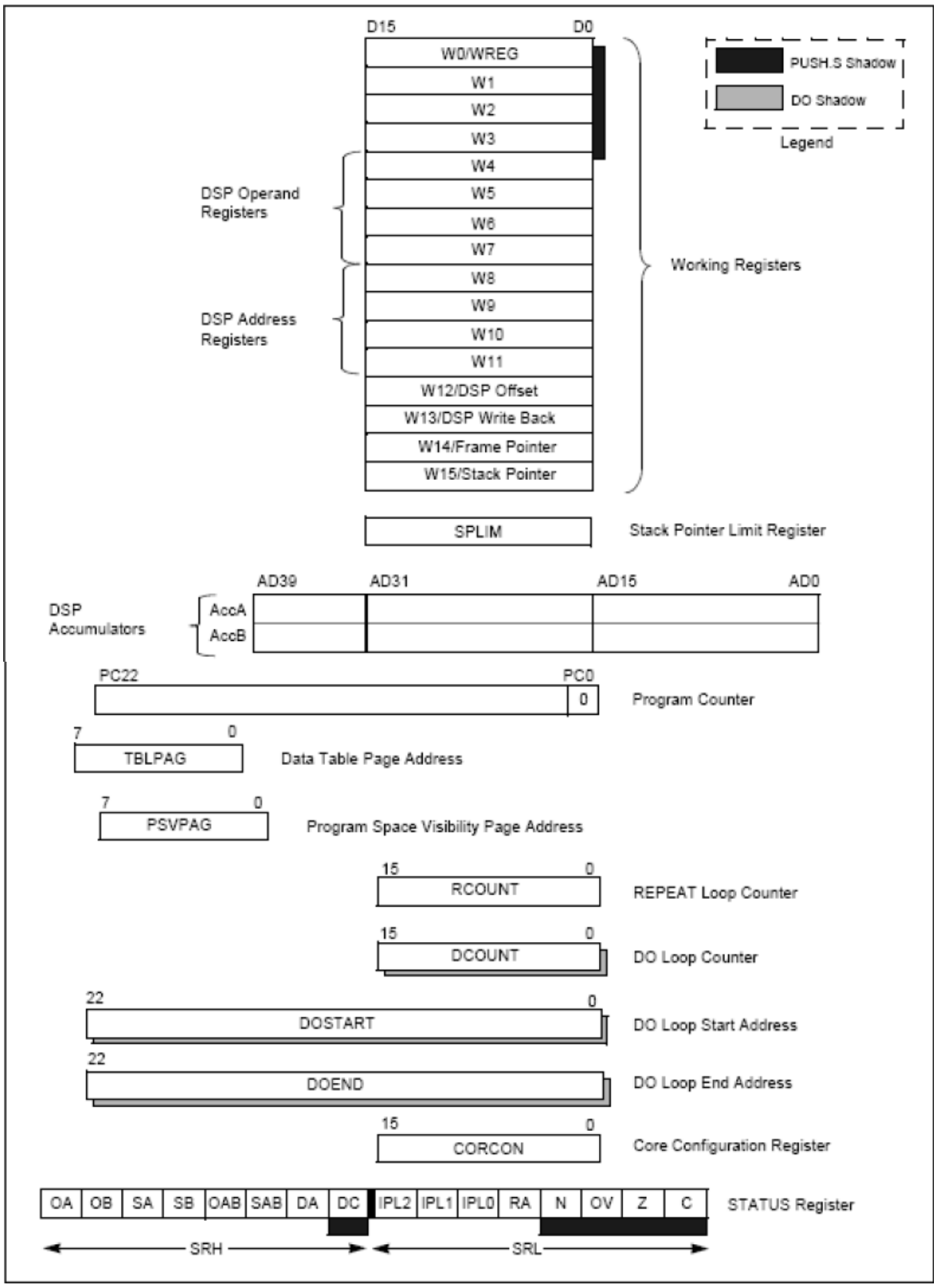


Figure III.1 : Le modèle des registres des dsPIC33

1. *Les registres à usage générale : W0-W15*
   - Registres W0-W3 : registres à usage général;
   - Registres W4-W7 : sont des registres d'opérandes DSP;
   - Registres W8-W11: sont des registres d'adresses DSP;
   - Registre W12 : Registre d'Offset DSP;
   - Registre W13 : Registre de restauration DSP;
   - Registre W14 : Registrer de pointeur de trame;
   - Registre W15 : Registre de pointeur de pile CPU;
2. *SPLIM* (Stack Pointer Limit) : C'est un registre de limite du pointeur de pile.
3. *ACCA / ACCB* (accumulateurs DSP) : Deux accumulateurs de 40 bits pour les opérations DSP.
4. *TBLPAGE* (Table Page Register) : Il est utilisé pour définir une zone de mots mémoire de 32K dans l'espace mémoire du programme.
5. *PSVPAG* (Program Space Visibility Page Address Register) : C'est un registre d'adresse de page de visibilité de l'espace de programme ; il est utilisé pour définir une page de mots mémoire de 16K dans l'espace mémoire du programme.
6. *RCOUNT* (Repeat Loop Counter) : ce registre compteur de boucle répétitive suit le nombre d'itérations d'une instruction qui doit être exécutée de manière répétitive jusqu'à ce que son fonctionnement a été effectuée.
7. DCOUNT (Do Loop Counter) : ce registre permet le comptage du nombre d'itérations d'une boucle DO;
8. DOSTART (DO Loop Start Address) : Ce registre contient l'adresse de départ d'une boucle DO particulière;
9. DOEND (Do Loop End Address) : par contre ce registre contient l'adresse de fin d'une boucle DO particulière
10. CORECON (Core Configuration Register): Ce registre contient les bits de configuration relatives pour gèrer certains types d'opérations du processeur.
11. STATUS (CPU Status Register) : constitué de plusieurs drapeaux (flags), dont la valeur indique l'état du CPU après une instruction donnée.

III. Outils de développements utilisés dans le portage

III.1. Compilateur C30

Le compilateur MPLAB C30 est un compilateur C destiné pour les microcontrôleurs 16 bits de la firme Microchip : PIC24, dsPIC30F et dsPIC33F. Il est entièrement compatible avec l'environnement de développement intégré (IDE) de Microchip permettant le débogage au niveau source avec le simulateur MPLAB SIM.

On peut citer quelques caractéristiques de ce compilateur :

1- Conforme avec la norme C ASCII : maths, conversion de données et bibliothèques mathématiques.
2- Génère les modules d'objet ré-adressable pour la réutilisation optimisée du code.
3- Il est optimisé et permet d'avoir des codes réduits jusqu'a 30% par rapport aux codes générés par d'autres compilateurs.
4- Permet de pointer le code et les données à des adresses absolues. [20]

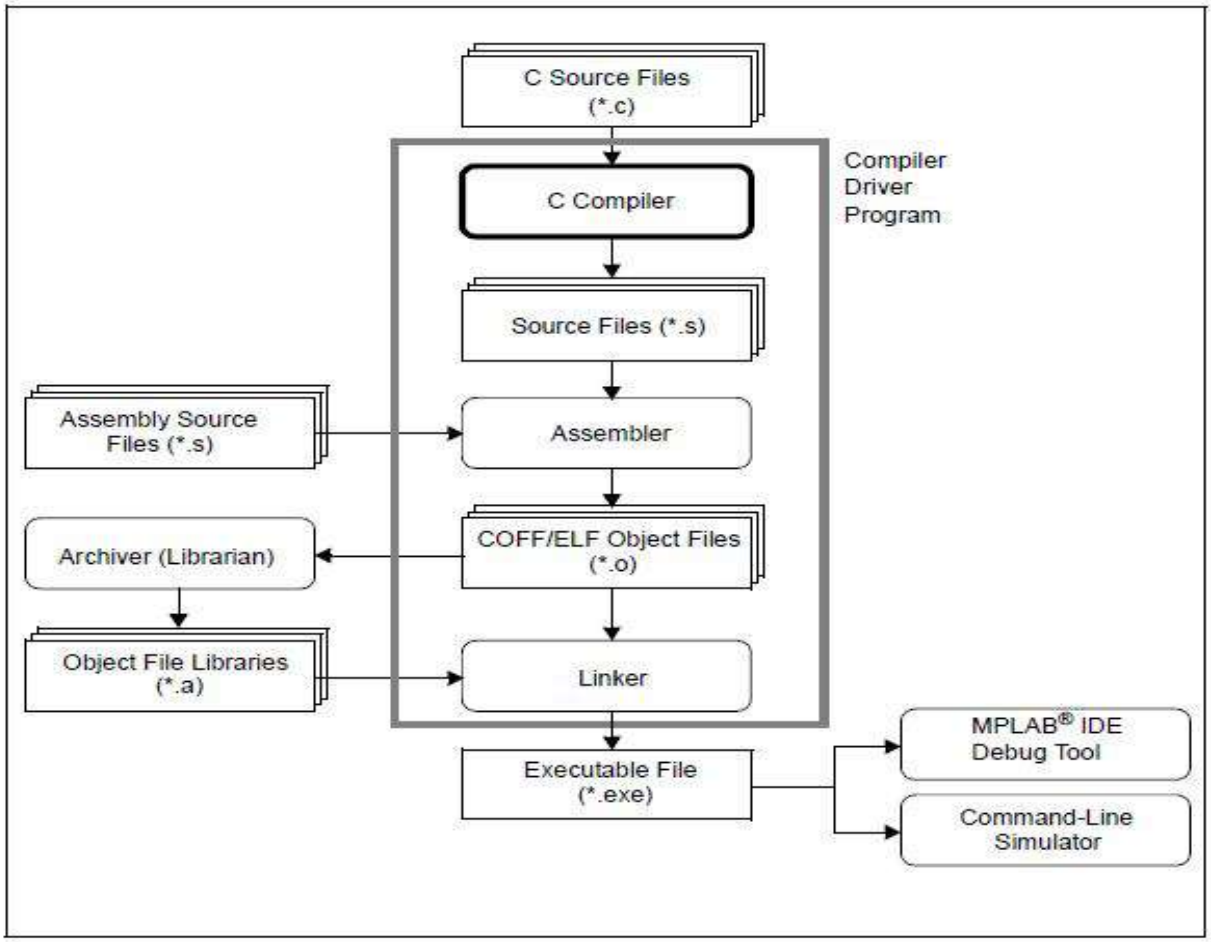


Figure III.2 : Structure de compilateur C30

III.2. Environnement de développement MPLAB

MPLAB (IDE) *Integrated Development Environment :* existe depuis de nombreuses années et il est devenu la plateforme de développement pour les microcontrôleurs PIC et dsPIC de Microchip. Cet environnement de développement est facile à mettre en œuvre, il est caractérisé par un support d'excellent niveau non seulement en raison des fonctions de son fichier d'aide, mais aussi grâce au forum en ligne et au support technique de Microchip.

MPLAB IDE comporte les modules suivants : l'assembleur MPAS, un programme de simulation MPSIM, un éditeur de lien (*linker*) MPLINK, un assistants (*wizard*) de projet et de paramétrage, un Project Manager, un initialiseur de dispositif visuel, des pilotes pour différents outils matériel.

IV. Organisation des répertoires et des fichiers du portage

L'organisation des fichiers du portage de µC/OS-II doit respecter une certaine forme imposée par la société Micrium afin de garantir une complication correcte de l'ensemble des fichiers du portage et pour garantir la portabilité du software vers d'autres environnements de développement. Les différents fichiers constituant le portage sont placés dans les répertoires suivants :

IV.1. µC/OS-II

- \Micrium\Software\µCOS-II\Source, Ce répertoire contient le code du RTOS µC/OS-II indépendant du type de processeur à utilisé.
- \Micrium\Software\µCOS-II\Ports\Microchip\PIC33FJ32\C30; Ce répertoire contient les fichiers spécifiques du processeur standard pour un portage µC/OS-II. Il contient les fichiers suivants: **os_cpu.h, os_cpu_a.s, os_cpu_c.c, os_cpu_util.s** et **os_dbg_c**

IV.2. µC/Probe

- \Micrium\Software\µC-Probe\Target\Communication\Generic\OS\µCOS-II, ce répertoire contient l'interface OS dépendante à la couche de communication de l'utilitaire µC/Probe. Si nous envisageons d'exécuter µC/Probe avec le RTOS, le fichier suivant devait être ajustés en conséquence: **probe_com_os.c**
- \Micrium\Software\µC-Probe\Target\Communication\Generic\RS-232\OS\µCOS-II, Ce répertoire contient le code d'interface OS dépendante à la partie spécifique RS-232

d'µC/Probe, en particulier le code nécessaire pour générer une tâche d'analyse de paquets Rx en option.

- \Micrium\Software\µC/Probe\Target\Communication\Generic\RS232\Ports\Microchip\dsPIC33\C30, Ce répertoire contient les fichiers port matériel uC / Probe pour le processeur dsPIC33: **probe_rs232c.c** et **probe_rs232_a.s.**
- \Micrium\Software\µC-Probe\Target\Communication\Generic\RS-232\Source, ce répertoire contient le code source indépendant de la cible de la couche de communication de µC/Probe RS-232. Plus précisément, ce répertoire contient les fichiers suivants: **probe_rs232.c** et **probe_rs232.h**
- \Micrium\Software\µC-Probe\Target\Communication\Generic\Source, ce répertoire contient le code source indépendant de la cible de la couche de communication µC/Probe. Plus précisément, ce répertoire contient les fichiers suivants: **probe_com.c** et **probe_com.h**
- \Micrium\Software\µC-Probe\Target\Plugins\µCOS-II, ce répertoire contient le code source indépendant de la cible pour µC/Probe. Plus précisément, ce répertoire contient les fichiers suivants: **os_probe.c** et **os_probe.h**

IV.3. µC/CPU

- \Micrium\Software\µC/CPU\, ce répertoire contient les fichiers indépendants du processeur dont le code permet de définir pour les sections critiques, ainsi que des définitions de macros pour les types de données utilisés dans la plupart des produits de Micrium, il comprend le fichier : **cpu_def.h**
- \Micrium\Software\µC/CPU\Microchip\PIC33FJ32\C30, ce répertoire contient les fichiers du portage du RTOS au processeur, ce répertoire comprend un seul fichier : **cpu.h**

IV.4. µC/LIB

- \Micrium\Software\µC-LIB, ce répertoire contient le fichier **lib_def.h**, qui fournit et définies les constantes utiles (comme DEF_TRUE et DEF_DISABLED) et un certains nombre de macros.

IV.5. Code d'application

- \Micrium\Software\Evalboards\Microchip\Explorer16\PIC33FJ256\MPLAB-C30\OS-Probe, ce répertoire contient le code source de l'application et les fichiers du projet MPLAB. Il contient l'ensemble des fichiers suivants:

app.c, app_cfg.h, app_hooks.c, app_probe.c, includes.h, os_cfg.h, OS-Probe.*, **Prob_com_cfg.h**

- ✓ *app.c:* contient le code d'aplication,
- ✓ *app_cfg.h:* contient des informations de configuration spécifiques à l'application telle que les priorités des tâches et la taille de la pile,
- ✓ *app_hooks.c:* contient le code hook d'utilisateur de µC/OS-II
- ✓ *app_probe.c:* contient le code d'initialisation du module µC/Probe.
- ✓ *os_cfg.h*: est le fichier de configuration de µC/OS-II
- ✓ *probe_com_cfg.h*: contient des constantes de configuration modifiables par l'utilisateur du module µC/Probe.

- \Micrium\Software\Evalboards\Microchip\Explorer16\PIC33FJ256\MPLAB_C30\BSP, ce répertoire contient les fichiers BSP de la carte d'évaluation Explorer16 et la dsPIC33 MCU. Alors que certaines parties du code dans ce répertoire peut travailler sur d'autres dérivés dsPIC33, des routines qui dépendent du matériel comme LED_ON () nécessitera la modification en fonction de la conception matérielle de notre carte d'évaluation. Ce répertoire contient les fichiers : **bsp.c**, **bsp.h**, **bsp_a.s**, **bsp_lcd.c**, **p33FJ32MC204.gld**, **p33FJ32MC204.h** et **p33FJ32MC204.inc.**

V. Portage du RTOS µC/OS-II au dsPic33FJ32MC204

V.1. Préparation des outils de développements

- Au départ nous avons commencé par télécharger les Software suivants :

  a- L'environnement de développement MPLAB, nous avons utilisé ***MPLAB IDE v8.02*** ;

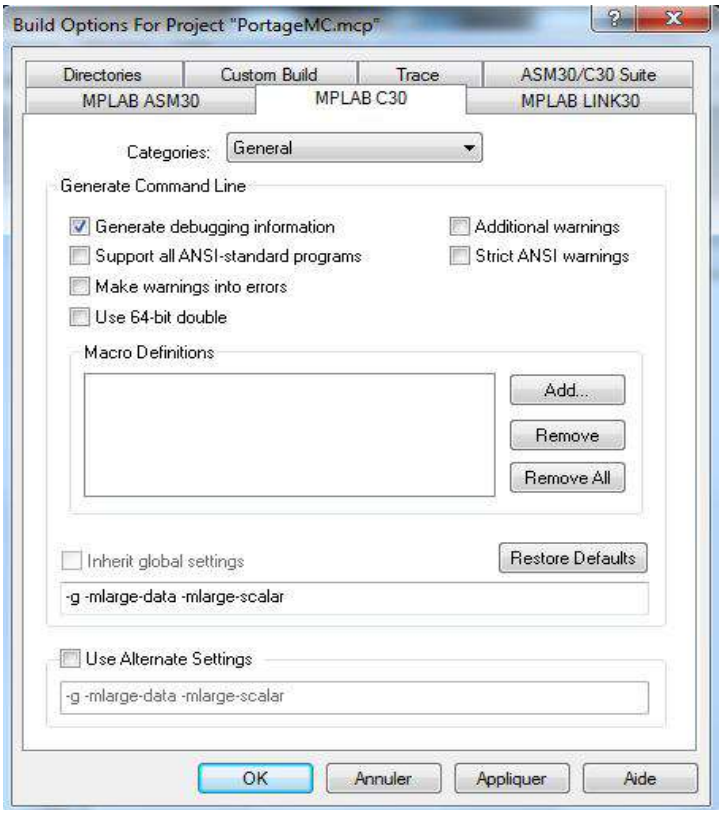
  b- Le compilateur C de Microchip ***C30***, nous avons utilisé la version ***v3.01*** ;

  c- Le code source du RTOS ***µC/OS-II***, nous avons utilisé la version ***v2.82.***

- Par la suite, nous avons décompressé l'ensemble des sources directement sous **C:**. Ainsi le répertoire ***µC/OS-II*** s'est logé dans la racine **C:**. Après nous avons installé MPLAB et le compilateur C30 en version gratuite. La version gratuite correspond à une version sans gestion des optimisations par le compilateur**..**
- Enfin nous avons créé sous MPLAB un projet intitulé ***PortageMC***

V.2. Création de projet PortageMC

Nous avons commencé par la configuration des options du projet créé sous MPLAB, en utilisant la boite de dialogue ***Project-> Build Option->Project*** ; nous avons précisé les chemins de sorties. Et par la suite les autres options de configuration montrées dans la Figure III.3.



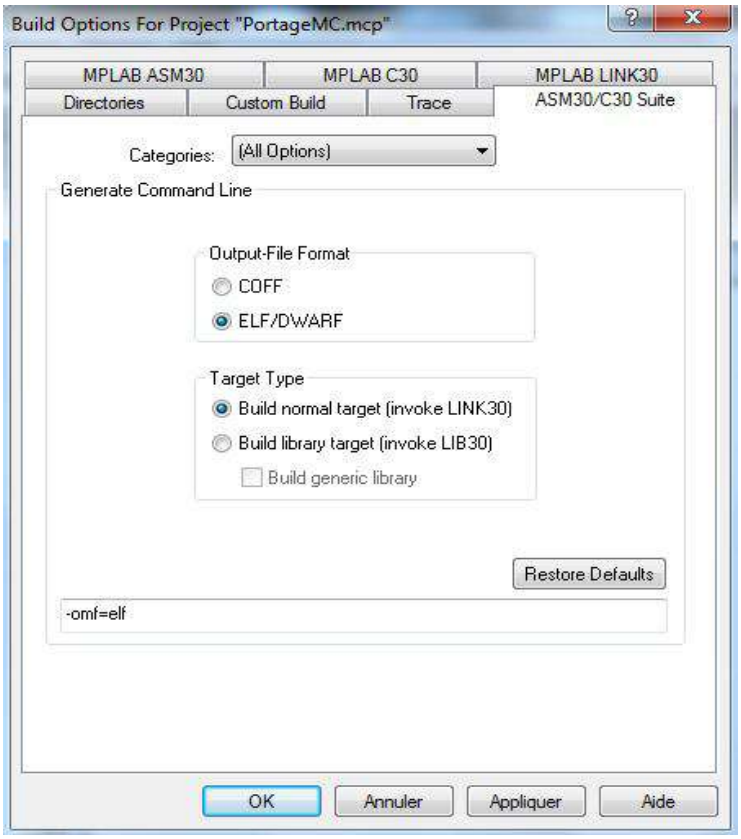


Figure III.3 : Les options de configuration MPLAB

Par la suite nous avons ajouté au projet un ensemble de fichier sources et header correspondants au RTOS µC/OS-II et au processeur cible le dsPic33FJ32MC204, la figure qui suit montre la structure du projet créer sous MPLAB:

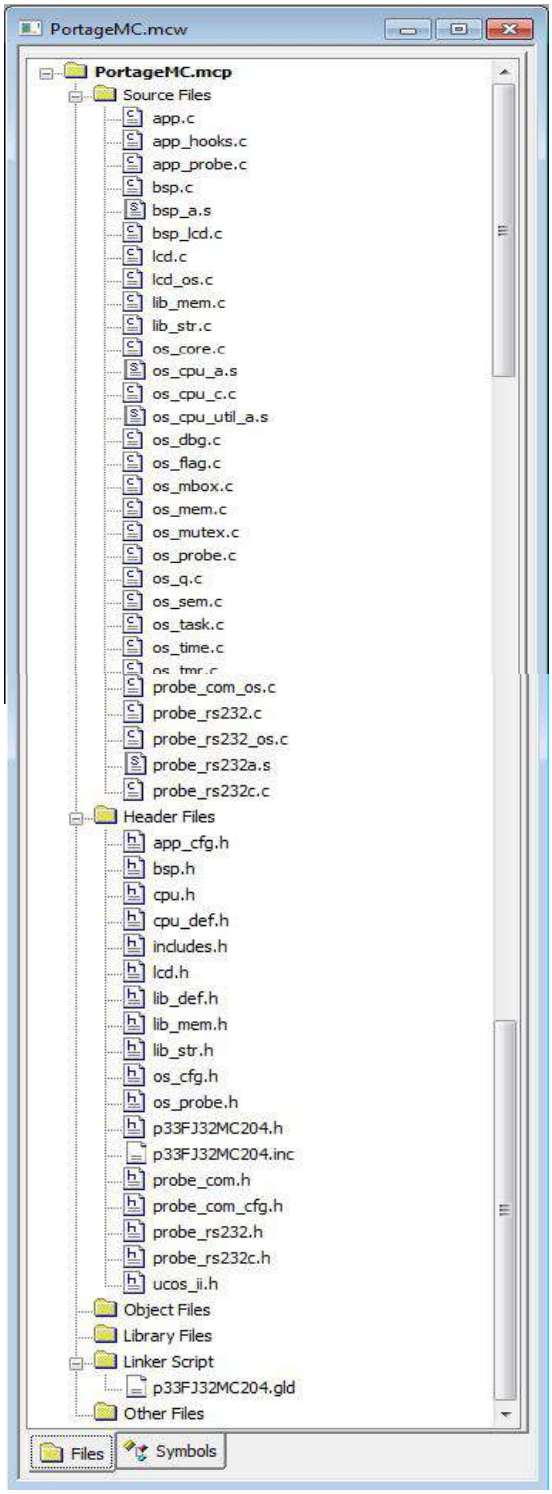


Figure III.4 : La structure du dossier

V.3. Description des fichiers du portage

V.3.1. OS_CPU.H: ce fichier contient : la définition d'un ensemble de constantes, de macros, et de typedefs relatif au processeur utilisé.

a- Les Macros Globals et Externs : les fonctions OS_CPU_GLOBALES et OS_CPU_EXT nous permet de déclarer les variables globales qui sont spécifiques pour ce portage. Ce portage ne contient aucune variable globale, mais les déclarations sont incluses en cas ou on a besoin de les ajouter au futur.

```
Globals and Externs
#ifdef OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
#define OS_CPU_EXT  extern
#endif
```

b- Définition des types de données:

```
Data Types
typedef unsigned char BOOLEAN;
typedef unsigned char INT8U;
typedef signed char INT8S;
typedef unsigned int INT16U;                    (1)
typedef signed int INT16S;
typedef unsigned long INT32U;
typedef signed long INT32S;
typedef float FP32;                             (2)
typedef long double FP64;
typedef unsigned int OS_STK;                    (3)
```

Les différents types de données utilisés avec µC/OS-II sont :

- (1) : int pour les données 16 bits;
- (2) : les données de types virgules flottantes 32 et 64 bits, ils sont inclus même si **µC/OS-II** n'utilise pas ce type de nombres;
- (3) : Toutes les piles des différentes taches sont déclarées en utilisant la fonction OS_STK qui est déclarée comme un type de donnée sur 16 bits pour les dsPIC33.

c- Sections critiques :

µC/OS-II à besoin de désactiver les interruptions pour accéder aux sections critiques du code, et les activer quand il termine le traitement. Il définit deux macros pour désactiver et activer les interruptions.

- OS_ENTER_CRITICAL () qui désactive les interruptions;
- OS_EXIT_CRITICAL () qui active les interruptions.

µC/OS-II définit trois méthodes pour désactiver les interruptions, ces méthodes sont choisis suivant le processeur et le compilateur utilisés, OS_CRITICAL_METHOD #3 est la méthode utilisée par **µC/OS-II** sous MPLAB avec les microcontroleurs dsPic33**.**

OS_CRITICAL_METHOD #3 implémente la macros OS_ENTER_CRITICAL() pour sauvegarde l'état du registre du CPU dans une variable. La macros OS_EXIT_CRITICAL() invoque une autre fonction pour restaurer l'état du registre du CPU.

```
OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
#define OS_CRITICAL_METHOD 3
#if OS_CRITICAL_METHOD == 3
#define OS_ENTER_CRITICAL() {cpu_sr = SR; SRbits.IPL = 7;}
#define OS_EXIT_CRITICAL()  {SR = cpu_sr;}
```

d- *Task Level Context Switch:*

Le changement de contexte au niveau des tache est effectué lorsque µC/OS-II invoque la macro OS_TASK_SW(). Parce que le changement de contexte est spécifique au processeur, OS_TASK_SW() a besoin d'exécuter une fonction en langage assembleur. Il faut noter que uC/S-II désactive toutes les interruptions avant d'appeler OSCtxSw().

```
Task Level Context Switch
#define OS_TASK_SW()     OSCtxSw()
```

V.3.2. OS_CPU_C.C: le portage de µC/OS-II exige l'écriture de l'ensemble de fonctions C suivantes :

OSInitHookBegin(), OSInitHookEnd(), OSTaskCreateHook(), OSTaskDelHook(), OSTaskIdleHook(), OSTaskStatHook(), OSTaskStkInit(), OSTaskSwHook(), OSTCBInitHook() et **OSTimeTickHook()**

Il faut noter qu'au départ nous devons mettre la constante # define OS_CPU_HOOKS_EN à 1 dans le fichier OS_CFG.H pour que le compilateur utilise les fonctions déclarées dans ce fichier.

- *OSInitHookBegin():* Cette fonction est appelée par OSInit() afin d'initialiser le RTOS µC/OS-II. Elle donne la possibilité d'ajouter le code d'initialisation supplémentaire spécifique au portage. Dans ce cas, il faut initialiser la variable globale OSTmrCtr (globale pour OS_CPU_C.C) qui est utilisée par le module OS_TMR.C si OS_TMR_EN est mis à 1, le registre SPLIM est réglé sur une valeur maximale de RAM.

OSInitHookBegin()

```
void OSInitHookBegin (void)
{
#if OS_TMR_EN > 0
   OSTmrCtr = 0;
#endif
   SPLIM = 0xFFFE;
 }
```

- *OSTaskCreateHook():* Cette fonction est appelée par OSTaskCreate() de uC/OS-II ou OSTaskCreateExt () quand une tâche est créée. OSTaskCreateHook () donne la possibilité d'ajouter un code spécifique pour le portage quand une tâche est créée. Dans notre cas, nous appelons une application hook défini dans app.c ou app_hooks.c. Le placement des applications hook est non critique et peut varier en fonction de projet.

OSInitHookEnd()

```
#if OS_CPU_HOOKS_EN > 0
void OSTaskCreateHook (OS_TCB *ptcb)
{ #if OS_APP_HOOKS_EN > 0
   App_TaskCreateHook(ptcb);
#else (void)ptcb;
#endif
} #endif
```

- *OSTaskStkInit():* Les taches sous µC/OS-II sont déclarer selon la forme du code suivante:

µC/OS-II Task:

```
void MyTask (void *p_arg)
{
        /* Do something with 'p_arg', optional */
         while (1) {
        /* Task body */
          OSTimeDlyHMSM(0, 0, 1, 0);
                    }
}
```

Ce code initialise la structure d'une pile pour la tâche qui est en cours de création. La tâche a reçu un argument optionnel «p_arg». La plupart des compilateurs passent un seul argument dans le registre W0 et donc, «p_arg est passé dans W0 lorsque la tâche est créée.

Le corps de la tâche devra appeler l'un des fonctions de l'OS Pend () ou OSTimedly() . En d'autres termes, une tâche doit toujours être en attente d'un événement. Un événement peut être la réception d'un signal ou un message d'une autre tâche ou l'activation d'une ISR. Un événement peut également être une temporisation.

- *OSTaskSwHook():* elle est appelée quand un changement de contexte se produit. Cette fonction permet au code de d'assurer d'autre fonctions telles que la mesure du temps d'exécution d'une tâche.

OSCtxSwHook()

```
#if OS_CPU_HOOKS_EN > 0
void OSTaskSwHook (void)
{
#if OS_APP_HOOKS_EN > 0
App_TaskSwHook();
#endif
}
#endif
```

- *OSTimeTickHook()*: cette fonction est appelée au début de OSTimeTick(). Elle permet au code d'être prolongé. Tout comme les autres hooks, cette fonction appelle une application hook défini par l'utilisateur placée dans les fichiers : app.c ou app_hooks.c. En outre, OSTimeTickHook() détermine s'il est le temps de mettre à jour les compteurs du RTOS µC/OS-II. Ceci est fait en signalant la tâche de timer.

```
void OSTimeTickHook (void)
{
#if OS_VIEW_MODULE > 0
   App_TimeTickHook();
#endif
#if OS_TMR_EN > 0
   OSTmrCtr++;
   if (OSTmrCtr >= (OS_TICKS_PER_SEC / OS_TMR_CFG_TICKS_PER_SEC)) {
      OSTmrCtr = 0;
      OSTmrSignal();
   }
#endi f
```

V.3.3. OS_CPU_A.S: Le portage µC/OS-II nécessite l'écriture de cinq fichiers en langage assembleur. Ces fonctions sont nécessaires parce qu'on ne peut pas sauvegarder / restaurer les registres à partir des fonctions en langage C. Les cinq fechiers sont :

OS_CPU_SR_Save(), OS_CPU_SR_Restore(), OSStartHighRdy(), OSCtxSw() et **OSIntCtxSw(),** ces derniers nous les avons importer du site de Micrium, nous nous contentons de les décrire dans ce qui suit.

Il convient de noter qu'avec les dsPIC33 on dispose de registres mappés en mémoire, les deux premières fonctions, OS_CPU_SR_Save () et OS_CPU_SR_Restore() ont été mis en œuvre sous forme de macros C. Étant donné que les fonctions restantes consistent à modifier le compteur de programme, L'instruction RETFIE va restaurer le PC à partir des données situées sur la pile des taches actuelles. C'est pourquoi une interruption est simulée lors de l'initialisation de la pile de tâche.

- *OSStartHighRdy():* est appelée par OSStart () pour commencer à exécuter la tâche la plus prioritaire qui a été créé avant d'appeler OSStart (). OSStart () définit OSTCBHighRdy pour pointer vers le OS_TCB de la tâche la plus prioritaire.

```
OSStartHighRdy()
_OSStartHighRdy:
   call _OSTaskSwHook              ; (1) Call user defined task switch hook
   mov #0x0001, w0                 ; (2) Set OSRunning to TRUE
   mov.b wreg, _OSRunning
  mov _OSTCBHighRdy, w0            ; (3) Resume Stack Pointer
  mov [w0], w15
  OS_REGS_RESTORE                  ; (4) Restore Context
  retfie                           ; (5) Run Task
```

- (1)Avant de commencer la tâche la plus prioritaire, nous appelons OSTaskSwHook () dans le cas d'une fonction de hook a été déclarée (voir OS_CPU_C.C).
- (2) Le drapeau (flag) OSRunning de µC/OS-II est défini à TRUE pour indiquer que µC/OS-II sera exécuté une fois que la première tâche est démarrée.
- (3) On obtient alors le pointeur de la tâche de haut de pile(task's top-of-stack) (été stockées par OSTaskCreate () ou OSTaskCreateExt ()).
- (4) Par la suite nous faisant apparaître tous les registres de la pile de la tâche. Cela se fait en appelant une macro en langage assembleur qui est déclarée dans OS_CPU_UTIL_A.S
- (5) En exécutant un return de l'instruction d'interruption, le dsPIC33 fait apparaître le registre PC et STATUS de la pile et ainsi, le dsPIC33 commence l'exécution du code de la tâche.

- *OSCtxSw():* Le code pour effectuer un changement de contexte au niveau de tâche est indiqué ci-dessous en pseudo-code. OSCtxSw () est appelée quand une tâche de priorité supérieure est prête à être exécuté par une autre tâche ou, lorsque la tâche en cours ne peut plus exécuter (par exemple, il appelle OSTimeDly(), OSSemPend() et le sémaphore n'est pas disponible, etc). Un changement de contexte au niveau d'une tache se produit lorsque µC/OS-II invoque la macro OS_TASK_SW () qui, dans le cas du port dsPIC33, correspond à l'appel OSCtxSw(). Normalement, il est idéal à utiliser une interruption logicielle ou TRAP pour entrer OSCtxSw(), cependant, car une interruption logicielle n'est pas disponible, le code doit ajuster la structure de pile pour simuler une interruption tels que l'instruction RETFIE peut être utilisé pour restaurer le PC, SR , et CORCON IPL3.

```
OS_TASK_SW()
OSCtxSw:
  Add SRL, IPL3 and the upper 7 bits of the return PC to simulate an ISR
  Save the CPU registers onto the old task's stack;
  OSTCBCur->OSTCBStkPtr = SP;
  OSTaskSwHook();
  OSPrioCur = OSPrioHighRdy;
  OSTCBCur = OSTCBHighRdy;
  SP = OSTCBHighRdy->OSTCBStkPtr;
```

- *OSIntCtxSw():* Lorsqu'une ISR (Interrupt Service Routine) est terminée, OSIntExit() est appelée pour déterminer si une tâche plus importante que la tâche interrompue doit s'exécuter. Si c'est le cas, OSIntExit() détermine quelle tâche est prête pour la prochaine exécution et appelle OSIntCtxSw() pour effectuer le changement de contexte réel pour cette tâche, si on nous voulons définir un point d'arrêt dans OSIntCtxSw(), on clique sur le point d'arrêt lors d'un changement de contexte au niveau de tâche (si OSIntCtxSw() était juste une étiquette dans OSCtxSw()).

```
OSIntCtxSw()
_OSIntCtxSw:
   call _OSTaskSwHook                    ; Call user defined task switch hook
    mov _OSTCBHighRdy, w1                   ; OSPrioCur = OSPrioHighRdy
    mov w1, _OSTCBCur
    mov.b _OSPrioHighRdy, wreg             ; OSTCBCur = OSTCBHighRdy
    mov.b wreg, _OSPrioCur
    mov [w1], w15                                  ; W15 (SP) = OSTCBHighRdy-
>OSTCBStkPtr
    OS_REGS_RESTORE                    ; Restore context of interrupted task
    retfie                                    ; Switch to the new context
```

V.3.4. OS_CPU_UTIL_A.S: OS_CPU_UTIL_A.S est un fichier en langage assembleur qui déclare deux macros pour sauvegarder le contexte de CPU. Ces deux macros sont appelés OS_REGS_SAVE et OS_REGS_RESTORE, et ils sont déclarés de la manière suivante :

```
.macro OS_REGS_SAVE
push.d  w0
push.d  w2
push.d  w4
push.d  w6
push.d  w8
push.d  w10
push.d  w12
push    w14
push    ACCAL
push    ACCAH
push    ACCAU
push    ACCBL
push    ACCBH
push    ACCBU
push    TBLPAG
push    PSVPAG
push    RCOUNT
push    DCOUNT
push    DOSTARTL
push    DOSTARTH
push    DOENDL
push    DOENDH
push    SR
push    CORCON
```

```
.macro OS_REGS_RESTORE
pop     CORCON
pop     SR
pop     DOENDH
pop     DOENDL
pop     DOSTARTH
pop     DOSTARTL
pop     DCOUNT
pop     RCOUNT
pop     PSVPAG
pop     TBLPAG
pop     ACCBU
pop     ACCBH
pop     ACCBL
pop     ACCAU
pop     ACCAH
pop     ACCAL
pop     w14
pop.d   w12
pop.d   w10
pop.d   w8
pop.d   w6
pop.d   w4
pop.d   w2
```

V.3.5. OS_DBG.C: c'est un fichier qui permet de faire extraire les informations de **µC/OS-II** et sa configuration**.** Spécifiquement**,** OS_DBG.C contient un nombre des constantes qui sont placées dans la ROM lequel le debugger peut lire et afficher. Ce fichier doit être inclus dans toutes les références.

V.4. Gestion des interruptions (Interrupt Handling)

Le dsPIC33 contient un ensemble d'interruptions et une table d'exception de vecteurs qui contient jusqu'à 54 entrées. Chacune de ces entrées pointent vers une interruption ou gestionnaire d'exception. Pour le RTOS µC/OS-II, chacun de ces gestionnaires d'interruption doit être écrit en langage assembleur.

Chaque fournisseur de chaîne d'outils a une méthode unique pour placer le code d'interruption et définissant la table de vecteurs d'entrées d'interruptions. La méthode suivante est employée par la chaine d'outils de Microchip C30 que nous avons utilisée.

V.5. Code d'application

V.5.1. APP.C, APP.H and APP_CFG.H

L'application est placé dans des fichiers appelés APP.C, APP.H et APP_CFG.H. Bien sûr, elle peut contenir beaucoup plus de fichiers selon les taches.

- APP.C où on place la fonction main (), elle peut être placée dans d'autres fichiers.
- APP.H contient la définitions des constantes, des macros, des prototypes,…etc qui sont spécifiques à notre application.
- APP_CFG.H contient la définition des constantes pour configurer l'application. On a placé les tailles de la pile de tache et les priorités des tâches et d'autres définition dans ce fichier. Cela nous permet de localiser les priorités des tâches et les tailles des piles dans un seul endroit.

```
main()
void main (void)
{
    INT8U  err;
    BSP_IntDisAll();                                                    (1)
    OSInit();                                                           (2)
    OSTaskCreateExt(AppStartTask,                                       (3)
                    (void *)0,
                    (OS_STK *)&AppStartTaskStk[TASK_STK_SIZE-1],
                    TASK_START_PRIO,
                    TASK_START_PRIO,
                    (OS_STK *)&AppStartTaskStk[0],
                    TASK_STK_SIZE,
                   (void *)0,
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);
#if OS_TASK_NAME_SIZE > 11
   OSTaskNameSet(TASK_START_PRIO, "Start Task", &err);                  (4)
#endif
       OSStart();                                                       (5)
}
```

- (1) Une fonction BSP appelée BSP_IntDisAll () permet de désactiver toutes les interruptions. ceci nous permet en général d'éviter que le contrôleur d'interruption interrompt l'exécution jusqu'à ce que l'application soit prête à s'éxécuter.
- (2) Comme pour toutes les applications de base du RTOS µC/OS-II, nous devons initialiser le noyau µC/OS-II en appelant la fonction OSInit ().
- (3) nous devons créer au moins une tâche. Dans ce cas, nous avons créé la tâche en utilisant la tâche prolongée qui crée l'appel. Ceci permet au µC/OS-II pour avoir plus d'informations sur la tâche.
- (4) Nous pouvons maintenant donner un nom à notre tâche.
- (5) Afin de commencer le multitâche, nous devons appeler OSStart ().

AppStartTask()

```
static void AppStartTask (void *p_arg)
{
   (void)p_arg;
   BSP_Init();                                                    (1)
#if OS_TASK_STAT_EN > 0
   OSStatInit();                                                  (2)
#endif
#if (uC_PROBE_OS_PLUGIN > 0) || (uC_PROBE_COM_MODULE > 0)
   AppProbeInit();
#endif
    AppTaskCreate();                                              (3)

    while (TRUE) {                                                (4)
       for (i = 0; i < 4; i++) {
            for (j = 1; j <= 8; j++) {
                 LED_On(j);
                OSTimeDlyHMSM(0,0,0,25)
          }
       }
}
```

- (1) Si nous avons décidé d'implémenter un BSP pour notre carte cible (la carte hardware de l'application), nous l'initialisons ici (dans la partie précédente du code).
- (2) Si nous avons activé la tâche statistique en mettant OS_TASK_STAT_EN dans le fichier OS_CFG.H à 1), alors, nous avons besoin de l'appeler ici. nous devons nous assurer que nous avons initialisé et activé l'impulsion d'horloge de µC/OS-II parce que OSStatInit () suppose la présence de tiks d'horloge. En d'autres termes, si l'impulsion d'ISR n'est pas actif quand nous appelons la fonctions OSStatInit (), notre application finissent comme une tâche inactive et µC/OS-II ne peut exécuter toutes les autres tâches.
- (3) A ce stade, nous pouvons créer des tâches supplémentaires. Nous avons décidé de mettre toutes nos initialisations tâches dans une fonction appelée AppTaskCreate ().
- (4) nous pouvons maintenant effectuer toutes les fonctions supplémentaires que nous voulons pour cette tâche. Nous avons décidé de basculer par exemple une LED en appelant une fonction appelée BSP LED_Toggle (). Il s'agit en fait de boucles imbriquées

qui effectuent la fonctionnalité de balayage des LED fournies dans beaucoup d'exemples d'applications.

V.5.2. INCLUDES.H : c'est le fichier maître à inclure au projet, il se trouve au sommet de tous les fichiers .C. Le seul inconvénient de ce fichier apparait lorsque ce dernier peut inclure des fichiers d'en-tête qui ne sont pas pertinents à l'actuel fichier .C en cours de compilation et le processus de compilation peut prendre plus de temps. on peut modifier INCLUDES.H pour ajouter nos propres fichiers d'en-tête, mais ces derniers doivent être ajoutés à la fin de la liste.

INCLUDES.H

```
#ifndef          INCLUDES_H
#define          INCLUDES_H

#include         <stdio.h>
#include         <string.h>
#include         <ctype.h>
#include         <stdlib.h>
#include         <p33FJ32MC204.h> /* Header file name varies with tool-chain */
#include         <cpu.h>
#include         <ucos_ii.h>
#include          <bsp.h>
#include          <lib_def.h>
#include          <lib_str.h>
#include          <lib_mem.h>
#if (uC_PROBE_OS_PLUGIN > 0)
#include          <os_probe.h>
#endif

#if (uC_PROBE_COM_MODULE > 0)
#include         <probe_com.h>
#if (PROBE_COM_METHOD_RS232 > 0)
#include         <probe_rs232.h>
#endif
#endif
#endif            /* End of File */
```

V.5.3. OS_CFG.H : Ce fichier permet au de configure le RTOS µC/OS-II pour une application donnée. La configuration consiste à spécifier combien de tâches notre application aura, le nombre de sémaphores à utilisé, le nombre de messages, ...etc.

V.6. Package BSP (Board Support Package)

Il est souvent convenable de créer un fichier Package (BSP) pour le matériel utilisé afin de simplifier la programmation des taches. Un fichier BSP pourrait nous permettre d'encapsuler les fonctionnalités suivantes :

- Initialisation du Timer (OS Ticker);
- Initialisation du module PLL;
- Configuration des gestionnaires d'ISR;
- Fonctions de contrôle d'un ensemble de LED reliés aux ports d'E/S;
- Lecture d'un ensemble de commutateurs reliés aux ports d'E/S;
- Mise en place des canaux de communication; ...etc.

Un BSP est composé de 2 fichiers de bases : BSP.C et BSP.H.

Par exemple, nous avons décidé de créer des fonctions de contrôle de LED en raison d'un grand nombre de cartes d'évaluation actuels qui sont équipées de ces composants, ceci est réaliser à l'aide des fonctions :

- *void LED_Init(void);*
- *void LED_On(INT8U led);*
- *void LED_Off(INT8U led);*
- *void LED_Toggle(INT8U led);*

Dans ce cas, les LEDs sont référencés "logiquement" au lieu de le faire physiquement. Lorsque nous rédigeons le BSP, nous déterminons quel DEL est la LED # 1, et celle qui est la LED # 2, et ainsi de suite. Lorsque nous souhaitons par exemple activer la LED n° 1, il suffit d'appeler la fonction **LED_On (1)**. Si nous voulons inverser l'état de la LED n° 2, il suffit d'appeler la fonction **LED_Toggle (2)**. En fait, nous devons associer des noms à nos LEDs en utilisant la derictive ***# define***. Nous pouvons ainsi préciser LED_Off (LED_PM), où LED_PM «pourrait» être définie comme LED # 2.

V.7. Compilation du portage sous MPLAB

La compilation du projet sous l'IDE MPLAB à poser beaucoup de problèmes au départ à cause d'un certain nombre de réglages internes des options de compilation qu'il fallait comprendre, par la suite des erreurs liées aux fichiers header sont apparus et que nous avons pus les réglées effectuant un ensemble de modifications et en revenant à l'organisation imposée par la société Micrium, d'autres erreurs été liées à la définition des type de variables, à la fin le projet à été compilé avec succès comme le montre la figure suivante :

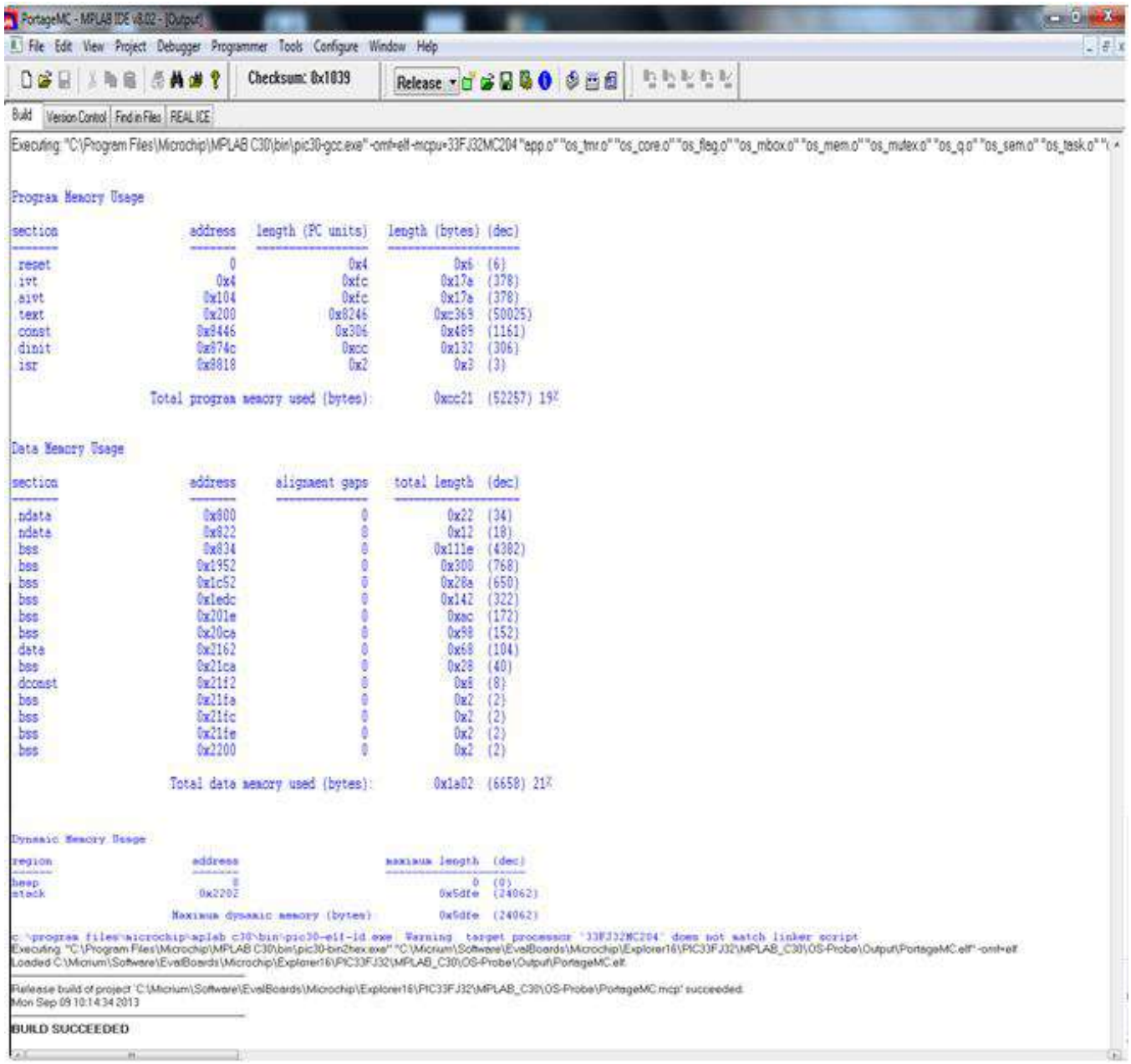


Figure III.5: Rapport de compilation du portage de µC/OS-II au dsPic33FJ32MC204

Il faut noter que la compilation du portage génère un grand nombre de fichiers de sorties dont le plus important est le fichier ***PortageMC.hex*** qui doit être programmé sur le dsPic33FJ32MC204, dans notre cas ce dernier à une empreinte mémoire de 11 K de mot instructions comme le montre la figure suivante :

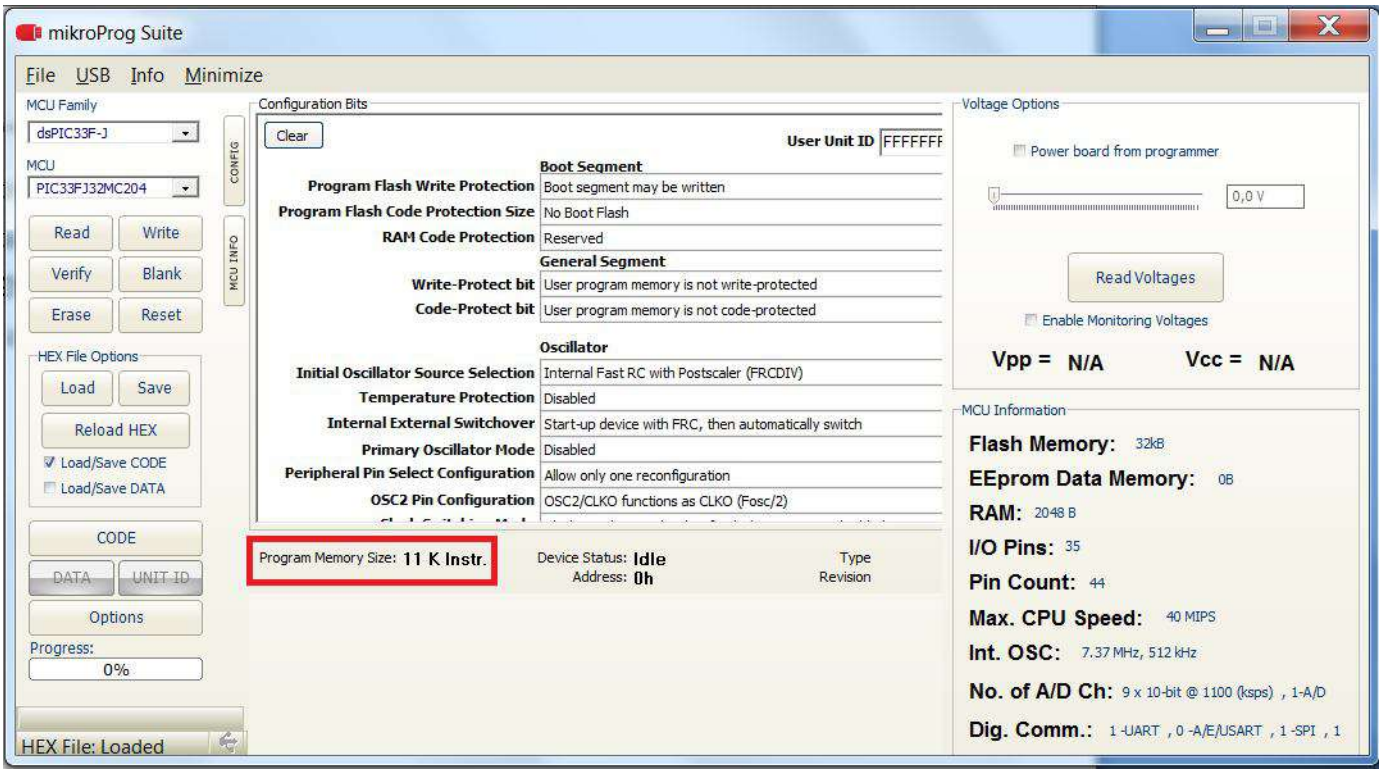


Figure III.6 : La taille de l'empreinte mémoire du portage

Afin de tester le portage réalisé nous avons pas pus le faire pratiquement sur une carte hardware car nous n'avons pas pus procurer une carte de développement ou d'évaluation à base d'un dsPic33FJ32MC204, mais nous avons procéder à quelques tests de simulation à l'aide du logiciel Proteus qui prend en charge le dsPic33FJ32MC204, ces tests se montrent assez satisfaisante du point de fonctionnent électrique mais ne permettent pas de faire une comparaison avec la programmation classique son RTOS du point de vue temps réel.

VI. Conclusion

Le portage du RTOS µC/OS-II a été réalisé avec succès, l'empreinte mémoire est d'une taille de 11 K de mots d'instructions (c-à-d 33 Koctets), la taille de cette empreinte peut etre réduit si on désactive un certains nombre de service (Mutex, boites aux lettres, ...etc) selon l'application envisagée.

Conclusion générale

Conclusion Générale

Dans ce travail nous nous somme familiariser à travailler avec le RTOS µC/OS-II de la société Micrium, ce RTOS offre des solutions intéressantes pour un bon nombre d'applications dans le domaine des systèmes embarqués à base de microprocesseurs, microcontrôleurs et DSP.

Ce travail nous a permis d'une part de manipuler des outils de développement pour microcontrôleurs dsPic tels que l'IDE MPLAB et le compilateur C30 de Microchip. D'autre part, la réalisation (le portage) nous a mis en face aux contraintes de la programmation temps réel ce qui nous a permis de comprendre beaucoup de mécanismes liés à l'exécution multitâches des processus.

Le travail réalisé est une modeste contribution au portage du RTOS µC/OS-II sur le microcontrôleurs 16 bits le dsPic33FJ32MC204, ce travail nécessite d'être améliorer par des tests pratiques sur une carte de développement ou d'évaluation afin d'optimiser le portage et de tester ses performances dans le monde réel, d'autre part la migration vers la version µC/OS-III s'impose de plus en plus puisque cette dernière version offre plus de performances et de capacités dans les domaines du temps réel.

Références bibliographique

- [1] Elham Firouzi « Les microcontrôleurs dans les systèmes embarqués » https://prof.hti.bfh.ch/fileadmin/home/fue1/Microcontroller_in_Embedded_Systems/Script/Manuscrit_uCES_V3_0a.pdf
- [2] H. ZAHDOURH et A. YOUBI « Le multitâche avec PIC18F452 », Mémoire D'ingénieur en électronique, Université des Sciences et Techniques d'Oran - 2007
- [3] Document pdf : Jacques-Olivier KLEIN « RTOS/Star12 UCOSII Guide de mise en route »
- [4] Site web : http://fr.wikipedia.org/wiki/Syst%C3%A8me_temps_r%C3%A9el
- [5] Site web : http://www.nicot31.fr/noyau-temps-reel
- [6] Site web: http://www.windriver.com
- [7] Site web: http://www.ucos-ii.com
- [8] Site web : http://pic24.ru/doku.php/en/osa/ref/intro
- [9] Site web : http://www.pumpkininc.com
- [10] Site web: http://www.lynuxworks.com
- [11] A. Hirchi « Simulation d'un RTOS sur un DATA LOGGER portatif dédié à l'AQM et à l'analyse des chutes ». Mémoire de Master en Génie Électrique, Université Dr. Yahia Fares de Médéa
- [12] Site web: http://micrium.com/rtos/ucosii/overview/
- [13] Jean J. Labrosse « MicroC/OS-II The Real-Time Kernel Second Edition »
- [14] Site web: http://fr.wikipedia.org/wiki/Nios_II
- [15] Site web: http://fr.wikipedia.org/wiki/Microcontr%C3%B4leur_PIC#PIC32
- [16] Site web: http://en.wikipedia.org/wiki/Fujitsu_FR
- [17] Site web: http://fr.wikipedia.org/wiki/ARM_Cortex-M
- [18] Site web: http://fr.wikipedia.org/wiki/Microblaze
- [19] Document pdf: dsPIC33FJ32MC202/204 and dsPIC33FJ16MC304 Data Sheet High-Performance, 16-bit Digital Signal Controllers
- [20] MicroChip « MPLAB® C30 C Compiler » http://ww1.microchip.com/downloads/en/devicedoc/51432c.pdf

Annexe

1- APP.C

```
#include <includes.h>
/*
*********************************************************************************************
*                                            CONSTANTS
*********************************************************************************************
*/
/*
*********************************************************************************************
*                                            VARIABLES
*********************************************************************************************
*/
OS_STK  AppStartTaskStk[APP_TASK_START_STK_SIZE];
OS_STK  AppLCDTaskStk[APP_TASK_LCD_STK_SIZE];

/*
*********************************************************************************************
*                                      FUNCTION PROTOTYPES
*********************************************************************************************
*/
static  void  AppStartTask(void *p_arg);
static  void  AppLCDTask(void *p_arg);
static  void  AppTaskCreate(void);
#if (uC_PROBE_OS_PLUGIN > 0) || (uC_PROBE_COM_MODULE > 0)
extern  void  AppProbeInit(void);
#endif
/*
*********************************************************************************************
*                                            main()
*
* Description : This is the standard entry point for C code.
* Arguments   : none
*********************************************************************************************
*/
CPU_INT16S  main (void)
{
   CPU_INT08U  err;
   BSP_IntDisAll();                                    /* Disable all interrupts
until we are ready to accept them */
   OSInit();                                           /* Initialize "µC/OS-II,
The Real-Time Kernel"            */
```

2- APP-CFG.H

```
/*
*********************************************************************************************
*                                       uC/OS-II Application Configuration
*
*                    DO NOT DELETE THIS FILE, IT IS REQUIRED FOR OS_VER > 2.80
*
*                                     CHANGE SETTINGS ACCORDINGLY
*
*
* File : app_cfg.h
* By   : Eric Shufro
*********************************************************************************************
*/
#ifndef  APP_CFG_H
#define  APP_CFG_H
/*
*********************************************************************************************
*                                            INCLUDES
*********************************************************************************************
*/
#include <lib_def.h>
/*
*********************************************************************************************
*                                       ADDITIONAL uC/MODULE ENABLES
*********************************************************************************************
*/
#define  LIB_STR_CFG_FP_EN              DEF_DISABLED
#define  uC_PROBE_OS_PLUGIN               DEF_ENABLED
#define  uC_PROBE_COM_MODULE                DEF_ENABLED
/*
*********************************************************************************************
*                                       TASK PRIORITIES
*********************************************************************************************
*/
#define  APP_TASK_START_PRIO                0                /* Lower numbers are of higher priority                 */
```

3- APP_HOOKS.C

```
#include  <includes.h>
/*
*********************************************************************
*********************************************
*                                App_TaskCreateHook()
*
* Description: This function is called when a task is created.
*
* Arguments  : ptcb   is a pointer to the task control block of the task being
created.
*
* Note(s)    : 1) Interrupts are disabled during this call.
*********************************************************************
*********************************************
*/
#if (OS_APP_HOOKS_EN > 0)
void  App_TaskCreateHook (OS_TCB *ptcb)
{
#if (uC_PROBE_OS_PLUGIN > 0) && (OS_PROBE_HOOKS_EN > 0)
   OSProbe_TaskCreateHook(ptcb);
#else
   (void)ptcb;
#endif
}
/*
*********************************************************************
*********************************************
*                                App_TaskDelHook()
*
* Description: This function is called when a task is deleted.
*
* Arguments  : ptcb   is a pointer to the task control block of the task being
deleted.
*
* Note(s)    : 1) Interrupts are disabled during this call.
*********************************************************************
*********************************************
*/
void  App_TaskDelHook (OS_TCB *ptcb)
{
   (void)ptcb;
}
/*
*********************************************************************
*********************************************
*
*
```

4- APP_PROBE.C

```
*********************************************************************************************
*                                            INCLUDES
*********************************************************************************************
*/
#include <includes.h>
/*
*********************************************************************************************
*                                            GLOBAL DATA
*********************************************************************************************
*/
#if (uC_PROBE_OS_PLUGIN    > 0) && \
   (uC_PROBE_COM_MODULE   > 0) && \
   (PROBE_COM_STAT_EN     > 0)
#if (OS_PROBE_USE_FP       > 0)
static  CPU_FP32      Probe_ComRxPktSpd;
static  CPU_FP32      Probe_ComTxPktSpd;
static  CPU_FP32      Probe_ComTxSymSpd;
static  CPU_FP32      Probe_ComTxSymByteSpd;
#else
static  CPU_INT32U    Probe_ComRxPktSpd;
static  CPU_INT32U    Probe_ComTxPktSpd;
static  CPU_INT32U    Probe_ComTxSymSpd;
static  CPU_INT32U    Probe_ComTxSymByteSpd;
#endif
static  CPU_INT32U    Probe_ComRxPktLast;
static  CPU_INT32U    Probe_ComTxPktLast;
static  CPU_INT32U    Probe_ComTxSymLast;
static  CPU_INT32U    Probe_ComTxSymByteLast;

static  CPU_INT32U    Probe_ComCtrLast;
#endif

#if (uC_PROBE_OS_PLUGIN > 0)
static  CPU_INT32U   Probe_Counts;
#endif

/*
*********************************************************************************************
*********************************************
*********************************************************************************************
*********************************************
```

5- INCLUDES.H

```
#ifndef    INCLUDES_H
#define    INCLUDES_H

#include   <stdio.h>
#include   <string.h>
#include   <ctype.h>
#include   <stdlib.h>

#include   <p33FJ32MC204.h>
#include   <cpu.h>
#include   <ucos_ii.h>

#include   <lib_def.h>
#include   <lib_str.h>
#include   <lib_mem.h>

#include   <bsp.h>
#include   <lcd.h>

#if (uC_PROBE_OS_PLUGIN > 0)
#include  <os_probe.h>
#endif

#if (uC_PROBE_COM_MODULE > 0)
#include  <probe_com.h>

#if (PROBE_COM_METHOD_RS232 > 0)
#include  <probe_rs232.h>
#endif
#endif

#endif                                          /* End of File
*/
```

6- OS_CFG.H

```
#ifndef OS_CFG_H
#define OS_CFG_H
                                /* ---------------------- MISCELLANEOUS ------
---------------- */
#define OS_APP_HOOKS_EN          1    /* Application-defined hooks
are called from the uC/OS-II hooks */
#define OS_ARG_CHK_EN            0    /* Enable (1) or Disable (0)
argument checking                */
#define OS_CPU_HOOKS_EN          1    /* uC/OS-II hooks are found
in the processor port files      */

#define OS_DEBUG_EN              1    /* Enable(1) debug variables
*/
#define OS_EVENT_MULTI_EN        1    /* Include code for
OSEventPendMulti()               */
#define OS_EVENT_NAME_SIZE       16    /* Determine the size of the
name of a Sem, Mutex, Mbox or Q    */

#define OS_LOWEST_PRIO           63    /* Defines the lowest priority
that can be assigned ...         */
                                /* ... MUST NEVER be higher than 254!
*/

#define OS_MAX_EVENTS            20    /* Max. number of event
control blocks in your application      */
#define OS_MAX_FLAGS             5    /* Max. number of Event Flag
Groups    in your application      */
#define OS_MAX_MEM_PART          5    /* Max. number of memory
partitions                       */
#define OS_MAX_QS                4    /* Max. number of queue control
blocks in your application      */
#define OS_MAX_TASKS             20    /* Max. number of tasks in your
application, MUST be >= 2       */

#define OS_SCHED_LOCK_EN         1    /* Include code for
OSSchedLock() and OSSchedUnlock()          */

#define OS_TICK_STEP_EN          1    /* Enable tick stepping feature
for uC/OS-View                   */
#define OS_TICKS_PER_SEC         1000    /* Set the number of ticks in
one second                       */


                                /* ---------------------- TASK STACK SIZE -------
--------------- */
```

7- PROBE_Com_CFG.H

```
/*
*********************************************************************
*********************************************
*                          CHOOSE COMMUNICATION METHOD
*********************************************************************
*********************************************
*/
#define  PROBE_COM_METHOD_RS232          DEF_TRUE
/*
*********************************************************************
*********************************************
*                       CONFIGURE GENERAL COMMUNICATION
PARAMETERS
*********************************************************************
*********************************************
*/

#define  PROBE_COM_RX_MAX_SIZE            64
#define  PROBE_COM_TX_MAX_SIZE            64

#define  PROBE_COM_SUPPORT_WR            DEF_TRUE
#define  PROBE_COM_SUPPORT_STR           DEF_TRUE

#if    (PROBE_COM_SUPPORT_STR  == DEF_TRUE)
/* If strings are supported                          */
#define  PROBE_COM_STR_BUF_SIZE           128                /*
(a) Set size of string buffer                    */
#endif

/*
*********************************************************************
*********************************************
*                       CONFIGURE STATISTICS AND COUNTERS
*********************************************************************
*********************************************
*/

#define  PROBE_COM_STAT_EN              DEF_ENABLED

/*
*********************************************************************
*********************************************
*                       CONFIGURE RS-232 SPECIFIC PARAMETERS
*********************************************************************
```