Programmation

Temps-réel avec «FreeRTOS»

Claude Guéganno

15 septembre 2024

Table des matières

Chapitre 1

La carte `Arduino`

Pour aborder les noyaux temps-réel, il est indispensable de faire les essais en situation réelle, c'est à dire en programmant une unité centrale immergée dans un contexte matériel fait d'entrées/sorties, elle même sujettes à des impératifs de temps.

Nous utiliserons comme support le hardware *open-source* de la carte Arduino.

`Arduino` est une mono carte basée sur un micro contrôleur et dédiée à la conception de systèmes informatisés simples ou d'objets connectés. L'électronique est *open source*, et architecturée autour d'un micro-contrôleur `Atmel AVR`, ou bien un processeur `Atmel` 32 bits ARM.

Les modèles courants sont dotés d'une interface USB, de 6 entrées analogiques et de 14 entrées/sorties digitales. Il existe de nombreuses cartes d'extensions.

La carte `Arduino` a été présentée en 2005, comme un moyen très bon marché de créer des systèmes permettant d'interagir avec des capteurs et des actionneurs. Les applications sont très diverses : thermostat, robots simples, détection de mouvements ... L'environnement de développement est fourni, et permet d'écrire des programmes en C ou C++.

Les cartes `Arduino` sont disponibles prêtes à l'utilisation auprès des fournisseurs. Il est également possible de les monter soi même. On estime à environ 700 000 le nombre de cartes distribuées en 2013.

1.1 Caractéristiques du matériel pour notre mise en œuvre

La carte Arduino Mega 2560 est une carte à micro-contrôleur basée sur un ATmega256.

Cette carte dispose :

- de 54 broches numériques d'entrées/sorties (dont 14 peuvent être utilisées en sorties PWM),
- de 16 entrées analogiques (qui peuvent également être utilisées en broches entrées/sorties numériques),
- de 4 UART (port série matériel),
- d'un quartz 16Mhz,
- d'une connexion USB,
- d'un connecteur d'alimentation jack,
- d'un connecteur ICSP [1],
- d'un bouton de réinitialisation (reset) ;
- de 256KB de mémoire programme flash (8KB sont pris par le *bootloader*) ;
- de 8KB de mémoire SRAM [2] ;
- de 4KB d'EEPROM [3].

1. La programmation in-situ (In-System Programming ou ISP) est une fonctionnalité qui permet aux composants électroniques (micro-contrôleurs en particulier) d'être (re)programmés alors qu'ils sont déjà en place dans le système électronique qu'ils doivent piloter.

   Ceci évite d'avoir besoin de programmer le composant en dehors du montage complet à l'aide d'un programmateur dédié.

2. La mémoire vive statique (ou SRAM pour l'anglais Static Random Access Memory) est un type de mémoire vive utilisant des bascules pour mémoriser les données. Mais contrairement à la mémoire dynamique, il n'y a pas besoin de rafraîchir périodiquement son contenu. Comme la mémoire dynamique, elle est volatile : elle ne peut se passer d'alimentation sous peine de voir les informations effacées irrémédiablement.

3. La mémoire EEPROM (Electrically-Erasable Programmable Read-Only Memory ou mémoire morte effaçable électriquement et programmable) (aussi appelée E2PROM) est un type de mémoire morte. Une mémoire morte est une mémoire utilisée pour enregistrer des informations qui ne doivent pas être perdues lorsque l'appareil qui les contient n'est plus alimenté

Précautions :

- L'intensité maxi disponible par broche E/S est de 40mA ;
- L'intensité maxi pour l'alimentation 3.3V est de 50mA ;
- L'intensité maxi pour la sortie 5V dépend de l'alimentation utilisée : 500mA maxi pour une alimentation par USB.

Elle contient tout ce qui est nécessaire pour le fonctionnement du microcontrôleur ; Pour pouvoir l'utiliser et se lancer, il suffit simplement de la connecter à un ordinateur à l'aide d'un câble USB (ou de l'alimenter avec un adaptateur secteur ou une pile, mais ceci n'est pas indispensable, l'alimentation étant fournie par le port USB).

La carte Arduino Mega 2560 est compatible avec les circuits imprimés prévus pour les cartes Arduino Uno.

1.2 Le micro-contrôleur ATMega256

L'ATmega256 est un micro contrôleur 8 bits à basse consommation, construit autour d'un processeur RISC (AVR *enhanced*) de Atmel. L'une des caractéristiques importantes est l'émetteur-récepteur à 2.4GHz.

La cadence d'exécution d'instructions est à 1 MIPS. L'émetteur-réepteur radio pemet des échanges de données de 250KB/s à 2MB/s.

---

en électricité.

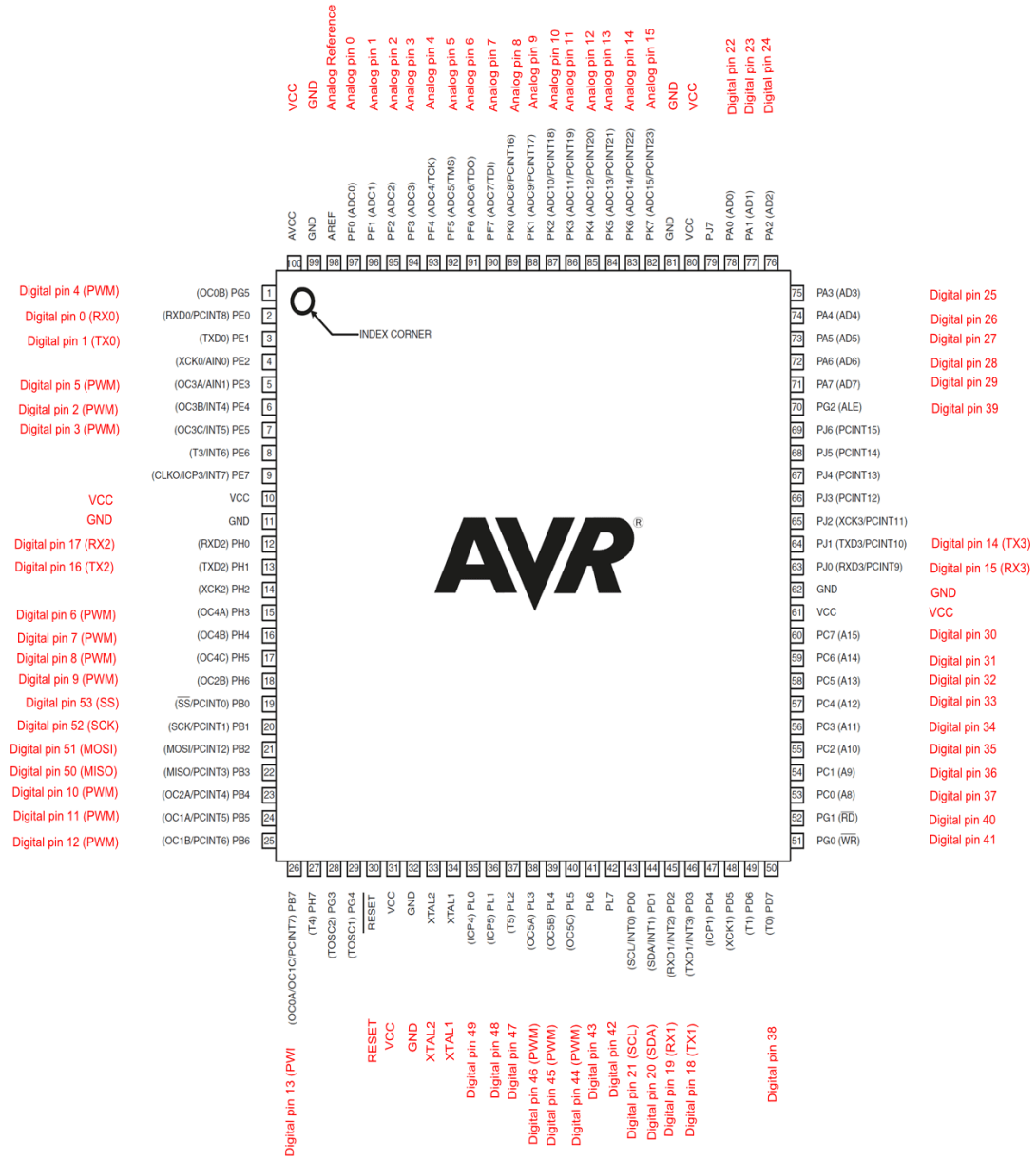

FIGURE 1.1 – *Brochage de l'ATMEGA2560*

Le cœur AVR permet un jeu d'instruction complet, et 32 registres de données à usage général. Ces 32 registres sont directement reliés à l'unité arithmétique et logique. 2 registres différents peuvent être accédés dans le même cycle d'horloge. Ceci permet au processeur d'être environ 10 fois plus rapide qu'une architecture CISC conventionnelle.

L'émetteur-récepteur radio est une solution Zigbee, et donc, il faut très peu de composants externes pour mettre en œuvre une communication basée sur Zigbee.

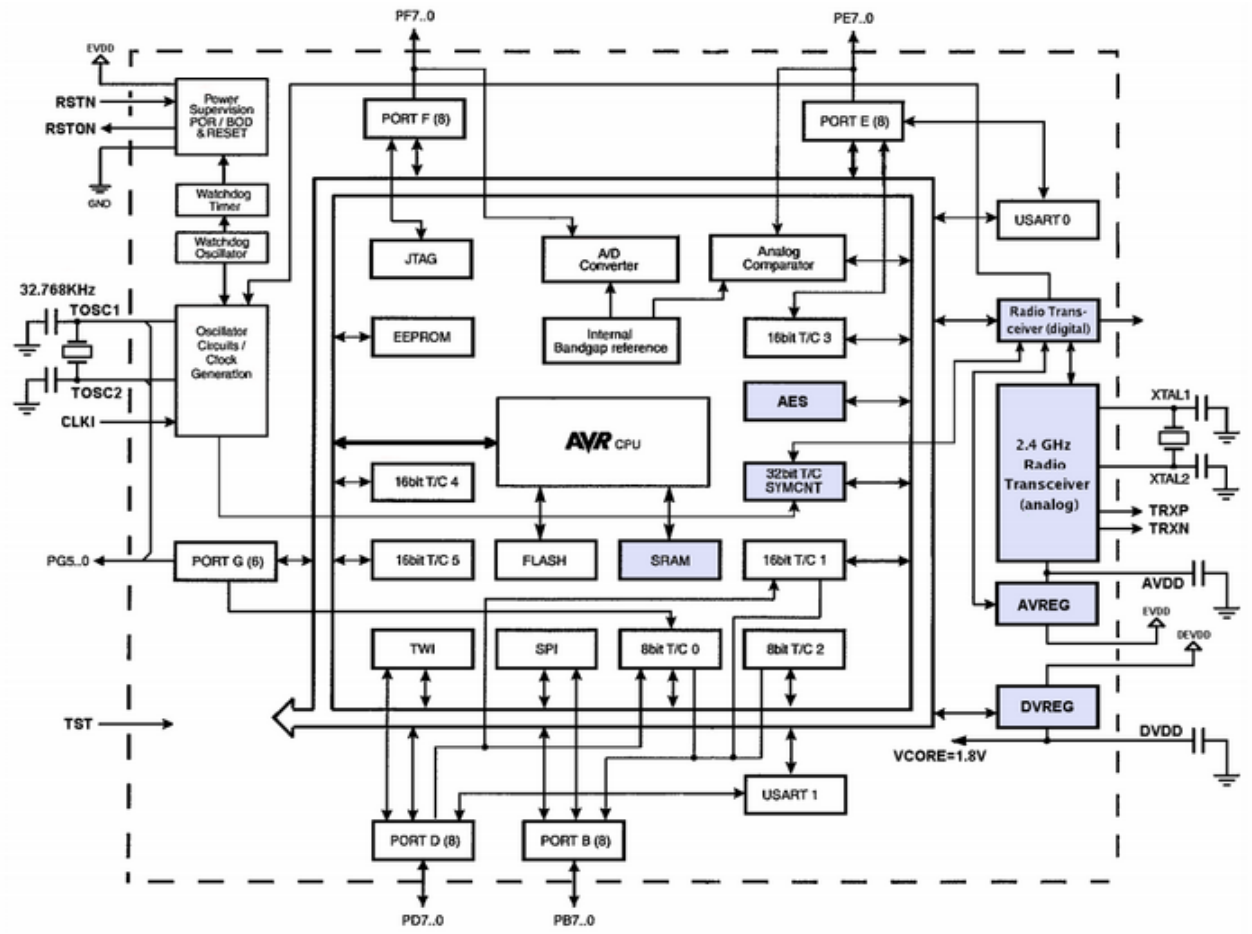

FIGURE 1.2 – *Diagramme des blocs de l'ATMEGA*

L'ATmega256 dispose de 256KB de flash ISP (*In system programming*), 8KB d'EEPROM, 32 KB de SRAM, jusqu'à 35 lignes I/O à usage général, 32 registres de calculs, un compteur temps-réel, 6 compteurs/timer programmables avec PWM, un compteur timer 32 bit, 2 USART, une interface série «2 wire», un convertisseur ADC 10 bits de 8 voies, un timer *watchdog*, un port série SPI [4] ; un port JTAG.

4. Une liaison SPI (pour Serial Peripheral Interface) est un bus de données série synchrone baptisé ainsi par Motorola, qui opère en mode Full-duplex. Les circuits communiquent selon un schéma maître-esclaves, où le maître s'occupe totalement de la communication. Plusieurs esclaves peuvent coexister sur un même bus, dans ce cas, la sélection du destinataire se fait par une ligne dédiée entre le maître et l'esclave appelée chip select.

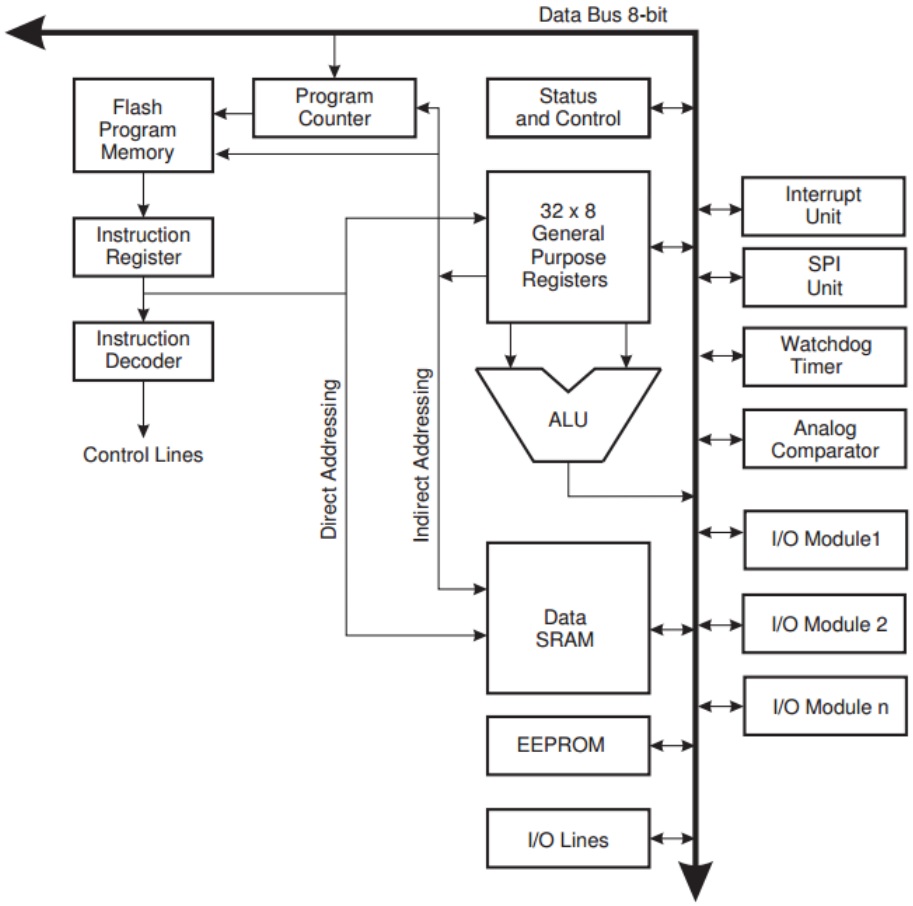

FIGURE 1.3 – *Diagramme des blocs de l'architecture AVR*

Le processeur AVR utilise une architecture Harvard, qui sépare les mémoires et les bus pour les programmes et les données. Les instructions sont exécutées dans un pipeline à un niveau. Pendant qu'une instruction est exécutée, la suivante est lue dans la mémoire programme. Ce concept permet l'exécution d'une instruction par cycle.

Les registres

L'AVR est doté de 32 registres de 8 bits. Trois d'entre eux peuvent être utilisés pour réaliser un adressage indirect vers la zone de données. L'un de ces registres d'adresse peut aussi être utilisé pour adresser la mémoire flash contenant le programme. Ces 3 registres s'appellent `X`, `Y` et `Z`.

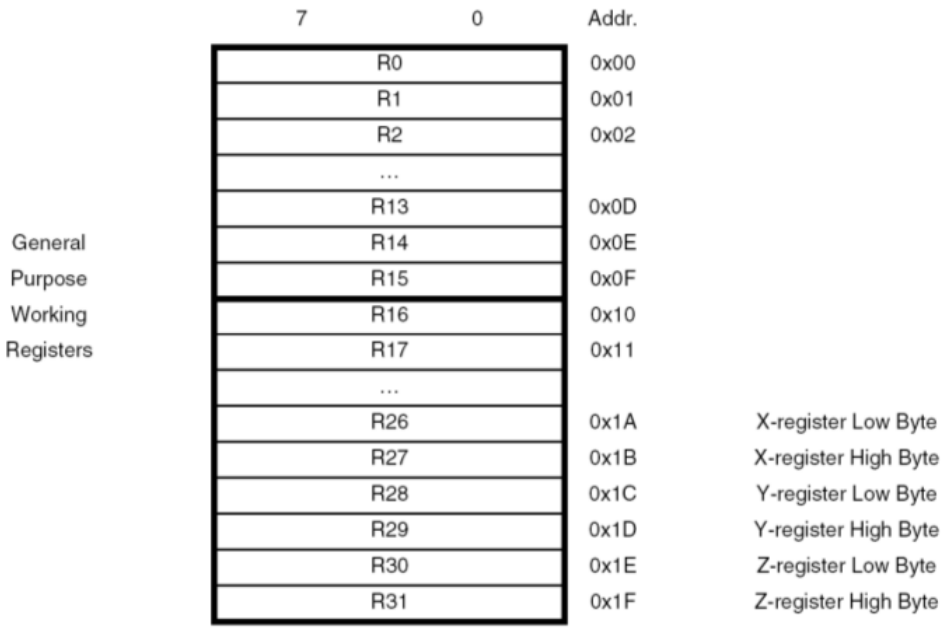

FIGURE 1.4 – *Les 32 registres de l'AVR.*

Mapping mémoire

Programme. Pour des raisons de sécurité, la mémoire flash est divisée en deux sections : la section de *boot* et la section pour le programme d'application. Toutes les instructions sont codées sur 16 ou 32 bits : la mémoire flash a donc une largeur de 16 bits.

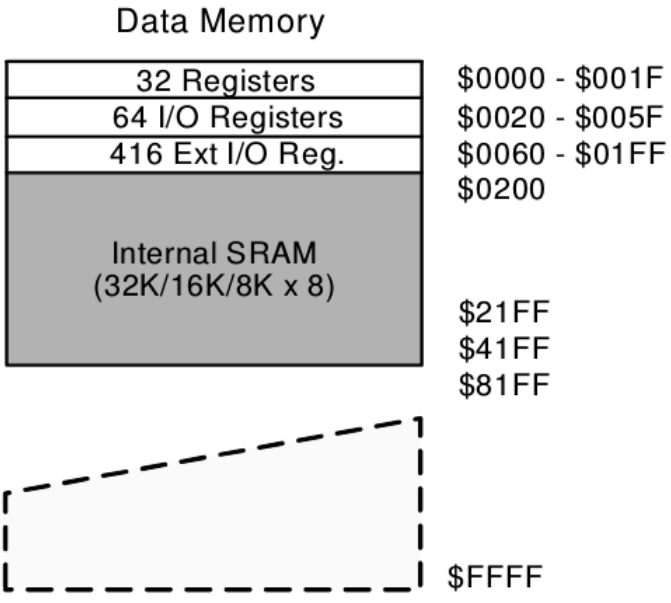

FIGURE 1.5 – *La mémoire programme de l'AVR.*

Données. Les adresses de `0000` à `001F` servent aux registres d'utilisation générale, alors que les adresses `0020` à `0005F` contiennent les registres d'I/O. De la position `0060` jusqu'à la `01FF` nous trouvons les registres I/O externes, puis de `0200` à `41FF` la SRAM interne du micro-contrôleur. Enfin, jusqu'à `FFFF` nous avons l'espace utilisable pour ajouter au micro-contrôleur de la mémoire SRAM externe.

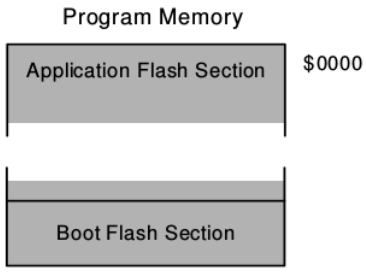

FIGURE 1.6 – *La mémoire des données l'AVR.*

1.3 Programmation standard

La manière la plus courante de programmer la carte utilise le C++. Le logiciel est divisé en deux fonctions : le `setup`, exécuté une seule fois au démarrage, dans lequel on place toute l'initialisation du système ; et la fonction `loop` qui est répétée indéfiniment ensuite.

```
1  int buttonPin = 3;
2
3  // la fonction setup initialise la communication série
4  // et une broche utilisée avec un bouton poussoir
5  void setup() {
6      Serial.begin(115200);
7      pinMode(buttonPin, INPUT);
8  }
9
10 // la fonction loop teste l'état du bouton à chaque passage
11 // et envoie au PC une lettre H si il est appuye, L sinon.
12 void loop() {
13         if (digitalRead(buttonPin) == 1) Serial.write('H');
14         else Serial.write('L');
15         delay(1000);
16 }
```

1.4 Mise en œuvre

Les cartes sont équipées d'une extension, qui nous permettra de tester efficacement nos programmes, sans perdre du temps à câbler.

Pour plus d'information sur la carte d'extension, voir l'annexe B (page 38).

Dans un premier temps, il convient de nous familiariser avec ce matériel, pour être efficace ensuite, au moment où nous aborderons le temps-réel à proprement parler.

Le code qui suit réalise le test des 4 sorties connectés aux LED.

```
1  // Définition des sorties
2  const byte S3 = 13;
3  const byte S2 = 12;
4  const byte S1 = 11;
5  const byte S0 = 10;
6  //définition des entrées
7  const byte E0 = 2;
8  const byte E1 = 3;
9
10 int T = 200; // 2/10 s
11
12 // initialisation basique
13 void setup() {
```

```
14    pinMode(S0,OUTPUT);
15    pinMode(S1,OUTPUT);
16    pinMode(S2,OUTPUT);
17    pinMode(S3,OUTPUT);
18    pinMode(E1,INPUT);
19    pinMode(E0,INPUT);
20  }
21
22  void loop() {
23         digitalWrite(S3, 0); digitalWrite(S0, 1); delay(T);
24         digitalWrite(S0, 0); digitalWrite(S1, 1); delay(T);
25         digitalWrite(S1, 0); digitalWrite(S2, 1); delay(T);
26         digitalWrite(S2, 0); digitalWrite(S3, 1); delay(T);
27  }
```

Ex. ***1***

Modifiez ce code pour que l'animation s'arrête lorsque l'interrupteur E0 est en position fermée.

Chapitre 2

Interruptions

2.1 Principe des interruptions

Les interruptions peuvent être causées par des sources externes ou par des sources internes :

Sources externes : GPIO, port série ...

Sources internes : timer, convertisseur AN ou NA, *Reset*.

Dans le programme, on associe un événement électronique matériel à une routine qui s'exécutera dès que l'événement se produira.

La séquence est :

1. Le processeur exécute un programme normalement.
2. Un événement survient ;
3. Le CPU achève l'instruction assembleur en cours de traitement.
4. L'état courant des registres est sauvé sur la pile système.
5. Le processeur traite le programme d'interruption associé à l'événement
6. Une fois la routine d'interruption terminée, le processeur reprend là où il en était, grâce à la restauration de la pile vers les registres...

Il est important d'avoir quelques notions sur les interruptions matérielles avant d'aborder le temps-réel, car le fonctionnement du noyau erepose sur l'utilisation d'interruptions périodiques.

2.2 Les interruptions externes

```
1  const byte E0 = 2;
2  const byte S1 = 11;
3  byte etat = 0;
4
5  void setup() {
6     Serial.begin(115200);
7     pinMode(S1, OUTPUT);
8     pinMode(E0, INPUT);
9     attachInterrupt(digitalPinToInterrupt(E0), onIT, CHANGE);
10    Serial.println(F("Système intialisé"));
11 }
12
13 void onIT() {
14   etat = 1 - etat;
15   Serial.println("<<IT>>");
16 }
17
18 void loop() {
19    digitalWrite(S1, etat);
20    Serial.print(digitalRead(interruptPin));
21    delay(300);
```

```
22  }
```

2.3 Les interruptions périodiques

Les «interruptions périodiques» permettent de déclencher une tâche régulièrement. Cette tâche vient interrompre le programme principal. L'exemple suivant nécessite l'installation de la librairie `TimerOne`.

```
1  #include <TimerOne.h>
2
3  // Définition d'une sortie sur le GPIO
4  const byte S0 = 10;
5
6  void setup() {
7    pinMode(S0,OUTPUT);
8
9    // Timer1 initialisé avec une période de 0.5s
10   Timer1.initialize(500000);
11   // Fonction lancée périodiquement
12   Timer1.attachInterrupt(commuteLED);
13   Serial.begin(115200);
14 }
15
16 // Pour sauver l'état de la LED
17 int X = 0;
18
19 void commuteLED() {
20   X = 1-X;
21   digitalWrite(S0, X);
22 }
23
24
25 void loop() {
26     Serial.print(".");
27     delay(1000);
28 }
```

Ex. 2

Testez cet exemple et vérifiez la sortie de la boucle principale dans un terminal série : `minicom -D /dev/ttyACM0`, ou directement dans le terminal série de l'IDE Arduino

Ex. 3

Toujours en utilisant cette interruption périodique, modifiez le programme pour que les quatre LEDs clignotent avec des périodes respectives de 400ms, 2000ms, 600ms et 1600ms. Apportez une attention particulière à l'organisation de vos données.

2.3.1 Création d'une région critique

Il s'agit d'un bloc de code utilisant une ressource particulière, et ne pouvant donc pas être interrompu par un autre code nécessitant la même ressource.

Le code de la boucle `loop()` devient :

```
1  void loop() {
2          char c;
3          for (c='A'; c<='D'; c++) { Serial.print(c); delay(10);}
4          Serial.print("\n\r");
5          delay(100);
6  }
```

Ex. 4

Testez le programme en apportant cette modification.
Observez la sortie du programme avec `minicom`.

Dans le programme d'interruption, nous allons aussi utiliser la ressource «port série», en ajoutant la ligne :

```
Serial.print("@");
```

Ainsi, à chaque commutation de LED, le caractère `'@'` est envoyé vers le terminal série par le programme d'interruption.

Ex. 5

Que remarquez vous cette fois dans le terminal ? Les messages du programme principal sont-ils intègres ?

⇒ .....

⇒ .....

Code ne pouvant pas être interrompu

Pour régler le problème, il suffit d'interdire les interruptions lorsque celles ci peuvent avoir un effet néfaste sur l'exécution du programme.

Pour cela, nous pouvons utiliser l'instruction `noInterrupts()`. Il faut bien sûr ré-autoriser les interruptions à l'issue du code non interruptible, avec la fonction `interrupts()`.

```
1  noInterrupts();
2    // code non interruptible
3  interrupts();
```

Ex. 6

Après avoir repéré les instructions qui doivent être protégées entre ces deux instructions, modifiez le programme pour que les messages de la boucle principale soient intègres.

Pilotage d'une séquence sur un servomoteur

On crée un nouveau projet Arduino. La librairie permettant de piloter des servomoteurs existe par défaut. Quelques informations basiques sont dans l'annexe C. Testez le code de cette annexe, pour bien vous approprier le pilotage du servomoteur.

Ex. 7

Proposez une autre réalisation, faisant la même chose mais qui utilise cette fois une interruption périodique. On utilisera le *timer* 3 de la librairie *TimerThree*. Il n'y aura donc plus rien dans la fonction `loop()`
Quelle sera la période de fonctionnement du *timer* ?

⇒ .....

⇒ .....

Pour la suite le *timer* 3 restera dédié au servomoteur.

Séquence de mouvement On souhaite que le moteur exécute cette fois une séquence périodique. Pour cela, les positions qu'il doit prendre figurent dans un tableau d'entiers.
`int Tpositions[] = {0, 10, 30, 70, 150, 180, 100, 50, -1};`
Dans ce cas, il y a donc 8 positions successives qui seront exécutées en boucle. La valeur (-1) n'est pas une position, elle indique qu'il faut reprendre au début. La durée entre chaque mouvement sera de 300ms.

Ex. ***8***

Modifiez votre code pour que cette séquence soit exécutée.

⇒ .....

⇒ .....

Structuration des données Il est fréquent que dans un automatisme, les durées consacrées à chaque position du servomoteur soient spécifiques à chaque position. Pour cela on associe une durée à chaque position :

Dans un premier temps, la structuration de données devient :

```
int Tpositions[] = {0, 10, 30, 70, 150, 180, 100, 50, -1};
int Tdurees[] = {450, 870, 495, 705, 840, 555, 315, 720, -1};
```

Ex. ***9***

Réalisez la commande du servomoteur, avec exactement cette séquence et toujours avec le support du *timer* 3

.

Quelle amélioration pourrions nous apporter à l'organisation de ces données ?

⇒ .....

⇒ .....

⇒ .....

⇒ .....

Pilotage d'un moteur pas à pas

L'annexe D explique l'essentiel pour comprendre le fonctionnement des moteur pas à pas. Il est recommandé de tester dans un premier temps le programme fourni dans cette annexe.

À la différence du servomoteur, à qui on donne une consigne, et il suffit ensuite d'attendre que la position soit atteinte, le moteur pas à pas doit être piloté pour chacun de ses mouvements élémentaires.

Pour piloter le moteur pas à pas, nous utiliserons le *timer* 4 de la librairie *TimerFour*.

Ex. ***10***

En s'inspirant des applications précédentes, créez un nouveau projet Arduino qui fait la même chose que le programme de l'annexe, mais en utilisant le *timer* 4. La fonction `loop()` sera donc laissée vide.

Synthèse

Il s'agit, pour conclure, de faire cohabiter l'ensemble des programmes précédents dans un seul projet. Les objectifs sont :

1. Faire clignoter les LEDs, avec le *timer* 1 ;
2. Exécuter la séquence du servomoteur avec le *timer* 3 ;
3. Faire tourner le moteur pas à pas avec le *timer* 4 ;
4. Le bouton *E0* inverse le sens de rotation du moteur pas à pas

Ex. ***11***

Réalisez ce projet, en vous basant directement sur tout ce qui a été fait précédemment.

Chapitre 3

Le multi-tâches avec «freeRTOS»

Provide a free product that surpasses the quality and service demanded by users of commercial alternatives

3.1 Présentation

FreeRTOS (Free Real Time Operating System) est un système d'exploitation temps réel embarqué, présentant la particularité d'être de très petite taille. Ses concepteurs le qualifient ainsi de *mini* noyau temps réel.

3.1.1 Caractéristiques et fonctionnalités

La principale caractéristique de FreeRTOS est sa petite taille : Les sources du noyau se composent de 3 ou 4 fichiers, et une image compilée du noyau pèse entre 6 et 12 Ko. l'empreinte mémoire du système est également très réduite.

La majeure partie du noyau est écrite en langage C, ce qui assure au système un haut degré de portabilité, et aux développeurs la manipulation d'un langage universel. Seule les morceaux de codes spécifiques aux architectures supportées contiennent des instructions en assembleur.

La version actuelle de FreeRTOS (v2022.10.01) supporte officiellement 35 types d'architectures différentes, parmi lesquelles des processeurs de marques *ARM*, *Xilinx*, *Microchip*.

FreeRTOS présente les fonctionnalités de base d'un système d'exploitation :

- Création, manipulation et destruction de tâches et de co-routines (Différentes des tâches, les co-routines sont utilisées sur de petits processeurs avec des limitations notamment en terme de mémoire).
- Ordonnancement temps réel préemptif, coopératif ou hybride ;
- Structures de communication inter-tâches : files de messages, sémaphores binaires ou avec compteurs, *mutex*.

De plus, certains outils de collecte de statistiques (notamment sur les temps d'exécution) et de traces sur le comportement du système sont intégrés au noyau, permettant aux utilisateurs d'analyser l'attitude des applications embarquées réalisées.

La communauté de FreeRTOS met également à disposition sur le site du projet (`http://www.freertos.org/`) des simulateurs pour Windows et Linux permettant aux utilisateurs ne possédant pas de cartes de tester le système.

3.1.2 Un système *open source*

FreeRTOS est distribué sous license MIT *open source* . Les utilisateur du système peuvent l'obtenir gratuitement, et l'utiliser librement (sans royalties), cela même dans une application commerciale.

Les développeurs qui souhaitent utiliser FreeRTOS avec des applications commerciales ne sont pas obligé d'ouvrir le code de ces applications. C'est le but de la clause d'exception de la licence MIT, qui

permet d'effectuer l'édition des liens avec du code source propriétaire. Cela est possible à condition que le code propriétaire fournisse des fonctionnalités autres que celles apportées par le noyau de FreeRTOS. De plus, tout changement apporté au noyau lui même doit être open-source. Enfin, toute utilisation de FreeRTOS doit pouvoir fournir un lien vers le site du projet : `http://www.freertos.org/`.

Il existe deux variantes de FreeRTOS, toutes deux disponibles sous licences commerciales plus restrictives.

OpenRTOS est une version payante de FreeRTOS, développée par la sociétée *High Integrity Systems*. Il est possible d'utiliser OpenRTOS sans citer ses auteurs. Un support commercial complet est également disponible, contrairement à FreeRTOS qui lui s'appuie sur un forum des utilisateurs.

SafeRTOS est une version dérivée et payante de FreeRTOS, développée tout comme OpenRTOS par High Integrity Systems. Cette version est particulièrement dédiée aux systèmes critiques : en effet, elle à été développée en conformité avec la norme de sécurité *IEC 61508*. Un support commercial complet est disponible.

3.1.3 La popularité de FreeRTOS

Selon le site officiel, FreeRTOS est téléchargé en moyenne toutes les 170 secondes.

Le système est toujours en développement actif : la dernière version est la 2022.12.01. Pour suivre le développement : `https://github.com/FreeRTOS/FreeRTOS`

Les utilisateurs peuvent obtenir de l'aide et de la documentation de diverses manières :

- Sur le site officiel, des tutoriaux et guides de démarrages sont disponibles. On peut également y trouver la référence de l'API du système ;
- Le support aux utilisateur est fourni par le biais du forum de FreeRTOS ;
- Des livres écrits par le principal concepteur de FreeRTOS (Richard Barry) donnent une documentation complète sur le système, et de nombreux exemples d'applications. Il en existe différentes versions, dont une généraliste dédiée au système, et d'autres spécifiques à son implémentation sur certaines architectures comme les puces *Cortex* ou *Microchip*. Ces livres sont disponibles pour environ 30$.

3.2 Objectifs du multi-tâches

Un programme n'occupe jamais 100% du temps de l'unité centrale. En particulier, lorsque le programme est en attente d'entrées-sorties sur un périphérique, le processeur serait inutilisé pendant le temps d'attente s'il ne devait gérer que ce programme.

Il s'agit alors de mettre en œuvre les règles de possession et de partage des ressources. Une contrainte importante, est la garantie de l'intégrité de chaque programme. Un programme donné ne doit pas être «nuisible» pour les autres.

Le multi-tâches fait coexister en mémoire plusieurs programmes, sachant que ces derniers seront exécutés alternativement dans le temps par une seule unité centrale. La répartition temporelle se en fonction des ressources d'entrées/sorties requises par les programmes en concurrence. La notion de priorité intervient également sur le partage du temps.

3.2.1 Les processus

Un **programme** est une entité composée de une ou plusieurs séquences d'instructions agissant sur un ensemble de données. Un programme est **statique**. C'est un ensemble séquentiel d'informations occupant une partie de la mémoire.

Un **processus** est une entité dynamique. Un processus représente l'exécution de un ou plusieurs programmes. Il présente des caractéristiques évoluant dans le temps.

On appelle souvent **tâche** un processus cyclique. C'est souvent le cas pour les applications industrielles.

Un processus peut être

- créé
- exécuté
- détruit

Un processus comporte en général trois zones (Fig. 3.1) :

1. une zone **programme** qui contient les instructions à exécuter et parfois les constantes de l'application
2. une zone de **données** accessible en lecture et en écriture : elle contient les variables globales de l'application.
3. une zone de **pile** permettant de ranger les données temporaires de l'application (paramètres des sous-programmes, adresse de retour des sous-programmes, variables locales ...).

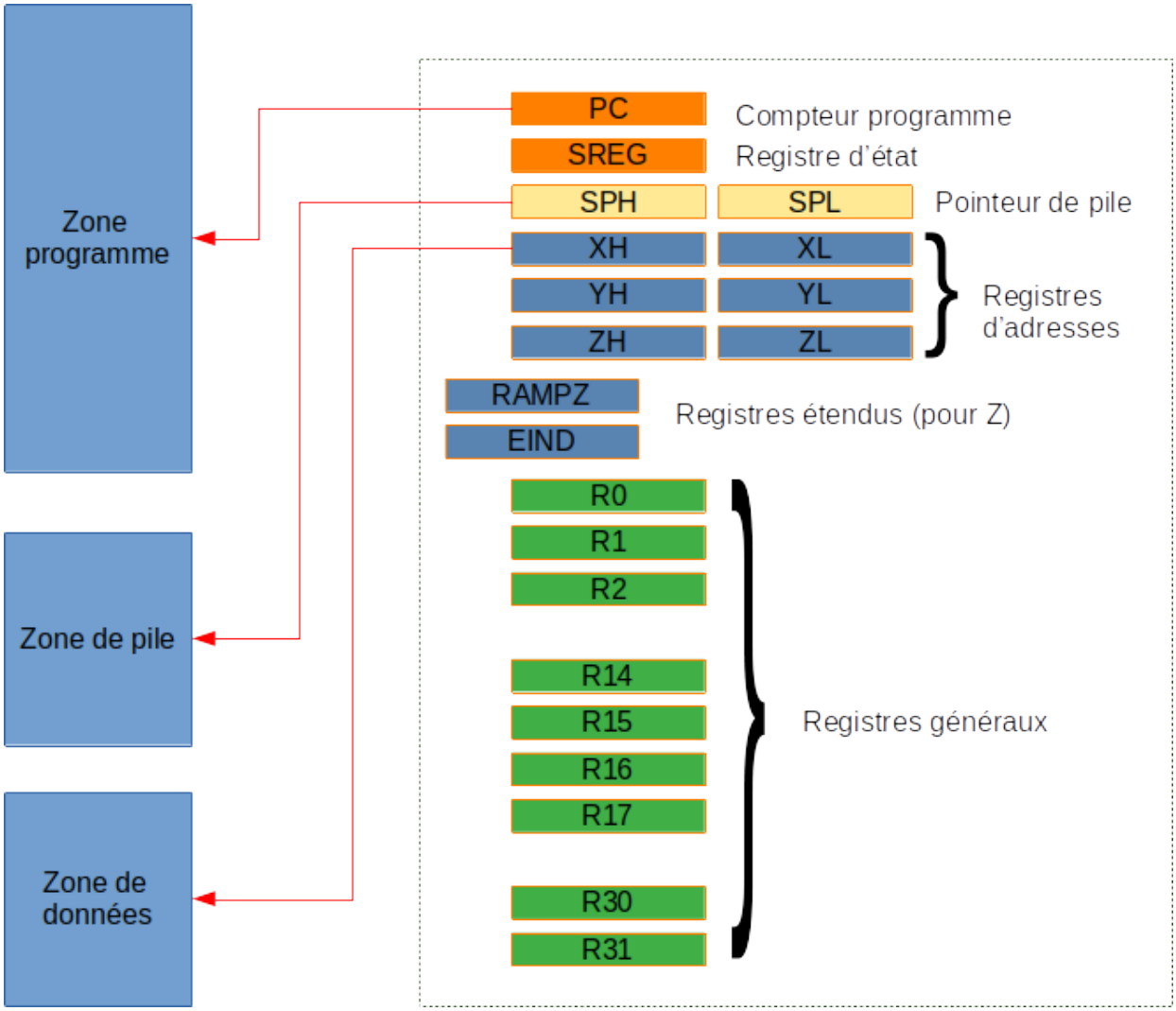

FIGURE 3.1 – *Occupation de la mémoire par un processus.*

L'accès aux différentes zones se fait par rapport aux contenus des registres du processeur. Dans la figure 3.1, les registres représentés sont ceux de l'AVR ATMEGA2569 L'instruction en cours est marquée par le «compteur programme» du microprocesseur (PC[1]). La pile est marquée par le registre SP[2]. La zone de donnée est marquée par un registre décidé par le système d'exploitation (Ici, c'est X).

Les différentes informations contenues dans les registres du processeur caractérisent le processus en cours d'exécution, «à l'instant t». À cet instant, ce processus est propriétaire du processeur. On appelle **contexte d'exécution** ces différentes informations.

Si on parvient à stopper le processus en cours en conservant en mémoire le contenu des registres du processeur,alors, on est capable de le relancer en rechargeant les registres du processeur avec ces mêmes

1. Program Counter
2. Stack Pointer

valeurs. Nous verrons plus loin le mécanisme qui permet de sauvegarder les registres du processeur sans perdre le contexte d'exécution du processus interrompu. La sauvegarde des contenus des registres d'un processus lorsqu'il est momentanément interrompu font partie du **contexte sauvegardé** de ce processus.

Plus généralement, dans le **contexte** d'un processus, on trouvera :

- l'état sauvegardé de chacun des registres du processeur, y compris le registre d'état ;
- le nom du processus ;
- un identificateur (attribué au moment de la création du processus)
- une priorité ;
- une variable permettant au système de mémoriser l'état courant du processus («en attente», «actif», ...).

Toutes ces informations sont contenues dans une structure de données appelée **descripteur de processus**.

3.3 Gestion des descripteurs de processus

Le système multi-tâches a pour objet de gérer l'ensemble des descripteurs de processus. En général, ils sont stockés dans des files. Il y a autant de files que d'états possibles pour les processus. Par exemple, il y aura une file pour les processus «en cours», une autre pour les processus «en attente».

Faire changer d'état à un processus consiste alors à le faire passer d'une file à une autre. Supprimer un processus, c'est récupérer la mémoire qu'il occupe et l'éliminer de la file où il figure.

3.4 Tâches et co-routines

Le système permet de diviser le code selon deux stratégies :

- Les *tâches* proprement dites ;
- Les *co-routines*, qui sont des tâches allégées.

Les co-routines, implémentées dans FreeRTOS depuis la version 4.0, sont des tâches destinées à être exécutées sur des processeurs avec de grandes contraintes en terme de mémoire.

FreeRTOS permet le développement d'applications utilisant seulement des tâches, ou seulement des des co-routines. Il est également possible pour des applications d'utiliser les deux en même temps. Il faut savoir que ces deux entités utilisant des fonctions de l'API différentes, il n'est pas possibles d'utiliser des outils tels que les files de messages ou les sémaphores pour la communication entre une tâche et une co-routine, et réciproquement.

3.4.1 Les différences entre les tâches et les co-routines

Dans une application, et à un moment donné, une seule tâche est exécutée par le processeur. Les tâches sont des entités indépendantes les unes des autres, c'est à dire qu'elle sont chacune pourvues de leur propre pile. Les co-routines, quant à elles, partagent toutes la même pile.

Le fait que les co-routines partagent toutes une seule pile permet au système qui les implémentent de consommer moins de mémoire, par rapport à un système fonctionnant avec des tâches. Par contre, l'usage des co-routines est plus restrictif, du fait de leur accès à une quantité de mémoire plus limitée.

Au niveau des priorités, il faut savoir que les co-routines sont classées par priorité les unes par rapport aux autres. De plus, une tâche sera toujours prioritaire par rapport à une co-routine.

3.5 Les tâches

Le cycle de vie d'une tâche

A tout moment de l'exécution d'une application sous FreeRTOS, chacune des tâches créées dans le système possède un *état*. La liste des états possibles est la suivante :

- **En cours d'exécution** (*Running*) : La tâche est exécutée en ce moment et elle utilise le processeur ;

- **Prête** (*Ready*) : la tâche est prête à être exécutée (elle n'est ni suspendue, ni bloquée), mais est en attente car une tâche d'une priorité égale ou supérieure utilise actuellement le processeur ;
- **Bloquée** (*Blocked*) : Une tâche peut être bloquée pour plusieurs raisons : Attente d'un événement temporel ou externe, d'un événement sur une file de message ou un sémaphore. Les tâches bloquées ont toutes un délai au delà duquel elle sont débloquées. Les tâches bloquées ne sont pas examinées par l'ordonnanceur ;
- **Suspendue** (*Suspended*) : Tout comme les tâches bloquées, les tâches suspendues ne sont pas examinées par l'ordonnanceur. la différence principale entre une tâche suspendue et une tâche bloquée est que la tâche suspendue peut l'être indéfiniment, contrairement à la tâche bloquée qui l'est jusqu'à l'expiration du délai.

On peut modéliser le cycle de vie d'une tâche sous FreeRTOS par l'automate sur la figure 3.2 page 20.

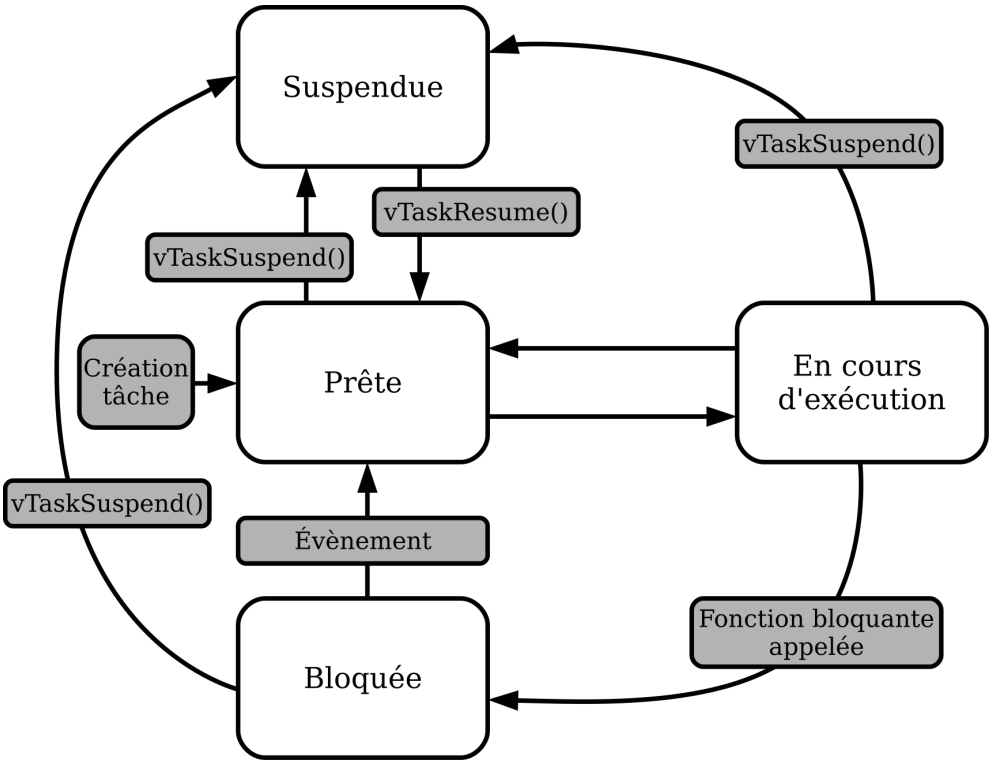

FIGURE 3.2 – *Le cycle de vie d'une tâche sous FreeRTOS.*

La *pile* d'une tâche

Chaque tâche créée dans le système possède une pile : c'est un espace continu en mémoire RAM, utilisé pour stocker les variables locales à la tâche, ainsi que pour sauvegarder le contexte de la tâche lors de sa suspension.

Création, manipulation, et destruction

Création d'une tâche Une tâche est créée grâce à la fonction `xTaskCreate` :

```
portBASE_TYPE xTaskCreate(
  pdTASK_CODE pvTaskCode,
  const portCHAR * const pcName,
  unsigned portSHORT usStackDepth,
  void *pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle *pvCreatedTask
);
```

La tâche nouvellement créée est ainsi directement ajoutée à la liste de tâches prêtes à être exécutées.
Parmi les paramètres de cette fonction on peut noter :

- `pvTaskCode` : Un pointeur sur la fonction d'entrée de la tâche ;
- `pcName` : Une chaîne de caractère représentant le nom de la tâche (utilisée pour le débogage) ;
- `usStackDepth` : Le nombre de variables que peut contenir la pile associée à la tâche (autrement dit, le nombre de variables locales à la tâche) ;
- `pvParameters` : Les paramètres passés à la fonction d'entrée de la tâche lors de sa création ;
- `uxPriority` : La priorité de la tâche ;
- `pvCreatedTask` Un *handle*, identifiant de la tâche créée servant par exemple pour la suppression de cette tâche à partir d'une autre tâche.

Destruction d'une tâche La destruction d'une tâche se fait par l'appel à la fonction `vTaskDelete`. Cela peut se faire de deux manières différentes :

- **Destruction de la tâche appelante :** L'instruction `vTaskDelete(NULL);` provoque la suppression de la tâche appelante. Cette dernière est alors retirée du système ;

- **Destruction d'une tâche à partir d'une autre tâche :** L'instruction `vTaskDelete(task_B_Handle);`, appelée à partir d'une tâche A, provoque la destruction d'une tâche B identifiée par le *handle* `task_B_Handle`.

Manipulation des tâches Les différentes fonctions de l'API dédiées à la manipulation des tâches sont :

```
void vTaskDelay( portTickType xTicksToDelay );
void vTaskDelayUntil( portTickType *pxPreviousWakeTime,
                      portTickType xTimeIncrement );
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
void vTaskPrioritySet( xTaskHandle pxTask,
                       unsigned portBASE_TYPE uxNewPriority );
void vTaskSuspend( xTaskHandle pxTaskToSuspend );
void vTaskResume( xTaskHandle pxTaskToResume );
portBASE_TYPE xTaskResumeFromISR( xTaskHandle pxTaskToResume );
```

Blocage des tâches : Les fonctions `vTaskDelay` et `vTaskDelayUntil` permettent de bloquer la tâche appelante. La première fonction bloque la tâche à partir du moment où la tâche est appelée, pendant le temps spécifié en paramètre (blocage temporel relatif). La seconde fonction prend deux paramètres qui représentent les temps entre lesquels la tâche appelante est bloquée (blocage temporel absolu).

Priorités : La fonction `uxTaskPriorityGet` renvoie la priorité de la tâche dont le *handle* est passé en paramètres. La fonction `vTaskPrioritySet`, quant à elle, permet de modifier la priorité de la tâche dont le *handle* est passé en paramètre.

Suspension et reprise des tâches : Pour suspendre une tâche (la faire passer à l'état *suspendue*), on utilise la fonction `vTaskSuspend`. Passer `NULL` en paramètre à cette fonction provoque la suspension de la tâche appelante. On peut également passer un *handle* en paramètre, pour suspendre une autre tâche via la tâche appelante.
Pour faire passer une tâche de l'état *suspendue* à l'état *prête*, on utilise la fonction `vTaskResume`, en passant en paramètre le *handle* de la tâche dont l'exécution doit reprendre.

3.6 L'ordonnancement

3.6.1 Un système multitâche

FreeRTOS est un système *multitâche* : plusieurs tâches présentes dans le système donnent l'impression de s'exécuter en même temps. En réalité, à un moment donné, une seule tâche utilise le processeur. C'est

le travail de l'ordonnanceur que d'allouer des tranches de temps processeur aux différentes tâche, et de commuter entre elles rapidement, donnant ainsi l'illusion que toutes les tâches son exécutées en même temps.

3.6.2 L'ordonnanceur temps réel de FreeRTOS

Le choix de la tâche à exécuter. L'ordonnanceur de FreeRTOS choisit quelle tâche doit être exécutée par le processeur, parmi la liste des tâches prêtes à s'exécuter (i.e. les tâche dans l'état *ready*).

Ce choix est basé sur la priorité des tâches. Dans une application utilisant des tâches et des co-routines, il faut savoir qu'une tâche proprement dite sera toujours prioritaire par rapport à une co-routine. On peut ainsi dire que les co-routines sont prioritaires entre elles, de même que les tâches.

L'ordonnanceur donne un temps processeur équitable aux tâches de même priorité.

Un exemple de scénario d'ordonnancement temps réel préemptif sous FreeRTOS est représenté sur la figure 3.3 page 23.

La tâche *inactive*. Le système implémente automatiquement une tâche inactive (*idle task*). C'est une tâche qui est exécutée sur le processeur lorsqu'il n'y à rien d'autre (tâche, co-routines) à exécuter. Cette tâche est créée automatiquement au lancement de l'ordonnanceur.

Sous FreeRTOS, l'une des fonctions de la tâche inactive est de libérer la mémoire occupée par les tâches précédemment supprimées. Il est ainsi important dans un système ou l'on supprime des tâches, de s'assurer que la tâche inactive dispose d'un temps processeur suffisant.

On peut également noter que l'on peut donner à une tâche la même priorité que la tâche inactive (la constante `tskIDLE_PRIORITY`).

3.6.3 Les commutations de contexte

Le *tic* du système

Le système d'exploitation FreeRTOS base toutes ses opérations autour d'une unité nommée le *tic*. À chaque interruption du *timer*, la variable `xTickCount` est incrémentée et permet au système de mesurer le temps. À intervalles de temps réguliers, une ISR (Interrupt Service Routine), une fonction dont l'exécution est déclenchée par la réception d'un signal d'interruption, ici un signal généré par le *timer* nommée `prvTickISR` est lancée, qui se charge entre autres d'incrémenter `xTickCount` via un appel à `vTaskIncrementTick`.

`vTaskIncrementTick` vérifie également si un délai associé au blocage d'une tâche se termine. Dans ce cas, la tâche en question est mise dans la liste des tâches à l'état prêt.

Les commutations de contexte

Lorsqu'une tâche A est interrompue pour passer la main à une autre tâche B de priorité supérieure, le système effectue ce qu'on appelle une *commutation de contextes* (Pour exemple, voir ce qu'il se passe au temps $t7$ sur la figure 3.3). Cela implique la sauvegarde du contexte de la tâche A interrompue (instruction et variables locales à la tâche), ainsi que la mise en place / la restauration de celui de la tâche B.

La figure 3.4 page 24 illustre le contexte d'une tâche A en cours d'exécution dans le système.

La commutation de contexte s'effectue de manière suivante. On prendra l'exemple d'une tâche A en cours d'exécution, qui va être préemptée par l'arrivée d'une tâche B (précédemment bloquée) de priorité supérieure :

1. Au départ, la tâche A est en cours d'exécution sur le processeur ;
2. Lorsque le tic du système se produit, le microcontrôleur place le «Program Counter» de la tâche A sur la pile (en mémoire) de cette même tâche ;
3. L'ensemble du contexte de la tâche A est placé en sommet de pile (la pile de la tâche A) par l'ISR `pvrTickISR`. Le pointeur de pile de la tâche A pointe maintenant vers le sommet de son propre contexte sauvegardé. Ce pointeur est sauvé ;

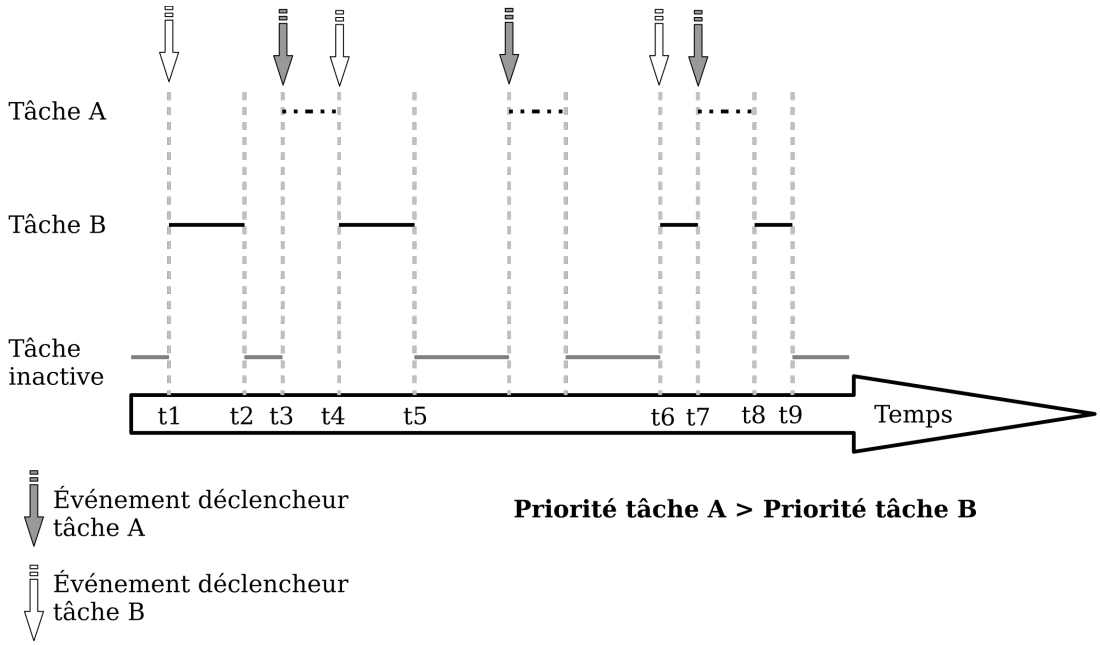

FIGURE 3.3 – *Scénario d'ordonnancement temps réel préemptif sous FreeRTOS :* On se place dans un système implémentant deux tâches : A et B. La priorité de la tâche A est supérieure à celle de la tâche B. Chaque tâche passe dans l'état *prête* suite à un événement déclencheur (par exemple l'appui sur une touche, ou un événement temporel déclenché à intervalles réguliers) : l'événement déclencheur pour la tâche A est représenté par la flèche grise, celui pour la tâche B par la flèche blanche. Au temps t_1, l'événement déclencheur de la tâche B survient. Cette dernière passe alors à l'état *prête*, et est choisie par l'ordonnanceur pour être exécutée. Une fois sont exécution terminée au temps t_2, la tâche B rend la main à l'ordonnanceur, qui choisit d'exécuter la tâche inactive car il n'y a aucune tâche de priorité supérieure prête. Au temps t_3, c'est le déclencheur de A qui survient, A est exécutée. À la fin de son exécution (temps t_4), suite au déclencheur de B, B est exécutée et rend la main en t_5. Au temps t_7, B est en cours d'exécution et l'événement déclencheur de A survient. Comme A a une priorité supérieure à B, l'exécution de B est stoppée, et A prend sa place. Lorsque A à finit son travail, au temps t_8, B peut terminer son exécution (temps t_9)

4. L'ISR fait alors appel à la fonction `vTaskIncrementTick`. Cette dernière s'aperçoit que le timer de blocage de la tâche B a expiré. La tâche B passe donc dans l'état *prête*. La tâche B a une priorité supérieure à la priorité de la tâche A. L'ISR lance alors la fonction `vTaskSwitchContext`, qui élit la tâche B pour être exécutée par le processeur ;
5. Il faut alors retrouver le contexte de la tâche B. Le pointeur de pile de la tâche B, précédemment sauvé tout comme celui de la tâche A, est retrouvé. Il est placé dans l'emplacement pour pointeur de pile du processeur, qui pointe maintenant vers le sommet de la pile de la tâche B. Cette pile contient le contexte de la tâche B. La fonction `portRESTORE_CONTEXT` est lancée et restaure le contexte de la tâche B, de sa pile vers les registres du processeur. Il ne reste plus que le *PC* de la tâche B dans la pile ;
6. L'ISR retourne. Le *PC* de B est alors restauré. Le système se retrouve dans l'état dans lequel il était lorsque la tâche B à été bloquée. La commutation de contexte est complète.

3.6.4 Exemple 1 : création de 4 tâches

Chaque tâche pilote une sortie, avec une période qui lui est propre. Ce premier exemple montre déjà l'intérêt du multitâche, puisque cela revient ici à écrire 4 fonctions très simples, plutôt que de faire un calcul dans une bouovle unique.

Contexte d'exécution d'une tâche :

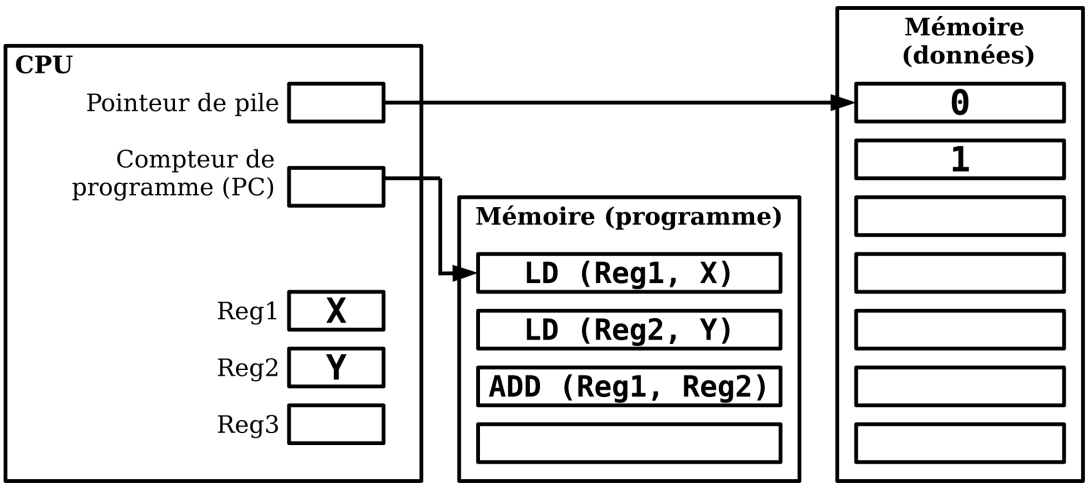

FIGURE 3.4 – *Représentation du contexte d'une tâche dans le système :* On peut différencier en mémoire d'un coté la zone occupée par les instruction de la tâche, et de l'autre la zone contenant les données (variables locales, ...). Dans les registres du processeur on retrouve le *Program Counter* (PC), qui pointe vers l'instruction en cours d'exécution de la tâche, et le pointeur de pile qui pointe vers la pile associée à la tâche. Différents registres sont également occupés par les opérandes de différentes instructions de la tâche.

```
1  /**
2   * 4 tâches pilotant les 4 sorties
3   **/
4  #include <Arduino_FreeRTOS.h>
5  // Définition des sorties
6  const byte S3 = 13;
7  const byte S2 = 12;
8  const byte S1 = 11;
9  const byte S0 = 10;
10 //définition des entrées
11 const byte E0 = 2;
12 const byte E1 = 3;
13
14 int T0 = 1000 / portTICK_PERIOD_MS; // 1 seconde
15 int T1 = 300 / portTICK_PERIOD_MS; // 300 ms
16 int T2 = 700 / portTICK_PERIOD_MS;
17 int T3 = 100 / portTICK_PERIOD_MS;
18
19 /*---------------------- Taches --------------------*/
20 void Tache_S0(void *pvParameters) {
21   for (;;) {
22     digitalWrite(S0, 1);
23     vTaskDelay(T0);
24     digitalWrite(S0, 0);
25     vTaskDelay(T0);
26   }
27 }
28
29 void Tache_S1(void *pvParameters) {
30   for (;;) {
31     digitalWrite(S1, 1);
32     vTaskDelay(T1);
33     digitalWrite(S1, 0);
34     vTaskDelay(T1);
35   }
36 }
37
38 void Tache_S2(void *pvParameters) {
```

```
39    for (;;) {
40       digitalWrite(S2, 1);
41       vTaskDelay(T2);
42       digitalWrite(S2, 0);
43       vTaskDelay(T2);
44    }
45  }
46  void Tache_S3(void *pvParameters) {
47    for (;;) {
48       digitalWrite(S3, 1);
49       vTaskDelay(T3);
50       digitalWrite(S3, 0);
51       vTaskDelay(T3);
52    }
53  }
54
55
56  void runRTOS() {
57    // Création des taches : code, nom, pile, param, priorité, &id
58    xTaskCreate(Tache_S0,"S0", 128, NULL, 1, NULL);
59    xTaskCreate(Tache_S1,"S1", 128, NULL, 1, NULL);
60    xTaskCreate(Tache_S2,"S2", 128, NULL, 1, NULL);
61    xTaskCreate(Tache_S3,"S3", 128, NULL, 1, NULL);
62    // Lancement RTOS
63    vTaskStartScheduler();
64  }
65
66  void setup() {
67    pinMode(S0,OUTPUT);
68    pinMode(S1,OUTPUT);
69    pinMode(S2,OUTPUT);
70    pinMode(S3,OUTPUT);
71    pinMode(E1,INPUT);
72    pinMode(E0,INPUT);
73    runRTOS();
74  }
75
76  void loop(){ /* VIDE */ }
```

3.6.5 Exemple 2 : optimisation de la quantité de code.

Lorsque plusieurs tâches font exactement la même chose mais entraitant des données différentes, il convient de les paramétrer. Dans l'exemple précédent, chaque tâche est conditionnée par une valeur de sortie, et une période.

freeRTOS permet de passer un pointeur vers une donnée. Il suffit donc de créerr une structure qui englobe les deux paramètres et de donner l'adresse de cette structure au moment de la création de la tâche.

```
1   /**
2    * 4 tâches pilotant les 4 sorties : une seule zone de code
3    **/
4   #include <Arduino_FreeRTOS.h>
5
6   struct Param {
7     const byte S;
8     int T;
9   } TP[] = {{13, 100 / portTICK_PERIOD_MS},
10            {12, 700 / portTICK_PERIOD_MS},
11            {11, 300 / portTICK_PERIOD_MS},
12            {10, 1000 / portTICK_PERIOD_MS}};
13
14  // Code commun aux tâches
15  void Tache_S(void *pvParameters) {
16    struct Param *p = (struct Param *)pvParameters;
```

```
17    for (;;) {
18       digitalWrite(p->S, 1);
19       vTaskDelay(p->T);
20       digitalWrite(p->S, 0);
21       vTaskDelay(p->T);
22    }
23  }
24
25
26  void runRTOS() {
27    // Création des taches : code, nom, pile, param, priorité, &id
28    xTaskCreate(Tache_S,"S3", 64, &TP[0], 1, NULL);
29    xTaskCreate(Tache_S,"S2", 64, &TP[1], 1, NULL);
30    xTaskCreate(Tache_S,"S1", 64, &TP[2], 1, NULL);
31    xTaskCreate(Tache_S,"S0", 64, &TP[3], 1, NULL);
32    // Lancement RTOS
33    vTaskStartScheduler();
34  }
35
36  void setup() {
37    pinMode(TP[0].S, OUTPUT);
38    pinMode(TP[1].S, OUTPUT);
39    pinMode(TP[2].S, OUTPUT);
40    pinMode(TP[3].S, OUTPUT);
41    runRTOS();
42  }
43
44  void loop(){ /* VIDE */ }
```

Par défaut le type du paramètre reçu est un `void *` : ceci permet à FreeRTOS de s'adapter à toutes les situations. Du coup, il faut procéder à la conversion de type pour éviter les erreurs de compilation :

```
struct Param *p = (struct Param *)pvParameters;
```

3.7 Exclusion mutuelle et synchronisation

Exclusion mutuelle Dans de nombreuses applications, les processus doivent partager des ressources qui ne doivent être utilisées que par un seul processus simultanément (impression, voie de communication, ...). On parle alors d'**exclusion mutuelle** et de **ressource critique**. Une **région critique** est à une séquence d'instructions non partageable. C'est à dire qu'une seule tâche peut être rendue à ce point d'exécution à un instant donné. Les instructions incriminées utilisent la ressource critique.

Exemple : Les lignes suivantes permettent de réaliser une conversion analogique → numérique sur une carte industrielle présentant plusieurs voies. Il est clair qu'une fois une voie sélectionnée, il faut faire complètement l'acquisition (jusqu'à obtenir le résultat) pour garantir la cohérence du programme.

```
Selection_Voie(V);
Attend_Fin_Conv;
Lecture(R);
```

Synchronisation : un processus doit attendre que un ou plusieurs autres processus aient établi un état du système adéquat, pour qu'il puisse se continuer.

Par exemple, un processus p doit attendre qu'une donnée ait été produite par un processus q avant de s'en servir. D'une manière plus générale, il existe dans le processus q une instruction I_q qui doit être exécutée avant une instruction I_p figurant dans le processus p.

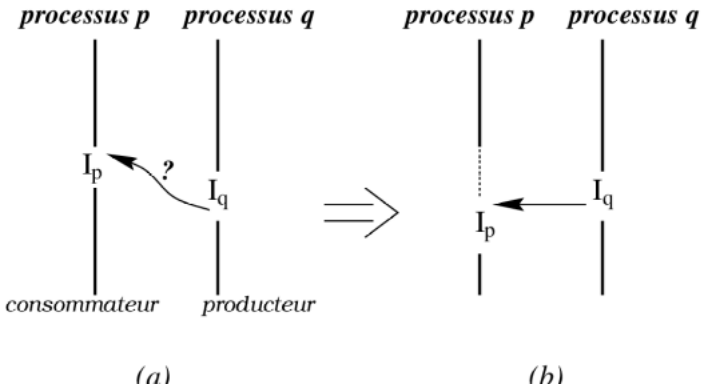

FIGURE 3.5 – *Synchronisation : le processus q produit une donnée qui doit être consommée par le processus p. Situation (a) : on ne sait pas si I_q sera exécutée avant I_p. Situation (b) : la synchronisation est réalisée.*

Solutions. Parmi les solutions qui ont été apportées pour gérer ces deux problèmes de coopération, citons :

1. les **sémaphores**, qui sont des outils «systèmes». Ils sont utilisés sur les systèmes ou noyaux multi-tâche, lorsque la programmation de l'application utilise un langage impératif ou objet ne présentant pas d'outils de synchronisation ;
2. les **«rendez-vous»** de `ada`, mécanisme de haut niveau prenant en compte tous les problèmes classiques de synchronisation ;
3. les méthodes synchronisées de `java` (mot clé = `synchronized`).

3.8 Les sémaphores

3.8.1 Définition du sémaphore

Un **sémaphore** S est constitué d'une variable entière $e(S)$ et d'une file d'attente $f(S)$. À sa création, la file d'attente est vide et $e(s)$ est initialisé à une valeur entière positive ou nulle, $e_0(S)$. Deux opérations indivisibles [3] et exclusives [4] permettent d'agir sur ces sémaphores : $P(S)$ et $V(S)$.

3. Les instructions qui les constituent ne sont jamais interrompues.
4. Il n'y en a pas d'autre.

$P(S)$

début
e(S) $\leftarrow$ e(S) - 1 ;
si (e(S) $<$ 0) ***alors***
début
$état(q)$ [a] $\leftarrow$ *bloqué* ;
mettre q dans la file f(S)
fin
fin

a. q étant le processus qui a appelé $P(S)$

$V(S)$

début
e(S) $\leftarrow$ e(S) + 1 ;
si (e(S) $\leq$ 0) ***alors***
début
sortir un processus q de la file f(S)
$état(q)$ $\leftarrow$ *actif* ;
fin
fin

À ces deux méthodes d'accès, on peut ajouter la phase de création du sémaphore qui fixera la valeur initiale $e_0(S)$ et qui créera la file d'attente pour les processus.

3.8.2 Utilisation du sémaphore pour gérer une exclusion mutuelle

L'utilisation d'un sémaphore passe par le respect de règles élémentaires :

- À chaque **ressource critique**, on associe un sémaphore spécifique S.
- La valeur initiale $e_0(S)$ est égale au nombre de processus pouvant utiliser simultanément la ressource (pour une exclusion mutuelle, $e_0(S) = 1$).
- Le début du code code de la **région critique** ($\Leftrightarrow$ le code non partageable) doit être précédé d'un appel à $P(S)$, ce qui équivaut, pour le processus à se mettre dans une file d'attente, en attendant que la ressource soit disponible.
- La fin de la région critique doit être suivie d'un appel $V(S)$, qui annonce que la ressource est libérée.

Ainsi, les lignes de programmes à protéger de l'exemple précédent seront entourées par les deux requêtes P et V :

```
...
P(S);
Selection_Voie(V);
Attend_Fin_Conv;
Lecture(R);
V(S);
...
```

La figure 3.6 représente la chronologie des actions dans le cas de la compétition entre deux processus p et q pour l'accès à une ressource critique pouvant être utilisée par un seul processus simultanément..

La valeur initiale $e_0(S) = 1$ affectée au sémaphore correspond au nombre de «places disponibles» dans la ressource critique. Lorsque la valeur du sémaphore est négative, sa valeur absolue donne le nombre de processus en attente dans la file $f(S)$.

3.8.3 Utilisation du sémaphore pour gérer une synchronisation

À chaque fois fois qu'un processus q produit une donnée qui doit être consommée par un processus p, le problème de la synchronisation se pose. Dans l'exemple général suivant, le processus q contient une instruction «`Produire`» qui doit être exécutée avant l'instruction «`Consommer`» du processus p.

Le sémaphore étant initialisé à $e_0(S) = 0$, le tableau suivant représente l'état du système dans le cas le moins favorable où le processus p cherche à consommer la donnée avant qu'elle ne soit produite.

`processus` p	`processus` q	$e(S)$	$f(S)$
...	...	0	{ }
P(S);	...	-1	{ p }
	Produire;	-1	{ p }
	V(S);	0	{ }
Consommer;	...	0	{ }
...	...	0	{ }

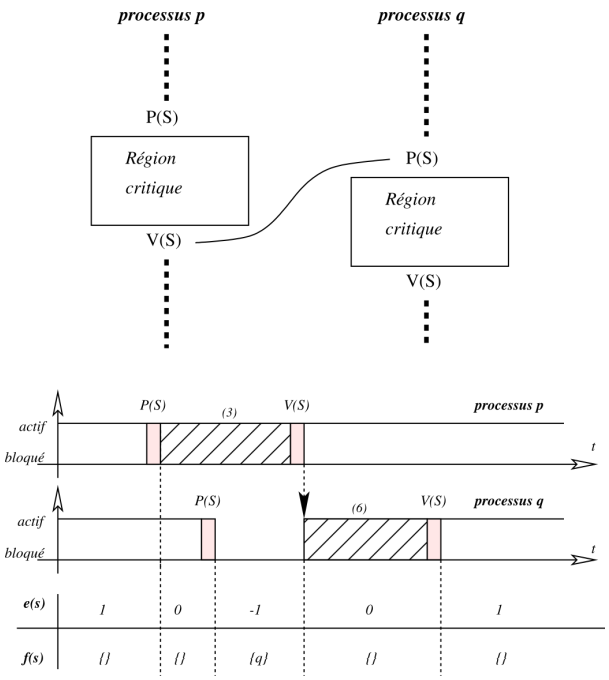

FIGURE 3.6 – *Deux tâches avec exclusion mutuelle.*

La figure 3.7 *(a)* représentent cette même situation sous forme de chronogramme. Le deuxième cas *(b)* illustre la situation la plus favorable (la donnée est prête).

3.9 L'exclusion mutuelle avec les «mutex»

Les *mutexes* sont des sémaphores binaires qui intègrent la notion de priorité. Ils permettent de protéger les ressources critiques des accès concurrents.

Voici une première mise en œuvre qui met en évidence une ressource critique sans utiliser le mécanisme de protection :

```
1  #include <Arduino_FreeRTOS.h>
2
3  struct Param {
4    int Periode;
5    char Message[16];
6  };
7
8  struct Param T[3] = {
9    { 50, "Lecture" },
10   { 70, "Ecriture"},
11   { 80, "Interruption"}
12 };
13
14
15 /*-------------------------------------------------*/
16 /*---------------------- Taches -------------------*/
17 /*-------------------------------------------------*/
18
19 void fCritique(char *str){
20   while (*str) {
21      Serial.print(*str);
22      vTaskDelay(1);
23      str++;
24   }
25   Serial.print('\n');
```

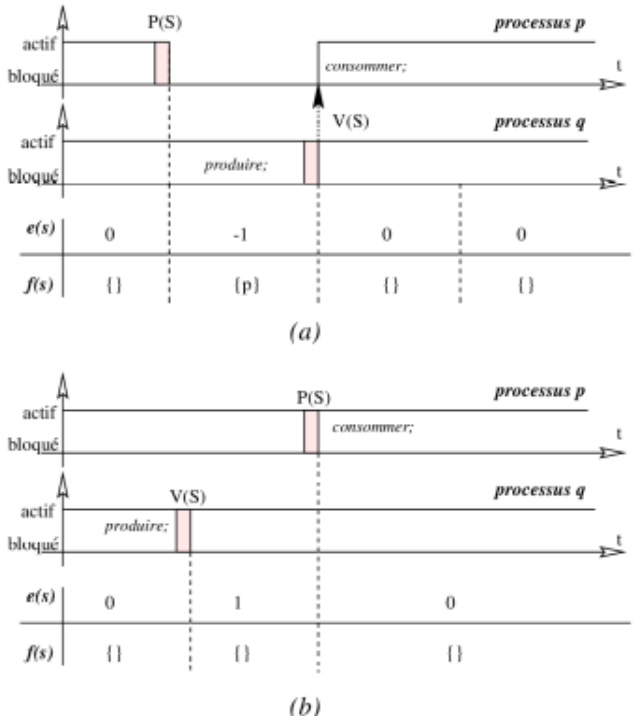

FIGURE 3.7 – *Synchronisation dans le cas défavorable (a) et dans le cas favorable (b).*

```
26  }
27
28  void TacheWR(void *pvParameters) {
29    struct Param *data = (struct Param *)pvParameters;
30    for (;;) {
31      fCritique(data->Message);
32      vTaskDelay(data->Periode);
33    }
34  }
35
36  /*-----------------------------------------------------*/
37  /*---------------------- INIT -------------------------*/
38  /*-----------------------------------------------------*/
39
40  // Démarrage de la carte
41  void setup() {
42
43    Serial.begin(9600);
44
45    // Création des taches : code, nom, pile, param, priorité, &id
46    xTaskCreate(TacheWR, "WR1", 128, (void *)(&T[0]), 1, NULL);
47    xTaskCreate(TacheWR, "WR2", 128, (void *)(&T[1]), 1, NULL);
48    xTaskCreate(TacheWR, "WR3", 128, (void *)(&T[2]), 1, NULL);
49
50
51  }
52
53  void loop(){ /* VIDE */ }
```

L'exécution du programme (Fig. 3.8) montre bien l'incohérence de l'affichage.

Dans la deuxième version, le mécanisme de *mutex* est mis en place. À la différence d'autres noyaux, il est important avec FreeRTOS de tester la valeur renvoyée par `xSemaphoreTake()`.

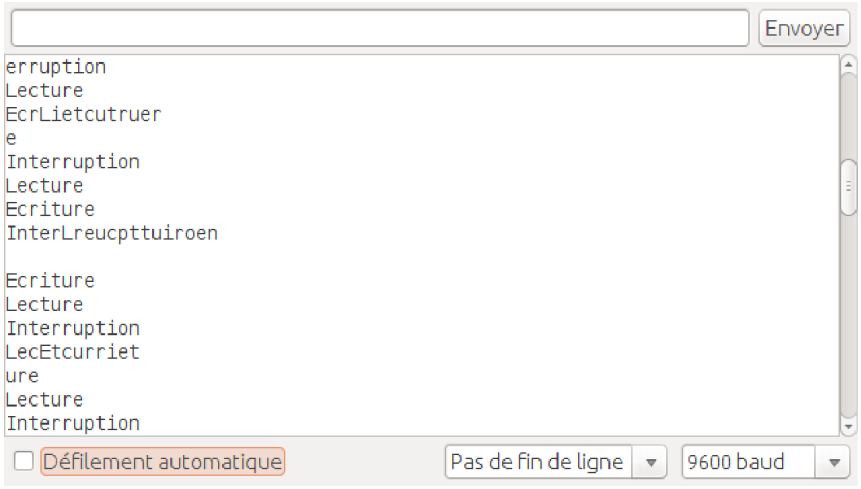

FIGURE 3.8 – *Trace d'exécution sans sémaphore.*

```
1  #include <Arduino_FreeRTOS.h>
2  #include <semphr.h>
3
4
5  struct Param {
6    int Periode;
7    char Message[16];
8  };
9
10 struct Param T[3] = {
11   { 50, "Lecture" },
12   { 70, "Ecriture"},
13   { 80, "Interruption"}
14 };
15
16 SemaphoreHandle_t xSema;
17
18 /*-----------------------------------------------------*/
19 /*---------------------- Taches -----------------------*/
20 /*-----------------------------------------------------*/
21
22 void fCritique(char *str){
23   while (*str) {
24      Serial.print(*str);
25      vTaskDelay(1);
26      str++;
27   }
28   Serial.print('\n');
29 }
30
31 void TacheWR(void *pvParameters) {
32   struct Param *data = (struct Param *)pvParameters;
33   for (;;) {
34     if (xSemaphoreTake(xSema, 0) == pdTRUE) {
35       fCritique(data->Message);
36       xSemaphoreGive(xSema);
37     }
38     vTaskDelay(data->Periode);
39   }
40 }
41
42 /*-----------------------------------------------------*/
43 /*---------------------- INIT -------------------------*/
44 /*-----------------------------------------------------*/
45
```

```
46  // Démarrage de la carte
47  void setup() {
48
49    Serial.begin(9600);
50
51    xSema = xSemaphoreCreateMutex();
52
53    // Création des taches : code, nom, pile, param, priorité, &id
54    xTaskCreate(TacheWR,"WR1", 128, &T[0], 1, NULL);
55    xTaskCreate(TacheWR,"WR2", 128, &T[1], 1, NULL);
56    xTaskCreate(TacheWR,"WR3", 128, &T[2], 1, NULL);
57
58    // Lancement RTOS
59    vTaskStartScheduler();
60  }
61
62  void loop(){ /* VIDE */ }
```

La nouvelle sortie du programme est donnée dans la figure Fig. 3.9.

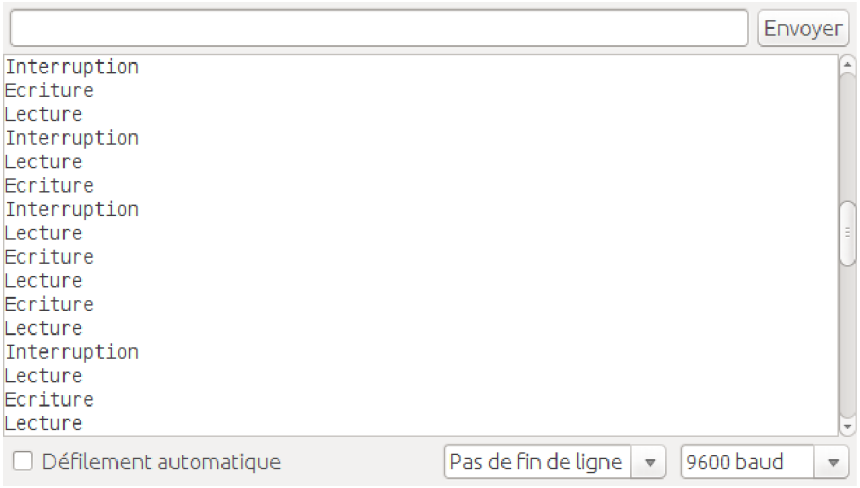

FIGURE 3.9 – *Trace d'exécution sans sémaphore.*

Chapitre 4

FreeRTOS et interruptions

Cette fois, les paramètres du programmes sont pilotés à partir d'une application sur le PC. Vous avez le choix entre mode texte, ou mode graphique. 3.4 Traitement des interruptions périodiques avec FreeRTOS Les interruptions matérielles sont spécifiques à l'architecture et non pas à FreeRTOS. Pour relier l'interruption au reste du code, il faut l'associer et la synchroniser avec au moins une tâche «FreeRTOS» différée. Les principes à respecter sont les suivants :

- un code le plus court possible pour le traitement de l'interruption (surtout pas de boucle . . .)
- pour un traitement pouvant être plus long, on place le code dans une tâche «normale» différée, qui respecte les règles de priorité de l'application ;
- dans ce cas, le code de l'interruption peut se limiter à un signalement de sémaphore binaire.

4.1 Solution avec une synchronisation par les fonctions du noyau

On utilise les fonctions *vTaskSuspend()* et *xTaskResumeFromISR()*. La tâche de synchronisation se suspend elle-même par un appel à
`vTaskSuspend(NULL);`
C'est la routine d'interruption qui «débloque» la tâche de synchronisation avec
`xTaskResumeFromISR(id);`
id étant l'identificateur de la tâche de synchronisation, obtenu au moment de sa création

```
1  /**
2   * Interruption et freertos
3   **/
4  #include <Arduino_FreeRTOS.h>
5  #include <semphr.h>
6
7  #define S 12
8  const byte BOUTON = 2;
9  int MARCHE = 1;
10
11 TaskHandle_t id;
12
13 void Tache_S(void *pvParameters) {
14   for (;;) {
15     if (MARCHE == 1) {
16      digitalWrite(S, 1);
17      vTaskDelay(100 / portTICK_PERIOD_MS);
18      digitalWrite(S, 0);
19      vTaskDelay(100 / portTICK_PERIOD_MS);
20     } else vTaskDelay(100 / portTICK_PERIOD_MS);
21   }
22 }
23
```

```
24  void ISR_Bouton() {
25    Serial.print("*");
26    xTaskResumeFromISR(id);
27  }
28
29  // Tache pour le traitement différé de l'ISR
30  void Tache_ISRBouton(void *pvParameters) {
31    while(1) {
32      vTaskSuspend(NULL);
33      MARCHE = 1-MARCHE;
34      Serial.println(MARCHE);
35      while (digitalRead(BOUTON) == 0) vTaskDelay(1);
36    }
37  }
38
39  void runRTOS() {
40    // Création des taches : code, nom, pile, param, priorité, &id
41    xTaskCreate(Tache_S,"S", 128, NULL, 1, NULL);
42    xTaskCreate(Tache_ISRBouton,"<<IT>>", 128, NULL, 1, &id);
43    // Lancement RTOS
44    vTaskStartScheduler();
45  }
46
47  void setup() {
48    Serial.begin(115200);
49    pinMode(S, OUTPUT);
50    pinMode(BOUTON, INPUT_PULLUP);
51    attachInterrupt(digitalPinToInterrupt(BOUTON), ISR_Bouton, INPUT_PULLUP);
52    Serial.println("GO");
53    runRTOS();
54  }
55
56  void loop(){ /* VIDE */ }
```

4.2 Synchronisation avec un sémaphore binaire

La deuxième possibilité consiste à utiliser un sémaphore binaire qui, bien utilisé, assurera la cohérence entre l'interruption matérielle et l'environnement RTOS.

```
1   /**
2    * Synchronisation ISR avec RTOS par sémaphore
3    **/
4   #include <Arduino_FreeRTOS.h>
5   #include <semphr.h>
6
7   #define S 12
8   const byte BOUTON = 2;
9   int MARCHE = 1;
10
11  SemaphoreHandle_t xSema = NULL;
12
13  void Tache_S(void *pvParameters) {
14    for (;;) {
15      if (MARCHE == 1) {
16       digitalWrite(S, 1);
17       vTaskDelay(100 / portTICK_PERIOD_MS);
18       digitalWrite(S, 0);
19       vTaskDelay(100 / portTICK_PERIOD_MS);
20      } else vTaskDelay(100 / portTICK_PERIOD_MS);
21    }
22  }
23
24  void ISR_Bouton() {
25    Serial.print("*");
```

```
26    static BaseType_t xHigherPriorityTaskWoken = pdTRUE;
27    xSemaphoreGiveFromISR(xSema, &xHigherPriorityTaskWoken);
28  }
29
30  // Tache pour le traitement différé de l'ISR
31  void Tache_ISRBouton(void *pvParameters) {
32    while(1) {
33      if (xSemaphoreTake(xSema, 0) == pdTRUE) {
34          MARCHE = 1-MARCHE;
35          Serial.println(MARCHE);
36          while (digitalRead(BOUTON) == 0) vTaskDelay(1);
37      }
38    }
39  }
40
41  void runRTOS() {
42    // Création des taches : code, nom, pile, param, priorité, &id
43    xTaskCreate(Tache_S,"S", 128, NULL, 1, NULL);
44    xTaskCreate(Tache_ISRBouton,"<<IT>>", 128, NULL, 1, NULL);
45    // Création du sémaphore de synchro
46    xSema = xSemaphoreCreateBinary();
47    // Lancement RTOS
48    vTaskStartScheduler();
49  }
50
51  void setup() {
52    Serial.begin(115200);
53    pinMode(S, OUTPUT);
54    pinMode(BOUTON, INPUT_PULLUP);
55    attachInterrupt(digitalPinToInterrupt(BOUTON), ISR_Bouton, INPUT_PULLUP);
56    Serial.println("GO");
57    runRTOS();
58  }
59
60  void loop(){ /* VIDE */ }
```

4.3 Les interruptions périodiques

```
1   /**
2    * RTOS et les timers
3    **/
4   #include <Arduino_FreeRTOS.h>
5   #include <timers.h>
6
7   #define S0 10
8   #define S1 11
9
10  TimerHandle_t xtimer0;
11  TimerHandle_t xtimer1;
12
13  void callback_S0() {
14    static int E = 0;
15    digitalWrite(S0, E);
16    E = 1 - E;
17  }
18
19
20  void callback_S1() {
21    static int E = 0;
22    digitalWrite(S1, E);
23    E = 1 - E;
24  }
25
26  void runRTOS() {
27    xtimer0 = xTimerCreate (
```

```
28          "Timer_S0", // nom du timer
29          100, // période en tics
30          pdTRUE, // auto reload
31          (void *) 0, // ID du timer
32          callback_S0 // fonction de callback
33    );
34    xtimer1 = xTimerCreate ("Timer_S1", 20, pdTRUE, (void *) 1, callback_S1);
35    xTimerStart(xtimer0, 0);
36    xTimerStart(xtimer1, 0);
37    vTaskStartScheduler();
38  }
39
40  void setup() {
41    Serial.begin(115200);
42    pinMode(S0, OUTPUT);
43    pinMode(S1, OUTPUT);
44    Serial.println("GO");
45    runRTOS();
46  }
47
48  void loop(){ /* VIDE */ }
```

Annexe A

Installations

A.1 Installations des logiciels de base

`arduino`

```
sudo apt install arduino
```

Dans le cas où la version des dépots officiels est trop ancienne :

```
sudo snap install arduino
```

En fonction de la source d'installation, l'installation manuelle des librairies peut différer.

Encore mieux : récupérez sur le site Arduino le fichier AppImage qui est un exécutable totalement autonome et à jour. Il y n'y a donc pas (ou peu) de dépendances.

Avec Linux, pour utiliser la liaison série, il convient d'inscrire l'utilisateur dans les groupes `dialout` et `tty` :

```
sudo usermod -a -G tty utilisateur
sudo usermod -a -G dialout utilisateur
sudo reboot
```

D'une manière générale, en mode «développeur», nous devons appartenir aux groupes *tty*, *dialout* et *plugdev*.

L'environnement de base est installé. Il suffit de lancer la commande `arduino` pour démarrer, sauf si vous avez téléchargé le document *AppImage*

A.2 Noyau temps réel « `Freertos` »

Dans l'onglet «Librairies» recherchez et installez la librairie *FreeRTOS*.

Pour réaliser certains exercices, vous pouvez en profiter pour installer également les librairies *TimerOne*, *TimerThree* et *TimerFour*.

Annexe B

Carte d'extension pour les TP

B.1 Description

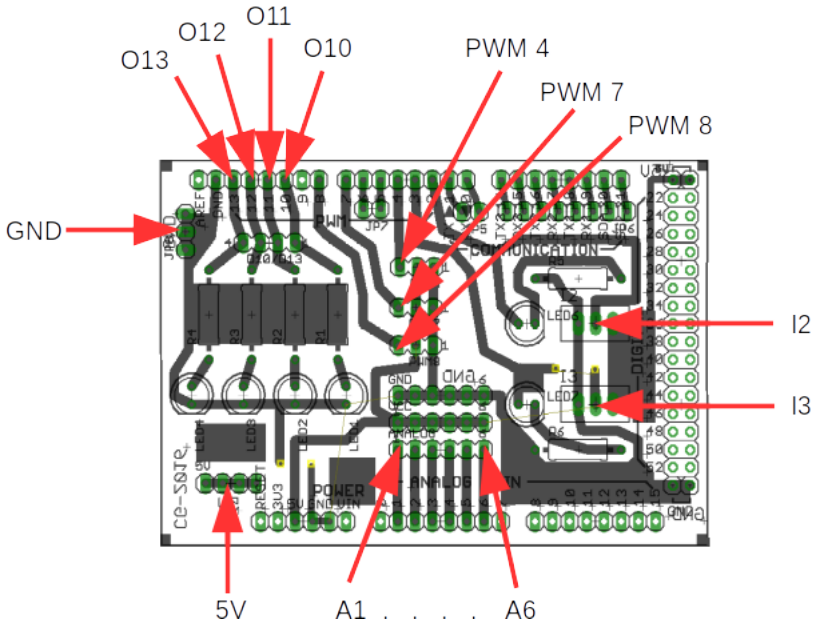

FIGURE B.1 – *Carte d'extension.*

Cette carte dispose
- des sorties digitales 10, 11, 12, 13
- des entrées analogiques A1 à A6
- de trois sorties PWM (4, 7 et 8)
- de 2 entrées digitales (2 et 3).
- de nombreuses sources Vcc et Gnd.

Nom	E/S arduino	Type
S0	10	sortie numérique
S1	11	sortie numérique
S2	12	sortie numérique
S3	13	sortie numérique
E0	2	entrée numérique
E1	3	entrée numérique
A1	1	sortie analogique
A2	2	sortie analogique
A3	3	sortie analogique
A4	4	sortie analogique
A5	5	sortie analogique
A6	6	sortie analogique
M0	2	sortie PWM
M1	3	sortie PWM
M2	3	sortie PWM

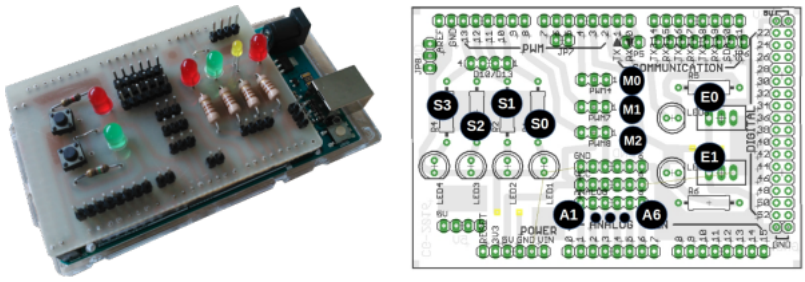

FIGURE B.2 – *La carte Arduino et son extension : 4 sorties digitales s_0 à S_4* équipées de LED, 3 sorties PWM M_0 .. M_3 pour piloter des servo-moteurs, 2 entrées E_0 et E_1 et enfin, 6 entrées analogiques à usage général A_1 à A_6

B.1.1 Programme de test pré-chargé

```
const int S3 =  13;
const int S2 =  12;
const int S1 =  11;
const int S0 =  10;

const int E0 = 2;
const int E1 = 3;


int T = 1000; // 1s

void setup() {
 pinMode(S0, OUTPUT);
 pinMode(S1, OUTPUT);
 pinMode(S2, OUTPUT);
 pinMode(S3, OUTPUT);

 pinMode(E0,INPUT);
 pinMode(E1,INPUT);
}


void testeTouche() {
  if (digitalRead(E0) == 1) {
    T += 100;
```

```
    while ( digitalRead(E0) == 1 ) delay(10);
  }
  if (digitalRead(E1) == 1) {
    T -= 100;
    if (T<100) T=100;
    while ( digitalRead(E1) == 1 ) delay(10);
  }
}

void loop() {
 digitalWrite(S3, 0); digitalWrite(S0, 1); testeTouche(); delay(T);
 digitalWrite(S0, 0); digitalWrite(S1, 1); testeTouche(); delay(T);
 digitalWrite(S1, 0); digitalWrite(S2, 1); testeTouche(); delay(T);
 digitalWrite(S2, 0); digitalWrite(S3, 1); testeTouche(); delay(T);
}
```

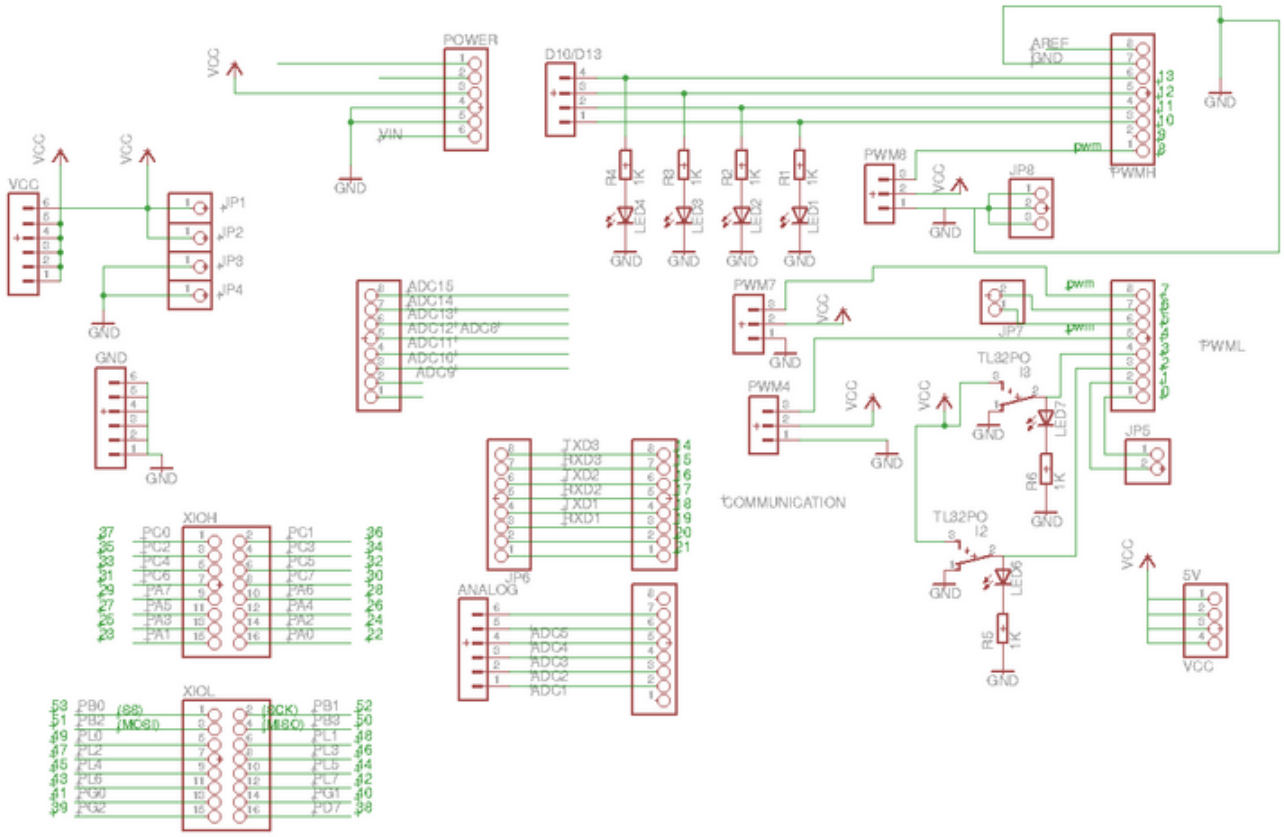

FIGURE B.3 – *Carte d'extension : schéma électrique.*

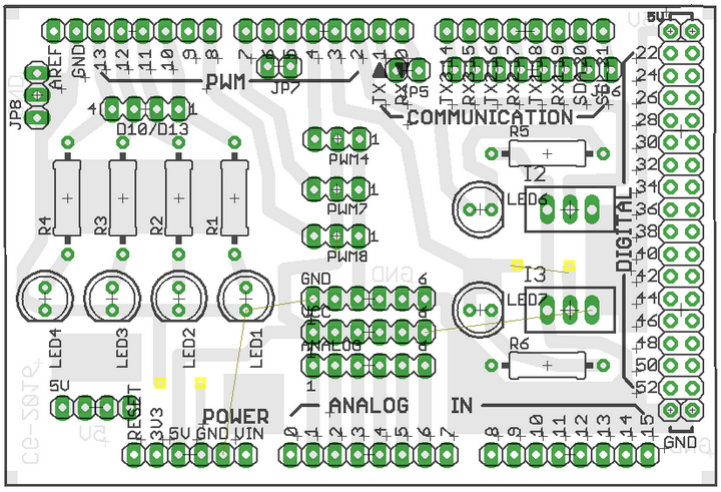

FIGURE B.4 – *Carte d'extension : implantation.*

Annexe C

Pilotage de servomoteurs

Pilotage : il faut générer un signal MLI (Modulation de Largeur d'Impulsion, PWM Pulse Width Modulation en anglais) de période 20 ms (= 50 Hz)

FIGURE C.1 – *Servo moteur connecté avec 3 fils : GND, Vcc et signal de commande PWM*

Le principe est représenté par la figure C.2. Attention, les valeurs peuvent varier en fonction du servomoteur.

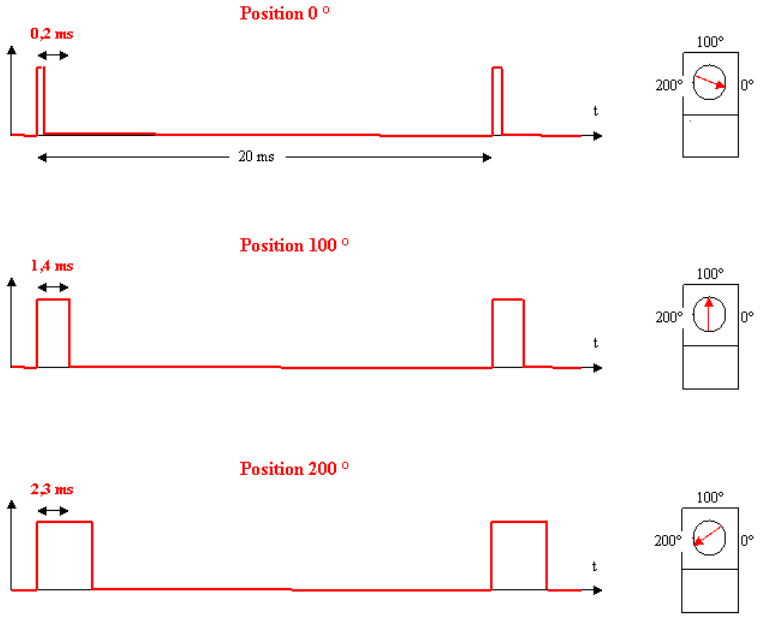

FIGURE C.2 – *Carte d'extension : implantation.*

C.1 Contrôle de la position avec Arduino

Exemple de mise en œuvre avec un moteur connecté sur la sortie 4 (M0).

```
1  #include <Servo.h>
2
3  Servo M0;
4
5  void setup() {
6     M0.attach(4);
7  }
8
9  void loop() {
10    M0.write(0);
11    delay(1000);
12    M0.write(180);
13    delay(1000);
14 }
```

Le signal qui commande le servomoteur est piloté par la librairie *servo.h*. Il n'y a rien d'autre à faire.

Annexe D

Commande de moteur pas à pas

Le moteur pas à pas unipolaire est constitué de bobines disposant d'un point milieu qui permettent d'alimenter un axe magnétique soit dans un sens soit dans un autre en alimentant seulement une demi bobine. La figure D.1 montre un exemple courant de moteurs unipolaire 6 fils. Le plus souvent, les deux communs sont reliés à l'intérieur, si ce n'est pas le cas, on les relient à l'extérieur et on obtient un moteur 5 fils.

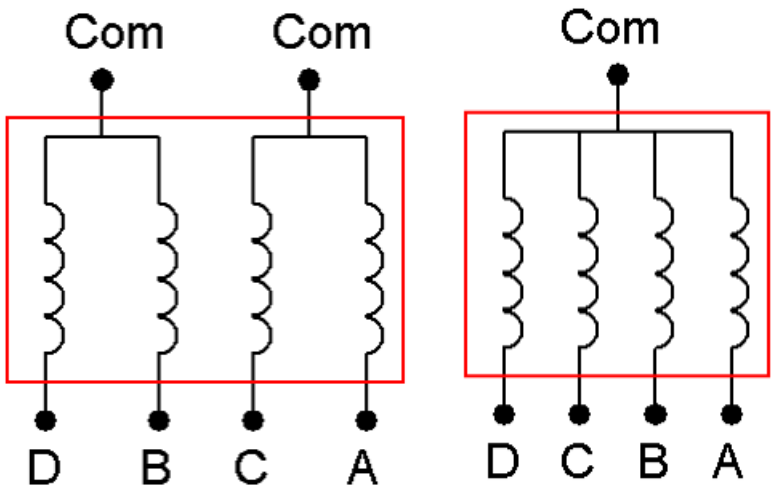

FIGURE D.1 – *Connexion interne des bobines.*

La commande de ces moteurs consiste à connecter le commun au + de l'alimentation et de mettre les phases successivement à 1 ou 0 selon la séquence désirée. Il existe plusieurs circuits intégrés permettant de réaliser cette opération. L'UNL2003, très répandu est adapté pour cela. C'est souvent le composant qui sert de base pour les cartes de puissance (Fig. D.2.

Séquence de contrôle des phases. Pour comprendre le fonctionnement, on peut partir d'un schéma simplifié, comme par exemple celui de la figure D.3. On considère que chaque phase (A,B,C,D) du moteur représente unee position. Une phase alimentée, se comporte comme un aimant, qui va attirer le rotor (représenté par une aiguille tournante).

Par exemple sur la figure D.3, seule la bobine A est alimentée. Du coup, le rotor est attiré vers la position 1.

Séquencements de la commande des 4 phases A B C D

On en distingue 4 :

1. **Fonctionnement en pas entier :** (Fig. D.4) on alimente les bobines A,B,C,D successivement, une bobine à la fois. Le rotor predn donc les positions successives 1,3,5,7.

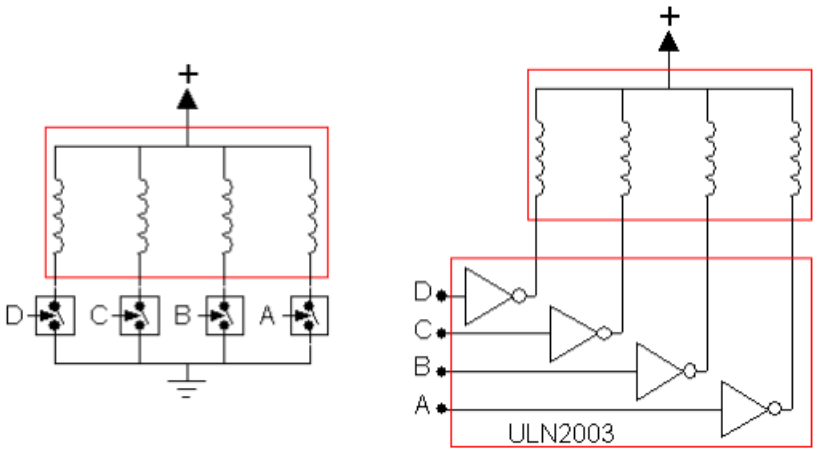

FIGURE D.2 – *Commande de l'alimentation électrique des bobines.*

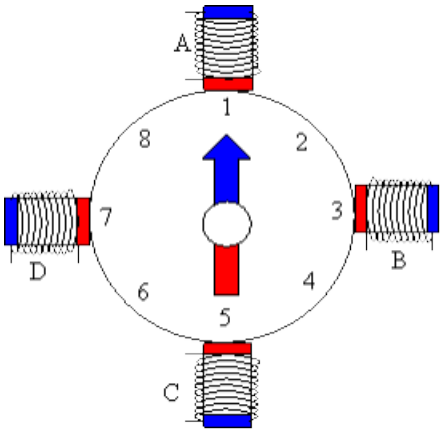

FIGURE D.3 – *Organisation physique simplifiée du moteur.*

2. **Fonctionnement avec deux phases alimentées simultanément.** (Fig. D.5) On alimente successivement AB, BC, CD, DA. Avec deux bobines alimentées simultanément, le champ magnétique est plus fort. On augmente donc le couple. Ceci se paie par une consommation supérieure. Les positions occupées sont cette fois : 2,4,6,8. Il s'agit aussi d'une commande en pas entier.
3. **Fonctionnement en $\frac{1}{2}$ pas.** (Fig. D.6) On alterne les 2 solutions précédentes pour que le moteur occupe successivement toutes les positions 1, ..., 8.

Exemple de mise en œuvre avec la carte d'extension

Le moteurs pas à pas est connecté aux sorties analogiques A3, A4, A5 et A6.

```
1  const byte TP[] = {A3,A4,A5,A6};
2  void setup() {
3    pinMode(TP[0],OUTPUT); pinMode(TP[1],OUTPUT);
4    pinMode(TP[2],OUTPUT); pinMode(TP[3],OUTPUT);
5  }
6
7  byte Tpas[][4] = {{0,0,0,1},
8                    {0,0,1,0},
9                    {0,1,0,0},
10                   {1,0,0,0}};
```

	BOBINES			
POSITION	D	C	B	A
1	0	0	0	1
3	0	0	1	0
5	0	1	0	0
7	1	0	0	0

FIGURE D.4 – *Commande en alimentant les bobines une par une.*

	BOBINES			
POSITION	D	C	B	A
2	0	0	1	1
4	0	1	1	0
6	1	1	0	0
8	1	0	0	1

FIGURE D.5 – *Deux bobines sont alimentées à la fois : on augmente le couple.*

```
11  int ipas = 0;
12  void loop() {
13    int i;
14    for (i=0; i<4; i++) digitalWrite(TP[i], Tpas[ipas][i]);
15    delay(20);
16    ipas = (++ipas)%4;
17  }
```

	BOBINES			
POSITION	D	C	B	A
1	0	0	0	1
2	0	0	1	1
3	0	0	1	0
4	0	1	1	0
5	0	1	0	0
6	1	1	0	0
7	1	0	0	0
8	1	0	0	1

FIGURE D.6 – *Fonctionnement en demi pas : on augmente la résolution.*