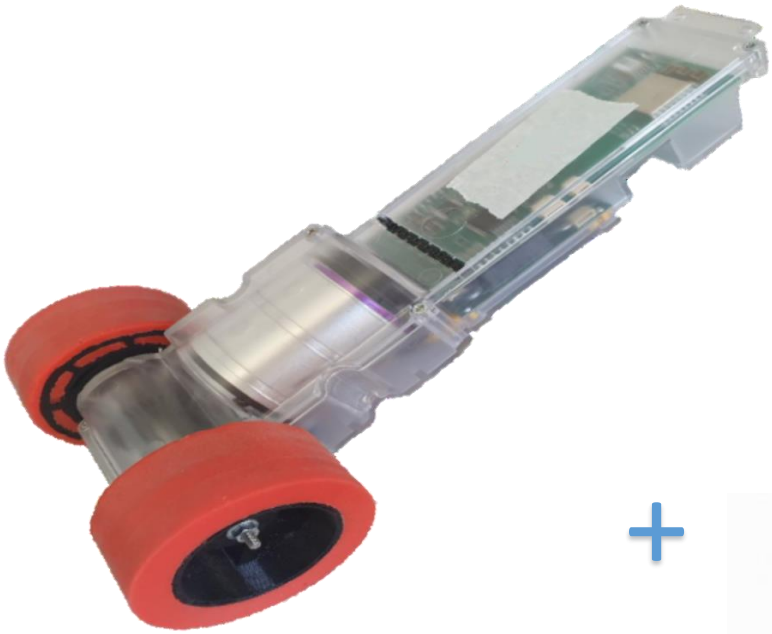
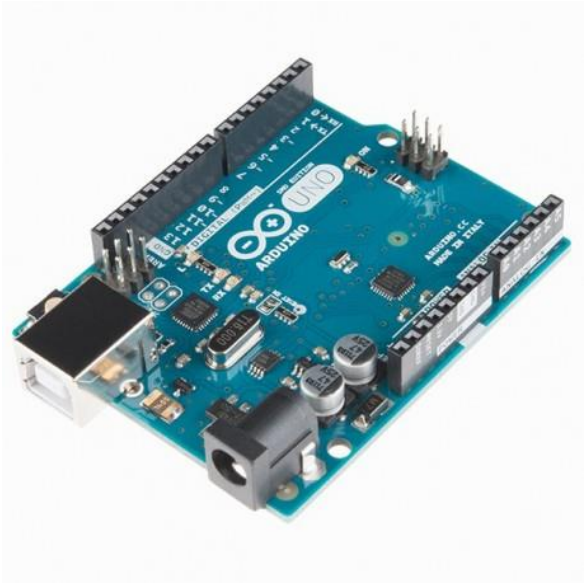
Course en cours :

Tutoriel utilisation d'une carte Arduino – Version 2.0



+



COURSE EN COURS

GRAND PRIX DES COLLÈGES ET LYCÉES

Objectif de ce tutoriel :

Au travers de ce document, vous apprendrez comment utilisez une carte Arduino pour dialoguer avec votre véhicule. Ce Tutoriel ne présente que le fonctionnement de base de la liaison entre votre moteur Course en cours et une carte Arduino, vous serez libre d'ajouter d'autre capteur lié à votre Arduino si vous le souhaitez.

Nous effectuerons des rappels mathématiques et électroniques pour vous permettre de pouvoir mieux comprendre comment s'effectue la liaison entre la carte Arduino et la carte moteur, ainsi que la communication.

Ce tutoriel s'applique à la **librairie Arduino Version 2.0** pour interagir avec le moteur Course en Cours.

Table des matières

I. Présentation de l'électronique

A. Carte mère Course en cours

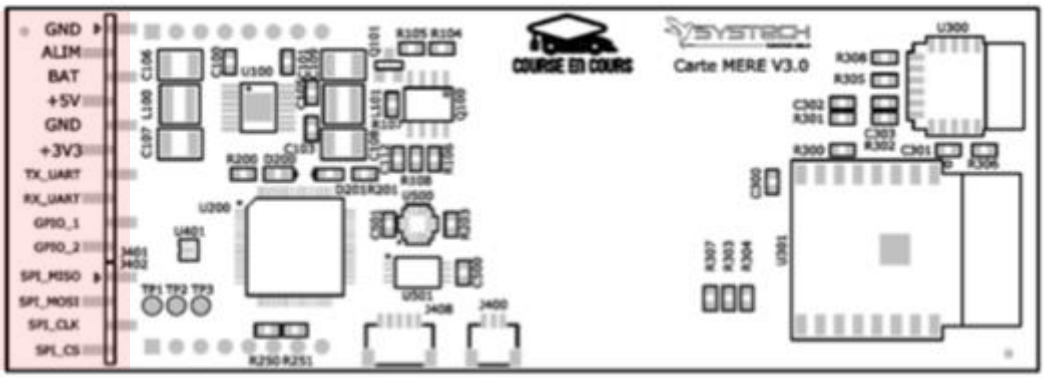


Figure 1: Carte Mère et connecteur Arduino

Le connecteur permettant de connecter un Arduino au moteur est celui à 14 points sur le dessus.

B. Description du connecteur d'interconnexion

Voici la description des signaux disponibles sur le connecteur d'interconnexion avec l'Arduino :

Pin	Nom	Description
1	GND	Masse électrique, 0V ou *Ground*
2	ALIM	ON/OFF de la carte. Logique inversée. ON = GND & ALIM, OFF = NC (non connecté)
3	BAT	Tension Batterie +11,1V / 450mAh : entre 9,6V min et 12,6V max
4	+5V	Alimentation stabilisée +5V / 2A
5	GND	Masse électrique, 0V ou *Ground*
6	+3V3	Alimentation stabilisée +3,3V / 3A
7	TX_UART	Communication UART avec le système : du système vers l'Arduino
8	RX_UART	Communication UART avec le système : de l'Arduino vers le système
9	GPIO_1	*Non utilisé*
10	GPIO_2	*Non utilisé*
11	SPI_MISO	*Non utilisé*
12	SPI_MOSI	*Non utilisé*
13	SPI_CLK	*Non utilisé*
14	SPI_CS	*Non utilisé*

Tableau 1: liste des signaux du connecteur Arduino

C. Configuration de la communication UART

Le port série de l'Arduino doit être configuré correctement afin de pouvoir communiquer avec la carte mère du bloc moteur :

Vitesse	mot	Stop	parité
115 200 baud	8 bits	1 bit	Aucune

Tableau 2: Configuration de la communication série UART

D. Carte Arduino UNO

L'illustration ci-dessous illustre une carte Arduino UNO, mais il est possible d'utiliser toute sorte de cartes ayant un port série configurable.

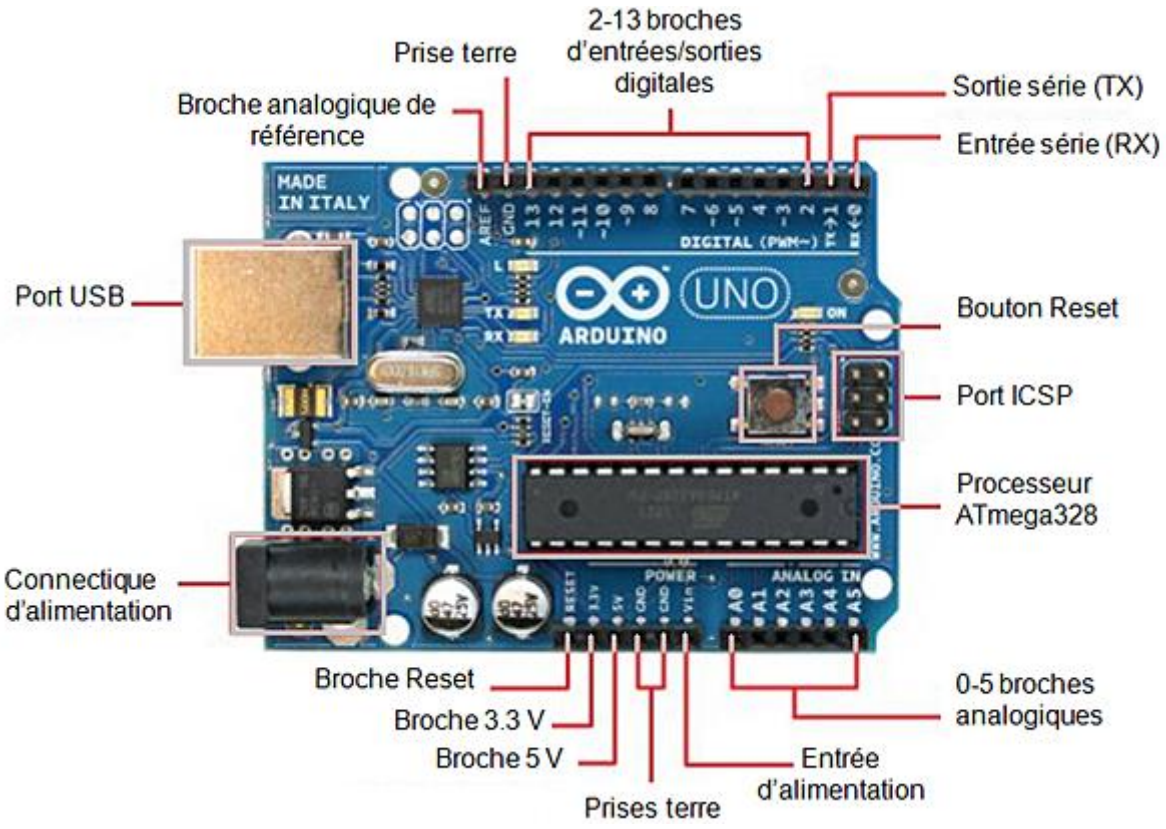


Figure 2 : Arduino UNO

Dans la suite du document, nous ferons principalement référence à la carte Arduino UNO, la plus simple des cartes Arduino.

E. Principe de communications entre les carte : UART

Le bloc moteur utilise la communication UART pour échanger avec la carte Arduino UNO. La liaison série UART (Universal Asynchronous Receiver Transmitter) permet de faire la liaison entre un ordinateur et le bloc moteur au travers d'un port série. La communication UART s'effectue par l'envoi d'une trame d'octets, soit un paquet de mots de 8 bits chacun. Pour ce faire, il est nécessaire d'utiliser deux broches :

- Une broche RX qui permet de recevoir les bits ;
- Une broche TX qui permet de transmettre les bits.

Le bloc moteur et la carte Arduino possèdent tous les deux ces broches, ils peuvent donc communiquer entre eux : la broche TX (transmission) d'un élément doit être connecté à la broche RX (réception) de l'autre élément, et vice-versa. Il est aussi indispensable de connecter une broche GND de chacun des éléments entre eux afin d'avoir la même référence de tension.

C'est de cette façon que votre moteur pourra exécuter les actions que vous lui aurez ordonné avec le programme de la carte Arduino.

L'arduino UNO n'ayant pas de 2ème liaison série (autre que celle avec l'ordinateur via USB), la librairie Arduino utilise un émulateur d'UART via la librairie « SoftwareSerial » et affecte les broches 10 et 11 de l'Arduino UNO comme pins RX et TX pour cette communication série.

II. Rappel Electronique : Pont diviseur de tension (appliqué à notre cas)

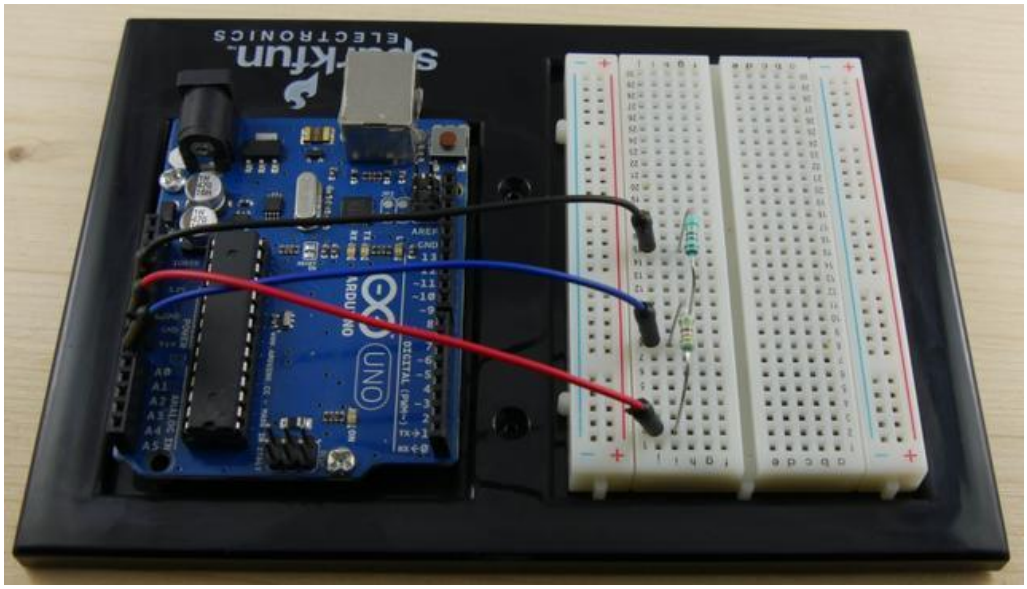


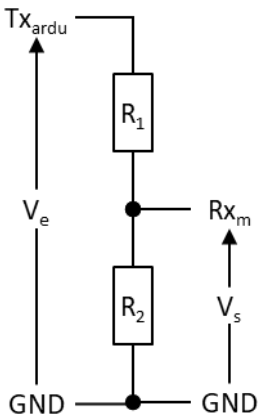
Figure 3 : Montage en pont diviseur de tension avec une Arduino

A. Utilité d'un pont diviseur de tension

Un pont diviseur de tension est souvent utilisé dans les cas où on cherche à effectuer une communication entre deux composants électroniques n'ayant pas une tension d'entrée commune. Dans notre cas, la carte Arduino envoie en sortie une tension de 5,0 volts tandis que notre carte moteur attend en entrée une tension de 3,3 volts. L'utilité d'un pont diviseur de tension sera de ne pas endommager la carte moteur en lui envoyant une tension trop élevée ce qui pourrait causer un dysfonctionnement de la carte voir la rendre inopérante.

B. Fonctionnement d'un pont diviseur de tension

Un pont diviseur de tension se représente de la manière ci-contre.



Dans notre cas on a :

- Tx_{ardu} : représente la Pin de transmission de la carte Arduino, elle envoie avec une tension de 5V, nous noterons sa tension V_e ;
- R_1, R_2 : deux résistances nous permettant d'établir le pont diviseur de tension;
- Rx_m : représente la Pin de réception du moteur, c'est elle qui attend une tension de 3.3V nous noterons cette tension V_s ;

Le pont diviseur de tension ne sera appliqué que dans ce sens, l'autre sens nous mettrons seulement une résistance de sécurité permettant de s'assurer du bon fonctionnement de notre branchement.

Dans notre cas nous avons la formule suivante :

$$V_s = V_e * \frac{R_2}{R_1 + R_2}$$

Nous connaissons les valeurs de V_s et de V_e ce qui nous permet d'en déduire :

$$\frac{R_2}{R_1 + R_2} = 0.66$$

Le couple de résistance devra respecter cette valeur pour assurer un bon fonctionnement de l'ensemble du circuit.

C. Recommandation

Pour le choix de vos résistances, l'objectif est aussi de ne pas utiliser trop d'énergie et donc de rester autour d'une intensité de 1mA. Les couples de résistances recommandés seront les suivants :

- $R_2 = 3.3\ k\Omega$ et $R_1 = 1.5\ k\Omega$
- $R_2 = 4.7\ k\Omega$ et $R_1 = 2.2\ k\Omega$

D. Alternatives aux ponts diviseurs de tensions : les convertisseurs de niveaux logique

Une alternative aux ponts diviseurs de tension est d'utilisé un convertisseurs de niveaux logique, ce composant permet de directement faire la conversion du signal de part et d'autre du système. Simplifiant par ce biais la visibilité au sein du circuit ainsi que les calculs à effectuer.

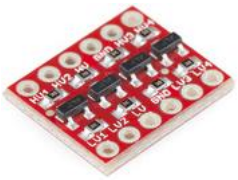


Figure 4 : Image de convertisseur de niveaux logiques Sparkfun

Exemple de branchement avec une Raspberry Pi :

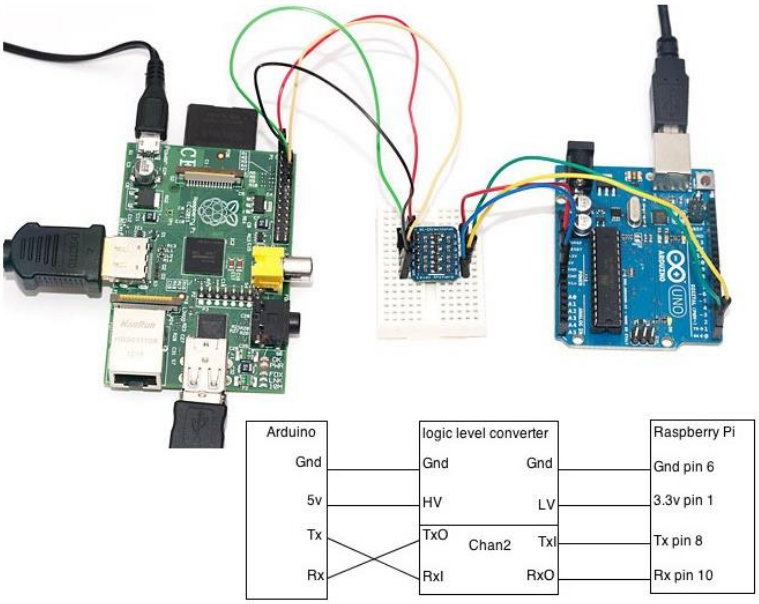


Figure 5 : Connexion à une Raspberry Pi via un convertisseur de niveau

La particularité dans notre cas avec une Arduino UNO est qu'il ne comporte qu'un seul port série matériel, affecté à la liaison avec le PC via un convertisseur série – USB présent sur la carte. Il faut donc créer de manière logiciel un second port série que l'on va dédier à la communication avec le moteur. Nous lui affectons la pin 10 pour RX et la min 11 pour TX (ligne 52 de CEC_lib.cpp : `SoftwareSerial motorSerial(10, 11); //Pin 10:RX  Pin 11:TX`)

Lien : https://www.sparkfun.com/products/12009

https://www.robot-maker.com/shop/convertisseur/212-convertisseur-de-niveau-logique-212.html

III. Rappel changement de base numérique : Base10, Base16, Base2

Dans l'électronique actuel, nous nous servons de deux grandes unités, l'*octet* et le *bit*. Ces deux unités sont liées car un octet est composé de 8 bits. Mais que représente en valeur réel ces différentes unités ?

A. Représentation d'un chiffre dans différentes bases

Prenons un nombre quelconque : **142**

Ce nombre en base 10 peux être transcrit en différentes base :

1. En base 2

Ce nombre devient : **1000 1110**

0n compte de la manière suivante :

$$142 = 1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$$

Ou quand on change les exponentiels :

$$142 = 1 * 128 + 0 * 64 + 0 * 32 + 0 * 16 + 1 * 8 + 1 * 4 + 1 * 2 + 1 * 0$$

2. En base 16

La base 16 est aussi appelée hexadécimale ; le principe est équivalent.

Tout d'abord il faut voir la notation en passant par un tableau :

Notation Hexa	Valeur décimale
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Tableau 3 : table de conversion hexadécimale

Si nous reprenons le nombre 142 cela nous donnera : 0x8E

0x : permet d'indiquer que nous sommes sous format Hexadécimal

Pour le reste, on convertit de la manière suivante :

$$142 = 16^1 * 8 + 16^0 * 14$$

Le E devient la valeur 14 quand on effectue la conversion.

B. Généralisation du principe de conversion

Quel que soit le nombre, la conversion s'effectue toujours de la manière suivante :

1. Dans le sens décimal vers Binaire

- On fait la division avec reste suivante :

$$Valeur\ binaire\ n = (reste\ de\ la\ division\ n + 1)/2^n$$

- Exemple :

Avec le nombre 142 :

$$2^7 = 128 < 142$$

Donc en effectuant la division nous auront :

$$reste = 14\ et\ quotient = 1$$

$$142 - 128 = 14$$

Ensuite nous aurons :

$$2^6 = 64 > 14, 2^5 = 32 > 14, 2^4 = 16, 2^3 = 8 < 14$$

Donc

$$reste = 6\ et\ quotient = 1$$

Nous poursuivons :

$$2^2 = 4 < 6\ reste = 2\ et\ quotient = 1$$

$$2^1 = 2\ reste = 0\ et\ quotient = 1$$

Nous obtenons donc le chiffre binaire :

$$(142)_2 = 1000\ 1110$$

2. Dans le sens décimal à Hexadécimal

Le principe reste le même.

- On effectue la division avec reste suivant :

$$valeur\ hexadecimal\ n = (reste\ de\ la\ division\ n + 1)/16^n$$

- Le quotient peut varier entre 0 et 15 en Hexadécimal donc il faut trouver le quotient pour lequel le reste est minimum ;
- Exemple :

Avec le nombre 142 :

$$16 * 9 = 144 > 142, 16 * 8 = 128 < 142$$

Nous aurons donc :

$$reste = 14\ et\ quotient = 8$$

Pour la suite :

$$E = 14$$

On obtient la notation hexadécimale suivante :

$$(142)_{16} = 0x8E$$

3. Dans le sens Binaire vers décimal

- On effectue une multiplication successive et nous noterons ici la valeur du bit $Q_n = 0\ ou\ 1\ n$ représentant la position du bit :

$$Résultat = Q_7 * 2^7 + Q_6 * 2^6 + Q_5 * 2^5 + Q_4 * 2^4 + Q_3 * 2^3 + Q_2 * 2^2 + Q_1 * 2^1 + Q_0 * 2^0$$

4. Dans le sens Hexadécimal vers décimal

- On effectue une multiplication successive et nous noterons ici la valeur du bit $Q_n = 0 \text{ à } 15\ n$ représentant la position du bit :

$$Résultat = Q_1 * 16^1 + Q_0 * 16^0$$

C. Utilité des notations

Les différentes notations présentées ont chacune une utilité différente, permettant d'apporter des aspects différents comme la rapidité de traitement ou l'allégement de certain message. Les avantages sont les suivant :

1. La base 10

Cette base est celle nous servant de référence de notation dans le langage naturel et permet de faire la correspondance entre différentes bases. Elle nous permet d'effectuer les calculs permettant des opérations simples ou complexes et d'afficher les valeurs dans le cadre numérique. Pour ce qui est de la communication, la base 10 n'est pas le moyen favorisé, en effet, le poids des variables en base 10 peuvent être plus important que celui en binaire ou en hexadécimal ;

2. La base 2

La base 2 ou le binaire est la base de référence dans le numérique, souvent on forme ce que l'on appelle des octets par tranche de 8 bits. Cette notation permet de contrôler les différentes actions et informations lors d'échanges entre différents systèmes. Il permet d'établir des trames d'échange et de géré les différentes particularités des informations (exemple : un bit est attribué à une action précise, ce bit est regroupé avec d'autre pour former un octet, permettant ainsi d'expliquer comme ce forme la trame).

3. La base 16

La base 16 ou hexadécimal est une base dérivant de la base binaire. En effet, dans le cadre de la simplification dans les échanges, l'hexadécimal sera souvent à privilégier, cela permet une notation compacte de la valeur des différents octets qui forme un message ou une information.

D. Les cas particuliers : traductions des variables composées de plusieurs octets

Dans le cadre du traitement de trames, c'est-à-dire de chaine d'octets portant les informations de communication au sein d'un système global, certaines informations regroupent un nombre différents d'octets. Il y a deux manière de voir ce traitement :

Dans les deux cas on a ce que l'on note le LSB, *Least Significant Byte* ou octet de poids faible, et le MSB, *Most Significant Byte* ou octet de poids fort. Ces deux éléments permettent de déterminer le sens dans lequel est organisée une chaine d'octets. Dans notre cas le MSB sera toujours placé avant le LSB.

- Les valeurs à virgules : dans le cadre de l'envoie de certaines informations, des valeurs à virgules peuvent être échangées, le LSB représentera généralement la valeur après la virgule, tandis que le MSB sera la valeur devant la virgule.

Exemple : Si l'on prend en Hexadécimal $0x055F$, on aura la répartition suivante :

$$MSB = 0x05 = 5 \; et \; LSB = 0x5F = 95$$

L'information sera en décimal:

$$valeur\ en\ décimal = 5{,}95$$

- Les valeurs à unités convertissable : L'exemple le plus simple pour ce cas est celui de la vitesse, qui comporte une unité de base étant le m/s (mètres par seconde) cette unité peut être convertie en cm/s permettant ainsi d'éliminer la virgule. Dans ce cas on calcul la valeur en décimal de la manière suivante :

$$Valeur\ décimale = (valeur\ décimale\ MSB) * 256 + (valeur\ décimale\ LSB)$$

Exemple : Si l'on prend en Hexadécimal $0x055F$ cela nous donne :

$$5 * 256 + 95 = 1375$$

Si l'on considère l'exemple de la vitesse pour repasser du cm/s au m/s il nous suffira de divisé le résultat par 100.

IV. Code Arduino pour la communication Arduino-Carte mère

A. Introduction codage Arduino

1. Présentation de l'IDE et des fonctionnalités de base

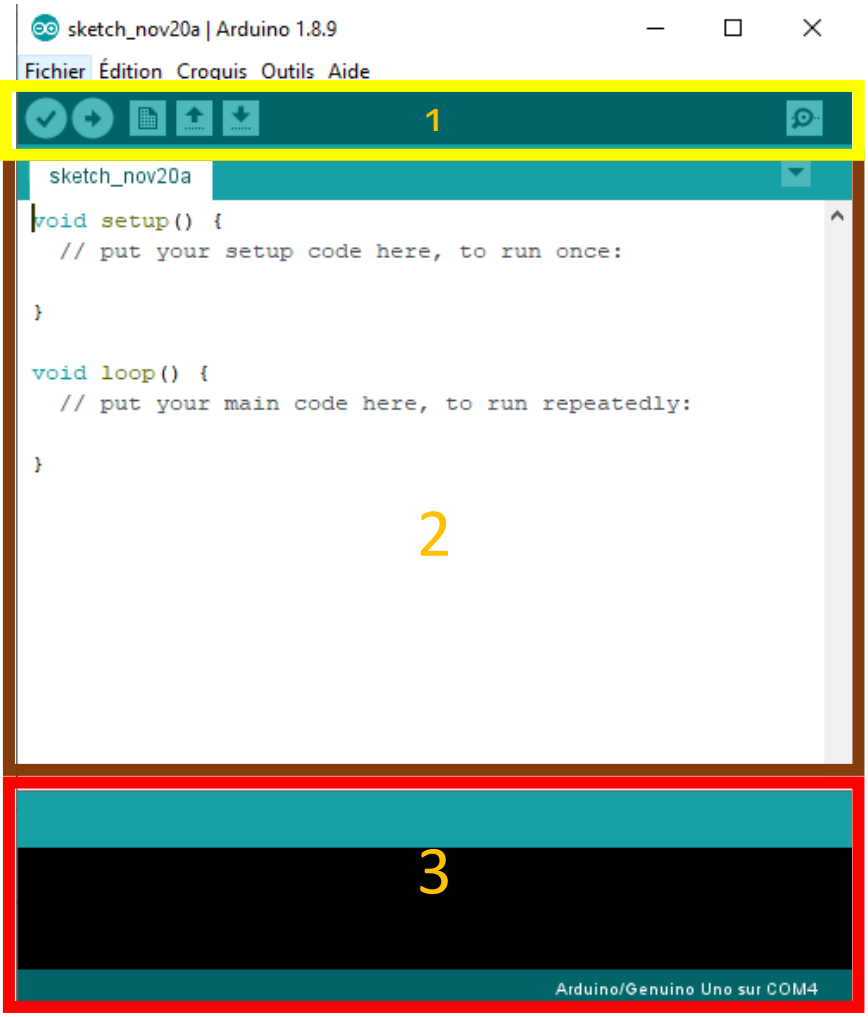


Figure 6 : Présentation de l'IDE Arduino

Un IDE est un Environnement de Développement Intégré, ou *Integrated Development Environment* en Anglais. Dans l'ordre vous pouvez voir trois partie distinct dans l'IDE Arduino :

- La 1ère : on y retrouve plusieurs boutons qui de la gauche vers la droite correspondent aux fonctionnalités suivante :
  - Le premier bouton vous servira à *compiler* le code vous permettant de le vérifier, l'IDE vérifiera la syntax de votre code et vous informera des erreurs potentielles ;
  - Le deuxième bouton vous servira à *Téléverser* votre code dans la carte Arduino, il vous sert donc à appliquer votre code dans votre Arduino, il compilera votre code si ce n'est déjà fait ;
  - Les trois suivants vous permettront dans l'ordre d'ouvrir une nouvelle fenêtre, d'ouvrir un ancien fichier et pour le dernier de sauvegarder votre code actuel ;
  - Le dernier tout à droite vous permettra d'ouvrir la fenêtre du *moniteur série*. Le moniteur série vous permettra d'observer les échanges entre votre Arduino et les différents composants. Elle peut aussi vous servir à afficher des messages pendant le fonctionnement.
- La deuxième fenêtre définie la zone où se situe le code que vous allez écrire. En règle générale, vous retrouverez toujours au moins deux fonctions :

- Une fonction `setup()` qui sera appelée une seule fois au tout début de l'exécution de votre programme ; vous y mettrez les tâches d'initialisation pour votre Arduino et pour l'ouverture de la liaison avec le moteur ;
- Une fonction `loop()` qui sera appelée en boucle après l'exécution de la fonction `setup()` ; elle vous permettra d'effectuer de manière répétitive des actions, c'est ici en général que vous devriez faire appel aux fonctions de la librairie Courses en Cours.

- La troisième fenêtre quant à elle est celle qui vous permettra de voir les différentes informations système : quand vous ferez la vérification de votre code, les erreurs apparaitront dans cette console ; de même, quand le programme aura réussi à être téléversé dans votre Arduino, vous serez aussi informés via cette console.

Voilà pour ce qui est de la présentation de la première vue qui apparaît lors de la création d'un nouveau projet Arduino.

2. Connexion d'une carte

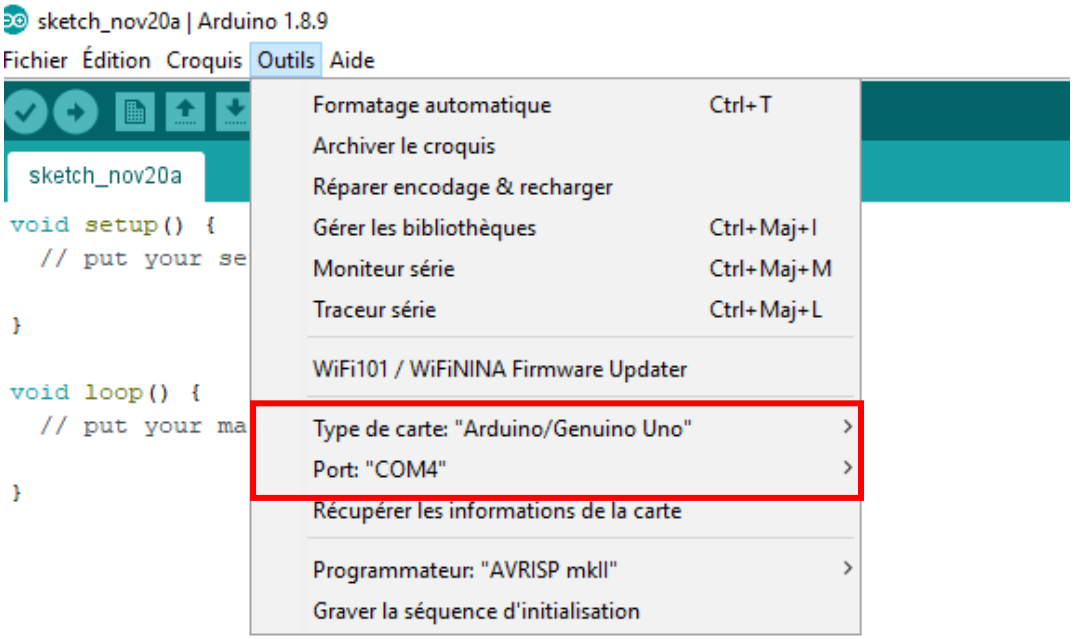


Figure 7 : IDE Arduino - Connexion d'une carte

Deuxième chose à savoir avant de se lancer dans le code, vous devrez renseigner le type de carte Arduino (ou générique compatible) avec laquelle vous travaillez ainsi que le port du PC sur lequel la carte est branchée (en général cela se fait automatiquement mais une vérification est toujours de mise).

3. Votre premier code : le blinking LED

Ce code est déjà fourni avec le logiciel Arduino téléchargeable sur le site officiel. Pour y accéder il vous suffit simplement d'aller dans fichier -> Exemples - > Digital et vous trouverez le code « *Blink without delay* ». Avec les commentaires présents dans le code cela vous expliquera le fonctionnement basique de l'arduino.

4. L'utilisation des ports série : Serial.function()

La deuxième partie à savoir est l'utilisation des fonctions liées au port série ; en effet cela fait partie du cœur de la programmation permettant une communication entre votre moteur et la carte Arduino. Ici vous pourrez vous servir de l'exemple SerialCallResponse présent dans l'onglet exemple-> communication.

Il y a 4 grandes utilisations d'un port série, ils sont les suivant :

- La lecture, le port série demande à lire une information qui lui a été transmise par `Serial.read()` ;
- L'écriture, le port série va envoyer une information à un composant qui lui est rattaché par `Serial.write()` ;
- L'écoute, le port série va attendre de recevoir des informations et va pouvoir le notifier par la fonction `Serial.available()` ;
- L'affichage, le port série va permettre d'afficher des informations par les fonction `Serial.print()` et `Serial.println()`.

A partir d'ici vous devriez normalement avoir compris le principe d'utilisation de base des fonctions simples. Si jamais vous ne comprenez toujours pas n'hésitez pas à regarder d'autres exemple de cas concret d'utilisation.

5. L'utilisation de librairie

Les librairies sont des éléments permettant d'ajouter des fonctionnalités à vos codes, comme par exemple vous interfacer à un composant vous permettant de mesure différentes choses ou de nouvelles fonctions mathématiques pour vos calculs. En règle générale, toutes les librairies sont déclarées avant la fonction `setup()`.

La librairie du moteur Course en Cours est composée de 2 fichiers : CEC_lib.cpp et CEC_lib.h. Ces 2 fichiers doivent être dans le même répertoire que votre fichier .ino, votre sketch arduino.

Pour utiliser cette librairie, vous utiliserez ce formalisme :

```
#include <Arduino.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "CEC_lib.h"
```

Le mot clé #include permet d'inclure un fichier à cet endroit, en l'occurrence un fichier .h d'une librairie. Il est indispensable de positionner dans le sens de lecture du fichier (de haut en bas) cette déclaration dans le sketch avant d'utiliser les fonctions de la librairie.

La fonction `setup()` est appelée après avoir inclus les différentes librairies dont nous avons besoin ici. On peut remarquer que la librairie CEC_lib.h est présente ; c'est cette librairie qui vous aidera à utiliser les fonctionnalités concernant le moteur.

6. La structure du code CEC

La structure du code est la suivante :

- Un fichier **.ino**, fichier de base d'un projet arduino. Il contient la configuration des ports et la boucle permettant l'exécution du code, il servira d'exemple d'utilisation de la librairie (la fonction `setup()` et la fonction `loop()`) ;

- Un **.cpp** permettant d'avoir la définition des fonctions utilisé dans le code **.ino** ;
- Un **.h** permettant d'effectuer le lien entre le **.cpp** et le **.ino**.

A partir d'ici nous allons faire une introduction de toutes les fonctions qui vous permettront de faire fonctionner votre moteur via votre Arduino.

V. Liste des méthodes

La librairie définit l'objet **Moteur** auquel est associé des *méthodes*, fonctions agissant sur le moteur et permettant d'interagir avec lui. Certaines méthodes sont accessibles par l'utilisateur, d'autres non car internes à l'objet.

Afin d'utiliser des fonctions liées au moteur, il faut déjà instancier un moteur dans les déclarations avant le setup(), c'est-à-dire déclarer son moteur afin qu'il ait une réalité en mémoire :

```
Moteur monMoteur;
```

Par la suite, l'utilisation des méthodes se fait par des commandes de type `monMoteur.methode()`.

Dans le principe, la librairie dialogue avec le moteur par échange de trames qui sont soit émises par les méthodes `monMoteur.sendTrame()`, soit reçues par les méthodes `monMoteur.readTrame()` ou `Trame` indique la trame mise en œuvre : Info, Config, Info, Move, Mesure, Bilan.

Les méthodes peuvent être regroupées en plusieurs groupes suivant le type de trame mise en œuvre : « Génériques », « Annexes », « Info », « Config », « Move », « Mesure », « Bilan ».

Voici la liste des méthodes.

A. Méthodes « Génériques »

Les méthodes « Génériques » ne font pas référence aux trames ; elles sont utilisées de manière plus générales.

1. Moteur

Il faut avant tout définir en mémoire l'objet **Moteur**, c'est-à-dire l'instancier ou encore lui réserver de la place en mémoire pour ses données internes, tout comme les cariables.

Prototype : `Moteur <nom_du_moteur>;`

Exemple : `Moteur monMoteur;`

Cet exemple instancie 1 objet **Moteur** dénommé *monMoteur*. A travers *monMoteur*, il est maintenant possible d'interagir avec le moteur CeC, notamment après l'ouverture de la communication.

2. start()

Permet d'ouvrir la liaison série avec le moteur afin de pouvoir communiquer avec lui. Doit être intégrée dans la fonction `setup()`. Le paramètre est la définition de la vitesse de communication en bits par seconde ou *baud*. La valeur doit être 115200 bauds pour le moteur CeC.

Prototype : `void start(uint32_t serialConfig);  // Démarrage de la liaison avec le moteur`

Exemple : `monMoteur.start(115200);`

Ceci permet d'ouvrir la liaison avec le moteur à 115200 bit/s.

3. setEchoOn(), setEchoOff()

Permet d'activer ou de désactiver la sortie texte (écho) permettant de mieux suivre l'évolution du code lors de sa mise au point. L'écho est activé par défaut.

Prototype :
```
void setEchoOn();                          // Active la sortie texte vers le terminal
void setEchoOff();                         // Désactive la sortie texte vers le terminal
```

Exemple :
```
setEchoOff() ;  // stop l'écho texte
setEchoOn() ;   // réactive l'écho texte
```

4. purge()

Permet de purger la liaison moteur en lecture (vide tout caractère en réception).

Prototype :
```
void purge();                              // Vide le buffer de réception de la liaison moteur
```

Exemple :
```
monMoteur.purge();
```

B. Méthodes « Annexes »

Les méthodes « Annexes » sont des fonctions pouvant être utilisées pour des conversions de paires d'octets en nombre notamment.

1. convertIntToHex()

Permet de convertir une valeur entière signée entre -32768 et +32767 en 2 octets, valMSB et valLSB, passés par adresse.

Prototype :
```
void convertIntToHex(byte *valMSB, byte *valLSB, int valeur);
```

Exemple :
```
int maValeurInt = -4582;
byte maValeurMSB;
byte maValeurLSB;
monMoteur.convertIntToHex(&maValeurMSB, &maValeurLSB, maValeurInt);
```
`maValeurMSB` prend la valeur `0xEE, ou 238` en décimal
`maValeurLSB` prend la valeur `0x1A, ou 26` en décimal

2. convertUIntToHex()

Permet de convertir une valeur entière non signée entre 0 et +65535 en 2 octets, valMSB et valLSB, passés par adresse.

Prototype :
```
void convertUIntToHex(byte *valMSB, byte *valLSB, unsigned int valeur);
```

Exemple :
```
unsigned int maValeurUInt = 60954;
byte maValeurMSB;
byte maValeurLSB;
monMoteur.convertIntToHex(&maValeurMSB, &maValeurLSB, maValeurUInt);
```
`maValeurMSB` prend la valeur `0xEE, ou 238` en décimal
`maValeurLSB` prend la valeur `0x1A, ou 26` en décimal

3. convertHexToInt()

Permet de convertir une paire d'octets valMSB et valLSB en une valeur entière signée entre -32768 et +32767, passés par adresse.

Prototype :
```
void convertHexToInt(byte valMSB, byte valLSB, int *valeur);
```

Exemple :
```
byte maValeurMSB = 0xEE;
byte maValeurLSB = 0x1A;
int maValeurInt;
monMoteur.convertHexToInt(maValeurMSB, maValeurLSB, &maValeurInt);
```

`maValeurInt` prend la valeur `-4582` en décimal

4. convertHexToUInt()

Permet de convertir une paire d'octets valMSB et valLSB en une valeur entière non signée entre 0 et +65535, passés par adresse.

Prototype :
```
void convertHexToUInt (byte valMSB, byte valLSB, unsigned int *valeur);
```

Exemple :
```
byte maValeurMSB = 0xEE;
byte maValeurLSB = 0x1A;
unsigned int maValeurUInt;
monMoteur.convertHexToUInt(maValeurMSB, maValeurLSB, &maValeurUInt);
```
`maValeurUInt` prend la valeur `60954` en décimal

C. Méthodes « Info »

Les méthodes « Info » sont relatives à la trame Info. Il est nécessaire de récupérer la trame Info du moteur par `readInfo()` avant de lire les paramètres de la trame par les autres méthodes.

1. readInfo()

Permet de récupérer la trame Info du moteur. Précède les fonctions `getInfoConnex()` et `printInfoVersion()`.

Prototype :
```
int readInfo();                 // Trame Infos: réception des valeurs
```

2. getInfoConnex()

Permet de récupérer l'information de connexion la trame Info moteur récupérée par la méthode `readInfo()`.

Prototype :
```
int getInfoConnex();            // Récupère l'information de connexion au moteur
```

Retour :
```
CEC_ERR_INTERNAL                // -8 : erreur interne moteur
CEC_ERR_OK                      //  0 : pas d'erreur
```

3. getInfoVersion()

Permet de récupérer de la trame Info moteur récupérée par la méthode `readInfo()` les informations de version.

Prototype :
```
byte getInfoVersion(byte type);  // Affiche la version suivant type = CEC_VERSION_xx
```

Paramètre :
```
CEC_VERSION_MERE_HARD           //  1 : Affiche la version majeure HW de la carte mère
CEC_VERSION_MERE_HARD_MIN       //  2 : Affiche la version mineure HW de la carte mère
CEC_VERSION_MERE_SOFT           //  3 : Affiche la version majeure SW de la carte mère
CEC_VERSION_MERE_SOFT_MIN       //  4 : Affiche la version mineure SW de la carte mère
CEC_VERSION_MOT_HARD            //  5 : Affiche la version majeure HW de la carte moteur
CEC_VERSION_MOT_HARD_MIN        //  6 : Affiche la version mineure HW de la carte moteur
CEC_VERSION_MOT_SOFT            //  7 : Affiche la version majeure SW de la carte moteur
CEC_VERSION_MOT_SOFT_MIN        //  8 : Affiche la version mineure SW de la carte moteur
CEC_VERSION_CAPT_HARD           //  9 : Affiche la version majeure HW de la carte capteur
CEC_VERSION_CAPT_HARD_MIN       // 10 : Affiche la version mineure HW de la carte capteur
CEC_VERSION_CAPT_SOFT           // 11 : Affiche la version majeure SW de la carte capteur
CEC_VERSION_CAPT_SOFT_MIN       // 12 : Affiche la version mineure SW de la carte capteur
```

Exemple :
```
byte maVersionMaj, maVersionMin;
maVersionMaj = monMoteur.getInfoVersion(CEC_VERSION_MERE_HARD);
maVersionMin = monMoteur.getInfoVersion(CEC_VERSION_MERE_HARD_MIN);
```

4. printInfoVersion()

Permet de sortir en texte de la trame Info moteur récupérée par la méthode `readInfo()` les informations de version majeure.mineure matériel et logiciel d'une carte.

Prototype :
```
int printInfoVersion(byte type);  // Affiche la version suivant type = CEC_VERSION_xx
```

Paramètre :
```
CEC_VERSION_MERE_HARD          // 1 : Affiche la version HW de la carte mère
CEC_VERSION_MERE_SOFT          // 3 : Affiche la version SW de la carte mère
CEC_VERSION_MOT_HARD           // 5 : Affiche la version HW de la carte moteur
CEC_VERSION_MOT_SOFT           // 7 : Affiche la version SW de la carte moteur
CEC_VERSION_CAPT_HARD          // 9 : Affiche la version HW de la carte capteur
CEC_VERSION_CAPT_SOFT          // 11 : Affiche la version SW de la carte capteur
```

Cet exemple utilise toutes les méthodes autour de la trame Info.

Exemple :
```
int err;                                  // Code dans loop()
if((err = monMoteur.readInfo()) < 0)
{ // Si erreur:
  Serial.print("err: ");
  Serial.println(err);
}
else
{
  Serial.print("version mère:    HW=");
  Err = monMoteur.printInfoVersion(CEC_VERSION_MERE_HARD);
  Serial.print(" SW=");
  Err = monMoteur.printInfoVersion(CEC_VERSION_MERE_SOFT);
  Serial.print("\nversion moteur:  HW=");
  Err = monMoteur.printInfoVersion(CEC_VERSION_MOT_HARD);
  Serial.print(" SW=");
  Err = monMoteur.printInfoVersion(CEC_VERSION_MOT_SOFT);
  Serial.print("\nversion capteur: HW=");
  Err = monMoteur.printInfoVersion(CEC_VERSION_CAPT_HARD);
  Serial.print(" SW=");
  Err = monMoteur.printInfoVersion(CEC_VERSION_CAPT_SOFT);
  Serial.println();
}
```

Sortie Texte :
```
version mère:    HW=3.0 SW=2.1
version moteur:  HW=4.10 SW=5.2
version capteur: HW=2.3 SW=1.0
```

D. Méthodes « Config »

Les méthodes « Config » sont relatives à la trame Config. Il est nécessaire de récupérer la trame Config du moteur par `readConfig()` avant de lire les paramètres de la trame par les méthodes `getConfig_()`. Il nécessaire d'initialiser la trame Config par `initConfig()` ou `readConfig()` avant de la modifier et configurer par les méthodes `setConfig_()` avant de l'envoyer au moteur par la méthode `sendConfig()`.

1. initConfig()

Permet d'initialiser une première trame de configuration avant de la modifier et configurer par les méthodes `set_()` avant de l'envoyer au moteur par la méthode `sendConfig()`.

Prototype :
```
int initConfig();              // Trame de configuration: initialisation
```

Retour :
```
CEC_ERR_OK;                    //  0 : pas d'erreur
```

Exemple :
```
monMoteur.initConfig();
```

2. readConfig()

Permet de récupérer trame Config du moteur avant de lire les paramètres de la trame par les méthodes `get_()`, ou la modifier et configurer par les méthodes `set_()` avant de l'envoyer au moteur par la méthode `sendConfig()`.

Prototype :
```
int readConfig();                // Trame de configuration: réception des valeurs de configuration
```

Retour :
```
CEC_ERR_KO_F1                    // -7 : erreur de réception message par le moteur
CEC_ERR_SERIAL_NOT_AVAILABLE     // -5 : erreur liaison série moteur
CEC_ERR_LENGTH                   // -4 : erreur de longueur trame
CEC_ERR_WRONG_CMD                // -3 : erreur de réception code commande
CEC_ERR_TIMEOUT                  // -2 : erreur de timeout (délai d'attente trop long)
CEC_ERR                          // -1 : erreur générique, début de trame erronée
CEC_ERR_OK                       //  0 : pas d'erreur
CEC_ERR_WRONG_CHECKSUM           //  1 : Ok mais mauvais checksum
```

Exemple :
```
int err;
err = monMoteur.readConfig();
```

Par la suite, pour les différentes fonctions `getConfig_()`, on suppose qu'une récupération de trame config a été faite précédemment.

3. sendConfig()

Permet d'envoyer la trame Config au moteur après l'avoir construite par les méthodes `initConfig()`, `readConfig()` et `set_()`. Force les paramètres du nombre de dents du pignon moteur à 12 et du nombre de dents de la couronne à 34.

Prototype :
```
int sendConfig();                // Trame de configuration: envoie vers le moteur
```

Retour :
```
CEC_ERR_KO_F1                    // -7 : erreur de réception message par le moteur
CEC_ERR_SERIAL_NOT_AVAILABLE     // -5 : erreur liaison série moteur
CEC_ERR_LENGTH                   // -4 : erreur de longueur trame
CEC_ERR_WRONG_CMD                // -3 : erreur de réception code commande
CEC_ERR_TIMEOUT                  // -2 : erreur de timeout (délai d'attente trop long)
CEC_ERR                          // -1 : erreur générique, début de trame erronée
CEC_ERR_OK                       //  0 : pas d'erreur
CEC_ERR_WRONG_CHECKSUM           //  1 : Ok mais mauvais checksum
CEC_ERR_OK_F0                    //  2 : Ok avec retour correct du moteur (F0)
```

Exemple :
```
int err;
err = monMoteur.readConfig();
…
err = monMoteur.sendConfig();
```

4. getConfigLongueurPiste ()

Permet de récupérer de la trame Config le paramètre de longueur de piste. Retourne la longueur de piste sur une valeur entière en cm.

Prototype :
```
unsigned int getConfigLongueurPiste();      // Récupère la longueur de piste
```

Exemple :
```
unsigned int uiLongPiste;
uiLongPiste = monMoteur.getConfigLongueurPiste();
```

5. getConfigTailleDamier ()

Permet de récupérer de la trame Config le paramètre de taille de damier, vernier sur la piste. La valeur est de 25cm par défaut. Pas utilisé pour le moment.

Prototype :
```
unsigned int getConfigTailleDamier();       // Récupère la taille du damier
```

Exemple :
```
unsigned int uiTailleDamier;
uiTailleDamier = monMoteur.getConfigTailleDamier();
```

6. getConfigDiametreRoues()

Permet de récupérer de la trame Config le paramètre de diamètre de roue. Retourne le diamètre de roue sur une valeur entière en dixième de mm.

Prototype :
```
unsigned int getConfigDiametreRoues();      // Récupère le diamètre des roues
```

Exemple :
```
unsigned int uiDiametreRoue;
uiDiametreRoue = monMoteur.getConfigDiametreRoues();
```

7. getConfigPignon ()

Permet de récupérer de la trame Config le paramètre de transmission côté pignon moteur. Retourne le nombre de dents du pignon moteur sur une valeur entière.

Prototype :
```
unsigned int getConfigPignon();      // Récupère le nombre de dent pignon
```

Exemple :
```
unsigned int uiNDentsPignon;
uiNDentsPignon = monMoteur.getConfigPignon();
```

8. getConfigCouronne()

Permet de récupérer de la trame Config le paramètre de transmission côté couronne essieu roues. Retourne le nombre de dents de la couronne essieu roue sur une valeur entière.

Prototype :
```
unsigned int getConfigCouronne();          // Récupère le nombre de dent Couronne
```

Exemple :
```
unsigned int uiNDentsCouronne;
uiNDentsCouronne = monMoteur.getConfigCouronne();
```

9. getConfigVehicule()

Permet de récupérer de la trame Config le paramètre de configuration du véhicule. Retourne le type de configuration, CEC_TYPE_PROPULSION ou CEC_TYPE_TRACTION.

Prototype :
```
unsigned int getConfigVehicule();          // Récupère le type de motorisation
```

Retour :
```
CEC_TYPE_PROPULSION                    //  1 : Type de véhicule = propulsion
CEC_TYPE_TRACTION                      //  0 : Type de véhicule = traction
```

Exemple :
```
unsigned int uiConfigVehicule;
uiConfigVehicule = monMoteur.getConfigVehicule();
```

10. getConfigCourse()

Permet de récupérer de la trame Config le paramètre de configuration de fin de course. Retourne le type de configuration, CEC_TYPE_ARRET_LIGNE ou CEC_TYPE_DAMIER (non implémenté dans le moteur).

Prototype :
```
unsigned int getConfigCourse();            // Récupère le type de fin de course
```

Retour :
```
CEC_TYPE_ARRET_LIGNE                   //  1 : Type de fin de course = arrêt sur ligne d'arrivée
CEC_TYPE_DAMIER                        //  2 : Type de fin de course = arrêt sur damier
```

Exemple :
```
unsigned int uiConfigCourse;
uiConfigCourse = monMoteur.getConfigCourse();
```

11. getConfigPeriodeEch()

Permet de récupérer de la trame Config le paramètre de période d'échantillonage. Retourne la valeur entière en ms de la période d'échantillonnage du moteur, normalement 20ms.

Prototype :
```
unsigned int getConfigPeriodeEch();        // Récupère la periode d'echantillonnage des mesures
```

Exemple :
```
unsigned int uiPeriodeEch;
uiPeriodeEch = monMoteur.getConfigPeriodeEch();
```

12. getConfigTempsMaxCourse()

Permet de récupérer de la trame Config le paramètre de temps maximum de la course. Retourne la valeur entière en ms du temps maximum de course.

Prototype : `unsigned int getConfigTempsMaxCourse();  // Récupère le temps maximum de la course`

Exemple :
```
unsigned int uiTempsMaxCourse;
uiTempsMaxCourse = monMoteur.getConfigTempsMaxCourse();
```

13. getConfigVitesseMin()

Permet de récupérer de la trame Config le paramètre de vitesse minimum, celle au démarrage. Retourne la valeur entière en cm/s de la vitesse minimum.

Prototype : `unsigned int getConfigVitesseMin();  // Récupère la vitesse minimale de la course`

Exemple :
```
unsigned int uiVitesseMin;
uiVitesseMin = monMoteur.getConfigVitesseMin();
```

14. getConfigVitesseMax()

Permet de récupérer de la trame Config le paramètre de vitesse maximum. Retourne la valeur entière en cm/s de la vitesse maximum.

Prototype : `unsigned int getConfigVitesseMax();  // Récupère la vitesse maximale de la course`

Exemple :
```
unsigned int uiVitesseMax;
uiVitesseMax = monMoteur.getConfigVitesseMax();
```

15. getConfigSegmentVitesse()

Permet de récupérer de la trame Config le paramètre de vitesse d'un des 10 segments possibles de configuration de la courbe de vitesse. Retourne la valeur entière en cm/s de la vitesse du segment.

Prototype : `unsigned int getConfigSegmentVitesse(int numSegment);  // Récupère la vitesse finale du segment indiqué de la trame Config`

Exemple :
```
unsigned int uiSegmentVitesse;
Int iSegmentId = 3 ;
uiSegmentVitesse = monMoteur.getConfigSegmentVitesse(iSegmentId);
```

16. getConfigSegmentTemps()

Permet de récupérer de la trame Config le paramètre de temps (durée) d'un des 10 segments possibles de configuration de la courbe de vitesse. Retourne la valeur entière en ms de la durée du segment.

Prototype : `unsigned int getConfigSegmentTemps (int numSegment);  // Récupère le Temps final du segment indiqué de la trame Config`

Exemple :
```
unsigned int uiSegmentTemps;
Int iSegmentId = 3 ;
uiSegmentTemps = monMoteur.getConfigSegmentTemps(iSegmentId);
```

17. getConfigVitesseLente()

Permet de récupérer de la trame Config le paramètre de vitesse lente, celle de la marche arrière. Retourne la valeur entière en cm/s de la vitesse lente.

Prototype : `unsigned int getConfigVitesseLente();  // Récupère la vitesse lente de la trame Config`

Exemple :
```
unsigned int uiVitesseLente;
uiVitesseLente = monMoteur.getConfigVitesseLente();
```

18. getConfigTimeHeures()

Permet de récupérer de la trame Config le paramètre Heure du temps moteur. Retourne la valeur entière des heures, entre 0 et 23.

Prototype :
```
unsigned int getConfigTimeHeures();     // Récupère la vitesse lente de la trame Config
```

Exemple :
```
unsigned int uiHeures;
uiHeures = monMoteur.getConfigTimeHeures();
```

19. getConfigTimeMinutes()

Permet de récupérer de la trame Config le paramètre minute du temps moteur. Retourne la valeur entière des minutes, entre 0 et 59.

Prototype :
```
int getConfigTimeMinutes();             // Récupère les minutes du temps actuel de la trame Config
```

Exemple :
```
unsigned int uiMinutes;
uiMinutes = monMoteur.getConfigTimeMinutes();
```

20. getConfigTimeSecondes()

Permet de récupérer de la trame Config le paramètre seconde du temps moteur. Retourne la valeur entière des secondes, entre 0 et 59.

Prototype :
```
int getConfigTimeSecondes();            // Récupère les secondes du temps actuel de la trame Config
```

Exemple :
```
unsigned int uiSecondes;
uiMSecondes = monMoteur.getConfigTimeSecondes();
```

21. getConfigTimeMillisecondes()

Permet de récupérer de la trame Config le paramètre milliseconde du temps moteur. Retourne la valeur entière des millisecondes, entre 0 et 999.

Prototype :
```
int getConfigTimeMillisecondes();               // Récupère les secondes du temps actuel de la trame Config
```

Exemple :
```
unsigned int uiMillisecondes;
uiMillisecondes = monMoteur.getConfigTimeMillisecondes();
```

22. setConfigTimeCourse()

Permet de configurer dans la trame Config le paramètre Temps de course du moteur, en secondes.

Prototype :
```
void setConfigTimeCourse(float Temps);          // Défini le temps de course
```

Exemple :
```
float fTimeCourse = 2.5f;
monMoteur.setConfigTimeCourse(fTimeCourse);
```

23. setConfigSpeed()

Permet de configurer dans la trame Config les paramètres Vitesse et Temps (durée) de chaque zone de course du moteur.

Prototype :
```
void setConfigSpeed(float Speed, float Temps, int zone); // Défini la vitesse pour une zone
```

Exemple :
```
float fSpeed = 2.5f;    // Vitesse à atteindre dans la zone en m/s
float fTemps = 1.0f;    // Durée de la zone en s
int iZone = 0;
monMoteur.setConfigSpeed(fSpeed, fTemps, iZone);
```

24. setConfigDiametreRoues()

Permet de configurer dans la trame Config le paramètre de Diamètre des roues du moteur.

Prototype :
```
void setConfigDiametreRoues(float DiamRoues);           // Défini le diamètre des roues
```

Exemple :
```
float fDiametreRoue = 62.5f;    // Diamètre des roues en mm
monMoteur.setConfigDiametreRoues(fDiametreRoue);
```

25. setConfigLongueurPiste()

Permet de configurer dans la trame Config le paramètre de longueur de la piste de course.

Prototype :
```
void setConfigLongueurPiste(float LongueurPiste);        // Défini la longueur de piste
```

Exemple :
```
float fLongueurPiste = 15.0f;   // Longueur de piste en m
monMoteur.setConfigLongueurPiste(fLongueurPiste);
```

26. setConfigMotorisationType()

Permet de configurer dans la trame Config le paramètre de type de motorisation du moteur, traction ou propulsion ; cela détermine le sens de rotation des roues en marche avant.

Prototype :
```
void setConfigMotorisationType(int Choix);              // Défini le type de motorisation
```

Paramètre :
```
CEC_TYPE_TRACTION               //  0 : Type de motorisation = TRACTION
CEC_TYPE_PROPULSION             //  1 : Type de motorisation = PROPULSTION
```

Exemple :
```
monMoteur. setConfigMotorisationType(CEC_TYPE_PROPULSION);
```

27. setConfigMotorisationType()

Permet de configurer dans la trame Config le paramètre de type de motorisation du moteur, traction ou propulsion ; cela détermine le sens de rotation des roues en marche avant.

Prototype :
```
void setConfigMotorisationType(int Choix);              // Défini le type de motorisation
```

Paramètre :
```
CEC_TYPE_TRACTION               //  0 : Type de motorisation = TRACTION
CEC_TYPE_PROPULSION             //  1 : Type de motorisation = PROPULSTION
```

Exemple :
```
monMoteur. setConfigMotorisationType(CEC_TYPE_PROPULSION);
```

28. setConfigCourseEndType()

Permet de configurer dans la trame Config le paramètre de type de fin de course, arrêt ligne d'arrivée ou bien damier. Ce dernier n'est pas implémenté, il ne doit pas être utilisé.

Prototype :
```
void setConfigCourseEndType(int EndType);               // Défini le type de fin de course
```

Paramètre :
```
CEC_TYPE_ARRET_LIGNE            //  1 : Type de fin de course = ARRET_LIGNE
CEC_TYPE_DAMIER                 //  2 : Type de fin de course = DAMIER
```

Exemple :
```
monMoteur.setConfigCourseEndType(CEC_TYPE_ARRET_LIGNE);
```

E. Méthodes « Move »

Les méthodes « Move sont » relatives à la trame Move ; elles sont utilisées pour agir sur le mouvement immédiat du moteur.

1. moveStart()

Permet de démarrer la course suivant la courbe de vitesse définie dans les segments (voir la partie Config) après un certain temps.

Prototype :
```
int moveStart(int temps_ms);    // Démarre la course après un délai de temps_ms (en ms)
```

Retour :
```
CEC_ERR_KO_F1                   // -7 : erreur de réception message par le moteur
CEC_ERR_SERIAL_NOT_AVAILABLE    // -5 : erreur liaison série moteur
CEC_ERR_LENGTH                  // -4 : erreur de longueur trame
CEC_ERR_WRONG_CMD               // -3 : erreur de réception code commande
CEC_ERR_TIMEOUT                 // -2 : erreur de timeout (délai d'attente trop long)
CEC_ERR                         // -1 : erreur générique, début de trame erronée
CEC_ERR_OK                      //  0 : pas d'erreur
CEC_ERR_WRONG_CHECKSUM          //  1 : Ok mais mauvais checksum
CEC_ERR_OK_F0                   //  2 : Ok avec retour correct du moteur (F0)
```

Exemple :
```
int temps_ms = 1000;            // défini un délai de 1s
monMoteur.moveStart(temps_ms);  // démarre après le délai temps_ms
```

2. moveStop()

Permet d'arrêter la course après un certain temps.

Prototype :
```
int moveStop(int temps_ms);     // arrête la course après un délai de temps_ms (en ms)
```

Exemple :
```
int temps_ms = 1000;            // défini un délai de 1s
monMoteur.moveStop(temps_ms);   // arrêt moteur après le délai temps_ms
```

3. movePause()

Permet de stopper la course pendant un certain temps puis de la reprendre au tout début.

Prototype :
```
int movePause(int temps_ms);    // Pause moteur pendant un délai de temps_ms (en ms)
```

Exemple :
```
int temps_ms = 1000;            // défini un délai de 1s
```

```
monMoteur.movePause(temps_ms);  // Pause moteur pendant le délai temps_ms
```

4. moveSpeedLimit()

Permet de converger rapidement à 9,81 m/s² vers la vitesse limite définie, jusqu'à la fin du segment de course en cours. La vitesse est bridée à 12 m/s.

Prototype :
```
int moveSpeedLimit(float Speed);        // Défini la limitation de vitesse en m/s
```

Exemple :
```
float fSpeedLimit = 2.0f;               // défini une vitesse limite de 2.0 m/s
monMoteur.moveSpeedLimit(fSpeedLimit);  // limitation de la vitesse moteur jusqu'à la fin du segment
```

5. configMoveForward()

Permet de converger en marche avant vers la vitesse définie en cm/s dans le temps défini en ms.

Prototype :
```
int configMoveForward(unsigned int temps_ms, unsigned int Speed); // Avance à la vitesse moteur
```

Exemple :
```
unsigned int uiTemps_ms = 1000;                   // défini un temps de 1s
unsigned int uiSpeed = 200;                       // défini une vitesse de 200 cm/s (positif)
monMoteur.configMoveForward(uiTemps_ms, uiSpeed); // avance, converge vers la vitesse iSpeed en
temps_ms
```

6. configMoveBackward()

Permet de converger en marche arrière vers la vitesse définie en cm/s dans le temps défini en ms.

Prototype :
```
int configMoveBackward(unsigned int temps_ms, unsigned int Speed); // Recule à la vitesse moteur
```

Exemple :
```
int temps_ms = 1000;                            // défini un temps de 1s
int iSpeed = 200;                               // défini une vitesse de 200 cm/s
monMoteur.configMoveBackward(temps_ms, iSpeed); // recule, converge vers la vitesse iSpeed en temps_ms
```

F. Méthodes « Mesure »

Les méthodes « Mesure » sont relatives à la trame Mesure ; elles sont utilisées pour récupérer la dernière série de mesures connues du moteur, en temps réel.

1. ReadMesures()

Permet de récupérer la trame Mesure du moteur avant de lire les paramètres de la trame par les méthodes `getMesure_()`.

Prototype :
```
int ReadMesures();              // Trame du dernier ensemble de mesures: réception des valeurs
```

Retour :
```
CEC_ERR_KO_F1                   // -7 : erreur de réception message par le moteur
CEC_ERR_SERIAL_NOT_AVAILABLE    // -5 : erreur liaison série moteur
CEC_ERR_LENGTH                  // -4 : erreur de longueur trame
CEC_ERR_WRONG_CMD               // -3 : erreur de réception code commande
CEC_ERR_TIMEOUT                 // -2 : erreur de timeout (délai d'attente trop long)
CEC_ERR                         // -1 : erreur générique, début de trame erronée
CEC_ERR_OK                      //  0 : pas d'erreur
CEC_ERR_WRONG_CHECKSUM          //  1 : Ok mais mauvais checksum
CEC_ERR_OK_F0                   //  2 : Ok avec retour correct du moteur (F0)
```

Exemple :
```
monMoteur.ReadMesures();        // Récupère les dernières mesures du moteur
```

2. getMesureNum()

Permet de récupérer de la trame Mesure le numéro de la série de mesures.

Prototype :
```
unsigned int getMesureNum();    // Récupère le numéro de la mesure
```

Exemple :
```
unsigned int uiNumMesure;
uiNumMesure = monMoteur.getMesureNum();
```

3. getMesureInst()

Permet de récupérer de la trame Mesure l'instant de la trame mesure.

Prototype :
```
unsigned int getMesureInst();   // Récupère l'instant 1 de la mesure
```

Exemple :
```
unsigned int uiInstantMesure;
uiInstantMesure = monMoteur.getMesureInst();
```

4. getMesureInstant()

Permet de récupérer de la trame Mesure l'instant de la mesure.

Prototype :
```
unsigned int getMesureInstant();// Récupère l'instant 2 de la mesure
```

Exemple :
```
unsigned int uiInstantMesure;
uiInstantMesure = monMoteur.getMesureInstant();
```

5. getMesureAccel()

Permet de récupérer de la trame Mesure les mesures de l'accéléromètre suivant l'axe indiqué.

Prototype :
```
int getMesureAccel(int axe);    // Récupère les valeurs de l'accéléromètre
```

Paramètre :
```
CEC_ACCEL_AXE_X                 //  1 : Accelerometre, axe X (longitudinal)
CEC_ACCEL_AXE_Y                 //  2 : Accelerometre, axe Y (lateral)
CEC_ACCEL_AXE_Z                 //  3 : Accelerometre, axe Z (vertical)
```

Exemple :
```
int iAccelValX, iAccelValY, iAccelValZ;
iAccelValX = monMoteur.getMesureAccel(CEC_ACCEL_AXE_X);
iAccelValY = monMoteur.getMesureAccel(CEC_ACCEL_AXE_Y);
iAccelValZ = monMoteur.getMesureAccel(CEC_ACCEL_AXE_Z);
```

6. getMesureGyro()

Permet de récupérer de la trame Mesure les mesures du gyromètre suivant l'axe indiqué.

Prototype :
```
int getMesureGyro(int axe);     // Récupère les valeurs du gyromètre
```

Paramètre :
```
CEC_GYRO_AXE_X                  //  1 : Gyrometre, axe X (longitudinal)
CEC_GYRO_AXE_Y                  //  2 : Gyrometre, axe Y (lateral)
CEC_GYRO_AXE_Z                  //  3 : Gyrometre, axe Z (vertical)
```

Exemple :
```
int iGyroValX, iGyroValY, iGyroValZ;
iGyroValX = monMoteur.getMesureGyro(CEC_GYRO_AXE_X);
iGyroValY = monMoteur.getMesureGyro(CEC_GYRO_AXE_Y);
iGyroValZ = monMoteur.getMesureGyro(CEC_GYRO_AXE_Z);
```

7. getMesureMagneto()

Permet de récupérer de la trame Mesure les mesures du magnétomètre suivant l'axe indiqué.

Prototype :
```
int getMesureMagneto(int axe);  // Récupère les valeurs du magnétomètre
```

Paramètre :
```
CEC_MAGNETO_AXE_X               //  1 : Magnetometre, axe X (longitudinal)
CEC_MAGNETO_AXE_Y               //  2 : Magnetometre, axe Y (lateral)
```

Exemple :
```
int iMagnetoValX, iMagnetoValY;
iMagnetoValX = monMoteur.getMesureMagneto(CEC_MAGNETO_AXE_X);
iMagnetoValY = monMoteur.getMesureMagneto(CEC_MAGNETO_AXE_Y);
```

8. getMesureRpm()

Permet de récupérer de la trame Mesure la mesure de vitesse pignon moteur en tours par minute (Révolution Par Minute = RPM).

Prototype :
```
unsigned int getMesureRpm();            // Récupère la vitesse de rotation du pignon moteur
```

Exemple :
```
unsigned int uiRotPignonMoteur;
uiRotPignonMoteur = monMoteur.getMesureRpm();
```

9. getMesureSpeed()

Permet de récupérer de la trame Mesure la mesure de vitesse moteur vue par les roues en cm/s.

Prototype :
```
unsigned int getMesureSpeed()           // Récupère la vitesse moteur à la roue
```

Exemple :
```
unsigned int uiVitesseMoteur;
uiVitesseMoteur = monMoteur.getMesureSpeed();
```

10. getMesureDistance()

Permet de récupérer de la trame Mesure la mesure de distance parcourue en cm.

Prototype :
```
unsigned int getMesureDistance();       // Récupère la distance parcourue
```

Exemple :
```
unsigned int uiDistance_cm;
uiDistance_cm = monMoteur.getMesureDistance();
```

11. getMesureColor()

Permet de récupérer de la trame Mesure la mesure de couleur sol détectée par le capteur RGB.

Prototype :
```
unsigned int getMesureColor();  // Récupère la couleur détectée
```

Retour :
```
RGB_BLACK                       //  0 : Couleur NOIR
RGB_BLUE                        //  1 : Couleur BLEU
RGB_GREEN                       //  2 : Couleur VERT
RGB_CYAN                        //  3 : Couleur CYAN
RGB_RED                         //  4 : Couleur ROUGE
RGB_VIOLET                      //  5 : Couleur VIOLET
RGB_YELLOW                      //  6 : Couleur JAUNE
RGB_WHITE                       //  7 : Couleur BLANC
```

Exemple :
```
unsigned int uiMesColor;
uiMesColor = monMoteur.getMesureColor();
```

12. getMesureTension()

Permet de récupérer de la trame Mesure la mesure de tension batterie en mV.

Prototype :
```
unsigned int getMesureTension();        // Récupère la tension batterie
```

Exemple :
```
unsigned int uiValTension_mV;
uiValTension_mV = monMoteur.getMesureTension();
```

13. getMesureCourant()

Permet de récupérer de la trame Mesure la mesure de courant moteur par unité de 10 mA.

Prototype :
```
unsigned int getMesureCourant();        // Récupère le courant batterie
```

Exemple :
```
unsigned int uiValCourant_10mA;
uiValCourant_10mA = monMoteur.getMesureCourant();
```

14. getMesureTemp()

Permet de récupérer de la trame Mesure la mesure de température moteur par unité de 0,1 °C.

Prototype :
```
unsigned int getMesureTemp();           // Récupère la température
```

Exemple :
```
unsigned int uiTemperature;
uiTemperature = monMoteur.getMesureTemp();
```

15. getMesureCourseEtat()

Permet de récupérer de la trame Mesure l'état de course. Cette fonction n'est peut-être pas implémentée dans le moteur actuellement.

Prototype :
```
unsigned int getMesureCourseEtat();     // Recupère l'état de course
```

Exemple :
```
unsigned int uiCourseEtat;
uiCourseEtat = monMoteur.getMesureCourseEtat();
```

16. getMesureCourseSegment()

Permet de récupérer de la trame Mesure le numéro de segment de course. Cette fonction n'est peut-être pas implémentée dans le moteur actuellement.

Prototype :
```
unsigned int getMesureCourseSegment();              // Recupere le segment de course
```

Exemple :
```
unsigned int uiCourseSegment;
uiCourseSegment = monMoteur.getMesureCourseSegment();
```

G. Méthodes Bilan

Les méthodes « Bilan » sont relatives à la trame Bilan ; elles sont utilisées pour récupérer les données de course en fin de course ainsi que pour suivre l'état de course en temps réel.

1. ReadBilan()

Permet de récupérer la trame Bilan du moteur avant de lire les paramètres de la trame par les méthodes `getBilan_()`.

Prototype :
```
int ReadBilan();                // Trame Bilan: réception des valeurs
```

Retour :
```
CEC_ERR_KO_F1                   // -7 : erreur de réception message par le moteur
CEC_ERR_SERIAL_NOT_AVAILABLE    // -5 : erreur liaison série moteur
CEC_ERR_LENGTH                  // -4 : erreur de longueur trame
CEC_ERR_WRONG_CMD               // -3 : erreur de réception code commande
CEC_ERR_TIMEOUT                 // -2 : erreur de timeout (délai d'attente trop long)
CEC_ERR                         // -1 : erreur générique, début de trame erronée
CEC_ERR_OK                      //  0 : pas d'erreur
CEC_ERR_WRONG_CHECKSUM          //  1 : Ok mais mauvais checksum
CEC_ERR_OK_F0                   //  2 : Ok avec retour correct du moteur (F0)
```

Exemple :
```
monMoteur.ReadBilan();          // Récupère les dernières mesures du moteur
```

2. getBilanFinalTime()

Permet de récupérer de la trame Bilan le temps de course en cours, ainsi que le temps final une fois la course finie.

Prototype :
```
unsigned int getBilanFinalTime();       // Récupère le temps final de la course
```

Exemple :
```
unsigned int uiCourseTemps;
uiCourseTemps = monMoteur.getBilanFinalTime();
```

3. getBilanFinalDistance()

Permet de récupérer de la trame Bilan la distance parcourue de course en cours, ainsi que la distance finale parcourue une fois la course finie, en cm.

Prototype :
```
unsigned int getBilanFinalDistance();   // Récupère la distance finale de la course
```

Exemple :
```
unsigned int uiCourseDistance_cm;
uiCourseDistance_cm = monMoteur.getBilanFinalTime();
```

4. getBilanFinalVmoy()

Permet de récupérer de la trame Bilan une fois la course finie la vitesse moyenne finale, en cm/s.

Prototype :
```
unsigned int getBilanFinalVmoy();       // Récupère la vitesse moyenne finale de la course
```

Exemple :
```
unsigned int uiCourseVmoy_cm_s;
uiCourseVmoy_cm_s = monMoteur.getBilanFinalVmoy();
```

5. getBilanFinalVmax()

Permet de récupérer de la trame Bilan une fois la course finie la vitesse maximale finale, en cm/s.

Prototype :
```
unsigned int getBilanFinalVmax();       // Récupère la vitesse maximale finale de la course
```

Exemple :
```
unsigned int uiCourseVmax_cm_s;
uiCourseVmax_cm_s = monMoteur.getBilanFinalVmax();
```

6. getBilanFinalCumulCourant()

Permet de récupérer de la trame Bilan une fois la course finie le cumul de courant, en 0.01 A.s.

Prototype :
```
unsigned int getBilanFinalCumulCourant();       // Récupère le cumul de courant de la course
```

Exemple :
```
unsigned int uiCourseIcumul_10mA_s;
uiCourseIcumul_10mA_s = monMoteur.getBilanFinalCumulCourant();
```

7. getBilanFinalPconso()

Permet de récupérer de la trame Bilan une fois la course finie la puissance consommée, en 0.1 J (Joules).

Prototype :
```
unsigned int getBilanFinalPconso();     // Récupère la puissance consommée de la course
```

Exemple :
```
unsigned int uiCoursePconso_100mJ;
uiCoursePconso_100mJ = monMoteur.getBilanFinalPconso();
```

8. getBilanTypeDepart()

Permet de récupérer de la trame Bilan une fois la course finie le type de départ.

Prototype :
```
unsigned int getBilanTypeDepart();      // Récupère le type de départ
```

Retour :
```
CEC_TYD_NONE                    //  0 : départ non défini
CEC_TYD_IMM                     //  1 : départ immédiat
CEC_TYD_PROG                    //  2 : départ programmé
```

Exemple :
```
unsigned int uiTypeDepart;
uiTypeDepart = monMoteur.getBilanTypeDepart();
```

9. getBilanTypeFin()

Permet de récupérer de la trame Bilan une fois la course finie le type de fin de course.

Prototype :
```
unsigned int getBilanTypeFin();         // Récupère le type d'arrivée
```

Retour :
```
CEC_TYF_NONE                    //  0 : type de fin de course non défini
CEC_TYF_FIN_ZONE                //  1 : fin des zones définies
CEC_TYF_LIGNE_FIN               //  2 : détection ligne de fin
CEC_TYF_ARRET_IMM               //  3 : arrêt immédiat
CEC_TYF_DIST_MAX                //  4 : distance max
CEC_TYF_TEMPS_MAX               //  5 : temps max
```

Exemple :
```
unsigned int uiTypeFin;
uiTypeFin = monMoteur.getBilanTypeFin();
```

10. getBilanFinalAccelMoy()

Permet de récupérer de la trame Bilan une fois la course finie l'accélération moyenne en cm/s².

Prototype :
```
unsigned int getBilanFinalAccelMoy();   // Récupère l'acceleration moyenne finale de la course
```

Exemple :
```
unsigned int uiAccelMoy_cm_s2;
uiAccelMoy_cm_s2 = monMoteur.getBilanFinalAccelMoy();
```

11. getBilanAccel()

Permet de récupérer de la trame Bilan l'accélération longitudinale actuelle en course vue par l'accéléromètre, en m/s².

Prototype :
```
float getBilanAccel();          // Récupère l'acceleration
```

Exemple :
```
float fAcceleroAccel;
fAcceleroAccel = monMoteur.getBilanAccel();
```

12. getBilanVitesse()

Permet de récupérer de la trame Bilan la vitesse longitudinale actuelle en course vue par l'accéléromètre, en m/s.

Prototype :
```
float getBilanVitesse();        // Récupère la vitesse
```

Exemple :
```
float fAcceleroVitesse;
fAcceleroVitesse = monMoteur.getBilanVitesse();
```

13. getBilanPosition()

Permet de récupérer de la trame Bilan la position longitudinale actuelle en course vue par l'accéléromètre, en m/s.

Prototype :
```
float getBilanPosition();       // Récupère la position
```

Exemple :
```
float fAcceleroVitesse;
fAcceleroVitesse = monMoteur.getBilanPosition();
```

14. getBilanCourseEtat()

Permet de récupérer de la trame Bilan l'état de course actuelle. En cours de course, les états sont généralement CEC_SEQ_COU_COU, ou CEC_SEQ_COU_CAL au passage de segment à segment.

Prototype :
```
byte getBilanCourseEtat();      // Récupère l'état actuel de course
```

Retour :
```
CEC_SEQ_COU_RIEN                //  0 : (non défini)
CEC_SEQ_COU_ATT                 //  1 : attente ordre course ou mouvement
CEC_SEQ_COU_DEM                 //  2 : initialisation pour une course
CEC_SEQ_COU_CAL                 //  3 : calcul
CEC_SEQ_COU_CAPT_ON             //  4 : activation carte capteur
CEC_SEQ_COU_COU                 //  5 : déroulement de la course
CEC_SEQ_COU_FIN_0               //  6 : dernier pas de la course
CEC_SEQ_COU_FIN_LIGNE           //  7 : pas après passage de la ligne d'arrivée
CEC_SEQ_COU_FIN                 //  8 : fin de la course
CEC_SEQ_COU_ANNULE              //  9 : annulation
CEC_SEQ_COU_RECUL_INIT          // 10 : initialisation pour le recul de la voiture
CEC_SEQ_COU_AVANCE_INIT         // 11 : initialisation pour l'avance de la voiture
CEC_SEQ_COU_POSITIONNE          // 12 : positionne la voiture
CEC_SEQ_COU_FIXE                // 13 : vitesse fixe
```

Exemple :
```
byte courseEtat;
courseEtat = monMoteur.getBilanCourseEtat();
```

15. getBilanCourseZone()

Permet de récupérer de la trame Bilan le numéro de zone (segment) actuelle en cours de course, compris entre 0 pour le 1er et 9 pour le dixième.

Prototype :
```
byte getBilanCourseZone();      // Récupère la zone actuelle de course
```

Exemple :
```
byte courseZone;
courseZone = monMoteur.getBilanCourseZone();
```