Systèmes Mécatroniques
Section de génie mécanique

Mini Projet 3 : Régulation de vitesse d'un moteur à courant continu

Enseignant :

Kossi Agbeviade

Étudiants :

Colomban Fussinger
André Gomes
Nicolas Lefebure

3 juillet 2019

Table des matières

1 Introduction

Ce TP a pour but d'initier au fonctionnement d'un microcontrôleur grâce à la réalisation d'une commande en vitesse d'un moteur à courant continu. Différentes fonctions seront étudiées comme la mesure d'une fréquence, la lecture d'une grandeur analogique ou encore la réalisation d'un signal PWM. Les commandes du moteur peuvent également être surveillées depuis un ordinateur grâce à une communication UART. Ainsi, dans ce rapport, on présentera rapidement le matériel utilisé puis on s'attardera sur l'aspect technique du code pour la mise en oeuvre des fonctionnalités citées.

2 Montage du dispositif

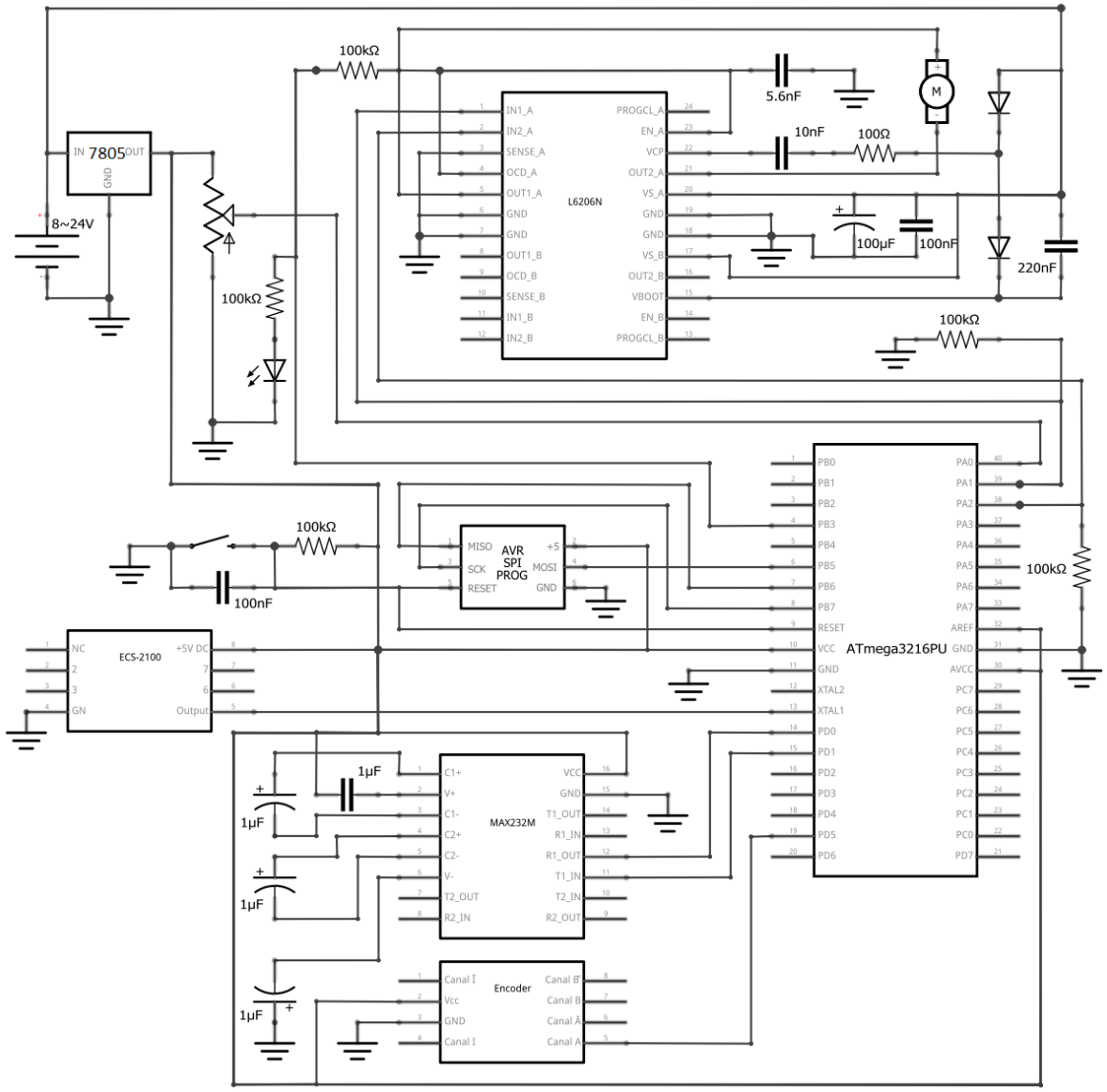

FIGURE 1 – Schéma électrique du montage utilisé.

Le montage électronique présenté à la figure 1 montre plusieurs composants. On trouve en premier le convertisseur de tension KA7805 qui adapte la tension de l'alimentation en une tension de 5 [V] compatible avec tous les circuits électroniques présents dans ce montage. Un dissipateur est placé sur ce composant car étant un convertisseur d'énergie, il a tendance à en dissiper beaucoup. On trouve également le composant ECS-2100 qui crée un signal d'horloge à 14.7456 [MHz] pour le microcontrôleur ATmega32. On utilise une source d'horloge externe (implicitement choisie en laissant les 4 premiers bits du registre CKSEL à 0) car le microcontrôleur dispose d'un système interne avec un circuit RC qui manque de précision. Le microcontrôleur peut être programmé à l'aide du programmateur AVR SPI qui est connecté en USB à un ordinateur personnel disposant du logiciel AtmelStudio 7. On peut également échanger des données avec le microcontrôleur grâce au logiciel TeraTerm et au convertisseur MAX232M (USART). Ce composant permet d'adapter les niveaux de tension entre les niveaux TTL du microcontrôleur et ceux de la communication RS-232. La communication RS-232 peut être convertie une nouvelle fois en USB pour être lisible par un ordinateur moderne. Le montage bénéficie d'un bouton de remise à zéro qui permet de relancer le microcontrôleur en cas de besoin. Un potentiomètre constitue une entrée analogique sur le montage. Enfin, un moteur à courant continu est alimenté par un pont en H intégré dans le boîtier L6206N (qui contient en fait deux ponts en H pour le contrôle de deux moteurs distincts, mais nous utilisons ici que le pont A). Ce boîtier est alimenté directement par la source d'alimentation principale afin de bénéficier de plus de puissance pour le moteur. Cette tension étant plus élevée que celle prévue pour le microcontrôleur, il convient d'utiliser un système de bootstrap afin d'assurer une alimentation de la base du transistor suffisamment élevée pour le transistor du haut des branches du pont.

3 Consigne de commande et convertisseur ADC

3.1 Objectif

La consigne de commande du moteur se fait à l'aide d'un potentiomètre. Afin d'améliorer le comportement du moteur, il est nécessaire d'implémenter un contrôleur, fait ici de manière numérique par le microcontrôleur. Il est dès lors nécessaire de convertir la consigne analogique donnée par le potentiomètre en une variable numérique. A cet effet, le convertisseur analogique-numérique du microprocesseur est utilisé.

3.2 Implémentation

Le convertisseur est d'abord mis en marche par le bit ADEN du registre ADCSRA mis à 1 (Code C en figure 2). Il s'agit ensuite de définir le pré-diviseur sur la fréquence du CPU afin de produire une fréquence d'horloge acceptable pour l'ADC. Le circuit d'approximation successif fournit une résolution maximale pour une fréquence comprise entre 50 et 200 [kHz]. Avec une fréquence d'échantillonnage plus élevée la résolution diminue. Le choix du pré-diviseur se fait avec la configuration des bits ADPS2 à ADPS0 du même registre. Dans le cas présent, avec un fréquence d'horloge d'environ 15 [MHz] et un pré-diviseur de 128, le fréquence de conversion est ainsi proche de 100 [kHz], ce qui est optimal pour la résolution et amplement suffisant pour la commande du moteur.

Le convertisseur de l'ATmega32 peut convertir directement l'entrée sur un des 8 canaux ADC (présents uniquement sur le PORT A) ou convertir la différence entre deux canaux. Il

est également possible d'y ajouter un gain. Le choix des canaux à convertir et le gain associé se fait à l'aide des bits MUX4 à MUX0 du registre ADMUX.

La conversion d'une tension analogique en une valeur numérique codée sur 10 bits se base sur une tension de référence et la masse. La valeur minimum (= 0), représente un voltage nul, la masse du microcontrôleur, alors que la valeur maximale (= 1023), est donnée par le voltage de référence. Celui-ci peut être externe (comme dans notre cas), lu sur la broche AREF du microcontrôleur, ou interne (5 [V] ou 2.56 [V]). Le choix de la référence se fait avec les bits REFS1 et REFS0 du registre ADMUX.

Le convertisseur a sa propre interruption (fanion ADIF), qui peut être activée par le bit ADIE du registre ADCSRA. Elle est déclenchée à la fin de chaque conversion. La routine consiste à attendre la fin de la conversion, récupérer la valeur et relancer une nouvelle conversion. Le fanion ADIF est automatiquement réinitialisé à cet instant.

```
volatile unsigned int potIn=0; // Mesure potentiomètre par ADC
uint16_t yc;            // Valeur de consigne (potentiomètre) [0-255]

// Single Ended Input ADC0
ADMUX|=(0<<MUX4)|(0<<MUX3)|(0<<MUX2)|(0<<MUX1)|(0<<MUX0);
// ADC registres de données: ajusté à gauche (8bits dans ADCH, 2 dans ADCL)
ADMUX|=(1<<ADLAR);
// Voltage: AVCC (broche 30) avec capacité externe sur la broche AREF
ADMUX|=(1<<REFS0) | (0<<REFS1);
// ADC activé, interruption autorisée, pré-diviseur = 128
ADCSRA|=(1<<ADEN)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
ADCSRA|=(1<<ADSC); // Démarre la conversion

int main(void) {
    sei(); // Autorise les interruptions
    while(1) {
        yc = potIn>>10;          // Consigne (potentiomètre) [0-255]
    }
}

/* Routine d'interruption de fin de conversion*/
ISR(ADC_vect) {
    while(ADCSRA & (1<<ADSC)); // Attente que la conversion soit terminée
    unsigned char adcl = ADCL;
    unsigned char adch = ADCH;
    potIn = (adch<<8) | adcl; // Récupère la valeur
    ADCSRA|=(1<<ADSC); // Relance une nouvelle conversion
}
```

FIGURE 2 – Code C pour le convertisseur ADC.

La lecture du résultat se fait alors sur les registres ADCH et ADCL. Le résultat numérique est donné par :

$$Resultat_{ADC} = \frac{U_{in}}{U_{ref}} \cdot 1024 - 1 \tag{1}$$

4 Lecture de vitesse par fréquencemètre

4.1 Objectif

Pour la commande en vitesse à l'aide du contrôleur PID, il est nécessaire de connaître aussi précisément que possible la vitesse actuelle du moteur. A cette fin, le microcontrôleur est utilisé comme fréquencemètre.

La lecture de la vitesse du moteur se fait à l'aide d'un encodeur à 500 impulsions par tour. A chaque impulsion correspond un front montant du signal carré qui est envoyé au microprocesseur. Le principe étant que le microcontrôleur mesure le temps entre chaque impulsion de l'encodeur et en déduise une vitesse de rotation.

4.2 Implémentation

Le registre 16 bit TCNT1 (du TIMER1) est incrémenté sur 65536 valeurs à chaque coup d'horloge du CPU (pré-diviseur de 1). Lors de l'overflow, une interruption est générée. Le compteur d'overflow est alors incrémenté et le timer remis à 0 (Code C en figure 3).

Lorsqu'un front montant est détecté sur la broche PD6, une interruption est générée. La valeur du compteur d'overflow est mémorisée, ainsi que le registre ICR1, qui contient la valeur du registre TCNT1 au moment de l'interruption.

Dans la boucle principal, la vitesse de rotation peut alors être calculée :

$$N_{moteur} = \frac{f_{CPU}}{N_{ipt} \cdot (n_{OVF} \cdot 65536 + res)} \text{ [tour/s]} \tag{2}$$

avec $N_{ipt} = 500$, le nombre d'impulsions de l'encodeur par tour du moteur, n_{OVF}, le nombre d'overflow du registre entre deux impulsions et res son résidu lors de l'interruption.

Le choix du registre 16 bits plutôt que 8 bits ainsi que du pré-diviseur n'est pas arbitraire. L'objectif est de limiter le nombre d'interruptions afin de diminuer le nombre d'instructions et ainsi être plus précis sur la fréquence mesurée. Il faut toutefois faire attention à ne pas descendre trop bas, sinon le nombre d'incrémentations du registre n'est plus suffisant pour avoir une bonne résolution. Dans la même optique, limiter le nombre d'instructions au sein même des interruptions en effectuant le traitement des valeurs enregistrées en dehors de l'interruption est préférable.

Dans le cas présent, le moteur atteint une vitesse maximale de 50 [tours/s]. Puisqu'il y a 500 impulsions par tour, le fréquence d'interruption sera de $f_{imp} = 25000$ [Hz]. Avec le registre 16 bits et un pré-diviseur à 1, il ne devrait y avoir dans ce cas aucun overflow, puisque sa fréquence d'overflow n'est que de $\frac{f_{CPU}}{65536} = 225$ [Hz]. Le nombre d'incrémentations minimal du registre sera donc de $\frac{f_{CPU}}{f_{imp}} = 589$. Cela garantit ainsi une bonne résolution même à vitesse maximale tout en limitant le nombre d'interruptions à basse vitesse.

```c
uint16_t cptOVF;        // Compteur d'overflow du timer
uint16_t nbOVF;         // Valeur compteur lors du front montant
uint16_t res;           // Résidu timer lors du  front montant
uint16_t y;             // Vitesse moteur mesurée [tour/s]

DDRD &=~ (1<<DDD6);            // Broche PD6 mis en entrée (lecture signal de l'encodeur)
TCCR1A = 0x00;  TCCR1B = 0x00; // Initialise les 8 bits bas et haut à 0
// Pré-diviseur=1 (CS10), front de détection montant (ICES), suppresseur de bruit (ICNC1)
TCCR1B |= (1<<CS10) | (1<<ICES1) | (1<<ICNC1);
// Autorise interruption sur overflow (TOIE1) and front montant sur entrée (TICIE1)
TIMSK |= (1<<TOIE1) | (1<<TICIE1);

int main(void) {
    sei(); // Autorise les interruptions
    while(1) {
        y = F_CPU/(500*(nbOVF*65536+res)); // Vitesse moteur mesurée [tour/s]
    }
}

/* Timer ISR */
ISR(TIMER1_OVF_vect) {
    cptOVF++; TCNT1=0;
}

/* Flanc montant sur l'entrée ISR */
ISR(TIMER1_CAPT_vect) {
    res = ICR1; nbOVF = cptOVF;
    TCNT1 = 0; cptOVF = 0;
}
```

FIGURE 3 – Code C pour la lecture de la vitesse du moteur.

5 Contrôleur PID

5.1 Implémentation

Un contrôleur PID numérique est implémenté pour améliorer le contrôle en vitesse du moteur (Code C en figure 4). Les gains proportionnel, intégrateur et dérivateur sont obtenus par expérimentation, avec pour objectif une erreur nulle en régime permanent et un système stable quelle que soit la consigne donnée par le potentiomètre. Plusieurs conditions sont également implémentées. Comme le moteur n'est commandé que dans une seule direction, il n'est pas envisageable de le ralentir en lui imposant une consigne négative. Aussi, afin d'éviter la saturation de l'intégrateur, celui-ci est limité à la valeur maximale admissible pour le PWM.

```c
// Déclaration des variables
uint16_t yc; uint16_t y; int32_t e;          // Commande, mesure et erreur
uint8_t Kp=1; uint8_t Ki=3; uint8_t Kd=2;    // Gains
int32_t ui; int32_t ui0;                     // Actuelle et ancienne consigne intégrateur
int32_t ud; int32_t ud0;
int32_t u;                                   // Consigne finale

int main(void) {
    while(1) {
        y = F_CPU/(500*(nbOVF*65535+res));  // Mesure fréq moteur
        yc = potIn>>10;                      // Consigne (potentiomètre) [0-255]
        e =  yc - y;                         // Erreur

        ui = ui0 + Ki*(e + e0);              // Consigne intégrateur
        if (Kp*ui>254){ui=255/Kp;}
        if (Kp*ui<-254){ui=-255;}
        ud = -ud0 + Kd*(e-e0);               // Consigne dérivateur
        if (Kp*ud>254){ud=255/Kp;}
        if (Kp*ud<-254){ud=-255;}
        u = Kp*(e+ui+ud);                    // Consigne finale
        if (u>254){u=255;}
        if (u<0){u=0;}

        e0 = e; ui0 = ui; ud0 = ud;  // Mise à jour des variables
    }
}
```

FIGURE 4 – Code C pour le contrôleur PID.

6 Commande du sens de rotation et de vitesse par PWM

6.1 Fonctionnement

Le PWM (en français : modulation de largeur d'impulsion) est la technique utilisée ici pour contrôler le moteur à courant continu (Code C en figure 7). En imposant une largeur d'impulsion, la puissance délivrée au moteur est contrôlée. Ceci se fait avec le registre de comparaison OCR0, un registre 8 bit. Avec des variables entières signées, les valeurs possibles sont donc [0, 1, ..., 255]. Ainsi, la tension moyenne du port dédié est :

$$V_{out} = \frac{0\text{CR0}}{255} \cdot 5 \text{ [V]} \tag{3}$$

La fréquence de PWM est donnée par :

$$f_{PWM} = \frac{f_{CPU}}{N \cdot 256} = 7200 \text{ [Hz]} \tag{4}$$

avec $f_{CPU} = 14.7456$ [MHz] la fréquence du CPU et $N = 8$ le pré-diviseur.

Puisque un cycle est très court en comparaison à la constante de temps du moteur, la puissance reçue par ce dernier semble continue et correspond à la moyenne du signal modulé (figure 5).

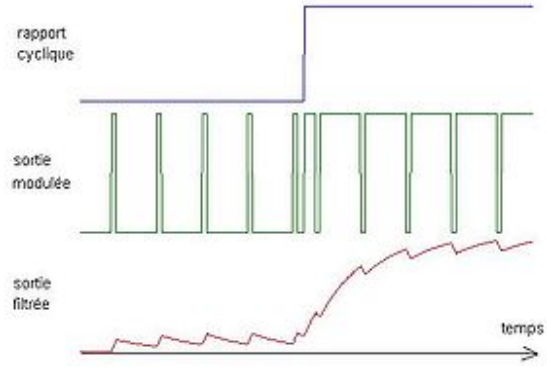

FIGURE 5 – PWM. [2]

6.2 Implémentation

Le choix du sens de rotation se fait en choisissant quels sont les transistors activés sur le pont en H. Pour ce faire, il faut commuter un transistor de la branche 1 du pont et l'opposé sur la branche 2 (voir la figure 6). Ainsi, le moteur a à ses bornes la tension d'alimentation du système (à la tension collecteur-émetteur des transistors près). Le choix du transistor se fait en mettant ou non une tension sur l'entrée IN1_A du composant L6206N pour la branche 1 et IN2_A pour la branche 2. Dans le code de la figure 7, le sens de rotation est fixe et est réalisé en mettant la sortie PA1 à l'état haut (le registre DDRA définit la direction de la patte correspondante et le registre PORTA l'état de la sortie). On notera d'ailleurs sur la figure 1 que les sorties correspondant au choix des transistors sur les branches disposent d'une résistance de pull-down afin d'éviter un contrôle erratique une fois la sortie désactivée (niveau de tension mal défini sans la résistance de pull-down). La gestion du PWM se fait grâce à l'entrée EN_A qui constitue en quelque sorte un interrupteur général sur la commande des transistors. La sortie PB3 du microcontrôleur commandant cette fonction dispose également d'une LED permettant un contrôle visuel de la commande PWM envoyée. Le tableau de commande du moteur est visible à la table 1.

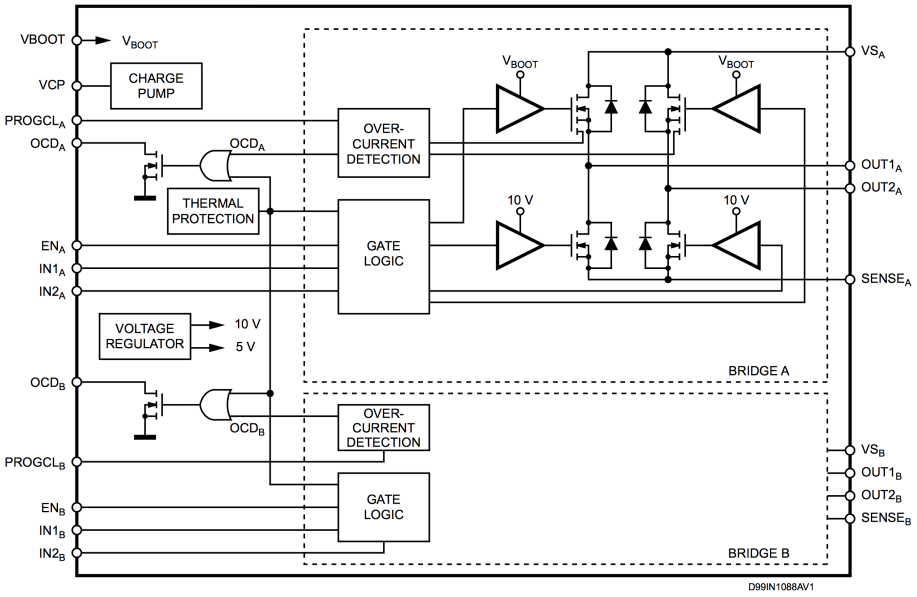

FIGURE 6 – Schéma bloc du composant L6206N. [3]

IN1_A	IN2_A	EN_A	PIN5	PIN21	Etat moteur
-	-	0	flottant	flottant	Libre
0	0	1	GND	GND	Frein
0	1	1	GND	V_{in}	Rotation sens 1
1	0	1	V_{in}	GND	Rotation sens 2
1	1	1	V_{in}	V_{in}	Frein

TABLE 1 – Table de vérité du pont en H (composant L6206N).

La programmation du PWM peut se faire selon deux approches. La première consiste à utiliser des fonctions d'attente (`_delay_ms()`) dans une boucle infinie du programme principal qui gèrent directement la mise à l'état haut et bas de la broche. Cette approche-là a le défaut de monopoliser le processeur pendant les phases d'attente, ce qui limite énormément les tâches annexes que peut effectuer le processeur. La deuxième solution qui permet de contourner ce défaut est l'utilisation des interruptions liées à un timer du microcontrôleur. L'ATmega32 dispose par exemple d'une architecture faite spécialement pour la programmation de PWM basée sur ce principe-là et est implémentée exclusivement sur la broche PB3 que nous utilisons pour l'activation générale du moteur.

```
DDRB |= (1<<PB3); // Mise en sortie de la broche de commande de la tension (PWM)
DDRA |= (1<<PA1); // Mise en sortie de la broche de commande du sens de rotation du moteur
// WGM: Fast PWM, COM: non-inversé, CS: pré-diviseur = 8
TCCR0 |=(1<<WGM00)|(1<<WGM01)|(0<<COM00)|(1<<COM01)|(1<<CS01);

void pwmControl(uImp) {                // Fonction de contrôle du PWM intégré
    PORTA &= (0<<PA2); PORTA |= (1<<PA1);   // Commande du sens de rotation
    OCR0 = uImp;                         // Commande de la largeur d'impulsion [0-255]
}

int main(void) {
    while(1) {
        pwmControl(u); // Appel de la fonction de contrôle avec la consigne u
    }
}
```

FIGURE 7 – Code C pour le PWM. Le moteur n'est contrôlé que dans un sens.

7 Suivi en temps réel et commande par UART

7.1 Objectif

Il est utile d'avoir accès, en temps réel, à la vitesse mesurée du moteur, aux différents paramètres du contrôleur ou encore de pouvoir en changer les valeurs si cela est souhaité. Pour cela une communication bi-directionnelle entre le microcontrôleur et le PC doit être établie.

7.2 Implémentation

Du côté microcontrôleur, il est nécessaire d'initialiser les ports de communications UART, de définir le Baud Rate (vitesse de communication de la ligne sériel [bits/s]), le registre de lecture/écriture et éventuellement d'implémenter les interruptions. A noter que pour transmettre des variables correspondant à des nombres (entiers, flottants, etc), il faut au préalable les convertir en chaîne de caractères avec la commande `itoa` par exemple. Pour la lecture de la ligne sériel, deux options sont possibles, soit faire du pooling dans la boucle principale, soit utiliser les interruptions disponibles pour l'UART. La gestion des cycles de lecture/écriture doit être bien établie pour ne pas perdre des données.

Le choix du Baud Rate dépend principalement de l'application, si une quantité importante de données doit être transférée ou si un long traitement doit être effectué dessus, la vitesse nécessaire ne sera pas la même.

Du côté PC, il faut s'assurer que le logiciel de lecture soit initialisé sur le bon port USB et que le Baud Rate soit le même que celui programmé sur le microcontrôleur.

```c
#define UART_BAUD_RATE 115200
// Asynchrone Normal Mode (U2X = 0)
#define UART_BAUD_CALC (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)

uint8_t i = 1;
unsigned char buffer[100];

uart_init(UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU)); // Initialise UART et défini Baud Rate.

UCSRB = (1 << TXEN)|(1 << RXEN); // Active transmission et réception
//  UCSZ: caractère de taille 8 bit, URSEL: sélection du registre
UCSRC = (1 << URSEL) | (1 << UCSZ0) | (1 << UCSZ1);
UBRRH = (UART_BAUD_CALC >> 8); UBRRL = UART_BAUD_CALC; // Registres USART Baud Rate

int main(void) {
    while(1) {
        i=0;
        while (i<100 && (UCSRA = (1 << RXC))) {
            buffer[i] = uart_getc();
            i++;
        }
        uart_puts(buffer);
        uart_putc('\n');
        _delay_ms(200);
    }
}
```

FIGURE 8 – Code C pour l'UART. Communication bi-directionnelle par pooling. Le microcontrôleur lit et renvoie les caractères reçus. Un délai est utilisé pour ne pas saturer visuellement l'interface utilisateur.

8 Conclusion

Réaliser une commande numérique précise d'un moteur à courant continu demande d'implémenter une grande variété de fonctions. Par la programmation directe des registres, une bonne souplesse et personnalisation de l'application est ainsi possible. Cette approche permet également d'avoir un excellent aperçu des possibilités offertes par un microcontrôleur et du circuit électrique dédié. Finalement, un tel travail est aussi une bonne introduction à la lecture de documentations des fournisseurs de composants, ce qui au premier abord n'est pas toujours évident.

D'un point de vue technique, ce travail met en évidence l'importance d'utiliser les outils adéquats suivant la réalisation à effectuer. Le choix du bon registre, d'un pré-diviseur adapté et une programmation efficace permettent ainsi d'obtenir des résultats bien meilleurs que si ces aspects-là sont bâclés.

A Code C

```
#define F_CPU 14745600UL
#define UART_BAUD_RATE 115200
#define UART_BAUD_CALC (((F_CPU / (USART_BAUDRATE * 16UL))) - 1)

#include <avr/io.h>
#include <stdlib.h>
#include <util/delay.h>
#include <avr/pgmspace.h>
#include "avr/interrupt.h"
#include "util/delay.h"
#include "inttypes.h"

/* Déclaration des variables */
volatile unsigned int potIn=0;              // Mesure potentiomètre par ADC
char buffer[5];
uint16_t cptOVF;                            // Compteur d'overflow du timer
uint16_t nbOVF;                             // Valeur compteur lors du front montant
uint16_t res;                               // Résidu timer lors du  front montant
uint16_t yc; uint16_t y; int32_t e;         // Commande, mesure et erreur
uint8_t Kp=1; uint8_t Ki=3; uint8_t Kd=2;   // Gains
int32_t ui; int32_t ui0;                    // Actuelle et ancienne consigne intégrateur
int32_t ud; int32_t ud0;
int32_t u;                                  // Consigne finale

/* Initialisations et fonctions */
void adc_init() {
    // Single Ended Input ADC0
    ADMUX|=(0<<MUX4)|(0<<MUX3)|(0<<MUX2)|(0<<MUX1)|(0<<MUX0);
    // ADC registres de données: ajusté à gauche (8bits dans ADCH, 2 dans ADCL)
    ADMUX|=(1<<ADLAR);
    // Voltage: AVCC (broche 30) avec capacité externe sur la broche AREF
    ADMUX|=(1<<REFS0) | (0<<REFS1);
    // ADC activé, interruption autorisée, pré-diviseur = 128
    ADCSRA|=(1<<ADEN)|(1<<ADIE)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
    ADCSRA|=(1<<ADSC); // Démarre la conversion
}

void timer_init() {
    DDRD &=~ (1<<DDD6);            // Pin PD6 mis en entrée (lecture signal de l'encodeur)
    TCCR1A = 0x00;  TCCR1B = 0x00; // Initialise les 8 bits bas et haut à 0
    // Pré-diviseur=1 (CS10), front de détection montant (ICES), suppresseur de bruit (ICNC1)
    TCCR1B |= (1<<CS10) | (1<<ICES1) | (1<<ICNC1);
    // Autorise interruption sur overflow (TOIE1) and front montant sur entrée (TICIE1)
    TIMSK |= (1<<TOIE1) | (1<<TICIE1);
}

void pwm_init() {
    DDRB |= (1<<PB3); // Mise en sortie broche de commande de la tension (PWM)
    DDRA |= (1<<PA1); // Mise en sortie broche de commande du sens de rotation du moteur
    // WGM: Fast PWM, COM: non-inversé, CS: pré-diviseur = 8
    TCCR0 |=(1<<WGM00)|(1<<WGM01)|(0<<COM00)|(1<<COM01)|(1<<CS01);
}

void pwmControl(uImp) {                     // Fonction de contrôle du PWM intégré
    PORTA &= (0<<PA2); PORTA |= (1<<PA1);   // Commande du sens de rotation
    OCR0 = uImp;                            // Commande de la largeur d'impulsion [0-255]
}

```

```c
/* MAIN */
int main(void) {
    sei(); // Autorisation globale des interruptions

    uart_init(UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU)); // Initialise UART and défini le Baud Rate
    pwm_init();
    timer_init();
    adc_init();

    while(1) {
        y = F_CPU/(500*(nbOVF*65535+res));  // Vitesse moteur mesurée [tour/s]
        yc = potIn>>10;                     // Consigne (potentiomètre) [0-255]
        e =  yc - y;                        // Erreur

        ui = ui0 + Ki*(e + e0);             // Consigne intégrateur
        if (Kp*ui>254){ui=255/Kp;}
        if (Kp*ui<-254){ui=-255;}
        ud = -ud0 + Kd*(e-e0);              // Consigne dérivateur
        if (Kp*ud>254){ud=255/Kp;}
        if (Kp*ud<-254){ud=-255;}
        u = Kp*(e+ui+ud);                   // Consigne finale
        if (u>254){u=255;}
        if (u<0){u=0;}

        pwmControl(u); // Appel de la fonction de contrôle avec la consigne u

        uart_puts("Consigne [Hz]:");    itoa(yc,buffer,10); uart_puts(buffer);
        uart_puts("   Sortie [Hz]:");   itoa(y,buffer,10); uart_puts(buffer);
        uart_puts("   Erreur [Hz]:");   itoa(e,buffer,10); uart_puts(buffer);
        uart_putc('\n');
        uart_puts("Intégrateur:");      itoa(ui,buffer,10); uart_puts(buffer);
        uart_puts("   Dérivateur:");    itoa(ud,buffer,10); uart_puts(buffer);
        uart_puts("   PWM:");           itoa(u,buffer,10); uart_puts(buffer);
        uart_putc('\n');

        e0 = e; ui0 = ui; ud0 = ud;  // Mise à jour des variables

        _delay_ms(100);
    }
}

/* Routines d'interruption */
ISR(ADC_vect) { // Fin de conversion de l'ADC
    while(ADCSRA & (1<<ADSC));      // Attente que la conversion soit terminée
    unsigned char adcl = ADCL;
    unsigned char adch = ADCH;
    potIn = (adch<<8) | adcl;       // Récupère la valeur
    ADCSRA|=(1<<ADSC);              // Relance une nouvelle conversion
}

ISR(TIMER1_OVF_vect) { // Overflow du compteur
    cptOVF++;
    TCNT1=0;
}

ISR(TIMER1_CAPT_vect) { // Front montant du signal de l'encodeur
    res = ICR1;
    nbOVF = cptOVF;
    TCNT1 = 0;
    cptOVF = 0;
}
```

Références

[1] Atmel
ATmega32 Datasheet

[2] `upload.wikimedia.org/wikipedia/commons/8/80/MLI1.jpg`

[3] ST
L6206 DMOS dual full bridge driver Datasheet