Université de Liège

Conception d'une plateforme "temps réel" enfouie

Travail de fin d'études réalisé par Hugues Smeets en vue de l'obtention du grade de licencié en informatique.

Année académique 2000-2001

Remerciements

Je tiens à remercier M^r Bernard Boigelot, M^{me} Suzanne Hostens et M^r Willy Smeets pour leurs conseils et la relecture de ce travail.

Table des matières

Chapitre 1

Introduction

1.1 Préliminaires

Par définition, on dit d'un système informatique qu'il est enfoui si il est un composant d'un système plus complexe. Historiquement, on a commencé à parler de tels systèmes quand on a eu besoin de placer un système informatique à bord d'un véhicule. Le bon fonctionnement de celui-ci nécessitait la réalisation de calculs en un temps très court, ces calculs étant entre autres basés sur des informations collectées par des capteurs. Par exemple, il s'agissait de la commande du système d'injection d'un moteur automobile ou d'un système de guidage de fusée. De cela découle le terme anglo-saxon d'*embedded* qui a été traduit au début par "embarqué". Ces systèmes étaient, de fait, *embarqués*.

De nos jours, de nombreux appareils utilisent de tels systèmes alors que leur fonction de base ne le nécessite pas d'office (frigo, téléviseur, four). Cependant, l'évolution des marchés de masse tels que l'électroménager et la HI-FI a amené les constructeurs à différencier leurs produits de ceux de leurs concurrents en y ajoutant des fonctions facultatives. C'est ce qu'on appelle des services ajoutés. Si on prend l'exemple d'un frigo, la conservation des aliments au frais est le service de base et son emballage est un service nécessaire (voir [6]). Si, en plus, il donne l'heure et affiche sa température interne, ou si il permet de gérer une base de données de son contenu, on a affaire à un service ajouté.
Dans d'autres cas, la conception de certains produits ne peut se faire qu'à l'aide de systèmes enfouis (téléphones portables et lecteurs portatifs de média, par exemple).
Des secteurs tels que l'électroménager, la HI-FI, la robotique, la mesure (scientifique, industrielle et médicale), les transports et l'informatique ne peuvent aujourd'hui plus se passer de ces systèmes, pourtant si discrets !
Cela explique pourquoi le terme "embarqué" a été remplacé par "enfoui".

1.2 Objectifs

L'objectif principal de ce travail est de fournir aux concepteurs et aux programmeurs d'une application enfouie des mécanismes qui sont habituellement réservés à des systèmes plus complexes, malgré les limitations très importantes qui caractérisent le matériel et le logiciel dans le contexte du développement de systèmes enfouis.

La modularité du système sera une de ses caractéristiques majeures. Les concepteurs pourront, au niveau matériel, choisir les modules dont ils auront besoin et les assembler de manière très facile. Si les modules existants ne fournissent pas toutes les fonctions désirées, ils pourront en développer de nouveaux en respectant quelques règles de conception simples de manière à les rendre compatibles avec les modules qui leurs sont fournis. Ces nouveaux modules seront ajoutés à la collection des modules existants et pourront être réutilisés à volonté pour une autre application.

Nous fournirons les modules matériels suivants :

1. un module d'alimentation qui fournira à l'ensemble des modules une tension stabilisée,
2. un module doté d'un microcontrôleur,
3. un module de télécommande basée sur une transmission unidirectionnelle asynchrone.

Un principe de modularité similaire sera utilisé pour le développement du logiciel. En effet, lorsqu'on développe le programme qui réside au cœur d'une telle réalisation, on constate qu'on réécrit chaque fois le code lié à des fonctions génériques telles que la prise en compte de l'appui sur certaines touches, l'affichage d'informations sur un affichage LCD[1], etc. . .L'implémentation de ces fonctions est noyée dans le code spécifique à l'application et n'est donc pratiquement pas réutilisable. Ce type de programmation, inextricable, est peu lisible, difficile à déboguer et à maintenir. De là vient à l'esprit l'idée d'appuyer le développement d'une application particulière sur l'utilisation d'un système d'exploitation générique et modulaire, adapté bien sûr aux faibles ressources matérielles dont on dispose. Des modules fournissant les services essentiels d'un système d'exploitation (malgré le cadre limité des

[1]Liquid Cristal Display ou afficheur à cristaux liquides.

systèmes enfouis) seront mis à la disposition des programmeurs. Le langage utilisé sera le C pour permettre une réutilisation, un portage et une lisibilité plus faciles du code. En général, les fabricants fournissent des modules qui permettent d'utiliser les périphériques, spécifiques, de leurs circuits. Nous ne fournirons pas de module destiné à l'interfaçage avec du matériel mais uniquement des modules implémentant des fonctions génériques, susceptibles d'être utilisées dans toutes les applications.

Ces fonctions génériques sont destinées à :

1. permettre l'exécution simultanée (en apparence) de plusieurs tâches avec attribution à chacune d'elles d'une priorité unique,

2. faciliter l'élaboration de mécanismes de communication entre les tâches à l'aide de sémaphores.

   Le but de ce travail est donc de décrire la réalisation d'une plateforme "temps réel" enfouie générique et modulaire.

1.3 Les caractéristiques spécifiques d'un système "temps réel" enfoui

Le terme "temps réel" veut dire qu'on garantit de manière quantitative le temps de réponse maximal à des événements matériels. Par "enfoui", nous voulons dire que notre système est invisible, caché à l'utilisateur final.

Par rapport à un système informatique généraliste, un système enfoui présente certaines caractéristiques particulières :

- l'interface utilisateur est spécifique à chaque application (quelques touches, un écran à cristaux liquides de deux lignes de 16 caractères par exemple),

- les ressources matérielles sont faibles (processeur 8 bits, quelques octets de RAM[2]),

- le système doit répondre à des événements extérieurs dans un délai court et imposé,

- le système doit avoir une fiabilité maximale (un système enfoui défaillant dans un appareil médical ou un vaisseau spatial peut entraî-

[2] Random Access Memory : mémoire volatile, qui perd son contenu lors d'une disparition de son alimentation.

ner la mort d'êtres humains, contrairement à un bogue[3] dans un programme à usage bureautique).

On voit donc apparaître des contraintes matérielles dont nous devrons tenir compte pour réaliser notre travail.

1.4 Contraintes matérielles

Pour des raisons évidentes, un système enfoui se doit d'être de taille la plus réduite possible, ce qui implique pratiquement l'utilisation d'un ou plusieurs microcontrôleurs. Un microcontrôleur est un circuit électronique qui regroupe sur une seule puce différents sous-ensembles : un microprocesseur, de la mémoire vive (RAM) qui permettra de stocker les variables, de la mémoire morte (ROM[4]) qui contiendra le programme du système ainsi que les données constantes, et des périphériques (interface sérielle, timer, etc...). Le corollaire de cette petite taille a évidemment un désavantage : le microprocesseur est relativement peu puissant, la taille de la RAM est de l'ordre d'une centaine d'octets, celle de la ROM de 1000 octets et les périphériques ne fournissent que des fonctions rudimentaires.

Une autre contrainte est imposée à ce type de systèmes : les coûts doivent être minimisés d'une telle façon qu'on puisse rendre "intelligent" un système qui au départ ne l'est pas, sans augmenter de manière sensible son prix de revient (par exemple un interrupteur 220V classique serait remplacé par une version télécommandable par ondes radio ou à reconnaissance vocale ; le prix de revient du second serait seulement supérieur d'une dizaine de pour cent à celui du premier). L'évaluation du coût de différentes solutions possibles sera présentée plus loin.

1.5 Le plan du travail

Dans un premier temps, nous allons réaliser une étude comparative des coûts de différentes solutions matérielles possibles (choix d'un microcontrôleur). Puis, un fois le choix effectué, nous décrirons la réalisation matérielle de la plateforme "temps réel".

Ensuite, nous analyserons plusieurs architectures logicielles pour systèmes enfouis, telles qu'elles apparaissent dans la littérature. Nous verrons

[3]Le mot "bogue" est masculin quand il désigne une erreur dans un programme informatique. Vu la confusion fréquente avec l'enveloppe de la châtaigne, le féminin est aussi toléré.

[4]Read Only Memory : mémoire non volatile qui ne peut être que lue.

alors quelles sont les contraintes imposées par le choix réalisé d'après l'étude de coûts et nous exposerons des solutions spécifiques à ce choix.

Le logiciel sera étudié en détails et illustré par un exemple d'utilisation théorique.

Pour terminer, un exemple d'application pratique sera exposé de manière à mettre en lumière les avantages et les limitations de notre solution logicielle.

Un dernier chapitre sera consacré à la conclusion finale.

Chapitre 2

Le matériel

2.1 Evaluation du coût du matériel

2.1.1 Hypothèses

Nous sommes amenés à faire certaines hypothèses pour évaluer la faisabilité (possibilité de commercialisation) de ce projet :

- seul le prix des composants sera déterminant ; en effet, le prix du circuit imprimé, compte tenu des économies d'échelle dans le cas d'une utilisation commerciale, sera négligeable ; le prix du logiciel sera lui aussi négligé,

- bien que le prix de ces composants achetés en grande quantité (production en série) diffère fortement du prix à l'unité (information facilement accessible), on pourra comparer les différentes solutions car on peut supposer que le *rapport* entre les prix de celles-ci sera sensiblement le même dans les deux cas,

- nous fixons arbitrairement le prix maximal des composants du système de base à 500 FB ; nous écarterons d'office les solutions plus chères. Comme on le verra, ce point sera nuancé plus tard.

2.1.2 Le cahier des charges du "système de base"

Ce système sera le siège de l'exécution de notre "système d'exploitation". Des modules annexes pourrons s'y connecter par l'intermédiaire d'un bus au protocole simple et peu demandeur de ressources matérielles (I^2C de la firme Philips ou SPI de Motorola par exemple). Ces modules, spécifiques à une application particulière ne font en principe pas l'objet de ce travail. Néanmoins, le développement de certains modules est prévu à des fins d'illustration du fonctionnement du système d'exploitation. Le débogage pourrait aussi être facilité par l'utilisation, par exemple, d'un module d'affichage.

Avant de pouvoir déterminer le coût des composants du système, il faut d'abord avoir une idée, ne fût-ce que vague, des sous-ensembles fonctionnels qu'il devra comporter (pour chaque sous-ensemble, un prix indicatif sera donné, sauf pour le microcontrôleur dont le cas sera traité plus tard) :

- une alimentation qui fournira, à partir de la tension du secteur, une tension constante de 5V (prix approximatif : 60 FB[1]),

- le microcontrôleur,

- un dispositif de surveillance de la tension d'alimentation qui initialise le système si la tension d'alimentation quitte sa valeur de consigne. Ce dispositif est *fondamental* car on ne peut permettre au système de se comporter de façon erratique lors d'une coupure de l'alimentation. Cette coupure étant progressive, le microcontrôleur sous-alimenté pourrait se mettre à exécuter des instructions de manière aléatoire, risquant par exemple d'altérer le contenu d'une EEPROM[2] stockant des paramètres importants (circuit de surveillance : TL7705 (de Texas Instruments, voir [11]), par exemple : 30 FB).

- une interface de programmation *in-situ* du microcontrôleur qui permet de reprogrammer celui-ci, sans l'extraire, à partir d'un ordinateur sur lequel se fait le développement de l'application (un connecteur et quelque résistances : 30 FB),

- une interface de/vers les modules annexes (prix arbitraire : 30 FB),

- une mémoire non-volatile réinscriptible (EEPROM) pour mémoriser les paramètres du système (*setup*) ; (le circuit 24C16, par exemple : 100 FB). Notons que certains microcontrôleurs possèdent déjà une EEPROM de capacité suffisante pour la majorité des applications et que celles-ci n'auront donc pas besoin de ce circuit.

On voit que le coût de ces sous-ensembles (sans le contrôleur) est d'environ 250 FB. Pour fixer les idées, voyons sur la figure 2.1 un synoptique de ce qui vient d'être décrit.

[1] on suppose l'utilisation d'un module secteur dont on ne compte pas le prix ; l'alimentation est constituée d'un régulateur 5V (7805), d'un régulateur 12V pour une éventuelle tension de programmation et de quelques résistances et capacités.

[2] Electrically Erasable Programmable Read Only Memory : mémoire non volatile qui peut être reprogrammée électriquement.

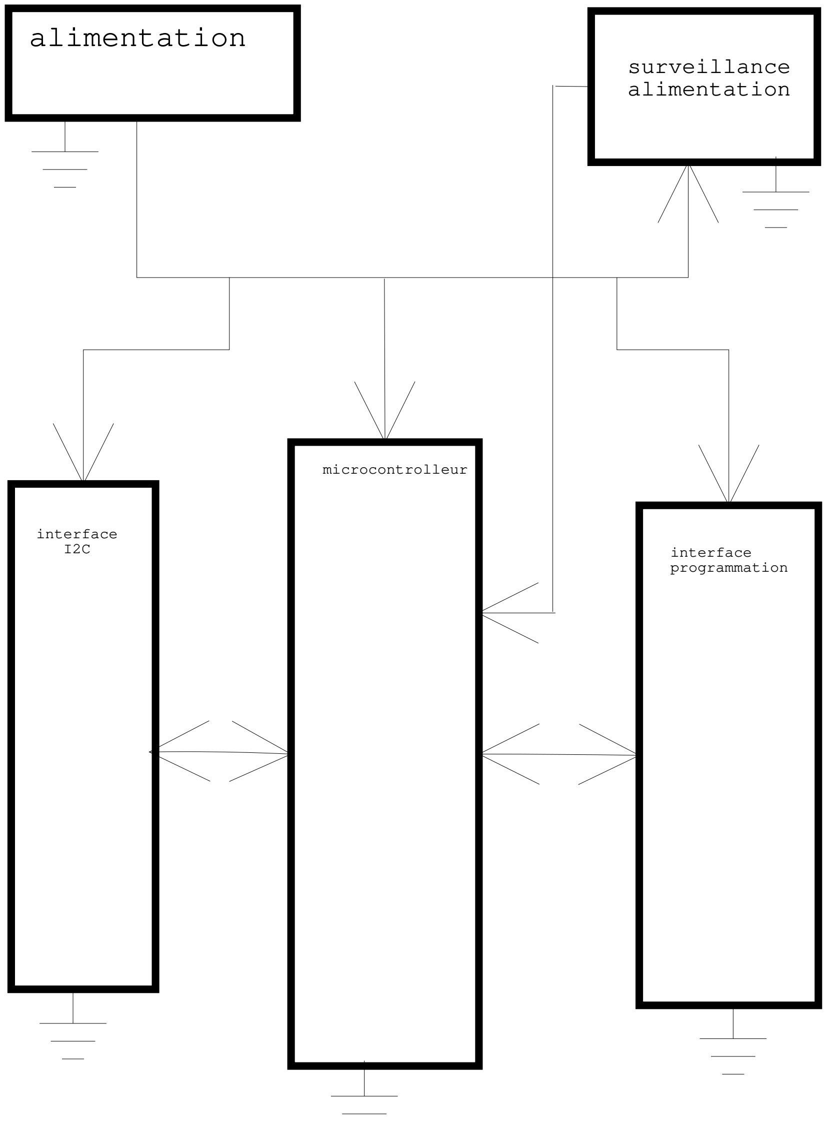

FIG. 2.1 – Synoptique du système minimal

2.1.3 Le choix du contrôleur

Comme on le voit, ce choix va être déterminant puisqu'il ne nous reste qu'une marche de manœuvre de 250 FB environ. . .Je vais comparer six microcontrôleurs dont les caractéristiques techniques conviennent pour ce projet. Il existe beaucoup plus de candidats (des milliers, sans doute), mais j'ai favorisé ceux dont les informations étaient les plus accessibles. Voici un tableau qui en présente les principales caractéristiques :

contrôleur	RAM (octets)	ROM (mots)	programmation	lignes d'entrée/sortie	prix
68HC11A1	256	256	in situ	38	420 FB
PIC16F84	68	1024	in situ	13	280 FB
SCENIX SX18	136	2048	in situ	13	300 FB
AT89S1200	32	512	in situ	15	200 FB
AT89C2051	128	2048	in situ	15	220 FB
80C32	256	0	eprom	14	170 FB

Ces contrôleurs présentent des avantages et inconvéniants spécifiques :

68HC11A1 : ce contrôleur (fabriqué par Motorola) se distingue des autres par les nombreux périphériques dont il dispose : plusieurs "timers", un port SPI (SERIAL PERIPHERAL INTERFACE), une interface sérielle asynchrone (RS-232), un "watchdog", 8 entrées analogiques et de nombreuses lignes d'entrée/sortie...Par contre, il dispose d'une faible quantité de mémoire de programme. Il est, de plus, très cher. Les versions de ce contrôleur qui possèdent plus de mémoire de programme sont encore plus chères ! Notons que cette mémoire est de type EEPROM. Malgré ce fait, vu sa faible capacité, nous utiliserions en plus une EEPROM externe, ce qui porte le prix de l'ensemble à 670 FB. C'est un peu cher !

PIC16F84 : son principal avantage est d'être très connu (grâce, entre autres, à la pression marketing de son constructeur : Microchip). Ainsi, on trouve sur Internet une documentation abondante, de nombreuses notes d'application ainsi qu'un compilateur de langage "C" gratuit fourni par la firme Hi-Tech Software. Mis à part cela, les ressources fournies par ce contrôleur sont très minces. La plus grande difficulté qu'entraîne la mise en œuvre de ce contrôleur est la suivante : la pile à huit niveaux (seulement), qui permet les appels à des procédures et le traitement d'interruptions matérielles, n'est accessible qu'avec les instructions "call" (sauve l'adresse de retour et saute à la procédure) et "return" (revient à l'adresse qui avait été sauvée sur la pile). Elle n'est donc pas adressable par d'autres mécanismes ce qui rend impossible pour le système d'exploitation de savoir facilement où se trouve un processus donné. Des mécanismes spécifiques devront alors être mis en œuvre pour compenser cette faiblesse. Notons la présence d'une EEPROM de 64 octets. Nous n'avons donc pas besoin d'EEPROM[3] externe. Le prix total re-

[3]Précisons bien que cette mémoire externe ne peut pas contenir du code mais uniquement des données ; elle n'est pas adressable en temps que mémoire de programme.

vient alors à 430 FB.

SCENIX SX18 : ce microcontrôleur (de Scenix, actuellement Ubicom), est compatible avec le PIC16F84 au niveau du brochage. Son avantage est de supporter une fréquence d'horloge beaucoup plus élevée : jusqu'à 100Mhz actuellement ! De plus, il possède 2048 mots de programme et 136 octets de RAM, donc le double du PIC16F84.

AT89S1200 : ce contrôleur (fabriqué par Atmel) est bon marché mais cela se paye par une très petite capacité de mémoire de programme : 1024 octets ce qui donne 512 instructions seulement car une instruction est codée sur deux mots dans ce circuit ! Il possède aussi une EEPROM interne de 64 octets. Le prix total est donc d'environ 350 FB.

AT89C2051 : il présente un bon rapport qualité/prix : pour à peine plus qu'un AT89S1200, il fournit 128 octets de RAM et 2Ko de mémoire de programme (c'est à dire 1500 instructions en moyenne compte tenu du microprocesseur de type MCS-51 utilisé). Mais il ne contient pas d'EEPROM. On lui en adjoindra donc une. Un autre aspect intéressant de ce circuit est qu'il dispose de nombreux périphériques : interface sérielle et "timers". Le prix total serait de 470 FB.

80C32 + EPROM : j'ai repris ce contrôleur[4] ici car il est extrêmement peu coûteux, malgré le fait qu'il ne contienne pas de mémoire de programme (contrairement à d'autre membres de sa famille, plus chers). Malheureusement, il est technologiquement dépassé : son horloge maximale ne lui permet d'exécuter un instruction que toute les microsecondes ; de plus, sa combinaison à une EPROM externe forme un ensemble de 5 à 8 fois plus volumineux qu'un PIC16F84 par exemple ! Il n'est forcément pas programmable in-situ ce qui pourrait être compensé par l'utilisation d'une "EPROM Flash" externe contenant un programme adéquat qui permettrait un téléchargement du programme principal. Cette possibilité n'est pas envisageable ici vu le prix actuel très élevé de ce type d'EPROM qui de plus ont une capacité bien plus élevée que ce dont nous avons besoin (128Ko à 512Ko). Notez bien que cela ouvrirait des possibilités très intéressantes pour le développement d'un *vrai* système d'exploitation : on pourrait imaginer d'ajouter aux fonctions de base que nous avons déjà décrites un système de fichier virtuel (VFS), des fonctionnalités d'allocation dynamique de la mémoire (il faudrait alors aussi adjoindre au système une mémoire RAM de capacité *décente* d'au

[4] à l'origine réalisé par Intel mais décliné depuis lors en de nombreuses versions par d'autres fabricants

moins 8Ko pour que cela ait un sens),...Quoi qu'il en soit, dans le cas qui nous occupe, le prix total serait de 480 FB[5].

Comme je l'ai déjà dit, il existe de très nombreuses autres possibilités, et pas des moins intéressantes du point de vue technologique : AT90S2313 (128 octets de RAM et 1024 mots de programme), AT90S4414 (256/2048), AT90S8515 (512/4096), 68HC811E2 (256/2048),...Malheureusement, tous ces contrôleurs ont un prix prohibitif actuellement.
Bien, nous avons maintenant en main les éléments qui vont nous permettre de choisir ! La première solution est à écarter vu le critère de coût. La dernière est obsolète mais pourrait être intéressante pour le développement (on utiliserait un circuit compatible avec le 80C32 pour la production en série). Nous l'écarterons malgré tout car nous voulons un système directement utilisable. Si le langage de programmation à utiliser doit impérativement être du "C", alors la seule solution pratique est celle du PIC16F84 car c'est le seul de la liste pour lequel il existe un *bon* compilateur gratuit[6] (le compilateur de Hi-Tech Software est une sorte d'échantillon gratuit des produits payants (compilateurs pour d'autres microcontrôleurs) que la firme propose ; la qualité du compilateur proposé se doit donc de reflèter les compétences de l'entreprise). Il n'est pas exclu qu'il existe des compilateurs "C" pour les circuits d'Atmel mais mes recherches n'ont encore rien donné sur ce point. Cependant, Atmel (comme Microchip d'ailleurs), fournit un assembleur pour ses produits ...

Il faut reconnaître que pour cette application, l'utilisation du "C" constituerait un "plus" indéniable en termes de lisibilité et de portabilité (vu le caractère générique du logiciel à développer). Gardons aussi à l'esprit la difficulté liée à l'impossibilité d'adresser la pile du PIC (cette limitation sera examinée en détail plus loin).

En définitive et après de mûres réflexions, nous choisissons le PIC (qui pourra être remplacé par le SCENIX, totalement compatible avec le PIC mais plus rapide que lui, si la vitesse de ce processeur est requise pour une application spécifique).

Le marché des microcontrôleur étant en évolution permanente, une surprise nous attendait ! En effet, alors que cette étude de coûts était terminée, Microchip sortait un nouveau microcontrôleur, le 16F877. Ce dernier, d'un coût très abordable (environ 350 FB vers Noël 2000), semblait très intéressant. Une section sera consacrée à sa description. Nous avons été amené à

[5]en comptant une EPROM (2764 : 60 FB) externe contenant le programme.
[6]dans le cadre d'un travail de fin d'études, nous sommes contraints d'utiliser des solutions logicielles gratuites vu le prix invraisemblable des logiciels professionnels dans le domaine du développement en électronique.

l'utiliser dans le cadre d'un cours se donnant à l'université de Liège : le cours de systèmes programmés enfouis. Nous devions réaliser un montage pour ce cours qui devait utiliser ce circuit. Il devenait alors naturel de réutiliser ce montage, pour autant qu'il soit suffisamment générique. Nous allons donc proposer deux versions de la partie matérielle de ce travail : la première découle directement de l'étude de coûts qui a été faite. Cette version, utilisant un PIC16F84, cependant, n'a pas été réalisée physiquement. Une deuxième version qui, elle, utilise un PIC16F877, a été construite et sera le siège du fonctionnement de notre logiciel. Elle permettra d'illustrer le fonctionnement de ce dernier dans un cas concret. Notez bien que le logiciel que nous allons développer sera compatible avec les *deux* versions. Avant de passer aux descriptions de ces deux versions de la réalisation matérielle, nous allons d'abord nous intéresser aux deux microcontrôleurs utilisés.

2.2 Présentation du PIC 16F84

2.2.1 Architecture

Ce PIC est un microcontrôleur 8 bits à architecture pseudo-RISC[7]. Pratiquement toutes les instructions, codées sur un mot de 14 bits, s'exécutent en 4 cycles d'horloge. Le processeur connaît un jeu de 35 instructions (d'où le terme RISC). Cette faible quantité d'instructions a permis de réaliser un processeur simple, qui exécute les instructions en un faible nombre de cycles. Malheureusement, l'impact d'une telle approche a quelques inconvénients : comme le code opératoire d'une instruction doit tenir dans 14 bits, les champs d'adresse qui permettent d'accéder aux différentes zones de mémoire du processeur ont une taille limitée. Des bits supplémentaires sont donc présents dans des registres spéciaux pour compléter ces adresses. Le programmeur doit modifier ces bits de manière à sélectionner le bon *banc* de mémoire lorsqu'il veut accéder à une variable (RAM) où à une sous-routine (ROM).

La figure 2.2 montre l'architecure interne du 16F84.

D'un point de vue logique, il y a deux zones de mémoire dans ce circuit : la mémoire de programme et la mémoire de données. On est donc en présence d'une architecture de type "Harvard" : les deux zones sont indépendantes et peuvent être adressées pendant le même cycle d'horloge.

[7]Reduced Instruction Set Computer. En fait, chaque cycle du processeur prend quatre cycles de l'horloge et une instruction nécessite quatre cycles du processeur (16 cycles d'horloge). Pour arriver à exécuter une instruction par cycle du processeur, quatre pipelines sont utilisés. Le processeur va donc quatre fois plus vite que si on n'avait pas utilisé de pipeline. Les instructions de saut prennent malgré tout deux cycles du processeur.

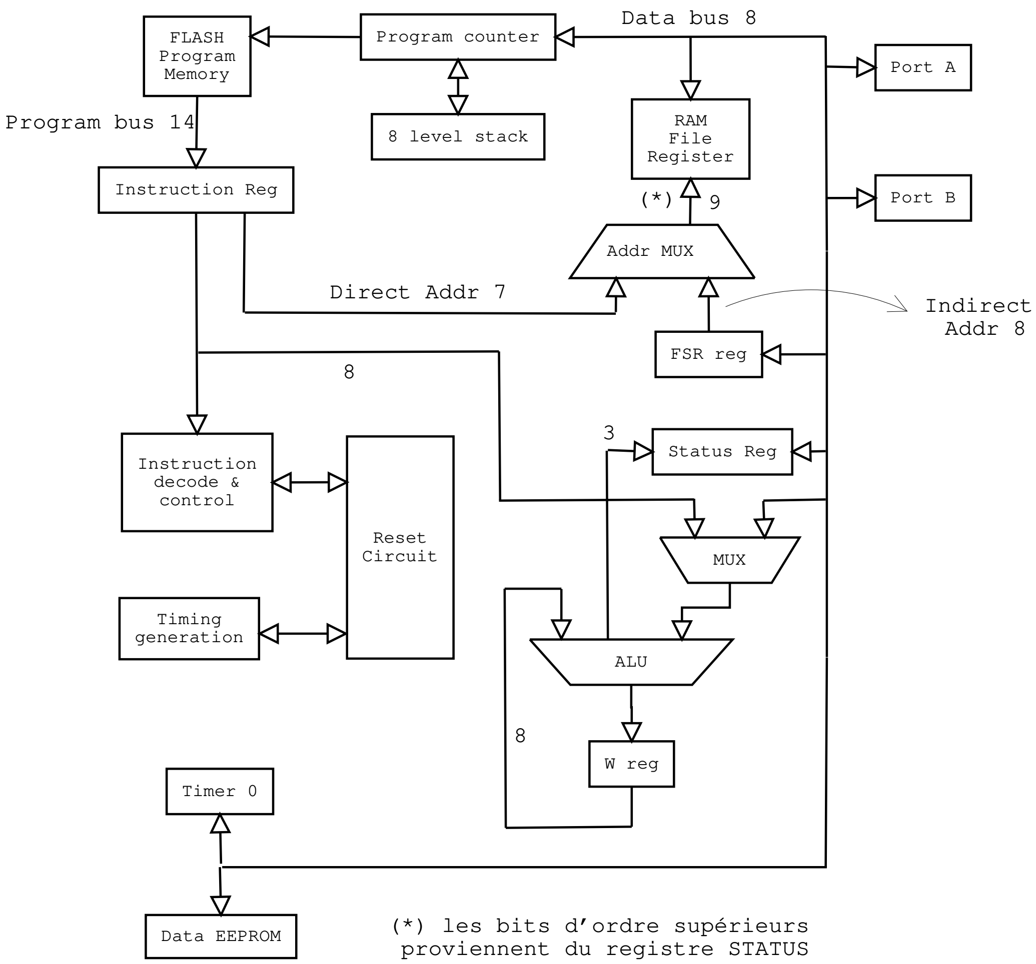

FIG. 2.2 – Synoptique du PIC 16F84

Comme nous sommes en présence d'un microcontrôleur, des périphériques sont aussi présents sur la puce du microprocesseur : un temporisateur 8 bits, un chien de garde, une logique de génération d'interruption, une EEPROM de 64 octets et des circuits d'entrée/sortie.

2.2.2 La mémoire de programme

Le compteur de programme du PIC (PC), a une taille de 13 bits : il permet donc d'adresser une mémoire de 8192 mots. Les mots de la mémoire de programme ont une taille de 14 bits : chaque mot contient le code opératoire d'une instruction. Sur le 16F84, seuls les 1024 premiers mots sont physiquement présents. Les autres zones sont des miroirs de cette zone physique.

Cette mémoire est de type FLASH, c'est-à-dire qu'elle peut être reprogrammée électriquement. Un système de programmation *in situ* est présent sur la puce du PIC. On peut donc reprogrammer le circuit sans l'extraire de la carte d'application sur laquelle il se trouve.

Les instructions de saut (GOTO) et d'appel de sous-routine (CALL) contiennent dans leur code opératoire un champ de 11 bits pour déterminer l'adresse du saut. Comme le compteur de progrmme fait 13 bits, les deux bits supplémentaires sont chargés à partir des bits 4 et 3 d'un registre spécial : PCLATH. Remarquez que dans le cas du 16F84, ce mécanisme est inutile vu que la mémoire de programme est adressable par un mot de 10 bits ($1024 = 2^1 0$ mots). Néanmoins, le registre PCLATH[8] est utilisé quand la cible d'une instruction est la partie basse de PC : PCL. Ce registre étant constitué de 8 bits, la partie haute de PC (PCH) est chargée à partir des 5 bits de poids faible de PCLATH.

2.2.3 La mémoire de données

Cette mémoire est constituée de deux bancs, chacun permettant l'accès à 128 adresses. Les registres à fonction spéciales et des zones de RAM se partagent ces adresses. De nombeuses zones ne contiennent physiquement rien. L'utilisateur dispose de 68 octets de RAM qui sont *mappés* dans les deux bancs. Cette mémoire peut être adressée de manière directe ou indirecte. Deux bits dans le registre spécial STATUS fournissent l'éventuelle partie manquante de l'adresse.

2.2.4 la pile matérielle

Le microprocesseur dispose d'une pile matérielle à 8 niveaux. Cette pile ne peut mémoriser que des adresses de programme et n'est accédée que lors de l'exécution d'une instruction CALL, de l'appel de la routine de traitement d'interruption, de l'éxécution d'une instruction RETURN (retour d'une sous-routine), RETFIE (retour de la routine de traitement des interruptions) ou RETLW (retour d'une sous-routine avec chargement d'une valeur immédiate dans l'accumulateur (W)). Dans les deux premiers cas, l'adresse de l'instruction qui suit celle qui est en cours d'exécution est placée au sommet de cette pile. Dans les trois derniers cas, l'adresse qui est au sommet de la pile est placée dans le registre PC.

Cette pile n'est pas présente dans l'espace d'adressage du PIC. Le programmeur n'a donc aucun moyen d'accéder à son contenu. Nous verrons par la suite l'impact que cette restriction a sur notre travail.

[8] PCLATH signifie Program Counter Latch High

2.2.5 Les interruptions

Il n'y a en fait qu'une seule interruption matérielle sur le PIC. Plusieurs périphériques peuvent la déclencher et le programme de traitement de l'interruption devra interroger les différents périphériques pour savoir lequel a réellement déclenché l'interruption. Si plusieurs périphériques demandent à être servis en même temps, l'ordre des tests présents dans la routine de traitement leur donnera un priorité. Les interruptions sont d'office interdites lors de l'appel de la routine de traitement. Si l'utilisateur le désire, il pourra réactiver ces interruptions explicitement au début de cette routine de traitement. Si on veut réellement que des périphériques aient des priorités différentes, on devra procéder de la sorte. En effet, imaginons qu'un périphérique à faible priorité soit en train d'être servi (le test qui lui correspond dans la routine de traitement d'interruption est placé *après* les tests de périphériques plus prioritaires). Si pendant ce temps, un périphérique plus prioritaire génère une interruption et que les interruptions sont interdites, il ne sera servi qu'après la terminaison de la routine de traitement d'interruption du périphérique qui est en train d'être servi.

Il faut noter ici une particularité importante du PIC : le drapeau de demande de service d'un périphérique est positionné, le cas échéant, indépendamment de l'état du masque d'interruption de ce périphérique. Si, par exemple, le temporisateur déborde, il positionnera son drapeau de débordement, même si il n'est pas autorisé à générer une interruption (et il n'en génèrera d'ailleurs pas). Le code C suivant, qui pourrait être utilisé pour programmer une routine de traitement d'interruption, est donc faux :

```
void interrupt(void) {

  if (TIMER_FLAG) {    /* interruption du timer ? */
    handleTimer();     /* traite la demande du timer */
    return;
  }
  if (OTHER_PERIPH_FLAG)
...
}
```

Il faut écrire :

```
void interrupt(void) {
```

```
/* il faut ensuite vérifier que le timer
   peut déclencher une interruption : */

  if (TIMER_FLAG && (TIMER_INTERRUPT_FLAG == ENABLED)) {
    handleTimer();
    return;
  }
  if (OTHER_PERIPH_FLAG)
...
}
```

J'ai bien sûr été victime de ce problème et, bien que des avertissements figurent en grand nombre dans la documentation de Microchip, on ne pense pas toujours à y faire attention !

2.2.6 Les périphériques

Voici les périphériques présents sur la puce du 16F84 :

1. Un temporisateur 8 bits : il s'agit d'un compteur 8 bits qui peut déclencher une interruption lorsque sa valeur passe de 0xFF à 0. L'horloge qui incrémente ce compteur peut être interne ou externe. Un prédiviseur est disponible ; cependant, il est utilisable soit par le temporisateur soit par le chien de garde.

2. Un chien de garde : c'est un compteur autonome qui déclenche une réinitialisation du processeur lorsque son contenu déborde. Le programme de l'application doit le réinitialiser périodiquement pour éviter que cette condition n'apparaisse. Si, pour une raison ou une autre, le programme ne se comporte pas comme prévu (parasite sur les lignes d'alimentation, erreur dans le programme, ...), il y a de grandes chances pour que le chien de garde ne soit pas réinitialisé, son débordement replacera le système dans un état connu (RESET).

3. Les lignes d'entrée/sortie : le PIC fournit 13 lignes qui peuvent chacune être dynamiquement configurées comme entrée ou comme sortie. Certaines de ces lignes peuvent aussi remplir d'autres fonctions (RB0 peut déclencher une interruption, par exemple).

4. L'EEPROM : cette mémoire est programmable électriquement par le programme de l'application (et par le programmateur externe). Elle

permet de stocker des données de configuration.

La fréquence d'horloge maximale du 16F84 est de 10 Mhz.

Ces informations sont tirées de [9].

2.3 Présentation du PIC 16F877

Le 16F877 présente de nombreuses similitudes avec le 16F84. Le processeur est le même. Les principales différences se situent au niveau du plus grand nombre de périphériques disponibles. La taille de la ROM et de la RAM ont été augmentées. La fréquence d'horloge maximale est passé à 20 Mhz. Des ports d'entrée/sortie ont aussi été ajoutés.

2.3.1 Architecture

La figure 2.3 montre l'architecture du 16F877. On constate, au premier coup d'œil que le nombre de périphériques a fortement augmenté.

2.3.2 La mémoire de programme

Elle est de 8 Kmots. Les mécanismes qui ont été vus pour le 16F84 sont d'application ici.

2.3.3 La mémoire de données

Le nombre de cellules de RAM disponibles est passé à 368. Cette fois, la mémoire de données est subdivisée en quatre bancs de 128 octets. Deux bits du registre STATUS permettent de sélectionner un des bancs.

2.3.4 Les périphériques

Mis à part les périphériques déjà présents dans le 16F84, les périphériques suivants sont aussi disponibles :

1. le temporisateur 1 : c'est un compteur 16 bits qui peut générer une interruption quand il déborde de 0xFFFF à 0. La source d'horloge peut être interne ou externe. Il dispose aussi d'un prédiviseur.

2. le temporisateur 2 : c'est un compteur 8 bits doté d'un prédiviseur et d'un postdiviseur dont la sortie peut produire une interruption. Le compteur compte (c'est normal, pour un compteur !) de 0 à la valeur d'un registre spécial (PR2). Quand cette valeur est atteinte, la valeur

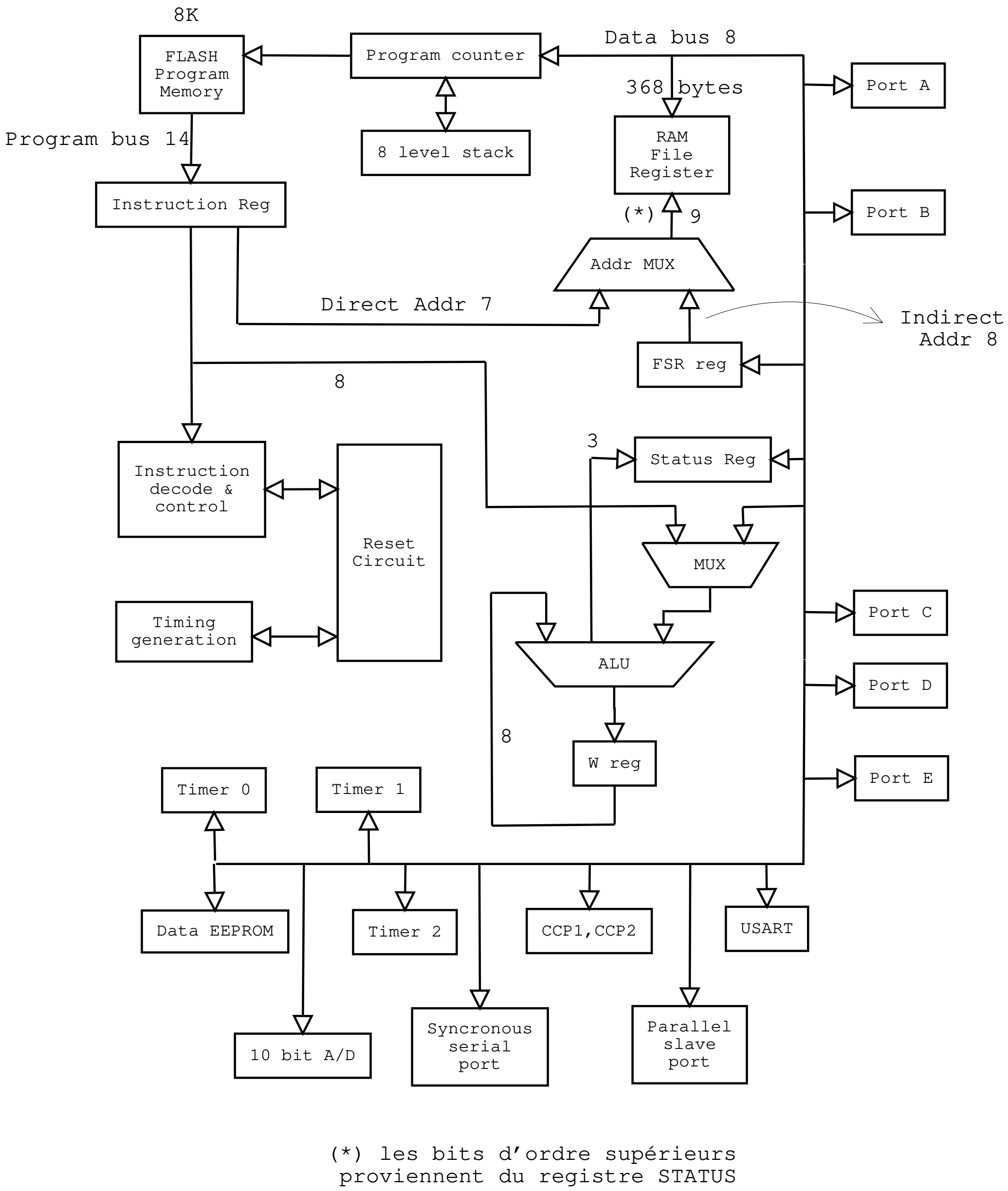

FIG. 2.3 – Synoptique du PIC 16F877

du compteur est remise à 0.

3. les deux unités de comparaison/capture : ces unités fonctionnent en collaboration avec les temporisateurs 1 et 2. La comparaison consiste à déclencher une interruption quand un compteur atteint une certaine

valeur. La capture consiste à mémoriser la valeur d'un compteur lorsqu'un événement a lieu (par exemple le passage d'un niveau bas à un niveau haut d'un broche d'entrée).

4. le convertisseur analogique $\longrightarrow$ numérique : ce convertisseur à 10 bits permet de convertir une tension analogique en une valeur binaire. Un multiplexeur permet de sélectionner une entrée parmi huit (les huit lignes du port d'entrée/sortie A peuvent être configurées comme entrées analogiques).

5. l'USART : c'est un circuit de communication sérielle synchrone ou asynchrone. On peut l'utiliser pour implémenter une interface RS-232, par exemple, ce qui est très pratique.

6. le SSP : c'est un circuit de communication synchrone exclusivement qui peut fonctionner, entre autres, aux normes I^2C, le bus inventé par Philips, et que nous utilisons dans notre réalisation. Bien que ce type d'interface puisse être simulé par programme, le fait de disposer d'une interface matérielle apporte un avantage certain : le processeur n'est plus sollicité pendant les transferts. Il ne doit intervenir que pour fournir un octet à envoyer ou lire un octet reçu. Le périphérique signale ces deux conditions au processeur par la génération d'une interruption.

7. le port parallèle esclave : ce système permet au PIC d'apparaître aux yeux d'un processeur externe sous la forme d'un périphérique intelligent auquel il est connecté par un bus de données de huit bits.

Ces informations concernant le 16F877 proviennent de [10].

2.4 Description succinte du bus I^2C

Le bus I^2C comporte 3 fils : une ligne de données bidirectionnelle (SDA), une ligne d'horloge, dans notre cas unidirectionnelle (SCL), et la masse. La vitesse maximale définie dans le standard original de Philips fut fixée à 100Kbps (plus tard, des vitesses plus élevées ont été introduites). Un "0" logique est caractérisé par une tension de 0 volts et un "1" logique par une tension de 5 volts. Les étages de sortie placés sur les lignes SDA et SCL sont de type "collecteur ou drain ouvert" : elles peuvent uniquement placer la ligne à la masse. Une résistance de tirage au niveau haut est présente sur chacune de ces lignes de manière à ce qu'un "1" logique soit présent si aucun étage de sortie n'est actif. Les conflits de bus matériels sont impossibles, ce qui n'est pas le cas des conflits logiciels. Nous utiliserons ce bus dans notre réalisation pour plusieurs raisons : de nombreux circuits périphériques dotés d'une interface I^2C existent sur le marché (horloge temps réel, syntoniseurs, circuits d'entrée sorties, convertisseurs analogiques $\longleftrightarrow$ numériques, ...). Ce bus est donc universel. De plus, il est facile d'implémenter ce type d'interface dans un microcontrôleur sous forme logicielle, si une interface I^2C n'est pas déjà présente.

Nous mettrons ce bus en pratique de la manière la plus simple : un seul circuit jouera le rôle de maître (le microcontrôleur). Le bus I^2C, utilisé dans ce mode, fonctionne de la manière suivante : losrque le maître veut lire ou écrire une donnée de ou vers un périphérique, il place une condition "start" sur le bus. Les lignes SDA et SCL sont, quand le bus est à l'état inactif, au niveau logique haut. Une condition "start" est générée quand le maître place la ligne SDA à 0 *avant* de placer la ligne SCL à 0. Ensuite, le maître envoie sous forme sérielle l'adresse du circuit qu'il veut accéder. Cette adresse est constituée de sept bits (on peut donc adresser 128 périphériques[9]. Il envoie aussi un bit qui indique au périphérique qu'il veut recevoir une valeur de celui-ci ou qu'il veut lui envoyer une valeur (bit R/W : 1 pour une lecture et 0 pour une écriture). Le maître désactive alors son étage de sortie au niveau de la ligne SDA et lit la valeur qui y est éventuellement placée par le périphérique. Si c'est un niveau 0, cela veut dire que le périphérique est présent et a bien reçu les informations envoyées par le maître. Ce bit s'appelle ACK (Acknowledge).

Ensuite, le maître envoie les huit bits de données (opération de lecture) ou reçoit huit bits du périphérique. Quel que soit l'émetteur de la donnée, c'est le maître qui génère le signal d'horloge. Le récepteur, quel qu'il soit, génèrera le bit ACK. Les bits transmis sont pris en compte lors du flanc descendant du signal d'horloge. L'état de la ligne SDA ne peut changer, en cours de transfert, que si la ligne SCL est au niveau bas, de manière à ne pas générer pendant ce transfert une condition "start" ou "stop" (voir plus loin).

Quand l'opération est terminée, le maître l'indique par la génération d'une condition "stop" qui consiste à faire remonter à 5 volts le ligne SCL *avant* de faire remonter la ligne SDA.

La figure 2.4 montre une opération de lecture du maître depuis un périphérique.

[9] Certains périphériques occupent plusieurs adresses. Les adresses sont formées d'une partie fixe qui dépend du type du périphérique et d'une partie variable qui permet de différencier des périphériques identiques

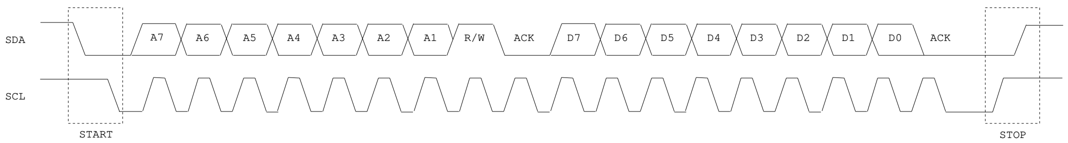

FIG. 2.4 – Un exemple de transfert sur le bus I^2C

2.5 La réalisation matérielle à base de 16F84

2.5.1 Une conception modulaire

L'implémentation matérielle retenue met l'accent sur la modularité. Différents modules de taille identique et de forme rectangulaire communiquent entre eux à l'aide d'un bus I^2C. Ces modules sont physiquement séparés et sont reliés à l'aide de connecteurs standards (SUB-D à 9 broches). Cette technique permet d'utiliser uniquement les modules dont on a réellement besoin pour une application donnée. Il s'assemblent en un clin d'œil. Le schéma 2.5 représente cette approche.

Les modules ont tous la même taille et, si on respecte leur orientation, s'assemblent très facilement. Les connecteurs véhiculent la tension d'alimentation, la ligne de *reset* et les lignes du bus I^2C (SCL, le signal d'horloge, et SDA, la ligne de données bidirectionnelle).

En ce qui concerne cette version, seuls deux modules ont été conçus : un module d'alimentation et un module abritant le PIC. Ces deux modules reprennent les fonctionnalités de la figure 2.1. Ils respectent la limite de 500FB décrite dans l'étude des coûts.

2.5.2 L'alimentation

L'alimentation est très simple : un régulateur tripode de type 7805 stabilise la tension redressée et filtrée qui lui est fournie par un transformateur externe 220 volts vers 9 volts. Le courant maximal qu'elle peut fournir est de une ampère pour autant qu'on dote le régulateur d'un radiateur de taille adéquate. Le circuit de l'alimentation comporte aussi un dispositif important : un circuit de contrôle de la tension d'alimentation de type TL7705. Ce circuit vérifie que la tension d'alimentation de 5 volts ne tombe pas en dessous de 3,6 volts. Si c'est le cas, il réinitialise le système. Certains microcontrôleur comprennent un système analogue (Brown Out Detection). Même si on utilise un de ces circuits, le TL7705 reste essentiel pour réinitialiser les autres composants du système et permet en outre d'utiliser un microcontrôleur qui ne serait pas équipé d'un tel système (comme c'est le cas, d'ailleurs, du 16F84). L'autre fonctionnalité importante de ce circuit est de générer un condition de réinitialisation (RESET) lorsqu'on appuie sur une touche qui

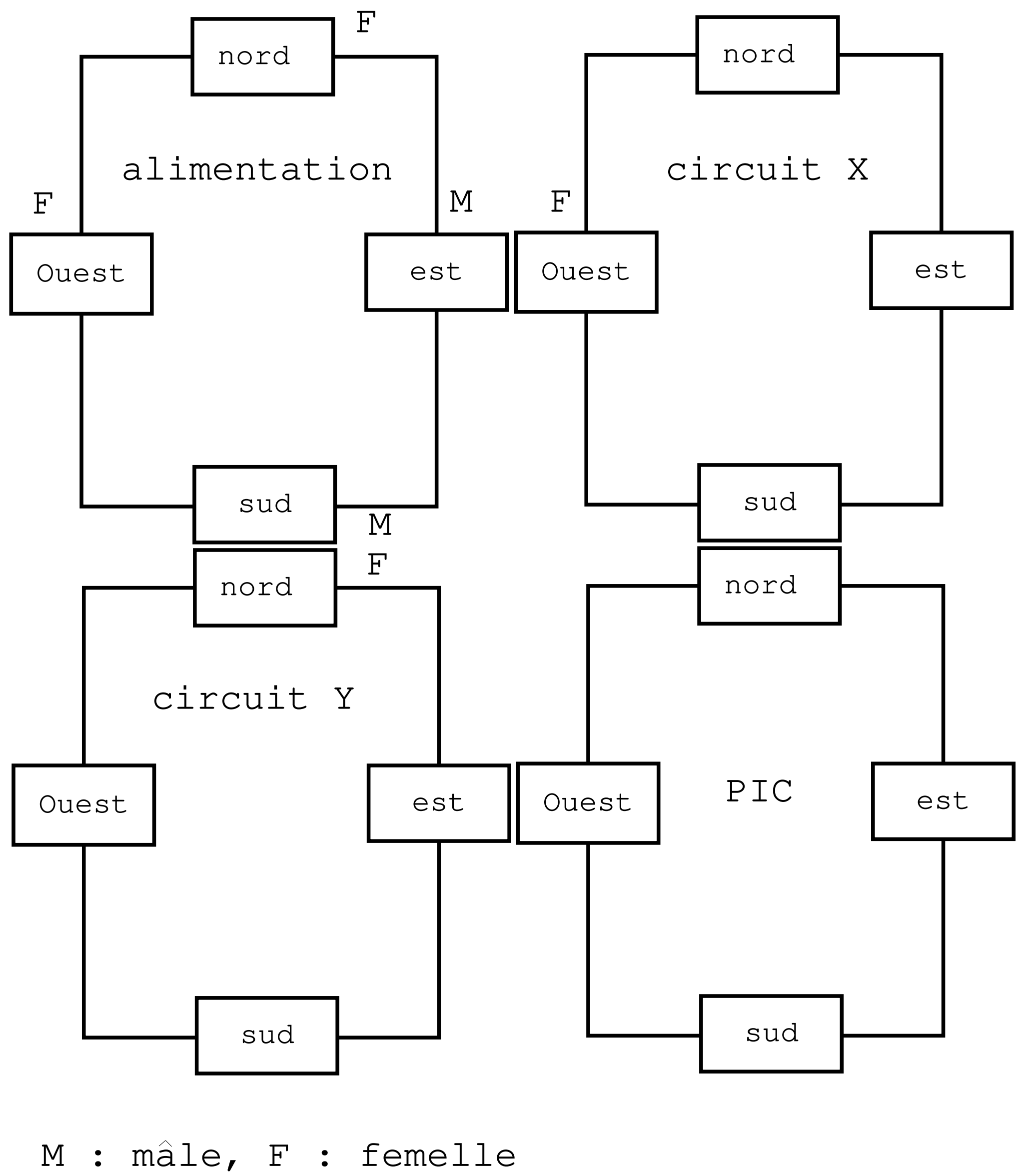

FIG. 2.5 – L'interconnection des modules

lui est connectée. Il gère les éventuels rebonds de cette touche et, lors d'un appui sur celle-ci, initialise le système pendant une durée qu'on détermine par la valeur d'un condensateur externe.

Le fait d'avoir placé l'alimentation sur un module autre que celui du PIC permet d'utiliser d'autres modules éventuels, dotés d'autres microcontrôleurs, sans changer l'alimentation. Le schéma de cette alimentation se trouve sur la figure 2.6.

2.5.3 Le circuit du microcontrôleur

Ce dernier est connecté par deux lignes (du port B) au bus I^2C. Les lignes qui peuvent servir d'entrées/sorties ou d'entrées analogiques (port A et B) sont regroupées sur des connecteurs spécifiques pour en permettre une utilisation facile. Un condensateur découple l'alimentation du PIC. Les lignes du bus I^2C sont dotées de résistances de tirage au niveau haut de valeur adéquate. Il n'y a pas grand-chose à dire de plus sur ce module, puisqu'il n'abrite que le PIC ! Le schéma de ce module est donné en figure 2.7

Bien que le PIC contienne un oscillateur qui puisse fonctionner sans composant externe, le circuit imprimé prévoit l'utilisation d'un quartz, ce qui permet alors d'obtenir une horloge plus stable en fréquence, pour les applications qui en auraient besoin.

2.5.4 Conclusion

L'utilisation de ces modules peut être extrêmement pratique pendant la phase de conception d'un projet. Néanmoins, il ne serait pas raisonnable d'utiliser ces modules dans le cadre d'une application commerciale. En effet, le coût des connecteurs de liaison serait prohibitif. Même si on relie les différents modules par des fils, soudés une fois pour toutes, il reste un désavantage. En effet, lorsque les circuits imprimés ont été réalisés, il est vite devenu clair que la taille et la disposition des connecteurs exigeraient un surface non négligeable. Ce gaspillage de la surface du circuit imprimé n'est pas acceptable dans le cadre d'une application finalisée. C'est pour cette raison que la deuxième version de cette réalisation abandonne l'idée de l'utilisation des connecteurs. Par contre, elle reste aussi modulaire que la solution précédente. Cependant, cette modularité n'apparaît plus à l'utilisateur final, mais bien à la personne qui conçoit le circuit imprimé.

2.6 La réalisation matérielle à base de 16F877

Comme nous venons de l'expliquer, cette réalisation reste modulaire mais n'utilise plus de connecteurs entre les modules. L'application finale n'est constituée que d'un seul circuit imprimé. Ce circuit est constitué d'autant de zones qu'il y a de modules. Ces zones sont reliées entre elles par des ponts de câblage. Le concepteur du circuit imprimé choisit les modules dont il a besoin et les place côte à côte dans un ordre quelconque, pour autant qu'il en respecte l'orientation. la place gagnée n'est pas négligeable : les modules de la première version ont une taille de 7,7 cm par 6,5 cm. Les modules de cette version-ci ne mesurent que 4 cm fois 5 cm. Le circuit imprimé est utilisé de manière beaucoup plus économique. Les modules de cette version, bien que moins adaptés à une expérimentation débridée, seraient utilisables pour une application finale à produire en grande série (moyennant certaines restrictions, comme nous le verrons dans la conclusion finale).

Quatre modules différents ont été développés, comme le montre La figure 2.8.

2.6.1 L'alimentation

L'alimentation est rigoureusement la même que pour la réalisation précédente, mise à part l'utilsation de ponts de câblage au lieu de connecteurs pour la liaisons des modules, comme on le voit sur la figure **??** qui se trouve en annexe.

2.6.2 Le circuit du microcontrôleur

Ici, ce sont deux lignes du port C qui servent à implémenter le bus I^2C. Contrairement au 16F84, le 16F877 contient une interface I^2C et les lignes associées ne peuvent être choisies librement. Les lignes du port A et celles des autres ports qui ne sont pas déjà utilisées sont amenées sur un connecteur spécifique. Les deux lignes de l'interface sérielle asynchrones (port C) sont amenées, elles aussi, sur un connecteur dédié. Les ports B, D et certaines lignes du port C sont réservées à une interface de déboguage (voir plus loin). Le schéma correspondant est donné en figure 2.9.

2.6.3 L'interface de programmation

Comme cette réalisation a été conçue dans un but pédagogique où le développement tenait une place essentielle (par rapport à une application finale et dédiée), il était logique de la doter d'une interface de programmation. Pour accentuer son caractère modulaire, seuls les composants génériques qu'on retrouve sur ce type d'interface sont présents sur la carte principale. Il s'agit du circuit de génération de la tension de programmation du PIC et du circuit de réinitialisation commandée depuis le “monde extérieur”, c'est-à-dire le programmateur lui-même. Ce dernier, qui peut prendre de nombreuses formes, n'est pas présent sur la carte. En effet, celui-ci pourrait être un microcontrôleur relié à une station de travail par l'intermédiaire d'une interface sérielle ou bien un ordinateur personnel relié par une interface parallèle à la carte du PIC, ...

Le principe de génération de la tension de programmation est très simple : un régulateur à trois broches de type LM317 est utilisé ici. Ce régulateur contient un système qui ajuste la tension de sortie de sorte qu'il y ait toujours 1,2 volts entre cette sortie et la broche dite “de commande”. En reliant cette dernière broche à la masse à l'aide de résistances de valeurs variables, on modifie son potentiel. La tension de sortie sera 1,2 volts plus élevée que la tension sur la broche de commande. La sortie à collecteur ouvert d'un circuit 7406 (inverseur TTL) permet de faire passer la tension de sortie du LM317 à 13 volts (la tension de programmation du PIC) ou à 5 volts. Cette sortie est connectée à l'entrée /MCLR du PIC. Si cette entrée est à 0 volts, le PIC est en mode de *RESET*. Si elle est de 5 volts, le PIC est en mode *RUNNING*. Si, enfin, elle est de 13 volts, le PIC passe en mode *PROGRAM* et il est prêt à recevoir des commandes de programmation sous forme sérielle synchrone sur ces broches RB6 et RB7, réservées à cet effet (dans ce mode, car en mode *RUNNING*, ces broches sont des lignes d'entrées/sorties standard). Un connecteur véhicule les informations de programmation du PIC par les lignes suivantes : 2 lignes pour RB6 et RB7 et deux lignes pour déterminer une des trois tensions possibles sur l'entrée /MCLR du PIC (0, 5 ou 13 volts).

Ce qui vient d'être décrit se trouve sur le schéma 2.9. Cependant, j'ai réalisé une interface de programmation qui permet de programmer le PIC à l'aide d'un PC à travers son interface parallèle. Le schéma de cette interface se trouve sur la figure 2.10. Le principe est simple : quand le PC n'est pas en train de programmer le PIC, il rend d'office inactives les lignes qui permettent sa programmation (RB6 et RB7 sur le PIC). De cette manière, l'application qui utilise éventuellement ces lignes ne sera pas perturbée. Les sorties du 7406 sont d'office inactives en dehors de la phase de programmation. De plus, deux résistances sont mises en série entre RB6, RB7 et les sorties du 7406 correspondantes pour éviter qu'un bogue dans le logiciel, qui mènerait peut-être à un conflit de bus potentiel, ne place le microcontrôleur dans une situation peu enviable !.

2.6.4 L'interface de déboguage

Il s'agit d'une interface constituée d'un bus d'adresses de 16 bits et d'un bus de données de 8 bits. Un bus de contrôle doté d'une ligne R/W (read et /write) et d'une ligne d'horloge permet d'échanger des informations dans les deux sens et de manière synchrone sur ce bus. On peut y connecter, par exemple, une carte d'affichage ou des périphérique typiques de ce genre de bus. En effet, les circuits 8 bits classiques de Motorola peuvent s'y connecter sans problème (UART 6850, PPI 6821, CRTC 6845...).

2.6.5 L'interface I^2C -> 145026

Cette interface a été développée à titre d'illustration. Le circuit intégré qui se trouve au cœur de cette partie est le MC145026 de Motorola. Il s'agit d'un encodeur sériel qui émet de manière asynchrone des informations d'adresse et de données. On peut encoder 9 "trits" d'informations (un trit peut prendre trois états : 0, 1 ou 2). Selon le décodeur utilisé, ces informations sont réparties en un champ d'adresse et un champ de données de tailles variables. Ici, nous utilisons un encodage binaire uniquement. Quatre bits sont réserver à encoder l'adresse du destinataire et les cinq autres représentent la donnée qui lui est envoyée. Ce type de circuit est utilisé dans de nombreuses applications : commandes infrarouges et HF, circuits de communications digitales,...

Une application amusante sinon intéressante de ce circuit est son utilisation par la firme Märklin dans son système de télécommande de trains miniatures (de première génération car actuellement, elle utilise des microcontrôleurs (!) pour l'encodage sériel, ceux-ci étant plus souples d'utilisation). Il était naturel de mettre en œuvre un tel système dans notre contexte. Comme nous utilisons 4 bits pour coder l'adresse du décodeur auquel nous voulons envoyer des informations, nous pouvons adresser un maximum de seize locomotives. Les cinq bits de données se répartissent en quatre bits pour définir la vitesse de la locomotive et un bit de fonction (par exemple pour allumer les phares). Le MC145026 émet en permanence les données qui lui sont fournies par deux PCF8574. Ces circuits sont des interfaces I^2C qui comportent huit bits d'entrée/sortie. Le système parcourt cycliquement une table qui contient les informations de vitesse des différentes locomotives. Il écrit successivement ces données dans les ports des PCF8574. La sortie sérielle du MC145026 est reliée à un amplificateur de puissance qui est connecté au rails. Les locomotives tirent ainsi du signal qu'elles reçoivent des informations logiques et de la puissance électrique (qui leur permet de rouler).

Le schéma de cette interface est donné en figure 2.11

Nous verrons plus en détail comment fonctionne cette interface dans le chapitre consacré à une application concrète de notre système.

2.6.6 La zone d'extensions

On l'a vu, la suppression des connecteurs de liaison entre modules a fait perdre à cette version son caractère modulaire du point de vue de l'utilisateur final. Pour compenser ce fait, j'ai placé dans le circuit deux modules dotés uniquement de connecteurs du type de ceux utilisés dans la première version. Des modules compatibles avec celle-ci peuvent ainsi s'enficher facilement dans notre deuxième version. Le schéma (**??**) correspondant à cette partie ne présentant que fort peu d'intérêt, il s'est vu reporté en annexe.

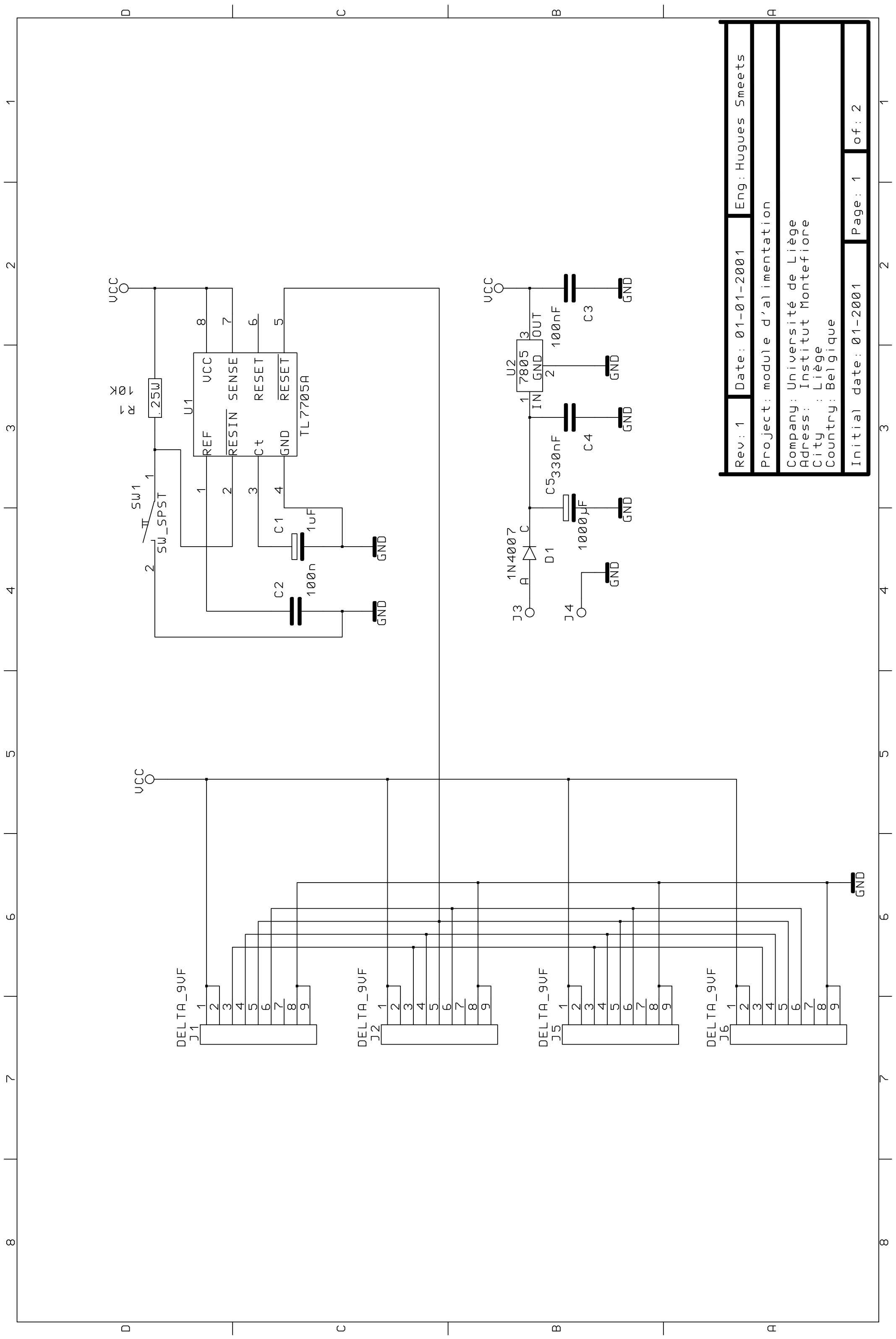

Fig. 2.6 – Alimentation et génération du reset : version 1

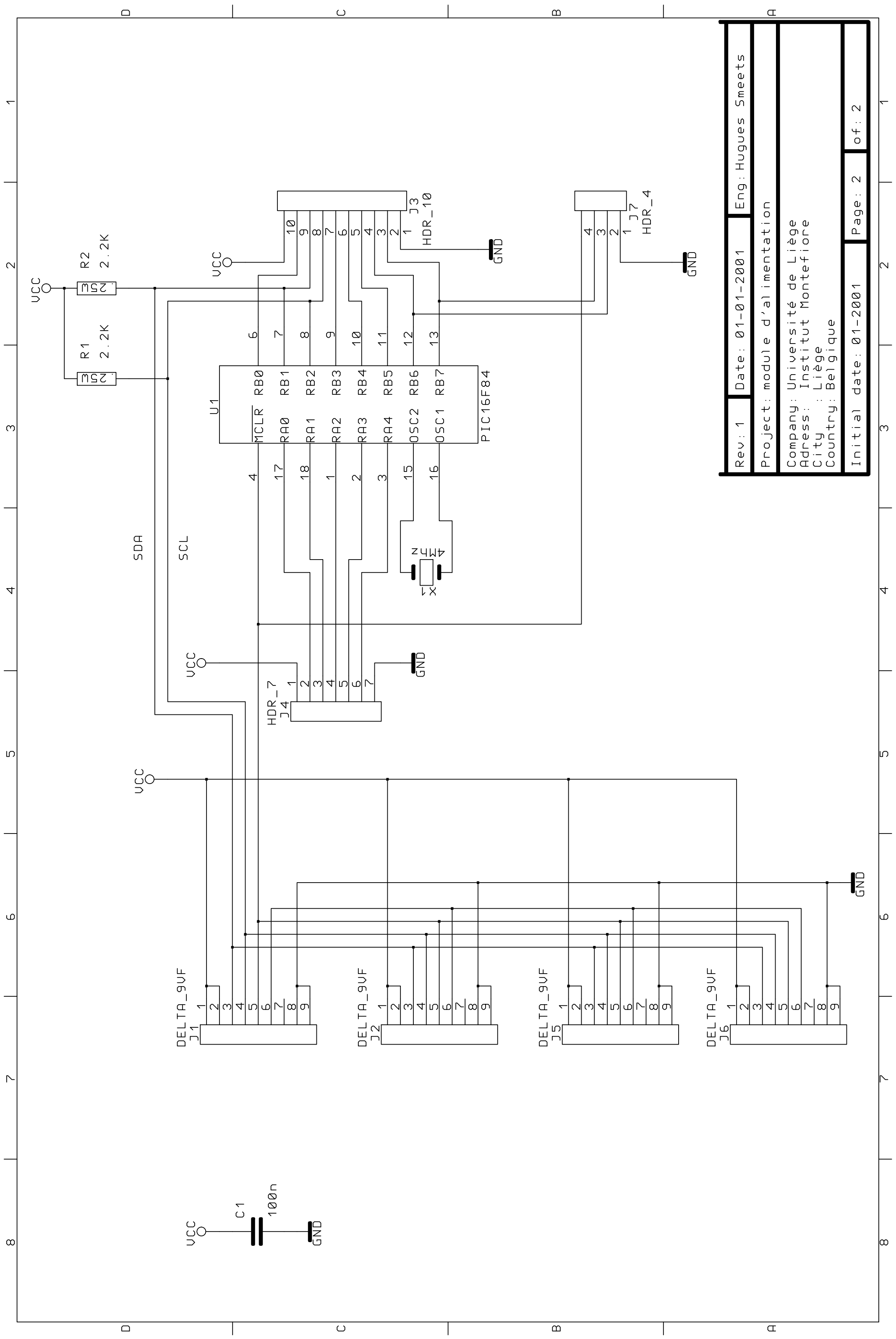

Fig. 2.7 – Module du microcontrôleur : version 1

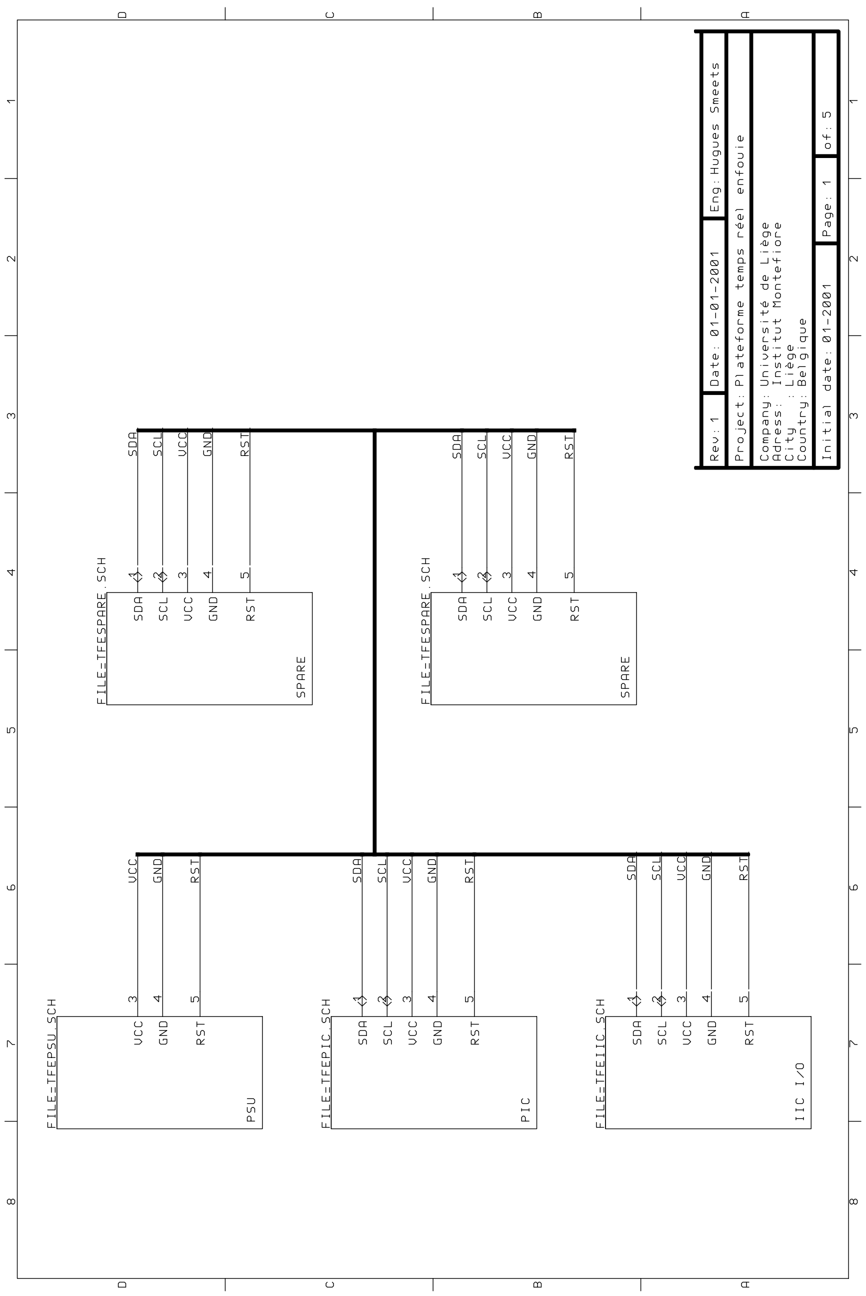

FIG. 2.8 – Synoptique du système

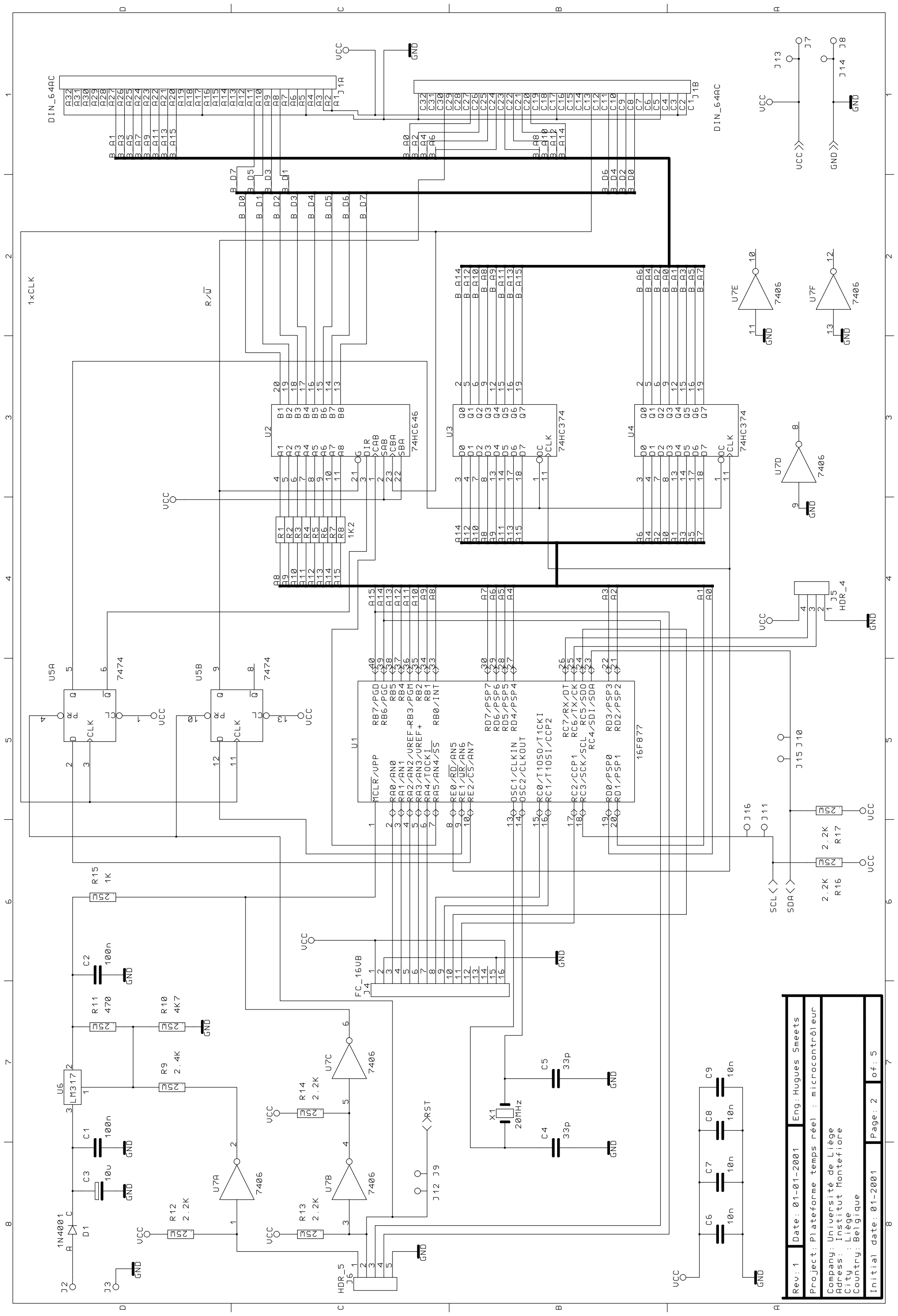

FIG. 2.9 – Module du microcontrôleur : version 2

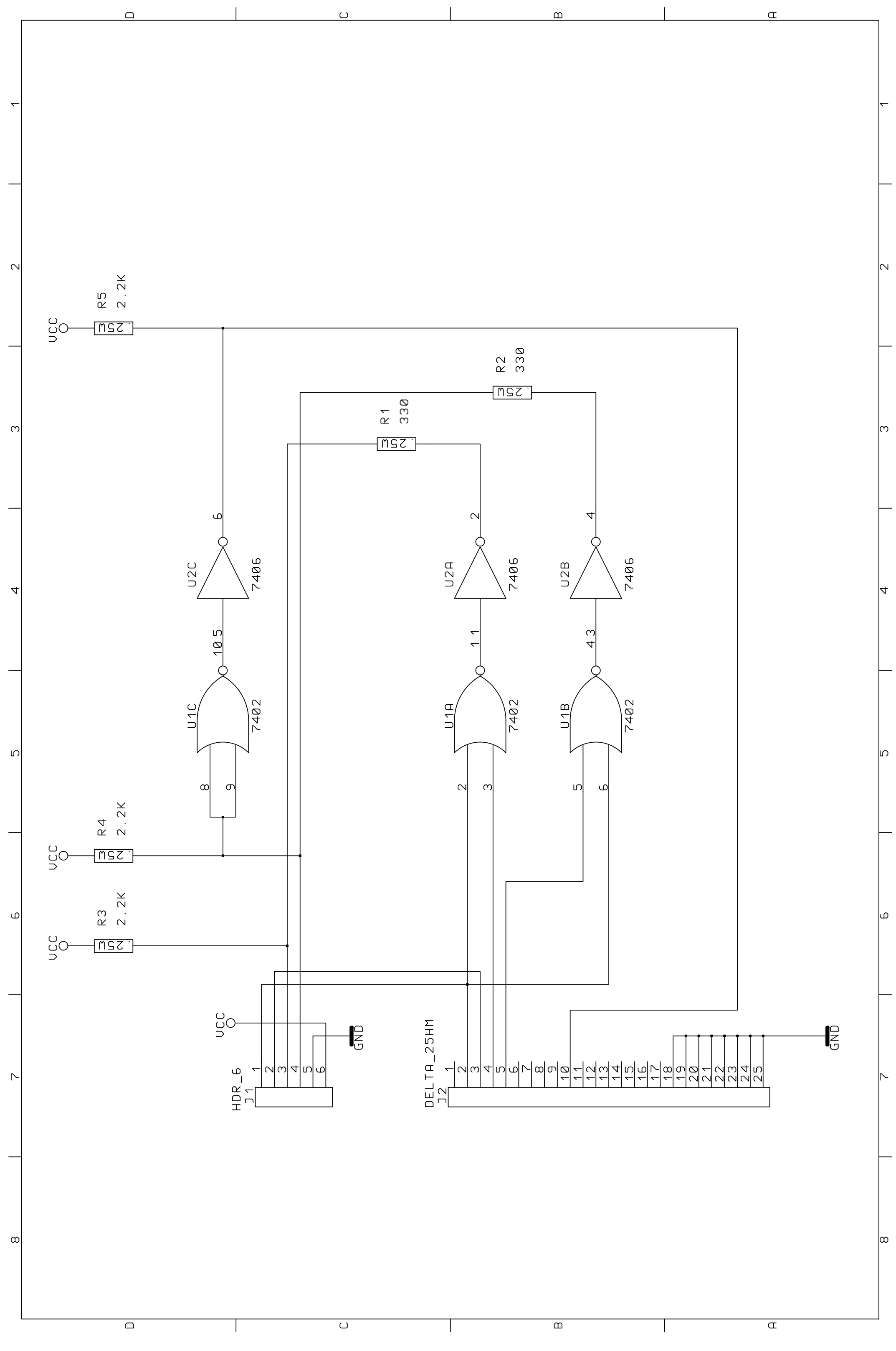

FIG. 2.10 – Interface de programmation

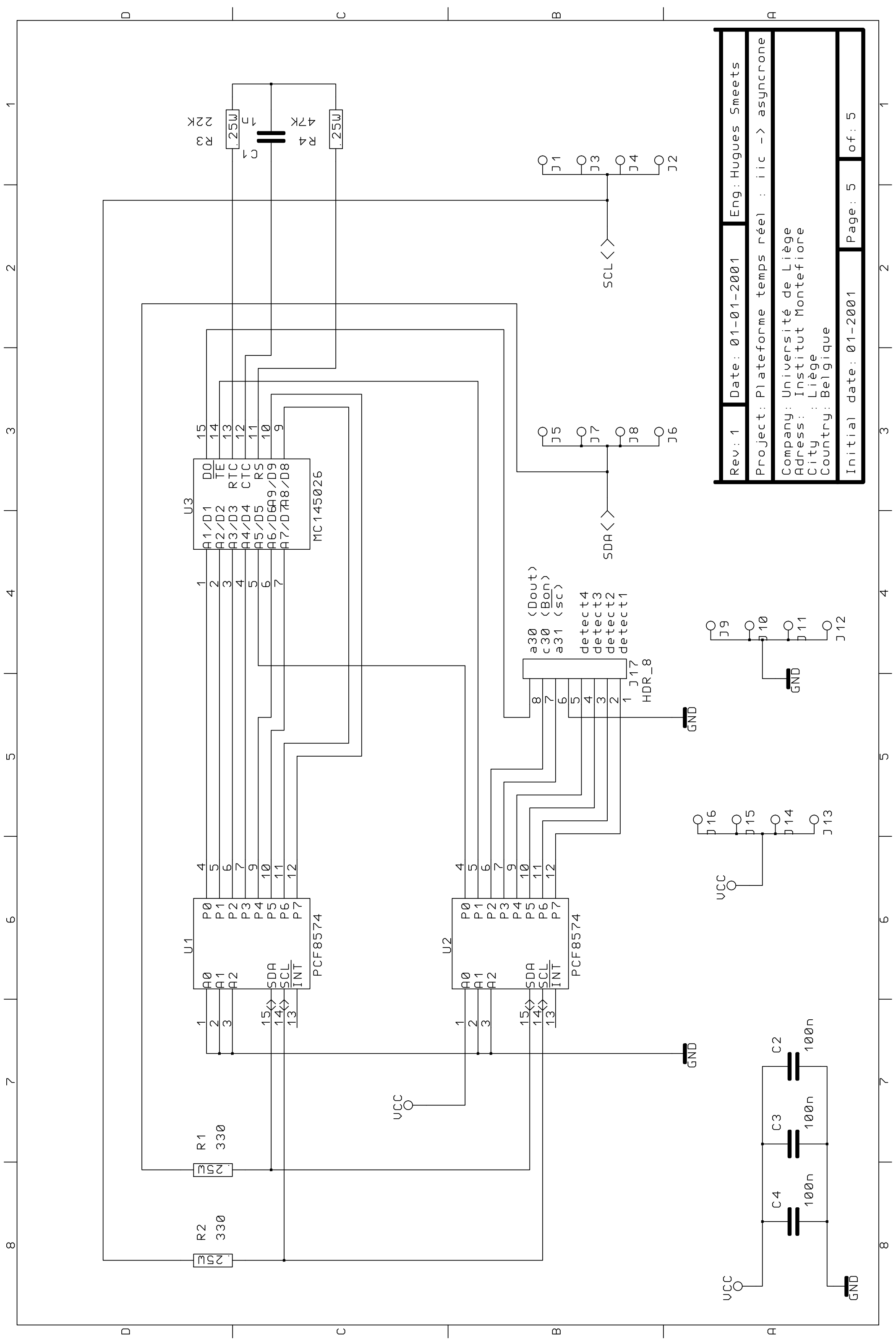

FIG. 2.11 – Interface I^2C -> encodeur sériel 145026

Chapitre 3

Le logiciel

Bien que notre but, ici, soit de fournir la plupart des fonctionnalités offertes par un système d'exploitation "temps réel", il est intéressant de se pencher sur différentes architectures logicielles et sur les raisons pour lesquelles elles ont été imaginées. Une première section présentera diverses techniques extraites de la littérature.

Dans la deuxième section, nous préciserons ce que nous attendons exactement de notre solution particulière.

Une troisième section exposera l'impact des limitations présentées par le microcontrôleur utilisé et les solutions *originales* spécifiques qui permettent de les contourner.

3.1 Les architectures logicielles

Après avoir vu comment fonctionne un système d'exploitation généraliste classique (non "temps rél"), nous passerons en revue différentes solutions plus adaptées au systèmes "temps réel".

3.1.1 Les caractéristiques d'un système d'exploitation généraliste

Un des rôles fondamentaux d'un système d'exploitation est de permettre l'exécution concurrente de plusieurs tâches. Une caractéristique importante d'un système généraliste est qu'il ne donne habituellement aucune garantie sur le délai qui s'écoule entre la demande d'un service par une application et l'exécution de celui-ci. Par exemple, si une application demande au système d'exploitation d'écrire un certain nombre d'octets sur un disque dur, cette opération pourra être réalisée 1 ms ou 1 heure plus tard, selon l'état courant du système (le nombre de processus qui sont déjà lancés, la puissance de la machine, les périphériques connectés,...).

Bien sûr, ce désavantage est compensé par la grande flexibilité de ce genre de systèmes : on peut lancer un grand nombre d'applications (qui seront chargées depuis un disque dur, par exemple, en fonction des actions de l'utilisateur). Ces applications sont exécutées sous la forme d'un nombre quelconque de processus et leurs domaines d'intérêt sont extrèmement variés. Ce genre de chose ne sera pas possible sur un système "temps réel".

Comment fonctionne un tel système ? Pour simplifier, imaginons que le système ne comporte qu'un seul processeur. Pour simuler le parallélisme, l'exécution d'un processus est interrompue à intervalles de temps réguliers et le contrôle est passé à un autre processus, choisi par le système d'exploitation. Un élément du système d'exploitation, appelé *scheduler*, ou ordonnanceur, s'occupe de classer dans un certain ordre les processus en fonction de leur priorité. Les processus qui ont été lancés sur le système sont placés dans une file d'attente. Un seul processus est en cours d'exécution à un moment donné, les autres attendent.

Un autre élément du système d'exploitation, appelé *dispatcher*, est de manière cyclique ou lorsqu'un processus appelle une fonction du système d'exploitation. Il sauve le contexte concernant le processus en cours d'exécution (état des registres, adresse de l'instruction en cours d'exécution), choisit le prochain processus dans la file d'attente et charge son contexte (qui avait été sauvé quand il avait été interrompu), relançant ainsi son exécution. Pour simplifier, nous regrouperons dans la suite ces deux éléments sous le terme "scheduler".

3.1.2 Le temps

Puisque les processus doivent sembler s'exécuter en parallèle, la technique du *time slicing* est utilisée. Le temps est divisé en durées discrètes fixes (quantum). Pendant l'écoulement d'un quantum, le processeur exécute les instructions d'un processus. Quand ce quantum est écoulé, le scheduler choisit un autre processus et le processeur exécute les instructions de celui-ci pendant le quantum suivant et ainsi de suite. Le système comporte une horloge qui s'incrémente après chacun de ces quanta. Cette horloge déclenche alors une interruption qui appelle le scheduler. Celui-ci va alors, en fonction d'une certaine politique d'ordonnancement, choisir un processus dans la file d'attente et en relancera l'exécution.

3.1.3 La politique d'ordonnancement

Une technique souvent utilisée parce qu'elle est à la fois efficace et facile à implémenter est la suivante : les processus sont placés dans une file (FIFO). Chaque fois qu'un processus a passé un quantum de temps "dans" le processeur, il est remis *en fin de file* et le prochain processus qui va disposer du processeur est celui en tête de file. Cette technique, dite du "tourniquet" est illustrée sur la figure 3.1.

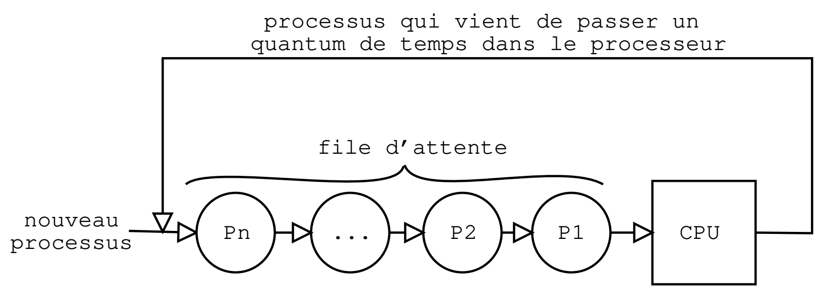

FIG. 3.1 – La technique du "tourniquet" dans un système généraliste

La théorie des files d'attentes nous apprend que le temps de réponse d'un tel système pour un processus dépend de la longueur de celui-ci. C'est-à-dire que plus un processus a un temps d'exécution long, plus le système mettra du temps pour l'exécuter complètement (le temps d'attente du processus dans le système est proportionnel à son temps d'exécution). Notons bien que ce temps de réponse comprend le temps que le processus aura passé dans la file d'attente. Ainsi, les processus courts ne sont pas défavorisés par rapport aux processus longs, ce qui est le cas d'une politique d'ordonnancement de type FCFS (*first come, first served*), par exemple. Dans le cas de cette politique, un processus court doit attendre que tous les processus qui sont arrivés avant lui soient exécutés pour l'être à son tour. Ce cas est similaire à ce qui se passe dans un supermarché quand on attend son tour à une caisse.

Quoi qu'il en soit, de nombreuses politiques d'ordonnancement peuvent être utilisées. Ce qu'il est important de remarquer ici, c'est que ces politiques sont non déterministes, ce qui veut dire, en fait, soit qu'a priori, tous les processsus ont la même *priorité* (dans le cas des politiques d'ordonnancement les plus simples), soit que celle-ci dépend du temps que le processus a déjà passé dans le système (à attendre ou à être exécuté). Autrement dit, personne ne peut dire avec certitude quel processus est en cours d'exécution à un moment donné. Vu que dans un système "temps réel", les "processus" sont en fait des suites d'instructions qui doivent être exécutées pour répondre à un événement matériel (la majorité d'entre eux en tous cas), on doit être capable de donner avec précision la succession des opérations qui ont lieu dans le système. C'est dans ce sens que le terme "déterminisme" doit être compris : à tout moment, si la succession des événements extérieurs est connue, on pourra prévoir quel est le processus qui s'exécute à un moment donné.

Imaginons, par exemple ce qui se passe dans le cas d'un microcontrôleur de clavier de PC. Deux processus sont à l'œuvre, principalement : le premier scrute les touches du clavier à intervalles *fixes* pour voir si l'utilisateur appuie sur l'une d'entre elles (le cas échéant, il place le code de cette touche dans un tampon d'émission) et le deuxième envoie sur une interface sérielle les codes qui se trouvent dans le tampon précité (si il n'est pas vide). On voit qu'une priorité plus grande doit être attribuée au premier processus car il faut impérativement éviter de rater l'appui sur une touche. Il ne faut pas que la scrutation des touches soit influencée par le trafic sur l'interface sérielle. Une horloge déclenche une interruption à intervalles fixes. Cette interruption déclenche l'exécution du premier processus indépendamment de ce que le processeur était en train de faire (soit rien, soit exécuter le second processus). Quel que soit l'enchaînement des événement externes, ont pourra toujours dire quel processus tourne à un moment donné. L'exigence de ce déterminisme dans le cas d'un système d'exploitation "temps réel" a conduit à l'élaboration de nouvelles architectures logicielles répondant à cette contrainte particulière.

Les informations données ici à propos des systèmes d'exploitation généralistes sont extraites de [2].

3.1.4 Définitions

Avant de nous pencher sur les architectures spécifiques aux systèmes "temps réel", nous devons d'abord définir de manière précise certains termes. Certains d'entres-eux sont utilisés dans de nombreux contextes et il est important de donner la signification exacte qu'ils prennent dans notre contexte :

- ◇ Temps de réponse (ou de latence) : c'est le temps qui peut s'écouler entre l'apparition d'un certain événement (au sens large) et l'exécution du traitement approprié. Cet événement va être en général le passage d'un périphérique du système dans un état qui requiert un traitement de la part du processeur. L'événement peut être signalé au processeur par le déclenchement d'une interruption par le périphérique. Dans d'autres cas, le processeur lui-même va devoir, par une consultation cyclique de l'état du périphérique, déterminer lui-même que l'événement a eu lieu.

- ⋄ Tâche : c'est un ensemble d'instructions qui doivent être exécutées en réponse à un événement. Si les événements sont signalés par des interruptions, le code des routines de traitement de ces interruptions n'est pas considéré comme faisant partie de la tâche correspondante. Cependant, le code de la routine de traitement de l'interruption d'un périphérique devra d'une manière ou d'une autre déclencher l'exécution du code de la tâche correspondante. Dans ce cas, le code à exécuter en réponse à un événement est, d'une part, le code de la routine d'interruption et, d'autre part, le code de la tâche dont on dira qu'ils sont, respectivement, urgent et non-urgent. Dans la littérature spécialisée, il semble que le terme "tâche" soit souvent utilisé dans un contexte "temps réel" à la place de "processus" qui semble souvent réservé aux systèmes d'exploitation généralistes. Nous utiliserons cependant l'un et l'autre indifféremment, sans distinction de sens.

- ⋄ Priorité : la priorité d'une tâche est en principe constante et unique. Elle représente le degré d'urgence de cette tâche. Elle est déterminée par le développeur de l'application en fonction du rôle de la tâche, de contraintes matérielles et logicielles, ...

- ⋄ Déterminisme : ce terme a ici un sens tout à fait particulier, comme nous l'avons déjà vu. On dira d'un système "temps réel" qu'il est déterministe parce qu'à tout moment, étant donné un enchaînement d'événement connu, on doit pouvoir dire de manière univoque quel est la tâche qui s'exécute dans le système. Ce déterminisme découle directement de l'unicité de la priorité de chaque tâche.

- ⋄ Scheduling (ou ordonnancement) : c'est le mécanisme qui permet de choisir et de démarrer (ou relancer) l'exécution d'une tâche dans le système.

- ⋄ kernel (ou noyau) : c'est le code qui fournit un ensemble de fonctionnalités particulières au programmeur d'une application informatique. La principale fonctionnalité que doit offrir le kernel est le scheduling déterministe.

- ⋄ scheduler (ou ordonnanceur) : code du kernel qui s'occupe du scheduling.

Nous allons voir maintenant quelles sont les principales architectures pour systèmes "temps réel" enfouis. Elles sont présentées dans [1].

3.1.5 L'architecture en boucle sans interruption

Le principe est le suivant : une boucle infinie teste successivement, pour chaque tâche, que l'événement correspondant *a eu* lieu. Si il s'est produit, la tâche qui lui est associée est exécutée, sinon on passe au test suivant. Aucun mécanisme d'interruption n'est utilisé. Le code suivant illustre ce principe.

```
void main(void) {

  for(;;) {

    if (needs_service(DEVICE1)) handle_dev(DEVICE1)

    if (needs_service(DEVICE2)) handle_dev(DEVICE2)
    ...
    if (needs_service(DEVICEn)) handle_dev(DEVICEn)
  }
}
```

où la fonction *needs_service()* rend TRUE si le périphérique dont elle a reçu le numéro en paramètre requiert un traitement particulier et FALSE sinon. La fonction *handle_dev()* effectue ce traitement pour le périphérique concerné.

Les avantages sont que :

- ce système est le plus simple à implémenter,
- comme les tâches s'exécutent en stricte alternance, ils n'y a aucun problème de communication entre elles. Elles peuvent consulter et modifier sans danger des variables qu'elles se partageraient, ces opérations s'effectuant d'office de manière atomique (puisque toute la tâche s'effectue sans être interrompue).

Les inconvénients sont les suivants :

- le temps de réponse à un événement est au pire le temps qu'il faut pour exécuter toutes les tâches (dans le cas où les événements associés aux autres tâches se seraient tous produits). Imaginons qu'un des périphériques du système demande un service toutes les millisecondes dans le pire des cas (c'est-à-dire que l'intervalle qui sépare deux demandes de service de ce périphérique peut être de 2 millisecondes mais pas de 0,5 milliseconde). Alors, si l'exécution la plus longue de la boucle principale dure disons, 3 millisecondes, le système ne fonctionnera pas correctement. Cette exécution *la plus longue* se produit quand tous les périphériques du système ont simultanément besoin d'être "servis" par le processeur,

- si on modifie le système, cela peut affecter les temps de réponses de toutes les tâches déjà implémentées. Si le programme de l'application est optimisé, on pourra utiliser un processeur moins rapide (et donc probablement moins cher). Quand le cas de l'exécution la plus longue de la boucle principale se présentera, ce processeur *minimal* n'exécutera que des instructions utiles de traitement de périphériques (il ne sera pas en attente). Mais si on modifie le système en ajoutant, par exemple, un périphérique et son code de traitement associé ou qu'on allonge le code, ce processeur ne conviendra plus ... Pire, si une seule routine de traitement d'un périphérique prend un temps qui est grand par rapport au temps de réponse maximal requis par l'application, l'architecture décrite ici ne fonctionnera pas du tout ! Si le périphérique le plus exigeant du système doit être servi dans un délai de maximum 2 millisecondes et qu'il existe une routine de traitement d'un des périphériques qui prend 2 secondes, on doit utiliser une autre architecture.

On voit que cette technique est à réserver aux systèmes simples et qui ne nécessiteront pas de modifications futures.

3.1.6 L'architecture en boucle avec interruptions

Le principe est le même que pour le système précédent mais ici, les opérations urgentes sont traitées par une routine d'interruption appropriée. Voici une implémentation simplifiée de cette architecture :

```
BOOL task1Activated = FALSE;
BOOL task2Activated = FALSE;
...
BOOL taskNActivated = FALSE;

void interrupt task1IntHandler(void) {

  /* code urgent qui traite la demande d'interruption du périphérique 1 */
  ...
  task1Activated = TRUE;
}

void interrupt task2IntHandler(void) {
```

```
  /* code urgent qui traite la demande d'interruption du périphérique 2 */
  ...
  task2Activated = TRUE;
}

...

void interrupt taskNIntHandler(void) {

  /* code urgent qui traite la demande d'interruption du périphérique N */
  ...
  taskNActivated = TRUE;
}

void main (void) {

  for(;;) {

    if (task1Activeted) {

      /* code non urgent relatif à l'interruption du périphérique 1 */
      ...
      task1Activated = FALSE;
    }

    if (task2Activeted) {

      /* code non urgent relatif à l'interruption du périphérique 2 */
      ...
      task2Activated = FALSE;
    }

    ...

    if (taskNActiveted) {

      /* code non urgent relatif à l'interruption du périphérique N */
      ...
      taskNActivated = FALSE;
    }

    /* code non urgent qui doit être exécuté de manière cyclique,
       indépendamment de toute interruption (par exemple la scrutation
       de l'état des touches d'un clavier)
```

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

L'avantage de cette approche est que maintenant, on peut diviser les traitements à effectuer en opérations urgentes et non-urgentes. Les opérations urgentes (effectuées dans les routines de traitement des interruptions) ont alors une priorité plus grande que les opérations non-urgentes (effectuées dans le corps de la boucle principale).

Mais il y a aussi des désavantages :

- les opérations non-urgentes ayant la même priorité, on retrouve le problème du temps de réponse long si une de ces opérations prend beaucoup de temps (par exemple, si une imprimante doit traiter un fichier Postscript, elle ne répondra plus pendant tout ce temps). De plus, on ne peut pas impunément placer des opérations non-urgentes longues dans les routines de traitement d'interruptions. En effet, cela placerait ces opérations au même niveau de priorité que les opérations réellement urgentes, ce que nous ne voulons pas parce que le temps de réponse du système est alors augmenté d'une manière qui peut être inacceptable,

- maintenant, les opérations de communication entre les tâches sont plus complexes à gérer. En effet, une routine d'interruption pouvant se déclencher à tout moment, des précautions doivent être prises quand une tâche manipule une variable partagée avec d'autres.

Le pire temps de réponse du système pour un périphérique correspond au scénario suivant : l'interruption déclenchée par ce périphérique survient juste après que le test de la variable booléenne correspondant à ce périphérique ait eu lieu (la variable taskNActivated pour le périphérique N) et que cette variable ait été évaluée à FALSE. En plus, toutes les interruptions des autres périphériques se sont aussi déclenchées et leurs codes correspondants ont positionné à TRUE les variables d'activation d'opérations non-urgentes associées. Le temps de réponse est alors la somme du temps de traitement de toutes les autres opérations non-urgentes des autres périphériques (auquel il faut aussi ajouter le temps d'exécution de toutes les routines de traitement d'interruptions qui peuvent se produire, mais on suppose ce temps court).

3.1.7 Le système d'exploitation "temps réel"

Puisque dans un système *généraliste*, un processus peut mettre un temps quelconque pour s'exécuter, et que celui-ci peut être long, le système doit s'assurer que tous les processus disposent du (ou des) processeurs de manière équitable. C'est en fait cette caractéristique qui implique que le temps d'exécution d'un processus n'est pas prévisible. La décision d'un changement de contexte se fait sur une base temporelle (le quantum de temps) suffisamment petite pour donner l'illusion que les processus s'exécutent en même temps. La priorité des tâches va donc varier en fonction du temps (plus un processus aura passé du temps dans le processeur et plus sa priorité diminuera) pour assurer l'équité entre les processus.

Par contre, dans un système "temps réel", des séquences d'instructions doivent être exécutées à l'occasion d'événements externes dans des délais strictement définis. Ces événements sont caractérisés par le degré d'urgence du traitement approprié. Ces traitements (qu'on appelle souvent *tâches* plutôt que *processus*) vont alors être caractérisés par leur *priorité*. C'est cette priorité que le scheduler va utiliser pour choisir la tâche qui va disposer du processeur lors du changement de contexte. La priorité des tâches ne dépend pas du temps. En règle générale, comme on veut que le système se comporte de manière déterministe, chaque tâche aura un priorités différente de celles de toutes les autres tâches.

3.2 Cahier des charges du système d'exploitation

Après avoir donné, dans les grandes lignes, une idée des différentes architectures utilisables, voyons maintenant ce que nous attendons de notre système d'exploitation particulier...

3.2.1 Processus et parallélisme

Nous voulons répartir la difficulté de résolution d'un problème en l'implémentant sous la forme de plusieurs processus. Les avantages par rapport à une programmation séquentielle sont entre autres :

1. une meilleure utilisation des ressources : quand un processus attendra une réponse d'un périphérique, le processeur pourra exécuter un autre processus ; le processeur sera donc utilisé d'une manière optimale,

2. une meilleure organisation du programme : chaque processus remplit une fonction précise ; le programme est plus lisible et plus facile à maintenir (mais pas forcément plus facile à déboguer !),

3. une plus grande indépendance par rapport au matériel, liée à la modularité du système.

4. plus de facilité lors de l'interconnexion de différentes machines : les mécanismes qui seront mis en œuvre pour faire communiquer entre eux les processus qui fonctionnent sur une même machine pourront être étendus de manière à ce que deux processus tournant sur deux machines physiquement différentes puissent communiquer facilement. Dans la pratique, en effet, de nombreux systèmes enfouis sont interconnectés au sein d'un même appareil (ordinateur personnel, fusée, robot industriel,...).

La fonctionnalité principale que doit remplir notre système est donc de répartir les ressources entre plusieurs tâches. A tout moment, on veut que la tâche la plus prioritaire (et qui n'est pas dans un état d'attente), soit en cours d'exécution. Nous appellerons cette fonctionnalité "scheduling". Comme nous l'avons déjà vu, ce scheduling doit être, de plus, totalement déterministe, c'est à dire tel qu'il n'y ait aucun doute lors du choix de la tâche à exécuter.

Nous voulons aussi que notre système soit écrit en langage "C" qui est plus lisible et plus portable que l'assembleur.

De plus, chaque appel système devra reprendre dans sa documentation le temps maximal exact qu'il met pour s'exécuter.

3.2.2 Communication et synchronisation entre les processus

Les processus devront pouvoir s'échanger des informations à l'aide d'un mécanisme de mémoire partagée. Les problèmes d'accès concurrent à cette *mémoire partagée* et aux autres ressources non préemptibles (comme les périphériques, par exemple) seront réglés grâce à l'utilisation de *sémaphores*.

3.3 Limitations matérielles et solutions possibles

La description des différentes architectures logicielles pour les systèmes "temps réel" était extraite de la littérature. Par contre, les limitations du PIC vont nous obliger à trouver des solutions *originales* qui nous permettrons d'atteindre nos objectifs. Certaines d'entre-elles sont malgré tout inspirées d'articles parus dans des revues ou disponibles sur Internet. Cela sera signalé, le cas échéant.

3.3.1 Les limitations du PIC

Comme il a déjà été dit à plusieurs reprises, la limitation majeure du PIC est que sa pile ne comporte que huit niveaux. Pire : cette pile n'est pas adressable par le programme du système : elle n'apparaît pas dans l'espace des données du système. Seules des adresses de retour peuvent être sauvées par l'instruction d'appel de sous-routine (call) et dépilées par les instructions de retour de sous-routines (retfie, retlw et return). On n'a pas à notre disposition une instruction d'empilage (PUSH) ou de dépilage (POP).

Pourquoi est-ce si problématique ? Pour le comprendre, nous allons décrire un scénario plausible de ce qui pourrait se passer dans un système d'exploitation quand le contrôle du processeur doit passer d'un processus à un autre. Imaginons que le processus A était en cours d'exécution et que le kernel a pris le contrôle du processeur. Ce transfert de contrôle a pu avoir lieu pour deux raisons : soit une interruption a eu lieu et celle-ci a appelé une fonction du kernel, soit le processus A a appelé lui-même une fonction du kernel. Dans les deux cas, le kernel va déterminer quel est le processus exécutable de priorité la plus élevée. Rappelons-nous qu'un appel système peut avoir rendu exécutable un processus qui était en attente et éventuellement de priorité supérieure à celle du processus A. Dans ce cas, le kernel va devoir effectuer ce qu'on appelle un "changement de contexte".

Un contexte représente l'état d'un processus, il inclut les contenus des registres du processeur et de la mémoire utilisée par le processus. Pour effectuer un changement de contexte, le kernel va sauver l'état des registres tels qu'ils étaient lorsque le processus courant a été interrompu. Il va les charger avec les valeurs correspondant à celles qu'avaient ces registres quand le processus dont on doit relancer l'exécution a été interrompu. Le kernel stocke toutes ces informations dans une structure qui lui est propre et qui contient un emplacement par processus et par registre. On suppose ici que tous les processus utilisent des zones de mémoire séparées ce qui fait que, bien que faisant partie du contexte (au sens large) d'un processus, la mémoire qu'il utilise n'a pas besoin d'être sauvée par le kernel.

Le registre dont le contenu est le plus important est le compteur de programme qui indique l'adresse de l'instruction courante exécutée par le processeur. *Si le contenu qu'avait ce registre, au moment où le processus a été interrompu, est inaccessible au kernel, ce dernier ne pourra pas lui rendre ultérieurement le contrôle.* Or, cette information est justement sauvée sur la pile du système lorsqu'une interruption se produit ou qu'une instruction d'appel de sous-routine est exécutée. Puis, le registre compteur de programme est chargé avec l'adresse de la routine à exécuter.

Mais sur le PIC, comme le code ne peut accéder au contenu de la pile, le scénario que nous venons de décrire n'est pas réalisable. Le système ne pourra pas être préemptif, c'est-à-dire que le kernel ne pourra pas retirer le contrôle à un processus indépendamment de la volonté de ce dernier. Il va donc falloir imaginer des techniques qui permettent de contourner cette difficulté ...

Une autre limitation importante vient du fait que le PIC présente une architecture matérielle qui rend ardue la manipulation de pointeurs de fonction. En effet, l'architecture utilisée sépare le code (mots de 14 bits) des données (mots de 8 bits). Le compilateur que j'ai utilisé (PICC de Hi-Tech Software), manipule de manière erronée de tels pointeurs, même si ils sont constants (ce qui aurait permis de les calculer lors de la compilation). Cela va nous empêcher, par exemple, de disposer d'une table de pointeurs sur les différentes tâches.

Il faut aussi rappeler ici que le PIC ne possède en fait qu'un seul vecteur de routine de traitement d'interruption matérielle. Dès que celle-ci s'est produite, le code de traitement doit déterminer quel est le périphérique qui l'a réellement déclenchée (polling). L'ordre de ces tests déterminera quelle priorité est attribuée à chaque interruption de périphérique ...

Nous allons voir à présent les techniques possibles d'implémentation de notre système qui tiennent compte des limitations du PIC.

3.3.2 Gestion des processus par la routine d'interruption

Ce mécanisme va utiliser la routine de traitement de l'interruption du timer (aussi appelé temporisateur). Lorsqu'un processus est interrompu par cette interruption, l'adresse de sa prochaine instruction à exécuter est sauvée sur la pile du processeur. On ne peut connaître cette adresse mais on sait qu'elle est sur la pile. La routine du timer va alors sauver le contexte du processus (le registre W et le registre des indicateurs) dans une pile séparée. Cette pile est explicitement gérée par le système d'exploitation, elle pourra être en RAM interne ou externe. Notons que, pour simplifier, on ne sauve pas d'autres informations relatives au processus interrompu. On suppose que chaque processus dispose d'un jeu de registres bien précis qui lui sont réservés sauf bien sûr ceux qu'il partage explicitement avec d'autres (sémaphores, zones de mémoire partagée,...). Si il faut qu'un processus soit réentrant, un traitement particulier devra être fait.

Une fois que toutes les informations utiles concernant le processus sont sauvées, le scheduler va choisir un autre processus à exécuter mais attention : il faudra que le processus en question ne soit pas déjà en cours d'exécution ! En effet, la seule adresse du processus connue par le scheduler est son adresse de début. Les processus de ce type doivent donc être vus comme des sous-routines. Les processus vont s'exécuter d'une manière qui fait penser à une poupée russe. Finalement, lorsque la pile du processeur sera pleine, le scheduler n'appellera pas de nouveau processus et rendra le contrôle au dernier à avoir été interrompu. Il finira bien par se terminer, rendant alors le contrôle grâce à un instruction jmp, au scheduler[1]. Le scheduler pourra alors restaurer le contexte de l'avant dernier processus et exécuter une instruction[2] pour lui rendre le contrôle.

Avantages

Cette approche ne peut s'envisager que dans un contexte non déterministe, dans le sens où ce terme a été défini plus haut (le processus exécutable le plus prioritaire ne sera pas d'office celui qui tournera sur le système). En effet, la notion de priorité des processus "temps réel" disparaît : l'utilisation de la pile matérielle empêche de relancer un processus qui s'y trouverait déjà. On ne peut pas avoir un processus fort prioritaire arrêté (et placé sur la pile) en faveur d'un autre processus moins prioritaire qui pourrait lui rendre la main *avant* d'être terminé car cela impliquerait que le processus plus prioritaire occuperait deux niveau sur la pile ce qui mênerait à un crash rapide du système.

Donc, puisqu'on est dans un contexte non déterministe, on utilise alors le système des quanta de temps.

Si un processus met moins de temps pour s'exécuter que la durée d'un quantum, il ne sera pas interrompu et s'exécutera donc d'un coup ! Les processus qui étaient avant lui dans la file verront leur exécution commencer plus tôt mais si leur exécution s'est prolongée jusqu'à ce qu'il démarre, il se terminera nécessairement avant eux. On pourrait appeler cette politique d'ordonnancement LCFS (*Last Come First Served*) ! Avec cette méthode, on voit qu'on a plus de garanties sur les conditions temporelles d'exécution des processus.

Un autre avantage est que le programmeur de l'application ne doit pas se préoccuper de rajouter du code lié au scheduling : les processus sont interrompus automatiquement.

Pourquoi présentons-nous cette méthode puisqu'elle ne convient pas à un système "temps réel" ? En fait, dans un système de ce type, lorsqu'aucun processus n'est en cours d'exécution, le processeur pourrait très bien exécuter des tâches qui ne sont pas soumises aux contraintes "temps réel". La technique décrite ici pourrait alors être utilisée pour le scheduling de ces tâches.

---

[1] qui l'avait lancé également à l'aide d'une instruction jmp de manière à ne pas comsommer inutilement des entrées sur la pile

[2] return ou plus judicieusement retlw qui fera d'une pierre deux coups puisqu'elle chargera W en plus de dépiler le sommet de la pile dans le compteur de programme.

Désavantages

Ce système n'est pas équitable pour les différents processus : si l'un d'eux est interrompu, tous ceux qui arriveront après lui devront avoir été exécutés complètement avant qu'il ne puisse continuer son exécution. L'implémentation n'est pas facile non plus car la routine de traitement d'interruption doit être réentrante. Le nombre de processus est aussi limité. Si on interdit à tous les processus du système d'exécuter des instructions *call*, le nombre de processus du deuxième type sera limité à huit dans le meilleur des cas ! Bien sûr, un mécanisme d'appel de sous-routine pourra être simulé par une macro spécifique et une pile gérée explicitement par le système d'exploitation ...

Restrictions et exemple d'utilisation

Chaque appel de sous-routine par une tâche consomme potentiellement un niveau de pile. Le programmeur doit donc déterminer quelle sera l'utilisation maximale de la pile dans le pire des cas. Si, par exemple, l'application utilise trois tâches A, B et C, que la tâche A peut appeler une sous-routine qui elle-même peut en appeler une, que la tâche B n'appelle pas de sous routines, que la tâche C peut appeler consécutivement deux sous-routines et que les trois tâches peuvent être exécutables en même temps, on aura besoin dans le pire des cas de $3 + 1 + 2 = 6$ niveaux de pile. En effet, le pire des cas apparaît quand A, B et C ont été interrompus. De plus A a été interrompu dans le code de la sous-routine appelée par celle qui a été appelée par A (3 niveaux de pile). B a été interrompu (1 niveau) et C a été interrompu au cours de l'exécution d'une des deux sous-routines qu'il appelle (2 niveaux). Si on imagine que B était le premier processus à s'exécuter sur le système, puis que A a été lancé et enfin C, voici l'aspect de la pile quand C est interrompu (par une routine d'interruption) :

adresse de retour dans la routine appelée par C	niveau 5
adresse de retour dans C	niveau 4
adresse de retour dans la routine appelée par la routine appelée par A	niveau 3
adresse de retour dans la routine appelée par A	niveau 2
adresse de retour dans A	niveau 1
adresse de retour dans B	niveau 0

Voici à quoi ressemblerait le code du “scheduler” et de l'application de notre exemple à 3 processus :

```
BOOL already_in_A = FALSE;
BOOL already_in_B = FALSE;
BOOL already_in_C = FALSE;

/* routine appelée par dummy 1 (donc indirectement par A) */
int dummy2() {
  return 2;
}

/* routine appelée par A */
int dummy1() {
  return (dummy2());
}

/* routine appelée par C */
int dummy3() {
  return 3;
}

void interrupt sched(void) {

  if (periphAready() && !already_in_A) {

/* code de A */

    already_in_A = TRUE;
    ...
    dummy1();
    ...
    already_in_A = FALSE;
  }

  if (periphBready() && !already_in_B) {

/* code de B */

    already_in_B = TRUE;
    ...
    already_in_B = FALSE;
  }

  if (periphCready() && !already_in_C) {
```

```
/* code de C */

    already_in_C = TRUE;
    ...
    dummy3();
    ...
    dummy3();
    ...
    already_in_C = FALSE;
  }
}

void main(void) {

/* initialisations diverses */
  ...
/* autorisation des interruptions */
  ...

  for(;;);
}
```

Les fonctions “periphXready()” rendent TRUE si et seulement si le périphérique correspondant doit être servi. Les variables “already_in_X” ne sont nécessaires que si il y a un viol potentiel des contraintes “temps réel”. En effet, dans ce cas un périphérique pourrait demander à être servi alors que le traitement correspondant à sa demande précédente n’est pas encore terminé. On remarquera aussi que dans ce cas, il faut prévoir un niveau de pile supplémentaire (passage dans la routine d’interruption sans service d’aucun périphérique dans le cas où toutes les “tâches” sont déjà actives).

Remarquons aussi qu’une tâche ne peut redémarrer que si les tâches qui l’ont interrompue se soient toutes terminées. Cela rend extrêmement rigide une éventuelle communication entre deux tâches ...

3.3.3 Les tâches cycliques et événementielles

C’est la technique qui correspond au schéma logiciel le plus simple que nous ayons vu ; elle est inspirée d’un article paru dans [4]. Le programme est divisé en tâches qui doivent s’exécuter à intervalle régulier et d’autres qui doivent s’exécuter une seule fois lorsqu’un certain événement a lieu. Chaque tâche est en fait un sous-programme que le scheduler appellera ou pas selon que la tâche soit à activer ou non. Cette information est représentée par un bit. Le scheduler parcourt une boucle. Dans cette boucle, il regarde pour chaque tâche si son bit d'activation est positionné. Si c'est le cas, pour une tâche cyclique, il l'appelle. Elle s'exécute sans être interrompue et rend le contrôle au scheduler. Pour une tâche liée à un événement, c'est pareil, mais après son exécution, le scheduler remet son bit d'activation à 0. Le bit est positionné par une autre tâche ou par une routine d'interruption lorsque la condition d'activation de la tâche est (à nouveau) satisfaite.

Tel quel, ce mécanisme ne consomme pas de niveau de pile. Il peut aussi fonctionner sans interruption. Dans ce cas, des tests dans la boucle principale détermineront si des périphériques doivent être servis ou pas (architecture de la boucle sans interruption). Si on utilise les interruptions, les routines de traitement de celles-ci positionneront le bit de la tâche à effectuer pour satisfaire le périphérique qui a généré l'interruption (architecture de la boucle avec interruption).

Avantages

L'avantage principal est la simplicité d'implémentation et la faible quantité d'informations requises (1 seul bit par tâche !).

Désavantages

Malheureusement, on ne dispose pas de la souplesse d'un vrai système d'exploitation. On n'a pas de parallélisme, donc les temps de latences vont dépendre du temps d'exécution de la tâche la plus longue. Quand une tâche est en attente d'un périphérique, le processeur ne peut être utilisé pour exécuter autre chose. Bref, ce mécanisme est trop simpliste pour être satisfaisant. Néanmoins, il peut être intéressant, notamment pour les tâches cycliques.

Restrictions et exemple d'utilisation

Toutes les tâches ayant la même priorité, il faut prendre en compte la durée maximale d'exécution de la boucle principale (avec exécution de toutes les tâches) pour calculer le temps de latence. La pile du système est totalement libre pour des appels de sous-routines. On pourra donc avoir 8 appels imbriqués si on n'utilise pas les interruptions. Sinon, il faudra en tenir compte sans oublier les interruptions en cascade dans le cas où on utilise une routine de traitement d'interruption réentrante. Voici un exemple de système programmé de cette manière :

```
const unsigned char bitPower[] = {1, 2, 4, 8, 16, 32, 64, 128};
```

```
unsigned char Tasks = 0;  /* char donc 8 tâches max (1 bit/tâche)
                             aucune tâche active au début */

void interrupt intHdl(void) {

  if(periph1ready()) Tasks |= bitPower[0];
  if(periph2ready()) Tasks |= bitPower[1];
  ...
  if(periph8ready()) Tasks |= bitPower[7];

}

void main(void) {

  for(i = 0; i < 8; i++) {

    if(Tasks & bitPower[i]) {
      switch(i) {
        case 0 : /* code tâche 1 */
                 break;

        case 1 : /* code tâche 2 */
                 break;
        ...

        case 7 : /* code tâche 8 */
                 break;
      }
      Task &= ~bitPower[i]; /* remet le bit de la tâche à 0 */

  }
}
```

Remarquons que nous n'avons pas utilisé de tâches cycliques et que la routine de traitement d'interruption ne fait ici rien d'autre qu'activer les tâches (code non urgent). On pourrait bien sûr faire tout cela sans restriction particulière.

3.3.4 collaboration des processus (utilisation de l'assembleur)

La première approche qui vient à l'esprit est de faire participer les processus au travail que doit effectuer le scheduler. Puisque ce dernier ne peut pas savoir où se trouvait le processus qui a été interrompu, le processus va lui-même donner cette information au scheduler. Cela semble impossible car, en effet, comment le processus peut-il savoir *quand* il sera interrompu ? La réponse est simple : le processus va s'interrompre lui-même, volontairement ! Le programmeur placera, à certains endroits clé du processus, des appels au scheduler précédés de la sauvegarde de la position courante du processus. Cette sauvegarde se fera dans un structure de données gérée par le système d'exploitation. Cette structure ne sera bien sûr pas la pile du processeur vu que celle-ci présente les lacunes évoquées plus haut et aussi parce qu'une structure de type pile ne convient pas ici. En effet, si on utilisait une pile (LIFO, *last in, first out*), le scheduler ne pourrait pas retrouver les informations concernant le processus à relancer sans dépiler les informations concernant les autres processus qui ont été partiellement exécutés depuis le dernier arrêt de celui-ci ...

Voyons dans quels cas un changement de contexte doit avoir lieu ...

Dans le cadre d'un système "temps réel", contrairement au cas d'un système généraliste, l'écoulement d'un quantum de temps ne doit pas donner lieu à un changement de contexte car ce n'est pas cet écoulement de temps qui va changer les priorités des processus dans le système. Voici les cas où ce changement peut avoir lieu :

- un processus se termine,
- un processus effectue un appel système qui peut le mettre en attente (wait sur un sémaphore, par exemple),
- un processus effectue un appel système qui peut relancer un processus plus prioritaire (signal sur un sémaphore),
- une routine de traitement d'interruption effectue un appel système dont l'effet correspond à un des trois cas précités.

Dans les autres cas, la priorité du processus qui est en train d'être exécuté est la plus élevée (parmi les processus qui sont exécutables). Le scheduler rend alors tout de suite le contrôle au processus qui l'a appelé sinon, il choisit le nouveau processus de priorité la plus élevée, récupère sa position courante et relance son exécution à partir de ce point. le schéma 3.2 illustre cette technique que j'appellerai la technique du "caillou" en imaginant que le processus dépose un caillou à l'endroit où il se trouve pour la désigner au kernel. Tel le Petit Poucet, le système retrouve son chemin grâce aux cailloux, déposés par les processus, quand il s'est perdu dans le kernel !

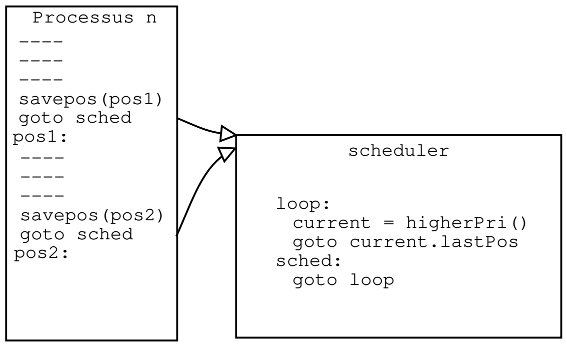

FIG. 3.2 – la technique du "caillou"

C'est cette technique que nous allons mettre en œuvre.

Malheureusement, dans notre cas, cette technique va devoir être implémentée de manière plus subtile. En effet, le choix du langage "C" comme langage de programmation va poser quelques problèmes. C'est pourquoi nous verrons une deuxième méthode, similaire quant au principe, qui est adaptée au "C".

Avantages

Cette technique permet de réaliser un système parfaitement déterministe.

De plus, l'implémentation de cette méthode est relativement simple. La tâche du programmeur de l'application finale est aussi simplifiée : le système d'exploitation met à sa disposition une fonction (ou plutôt une macro car l'appel d'une fonction consomme un niveau de pile qui peut être précieux). Cette macro permet au processus de sauver sa position courante et d'appeler le scheduler. La structure de données qui permet de sauver ces informations pour tous les processus n'est pas soumise à des contraintes matérielles fortes (ce qui est le cas, par exemple de la pile du processeur). Seule sa taille est limitée si on utilise la RAM interne du PIC. Et précisément, cette contrainte peut être levée grâce à l'utilisation d'une RAM externe. Celle-ci est certes plus lente d'accès mais son utilisation permettrait de gérer un grand nombre de processus. Cette RAM sera implémentée sous forme de module de telle sorte qu'en fonction de l'application finale, on pourra choisir de l'utiliser ou pas, selon le nombre de processus qui devront s'exécuter *en même temps* sur le système.

Désavantages

Le système n'est plus préemptif dans le sens où le scheduler est appelé par les processus. Si une routine d'interruption exécutait un appel système qui devait conduire à un changement de contexte, celui-ci n'aurait pas lieu tant que le processus en cours d'exécution n'ait fait un appel au scheduler. Dans le cadre d'un système "temps réel", cela n'est absolument pas gênant car les processus ont un code figé en ROM (en général). Par conséquent on peut supposer qu'ils sont corrects et ont un comportement prévisible.

Un autre désavantage manifeste (plus ennuyeux) est que le programmeur final doive se préoccuper des temps qui peuvent s'écouler entre deux appels successifs au scheduler dans le code des tâches qu'il écrit. En effet, pendant ces moments, un processus exécutable de priorité potentiellement plus élevée que celui qui s'exécute devra attendre l'appel au scheduler du processus en cours d'exécution. De même, cela allonge virtuellement les temps d'exécution des tâches et il faudra en tenir compte quand on vérifiera que le système respecte ses contraintes "temps réel".

Restrictions et exemple d'utilisation

La programmation en assembler sur PIC est relativement fastidieuse : il faut en permanence modifier des bits d'adresse dans le registre STATUS pour accéder aux variables ou à des routines se trouvant dans des bancs différents, les instructions de type RISC sont peu puissantes, ...

Mis à part ce fait, une difficulté supplémentaire apparaît : le programmeur des tâches devra espacer les appels au scheduler de façon à trouver un compromis entre l'augmentation artificielle du temps de latence du système et le temps que le système passera dans le scheduler. Nous allons clarifier ce point qui peut paraître obscur.

Imaginons que le programmeur place un appel au scheduler après chaque instruction utile de son code. Supposons, de plus, que chaque instruction prenne un cycle d'horloge pour s'exécuter (c'est le cas de la plupart des instructions du PIC) et que la préparation de l'appel au scheduler et l'exécution de celui-ci (déterminer la tâche la plus prioritaire et (re)lancer son exécution) prenne, disons, 100 cycles. Dans ce cas de figure, le système sera virtuellement préemptif parce que dès qu'une tâche plus prioritaire que celle qui s'exécute est rendue exécutable, le système s'en rendra compte très vite. Une tâche ne pourra exécuter qu'une instruction utile en n'étant éventuellement pas la plus prioritaire à un moment donné. Mais le problème ici est que le système passe pratiquement cent pour cent du temps dans le scheduler ! Ce n'est pas acceptable.

Si au contraire, le programmeur place trop peu d'appels au scheduler, les temps d'exécutions apparents des tâches du système seront fortement augmentés. Le programmeur devra ajouter au temps d'exécution de toutes les tâches le temps maximal qui peut s'écouler entre deux appels au scheduler. Remarquons que les temps de latence n'augmentent pas. En effet, le temps de latence pour un événement donné est le temps qui s'écoule entre l'apparition de cet événement et l'exécution du code approprié. Or, ce code (urgent) est placé dans une routine de traitement d'interruption, d'office plus prioritaire que la plus prioritaire des tâches.

Voici un exemple de code utilisant un scheduler basé sur la technique décrite ici :

```
intHdl ; routine de traitement d'interruption

SAVE_CONTEXT ; macro qui sauve le contexte du
; processeur
call  evt1_occured ; positionne CARRY si l'événement 1
; a eu lieu
btfsc  STATUS,CARRY ; si CARRY vaut 0, on saute au
; dessus de l'instruction goto
goto  activate1

call  evt2_occured
btfsc  STATUS,CARRY
goto activate2
...
call  evtN_occured
btfsc  STATUS,CARRY
goto  activateN

exit_isr
RESTORE_CONTEXT ; macro qui restaure le contexte du
; processeur
retfie ; retour de la routine d'interruption

activate1
movlw 1
call startTask
goto exit_isr
```

```
activate2
movlw 2
call startTask
goto exit_isr
...

activateN
movlw N
call startTask
goto exit_isr


task1
...
movlw pos1_1 ; on place la position courante
; dans l'accumulateur (W)
goto sched
pos1_1
...
movlw pos1_2
goto sched
pos1_2
...
goto exitTask

...

taskN
...
movlw posN_1
goto sched
posN_1
...
movlw posN_2
goto sched
posN_2
...
goto exitTask
```

J'ai volontairement omis le code nécessaire au changement de banc de RAM, qui est requis lorsqu'on accède successivement à des variables se trouvant dans des bancs différents. On remarque que l'appel aux "fonctions" *sched* et *exitTask* du kernel se font par saut ("goto") au lieu d'un appel de sous-routine ("call"). Dans le cas de *sched*, nous avons vu pourquoi nous procédons de la sorte : le scheduler doit reprendre le contrôle sans consommation inutile d'un niveau de pile et de plus le scheduler ne pourrait pas restaurer la pile correctement vu qu'il n'y a pas un accès direct. L'information de la position courante du processus qui aurait été sauvée sur la pile par l'instruction d'appel de sous-routine n'aurait été d'aucune utilité au scheduler ! Cette information de position du processus est donc passée au scheduler dans l'accumulateur du PIC (W). Le scheduler stocke cette information dans une table et, lorsqu'il devra rendre le contrôle à ce processus, il utilisera cette information pour exécuter une instruction de saut au bon endroit dans le code de celui-ci. Le cas de *exitTask* est similaire. L'utilisation d'une instruction d'appel de sous-routine aurait posé un problème au kernel : il aurait dû en effet supprimer l'entrée de pile consommée par l'instruction d'appel ("call"), ce qui est impossible sur un PIC, vu que seule une instruction de retour de sous-routine peut dépiler un élément. L'exécution se poursuivrait alors dans le code du processus qui voulait ce terminer ce qui n'est certainement pas ce que nous voulons.

On peut trouver dans [5] un exemple concret de l'utilisation d'une méthode similaire.

3.3.5 collaboration des processus (utilisation du "C")

L'utilisation du langage C présente de nombreux avantages par rapport à l'utilisation du langage d'assemblage :

- une plus grande lisibilité du code (certaines personnes disent du code C qu'il est de l'*assembleur lisible* ce qui est d'autant plus vrai dans le cas de l'utilisation d'un processeur RISC, comme c'est le cas ici),

- la portabilité : si un compilateur C existe pour une plateforme donnée, on peut facilement réutiliser le code générique déjà écrit sur une autre plateforme. L'opération de portage est d'autant plus facilitée que les programmeurs ont placé le code générique (indépendant de la plateforme cible) et le code spécifique à une plateforme donnée dans des modules séparés,

- le code C rend la maintenance (modifications ultérieures) beaucoup plus aisée que le code assembleur,

- les compilateurs modernes sont pratiquement tous dotés de capacités d'optimisations avancées qui leur permettent de produire du code machine aussi efficace que celui produit par un être humain. De plus, dans les cas (rares) où le recours à l'assembleur est absolument nécessaire, ces compilateurs fournissent la possibilité d'écrire des séquences d'assembleur en ligne (directement dans le fichier source C).

Comme un compilateur C existe pour les microcontrôleurs PIC, aucun obstacle ne s'oppose à ce qu'on bénéficie des avantages précités. Cependant, lorsqu'on veut implémenter en C la solution en assembleur vue au point précédent, des difficultés apparaissent, comme nous allons le constater ...

La première idée à laquelle on pense est la suivante : on sépare le code des tâches en sections commençant par un label. A la fin de chaque section, on effectue un appel au scheduler en lui passant le label qui marque le début de la section suivante. La première difficulté est que le compilateur utilisé[3] gère très mal les pointeurs sur du code exécutable. De plus, et c'est plus grave, un appel au scheduler qui donnerait lieu à un changement de contexte ne libérerait pas l'emplacement de pile utilisé pour sauver l'adresse de retour correspondant à cet appel. La pile (qui, rappelons-le, ne contient que huit emplacements), se remplirait très vite. Pire, le scheduler devrait, lui aussi, effectuer un appel de fonction pour passer le contrôle à la nouvelle tâche (toujours sous l'hypothèse qu'un changement de contexte ait lieu). On voit donc que cette technique est impraticable.

La parade semble évidente : une tâche utilise l'instruction "goto" du C pour donner le contrôle au scheduler. Ce dernier fait la même chose pour relancer l'exécution d'une tâche où elle avait été interrompue. C'est en fait l'approche utilisée dans la version en assembleur de cette solution mais, bien que viable techniquement, on ne peut l'utiliser en C sans déstructurer complètement le code du programme...

Les difficultés que je viens d'évoquer m'ont amené à réfléchir au problème de manière plus approfondie. La solution qui a émergé de ces cogitations paraîtra peut-être triviale, une fois révélée. Il m'a cependant fallu plusieurs jours d'intense réflexion pour la découvrir !

La solution est la suivante : les tâches sont appelées par le scheduler à l'aide d'un appel de fonction C classique (le mécanisme est un peu moins simple mais ne compliquons pas les choses pour l'instant). Le scheduler passe à la tâche un numéro de position qui correspond à l'endroit où elle doit reprendre son exécution. Le code de la tâche commence par une instruction "switch" qui reçoit cette position sous forme de paramètre. Cela lui permet de sauter au bon endroit au sein de son code. Lorsque la tâche décide d'appeler le scheduler, elle effectue une instruction "return". Toute l'astuce réside

[3]PICL de Hi-Tech software, voir "système de développement" en annexe

dans ce point précis ! Ainsi, le scheduling ne requiert qu'un seul niveau de pile.

Bien sûr, le même mécanisme est utilisé pour les différents appels système que le kernel met à la disposition des tâches. Lorsque la tâche exécute l'instruction "return", elle renvoie au kernel un pointeur sur une structure[4] qui contient un champ spécifiant l'appel système demandé. Le scheduler réalise cet appel pour la tâche demandeuse, puis il recherche quelle est la tâche exécutable la plus prioritaire (l'appel système peut avoir rendu exécutable une tâche plus prioritaire que celle qui s'est interrompue), et relance son exécution.

Au démarrage, la fonction principale *main* rend exécutable au moins une tâche à l'aide d'un appel direct de la fonction appropriée (startTask) et appelle le scheduler. Celui-ci est constitué d'une boucle infinie. C'est au sein de cette boucle que le scheduler appelle la tâche la plus prioritaire, et récupère son code de retour qui correspond à un appel système à effectuer. Il le réalise puis la boucle reprend, sans fin.

Pour faciliter le travail du programmeur de l'application finale, les appels système que peuvent effectuer les tâches sont définis à l'aide de macros. Celles-ci remplissent correctement les différents champs de la structure d'échange d'information entre la tâche et le kernel et effectuent le "return" évoqué plus haut.

On pourrait se demander si il est nécessaire que les tâches appellent le scheduler si elles ne doivent pas demander un service au kernel. En effet, on a affaire ici à un système déterministe où chaque tâche a une priorité unique. Seul un appel système peut rendre à nouveau une tâche exécutable et de ce fait rendre éventuellement un rescheduling nécessaire. C'est vrai, mais le problème est qu'un appel système peut être placé dans une routine de traitement d'interruption. Or aucune opération de changement de contexte ne peut prendre place au sein d'une telle routine, vu les limitations du PIC déjà largement évoquées. C'est ce qui oblige les tâches à appeler régulièrement le scheduler de manière à minimiser les instants où une tâche qui n'est pas la tâche exécutable la plus prioritaire est en cours d'exécution. Remarquons que les routines de traitement d'interruption ne pourront en aucun cas se servir des macros décrites plus haut pour effectuer des appels systèmes ; elles appelleront les fonctions système *directement*.

Comme le kernel ne connaît les tâches que par leur numéro, la méthode la plus élégante pour que ce dernier puisse les appeler était d'utiliser une

[4]voir le fichier common.h décrit en annexe

fonction écrite par le programmeur de l'application finale (qui écrit le code des tâches). Cette fonction, nommée "dispatch", est utilisée par le kernel quand celui-ci veut relancer une tâche. Il passe à cette fonction son numéro. La fonction "dispatch" appelle alors la fonction adéquate. On verra plus tard comment cette fonction peut être programmée pour régler les problèmes d'inversion de priorité sans avoir à utiliser des appels systèmes spéciaux (coûteux en taille de code), la place nous étant comptée dans la minuscule mémoire ROM des PIC !

Comme la solution évoquée ici a été utilisée pour programmer notre kernel, nous lui consacrons une section complète.

3.4 L'implémentation du kernel

Cette section a pour but de décrire dans le détail le fonctionnement de notre kernel. Nous allons d'abord illustrer la manière générale dont l'ensemble est mis en œuvre puis nous détaillerons l'implémentation de chaque appel système. Ensuite, nous terminerons cette section par un exemple théorique qui illustrera la manière d'utiliser les fonctionnalités du kernel. Un chapitre entier sera consacré à la description d'une application réelle utilisant ce kernel.

3.4.1 Les différents modules et leurs dépendances

A l'instar de la réalisation matérielle, le kernel a été conçu de manière extrêmement modulaire. Un module regroupe les fonctions liées au scheduling et un autre les fonctions liées à l'utilisation des sémaphores.

La figure 3.3 illustre les dépendances entre les différents fichiers de l'application. Un flèche part d'un module qui inclut le fichier où elle aboutit.

Voici les rôles des différents fichiers :

- "user.c" : contient le code des différentes tâches ainsi que la fonction principale de l'application (la fonction *main*). Ce fichier est réalisé par le programmeur de l'application.

- "user.h" : ce fichier doit contenir les prototypes des fonctions représentant les tâches, codées dans "user.c".

- "dispatch.c" : contient la fonction *dispatch* programmée par l'utilisateur. Elle est appelée par le kernel pour (re)lancer l'exécution des

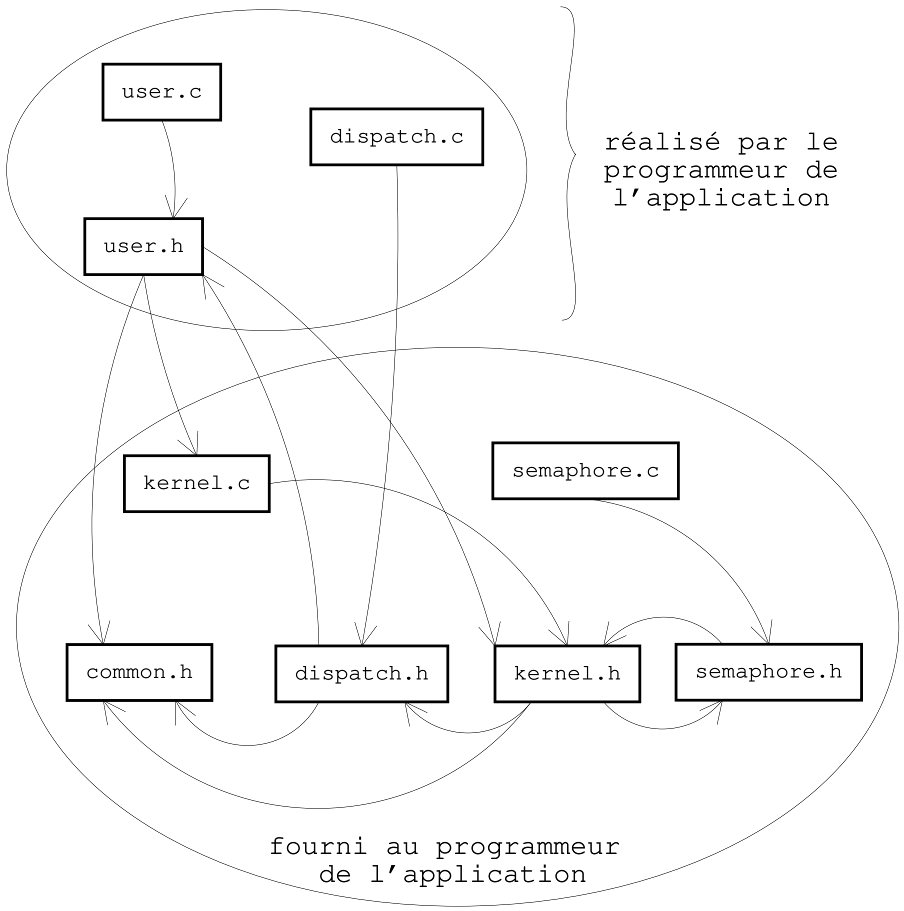

FIG. 3.3 – les dépendances entre les fichiers du projet

tâches. Nous verrons en détail les critères que doit remplir cette fonction.

- "dispatch.h" : contient le prototype de la fonction *dispatch*. Ce fichier ne doit pas être modifié par le programmeur de l'application.

- "kernel.c" : contient toutes les fonctions essentielles du kernel, c'est-à-dire celles liées au scheduling.

- "kernel.h" : définit les macros et les prototypes de fonctions utilisables par le code de l'application ainsi que les fonctions utilisées par les autres modules du "système d'exploitation" tel que le module qui s'occupe de la gestion des sémaphores. Ce fichier contient aussi des macros qui permettent le paramétrage de compilation du code du kernel. Ces macros permettrons, par exemple, de ne pas implémenter les fonctions ayant trait aux sémaphores, si on n'en a pas besoin. On gagnera ainsi une place précieuse dans la ROM du PIC. Ce fichier peut donc être modifié par le programmeur de l'application de manière à ce qu'il obtienne un kernel "sur mesure". Cependant, il est préférable de ne pas modifier ce fichier mais plutôt de définir des macros par l'intermédiaire de la ligne de commande de compilation qui invoque le compilateur C utilisé. Ainsi, par exemple, si on veut pouvoir utiliser sept sémaphores, on utilisera la ligne de commande suivante (en supposant que le compilateur s'appelle "gcc") :

```
gcc *.c -Wall -DMAX_SEM=7 -o kernel
```

- "common.h" : ce fichier définit le type de la structure d'échange d'informations entre les tâches (user.c) et le kernel (kernel.c) ainsi que le type de la structure des priorités de tâches qu'utilisent le module principal du kernel (kernel.c) et le module des sémaphores (sema.c). Il ne doit pas être modifié.

- "sema.c" : contient toutes les fonctions liées à la manipulation de sémaphores. Les fonctions qui se trouvent dans ce fichier ne seront pas intégrées au programme final si la macro MAX_SEM a une valeur nulle.

- "sema.h" : contient les prototypes des fonctions liées à la manipulation de sémaphores.

3.4.2 Les états des tâches

Il est intéressant de discuter ici des différents états que peuvent prendre les tâches dans notre système, avant de détailler les fonctions qui manipuleront ces états. La figure 3.4 montre dans quels états une tâche peut se trouver.

Il faut deux bits pour coder ces états : un bit sera utilisé de manière globale pour indiquer si la tâche est exécutable ou pas. Un deuxième bit qui existera en autant d'exemplaires qu'il y a de sémaphores définis, indiquera si la tâche est en attente sur le sémaphore correspondant. Le système s'assurera qu'une tâche ne peut être à la fois exécutable et en attente sur un sémaphore. Les bits ont la même valeur pour les états "exécutable" et "en cours d'exécution". Le kernel stocke simplement dans une variable le numéro de la tâche courante.

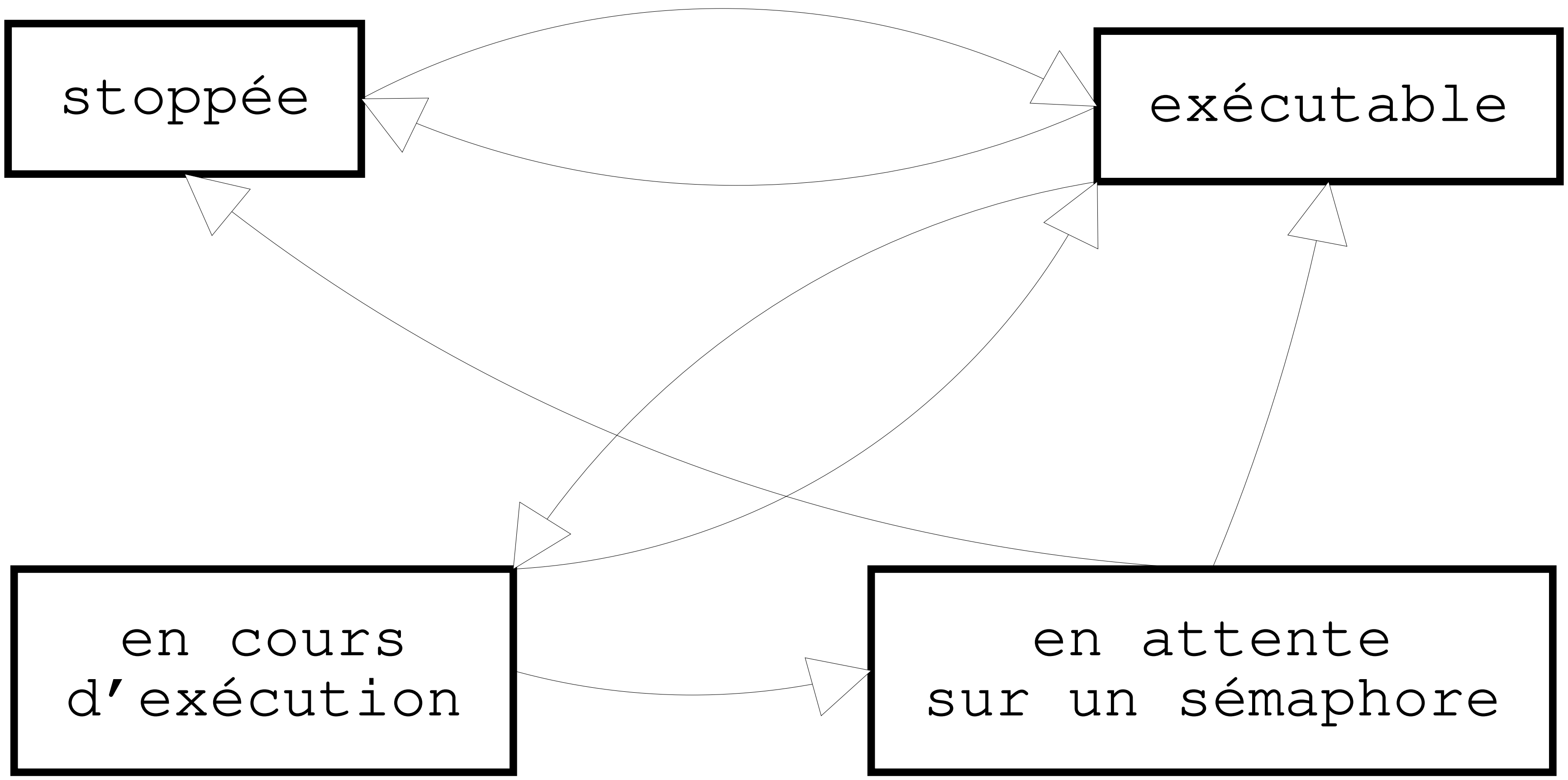

FIG. 3.4 – les états des tâches

3.4.3 Les positions des tâches

Comme nous l'avons déjà expliqué dans la section qui décrit la solution logicielle utilisée ici, les tâches sont codées sous forme de fonctions C et reçoivent en paramètre la position courante du code qu'elles doivent exécuter. Le code d'une tâche ressemble à ceci :

```
taskParmStruc taskParm;

taskParmStruc *task(unsigned char pos)
{
  switch(pos)
    {
```

```
    case 0:
...
      SCHEDULE(taskParm, 1);

    case 1:
...
      SCHEDULE(taskParm, 2);

...
    case n:
      EXIT_TASK(taskParm, 0, 6);   /* 0 est ici le numéro
      de la tâche à terminer et 6
      une position quelconque */
}
```

Au début, la tâche est rendue exécutable grâce à l'appel, dans la fonction *main* ou dans une routine de traitement d'interruption, de la fonction *startTask*. La tâche sera appelée avec un paramètre de position à 0. La tâche va alors exécuter son code à partir de cette position jusqu'à ce qu'elle exécute une macro fournie par le kernel. Dans le cas de la macro SCHEDULE, la tâche rend simplement le contrôle au kernel en lui indiquant que lorsqu'elle reprendra le contrôle, elle désire que ce soit à la position 1. Le kernel va déterminer quelle est la tâche la plus prioritaire qui doit obtenir le contrôle et l'appellera en lui fournissant sa position courante. Quand la tâche dont on discute ici recevra à nouveau le contrôle, elle reprendra son exécution à la position 1 et ainsi de suite. Finalement, elle effectuera un appel à la macro EXIT_TASK, en demandant au kernel sa terminaison.

Le cas de l'utilisation de boucles est plus délicat. La technique illustrée ci-dessus utilise l'instruction C bien connue *switch*. Si la tâche contient une boucle (*for, do while, while*) dont le temps d'exécution est suffisamment court pour prendre place entre deux appels au scheduler, il n'y a pas de problème :

```
taskParmStruc *task(unsigned char pos)
{
  int i;

  switch(pos)
    {
    case 0:
...
      for(i=0; i<10; i++)
```

```
      {
        ...
      }
...
      SCHEDULE(taskParm, 1);

    case 1:
...
```

Si, par contre, le temps de l'exécution de la boucle est trop long et qu'il faut, lors de certaines itérations de la boucle, appeler le scheduler, un problème apparaît. En effet, on ne peut pas imbriquer une instruction *switch* avec le corps d'une boucle :

```
taskParmStruc *task(unsigned char pos)
{
  static int i; /* variable statique pour qu'elle
       conserve son contenu lors de chaque éxécution
   d'une portion de code de la boucle */

  switch(pos)
    {
    case 0:
...
      for(i=0; i<10; i++)
      {
...
      SCHEDULE(taskParm, 1);

    case 1:  /* imbrication non permise */
        ...
      }      /* for(i=0; i<10; i++) */
...
```

Il faut alors se résoudre à utiliser une autre technique. Bien que ce soit une instruction peu élégante en C, l'instruction *goto* va nous apporter la solution. Voici un exemple :

```
taskParmStruc *task(unsigned char pos)
{
  static unsigned char i = 0; /* remarquez la classe static */

  switch(pos)
    {
    case 0:
      i = 0;
    loop1:              /* label pour l'utilisation de goto */
      i++;

...
      SCHEDULE(task1Parm, 1);   /* retour à ``case 1:'' */

    case 1:
      if (i < 9) goto loop1;    /* on boucle grâce à ``goto'' */
      SCHEDULE(task1Parm, 2);   /* retour à ``case 2:'' */

    case 2:
...
```

Donc, quand l'exécution d'une boucle doit être interrompue par des passages de contrôle au kernel (pour le rescheduling mais aussi pour d'autres opérations), on doit construire la boucle à l'aide de l'instruction *goto*. De plus, les variables qui interviennent dans le gardien d'une telle boucle doivent impérativement être de classe *statique* pour ne pas perdre leur valeur d'une prise de contrôle de la tâche à la suivante. Cela vaut d'ailleurs aussi pour toutes les autres variables locales de la tâche à l'exeption, justement, des variables qui servent uniquement à garder les boucles classiques (*for*, ...), ces boucles s'exécutant de manière atomique (si on ne prend pas en compte les interruptions, bien sûr), étant donné qu'elles sont intercalées entre deux appels consécutifs au kernel.

3.4.4 Les fonctions implémentées dans le module principal du kernel

La fonction *schedule*

Le système démarre réellement quand la fonction *main* appelle la fonction *schedule*. Cette dernière, comme nous l'avons déjà expliqué, parcourt une boucle infinie. A chaque itération, elle détermine quelle est la tâche exécutable la plus prioritaire en appelant *higherTask*. Le kernel mémorise la position courante de chaque tâche dans un tableau : *taskBuf*. Ensuite, *schedule* appelle la fonction *dispatch* fournie par le programmeur de l'application. Elle lui fournit comme paramètres la position courante de la tâche qui doit s'exécuter et sa position courante. *Dispatch* va appeler à son tour la fonction correspondant à la tâche dont le numéro lui a été fourni. La tâche a donc maintenant le contrôle. Après un certain temps, celle-ci exécutera une instruction “return”, après avoir préparé les champs de la structure qui lui premet d'échanger des informations avec le kernel. Cette structure est du type *taskParmStruc*. C'est ce type qui est défini dans le fichier “common.h”. Deux champs dans cette structure doivent impérativement être remplis avant que la tâche ne rende le contrôle au scheduler. Il s'agit du champ *pos* qui contient la nouvelle position de la tâche et que le scheduler devra communiquer à la tâche lors de son prochain appel. L'autre champs s'appelle *sysCallCode*. La tâche indique, par la valeur qu'elle lui donne, le service qu'elle attend du kernel. Le service qui sera le plus souvent demandé est simplement le rescheduling.

Pour que le programmeur des tâches n'ait pas à manipuler directement la structure d'échange (ce qui serait fort peu élégant), nous mettons à sa disposition des macros qui font ce travail pour lui. Celles-ci sont définies dans le fichier “kernel.h”. Toutes ces macros fonctionnent selon un même principe : elles modifient des champs de la structure d'échange et effectue un “return” qui rend le contrôle à la fonction *dispatch* (qui avait appelé la tâche). *Dispatch* rend à son tour le contrôle à *schedule*. Passons ces macros en revue :

- SCHEDULE(p, n) : demande au kernel de rescheduler. n est la position à laquelle la tâche demande que le contrôle lui soit rendu. p est l'adresse de la structure d'échange.

- EXIT_TASK(p, t, n) : demande au kernel de terminer l'exécution de la tâche t. n est la position à laquelle la tâche demande que le contrôle lui soit rendu. p est l'adresse de la structure d'échange. Si la tâche qui invoque cette macro est celle qui doit se terminer, la valeur de n est sans importance.

- START_TASK(p, t, n) : demande au kernel de démarrer l'exécution de la tâche t. n est la position à laquelle la tâche demande que le contrôle lui soit rendu. p est l'adresse de la structure d'échange. La tâche qui est démarrée voit sa position courante mise à une valeur nulle.

- WAIT(p, s, n) : cette demande correspond à la classique opération d'attente sur un sémaphore de numéro s. Si p est différent de 0, la valeur du sémaphore est décrémentée, sinon, la tâche est mise en attente (elle perd son statut *exécutable*). n est la position à laquelle la tâche demande que le contrôle lui soit rendu. p est l'adresse de la structure d'échange. Remarquez que pour disposer de cette fonctionnalité, la macro MAX_SEM doit avoir une valeur différente de zéro. C'est le cas aussi pour les trois opérations qui suivent.

- SIGNAL(p, s, n) : effectue l'opération "signal" sur le sémaphore s : si il n'y a aucune tâche en attente sur ce sémaphore, sa valeur est simplement incrémentée, sinon la tâche la plus prioritaire qui était en attente est rendue à nouveau exécutable et la valeur du sémaphore reste inchangée. n et p ont le même sens que pour les macros précédentes.

- GET_SEM(p, s, n) : la tâche qui utilise cette macro récupèrera la valeur du sémaphore s dans le champ *val* de la structure d'échange dont elle donnera l'adresse (p). n a le même sens que précédemment.

  SET_SEM(p, s, v, n) : cette opération est la symétrique de la précédente. Elle permet à une tâche de modifier directement la valeur d'un sémaphore s en lui assignant la valeur v. n et p ont le même sens que pour les macros précédentes.

La fonction *schedule* récupère le numéro d'appel système à effectuer (les macros l'on modifié en conséquence). Elle exécute alors le code correspondant au service demandé en utilisant certaines fonctions que nous détaillerons par la suite.

Ces explications sont en principe suffisantes pour que le code de la fonction *schedule* ne présente plus le moindre mystère. Ce dernier, ainsi que le code des autres fonctions sont fournis en annexe. Notez que dans le code source, les modalités d'utilisation de chaque fonction sont abondamment commentées. On y trouve, pour chacune d'elles, les paramètres d'entrée, la valeur de retour ainsi qu'une description de son rôle. Nous donnons aussi le temps approximatif[5] d'exécution de ces fonction en nombre de cycles d'exécution du PIC.

La fonction *startTask*

Cette fonction fait tout simplement passer la tâche, dont on lui fournit le numéro, à l'état exécutable. Elle s'exécute en 25 cycles maximum.

[5]le compilateur PICC de Hi-Tech Software a été utilisé avec l'option d'optimisation (-Zg). Les temps d'exécution ont été déterminés par l'examen du fichier assembleur généré par le compilateur

La fonction *killTask*

Cette fonction fait passer la tâche requise à l'état "stoppée". De plus, elle remet sa position courante à 0. Elle va aussi modifier les structures de données du kernel de manière à ce que la tâche ne soit plus répertoriée dans les ensembles de tâches qui attendent sur les différents sémaphores. Elle s'exécute en 1000 cycles maximum.

La fonction *stopTask*

Cette fonction fait passer la tâche donnée à l'état "non-exécutable". Elle s'exécute en 30 cycles maximum.

La fonction *restartTask*

Cette fonction fait l'opération contraire de la précédente : elle fait passer une tâche dans l'état "exécutable", sans modifier sa position. Elle s'exécute en 15 cycles maximum.

La fonction *higherTask*

Cette fonction retourne la valeur de la tâche exécutable la plus prioritaire présente dans la structure de priorité qu'on lui fournit. Elle utilise une astuce décrite dans [12] pour optimiser la recherche de la tâche la plus prioritaire dans le cas où le nombre maximum de tâches a été fixé à 64. Huit octets sont utilisés pour stocker les 64 bits représentant les états des tâches dans la structure de priorité. Un neuvième octet contient un bit pour chacun des huit octets précédents. Un "1" dans un de ces bits signifie que l'octet correspondant contient au moins un bit à "1" (voir la figure 3.5).

Dans le cas où le nombre maximum de tâches est de huit, un seul octet est nécessaire, ce qui est très intéressant dans le cas du PIC16F84, où chaque octet de RAM est précieux.

Elle s'exécute 160 cycles maximum.

La fonction *setTaskBit*

Cette fonction positionne le bit "exécutable" de la tâche donnée dans la structure appropriée du kernel. Elle s'exécute en 10 cycles maximum.

La fonction *clearTaskBit*

Cette fonction met à zéro le bit "exécutable" de la tâche donnée. Elle s'exécute en 10 cycles maximum.

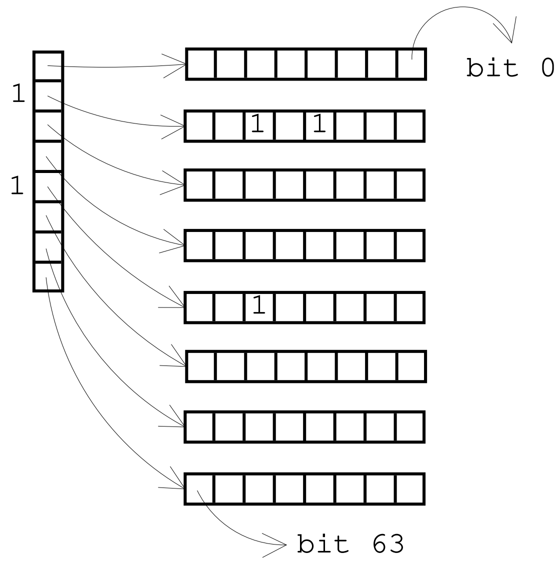

FIG. 3.5 – la structure de priorité des tâches

3.4.5 Les fonctions implémentées dans le module gérant les sémaphores

Les fonctions qui suivent sont fort simples. Leur code étant, de plus, abondamment commenté, on peut s'y reporter pour avoir plus de détails. Nous nous limiterons ici à une description de leurs rôles respectifs.

La fonction *getSemValue*

Cette fonction retourne la valeur d'un sémaphore donné. Elle s'exécute en 20 cycles maximum.

La fonction *setSemValue*

Cette fonction positionne la valeur d'un sémaphore donné. Elle s'exécute en 20 cycles maximum.

La fonction *higherWaitingTask*

Cette fonction détermine la tâche de plus haute priorité qui attend sur un sémaphore donné. Elle s'exécute en 170 cycles maximum.

La fonction *addWaitingTask*

Cette fonction ajoute une tâche à l'ensemble des tâches qui attendent sur un sémaphore donné. Elle s'exécute en 30 cycles maximum.

La fonction *wait*

Cette fonction réalise l'opération "wait" sur un sémaphore donné. Si la valeur du sémaphore est nulle, la tâche est mise en attente. Si elle n'est pas nulle, elle est décrémentée et la tâche peut poursuivre son exécution. Elle s'exécute en 60 cycles maximum.

La fonction *signal*

Cette fonction réalise l'opération "signal" sur un sémaphore donné. Si il y a une tâche en attente sur le sémaphore, celle-ci est relancée. Sinon, la valeur du sémaphore est incrémentée. Elle s'exécute en 230 cycles maximum.

3.5 Le problème de l'inversion de priorité

Ce problème se pose quand un processus de priorité élevée doit attendre un processus de priorité faible (le processus de faible priorité doit réaliser une opération SIGNAL sur un sémaphore pour débloquer le processus de priorité élevée qui est arrêté dans une opération WAIT sur ce sémaphore). Si un processus de priorité intermédiaire s'exécute, il empêche le processus de faible priotité de débloquer le processus de priorité élevée. On dit alors qu'il y a inversion de priorité entre le processus de priorité intermédiaire et le processus de priorité élevée. la figure 3.6 illustre le problème.

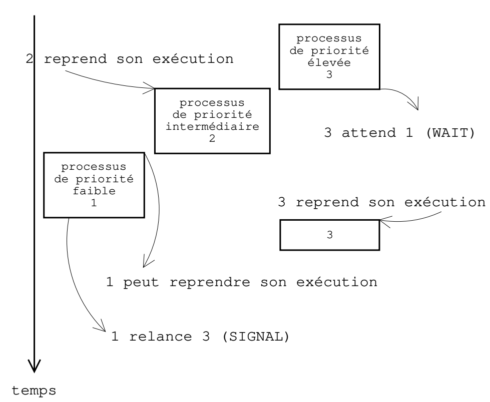

FIG. 3.6 – l'inversion de priorité

Le programmeur de l'application peut contourner ce problème grâce à une modification *ad-hoc* de la fonction *dispatch*. En effet, c'est cette fonction, et elle uniquement, qui fait le lien entre une priorité et une tâche. Le code de cette fonction pourra alors être écrit pour modifier dynamiquement les priorités des tâches. Bien sûr, cette technique doit être utilisée avec précaution.

3.6 Un exemple d'utilisation

Voici un exemple théorique qui illustre la manière d'utiliser le kernel. Deux tâches sont présentes dans le système. La fonction *main* instancie la tâche de numéro 1. celle-ci, après quelques itérations dans une boucle, démarre l'exécution de la tâche numéro 2, qui est plus prioritaire. Cette deuxième tâche prend alors le contrôle du processeur. Elle effectue aussi quelques itérations puis se place en attente sur le sémaphore numéro 0, celui-ci ayant une valeur nulle. La tâche 1 reprend immédiatement son exécution. Ensuite, cette dernière va effectuer un opération “signal” sur le sémaphore 0, ce qui va relancer la tâche 2. Lorsque celle-ci se terminera, la tâche 1 continuera son exécution jusqu'à ce qu'elle se termine. La tâche 2 teste aussi l'opération SET_SEM : elle force la valeur du sémaphore 0 à 48, puis effectue un WAIT ce qui a pour effet de faire passer la valeur du sémaphore à 47. La tâche 1 lira cette valeur grâce à GET_SEM. Voici le code correspondant du fichier “user.c” :

```
#include "user.h"


taskParmStruc task1Parm;
taskParmStruc task2Parm;

taskParmStruc *task1(unsigned char pos)
{
  static unsigned char i = 0;

  switch(pos)
    {
    case 0:
      i = 0;
    loop1:
#ifdef WSDEBUG
      printf("task 1 is at pos %d, loop iteration number %d\n",pos,i);
#endif
      i++;
```

```
      if (i == 5)
        {
          START_TASK(task1Parm, TASK_2, 1);
        }

      if (i == 7)
        {
          SIGNAL(task1Parm, 0, 1);
        }

      SCHEDULE(task1Parm, 1);

    case 1:
      if (i < 9) goto loop1;
      SCHEDULE(task1Parm, 2);

    case 2:
      GET_SEM(task1Parm, 0, 3);

    default:
#ifdef WSDEBUG
      printf("Sem 0 value is %d\n", task1Parm.val);
      printf("task 1 is about to exit\n");
#endif
      EXIT_TASK(task1Parm, TASK_1, 3);
    }
}

taskParmStruc *task2(unsigned char pos)
{
  static unsigned char i = 0;

  switch(pos)
    {
    case 0:
      i = 0;
    loop2:
#ifdef WSDEBUG
      printf("task 2 is at pos %d %d\n",pos,i);
#endif
      i++;

      if (i == 3)
        {
```

```
          WAIT(task2Parm, 0, 1);
        }

      SCHEDULE(task2Parm, 1);

    case 1:
      if (i < 9) goto loop2;
      SCHEDULE(task2Parm, 2);

    case 2:
      SET_SEM(task2Parm, 0, 48, 3);

    case 3:
      WAIT(task2Parm, 0, 4);

    case 4:
      GET_SEM(task2Parm, 0, 5);

    case 5:
#ifdef WSDEBUG
      printf("Sem 0 value is %d\n", task2Parm.val);
#endif
      SIGNAL(task2Parm, 0, 6);

    default:
#ifdef WSDEBUG
      printf("task 2 is about to exit\n");
#endif
      EXIT_TASK(task2Parm, TASK_2, 6);
    }
}

int main()
{
  startTask(TASK_1);
  //startTask(TASK_2);
  schedule();
  return 0;
}
```

Remarquez l'utilisation abondante de l'instruction *goto* et la manière d'invoquer les opérations du kernel par les tâches en utilisant des macros. La fonction *main*, par contre appelle directement les fonctions du kernel. Les macros ne peuvent être utilisées que par les tâches, évidemment : n'oublions pas qu'elles dissimulent une instruction *return*.

Le fichier *include* du programme de l'application (user.h) définit les prototypes des fonctions des deux tâches :

```
#ifndef __USER_H

#define __USER_H


#include "kernel.h"
#include "common.h"

#define TASK_1 1
#define TASK_2 2

taskParmStruc *task1(unsigned char pos);
taskParmStruc *task2(unsigned char pos);

#endif
```

La fonction *dispatch* fait le lien entre un numéro de tâche et la fonction correspondante :

```
#include "dispatch.h"
#include <stdlib.h> /* exit() */

/*----------------------------------------------------------------
 *
 *----------------------------------------------------------------
 */
taskParmStruc *dispatch(unsigned char task, unsigned char pos)
{
  switch(task)
    {
    case TASK_1:
      return task1(pos);
    case TASK_2:
      return task2(pos);
    default:
      printf("dispatch: unknown task %d\n", task);
      exit(-1);
    }
}
```

L'exécution de ce programme sur un système doté d'un périphérique d'affichage donne ceci :

```
task 1 is at pos 0, loop iteration number 0
task 1 is at pos 1, loop iteration number 1
task 1 is at pos 1, loop iteration number 2
task 1 is at pos 1, loop iteration number 3
task 1 is at pos 1, loop iteration number 4
task 2 is at pos 0 0
task 2 is at pos 1 1
task 2 is at pos 1 2
Task 2 is waiting on semaphore 0
task 1 is at pos 1, loop iteration number 5
task 1 is at pos 1, loop iteration number 6
Task 2, previously waiting on semaphore 0, is restarted
task 2 is at pos 1 3
task 2 is at pos 1 4
task 2 is at pos 1 5
task 2 is at pos 1 6
task 2 is at pos 1 7
task 2 is at pos 1 8
Sem 0 value is 47
task 2 is about to exit
task 1 is at pos 1, loop iteration number 7
task 1 is at pos 1, loop iteration number 8
Sem 0 value is 48
task 1 is about to exit
```

Chapitre 4

Exemple d'application pratique

Comme nous l'avons vu, la deuxième version de la réalisation matérielle présentée ici a été utilisée pour un autre travail [3]. L'idée naturelle qui vient alors à l'esprit est d'utiliser l'objectif de ce dernier travail pour illustrer l'utilisation de notre kernel. Dans un premier temps, nous présenterons succintement cet objectif ; ce dernier sera modifié de manière à utiliser toutes les fonctionnalités importantes du kernel. Ensuite, nous donnerons une implémentation, en insistant sur tous les points délicats auxquels un éventuel concepteur d'application qui utiliserait notre kernel devrait faire attention.

4.1 Objectifs

L'objectif du travail [3] était de mesurer la vitesse d'une locomotive miniature et d'envoyer cette information par une interface sérielle. De plus, la locomotive devait évoluer sur une voie à deux extrémités. Donc, elle devait effectuer en permanence des allers et retours sur celle-ci. Ce mouvement de va-et-vient était commandé par le PIC qui détectait l'arrivée de la locomotive aux extrémités de la voie. Le PIC profitait du passage de la locomotive sur les capteurs pour mesurer la vitesse de celle-ci. Il préparait alors un paquet de données qu'il envoyait sur l'interface sérielle.

Notre solution à ce problème utilisait l'architecture en boucles avec interruptions. Cette dernière était parfaitement adaptée. Ici, nous allons compliquer un peu les choses de manière à ce que l'utilisation des fonctionnalités d'un système d'exploitation "temps réel" soit pratiquement nécessaire pour résoudre le problème, ou, du moins, faciliter le travail de manière sensible.

Voici le problème que je propose : deux locomotives doivent se déplacer d'une gare A à un gare B, en respectant un horaire qui leur est propre. Chaque gare possède deux voies de garage, ce qui permet aux deux locomotives de se trouver dans la même gare en même temps. Chacune de ces voies sera associée à une locomotive, c'est-à-dire qu'un train arrive et part toujours de la même voie dans une gare donnée, comme c'est généralement le cas dans une situation réelle. La difficulté provient du fait que les deux gares sont reliées par une voie unique. Le système doit s'assurer qu'une seule locomotive au plus soit présente sur cette voie pour éviter une collision. On suppose que la voie n'est constituée que d'un canton, ce qui empêche deux locomotives se dirigeant vers la même gare d'emprunter la voie unique l'une à la suite de l'autre. La deuxième locomotive devra attendre que la première ait atteint la gare de destination pour pouvoir démarrer. La figure 4.1 illustre le problème.

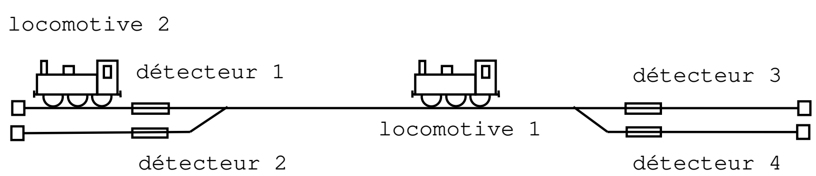

FIG. 4.1 – le problème des trains

Notre système devra donc réaliser les opérations suivantes :

- envoyer sur la voie les informations de vitesse et de direction des locomotives,
- envoyer sur cette même voie les ordres de commutation des aiguillages,
- s'assurer que les locomotives utilisent la voie unique qui relie les deux gares en exclusion mutuelle,
- et enfin : faire exécuter aux locomotives les trajets entre les deux gares en respectant à celles-ci un temps d'arrêt prédéfini.

4.2 Implémentation

Avant de se pencher sur la manière dont le logiciel a été réalisé, il peut être bon de considérer certains aspects matériels qui n'ont pas encore été éclaircis. Le synoptique du système se trouve sur la figure 4.2.

4.2.1 Considérations matérielles

La description qui suit n'a pas sa place dans la section qui décrit le module d'interfaçage I^2C vers encodeur sériel parce qu'elle est spécifique à notre

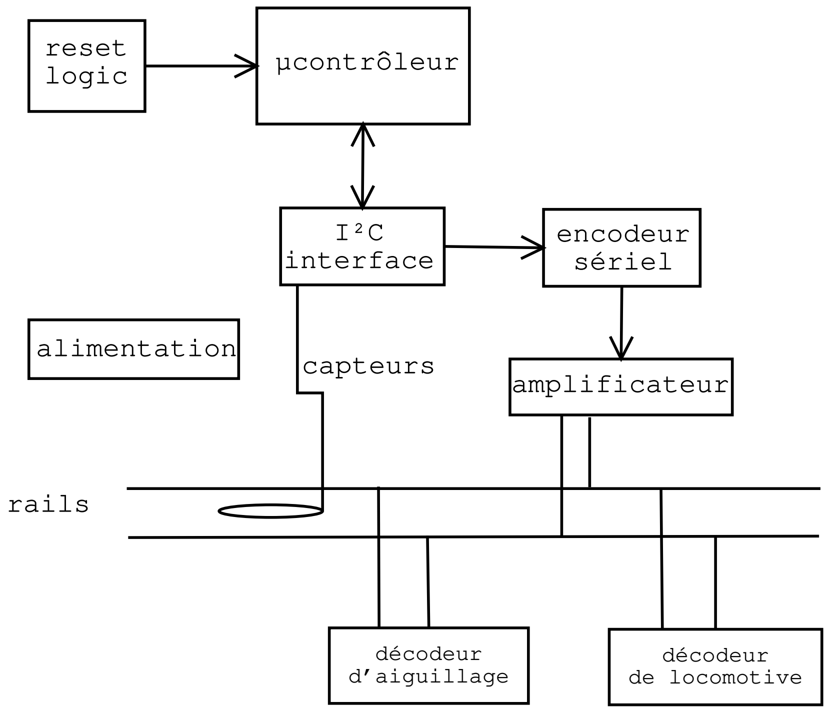

FIG. 4.2 – le synoptique

application. En effet, on peut connecter toutes sortes de périphériques à cet encodeur sériel. Son utilisation pour la commande d'un système de pilotage de trains miniatures n'en est qu'un exemple.

L'encodeur sériel de Motorola (le 145026), qui se trouve sur un de nos modules décrit dans le chapitre relatif au matériel, est utilisé de la manière suivante : il encode les neufs "trits" d'informations qui lui sont fournis par le PIC à travers une inteface I^2C (PCF8474). Un "trit" peut prendre trois valeurs. Au niveau matériel, ces trois valeurs sont représentées par 5 volts (1 logique), 0 volt (0 logique) ou une haute impédance (2 logique). Le circuit d'interfaçage I^2C ne peut pas placer ses sorties à l'état de haute impédance, car, même quand on configure ses lignes d'entrée/sortie en entrée, un système de "pull up actif" interne fournit à la broche en question un courant de 100 microampères environ : la ligne n'est donc pas à haute impédance. Les 5 premiers "trits" sont utilisés pour coder l'adresse du récepteur qui se trouve connecté aux rails (récepteur d'aiguillage ou de locomotive). Vu que nous n'utilisons que deux des trois valeurs possibles pour ces "trits" (qui, du coup, deviennent des bits), nous pouvons adresser 2^5 décodeurs, donc 32. Les quatre autres "trits", eux, servent de données (ici aussi, seules deux valeurs sur les trois possibles sont utilisées. Les décodeurs de locomotive utilisent ces bits comme information de vitesse. Ils utilisent en fait le cinquième bit d'adresse comme bit de fonction. Les décodeurs d'aiguillage, par contre, utilisent les cinq bits d'adresse comme tels et les quatre derniers comme information de commutation. La transmission de l'information se fait de la manière suivante : les neuf unités d'informations sont codées à l'aide d'une suite d'impulsion dont la largeur donne la valeur à coder. Il faut deux impulsions par "trit". La transmission d'un mot prend 3,8 millisecondes. Ce temps est déterminé par les valeurs des composants externes connectés à l'oscillateur du 145026. le protocole prévoit l'émission de tout mot en deux exemplaires. Un récepteur ne considère un mot qui lui est destiné comme correct que si il a reçu deux fois ce mot consécutivement. La pause entre chaque mot est fixée à 2 millisecondes. Donc, transmettre une information à un récepteur prend $((3,8+2)*2)ms$ c'est-à-dire 11,6 millisecondes. Nous aurons besoin de cette information par la suite.

Vu l'utilsation par les décodeurs de locomotives du cinquième bit d'adresse comme bit de fonction, on ne pourra en fait pas avoir 32 décodeurs connectés au système : l'équation $16 = nombre\_de\_dcodeurs\_de\_locomotives + 2 * nombre\_de\_dcodeurs\_d'aiguillages$, doit être respectée. Un décodeur d'aiguillage peut commander quatre aiguillages.

Les informations qui précèdent ont été tirées de [7] et [8].

4.2.2 Le programme de l'application

Les processus de pilotage des locomotives

L'idée est d'utiliser un processus par locomotive. Chacun de ces processus connaîtra à tout moment la position de la locomotive qu'il dirige. Ces états sont indiqués sur la figure 4.3.

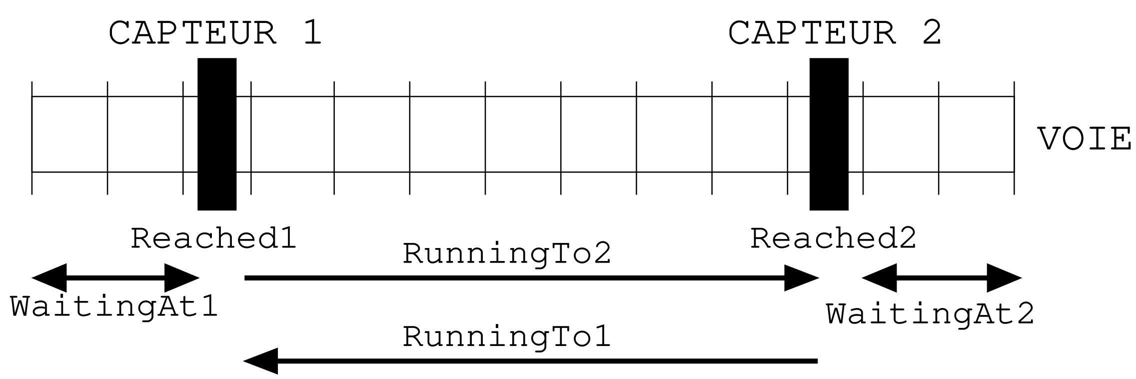

FIG. 4.3 – les états d'une locomotive

Lorsqu'un de ces processus aura changé le sens de marche de sa locomotive (quand celle-ci sera arrêtée dans une des gares et prête à repartir), il entrera dans une section critique protégé par un sémaphore (opération WAIT sur le sémaphore TRACK_SEM) avant de modifier la position des aiguillages et de lancer la locomotive vers la gare de destination. Quand la locomotive sera arrivée à destination, il signalera qu'il quitte la section critique (opération SIGNAL).

Les processus sont exécutés à intervalles réguliers. Au bout de l'écoulement d'un intervalle de temps, la routine de traitement d'interruption du temporisateur est activée. Elle lance l'exécution des deux processus de locomotives. Ceux-ci évaluent l'état des capteurs et prennent les décisions appropriées. Ensuite, ils terminent leur exécution jusqu'à leur prochain réveil. Il n'est pas possible de maintenir un processus de gestion de locomotive en fonctionnement permanent : en effet, cette approche impliquerait qu'un seul des deux processus soit exécutable, celui de priorité la plus élevée. Or, les deux processus doivent piloter leur locomotive respective pour éviter, par exemple, qu'elle n'arrive au bout d'une voie de garage. De plus, ces processus utiliseraient le système à cent pour cent, ce qui est inacceptable.

Il y a cependant une exeption à cette règle. En effet, si un des deux processus veut entrer dans sa section critique pour envoyer sa locomotive sur la voie unique, mais que la locomotive de l'autre processus s'y trouve déjà, le premier processus sera bloqué lors de l'exécution de l'opération WAIT. Il ne pourra donc pas se terminer avant le déclenchement de la prochaine interruption du temporisateur. Cela n'est pas gênant puisqu'il est arrêté. Si il est de priorité supérieure à celle du processus qui "possède" la voie unique, il ne gênera en rien le fonctionnement de ce dernier. Pour résumer, on peut dire qu'un processus de pilotage n'est arrêté que si la locomotive qu'il pilote est à l'arrêt, elle aussi.

Processus d'émission des données sur les rails

Un processus est utilisé pour émettre, en boucle, les données concernant les informations de vitesse et de direction des locomotives. Ces informations sont stockées dans un tampon d'émission. Un octet par locomotive y est réservé. Les quatre bits de poids faible contiennent la vitesse. Un changement de direction s'opère en spécifiant la valeur de vitesse spéciale "1" qui indique en fait au décodeur qu'il doit inverser le sens de circulation de la locomotive. Les quatre bits de poids fort sont inutilisés. Ce tableau est modifié par un processus de pilotage de locomotive lorsqu'il veut changer le comportement de la locomotive à laquelle il est associé. Il n'est pas nécessaire de prévoir un mécanisme d'exclusion mutuelle pour l'accès à ce tableau (l'accès à un octet se fait de façon atomique et, surtout, la manière dont les tâches sont lancées ici, c'est-à-dire par la routine de traitement d'interruption et le fait qu'elles se terminent *avant* l'interruption suivante, évitent qu'une tâche soit interrompue par une autre pendant qu'elle manipule ce tableau). On pourrait sans problème particulier, utiliser une structure plus complexe pour le tampon (autrement dit plusieurs octets par locomotive). On y stockerait, par exemple, seulement les informations des locomotives actives, il faudrait donc aussi stocker les adresses de ces locomotives. Ici, la position des informations d'une locomotive est fixe dans le tableau.

Le processus d'émission est aussi activé par la routine de traitement d'interruption du temporisateur. Si un mot a été transmis, il démarre l'envoi du mot suivant. Il attend alors environ 25 millisecondes en comptant le nombre de fois qu'il est démarré par la routine d'interruption. Quand il a été exécuté 13 fois, le temps de transmission est écoulé, le 145026 a transmis le mot[1]. Le processus d'émission démarre alors la transmission du mot suivant en positionnant les sorties du PCF8574 (port d'entrée/sortie I^2C) avec l'adresse et la donnée à envoyer. Il utilise, pour cela, une des sorties (connectée à l'entrée TE du 145026), du deuxième PCF8574 pour lancer la transmission. La ligne TE (active au niveau bas) est mise à 0 pendant un court instant, puis elle est ramenée au niveau haut. Le 145026 démarre alors la transmission du mot.

Processus de commande des aiguillages

Les décodeurs d'aiguillage se trouvent dans le même espace d'adresses I^2C que les décodeurs de locomotives. Ce processus utilise donc aussi le tampon d'émission de données et écrit à l'adresse qui correspond au décodeur d'aiguillage la valeur qui permettra de modifier la position de l'aiguillage requis. Le processus active une bobine d'aiguillage pendant 250 ms environ, puis il la désactive. Il génère ce délai en attendant, dans une boucle, que le nombre de "ticks" suffisant ait été compté (250 / TICK_DELAY, cette dernière macro valant le temps entre chaque "tick", en millisecondes). Ce processus attend donc de manière active, ce qui n'est pas grave car il est le processus le moins prioritaire. Il est lancé par les tâches de commande des locomotives si un aiguillage doit être actionné. Il partage avec ces dernières une pile qui contient les ordres de changement de positions d'aiguillages. Aucun sémaphore n'est cependant requis pour assurer l'exclusion mutuelle de cette pile car les tâches des locomotives lancent la tâche de commande des aiguillages et exécutent une opération WAIT sur un sémaphore juste après. Ce sémaphore (SWITCH_CHANGED_SEM) ne sert pas à l'exclusion mutuelle de l'accès à la pile mais à signaler que les aiguillages ont été mis dans la position demandée.

[1] Le processus ne charge donc pas le processeur pendant l'attente

Les contraintes "temps-réel"

Il y a plusieurs contraintes :

- les mots émis sur les rails doivent être envoyés toutes les 40 ms, dans le pire des cas. Sinon, les décodeurs pensent qu'il y a une erreur et se désactivent !

- pour différentes raisons (notamment un impératif de modularité), la lecture des détecteurs placés entre les rails se fait aussi par bus I^2C et, de plus, aucune interruption n'est générée lors d'un changement de l'état des détecteurs. Cette lecture est donc relativement lente et rend non-triviale la contrainte selon laquelle ces détecteurs doivent être consultés dans un intervalle de temps relativement court : dans le cas le plus défavorable, la locomotive se déplace à $(300/87)km/h$. Si sont frotteur (qui actionne le capteur) est long de 6 cm, le temps minimal d'activation du capteur est de $0,06/(300000/87/3600) = 63millisecondes$. Il faut détecter une telle durée d'activation du capteur.

- on ne veut "rater" aucun tick, car cela signifierait que le temps du système n'est plus correct ; ce temps pourrait être utilisé pour faire, par exemple, des mesures de vitesse. De plus, si aucune interruption du temporisateur n'échappe au système, l'analyse du respect des contraintes est facilitée.

Un accès à un périphérique I^2C prend environ 220 microsecondes (environ 22 cycles à 100KHz). Dans le pire des cas, cinq accès[2] sont faits entre deux interruptions. Le temps d'exécution du reste du code est négligeable par rapport aux accès au bus I^2C, estimons-le à 100 microsecondes (surestimation). Le temps maximal de l'exécution de code entre deux interruptions est alors de $5 * 220 + 100 = 1200microsecondes$. Si on utilise un temps de 2 microsecondes entre deux interruptions, les contraintes "temps réel" ci-dessus seront respectées : en effet, si une tâche de pilotage de locomotive ne s'est pas terminée dans ce temps, c'est qu'elle est en attente sur un sémaphore, elle ne perturbe donc pas le système. La tâche de commande des aiguillages, dont l'exécution peut durer jusqu'à 0,5 secondes[3], ne gêne personne car elle a la priorité la plus faible.

Le compilateur

Certaines limitations du PIC sont prises en charge par les compilateurs PICC (pour le 16F877) et PICLITE (pour le 16F84) de Hi-Teck Software.

---

[2]deux accès pour lire les capteurs qui ne sont lus qu'une fois entre deux interruptions, et trois accès pour écrire le prochain mot à envoyer sur la voie si le processus d'émission le décide.

[3]activation de deux aiguillages, donc 2 * 250 ms.

L'impossibilité d'utiliser la pile matérielle pour stocker les variables locales des fonctions conduit le compilateur à imposer que ces fonctions ne soient pas réentrantes. Le compilateur utilise en fait des zones fixes pour stocker les variables locales des fonctions. Cela explique pourquoi elles ne peuvent pas être utilisées de façon réentrante. Par défaut, il interdit d'appeler la fonction dans une routine d'interruption et depuis la fonction principale *main*, ou dans une fonction appelée par celle-ci. Si l'utilisateur assure lui-même que cela ne puisse pas arriver (en utilisant *ei()* et *di()* pour, respectivement, autoriser et interdire les interruptions), il le signale au compilateur à l'aide de la directive *#pragma interrupt level*, placée avant la définition de la fonction incriminée et avant les définitions des fonctions qui appellent celle-ci.... Le compilateur sait alors que la fonction ne pourra pas être appelée de plusieurs endroits en même temps.

4.3 Le code de l'application

Ce code n'est pas disponible dans cette version ; si vous voulez absolument une version de ce code, signalez-le moi...

Chapitre 5

Conclusion

Nous pouvons dire que les objectifs de départ sont atteints. En effet, tant les deux réalisations matérielles que le logiciel sont extrêmement modulaires. De plus, le logiciel fournit les services essentiels d'un système d'exploitation "temps réel". Voyons cela en détail ...

5.1 Le matériel

La première version (les cartes qui s'enfichent les unes dans les autres à volonté) est très pratique à utiliser dans un cadre expérimental. Le concepteur peut emboîter les différents modules dont il a besoin. Cette flexibilité, on l'a vu, a un prix : la surface des circuits imprimés est sous-utilisée. On ne pourra donc pas utiliser cette version dans le contexte d'une application qui doit être réalisée à grande échelle, si ce n'est dans un but pédagogique.

C'est là que la deuxième version intervient : sa modularité a été ramenée au niveau du concepteur de circuits imprimés : par des opérations simples de "copier/coller", ce dernier réalise le circuit imprimé (unique) de la réalisation en un temps très court. La perte de place liée à la redondance des connections aux quatre points cardinaux des modules, est beaucoup plus faible que dans le cas de la première version, puisqu'ici, les modules sont reliés par des ponts de câblage peu encombrants au lieu de connecteurs.

Il faut cependant être conscient du fait que ces deux réalisations matérielles ont été conçues sans tenir compte des normes de compatibilité électromagnétique européennes (CEM). Je n'ai eu ni le temps ni les ressources matérielles (logiciels de simulation) pour respecter ces normes. Par contre, les schémas des différents modules ainsi que le principe de la deuxième version peuvent, sans problème, être utilisés comme base d'un application commerciale. Il faut noter ici que la conception d'un circuit électronique qui respecte ces normes de compatibilité électromagnétique pourrait constituer, à elle seule, un sujet de travail de fin d'études.

5.2 Le logiciel

Le kernel est, lui aussi, très modulaire : les fonctions liées au scheduling et celles qui ont trait aux sémaphores sont placées dans deux modules séparés. Le code du kernel a une taille qui est inférieure à 500 mots quand on le compile pour un PIC, ce qui laisse, dans le cas du 16F84, la moitié de la mémoire de programme disponible pour le code spécifique à l'application. Si on n'a pas besoin de sémaphores dans une application particulière, cette taille peut être encore fortement réduite. Le nombre d'octets de RAM requis par le kernel est aussi réduit le plus possible : si on limite le nombre de tâches à huit, tout sémaphore consomme un octet de RAM. Le kernel lui-même a besoin de huit octets pour stocker les positions courantes des tâches et d'un neuvième octet pour connaître l'état d'une tâche (exécutable ou pas). Chaque tâche a besoin d'une structure d'échange avec le kernel. Une telle structure prend quatre octets de RAM. Si les tâches ne consultent pas directement la valeur d'un sémaphore (opération GET_SEM), elles peuvent se partager une structure d'échange sans risque de conflit.

Pour les applications plus importantes, le nombre maximum de tâches peut être fixé à 64. Dans ce cas, chaque sémaphore supplémentaire nécessite neuf octets de RAM. Le kernel a besoin alors de 64 octets pour les positions respectives de tâches et de neuf octets pour les états de ces tâches.

Des fonctionnalités plus complexes n'ont pas été implémentées pour ne pas alourdir le kernel de fonctions qui seraient inutiles dans de nombreux cas. En effet, les contraintes de place imposées à la taille du code et de la RAM utilisée, dans la cas du PIC, amènent le concepteur de l'application à concevoir une solution ad-hoc qui est optimisée pour son application. L'utilisation de modules ne serait plus assez flexible pour permettre au concepteur de l'application d'utiliser uniquement les fonctions dont il aurait réellement besoin au sein d'un module particulier. Le programmeur de l'application devra, par exemple, écrire lui-même les fonctions d'implémentation d'un système de file FIFO. Il sera, cependant, aidé dans sa tâche par les fonctionnalités du module de gestion des sémaphores.

Un point important doit être souligné ici : les contraintes du PIC (pile matérielle non adressable), nous ont forcé à réaliser un kernel non préemptif qui n'utilise pas, lui-même, de mécanisme d'interruption ou de manipulation de la pile du système. Ce fait a permis de réaliser un kernel totalement portable car il ne contient pas la moindre ligne de code en assembleur (manipulation directe de la pile). Grâce à cela, il peut être compilé dans tout environnement qui dispose d'un compilateur C. Si on dispose, dans cet environnement, d'un mécanisme qui permet à un programme d'obtenir la priorité maximale sur le système, on peut réaliser une application “temps-réel”. La portabilité totale du kernel m'a permis, entre autres, de le mettre au point dans un environnement de type UNIX.

Chapitre 6

Annexes

6.1 Code source

6.1.1 Le kernel : kernel.c

```
#include "kernel.h"
#include <stdlib.h> /* exit() */

const unsigned char bitPower[] = {1, 2, 4, 8, 16, 32, 64, 128};

#if MAX_TASK == 64
prtyStruc runningTasks = {0, {0, 0, 0, 0, 0, 0, 0, 0}};
#else
prtyStruc runningTasks = {0};
#endif

taskStruc taskBuf[MAX_TASK];

#ifdef HI_TECH_C
#pragma interrupt_level 0
#endif
/*-----------------------------------------------------------------
 * setTaskBit positionne le bit correspondant à une tâche dans une
 * structure de stockage de priorités de tâches.
 *
 * arguments : prtyTblPtr est un pointeur vers la structure de
 *             priorités à modifier.
 *             task est le numéro de la tâche dont on doit
 *             positionner le bit correspondant.
 *
 * valeur de retour : aucune.
 *
```

```
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN PRINCIPE PAS appeler directement
 * cette fonction.
 * type : nécessaire.
 *----------------------------------------------------------------
 */
void setTaskBit(prtyStruc *prtyTblPtr, unsigned char task) {
#if MAX_TASK == 64
  unsigned char posY;

  posY = task >> 3;
  prtyTblPtr->x[posY] |= bitPower[task & 7];
  prtyTblPtr->y |= bitPower[posY];
#else
  prtyTblPtr->x |= bitPower[task & 7];
#endif
}


/*----------------------------------------------------------------
 * clearTaskBit efface le bit correspondant à une tâche dans une
 * structure de stockage de priorités de tâches.
 *
 * arguments : prtyTblPtr est un pointeur vers la structure de
 *             priorités à modifier.
 *             task est le numéro de la tâche dont on doit
 *             effacer le bit correspondant.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN PRINCIPE PAS appeler directement
 * cette fonction.
 * type : nécessaire.
 *----------------------------------------------------------------
 */
void clearTaskBit(prtyStruc *prtyTblPtr, unsigned char task) {
#if MAX_TASK == 64
  unsigned char posY;

  posY = task >> 3;
```

```
  if (!(prtyTblPtr->x[posY] &= ~bitPower[task & 7]))
    prtyTblPtr->y &= ~bitPower[posY];
#else
  prtyTblPtr->x &= ~bitPower[task & 7];
#endif
}

/*--------------------------------------------------------------
 * getTaskBit renvoie 0 si le bit correspondant à une tâche
 * est positionné dans une structure de priotité de tâche.
 *
 * arguments : prtyTblPtr est un pointeur vers la structure de
 *             priorités.
 *             task est le numéro de la tâche.
 *
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN PRINCIPE PAS appeler directement
 * cette fonction.
 * type : nécessaire.
 *--------------------------------------------------------------
 */
unsigned char getTaskBit(prtyStruc *prtyTblPtr, unsigned char task) {
#if MAX_TASK == 64
  unsigned char posY;

  posY = task >> 3;

  return (prtyTblPtr->x[posY] & bitPower[task & 7]);
#else
  return (prtyTblPtr->x & bitPower[task & 7]);
#endif
}


/*--------------------------------------------------------------
 * higherTask donne le numéro de la tâche la plus prioritaire dont
 * le bit est positionné dans la structure de stockage de
 * priorités de tâches donnée.
 *
 * argument : taskPrty est la structure de priorités à consulter.
 *
 * valeur de retour : le numéro de la tâche de priorité la plus
 *                    élevée dont le bit est positionné dans la
```

```
 *                    table.
 *
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN PRINCIPE PAS appeler directement
 * cette fonction.
 * type : nécessaire.
 *----------------------------------------------------------------
 */
unsigned char higherTask(prtyStruc tasksPrty) {
#if MAX_TASK == 64
  char posY, posX;

  for(posY = 7; posY >= 0; posY--)
    if (tasksPrty.y & bitPower[(unsigned char) posY]) break;

  if (posY < 0) return MAX_TASK;    /* aucune tâche n'est active? */

  for(posX = 7;; posX--)            /* le gardien est inutile ici
                                       car il doit y avoir une
                                       tâche active si l'exécution
                                       se poursuit jusqu'ici */
    if (tasksPrty.x[(unsigned char) posY] &
        bitPower[(unsigned char) posX]) break;

  return (posX + (posY << 3));
#else
  char posX;

  for(posX = 7; posX >= 0; posX--)
    if (tasksPrty.x & bitPower[(unsigned char) posX]) break;

  if (posX < 0) return MAX_TASK;    /* aucune tâche n'est active? */
  return posX;
#endif
}


#ifdef HI_TECH_C
#pragma interrupt_level 0
#endif
/*----------------------------------------------------------------
 * startTask indique au kernel qu'une tâche devient exécutable.
 * Cette tâche verra ses instructions exécutées par le processeur
```

```
 * quand elle sera la tâche exécutable la plus prioritaire
 * instanciée sur le système. FIX_ME completer
 *
 * argument : task est le numéro de la tâche.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction peut être appelée dans une fonction de
 * traitement d'interruption. Elle est principalement utilisée de
 * manière interne par le kernel. Les tâches NE devraient PAS
 * appeler directement cette fonction.
 * type : nécessaire.
 *-------------------------------------------------------------
 */
void startTask(unsigned char task) {

 if (!getTaskBit(&runningTasks, task)) {
   taskBuf[task].pos = 0;
   setTaskBit(&runningTasks, task);
 }
}


/*-------------------------------------------------------------
 * killTask indique au kernel qu'une tâche n'est plus exécutable.
 * Cette fonction retire aussi la tâche de tous les ensembles de
 * tâches en attente sur les sémaphores et réinitialise sa
 * position courante d'exécution au début de son code.
 *
 * argument : task est le numéro de la tâche.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction peut être appelée dans une fonction de
 * traitement d'interruption. Elle est principalement utilisée de
 * manière interne par le kernel. Les tâches NE devraient PAS
 * appeler directement cette fonction.
 * type : nécessaire.
 *-------------------------------------------------------------
 */
void killTask(unsigned char task) {

#ifdef USE_SEMAPHORE
  unsigned char i;
```

```
#endif

  clearTaskBit(&runningTasks, task);
  taskBuf[task].pos = 0;

#ifdef USE_SEMAPHORE
  /* code de suppression de la tâche dans les ensembles de
     tâches en attente sur les sémaphores */
  for(i = 0; i < MAX_SEM; i++) remWaitingTask(i, task);
#endif
}


/*-----------------------------------------------------------------
 * stopTask indique au kernel qu'une tâche n'est plus exécutable.
 * La position courante d'exécution de la tâche ainsi que les
 * informations concernant l'éventuelle attente de cette tâche
 * sur des sémaphores ne sont pas modifiées par cette fonction.
 *
 * argument : task est le numéro de la tâche.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction peut être appelée dans une fonction de
 * traitement d'interruption. Elle est principalement utilisée de
 * manière interne par le kernel. Les tâches NE devraient PAS
 * appeler directement cette fonction.
 * type : facultative.
 *-----------------------------------------------------------------
 */
void stopTask(unsigned char task) {
  clearTaskBit(&runningTasks, task);
}


/*-----------------------------------------------------------------
 * restartTask rend à nouveau exécutable une fonction qui a
 * précédemment été stoppée par la fonction stopTask.
 *
 * argument : task est le numéro de la tâche.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction peut être appelée dans une fonction de
```

```
 * traitement d'interruption. Elle est principalement utilisée de
 * manière interne par le kernel. Les tâches NE devraient PAS
 * appeler directement cette fonction.
 * type : facultative.
 *----------------------------------------------------------------
 */
void restartTask(unsigned char task) {
  setTaskBit(&runningTasks, task);
}


/*----------------------------------------------------------------
 * schedule est la fonction la plus importante du kernel.
 * Lorsqu'elle reprend le contrôle du système, elle sélectionne
 * la tâche exécutable la plus prioritaire et relance son
 * exécution. De plus, elle réalise effectivement l'appel système
 * demandé par une tâche.
 *
 * argument : aucun.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction NE DOIT ETRE APPELEE QU'UNE SEULE FOIS,
 * dans la fonction main de l'application. Le mécanisme qui permet
 * de lui rendre le contrôle pour lui faire exécuter des appels
 * systèmes est expliqué en détail dans le rapport.
 * type : nécessaire.
 *----------------------------------------------------------------
 */
void schedule() {
  unsigned char task, sysCallCode;
  taskParmStruc *taskParmPtr;

  for(;;) {

    /* dans une vraie application, on boucle indéfiniment si il n'y
       a pas de tache active mais pour les tests sur une station de
       travail, on s'arrete...
    */

#ifdef WSDEBUG
    if ((task = higherTask(runningTasks)) == MAX_TASK) break;
#else
    if ((task = higherTask(runningTasks)) == MAX_TASK) continue;
```

```
#endif

    /* en principe, on ne peut pas interdire les interruptions ici
       mais si les tâches utilisent les huit niveaux de la pile,
       plus aucune interruption ne pourra se déclencher sans mener
       à un crash du système... (cas du PIC)
    */

    di(); /* à enlever */

    taskParmPtr = dispatch(task, taskBuf[task].pos);

    ei(); /* à enlever */


    taskBuf[task].pos = taskParmPtr->pos;
    sysCallCode = taskParmPtr->sysCallCode;

    switch(sysCallCode) {
    case RESCHEDULE:
      break;

    case TASK_START:
      di();
      startTask(taskParmPtr->task);
      ei();
      break;

    case TASK_EXIT:
      di();
      killTask(taskParmPtr->task);
      ei();
      break;

#ifdef USE_SEMAPHORE
    case SIGNAL_SEM:
      di();
      signalSem(taskParmPtr->sem);
      ei();
      break;

    case WAIT_SEM:
      waitSem(task, taskParmPtr->sem);
      break;
```

```
    case SET_SEM_VALUE:
      di();
      setSemValue(taskParmPtr->sem, taskParmPtr->val);
      ei();
      break;

    case GET_SEM_VALUE:
      di();
      taskParmPtr->val = getSemValue(taskParmPtr->sem);
      ei();
      break;
#endif                          /* #ifdef USE_SEMAPHORE */

#ifdef WSDEBUG
    default:
      printf("unknown syscall code %d\n", sysCallCode);
      exit(-1);
#endif
    }
  }
}
```

6.1.2 Le fichier *include* du kernel : kernel.h

```
#ifndef __KERNEL_H

#define __KERNEL_H

/* macros générales */

#ifndef MAX_SEM
#define MAX_SEM 0   /* par défaut, on n'utilise pas les sémaphores */
#endif

#ifndef MAX_TASK
#define MAX_TASK 8  /* par défaut : 8 tâches maximum. Ceci est
                       adapté au 16F84, qui a peu de ressources */
#endif

#if (MAX_TASK != 64) && (MAX_TASK != 8)
#define MAX_TASK 64
#endif

#if MAX_SEM > 0
#define USE_SEMAPHORE
#endif

/* fin macros générales */
#ifndef di
#define di() ;
#endif
#ifndef ei
#define ei() ;
#endif

#include "dispatch.h"
#include "common.h"

#ifdef HI_TECH_C
#include <pic.h>
#endif

#ifdef USE_SEMAPHORE
#include "sema.h"
#endif
```

```c
/* macros permettant de réaliser les appels système depuis les
   tâches */

#define TASK_START 1
#define TASK_EXIT 2
#define RESCHEDULE 3

#ifdef USE_SEMAPHORE
#define GET_SEM_VALUE 10
#define SET_SEM_VALUE 11
#define WAIT_SEM 12
#define SIGNAL_SEM 13
#endif

#define START_TASK(p, t, n) \
p.pos = n; \
p.task = t; \
p.sysCallCode = TASK_START; \
return &p;

#define EXIT_TASK(p, t, n) \
p.pos = n; \
p.task = t; \
p.sysCallCode = TASK_EXIT; \
return &p;

#define SCHEDULE(p, n) \
p.pos = n; \
p.sysCallCode = RESCHEDULE; \
return &p;

#ifdef USE_SEMAPHORE
#define WAIT(p, s, n) \
p.pos = n; \
p.sem = s; \
p.sysCallCode = WAIT_SEM; \
return &p;

#define SIGNAL(p, s, n) \
p.pos = n; \
p.sem = s; \
p.sysCallCode = SIGNAL_SEM; \
return &p;
```

```
#define GET_SEM(p, s, n) \
p.pos = n; \
p.sem = s; \
p.sysCallCode = GET_SEM_VALUE; \
return &p;

#define SET_SEM(p, s, v, n) \
p.pos = n; \
p.sem = s; \
p.val = v; \
p.sysCallCode = SET_SEM_VALUE; \
return &p;
#endif          /* #ifdef USE_SEMAPHORE */

/* fin des macros permettant de réaliser les appels système
   depuis les tâches */

/* définitions des structures utilisées par le kernel */

typedef struct
{
  unsigned char pos;
} taskStruc;

/* fin des définitions des structures utilisées par le kernel */

/* prototypes des fonctions du kernel qui peuvent être appelées
   depuis d'autres modules */

void startTask(unsigned char task);
void schedule();
void setTaskBit(prtyStruc *prtyTblPtr, unsigned char task);
unsigned char getTaskBit(prtyStruc *prtyTblPtr, unsigned char task);
void clearTaskBit(prtyStruc *prtyTblPtr, unsigned char task);
void stopTask(unsigned char task);
void restartTask(unsigned char task);
void killTask(unsigned char task);
unsigned char higherTask(prtyStruc tasksPrty);

/* fin des prototypes des fonctions du kernel */

#endif             /* #ifndef __KERNEL_H */
```

6.1.3 La gestion des sémaphores : sema.c

```c
#include "sema.h"

semStruc semBuf[MAX_SEM];

/*--------------------------------------------------------------
 * getSemValue retourne la valeur d'un sémaphore.
 *
 * argument : sem est le numéro identifiant le sémaphore.
 *
 * valeur de retour : la valeur du sémaphore.
 *
 * note : cette fonction peut être appelée dans une fonction de
 * traitement d'interruption. Elle est principalement utilisée de
 * manière interne par le kernel. Les tâches NE devraient PAS
 * appeler directement cette fonction.
 * type : facultative.
 *--------------------------------------------------------------
 */
unsigned char getSemValue(unsigned char sem) {
  return semBuf[sem].value;
}


/*--------------------------------------------------------------
 * setSemValue modifie la valeur d'un sémaphore.
 *
 * arguments : sem est le numéro du sémaphore
 *             val est la nouvelle valeur à attribuer au sémaphore
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction peut être appelée dans une fonction de
 * traitement d'interruption. Elle est principalement utilisée de
 * manière interne par le kernel. Les tâches NE devraient PAS
 * appeler directement cette fonction.
 * type : facultative.
 *--------------------------------------------------------------
 */
void setSemValue(unsigned char sem, unsigned char val) {
  semBuf[sem].value = val;
}
```

```
/*------------------------------------------------------------------
 * higherWaitingTask donne le numéro de la tâche la plus
 * prioritaire qui est en attente sur un sémaphore.
 *
 * argument : sem est le numéro du sémaphore.
 *
 * valeur de retour : le numéro de la tâche de priorité la plus
 *                    élevée qui attend sur le sémaphore.
 *
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN AUCUN CAS appeler directement
 * cette fonction.
 * type : facultative.
 *------------------------------------------------------------------
 */
unsigned char higherWaitingTask(unsigned char sem) {
  return higherTask(semBuf[sem].waitingTasks);
}


/*------------------------------------------------------------------
 * addWaitingTask ajoute une tâche à l'ensemble de celles qui
 * attendent sur un sémaphore.
 *
 * arguments : sem est le numéro du sémaphore.
 *             task est le numéro de la tâche à ajouter.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN AUCUN CAS appeler directement
 * cette fonction.
 * type : facultative.
 *------------------------------------------------------------------
 */
void addWaitingTask(unsigned char sem, unsigned char task) {
  setTaskBit(&(semBuf[sem].waitingTasks), task);
}

/*------------------------------------------------------------------
 * remWaitingTask enlève une tâche de l'ensemble de celles qui
```

```
 * attendent sur un sémaphore.
 *
 * arguments : sem est le numéro du sémaphore.
 *             task est le numéro de la tâche à retirer.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN AUCUN CAS appeler directement
 * cette fonction.
 * type : facultative.
 *----------------------------------------------------------------
 */
void remWaitingTask(unsigned char sem, unsigned char task) {
  clearTaskBit(&(semBuf[sem].waitingTasks), task);
}


/*----------------------------------------------------------------
 * waitSem réalise l'opération classique d'attente d'une tâche sur
 * un sémaphore. Si la valeur du sémaphore est nulle, la tâche
 * appelante est mise en attente sur ce sémaphore. Elle est
 * ajoutée à l'ensemble des tâches qui sont déjà en attente sur ce
 * sémaphore et stoppée. Si la valeur du sémaphore n'est pas
 * nulle, celle-ci est décrémentée et la tâche peut poursuivre
 * son exécution.
 *
 * arguments : task est le numéro de la tâche.
 *             sem est le numéro du sémaphore.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction est utilisée de manière interne par le
 * kernel. Les tâches et les fonctions de traitement
 * d'interruption NE PEUVENT EN AUCUN CAS appeler directement
 * cette fonction.
 * type : facultative.
 *----------------------------------------------------------------
 */
void waitSem(unsigned char task, unsigned char sem) {
  unsigned char semVal;

  if ((semVal = getSemValue(sem)))
```

```
    setSemValue(sem, semVal - 1);

  else {
    addWaitingTask(sem, task);
    stopTask(task);
#ifdef WSDEBUG
    printf("Task_%d_is_waiting_on_semaphore_%d\n", task, sem);
#endif
  }
}


/*----------------------------------------------------------------
 * signalSem réalise l'opération classique signal. Si la valeur
 * du sémaphore est non nulle, ce qui veut dire qu'aucune tâche
 * n'est en attente sur un sémaphore, la valeur decelui-ci est
 * simplement incrémentée. Si sa valeur est nulle (ce qui veut
 * dire qu'il y a peut-être une tâche en attente sur ce
 * sémaphore), la tâche la plus prioritaire en attente sur ce
 * sémaphore, si elle existe, est à nouveau rendue exécutable.
 * Si aucune tâche n'était en attente, la tâche ayant exécuté
 * l'appel poursuit son exécution.
 *
 * arguments : sem est le numéro du sémaphore.
 *
 * valeur de retour : aucune.
 *
 * note : cette fonction peut être appelée dans une fonction de
 * traitement d'interruption. Elle est principalement utilisée de
 * manière interne par le kernel. Les tâches NE devraient PAS
 * appeler directement cette fonction.
 * type : facultative.
 *----------------------------------------------------------------
 */
void signalSem(unsigned char sem) {
  unsigned char semVal, task;

  task = higherWaitingTask(sem);

  /* si task == MAX_TASK, cela veut dire qu'aucune tâche
     n'attend sur le sémaphore, donc on incrémente simplement
     sa valeur
  */
```

```
  if ((semVal = getSemValue(sem)) || task == MAX_TASK)
    setSemValue(sem, semVal + 1);

  else {
    restartTask(task);
    remWaitingTask(sem, task);
#ifdef WSDEBUG
    printf("Task %d, previously waiting on semaphore %d, " \
           "is restarted\n", task, sem);
#endif
  }
}
```

6.1.4 Le fichier *include* correspondant à “sema.c” : sema.h

```
#ifndef __SEMAPHORE_H

#define __SEMAPHORE_H

#include "kernel.h"

unsigned char getSemValue(unsigned char sem);
void setSemValue(unsigned char sem, unsigned char val);
unsigned char higherWaitingTask(unsigned char sem);
void addWaitingTask(unsigned char sem, unsigned char task);
void remWaitingTask(unsigned char sem, unsigned char task);
void waitSem(unsigned char task, unsigned char sem);
void signalSem(unsigned char sem);

typedef struct
{
  prtyStruc waitingTasks;
  unsigned char value;
} semStruc;

#endif
```

6.1.5 Le fichier *include* de la fonction utilisateur *dispatch* : dispatch.h

```
#ifndef __DISPATCH_H

#define __DISPATCH_H

#include "user.h"
#include "common.h"

taskParmStruc *dispatch(unsigned char task, unsigned char pos);

#endif
```

6.1.6 Le fichier *include* commun (structure d'échange entre le kernel et les tâches) : common.h

```
#ifndef __TASK_STRUC_H

#define __TASK_STRUC_H

#ifdef WSDEBUG
#include <stdio.h>
#endif


/* cette structure permet à une tâche de communiquer des
 * informations au kernel et d'en recevoir de celui-ci...
*/
typedef struct
{
  unsigned char task;
  unsigned char sem;
  unsigned char val;
  unsigned char sysCallCode;
  unsigned char pos;
} taskParmStruc;

#if MAX_TASK == 64
typedef struct
{
  unsigned char y;
  unsigned char x[8];
} prtyStruc;
#else                   /* MAX_TASK == 8 */
typedef struct
{
  unsigned char x;
} prtyStruc;
#endif


#endif
```

6.2 Sérigraphies et circuits imprimés

Les dessins des circuits imprimés et les sérigraphies correspondantes sont temporairement indisponibles ; si vous désirez les obtenir, il vous suffit de me le signaler.

6.3 Liste des composants de la deuxième version

nb	Device	Value	Package type	Refdes	Prix (estim.)
1	7406		DIP14	U7	20
1	74HCT74		DIP14	U5	20
1	16F877		DIP40	U1	350
2	74HCT374		DIP20	U3	20
				U4	20
1	74HCT646		ISDIP24	U2	20
4	CER.CAP	10n	CRM5A	C6	5
				C7	5
				C8	5
				C9	5
2	CER.CAP	10p	CRM5A	C4	5
				C5	5
2	CER.CAP	100n	CRM5A	C1	5
				C2	5
1	DIN-64AC	DIN-64AC	DIN41612AC	J1	100
1	DIODE	1N4001	DIOD1	D1	10
1	FC-16VB	FC-16VB	header2x8	J4	50
1	HDR-4	HDR-4	header1x4	J5	8
1	HDR-6	HDR-6	header1x6	J6	12
1	POL.CAP	10u	RAD1	C3	20
1	P-REGULATOR	LM317	TO220	U6	20
1	RES.25W	1K	res12	R15	5
8	RES.25W	220	res12	R1	5
				R2	5
				R3	5
				R4	5
				R5	5
				R6	5
				R7	5
				R8	5
2	RES.25W	2.2K	res12	R16	5
				R17	5
1	RES.25W	470	res12	R11	5
3	RES.25W	2.2K	res12	R12	5
				R13	5
				R14	5
1	RES.25W	2.4K	res12	R9	5
1	RES.25W	4K7	res12	R10	5
1	XTAL	20MHz	XTAL	X1	100

nb	Device	Value	Package type	Refdes	Prix (estim.)
4	DELTA-9VF	DELTA-9VF	D9VF	J48	30
				J49	30
				J70	30
				J71	30
1	78XX	7805	TO220	U8	20
1	CER.CAP	100nF	CRM5A	C10	5
1	CER.CAP	100nF	CRM5A	C14	5
1	CER.CAP	330nF	CRM5A	C11	5
1	DIODE	1N4007	DIOD1	D2	10
1	POL.CAP	1000uF	RAD6	C12	5
1	POL.CAP	1uF	rad1	C13	5
1	RES.25W	10K	RES12	R18	5
1	SW-SPST	SWITCH	switch	SW1	20
1	TL7705A		DIP8	U9	30
1	CER.CAP	1n	CRM5A	C15	5
3	CER.CAP	100n	CRM5A	C16	5
				C17	5
				C18	5
1	HDR-8	HDR-8	header1x8	J47	16
1	MC145026		DIP16	U12	50
2	PCF8574		DIP16	U10	50
				U11	50
1	RES.25W	22K	RES12	R21	5
1	RES.25W	47K	RES12	R22	5
2	RES.25W	330	RES12	R20	5
				R19	5
1	support	tulipe 40b			40
1	support	tulipe 24b	(étroit !)		24
2	support	tulipe 20b			40
2	support	tulipe 14b			28
1	support	tulipe 8b			8
					1461

J47, J5 et J6 sont en fait des barrettes simples sécables de picots au pas de 2,54mm. J4, lui ,est constitué d'une barrette sécable double...
J48, 49, 70 et 71 sont des connecteurs SUB-D à 9 broches femelles pour montage vertical sur circuit imprimé.

Bibliographie

[1] Simon, David E.,*An Embedded Software Primer,* Addison-Wesley, 1999.

[2] Habermann, A. N., *Introduction to Operating System Design,* SCIENCE RESEARCH ASSOCIATES, INC, Carnegie-Mellon University, 1976.

[3] Briquet, Holzemer, Smeets, *Télésurveillance d'un micro-réseau ferroviaire*, Université de Liège, Avril 2001.

[4] Elektor numéro 239. *PICXEX, un système d'exploitation pour les processeurs PIC16C7x*, pages 58 à 61, Mai 1998.

[5] Farmer, Jerry, *A Real-Time Operating System for PICmicroTM Microcontrollers,* Application Note 585, Microchip Technology Inc., 1997.

[6] Lambin, Jean-Jacques, *LE MARKETING STRATEGIQUE*, EDISCIENCE international, 1999.

[7] Elektor numéro 267, *Digital Railrunner*, pages 40 à 43, Septembre 2000.

[8] Motorola, *MC145026 encoder, MC145027 decoder*, *date inconnue.*

[9] Microchip, *PIC16F8X, 18-pin Flash/EEPROM 8-Bit Microcontrollers*, 1998.

[10] Microchip, *PIC16F87X Data Sheet, 28/40-Pin 8-Bit CMOS FLASH Microcontrollers*, 2001.

[11] Texas Instruments, *TL7702A, TL7705A, TL7709A, TL7712A, TL7715A, SUPPLY-VOLTAGE SUPERVISORS*, 1999.

[12] Labrosse, Jean J., *MicroC/OS-II*, R&D books, Lawrence, KS 66046, 1999.