

Programmation

Arduino : approche «temps-réel»

CLAUDE GUÉGANNO

17 septembre 2017

Table des matières

1	La carte Arduino	3
1.1	Caractéristiques	3
1.2	Le micro-contrôleur ATmega256	4
2	Programmation «classique» de l'Arduino	9
2.1	Installation de base	9
2.2	Programmation avec «ArduBlock»	9
2.3	Programmation «Scratch»	9
2.4	Programmation en C/C++	10
2.5	Programmation en python	11
3	nilRTOS	14
3.1	Objectifs du multitâche	14
3.2	Les processus	14
3.2.1	Gestion des descripteurs de processus	16
3.3	Mise en œuvre avec nilRTOS	16
3.3.1	Contexte sauvegardé	16
3.3.2	États d'une tâche	17
3.4	Principes et création de <i>threads</i>	17
3.4.1	L'ordonnanceur de tâches	21
3.5	Les sémaphores	21
3.5.1	Exclusion mutuelle et synchronisation	21
3.5.2	Les sémaphores	22
3.5.3	Mise en œuvre de l'exclusion mutuelle avec nilRTOS	23
3.5.4	Mise en œuvre de la synchronisation avec nilRTOS	27
3.6	Interruptions matérielles	28
4	«freeRTOS»	31
4.1	Présentation	31
4.1.1	Caractéristiques et fonctionnalités	31
4.1.2	Un système <i>open source</i>	32
4.1.3	La popularité de FreeRTOS	32
4.2	Les tâches	32
4.2.1	Les différences entre les tâches et les co-routines	33
4.2.2	Les tâches	33
4.2.3	Les co-routines	35
4.3	L'ordonnancement	36

4.3.1	Un système multitâche	36
4.3.2	L'ordonnanceur temps réel de FreeRTOS	36
4.3.3	Les commutations de contexte	38
4.4	Exemples de mise en œuvre	40
4.4.1	Exemple avec les co-routines	40
4.4.2	Exemple avec les tâches	42
4.4.3	Les «mutex»	46
A	Installations	50
A.1	Installations des logiciels de base	50
A.2	Noyau temps réel « Ni1RTOS »	51
B	Plans de la «mega2560»	52
C	Mise en œuvre	54
D	Carte d'extension pour les TP	57
D.0.1	Programme de test pré-chargé	57
E	Librairie python pour arduino	60
F	Firmware pour arduino : utilisation avec <i>pyduino</i>	65

Chapitre 1

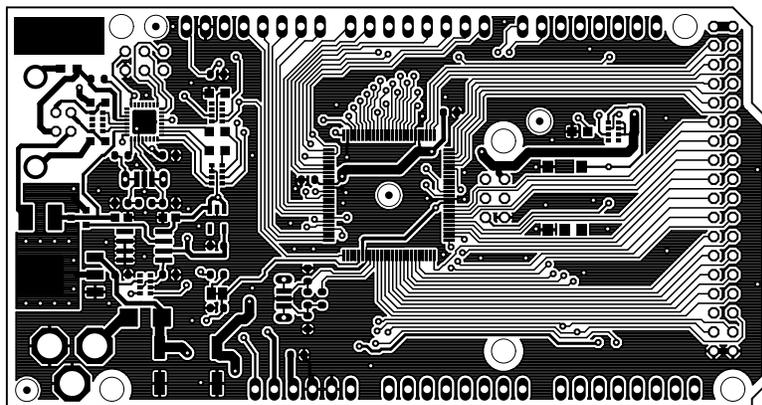
La carte Arduino

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 microcontrôleur 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++. Il est possible également de programmer par blocs (approche graphique).

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.



Arduino Mega 2560 Reference Design

Reference Designs ARE PROVIDED "AS IS" AND "WITH ALL FAULTS. Arduino DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, REGARDING PRODUCTS, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Arduino may make changes to specifications and product descriptions at any time, without notice. The Customer must not rely on the absence or characteristics of any features or instructions marked "Reserved" or "Undefined." Arduino reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The product information on the Web Site or Materials is subject to change without notice. Do not finalize a design with this information.

1.1 Caractéristiques

La carte Arduino Mega 2560 est une carte à microcontrôleur 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 ICSP¹,
- d'un bouton de réinitialisation (reset);
- de 256KB de mémoire programme flash (8KB sont pris par le *bootloader*);
- de 8KB de mémoire SRAM²;
- de 4KB d'EEPROM³.

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.

L'annexe B (page 52) donne les plans de câblage de la Mega2560.

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-récepteur radio permet des échanges de données de 250KB/s à 2MB/s.

1. La programmation in-situ (In-System Programming ou ISP) est une fonctionnalité qui permet aux composants électroniques (microcontrôleurs 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 programmeur 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 en électricité.

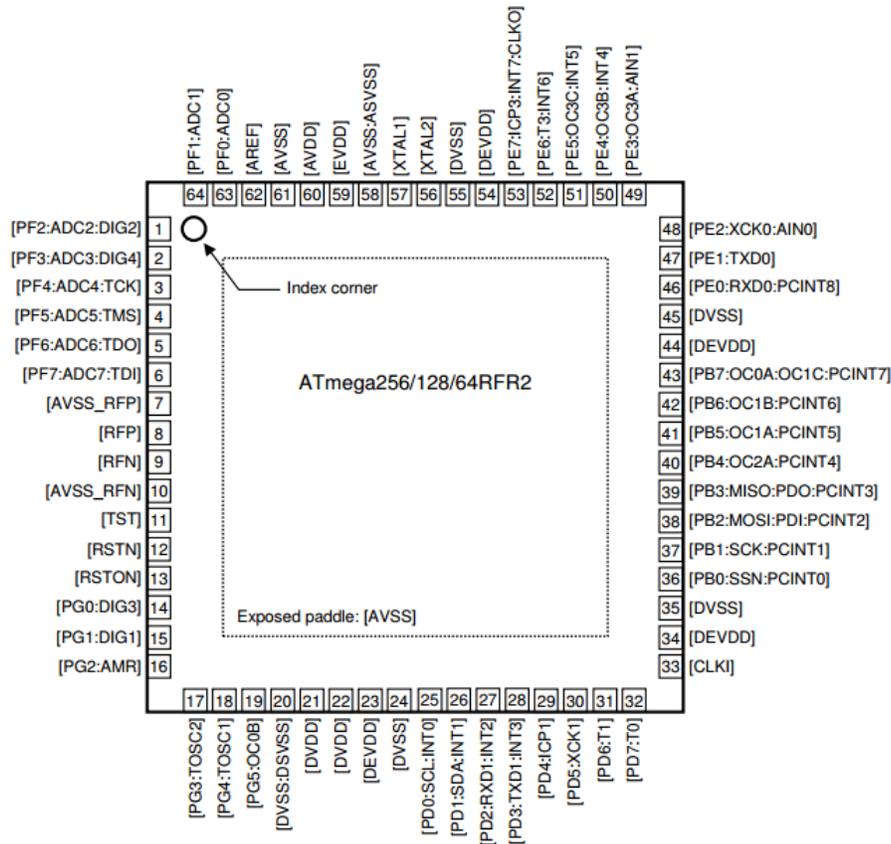


FIGURE 1.1 – Brochage de l'ATMEGA256

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.

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 SPI⁴; un port JTAG.

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.

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.

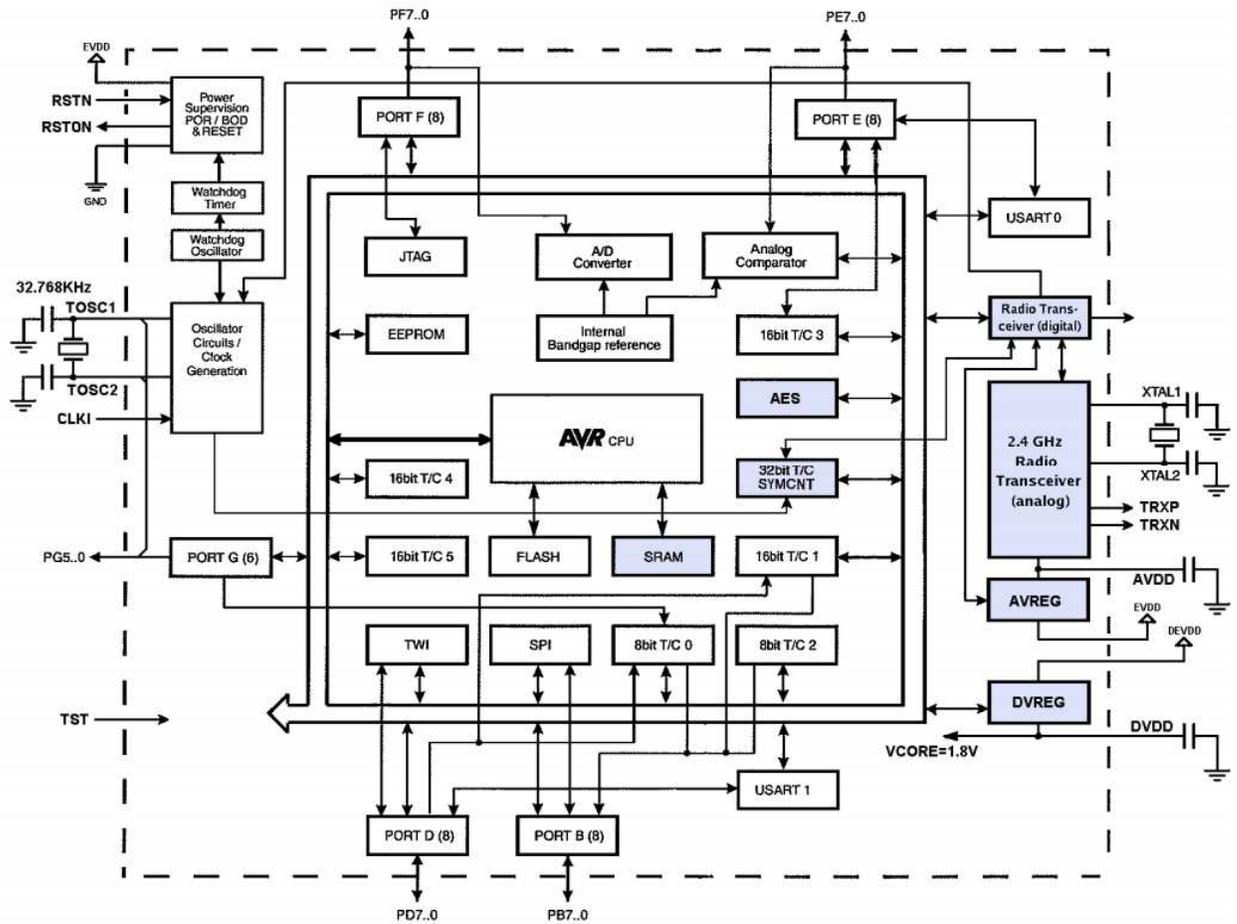


FIGURE 1.2 – Diagramme des blocs de l'ATMEGA

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.

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.

Données. Les adresses de 0000 à 001F servent aux registres d'utilisation générale, alors que les adresses 0020 à 005F 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 microcontrôleur. Enfin, jusqu'à FFFF nous avons l'espace utilisable pour ajouter au microcontrôleur de la mémoire SRAM externe.

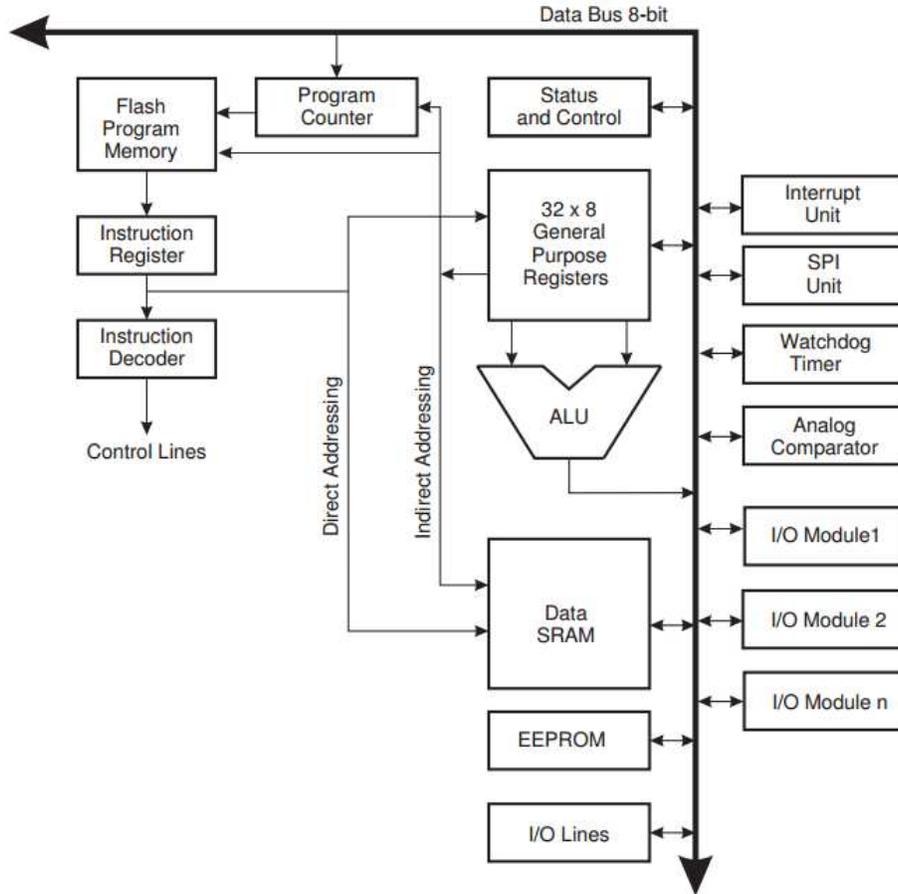


FIGURE 1.3 – Diagramme des blocs de l'architecture AVR

	7	0	Addr.	
General Purpose Working Registers	R0		0x00	
	R1		0x01	
	R2		0x02	
	...			
	R13		0x0D	
	R14		0x0E	
	R15		0x0F	
	R16		0x10	
	R17		0x11	
	...			
	R26		0x1A	X-register Low Byte
	R27		0x1B	X-register High Byte
	R28		0x1C	Y-register Low Byte
	R29		0x1D	Y-register High Byte
	R30		0x1E	Z-register Low Byte
	R31		0x1F	Z-register High Byte

FIGURE 1.4 – Les 32 registres de l'AVR.

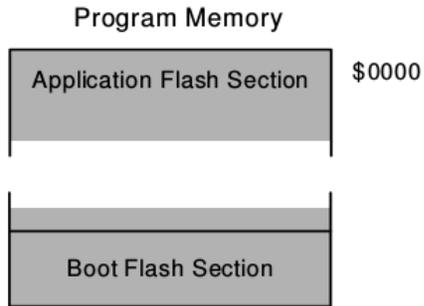


FIGURE 1.5 – La mémoire programme de l'AVR.

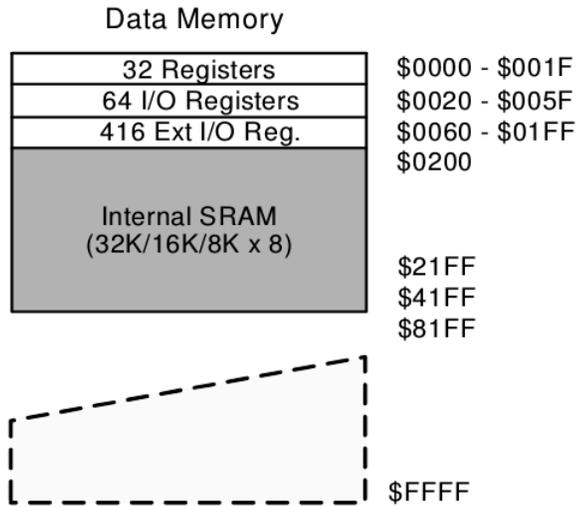


FIGURE 1.6 – La mémoire des données l'AVR.

Chapitre 2

Programmation «classique» de l'Arduino

Dans ce chapitre, nous évoquons rapidement le style de programmation basique pour la programmation de la carte, sans noyau ni librairie additionnelle.

2.1 Installation de base

La programmation des Arduino se fait directement avec un IDE dédié (`arduino`). Des développements en compilation croisée sont possibles, mais pour ce matériel, ils n'apporteront rien de plus.

L'installation de l'IDE est décrit en annexe A.1.

Cet environnement est suffisant pour développer efficacement. Cependant, voici dans les 2 sections suivantes, deux possibilités qui rendent la programmation de la carte extrêmement accessible, de manière graphique (programmation par bloc).

2.2 Programmation avec «ArduBlock»

ArduBlock est un environnement de programmation par blocs, qui évite donc les problèmes usuels de syntaxe, d'écriture de code ... Il permet de visualiser le programme graphiquement, comme *Scratch*.

L'installation de *ArduBlock* est décrite dans l'annexe A.1. Les programmes réalisés avec *ArduBlock* sont compilés sur la machine hôte, puis chargés dans la carte arduino. Il s'agit d'une compilation croisée.

La prise en compte des broches de l'Arduino est complète, y compris pour le pilotage des servo moteurs par PWM, des capteurs à ultra son, des périphériques i2c courants, des périphériques Adafruit, Grove, Tinker, Keenlon, LittleBits, DFRobot ...

2.3 Programmation «Scratch»

Le fameux langage du MIT est une possibilité intéressante pour la programmation de l'arduino. Le programme chargé dans l'arduino est un serveur d'E/S qui dialogue avec

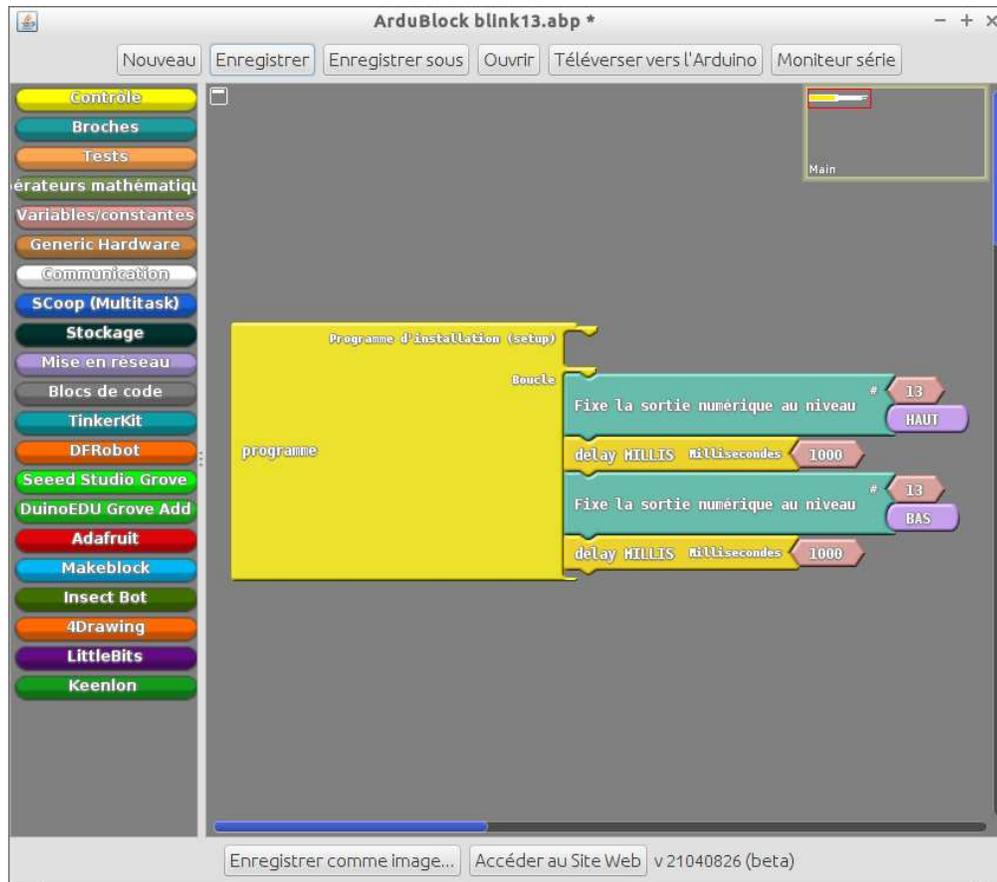


FIGURE 2.1 – *Programmation avec ArduBlock*

l'interface *scratch*. À la différence d'ArduBlock, ce n'est pas une compilation croisée, mais un dialogue entre *scratch* et le *firmware* de la carte arduino.

Une image de *scratch* appelée «S4A» est disponible sur <http://s4a.cat/> : elle permet de programmer directement la carte arduino. Un *firmware* est également disponible à partir de ce site.

2.4 Programmation en C/C++

C'est la manière la plus courante de programmer la carte. Le logiciel est divisé en deux fonctions : le `setup` dans lequel on place toute l'initialisation du système, et la fonction `loop` qui est la boucle naturelle de l'application.

```
int boutonPin = 3;

// la fonction setup initialise la communication série
// et une broche utilisée avec un bouton poussoir

void setup() {
  Serial.begin(9600);
  pinMode(boutonPin, INPUT);
}
```

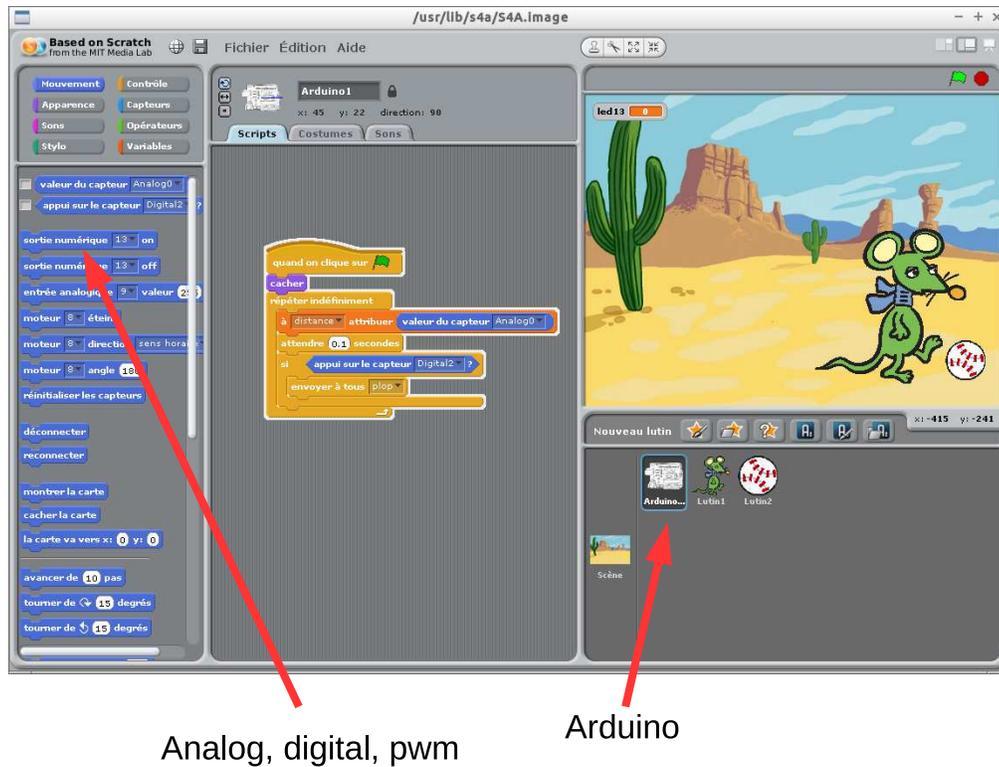


FIGURE 2.2 – Programmation avec Scratch for Arduino (S4A)

```
// la fonction loop teste l'état du bouton à chaque passage
// et envoie au PC une lettre H si il est appuyé, L sinon.
```

```
void loop() {
  if (digitalRead(buttonPin) == HIGH)
    Serial.write('H');
  else
    Serial.write('L');

  delay(1000);
}
```

Même si on n'utilise pas la fonction loop, elle doit être présente, car elle est référencée dans le programme principal du BIOS de la carte.

2.5 Programmation en python

Comme pour «Scratch», il ne s'agit pas d'une compilation croisée, mais d'une application résidente dans la carte arduino qui est en communication avec les instructions python.

Cette stratégie permet de tester rapidement une application, et/ou de fournir à la machine hôte une interface simple vers l'environnement extérieur.

Pour mettre en œuvre la programmation avec `python`, il faut au préalable :

1. charger le *firmware* adéquat dans la carte arduino : annexe F ;
2. placer la librairie `pyduino.py` dans votre répertoire de travail : annexe E ;

Ces deux opérations ne seront plus à refaire, et il est possible à partir de là d'accéder très simplement à la carte arduino, par des instructions `python` élémentaires.

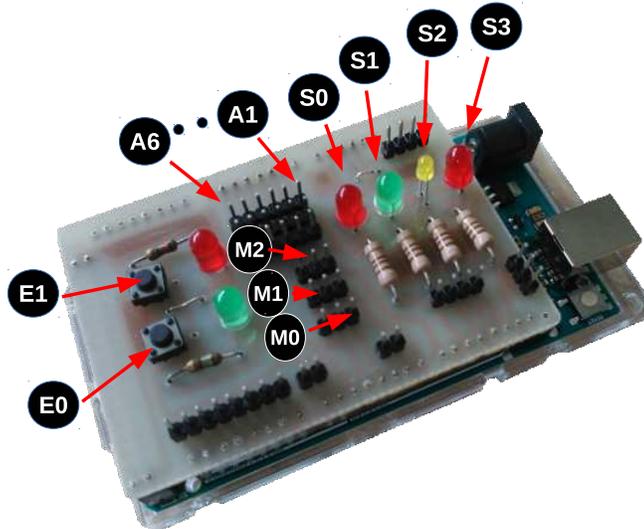


FIGURE 2.3 – Mise en œuvre de la carte arduino équipée de son extension, avec `python`

Exemple avec les sorties

```
from pyduino import *
import time

a = Arduino()
time.sleep(3)

i = 0
while True:
    if i%2 == 0:
        a.S0(1)
        a.S1(0)
    else:
        a.S0(0)
        a.S1(1)
    if i%3 == 0:
        a.S2(1)
        a.S3(0)
    else:
        a.S2(0)
        a.S3(1)
    i = i+1
```

Exemple avec les entrées

```
from pyduino import *
import time

a = Arduino()
time.sleep(3)
while True:
    v = a.E0() + 2*a.E1()
    print(' ' + repr(v))
```

Exemple avec les entrées analogiques

```
from pyduino import *
import time

a = Arduino()
time.sleep(3)

while True:
    print(' A1 = ' + repr(a.A(1)))
```

Exemple avec les sorties PWM

```
from pyduino import *
import time

a = Arduino()
time.sleep(3)

while True:
    a.M0(0)
    time.sleep(1)
    a.M0(180)
    time.sleep(1)
    a.M0(90)
    time.sleep(1)
```

Chapitre 3

ni1RTOS

«Nil RTOS» is so small ... it's almost nil!

3.1 Objectifs du multitâche

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 multitâche 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 compétition. La notion de priorité intervient également sur le partage du temps.

3.2 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.

- 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 ...).

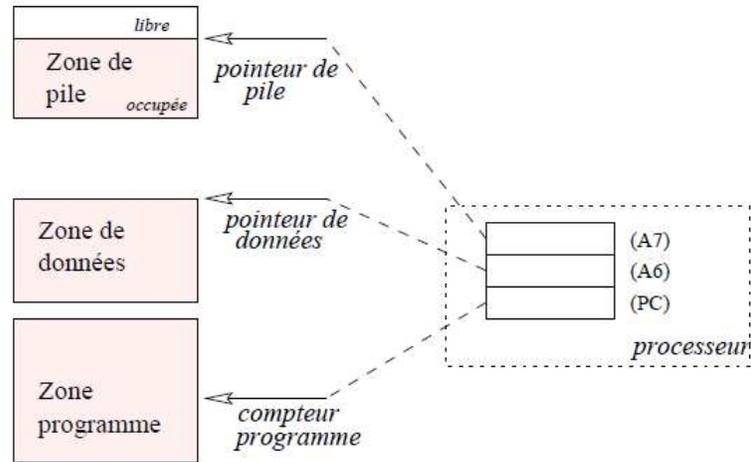


FIGURE 3.1 – Occupation de la mémoire par un processus.

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 du M68000. L'instruction en cours est marquée par le «compteur programme» du microprocesseur (PC^1). La pile est marquée par le registre USP^2 . La zone de donnée est marquée par un registre décidé par le système d'exploitation (Ici, c'est $A6$).

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 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**.

1. Program Counter
2. User Stack Pointer

3.2.1 Gestion des descripteurs de processus

Le système multitâche 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.

Dans la pratique, de nombreux exécutifs utilisent le mécanisme du double chaînage consistant à relier un à un et dans les deux sens tous les descripteurs de processus de façon à pouvoir intervenir rapidement sur un processus donné (Fig. 3.2).

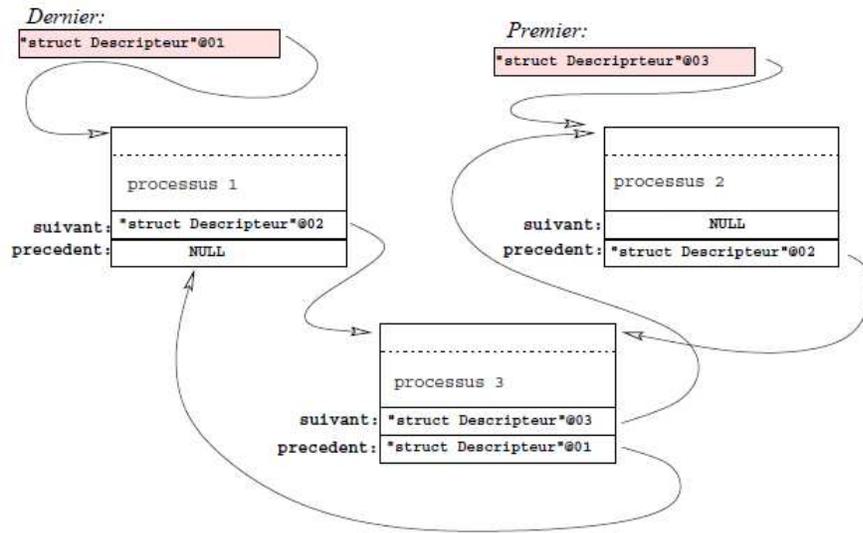


FIGURE 3.2 – Chaînage des descripteurs de processus.

3.3 Mise en œuvre avec niRTOS

3.3.1 Contexte sauvegardé

Le déroulement des processus temps-réel est dépendant des événements externes au calculateur. Ces événements n'étant pas sous le contrôle du système, plusieurs tâches peuvent se trouver en concurrence pour l'octroi du processeur. Ce conflit est réglé en attribuant à chacune d'entre elles un niveau de priorité. La tâche exécutée est choisie par un algorithme de choix : **l'ordonnanceur**.

Dans le cas de niRTOS, il n'est pas possible de créer des tâches dynamiquement. Le programmeur définit un tableau de processus, de manière statique. L'ordre de priorité est l'ordre dans lequel sont déclarés les processus. Cette solution donne moins de souplesse, mais permet d'optimiser la taille du noyau, ce qui le rend particulièrement adapté aux applications complexes fonctionnant avec peu de ressources.

L'ordonnanceur (*scheduler*) constitue le cœur du noyau temps-réel. Il est invoqué à chaque commutation de tâches ainsi qu'à chaque appel à une fonction système (sémaphore, signal, file d'attente ...)

Le principe de commutation de tâche conduit à sauver en RAM l'état de chaque tâche : c'est le **contexte sauvegardé**. On y trouve :

- le *thread* d'exécution de la tâche (c'est simplement le compteur-programme pointant sur l'adresse où est rendu le programme ;
- les registres du CPU, et éventuellement, ceux de l'unité de calcul en virgule flottante ;
- une pile mémorisant les variables dynamiques et les appels à des fonctions ;
- les canaux d'E/S standards associés ;
- un délai d'attente (compteur) ;
- un compteur de tranche de temps (*timeslice*) les

3.3.2 États d'une tâche

Le noyau met à jour l'état de chaque tâche en cours dans l'application. Une tâche change d'un état à un autre suivant le résultat de chaque appel fait à une fonction du noyau par une tâche quelconque de l'application.

Lorsqu'une tâche est créée, elle est dans l'état *suspendu*. Pour passer à l'état *prêt*, elle doit être activée. La phase d'activation est très rapide. Ceci permet de «pré-crérer» plusieurs tâches (\Rightarrow préparation des contextes), puis de les activer en un temps très court. Une autre possibilité est le lancement des tâches (*spawning*), qui permet d'enchaîner la création et l'activation de tâches.

3.4 Principes et création de *threads*

Un *thread* est défini par trois macros, `NIL_WORKING_AREA()`, `NIL_THREAD()`, et `NIL_THREADS_TABLE_ENTRY()`.

```
#include <NilRTOS.h>
#include <NilSerial.h>
#include <NilAnalog.h>
#define Serial NilSerial

const uint8_t LED_PIN = 13;

// THREAD 1 -----
NIL_WORKING_AREA(pile1, 32);
NIL_THREAD(Thread1, arg) {
    while (TRUE) {
        // Turn LED off.
        nilThdSleepMilliseconds(1000);
        digitalWrite(LED_PIN, LOW);
        Serial.write('L');

        nilThdSleepMilliseconds(1000);
    }
}
```

```

    digitalWrite(LED_PIN, HIGH);
    Serial.write('H');
}
}

// Table des processus
NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY("T1", Thread1, NULL, pile1, sizeof(pile1))
NIL_THREADS_TABLE_END()

void setup() {
    pinMode(LED_PIN, OUTPUT);
    nilSysBegin();
}

// Loop est un thread masqué
// pour nilRTOS, on le laisse vide
void loop() {
    // Non utilisé
}

```

Ex. 1

1. Lancez la commande `arduino` et assurez vous que le logiciel fonctionne correctement (détection de la carte, du port série USB (souvent appelé `/dev/ttyACM0`)).
2. Vérifiez l'existence des bonnes librairies dans `/usr/share/arduino/libraries/` (cf. annexe A.2).
3. Testez.
4. Quelle est, en nombre d'octets, la taille de ce programme?
.....
.....
5. Y a t-il un intérêt à utiliser un noyau temps réel dans ce cas?
.....
.....
.....

Ex. 2

Avec plusieurs *threads* : chacun disposant de sa zone de code et de ses données.

1. En vous inspirant de l'exemple précédent, créez un *thread* de clignotement pour chacune des 4 sorties câblées de la carte d'extension :
 - Sortie S0 : période de 1,5 secondes
 - Sortie S1 : période de 0,95 secondes
 - Sortie S2 : période de 0,49 secondes
 - Sortie S3 : période de 0,27 secondes

`NIL_WORKING_AREA(pile1, 32);` Dans cet exemple, un seul *thread* est créé. Sa zone de travail a pour nom `pile1`. C'est une pile de 32 octets seulement. Ces 32 octets sont comptés au delà de la quantité de mémoire strictement nécessaire pour le traitement de l'interruption et la commutation de tâche.

`NIL_THREAD(Thread1, arg)` Cette expression définit le point d'entrée du *thread*. La fonction `Thread1` est lancée avec l'argument `void *arg`.

`NIL_THREADS_TABLE_ENTRY("nom", Thread1, NULL, pile, sizeof(pile))`

Cette expression déclare la table des processus pour RTOS. Chaque processus est créé avec un nom, l'adresse du code à exécuter (une fonction), un pointeur sur un éventuel paramètre, l'adresse de la zone de donnée qui est allouée comme pile pour les variables locales et les paramètres passés aux fonctions, et enfin la taille de cette zone de données.

Zone de donnée et zone de code Pour illustrer cette notion voici un exemple où deux threads «T1» et «T2» sont créés. Ils utilisent tous les deux le même code (exactement les mêmes octets en mémoire). Ils sont simplement différenciés par leur zone de pile, et par le paramètre qu'ils reçoivent au moment de leur création.

```
#include <NilRTOS.h>
#include <NilSerial.h>
#define Serial NilSerial

struct Param {
    int Periode;
    char H;
    char L;
};

struct Param P1 = { 1000, 'H', 'L' };
struct Param P2 = { 1500, '1', '0' };

// THREAD 1 -----
NIL_WORKING_AREA(pile1, 32);
NIL_THREAD(Thread1, arg) {
    struct Param *p = (struct Param *)arg;
    while (TRUE) {
        nilThdSleepMilliseconds(p->Periode);
    }
}
```

```

    Serial.write(p->H);
    nilThdSleepMilliseconds (p->Periode);
    Serial.write(p->L);
}
}

// THREAD 2 -----
NIL_WORKING_AREA(pile2, 32);

// Table des processus
NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY("T1", Thread1, &P1, pile1, sizeof(pile1))
NIL_THREADS_TABLE_ENTRY("T2", Thread1, &P2, pile2, sizeof(pile2))
NIL_THREADS_TABLE_END()

//-----
void setup() {
    Serial.begin(9600);
    nilSysBegin();
}
//-----
void loop() {}

```

Observez bien que dans le code, il y a une seule fonction et deux exécutions lancées en parallèle dans la table des processus.

Ex. 3

1. Testez l'exemple.
2. Quelle est la taille de la mémoire allouée aux données des processus T_1 et T_2 ?
3. Quelle taille de mémoire est utilisée par les paramètres ? par les données locales ?
4. Ajoutez un troisième *thread* qui utilise le même code mais avec des paramètres et une pile différents.

Une utilisation possible de ce principe est l'échantillonnage de grandeurs physiques par des *threads* simplement différenciés par un numéro d'entrée analogique et une période d'échantillonnage.

Ex. 4

1. Proposez une structure de données générique, qui contient une période d'échantillonnage ainsi qu'un numéro d'entrée analogique.
2. Sur la carte d'extension, câblez deux capteurs (au choix), en notant les entrées sur lesquelles vous les avez câblé.
3. Créez la table des *threads* : données différentes, mais code partagé.
4. Écrivez le code des *threads* (une seule fois). Chaque *thread* vient lire son entrée, avec la période qui lui est propre, et affiche le résultat sur le port série.
5. Testez.

Remarque : il peut y avoir des incohérences dans l'affichage. Ce problème sera réglé ultérieurement.

3.4.1 L'ordonnanceur de tâches

La priorité d'un processus est déterminée en fonction de son emplacement dans cette table. Le premier déclaré est le plus prioritaire.

Le *thread* «inactif» : la fonction `loop()` est exécutée par RTOS uniquement lorsque tous les processus sont bloqués. La fonction `loop()` ne doit appeler aucune des primitives du noyau. En général on ne l'utilise même pas.

C'est un ordonnanceur préemptif à priorité fixe. Les *threads* de forte priorité doivent être bloqués pour autoriser les autres *threads* à s'exécuter (*sleep*, attente d'un sémaphore)

3.5 Les sémaphores

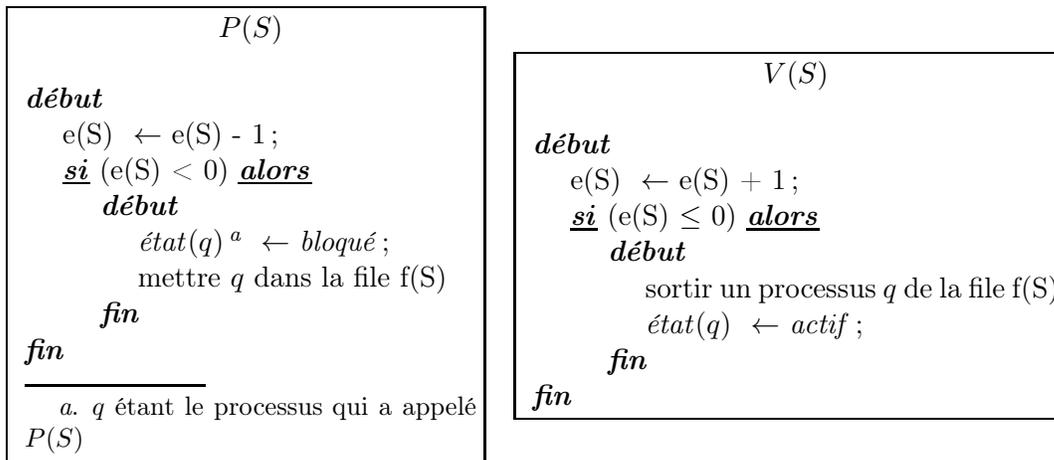
L'ensemble des processus doivent **coopérer** à la réalisation de l'application. Cette coopération recouvre deux aspects souvent mêlés :

- les coordinations ou **synchronisations** des processus entre eux,
- les **communications** de valeurs entre processus.

3.5.1 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.



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.

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 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 3.4 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 $f(S)$.

3.5.3 Mise en œuvre de l'exclusion mutuelle avec niRTOS

```

#include <NilRTOS.h>
#include <NilSerial.h>

```

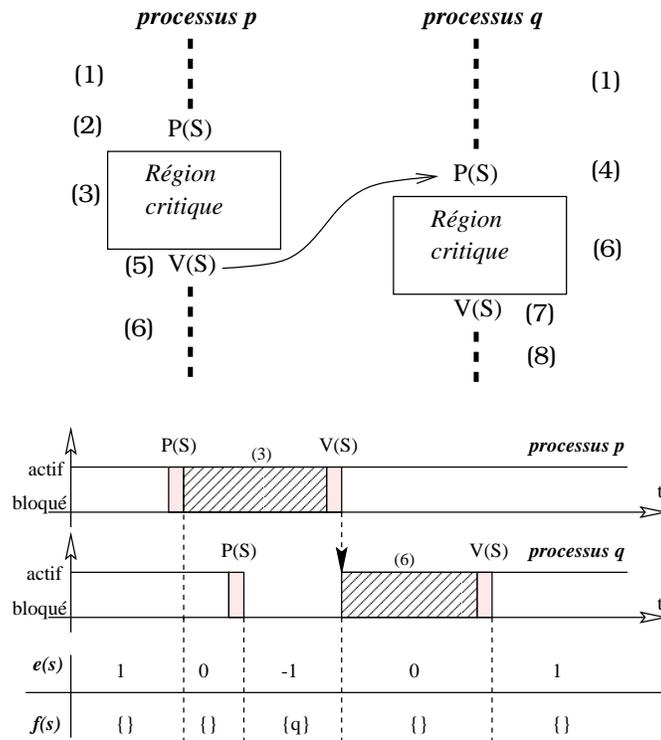


FIGURE 3.4 – Protection d’une région critique par un sémaphore.

```

#define Serial NilSerial

const uint8_t LED = 13;

SEMAPHORE_DECL(Sem, 0); // déclaration du sémaphore

void unDeuxTrois() {
    nilThdSleepMilliseconds(5);
    Serial.write('1');
    nilThdSleepMilliseconds(5);
    Serial.write('2');
    nilThdSleepMilliseconds(5);
    Serial.write('3');
}

// THREAD 1 -----
NIL_WORKING_AREA(pile1, 32);
NIL_THREAD(tache1, arg) {
    while (TRUE) {
        nilThdSleepMilliseconds(50);
        unDeuxTrois();
    }
}

// THREAD 2 -----

```

```

NIL_WORKING_AREA(pile2, 32);
NIL_THREAD(tache2, arg) {
    uint8_t X = 1;
    while (TRUE) {
        nilThdSleepMilliseconds(70);
        unDeuxTrois();
    }
}

NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY("T1", tache1, NULL, pile1, sizeof(pile1))
NIL_THREADS_TABLE_ENTRY("T2", tache2, NULL, pile2, sizeof(pile2))
NIL_THREADS_TABLE_END()

//-----
void setup() {
    pinMode(LED, OUTPUT);
    Serial.begin(9600);
    nilSysBegin();
}

void loop() {}

```

La trace d'exécution est donnée dans la figure 3.5. Une lecture attentive montre bien que l'affichage n'est pas cohérent et qu'il y a un mélange des messages.

Dans cet exemple, les délais choisis permettent de mettre le problème en évidence assez rapidement. Avec un rapport

$$\frac{\text{temps d'exécution}}{\text{période d'exécution}}$$

très faible, on pourrait fort bien attendre des heures avant de provoquer ce problème. Il convient donc de bien identifier la «**région critique**», c'est à dire la séquence d'instruction qui ne peut pas être exécutée simultanément par deux tâches.

Pour des questions d'efficacité, il convient de minimiser le nombre d'instructions de la région critique.

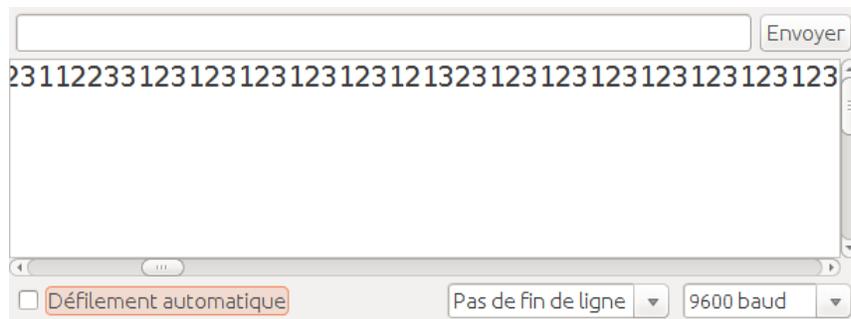


FIGURE 3.5 – *Deux tâches sans exclusion mutuelle.*

Dans notre cas, la région critique est à l'intérieur de la fonction `unDeuxTrois()`, et ce sont ces lignes de code qu'il faut protéger par un sémaphore :

```

void unDeuxTrois() {

```

```

nilThdSleepMilliseconds(5);

// P(S) /* Début de la région critique */
Serial.write('1');
nilThdSleepMilliseconds(5);
Serial.write('2');
nilThdSleepMilliseconds(5);
Serial.write('3');
// V(S) /* Fin de la région critique */

}

```

Dans la pratique, avec `nilRTOS`, il suffit de déclarer un sémaphore à 1 place en début de programme :

```
SEMAPHORE_DECL(mutex, 1);
```

puis de modifier la fonction `unDeuxTrois()` de la manière suivante :

```

void unDeuxTrois() {
    nilThdSleepMilliseconds(5); // il n'y a aucun intérêt à inclure cette ligne

    nilSemWait(&mutex); // début de l'exclusion
    Serial.write('1');
    nilThdSleepMilliseconds(5);
    Serial.write('2');
    nilThdSleepMilliseconds(5);
    Serial.write('3');
    nilSemSignal(&mutex); // fin de l'exclusion
}

```

Ex. 5

1. Reprenez cet exemple en considérant cette fois trois tâches accédant à la ressource critique.

Utilisation du sémaphore pour gérer une synchronisation

À chaque 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	{ }

La figure 3.6 (a) représentent cette même situation sous forme de chronogramme. Le deuxième cas (b) illustre la situation la plus favorable (la donnée est prête).

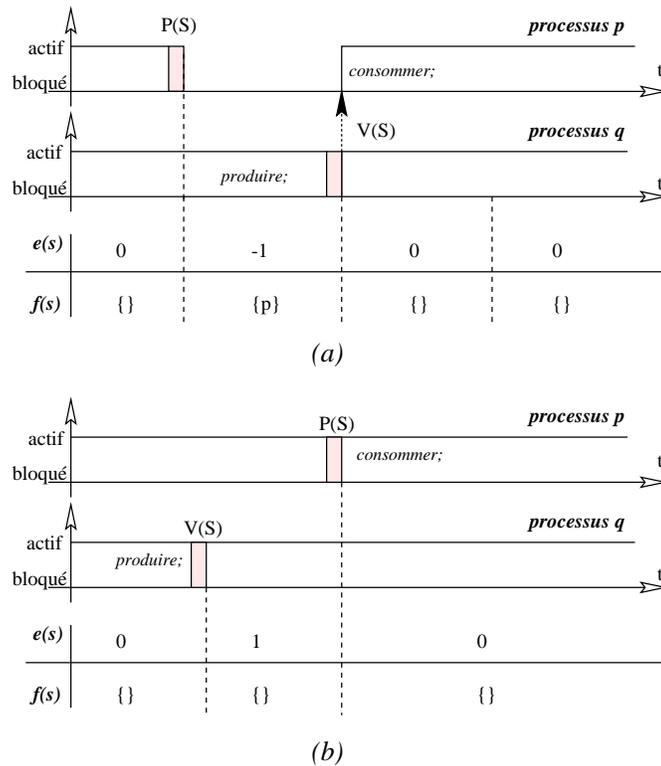


FIGURE 3.6 – Synchronisation dans le cas défavorable (a) et dans le cas favorable (b).

3.5.4 Mise en œuvre de la synchronisation avec nilRTOS

C'est un mécanisme qui permet à une tâche donnée de signaler à une autre tâche qu'un événement vient de se produire.

```
#include <NilRTOS.h>
#include <NilSerial.h>
#define Serial NilSerial

const uint8_t LED = 13;

SEMAPHORE_DECL(Sem, 0); // déclaration du sémaphore

// THREAD 1 -----
NIL_WORKING_AREA(pile1, 32);
NIL_THREAD(tache1, arg) {
    while (TRUE) {
        nilThdSleepMilliseconds(500);
        Serial.write('H');
        nilSemSignal(&Sem);
        nilThdSleepMilliseconds(100);
        Serial.write('L');
    }
}
```

```

    nilSemSignal(&Sem);
}
}

// THREAD 2 -----
NIL_WORKING_AREA(pile2, 32);
NIL_THREAD(tache2, arg) {
    uint8_t X = 1;
    while (TRUE) {
        nilSemWait(&Sem);
        digitalWrite(LED, X);
        X = 1 - X;
    }
}

NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY("T1", tache1, NULL, pile1, sizeof(pile1))
NIL_THREADS_TABLE_ENTRY("T2", tache2, NULL, pile2, sizeof(pile2))
NIL_THREADS_TABLE_END()

//-----
void setup() {
    pinMode(LED, OUTPUT);
    Serial.begin(9600);
    nilSysBegin();
}

void loop() {}

```

Dans cet exemple la tâche 1 est périodique et envoie des signaux. La tâche 2 attend le signal avant d'exécuter son action, puis se remet en attente.

Souvent la tâche émettrice est une tâche très prioritaire. Elle notifie des tâches moins prioritaires par ce mécanisme. L'événement est alors traité en «différé», ce qui permet à la tâche principale d'être à nouveau disponible immédiatement.

Ex. 6

1. Créez une application `nilRTOS` composée de deux *threads* :
 - T_1 réalise des mesures périodiques sur un capteur (au choix). Lorsque la mesure dépasse 80% de la valeur maximale, un signal est envoyé au *thread* T_2 .
 - T_2 est en attente d'un signal de synchronisation. Lorsqu'il reçoit le signal, il envoie un message d'erreur sur le port `Serial`, puis se remet en attente d'un nouveau signal. mù !gyg

3.6 Interruptions matérielles

Elle permettent de déclencher une séquence de code lors d'un signal électrique sur l'une des broches d'entrée de l'AVR. On appelle cette séquence de code un «programme d'interruption» ou une «tâche immédiate». Bien évidemment, un tel programme doit être très court et ne pas contenir de boucle infinie ...

Une pratique courante, pour les traitements conséquents est d'utiliser la routine d'interruption pour synchroniser une tâche normale (différée) qui est en attente passive de l'événement.

Dans l'exemple suivant, un front descendant sur la broche 2 débloque une tâche différée en attente d'un signal. C'est bien sûr le programme d'interruption qui envoie le signal.

```
#include <NilRTOS.h>
#include <NilSerial.h>
#define Serial NilSerial

const uint8_t LED = 13;

// interruption
const uint8_t INTERRUPT_PIN = 2;
SEMAPHORE_DECL(isrSem, 0);

// La routine d'interruption:
NIL_IRQ_HANDLER(isrFcn) {
    NIL_IRQ_PROLOGUE(); // sauvegarde du contexte
    nilSemSignalI(&isrSem);
    NIL_IRQ_EPILOGUE(); // restitution du contexte
}

// La tâche synchronisée par l'interruption:
NIL_WORKING_AREA(pile, 64);
NIL_THREAD(tache, arg) {
    uint8_t X = 0;
    attachInterrupt(0, isrFcn, FALLING);
    while (1) {
        nilSemWait(&isrSem);
        Serial.println("<<IT>>");
        X = 1 - X;
        digitalWrite(LED, X);
        nilThdSleepMilliseconds(300);
        nilSemReset(&isrSem, 0); // évite les rebonds
    }
}

NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY("T", tache, NULL, pile, sizeof(pile))
NIL_THREADS_TABLE_END()

void setup() {
    pinMode(INTERRUPT_PIN, INPUT);
    pinMode(LED, OUTPUT);
    Serial.begin(9600);
    nilSysBegin();
}

void loop() {}
```

Dans cet exemple, c'est la tâche en attente de synchronisation par sémaphore qui installe la

routine d'interruption, avant d'entrer dans sa boucle.

On remarque que dans ce cas, la seule action exécutée par la routine d'interruption est d'envoyer un signal.

Ex. 7

1. Créez une application `ni1RTOS` composée de deux *threads* :
 - T_1 réalise un chenillard sur les quatre sorties $S_{0..3}$
 - T_2 pilote un servomoteur M_0 en lui faisant faire des mouvements alternatifs, allant d'une position extrême à l'autre
 - Créez une interruption matérielle I_0 associée au bouton E_0 : lorsqu'on appuie une fois, le chenillard s'arrête, puis l'appui suivant, il reprend là où il s'était arrêté, et ainsi de suite.
 - Créez une interruption matérielle I_1 associée au bouton E_1 : lorsqu'on appuie une fois, le moteur termine un cycle complet, puis s'arrête. En cas d'appui pendant l'arrêt d'un cycle, il ne se passe rien. Le cycle alternatif reprend à l'appui suivant.

Chapitre 4

«freeRTOS»

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

4.1 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.

4.1.1 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 5 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 (v 9.0.0) 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.

4.1.2 Un système *open source*

FreeRTOS est distribué sous licence GNU General Public Licence (GPL) modifiée. Les utilisateurs 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 obligés d'ouvrir le code de ces applications. C'est le but de la clause d'exception de la licence GPL, 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 OpenRTOS [?] est une version payante de FreeRTOS, développée par la société *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 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 a été développée en conformité avec la norme de sécurité *IEC 61508* [?]. Un support commercial complet est disponible.

4.1.3 La popularité de FreeRTOS

Selon le site officiel, FreeRTOS a été téléchargé plus de 113 000 fois en 2014.

Le système est toujours en développement actif : la dernière version (9.0.0) a été lancée le 25 mai 2016.

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 utilisateurs 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\$.

4.2 Les tâches

Dans cette section, on présente la manière dont les tâches sont implémentées dans FreeRTOS. Le système permet l'utilisation de deux types de tâche distinctes :

- 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.

4.2.1 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 (voir section 4.2.2, page 33). 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.

4.2.2 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 4.1 page 34.

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.

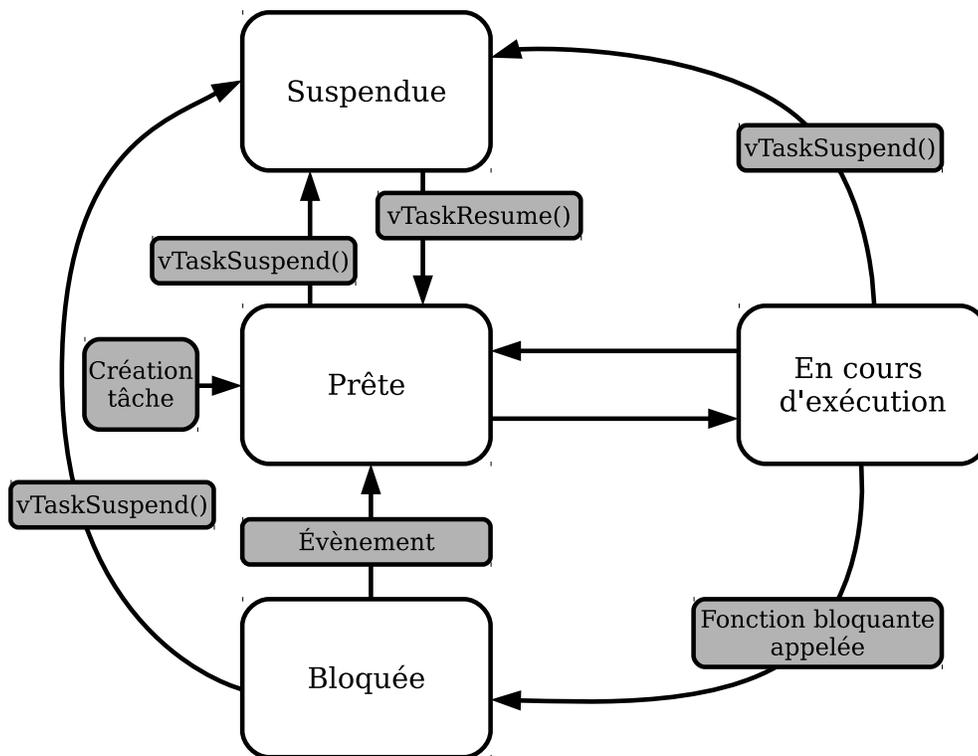


FIGURE 4.1 – Le cycle de vie d’une tâche sous FreeRTOS.

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 les suivantes :

```
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.

4.2.3 Les co-routines

Les co-routines sont des sortes de tâches "allégées" en besoins mémoires car elles partagent une pile unique.

Le cycle de vie d'une co-routine

Tout comme une tâche, une co-routine est à tout moment dans un état quelconque. Une co-routine ne peut pas être dans l'état *suspendue*, ainsi la liste des états possibles pour une co-routine est :

- **En cours d'exécution** (*Running*);
- **Prête** (*Ready*);
- **Bloquée** (*Blocked*).

On peut modéliser le cycle de vie d'une co-routine grâce à l'automate présent sur la figure 4.2.

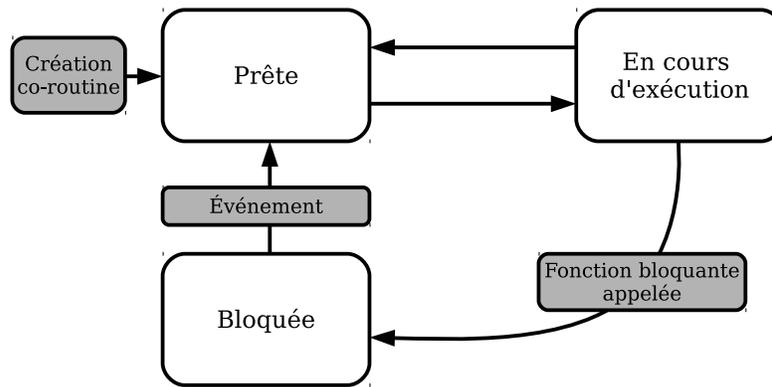


FIGURE 4.2 – Le cycle de vie d'une co-routine sous FreeRTOS.

Création, manipulation et destruction

Les co-routines sont créées via un appel à la fonction `xCoRoutineCreate`. On passe en paramètre à cette fonction :

- Le nom d'une fonction qui décrit le comportement de la co-routine ;
- La priorité de la co-routine ;
- Les paramètres à passer à la fonction nommée dans le premier argument.

Une fois créée, on lance la co-routine via la fonction `crSTART()`, dans la fonction qui définit son comportement.

Une co-routine se libère elle-même en appelant, au sein de la fonction qui décrit son comportement, la fonction `crEND()`.

On peut bloquer une co-routine pour un temps donné avec la fonction `crDELAY(cr_Handle, temps)`. `cd_Handle` représente le *handle* de la co-routine.

4.3 L'ordonnancement

4.3.1 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 tâches, et de commuter entre elles rapidement, donnant ainsi l'illusion que toutes les tâches sont exécutées en même temps.

4.3.2 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 tâches 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 4.3 page 37.

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 a 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 où 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`).

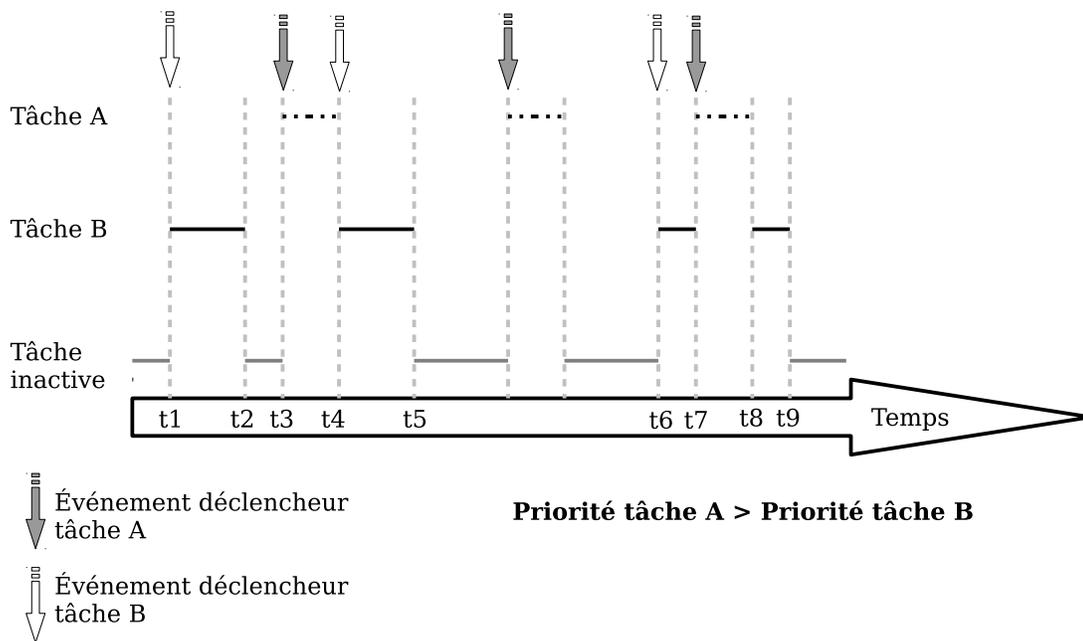


FIGURE 4.3 – Scénario d'ordonnancement temps réel préemptif sous FreeRTOS : On se place dans un système implémentant deux tâches : A et B. La priorité de la tâche A est supérieure à celle de la tâche B. Chaque tâche passe dans l'état *prête* suite à un événement déclencheur (par exemple l'appui sur une 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 son 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 a fini son travail, au temps t_8 , B peut terminer son exécution (temps t_9)

4.3.3 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 t_7 sur la figure 4.3). 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 [?].

Contexte d'exécution d'une tâche :

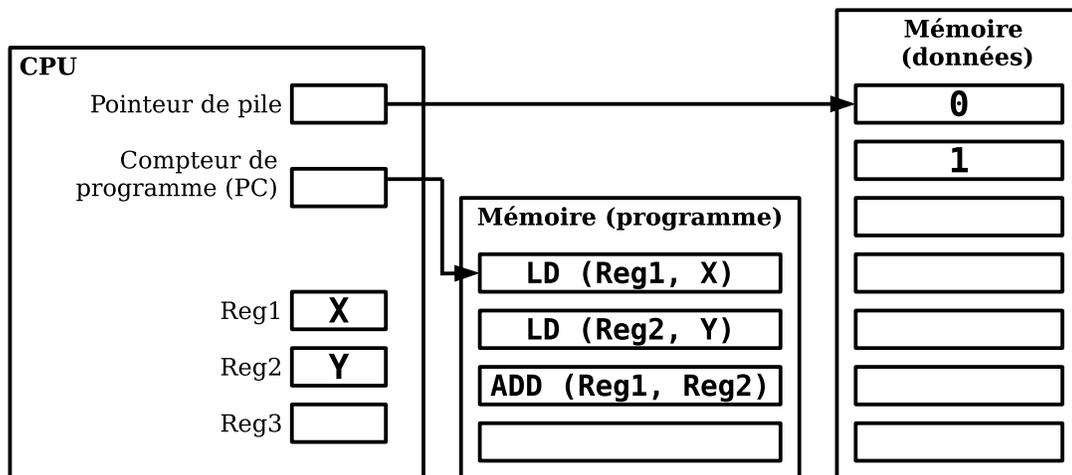


FIGURE 4.4 – Représentation du contexte d'une tâche dans le système : On peut différencier en mémoire d'un côté la zone occupée par les instructions de la tâche, et de l'autre la zone contenant les données (variables locales, ...). Dans les registres du processeur on retrouve le *Program Counter* (PC), 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.

La figure 4.4 page 38 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 ;
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 a été bloquée. La commutation de contexte est complète.

4.4 Exemples de mise en œuvre

4.4.1 Exemple avec les co-routines

Pré-requis

Pour que les co-routines puissent être utilisées dans FreeRTOS, il faut s'assurer, dans le fichier *FreeRTOSConfig.h* de la présence des lignes :

```
#define configUSE_IDLE_HOOK 1
```

```
...
```

```
/* Co-routine definitions. */
```

```
#define configUSE_CO_ROUTINES 1
```

```
#define configMAX_CO_ROUTINE_PRIORITIES ( (UBaseType_t) 1 )
```

- `configUSE_IDLE_HOOK` mis à 1 indique que la tâche «inactive» sera utilisée.
- `configUSE_CO_ROUTINES` mis à 1 indique à RTOS de lancer l'ordonnanceur de co-routines.

L'exemple met en œuvre 4 co-routines qui utilisent le même code mais avec un paramètre différent (`uxIndex`). Cet index permet à chaque co-routine de reconnaître ses données dans les variables globales.

Chaque co-routine fait clignoter une LED de la carte d'extension des TP, et avec une période spécifique.

```
#include <Arduino_FreeRTOS.h>
```

```
#include <croutine.h>
```

```
#define PRIORITY_0 0
```

```
#define NUM_COROUTINES 4
```

```
const int iFlashRates[NUM_COROUTINES] = {50 ,60 ,70, 80}; // Période des co-routines
```

```
const int iLEDTToFlash[NUM_COROUTINES] = {10, 11, 12, 13}; // Sorties associées
```

```
int iLEDState[NUM_COROUTINES] = {0, 0, 0, 0}; // Etat des sorties
```

```
void vFlashCoRoutine(CoRoutineHandle_t xHandle, UBaseType_t uxIndex) {
```

```
    // Les co-routines doivent démarrer par un appel à crSTART().
```

```
    crSTART( xHandle );
```

```
    for(;;) {
```

```
        // Attente pendant une période (cf index uxIndex dans iFlashRates[]
```

```
            crDELAY( xHandle, iFlashRates[uxIndex] );
```

```
        // bascule sur la LED de la sortie correspondante
```

```
            iLEDState[uxIndex] = 1 - iLEDState[uxIndex];
```

```
            digitalWrite( iLEDTToFlash[uxIndex], iLEDState[uxIndex] );
```

```
    }
```

```
    // Les co-routines doivent se terminer par un appel à crEND().
```

```
    crEND();
```

```
}
```

```

// Tache de fond "inactive" (lancée automatiquement)
void vApplicationIdleHook(){
    for(;;){
        vCoRoutineSchedule();
    }
}

// Fonction exécutée au reset de la carte Arduino
void setup() {
    int i;
    // initialize serial communication at 9600 bits per second:
    pinMode(10, OUTPUT); pinMode(11, OUTPUT);
    pinMode(12, OUTPUT); pinMode(13, OUTPUT);

    for( i = 0; i < NUM_COROUTINES; i++ ){
        xCoRoutineCreate( vFlashCoRoutine, PRIORITY_0, i );
    }

    // Lancement de RTOS
    vTaskStartScheduler();
}

void loop() {
    // Rien à coder ici, tout est fait par le noyau
}

```

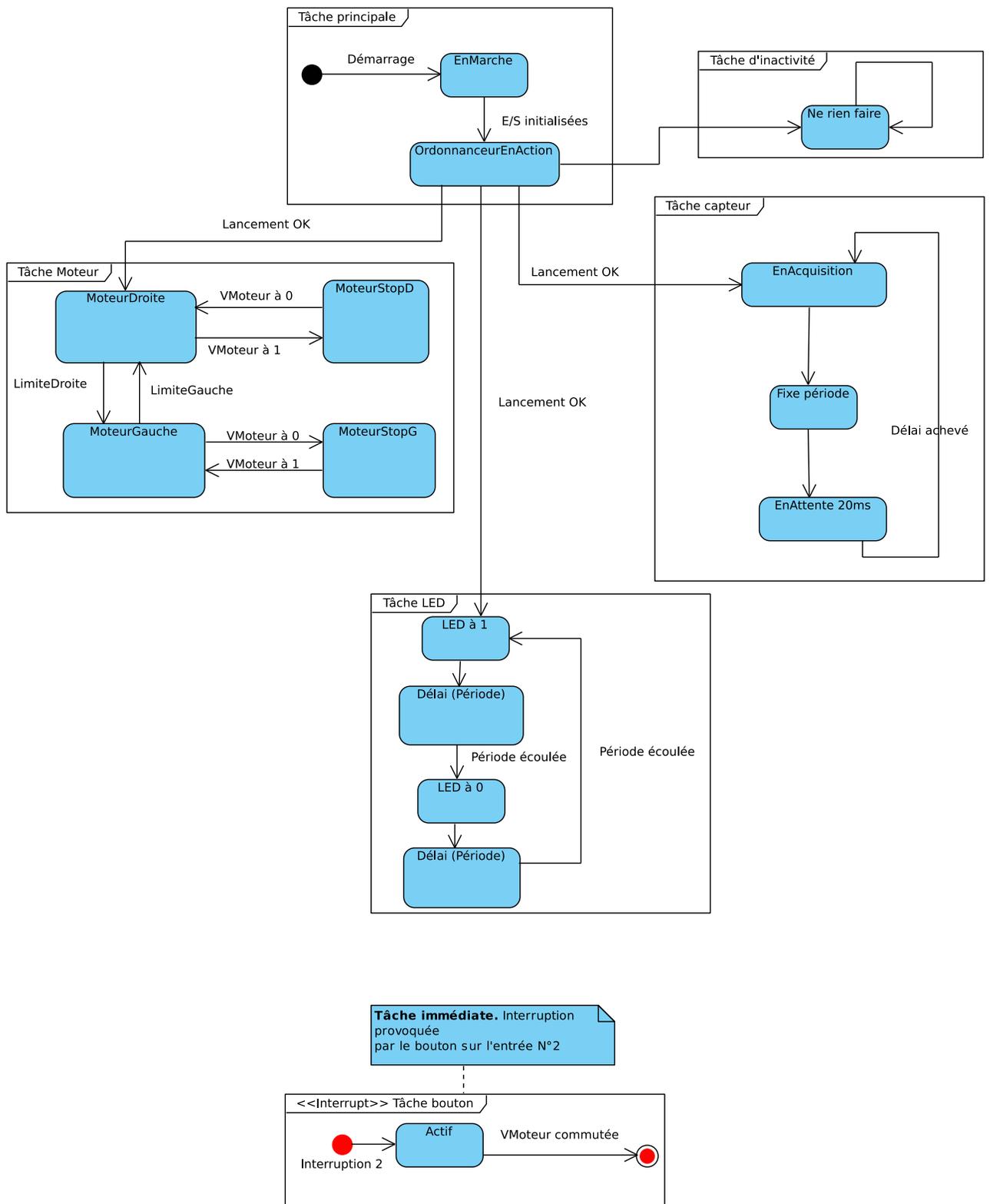


FIGURE 4.6 – Représentation graphique des tâches de l'exemple :

```

#include <Servo.h>
#include <semphr.h>
  
```

```

#define LED 10
const byte BOUTON = 2;

int PERIODE, PERIODE1, PERIODE4, PERIODE_CAPT;
int SEUIL = 100;

// Servo moteur
Servo mD;
int posD = 90;
int valD=0; // capteur droite
int MARCHE = 1;
int PERIODE_MOTEUR;

// ISR / tache
SemaphoreHandle_t xSema = NULL;

/*-----*/
/*----- Taches -----*/
/*-----*/

void TacheLED(void *pvParameters) {
  for (;;) {
    digitalWrite(LED, HIGH); // turn the LED on (HIGH is the voltage level)
    vTaskDelay(PERIODE); // wait for one second
    digitalWrite(LED, LOW); // turn the LED off by making the voltage LOW
    vTaskDelay(PERIODE); // wait for one second
  }
}

void TacheCAPTEUR(void *pvParameters) {
  for (;;) {
    // lecture de l'entrée capteur
    int val = analogRead(A1);
    if (val > 400) PERIODE = PERIODE4; else PERIODE = PERIODE1;
    Serial.println(val);
    vTaskDelay(10);
  }
}

void TacheMOTEUR(void *pvParameters) {
  for (;;) {
    while (posD <= 170) {
      vTaskDelay(PERIODE_MOTEUR);
      if (!MARCHE) continue;
      posD++;
      mD.write(posD);
    }
    while (posD >= 20) {

```

```

        vTaskDelay(PERIODE_MOTEUR);
        if (!MARCHE) continue;
        posD--;
        mD.write(posD);
    }
}

// Tache d'inactivité
void vApplicationIdleHook(){
    for (;;) {

    }
}

/*-----*/
/*----- <<ISR>> -----*/
/*-----*/

void ISR_Bouton() {
    static BaseType_t xHigherPriorityTaskWoken = pdTRUE;
    xSemaphoreGiveFromISR(xSema, &xHigherPriorityTaskWoken);
}

// Tache pour le traitement différé de l'ISR
void Tache_ISR_Bouton(void *pvParameters) {
    for (;;) {
        if (xSemaphoreTake(xSema, 0) == pdTRUE) {
            MARCHE = 1-MARCHE;
            digitalWrite(12, MARCHE);
            while (digitalRead(BOUTON) == 1) vTaskDelay(2);
        }
    }
}

/*-----*/
/*----- INIT -----*/
/*-----*/

// Démarrage de la carte
void setup() {
    // initialisation IO
    pinMode(LED, OUTPUT);
    pinMode(12, OUTPUT);

    // initialisation DATA
    PERIODE1 = 1000 / portTICK_PERIOD_MS; // 1 seconde
    PERIODE4 = 250 / portTICK_PERIOD_MS; // 1/4 seconde
    PERIODE_CAPT = 100 / portTICK_PERIOD_MS; // 1/10 seconde
}

```

```

PERIODE = PERIODE1;
PERIODE_MOTEUR = 30 / portTICK_PERIOD_MS; // 30 ms
mD.attach(8);
mD.write(posD);
delay(2000);

// Création des taches : code, nom, pile, param, priorité, &id
xTaskCreate(TacheLED,(const portCHAR *)"LED", 128, NULL, 1, NULL);
xTaskCreate(TacheCAPTEUR,(const portCHAR *)"CAPTEUR", 128, NULL, 2, NULL);
xTaskCreate(TacheMOTEUR,(const portCHAR *)"MOTEUR", 128, NULL, 1, NULL);
xTaskCreate(Tache_ISRButton,(const portCHAR *)"BOUTON", 128, NULL, 1, NULL);

// Interruptions matérielles (arduino, pas spécifique à RTOS)
attachInterrupt(0, ISR_Button, INPUT_PULLUP); // Mega : entrée 2 => vect. interruption 0
xSema = xSemaphoreCreateBinary();

// Lancement RTOS
vTaskStartScheduler();
}

void loop(){ /* VIDE */ }

```

Les interruptions matérielles sont spécifiques à l'architecture et on 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.

4.4.3 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 :

```

#include <Arduino_FreeRTOS.h>

struct Param {
    int Periode;
    char Message[16];
};

```

```

struct Param T[3] = {
  { 50, "Lecture" },
  { 70, "Ecriture"},
  { 80, "Interruption"}
};

/*-----*/
/*----- Taches -----*/
/*-----*/

void fCritique(char *str){
  while (*str) {
    Serial.print(*str);
    vTaskDelay(1);
    str++;
  }
  Serial.print('\n');
}

void TacheWR(void *pvParameters) {
  struct Param *data = (struct Param *)pvParameters;
  for (;;) {
    fCritique(data->Message);
    vTaskDelay(data->Periode);
  }
}

/*-----*/
/*----- INIT -----*/
/*-----*/

// Démarrage de la carte
void setup() {

  Serial.begin(9600);

  // Création des taches : code, nom, pile, param, priorité, &id
  xTaskCreate(TacheWR, (const portCHAR *) "WR", 128, (void *)&T[0], 1, NULL);
  xTaskCreate(TacheWR, (const portCHAR *) "WR", 128, (void *)&T[1], 1, NULL);
  xTaskCreate(TacheWR, (const portCHAR *) "WR", 128, (void *)&T[2], 1, NULL);

  // Lancement RTOS
  vTaskStartScheduler();
}

void loop(){ /* VIDE */ }

```

L'exécution du programme (Fig. 4.7) montre bien l'incohérence de l'affichage. Dans la deuxième version, le mécanisme de *mutex* est mis en place. À la différence



FIGURE 4.7 – Trace d'exécution sans sémaphore.

d'autres noyaux, il est important avec FreeRTOS de tester la valeur renvoyée par `xSemaphoreTake()`.

```
#include <Arduino_FreeRTOS.h>
#include <semphr.h>

struct Param {
    int Periode;
    char Message[16];
};

struct Param T[3] = {
    { 50, "Lecture" },
    { 70, "Ecriture"},
    { 80, "Interruption"}
};

SemaphoreHandle_t xSema;

/*-----*/
/*----- Taches -----*/
/*-----*/

void fCritique(char *str){
    while (*str) {
        Serial.print(*str);
        vTaskDelay(1);
        str++;
    }
    Serial.print('\n');
}

void TacheWR(void *pvParameters) {
    struct Param *data = (struct Param *)pvParameters;
```

```

for (;;) {
    if (xSemaphoreTake(xSema, 0) == pdTRUE) {
        fCritique(data->Message);
        xSemaphoreGive(xSema);
    }
    vTaskDelay(data->Periode);
}
}

/*-----*/
/*-----  INIT  -----*/
/*-----*/

// Démarrage de la carte
void setup() {

    Serial.begin(9600);

    xSema = xSemaphoreCreateMutex();

    // Création des taches : code, nom, pile, param, priorité, &id
    xTaskCreate(TacheWR, (const portCHAR *)"WR1", 128, (void *)&T[0], 1, NULL);
    xTaskCreate(TacheWR, (const portCHAR *)"WR2", 128, (void *)&T[1], 1, NULL);
    xTaskCreate(TacheWR, (const portCHAR *)"WR3", 128, (void *)&T[2], 1, NULL);

    // Lancement RTOS
    vTaskStartScheduler();
}

void loop(){ /* VIDE */ }

```

La nouvelle sortie du programme est donnée dans la figure Fig. 4.8.

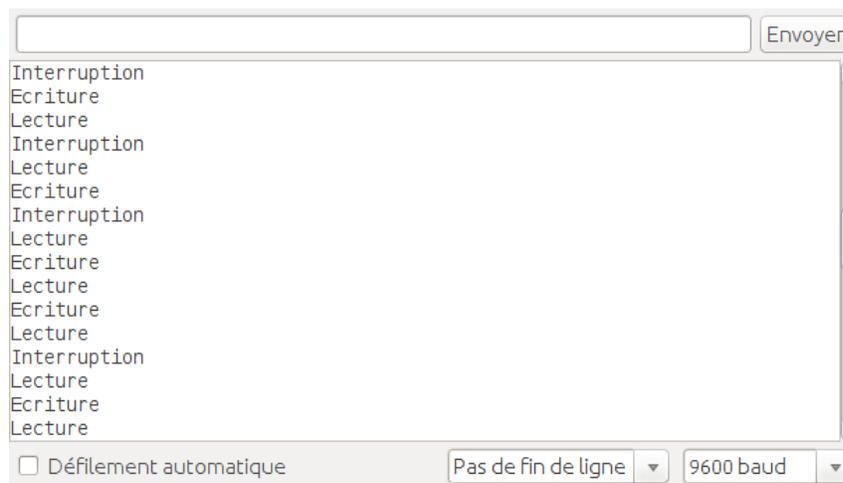


FIGURE 4.8 – Trace d'exécution sans sémaphore.

Annexe A

Installations

A.1 Installations des logiciels de base

arduino

```
sudo apt-get install arduino
```

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
```

L'environnement de base est installé. Il suffit de lancer la commande `arduino` pour démarrer.

ardublock

Cet outil permet de programmer la carte de manière graphique. Il n'est pas indispensable, mais permet de récupérer rapidement du code C généré automatiquement. La bibliothèque de composants est assez conséquente.

```
wget http://sourceforge.net/projects/ardublock/files/ardublock-all-20130712.jar
```

1. Lancer au moins une fois `arduino`. Cela a pour effet de créer le dossier `sketchbook` dans votre répertoire personnel.
2. Installation du plugin :

```
cd
cd sketchbook
mkdir tools
cd tools
mkdir ArduBlockTool
cd ArduBlockTool
mkdir tool
cd tool
wget http://sourceforge.net/projects/ardublock/files/ardublock-beta-20140828.jar
```

En résumé, vous devez disposer du fichier :

```
~/sketchbook/tools/ArduBlockTool/tool/
```

Voir les informations sur la page <http://sourceforge.net/projects/ardublock/files/> :
(en particulier le nom du fichier *jar* qui évolue en fonction des versions).

A.2 Noyau temps réel « NilRTOS »

Éventuellement :

```
sudo apt-get install git
```

puis,

```
git clone https://github.com/greiman/NilRTOS-Arduino.git
cd NilRTOS-Arduino/
cd libraries/
sudo cp -r NilAnalog/ /usr/share/arduino/libraries/
sudo cp -r NilRTOS/ /usr/share/arduino/libraries/
sudo cp -r NilTimer1/ /usr/share/arduino/libraries/
sudo cp -r TwiMaster/ /usr/share/arduino/libraries/
```

Donner l'accès en lecture pour tous aux dossiers précédents. L'installation du noyau est terminée.

Avec Linux, pour utiliser le moniteur 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
```

Annexe B

Plans de la «mega2560»

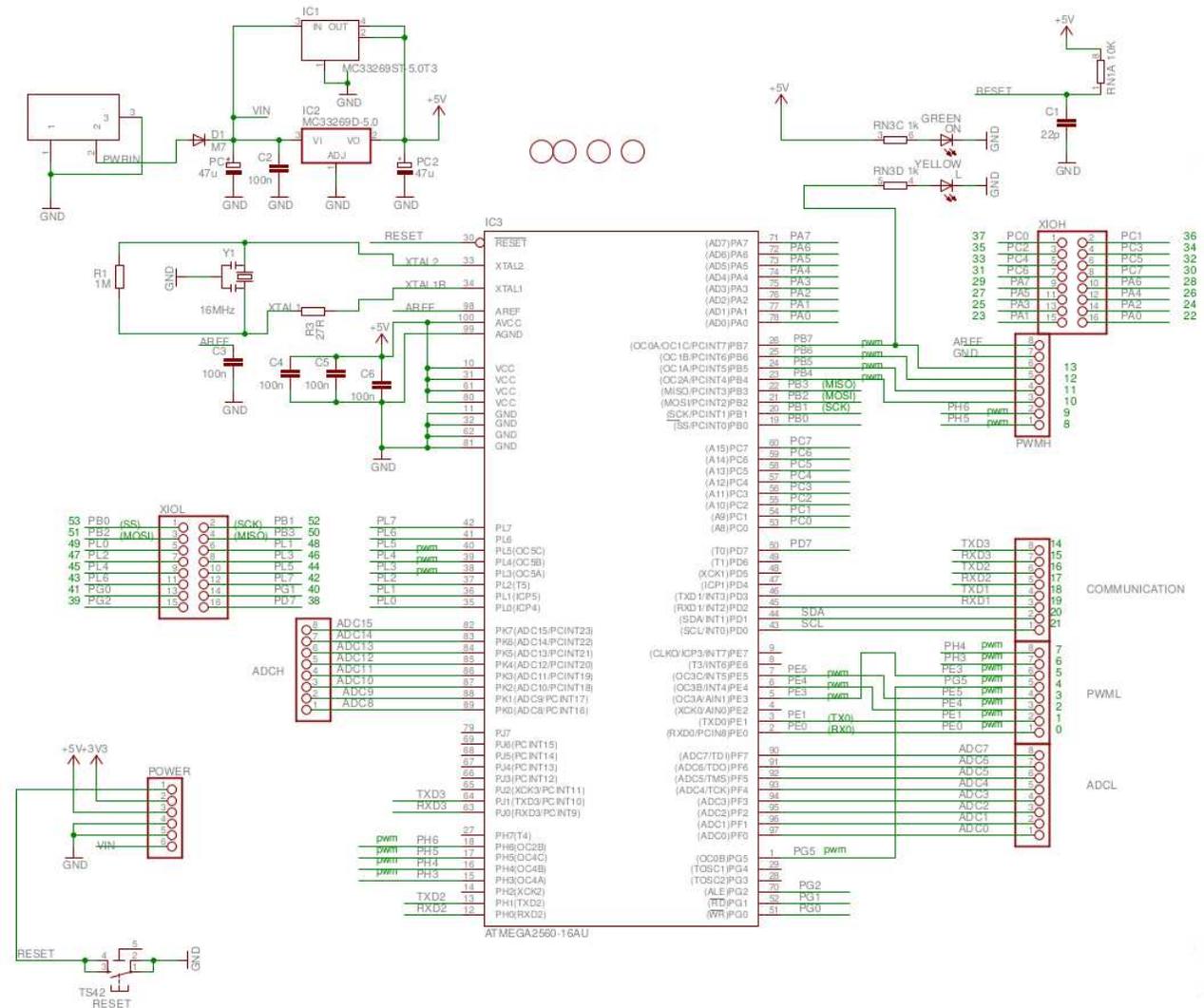


FIGURE B.1 – Schéma électrique de l'Arduino Mega2560 (1/2)

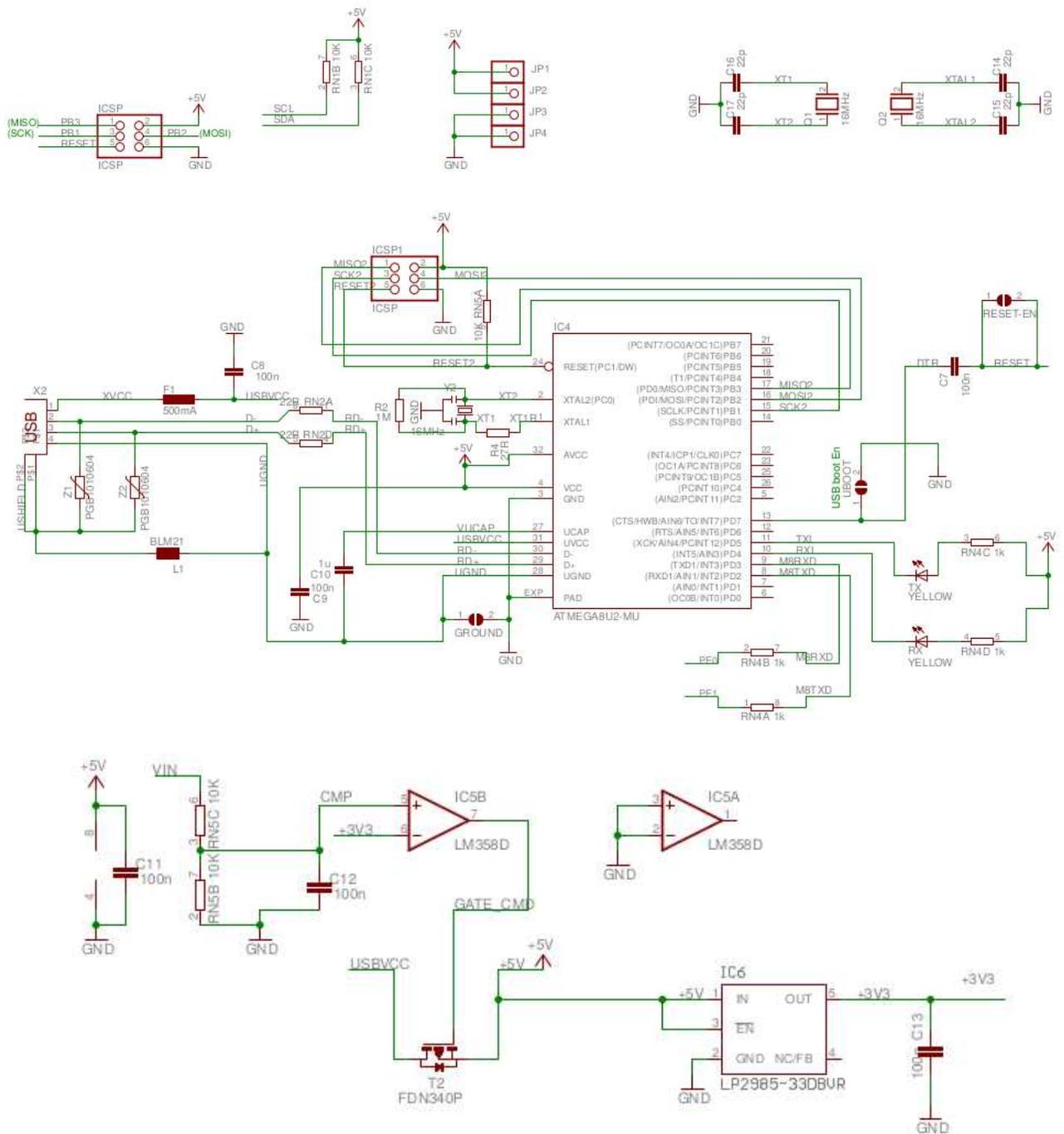


FIGURE B.2 – Schéma électrique de l'Arduino Mega2560 (2/2)

Annexe C

Mise en œuvre

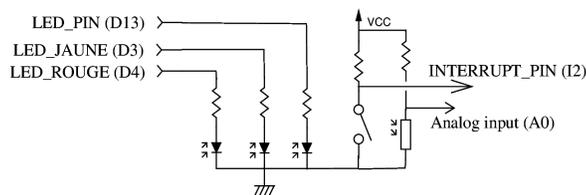


FIGURE C.1 – Montage de test pour la mise en œuvre.

```
/*
 * MEO nilRTOS
 * threads / ISR / semaphore / RS
 * CG - 1er oct. 2014
 */
#include <NilRTOS.h>
#include <NilSerial.h>

// redéfinition pour utiliser "Serial" de NilRTOS
#define Serial NilSerial

// The LED is attached to pin 13 on Arduino.
const uint8_t LED_PIN = 13;
const uint8_t LED_JAUNE = 3;
const uint8_t LED_ROUGE = 4;
int X=0;

// interruption :
const uint8_t INTERRUPT_PIN = 2;
SEMAPHORE_DECL(isrSem, 0); // synchro IT et thread associé
int flag=0;// pour la bascule
// <<< IT >>>>
NIL_IRQ_HANDLER(isrFcn){
    /* sauvegarde r18-r31.*/
    NIL_IRQ_PROLOGUE();
```

```

    nilSemSignalI(&isrSem); // débloque le thread associé
    NIL_IRQ_EPILOGUE();
}

// Handler thread , marche avec l'IT
NIL_WORKING_AREA(waThreadIT, 64);
NIL_THREAD(ThreadIT, arg) {
    Serial.begin(9600);
    attachInterrupt(0, isrFcn, RISING); // init de l'IT
    while (1) {
        nilSemWait(&isrSem);
        flag = 1 - flag;
        if (flag) digitalWrite(LED_JAUNE, HIGH);
        else digitalWrite(LED_JAUNE, LOW);
        nilThdSleepMilliseconds(300);
        nilSemReset(&isrSem, 0); // pour éviter les rebonds
        Serial.print(F("A"));
    }
}

// Sémaphore init à 0
SEMAPHORE_DECL(sem, 0);
//-----
/*
 * Thread 1, marche avec thread 2 ( synchro avec le sémaphore sem
 * la LED est éteinte seulement s'il y a une présence
 */
NIL_WORKING_AREA(waThread1, 128);
NIL_THREAD(Thread1, arg) {
    while (TRUE) {
        nilSemWait(&sem);
        X = analogRead(A0);
        if (X>100) digitalWrite(LED_PIN, LOW);
    }
}

//-----
/*
 * Thread 2, LED allumée + demande à thread 1 de l'éteindre
 */
NIL_WORKING_AREA(waThread2, 128);
NIL_THREAD(Thread2, arg) {
    pinMode(LED_PIN, OUTPUT);
    while (TRUE) {
        digitalWrite(LED_PIN, HIGH);
        nilThdSleepMilliseconds(200);
        nilSemSignal(&sem);
        nilThdSleepMilliseconds(200);
    }
}
}

```

```

// Thread 3 - périodique
NIL_WORKING_AREA(waThread3, 128);
NIL_THREAD(Thread3, arg) {
    int i=100;
    pinMode(LED_JAUNE, OUTPUT);

    while (TRUE) {
        digitalWrite(LED_ROUGE, HIGH);
        nilThdSleepMilliseconds(1000);
        digitalWrite(LED_ROUGE, LOW);
        nilThdSleepMilliseconds(i);
        i+=10;
        if (i==300) i=100;
    }
}

NIL_THREADS_TABLE_BEGIN()
NIL_THREADS_TABLE_ENTRY("thread1", Thread1, NULL, waThread1, sizeof(waThread1))
NIL_THREADS_TABLE_ENTRY("threadIT", ThreadIT, NULL, waThreadIT, sizeof(waThreadIT))
NIL_THREADS_TABLE_ENTRY("thread2", Thread2, NULL, waThread2, sizeof(waThread2))
NIL_THREADS_TABLE_ENTRY("thread3", Thread3, NULL, waThread3, sizeof(waThread3))
NIL_THREADS_TABLE_END()
//-----
void setup() {
    // Start Nil RTOS.
    pinMode(INTERRUPT_PIN, INPUT);
    pinMode(LED_ROUGE, OUTPUT);
    digitalWrite(LED_ROUGE, HIGH);

    nilSysBegin();
}
//-----
void loop() {
    // Not used.
}

```

Annexe D

Carte d'extension pour les TP

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.

D.0.1 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);
}

```

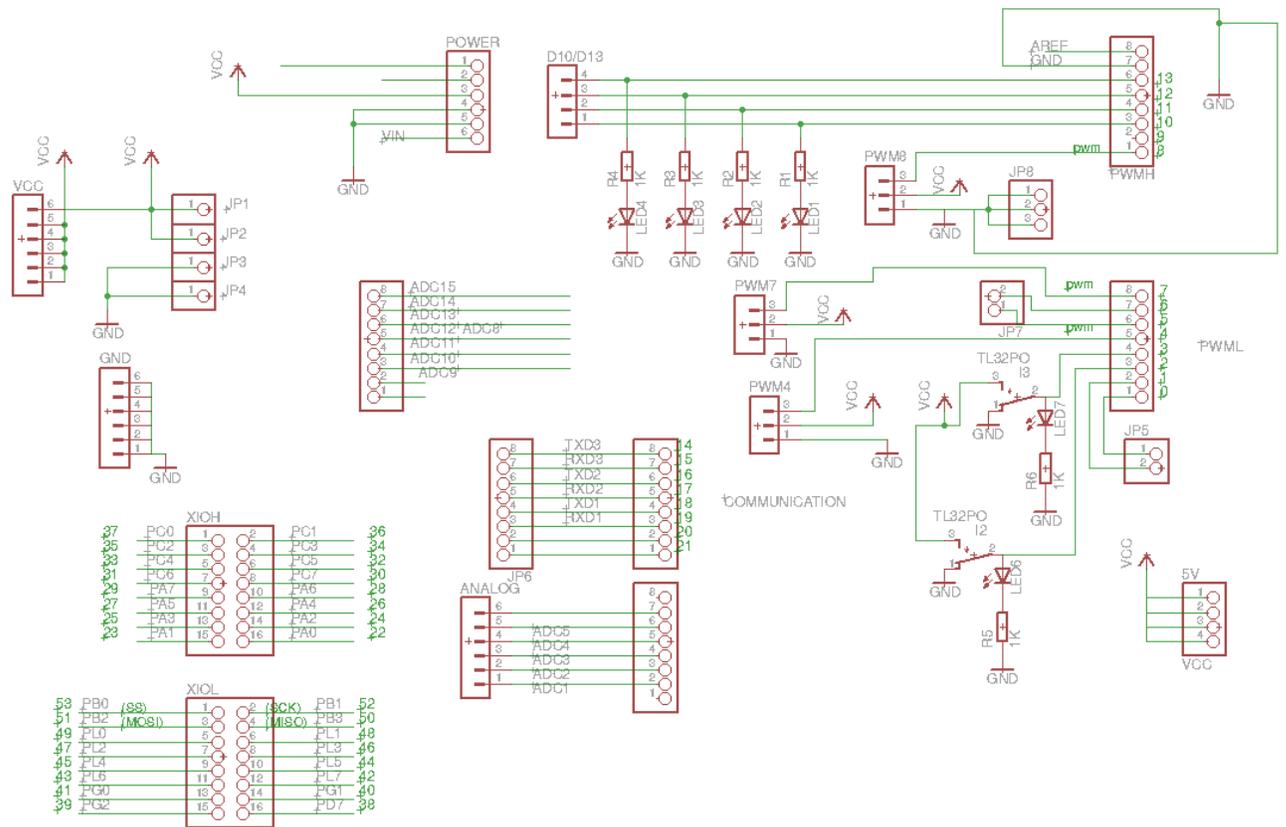


FIGURE D.1 – Carte d'extension.

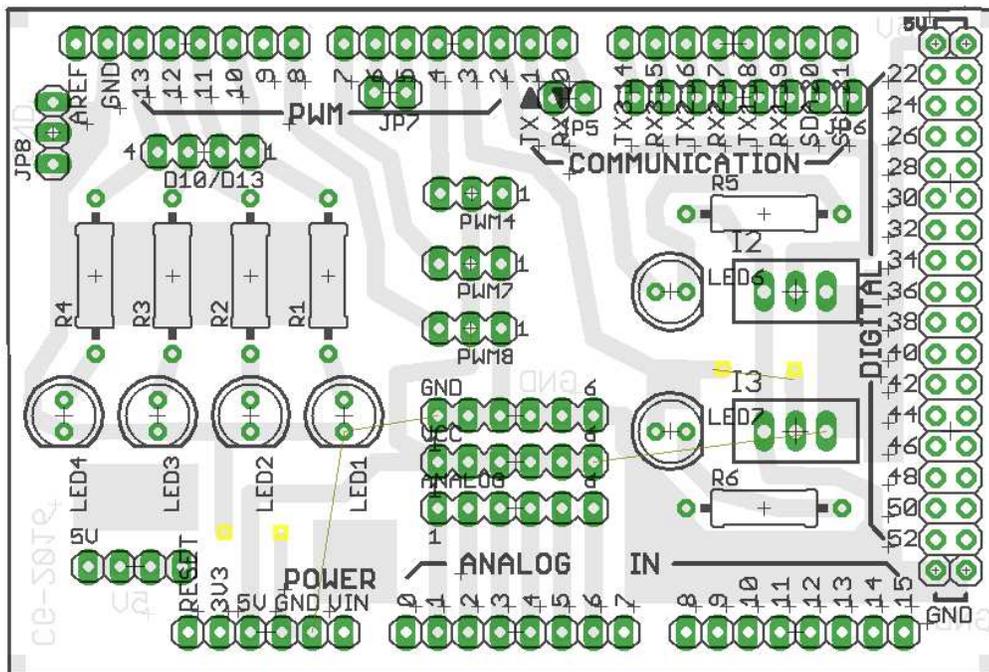


FIGURE D.2 – Carte d’extension : implantation.

Annexe E

Librairie python pour arduino

```
"""
pyduino.py
Librairie python pour arduino : à placer dans le répertoire de travail
"""
import serial

class Arduino():
    def __init__(self, serial_port='/dev/ttyACMO', baud_rate=9600,
                 read_timeout=5):
        """
        Initializes the serial connection to the Arduino board
        """
        self.conn = serial.Serial(serial_port, baud_rate)
        self.conn.timeout = read_timeout # Timeout for readline()

    def set_pin_mode(self, pin_number, mode):
        """
        Performs a pinMode() operation on pin_number
        Internally sends b'M{mode}{pin_number}' where mode could be:
        - I for INPUT
        - O for OUTPUT
        - P for INPUT_PULLUP MO13
        """
        command = (''.join(('M',mode,str(pin_number)))).encode()
        #print 'set_pin_mode =',command,(''.join(('M',mode,str(pin_number))))
        self.conn.write(command)

    def digital_read(self, pin_number):
        """
        Performs a digital read on pin_number and returns the value (1 or 0)
        Internally sends b'RD{pin_number}' over the serial connection
        """
        command = (''.join(('RD', str(pin_number)))).encode()
```

```

self.conn.write(command)
line_received = self.conn.readline().decode().strip()
header, value = line_received.split(':') # e.g. D13:1
if header == ('D'+ str(pin_number)):
    # If header matches
    return int(value)

def digital_write(self, pin_number, digital_value):
    """
    Writes the digital_value on pin_number
    Internally sends b'WD{pin_number}:{digital_value}' over the serial
    connection
    """
    command = (''.join(('WD', str(pin_number), ':',
        str(digital_value))))).encode()
    self.conn.write(command)

def M0(self, pwm):
    """
    Writes the digital_value on pin_number
    Internally sends b'P:{pwm_value}' over the serial
    connection
    """
    command = (''.join(('WPO:', str(pwm))))).encode()
    self.conn.write(command)

def M1(self, pwm):
    """
    Writes the digital_value on pin_number
    Internally sends b'Q:{pwm_value}' over the serial
    connection
    """
    command = (''.join(('WQO:', str(pwm))))).encode()
    self.conn.write(command)

def M2(self, pwm):
    """
    Writes the digital_value on pin_number
    Internally sends b'R:{pwm_value}' over the serial
    connection
    """
    command = (''.join(('WRO:', str(pwm))))).encode()
    self.conn.write(command)

def analog_read(self, pin_number):
    """

```

```

    Performs an analog read on pin_number and returns the value (0 to 1023)
    Internally sends b'RA{pin_number}' over the serial connection
    """
    command = (''.join(('RA', str(pin_number)))).encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A4:1
    if header == ('A'+ str(pin_number)):
        # If header matches
        return int(value)

def A(self, pin_number):
    """
    Performs an analog read on pin_number and returns the value (0 to 1023)
    Internally sends b'RA{pin_number}' over the serial connection
    """
    command = (''.join(('RA', str(pin_number)))).encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A4:1
    if header == ('A'+ str(pin_number)):
        # If header matches
        return int(value)

def analog_write(self, pin_number, analog_value):
    """
    Writes the analog value (0 to 255) on pin_number
    Internally sends b'WA{pin_number}:{analog_value}' over the serial
    connection
    """
    command = (''.join(('WA', str(pin_number), ':',
        str(analog_value)))).encode()
    self.conn.write(command)

def S0(self, val):
    command = (''.join(('WD', str(10), ':',
        str(val)))).encode()
    self.conn.write(command)

def S1(self, val):
    command = (''.join(('WD', str(11), ':',
        str(val)))).encode()
    self.conn.write(command)

def S2(self, val):
    command = (''.join(('WD', str(12), ':',
        str(val)))).encode()

```

```

self.conn.write(command)

def S3(self, val):
    command = (''.join(('WD', str(13), ':',
        str(val))))).encode()
    self.conn.write(command)

def E0(self):
    return self.digital_read(2)

def E1(self):
    return self.digital_read(3)

def A1(self):
    command = ('RA1').encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A1:1
    if header == ('A1'):
        # If header matches
        return int(value)

def A2(self):
    command = ('RA2').encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A1:1
    if header == ('A2'):
        # If header matches
        return int(value)

def A3(self):
    command = ('RA3').encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A1:1
    if header == ('A3'):
        # If header matches
        return int(value)

def A4(self):
    command = ('RA4').encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A1:1
    if header == ('A4'):
        # If header matches

```

```

        return int(value)

def A5(self):
    command = ('RA5').encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A1:1
    if header == ('A5'):
        # If header matches
        return int(value)

def A6(self):
    command = ('RA6').encode()
    self.conn.write(command)
    line_received = self.conn.readline().decode().strip()
    header, value = line_received.split(':') # e.g. A6:1
    if header == ('A6'):
        # If header matches
        return int(value)

def close(self):
    """
    To ensure we are properly closing our connection to the
    Arduino device.
    """
    self.conn.close()
    print 'Connection to Arduino closed'

```

Annexe F

Firmware pour arduino : utilisation avec *pyduino*

Ce fichier *arduinoPy.ino* doit être placé dans un dossier *arduinoPy* pour être reconnu par l'IDE *arduino*.

```
#include <Servo.h>

/*
 * arduinoPy.ino
 * Commandes par l'exemple :
 *
 * - RD13 -> lecture de l'entrée digitale 13
 * - RA4 -> lecture de l'entrée analogique 4
 * - WD13:1 -> écrit 1 (HIGH) sur la sortie digitale 13
 * - WA6:125 -> écrit 125 sur la sortie analogique 6 (PWM)
 */

char operation; // operation (R, W, ...)
char mode; // mode (D, A)
int pin_number; // num. broche
int digital_value; // valeur digitale
int analog_value; // valeur analogique
int value_to_write; // valeur à écrire
int wait_for_transmission = 5; // délai en ms

Servo M0;
Servo M1;
Servo M2;
int pos0 = 90;
int pos1 = 90;
int pos2 = 90;

void set_pin_mode(int pin_number, char mode){
```

```

/*
 * pinMode() operation
 * mode :
 * - I: mode = INPUT
 * - O: mode = OUTPUT
 * - P: mode = INPUT_PULLUP
 */

switch (mode){
  case 'I':
    pinMode(pin_number, INPUT);
    break;
  case 'O':
    pinMode(pin_number, OUTPUT);
    break;
  case 'P':
    pinMode(pin_number, INPUT_PULLUP);
    break;
}
}

void digital_read(int pin_number){
  /*
   * lecture d'une entrée digitale
   * D{pin_number}:{value}\n value = 0 ou 1
   */

  digital_value = digitalRead(pin_number);
  Serial.print('D');
  Serial.print(pin_number);
  Serial.print(':');
  Serial.println(digital_value);
}

void analog_read(int pin_number){
  /*
   * Lecture d'une entrée analogique
   * A{pin_number}:{value}\n 0<= value <= 1023
   */

  analog_value = analogRead(pin_number);
  Serial.print('A');
  Serial.print(pin_number);
  Serial.print(':');
  Serial.println(analog_value);
}

```

```

void digital_write(int pin_number, int digital_value){
    /*
     * écriture de 0 ou 1 sur une sortie analogique
     */
    digitalWrite(pin_number, digital_value);
}

void analog_write(int pin_number, int analog_value){
    /*
     * écriture d'une valeur analogique
     * 0 <= analog_value <= 255
     */
    analogWrite(pin_number, analog_value);
}

void setup() {
    Serial.begin(9600); // 9600 baud
    Serial.setTimeout(100); // au lieu de 1000ms

    pinMode(10, OUTPUT);
    pinMode(11, OUTPUT);
    pinMode(12, OUTPUT);
    pinMode(13, OUTPUT);
    pinMode(2, INPUT);
    pinMode(3, INPUT);
    M0.attach(4);
    M1.attach(7);
    M2.attach(8);
    delay(30);
    M0.write(pos0);
    M1.write(pos1);
    M2.write(pos2);
}

void loop() {
    // caractères disponibles ?
    if (Serial.available() > 0) {
        operation = Serial.read();
        delay(wait_for_transmission);
        mode = Serial.read();
        pin_number = Serial.parseInt(); // attend un entier
        if (Serial.read()==':') {
            value_to_write = Serial.parseInt();
        }
        switch (operation){
            case 'R': // lecture Ex: RD12, RA4
                if (mode == 'D'){

```

```

        digital_read(pin_number);
    } else if (mode == 'A'){
        analog_read(pin_number);
        } else {
            break;
        }
    break;

case 'W': // écriture, Ex: WD3:1, WA8:255
    if (mode == 'D'){
        digital_write(pin_number, value_to_write);
    } else if (mode == 'A'){
        analog_write(pin_number, value_to_write);
    } else if (mode == 'P'){ // pwm0
        pos0 = value_to_write;
        M0.write(pos0);
    } else if (mode == 'Q'){ // pwm1
        pos1 = value_to_write;
        M1.write(pos1);
    } else if (mode == 'R'){ // pwm2
        pos2 = value_to_write;
        M2.write(pos2);
    } else {
        break; // Unexpected mode
    }
    break;

case 'M': // Pin mode, Ex: MI3, MO3, MP3
    set_pin_mode(pin_number, mode);
    break;

default: // Unexpected char
    break;
}
}
}

```

Index

Ardublock, 9
attachInterrupt, 28

compteur programme, 15
contexte d'un processus, 15

exclusion mutuelle, 21, 22

Interruption, 28

loop, 10

multitâche, 14

NIL_IRQ_EPILOGUE, 28
NIL_IRQ_PROLOGUE, 28
NIL_THREAD, 17
NIL_THREADS_TABLE_BEGIN, 17
NIL_THREADS_TABLE_END, 17
NIL_THREADS_TABLE_ENTRY, 17
NIL_WORKING_AREA, 17
nilRTOS, 16
nilSemReset, 28
nilSemSignal, 25
nilSemSignalI, 28
nilSemWait, 25
nilSysBegin, 17

ordonnanceur, 20

pile, 15
pointeur de pile, 15
processus, 14
 $P(S)$, 22
python, 11

région critique, 22
rendez-vous, 21
ressource critique, 21

sémaphores, 20
scratch, 9
S4A, 10

sémaphore, 22
SEMAPHORE_DECL, 23, 25
setup, 10
synchronisation, 21, 26
synchronized, 22

tâche, 14

 $V(S)$, 22