Informatique industrielle

Projet Robot Gilbert - Capteur US

M1 E3A
Semestre 1

Rédigé par
Maxime SECHEHAYE

11/01/2021
Encadrant : Gael PONGNOT

Table des matières

Introduction

L'objectif de ce projet est de permettre à un robot de se déplacer tout en évitant des obstacles. Pour ce faire, un capteur ultrason est fixé sur l'avant du véhicule.

1 Capteur ultrason

1.1 Principe de fonctionnement

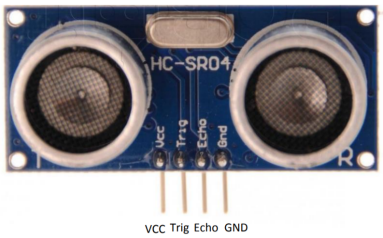

FIGURE 1 – Image du capteur ultrason HC-SR04

Le capteur ultrason utilisé est le HC-SR04 (voir figure 1).
Son principe de fonctionnement est simple et se décrit en trois étapes :

— On envoie un signal *Trig* à l'état haut pendant 10 µs qui va enclencher l'envoi d'ultrasons
— Le capteur envoie alors 8 salves d'ultrasons à 40 kHz
— Un signal *Echo* à l'état haut est reçu par le capteur et sa durée à l'état haut t_h est proportionnel au temps entre l'émission et la réception des ultrasons

Ce principe de fonctionnement est illustré à la figure 2.

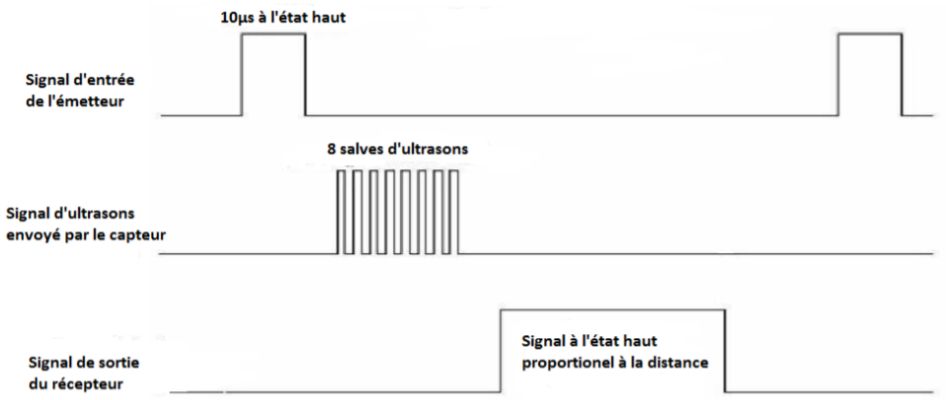

FIGURE 2 – Schéma temporel illustrant le fonctionnement du capteur, schéma tirée de la documentation technique fourni par le site *robot-maker*

On peut alors calculer la distance séparant le capteur de l'obstacle avec :

$$d_{obs} = \frac{t_h v_{us}}{2} \tag{1}$$

avec $v_h = 340\text{m.s}^{-1}$.

Il est recommandé par la documentation technique d'espacer les salves d'ultrasons d'au moins 60ms pour s'assurer que les ondes relatives à l'objet précédent aient été reçues par le capteur.

1.2 Codage des fonctions associées

Tout d'abord, la configuration initiale reste la même :

```
  //Activation des périphériques utilises
LPC_SYSCON->SYSAHBCLKCTRL0 |= GPIO | CTIMER0 | SWM | GPIO_INT | MRT;

//Configuration en sortie de la broche P0_17 (signal Trig)
LPC_GPIO_PORT->DIR0 |= (1 << 17);

// Configuration de l'horloge a sa frequence maximale
LPC_PWRD_API->set_fro_frequency(30000);
);
```

1.2.1 Envoi du signal *Trig*

Dans un premier temps, j'ai généré le signal *Trig* par une PWM tel que nous l'avons appris lors du TP n°4. Bien que le signal obtenu était cohérent avec mes attentes, j'ai du changé de méthode de génération car la réception du signal *Echo* nécessite le mode *Capture* du timer. Or ce mode utilise et manipule le compteur *TC* déjà utilisé pour une PWM.

J'ai alors opté pour la génération d'une PWM par un timer MRT couplé à une fonction d'interruption. C'était une proposition déjà évoquée lors du projet de l'orgue mais, ne l'ayant pas utilisé pour cet ancien projet, je me propose d'expliquer son fonctionnement ici.

Le principe du timer MRT est celui d'un "décompteur" à partir d'une valeur rentrée par l'utilisateur. Lorsque le timer atteint 0, on peut soit le relancer manuellement en entrant une nouvelle valeur de départ dans le registre idoine, ou bien relancer systématiquement un décompte à partir de la même valeur grâce au mode par "interruptions répétées".

Pour générer une PWM, on va réaliser des interruptions répétées toutes les microsecondes afin de mettre à jour le signal.

On utilise le timer MRT0. On le configure alors pour qu'il génère des interruptions :

```
LPC_MRT->Channel[0].CTRL |= (1<<0); //Active l'interruption du MRT0
LPC_MRT->Channel[0].CTRL |= (0<<1); //Active le mode a interruptions ↩
    repetees

NVIC_EnableIRQ(MRT_IRQn);  //Active les interruptions du MRT
```

La configuration du registre CTRL0 se justifie avec la figure 3

Table 251. Control register (CTRL[0:3], address 0x4000 4008 (CTRL0) to 0x4000 4038 (CTRL3)) bit description

Bit	Symbol	Value	Description	Reset value
0	INTEN		Enable the TIMERn interrupt.	0
		0	Disable.	
		1	Enable.	
2:1	MODE		Selects timer mode.	0
		0x0	Repeat interrupt mode.	
		0x1	One-shot interrupt mode.	
		0x2	One-shot bus stall mode.	
		0x3	Reserved.	
31:3	-		Reserved.	0

FIGURE 3 – Explication du registre CTRL0 d'après la documentation du LPC804

On définit ensuite la valeur à partir de laquelle le timer va décompter :

```
LPC_MRT->Channel[0].INTVAL=15; //Il faut 15 tics d'horloge pour atteindre 0
```

Comme il y a 15×10^6 tics d'horloge par seconde, cela correspond bien à des interruptions toutes les microsecondes.

On écrit alors la fonction d'interruption à l'intérieur de laquelle nous allons modeler le signal PWM :

```
#define borne_mrt0_bas 60000
#define borne_mrt0_haut 60010
uint32_t compteur_mrt0=0;

void MRT_IRQHandler(void){
  if ((LPC_MRT->Channel[0].STAT & 1) == 1){ //Interruption due au MRT0
    if (TRIG == 0 && compteur_mrt0 < borne_mrt0_bas){
      compteur_mrt0++;
    //Continuer a compter si on n'est pas au debut de la PWM
    }
    else if (TRIG == 0 && compteur_mrt0 >= borne_mrt0_bas){
      compteur_mrt0++;
      TRIG=1;
    //Passer le signal a l'etat haut si on depasse l'instant initial
    }
    else if (TRIG == 1 && compteur_mrt0 < borne_mrt0_haut){
      compteur_mrt0++;
    //Continuer à compter si on n'a pas encore depasse l'instant final
    }
    else if (TRIG == 1 && compteur_mrt0 >= borne_mrt0_haut){
      compteur_mrt0=0;
      TRIG=0;
    //Tout remettre a zero des qu'un cycle de PWM est termine
    }
    LPC_MRT->Channel[0].STAT |= 1; //On pense a abaisser le drapeau d'↩
       interruption
  }
}
```

1.2.2 Réception du signal *Echo*

Pour réceptionner le signal *Echo* et déterminer sa durée à l'état haut, on utilise le mode capture du timer CTIMER0.

On assigne tout d'abord le signal reçu au signal de référence du mode capture grâce à la Switch Matrix via le registre PINASSIGN3 :

```
  LPC_SWM->PINASSIGN3 &= ~(0xFF00); // Reinitialisation du registre
  LPC_SWM->PINASSIGN3 |= (16 << 8); // Le capture se fait vis à vis de Trig
```

On configure ensuite les registres CTCR et CCR définissant le fonctionnement du mode capture (voir figures 4 et 5).

```
  LPC_CTIMER0->CTCR |= (1 << ENCC) | (0 << SELCC); // Remise à zéro du ↩
     timer lors d'un front montant
  LPC_CTIMER0->CCR |= (0b011 << 0); // On capture sur front descendant
```

Table 234. Count Control Register (CTCR, offset 0x070) bit description

4	ENCC	-	Setting this bit to 1 enables clearing of the timer and the prescaler when the capture-edge event specified in bits 7:5 occurs.	0
7:5	SELCC		Edge select. When bit 4 is 1, these bits select which capture input edge will cause the timer and prescaler to be cleared. These bits have no effect when bit 4 is low. Values 0x6 to 0x7 are reserved.	0
		0x0	Channel 0 Rising Edge. Rising edge of the signal on capture channel 0 clears the timer (if bit 4 is set).	
		0x1	Channel 0 Falling Edge. Falling edge of the signal on capture channel 0 clears the timer (if bit 4 is set).	
		0x2	Channel 1 Rising Edge. Rising edge of the signal on capture channel 1 clears the timer (if bit 4 is set).	
		0x3	Channel 1 Falling Edge. Falling edge of the signal on capture channel 1 clears the timer (if bit 4 is set).	
		0x4	Channel 2 Rising Edge. Rising edge of the signal on capture channel 2 clears the timer (if bit 4 is set).	
		0x5	Channel 2 Falling Edge. Falling edge of the signal on capture channel 2 clears the timer (if bit 4 is set).	
		0x6	Reserved	
		0x7	Reserved	

FIGURE 4 – Extrait du registre CTCR d'après la documentation du LPC804

Table 231. Capture Control Register (CCR, offset 0x028) bit description

Bit	Symbol	Description	Reset Value
0	CAP0RE	Rising edge of capture channel 0: a sequence of 0 then 1 causes CR0 to be loaded with the contents of TC. 0 = disabled. 1 = enabled.	0
1	CAP0FE	Falling edge of capture channel 0: a sequence of 1 then 0 causes CR0 to be loaded with the contents of TC. 0 = disabled. 1 = enabled.	0
2	CAP0I	Generate interrupt on channel 0 capture event: a CR0 load generates an interrupt.	0
3	CAP1RE	Rising edge of capture channel 1: a sequence of 0 then 1 causes CR1 to be loaded with the contents of TC. 0 = disabled. 1 = enabled.	0
4	CAP1FE	Falling edge of capture channel 1: a sequence of 1 then 0 causes CR1 to be loaded with the contents of TC. 0 = disabled. 1 = enabled.	0
5	CAP1I	Generate interrupt on channel 1 capture event: a CR1 load generates an interrupt.	0
6	CAP2RE	Rising edge of capture channel 2: a sequence of 0 then 1 causes CR2 to be loaded with the contents of TC. 0 = disabled. 1 = enabled.	0
7	CAP2FE	Falling edge of capture channel 2: a sequence of 1 then 0 causes CR2 to be loaded with the contents of TC. 0 = disabled. 1 = enabled.	0
8	CAP2I	Generate interrupt on channel 2 capture event: a CR2 load generates an interrupt.	0
31:9	-	Reserved. Read value is undefined, only zero should be written.	NA

FIGURE 5 – Explication du registre CCR d'après la documentation du LPC804

On pense enfin à activer les interruptions liées à la capture :

```
1 NVIC_EnableIRQ(CTIMER0_IRQn);
```

On peut finalement coder la fonction d'interruption qui va permettre de déterminer la distance séparant le capteur de l'obstacle détecté :

```
1  void CTIMER0_IRQHandler(void) {
2    int delta_t = 0;
3    int distance = 0;
4    char temps[32];
5    delta_t = LPC_CTIMER0->CR[0] / 15000000;
6    //CR[0] contient le nombre de tics d'horloge depuis la remise a 0 du
7    //timer, c'est a dire depuis le front montant d'Echo
8    distance = (delta_t*34000*)/2;
9    sprintf(temps, "Distance :  %d cm", distance);
10   lcd_puts(temps);
11   lcd_gohome();
12   LPC_CTIMER0->IR |= (1 << 4); //On abaisse le drapeau d'interruption
13 }
```

Le code suivant a été implémenté pour tester le fonctionnement général du programme en remplaçant le signal *Echo* par un appui sur le BP1. On peut alors mesurer le temps d'appui sur ce bouton grâce au mode Capture. Après test, ce programme fonctionne très bien.

```
void CTIMER0_IRQHandler(void) {
  int delta_t = 0;
  char temps[32];
  delta_t = LPC_CTIMER0->CR[0] / 15000000;
  sprintf(temps, "Temps :  %d ms", delta_t);
  lcd_puts(temps);
  lcd_gohome();
  LPC_CTIMER0->IR |= (1 << 4);
}

int main(void) {

  //Activation du périphérique d'entrées/sorties TOR
  LPC_SYSCON->SYSAHBCLKCTRL0 |= GPIO | CTIMER0 | SWM | GPIO_INT | MRT;

  // Configuration de l'horloge a sa frequence maximale
  LPC_PWRD_API->set_fro_frequency(30000);

  // Configuration de la capture de la duree de l'onde recue
  LPC_CTIMER0->TCR = (1 << CEN);

  LPC_CTIMER0->PR = 0;

  LPC_SWM->PINASSIGN3 &= ~(0xFF00);
  LPC_SWM->PINASSIGN3 |= (13 << 8); // Le capture se fait vis à vis du BP1

  LPC_CTIMER0->CTCR |= (1 << ENCC) | (1 << SELCC); // Remise à zéro du ↩
     timer lors d'un front descendant
  LPC_CTIMER0->CCR |= (0b101 << 0); // On capture sur front montant

  NVIC_EnableIRQ(CTIMER0_IRQn);

  init_lcd();

  while(1){

  }
  return 0;

}
```

La mise en commun des deux programmes émission/réception pose cependant problème. En effet, la capture ne se fait plus dès lors que l'émission du signal *Trig* est implémentée. J'ai pu le constater en utilisant de nouveau mon programme test avec le BP1 simulant le signal *Echo*.

Ce problème a été résolu en espaçant davantage les interruptions générant la PWM d'envoi du signal d'impulsion. En effet, une interruption toutes les microsecondes ne permettait pas au programme d'exécuter d'autres tâches (comme la Capture...). En espaçant les interruptions de 10 microsecondes, le problème a disparu.

On obtient finalement le programme complet suivant :

```
#include <cr_section_macros.h>
```

```c
#include <stdio.h>
#include "LPC8xx.h"
#include <stdint.h>
#include "fro.h"
#include "rom_api.h"
#include "syscon.h"
#include "swm.h"
#include "i2c.h"
#include "ctimer.h"
#include "core_cm0plus.h"
#include "dac.h"
#include "lib_ENS_II1_lcd.h"
#include "mrt.h"

#define TRIG LPC_GPIO_PORT->B0[17]
#define ECHO LPC_GPIO_PORT->B0[16]
#define M1SR LPC_GPIO_PORT->B0[14]
#define M1PWM LPC_GPIO_PORT->B0[15]
#define M2SR LPC_GPIO_PORT->B0[18]
#define M2PWM LPC_GPIO_PORT->B0[20]

#define borne_mrt0_bas 6000
#define borne_mrt0_haut 6001

uint32_t compteur_mrt0=0;

void MRT_IRQHandler(void){
  if ((LPC_MRT->Channel[0].STAT & 1) == 1){ //Interruption due au MRT0
    if (TRIG == 0 && compteur_mrt0 < borne_mrt0_bas){
      compteur_mrt0++;
    }
    else if (TRIG == 0 && compteur_mrt0 >= borne_mrt0_bas){
      compteur_mrt0++;
      TRIG=1;
    }
    else if (TRIG == 1 && compteur_mrt0 < borne_mrt0_haut){
      compteur_mrt0++;
    }
    else if (TRIG == 1 && compteur_mrt0 >= borne_mrt0_haut){
      compteur_mrt0=0;
      TRIG=0;
    }
    LPC_MRT->Channel[0].STAT |= 1; //On pense a abaisser le drapeau d'↩
       interruption
  }
}

void CTIMER0_IRQHandler(void) {
  int delta_t = 0;
  int distance = 0;
  char temps[32];
  delta_t = (int) LPC_CTIMER0->CR[0] / 15000000;
  distance = (int)(delta_t*34000)/2;
  sprintf(temps, "Distance :  %d cm", distance);
  lcd_puts(temps);
  lcd_gohome();
  LPC_CTIMER0->IR |= (1 << 4);
}

int main(void) {
```

```
61
62    //Activation des peripheriques
63    LPC_SYSCON->SYSAHBCLKCTRL0 |= GPIO | CTIMER0 | SWM | GPIO_INT | MRT;
64
65    //Configuration en sortie des broches P0_14, 17, 18 et 20
66    LPC_GPIO_PORT->DIR0 |= (1 << 17) | (1 << 14) | (1 << 18) | (1 << 20);
67
68    // Configuration de l'horloge a sa frequence maximale
69    LPC_PWRD_API->set_fro_frequency(30000);
70
71    // PWM envoyant les impulsions pour les salves d'ultrasons
72    LPC_MRT->Channel[0].CTRL |= (1<<0); //Active l'interruption du MRT0
73    LPC_MRT->Channel[0].CTRL |= (0<<1); //Active le mode a interruptions ↩
          repetees
74
75    NVIC_EnableIRQ(MRT_IRQn);  //Active les interruptions du MRT
76
77    LPC_MRT->Channel[0].INTVAL = 150; //La valeur du timer du MRT0 diminue ↩
          chaque dix microsecondes
78
79
80
81    // Configuration de la capture de la duree de l'onde recue
82    LPC_CTIMER0->TCR = (1 << CEN);
83
84    LPC_CTIMER0->PR = 0;
85
86    LPC_SWM->PINASSIGN3 &= ~(0xFF00);
87    LPC_SWM->PINASSIGN3 |= (16 << 8); // Le capture se fait vis à vis d'Echo
88
89    LPC_CTIMER0->CTCR |= (1 << ENCC) | (0 << SELCC); // Remise à zéro du ↩
          timer lors d'un front montant
90    LPC_CTIMER0->CCR |= (0b011 << 0); // On capture sur front descendant
91
92    NVIC_EnableIRQ(CTIMER0_IRQn);
93
94    init_lcd();
95
96    while(1){
97
98    }
99    return 0;
100
101 }
```

1.3 Câblage du capteur

Le câblage du capteur à la LPC804 retenu est le suivant :

- Signal *Trig* sur PIO0_17
- Signal *Echo* sur PIO0_16
- Masse sur celle de CN4
- Alimentation de 3.3V proposée par CN4

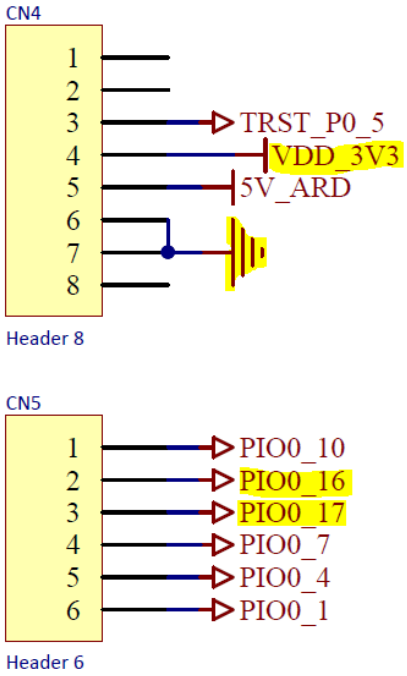

FIGURE 6 – Broches utilisées pour le câblage du capteur ultrason surlignées en jaune

Après branchement sur l'oscilloscope, on a pu constater le bon fonctionnement du capteur : on note l'émission d'une impulsion brève correspondant au signal *Trig* et une autre impulsion brève correspondant au signal *Echo*.

Cependant, la largeur du signal de réception ne change pas selon la distance séparant le capteur du premier obstacle. Un autre capteur du même modèle a été testé avec les mêmes observations. Ainsi, lorsque le programme est implémenté, l'écran LCD affiche systématiquement une distance de 0cm car la largeur de signal *Echo* est très faible (de l'ordre de 50 microsecondes).

2 Déplacement du robot

Nous allons ici programmer le déplacement du robot maintenant que nous connaissons la distance le séparant de son obstacle frontal.

Toutes les fonctions présentées ci-dessous ont été codées pour la forme. Elles n'ont en effet pas pu être implémentées sur le robot car je n'ai pas réussi à le faire fonctionner même avec l'aide de mon encadrant. Toutefois, il est intéressant d'essayer de coder de manière théorique le déplacement autonome de ce robot.

Le schéma de principe du déplacement autonome lorsqu'un obstacle est détecté est présenté à la figure 7.

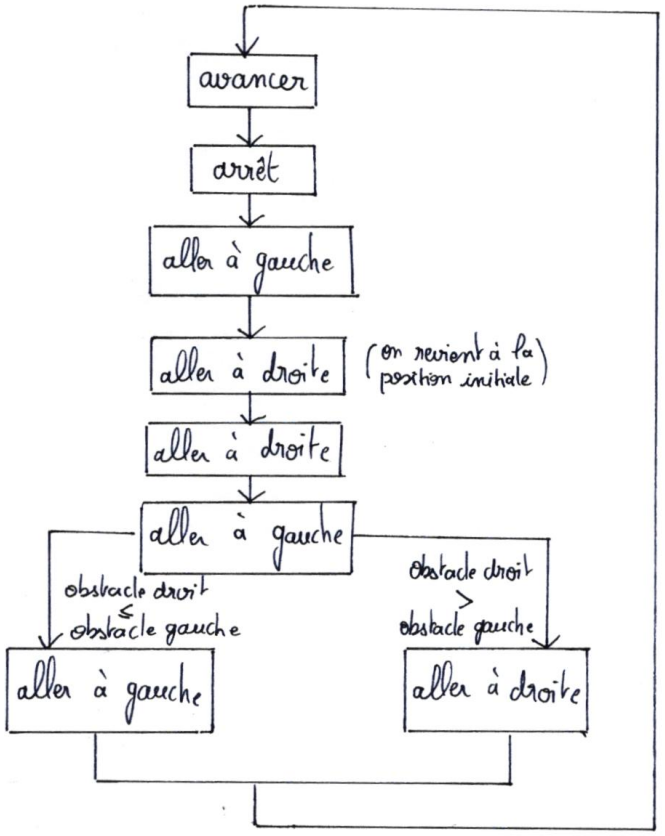

FIGURE 7 – Schéma présentant le principe du déplacement autonome lorsqu'un obstacle est détecté

2.1 Détection de l'obstacle

Nous définissons dans un premier temps la fonction *obstacle()* qui nous indique si le robot se situe en face d'un obstacle ou non.

Pour ce faire, nous utilisons bien évidemment la distance calculée par le mode Capture comme vu précédemment. Cette valeur est placée maintenant dans une variable globale pour pouvoir être accessible plus facilement.

On définit également une tolérance à la détection d'un obstacle. En effet, le HC-SR04 peut en théorie détecter des objets situés à 4m. Sans tolérance, le robot risquerait de faire du surplace. On note *d_lim* cette tolérance dont la valeur sera définie par un *#define*.

On a alors la fonction ci-dessous :

```
int obstacle(void) {
  if (distance < d_lim) {
    return obst_det;
  } else {
    return no_obst;
  }
}
```

On pourra également attribuer une valeur à *obst_det* et *no_obst* grâce à un *#define*.

2.2 Déplacement dans une direction

Intéressons-nous maintenant à la fonction définissant le sens de déplacement du robot, fonction sobrement appelée *deplacement()*.

Les moteurs de chaque roue sont alimentées par des PWM (définies par deux timers MRT distincts de la même manière que la PWM générant le signal *Trig* du capteur). On considère ici que le rapport cyclique de ces PWM est constant tout au long de l'utilisation du robot, et donc que la vitesse des roues est unique. On considère que le MRT1 est le timer MRT définissant les PWM utilisées par les moteurs des roues.

Les variables *M1SR* et *M2SR* qui définissent le sens de rotation de chacune des roues sont les variables permettant de changer l'orientation du véhicule.

La fonction *deplacement()* se contente donc de changer le sens de rotation des roues :

```
void deplacement(char direction) {

  switch (direction) {

  case "avant":
    M1SR=avant;
    M2SR=arriere;
    break;

    case "gauche":
    M1SR=arriere;
    M2SR=avant;
    break;

    case "droite":
    M1SR=avant;
    M2SR=arriere;
    break;
  }

}
```

Les variables *avant* et *arriere* sont définies via un *#define* de manière à être en accord avec le fonctionnement du robot.

2.3 Fonctions intermédiaires

D'autres fonctions sont nécessaires au déplacement autonome du robot.

Tout d'abord, il faut mettre le robot à l'arrêt lorsqu'un obstacle est détecté ou avant une mesure de distance. C'est ce que réalise la fonction *arret()* en arrêtant le timer MRT1 et en mettant à 0 les signaux alimentant les deux moteurs :

```
void arret(void) {
  LPC_MRT->Channel[1].INTVAL = 0;
  // On stop la generation des PWM
  M1PWM=0;
  M2PWM=0;
  // On pense a mettre les PWM a 0, sinon elle pourrait
  // potentiellement rester tout le temps a 1
}
```

Il faut ensuite évidemment une fonction qui permette au robot de continuer à avancer après un arrêt. C'est le rôle de la fonction *reprise* qui ne consiste qu'en la remise en marche du timer MRT1 :

```
void reprise(void) {
  LPC_MRT->Channel[1].INTVAL = 1;
  // Remise en route du MRT1 definissant les PWM des moteurs
}
```

Enfin, lorsque l'on souhaite faire pivoter le robot sur la gauche ou sur la droite, il faut définir une période d'attente durant laquelle le robot va se déplacer. Cette fonction *attente* consiste en une simple boucle *for* :

```
void attente(void) {
  uint32_t i;
  for (i = 0; i < 10000; i++) {
  }
}
```

Le nombre d'itérations (10000 ici) sera réglé empiriquement selon la vitesse des roues désirée.

2.4 Déplacement autonome

On utilise enfin les fonctions précédemment évoquées dans la boucle infinie pour permettre au robot de se déplacer de façon autonome. Nous suivons le principe décrit sur le schéma de la figure 7.

```
int main(void) {
  while (1) {

    uint32_t d_gauche;
    uint32_t d_droite;

    etat_capteur = obstacle();

    if (etat_capteur == obst_det) {

      arret();

      reprise();
      deplacement("gauche");
      attente();
```

```
16        arret();
17        d_gauche = distance;
18
19        reprise();
20        deplacement("droite");
21        attente();
22
23        deplacement("droite");
24        attente();
25        arret();
26        d_droite = distance;
27        reprise();
28        deplacement("gauche");
29        attente();
30        arret();
31
32        if (d_droite < d_gauche) {
33          reprise();
34          deplacement("gauche");
35          attente();
36
37        }
38        else {
39          reprise();
40          deplacement("droite");
41          attente();
42        }
43        deplacement("avant");
44      }
45    }
46  }
```

2.5 Câblage du robot

Pour les signaux PWM et de sens de rotation des roues, de simples ports Entrée/Sortie peuvent être utilisés. Ont été choisis :

- Signal M1SR sur PIO0_14
- Signal M2SR sur PIO0_18
- M1PWM sur PIO0_15
- M2PWM sur PIO0_20

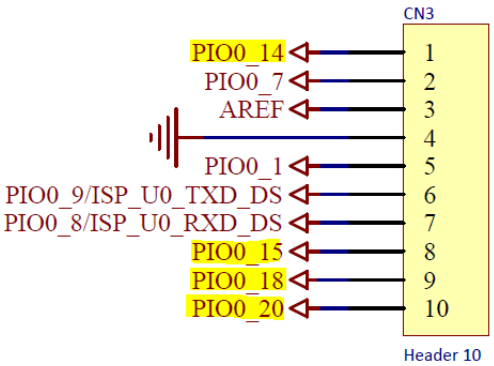

FIGURE 8 – Broches utilisées pour le câblage des moteurs surlignées en jaune