Méthodes de la bibliothèque arduino-pinchangeint.h.

Note placée en page sur les interruptions dans SYNTAXE.

NOTE : Les ATMega328 peuvent également gérer des interruptions de changement d'état sur vingt de leurs broches. Mais leur traitement n'est pas aussi simple que pour les interruptions externes car il faut aiguiller le code après avoir déterminé d'ou vient le requête. La librairie Arduino arduino-pinchangeint.h a été développée afin de permettre l'utilisation de ces interruptions internes au µP.

Synthèse d'utilisation des diverses bibliothèques.

Librairie Serial : Communications série USB avec la carte Arduino.
Wire.h : Protocoles de communication Arduino par ligne I2C / TWI.
I2Cdev.h : Complète Wire.h pour des C.I. électroniques spécifiques.
EEPROM.h : Lire et écrire dans l'EEPROM du µP d'Arduino.
LedControl.h : Gestion par la SPI d'un multiplexeur type MAX7219.
Servo.h : Pilotage angulaire de servomoteurs par PWM.
Stepper.h : Pilotage "direct" des bobines d'un moteur pas à pas.
AFMotor.h : Gestion d'une carte électronique ADAFRUIT motors.
AccelStepper.h : Complément de Stepper.h > Voir page 11.
LiquidCrystal.h : Gestion d'afficheurs LCD avec "chipset HD44780".
Perso_ST7565.h : Gestion de l'afficheur rétroéclairé **RBV** ST7565.
Adafruit_ssd1306syp.h : Gestion de l'afficheur OLED 128 x 64.
avr/eeprom.h : Remplace avantageusement EEPROM.h.
avr/pgmspace.h : Stocker des constantes en mémoire de programme.
SD.h : Permet de lire et d'écrire sur des cartes mémoires SD et SDHC.

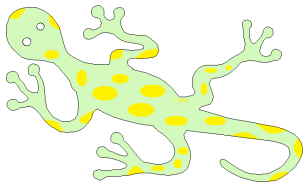
TERMINOLOGIE.

Bibliothèque : Ensemble de procédures utilitaires gérant un domaine de la programmation, et mises à disposition des utilisateurs d'Arduino.

- Library est un faux ami. La traduction française est Bibliothèque et non pas Librairie comme trop souvent rencontré.
- **SHIELD** *(Bouclier.)* : Petite carte électronique spécifique dédiée à une application dont l'agencement permet le branchement "gigogne" directement sur les connecteurs d'une carte Arduino.
- **Breakout board** *(Carte contrôleur.)* : Module circuit imprimé, souvent de très petites dimensions, dédié généralement à un circuit intégré spécialisé complexe pour le rendre directement utilisable dans l'environnement d'Arduino.

LES BIBLIOTHÈQUES

TABLE DES MATIÈRES



Bibliothèque Serial pour la communication série.

La librairie Serial est utilisée pour les communications par le port série entre la carte Arduino et un ordinateur ou d'autres composants. Toutes les cartes Arduino ont au moins un port Série qui communique sur les broches 0 (RX) et 1 (TX) avec l'ordinateur via la ligne USB. Si cette fonctionnalité est utilisée dans le programme, les broches 0 et 1 en tant qu'entrées ou sorties numériques ne sont plus disponibles. Le terminal série intégré à l'environnement Arduino, dont on peut définir le débit, communique directement avec une carte Arduino. Noter que la carte **Arduino Mega** , oute Serial sur 0 et 1 dispose de trois port série supplémentaires : Serial1 sur les broches 19 (RX) et 18 (TX), Serial2 sur les broches 17 (RX) et 16 (TX), Serial3 sur 15 (RX) et 14 (TX).

NOTE : Si une carte **Arduino Mega** est utilisée, pour utiliser l'un des trois ports supplémentaires il suffit de faire suivre le mot clef Serial de 1, 2 ou 3. Exemple : **Serial1**.begin(300); **Serial2**.begin(9600);

Serial.begin(Débit);

Fixe le débit de communication en baud pour l'UART du µP. Les débits standard sont respectivement 300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. On peut toutefois spécifier d'autres débits pour communiquer sur les broches 0 et 1 avec un composant qui nécessite une vitesse particulière.

Serial.flush();

Anciennement vidait le buffer de réception des données du port série. Actuellement c'est une fonction "inhibante" qui permet de s'assurer que le tampon de transmission soit vide avant de repasser la main.

Serial.available();

Fonction qui retourne le nombre d'octets disponibles pour lecture dans la file d'attente *(Buffer)* du port série, ou 0 si elle est vide. Si une donnée est arrivée, Serial.available() sera supérieur à 0. La file d'attente de la PILE FIFO peut contenir jusqu'à 63 octets.

Serial.read();

Fonction de type int qui retourne le premier octet de donnée entrant disponible dans le buffer du port série, ou -1 si aucune donnée n'est disponible. L'octet lu est enlevé de la file d'attente qui est de type FIFO. Le prochain appel à Serial.read() lira l'octet suivant etc ...

Serial.end();

Désactive la communication série, ce qui permet de se servir des broches RX et TX comme entrées /sorties générales. Pour réactiver la communication série, il suffit de réutiliser **Serial**.begin(Débit).

Serial.peek();

Fonction de type int qui retourne le premier octet de donnée entrant disponible sans l'extraire du FIFO de l'UART, ou -1 si aucune donnée n'est disponible. Des appels successifs à peek() retournent le même caractère, tout comme le prochain appel à la fonction read().

Serial.write(Donnée); *(Donnée : Valeur ASCII ou Chaîne)*

Écrit les données binaires sur le port série. Ces données sont envoyées comme un octet ou une série d'octets. Pour envoyer les caractères d'une variable numérique utiliser la fonction Serial.print (). >>> Pour les formats voir page 47 livret sur la SYNTAXE.

Serial.print(Donnée,Format)

Affiche les données sur le port série sous une forme claire pour un utilisateur Lambda. *(Texte ASCII)* Plusieurs formes sont possibles :

- Les nombres entiers sont affichés de façon banale.
- Les nombres à virgules float sont affichés de la même manière **avec par défaut deux décimales après "la virgule"**.
- Les types byte sont affichés sous la forme d'un caractère ASCII.
- Les caractères 'C' et les chaînes "CCC" sont affichés directement.

>>> Pour les formats voir page 47 livret sur la SYNTAXE.
>>> Pour les caractères spéciaux voir page 48 livret sur la SYNTAXE.

NOTE : Pour les formats d'un float, utiliser BIN, OCT, DEC ou HEX va imposer un nombre de décimales respectivement égal à 2, 8, 10 et 16 tout en limitant le nombre de chiffres significatifs à sept ou huit.

La procédure dédiée serialEvent().

Ce n'est qu'une petite subtilité : Si on donne à une procédure l'identificateur réservé "serialEvent()" cette dernière sera **invoquée automatiquement** en tout début de la boucle de base Loop sans avoir à insérer une instruction pour son appel. *(Attention à la casse.)* Rien n'interdit de donner à notre routine un nom différent, et à en faire l'appel où bon nous semble dans code source du logiciel. Le petit programme ESSAI_de_SERIAL-EVENT.ino propose un exemple d'utilisation.

Méthodes de la bibliothèque Wire.h. (Compléments en page 6)

Cette librairie permet de communiquer avec des composants utilisant le protocole I2C / TWI. *(Communication série sur 2 fils.)*

Connexions. Sur la plupart des cartes Arduino :

SDA *(Ligne de données)* est sur la broche analogique A4.
SCL *(Signal d'horloge)* est sur la broche analogique A5.
Sur Arduino Mega : SDA est sur Numérique 20 et SCLsur N21.

Fonctions d'initialisation :

- begin() initialise la communication avec un Arduino "maître".
- begin(Adresse) initialise la liaison avec un Arduino "esclave".

Fonctions du "mode maître".

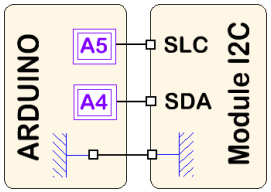
- requestFrom(Adresse, Quantité) : Demande de données à un module esclave.
- beginTransmission(Adresse) : Débute la communication avec un module esclave. *(Ouvre le stockage de données à envoyer avec write)*
- endTransmission() : Envoi des données vers l'esclave.
- write() : Écriture des données à envoyer vers l'esclave.
- available() : Test si des données sont disponibles sur l'esclave.
- read() : Lecture des données en provenance de l'esclave.

Fonctions du "mode esclave".

- write() : Envoie les données vers le maître après requête.
- available() : Test si des données sont disponibles en provenance du maître. *(cf onReceive)*
- read() : Lecture des données en provenance du maître.
- onReceive(Fonction) : Définit la fonction à appeler sur **réception** de données en provenance du maître.
- onRequest(Fonction) : Définit la fonction à appeler sur **requête** du module maître.
- send() et receive() sont actuellement des fonctions obsolètes.

Exemple d'utilisation :

```
#include <Wire.h>
void setup() {
  ...
  Wire.begin();
```



Les broches analogiques **A4** et **A5** sont imposées par l'usage de la bibliothèque Wire.h

Le bus I2C est commenté en détails sur :
http://www.monnetamoi.net/articles.php?lng=fr&pg=352

Compléments relatifs à la bibliothèque Wire.h.

Cette librairie est documentée en page 4.

Il existe plusieurs évolutions de la norme qui permettent de communiquer à des débits toujours plus importants :

Standard mode : ≤ 100 kbit/s, Fast mode : ≤ 400 kbit/s,
Fast plus mode : ≤ 1 Mbit/s, High-speed mode : ≤ 3,4 Mbit/s,
Ultra-fast mode : ≤ 5 Mbit/s

Pour nos réalisations nous allons surtout utiliser le **Standard mode** et éventuellement le "Fast mode". Pour Arduino, la bibliothèque Wire.h est par défaut configurée pour des échanges en **Standard mode**.

Les équipements connectés au bus dialoguent suivant un mécanisme maître/esclave. Un équipement esclave ne répond qu'à des demandes en provenance d'un maître. Il peut y avoir N maîtres et N esclaves sur un bus commun. Attention cependant, en cas d'utilisation de plusieurs maîtres, ces derniers doivent être capables de détecter qu'un autre maître utilise déjà le bus avant d'émettre. Un équipement ne peut pas être à la fois maître et esclave néanmoins il peut changer de rôle au cours du temps. *(Passer de maître à esclave et inversement)*

Toutes les spécifications du bus I2C sont décrites sans le document :
http://www.diolan.com/downloads/i2c-specification-version-4.0.pdf

Méthodes de la bibliothèque EEPROM.h. *(Voir @)*

L'EEPROM est une mémoire **non-volatile** *(Un kilo-octet sur l'ATmega328 : $0000 à $0400)* mise à la disposition du programmeur pour y stocker des données "constantes" destinées au long-terme.

Utilisation de la bibliothèque EEPROM.h.

Suite à la déclaration #include <EEPROM.h> on peut utiliser :

Octet = EEPROM.read(ADR);

Fonction qui lit la valeur contenue dans la cellule d'EEPROM à l'adresse ADR présente dans le l'EEPROM d'Arduino. Pour le premier octet ADR = $0000. Les emplacements qui n'ont jamais été écrit contiennent $FF. Elle retourne une donnée de type byte.

EEPROM.write(ADR,OCTET);

Procédure qui écrit le contenu du byte OCTET à l'emplacement EEPROM pointé par ADR. (Type int) L'EEPROM est limitée à environ 100 000 cycles d'écriture. Donc ne pas l'utiliser comme une RAM.

@ : Préférer la bibliothèque avr/eeprom.h décrite en page 32 optimisée en taille, en rapidité et qui permet de manipuler des données de tailles variables.

La bibliothèque I2Cdev.h.

Fondamentalement, la Bibliothèque de périphériques I2C est une collection de classes homogènes pour fournir des interfaces simples et intuitives à une variété toujours croissante de dispositifs I2C. Chaque appareil est conçu pour utiliser le code générique d'I2Cdev.h, qui permet de faire abstraction de la communication I2C au niveau du bit et au niveau de l'octet. Non spécifique à une classe d'appareil particulière, le code propre à un système est plus facile à "transporter" sur différentes plates-formes.
Le code relatif à I2Cdev.h est programmé pour être utilisé de manière statique, ce qui réduit les besoins en mémoire si vous avez plusieurs périphériques I2C inclus dans le projet en cours de développement. Une seule instance de la classe I2Cdev est nécessaire.

Cette bibliothèque I2Cdev.h n'est pas autonome, elle fait appel à Wire.h et doit être complétée par une bibliothèque spécifique au composant en communication I2C avec le programme en cours de développement. La logique d'un programme utilisant ces trois bibliothèques adopte une structure du type :

```
// Librairie requise pour communiquer avec des modules sur l'interface I2C.
#include "Wire.h" // (Bibliothèque installée par défaut sur Arduino.)
// Inclusion d'une bibliothèque permettant de communiquer avec un
// module particulier communiquant à travers I2C.
#include "I2Cdev.h"
// Inclusion d'une bibliothèque intégrant de nombreuses fonctions
// permettant d'accéder aux informations issues du module connecté :
// Dans ce programme un capteur accélérométrique et gyroscopique 3
// axes MPU6050. // (Librairy intégrée dans le "paquet MPU6050.h")
#include "MPU6050_6Axis_MotionApps20.h"
```

Puis suivent les déclarations propres au capteur, les initialisations et enfin les procédures et fonctions du programme.

Voici les structures de données pour I2Cdev.h avec une description disponible sur : http://www.i2cdevlib.com/docs/html/index.html

ADS1115	DateTime	HMC5883L	MPR121	SSD1308
ADXL345	DS1307	I2Cdev	MPU6050	TCA6424A
AK8975	HMC5843	ITG3200	MYDEVSTUB	

Téléchargeable sur http://www.3sigma.fr/telechargements/I2Cdev.zip

Méthodes de la bibliothèque LedControl.h.

Cette librairie facilite grandement la gestion par la SPI spécifique d'un multiplexeur spécialisé pour les afficheurs 7 segments ou les matrices de LED et nécessite un circuit intégré MAX7219 ou MAX7221.

Initialisations diverses.

LedControl Mon_afficheur = LedControl(Din,CLK,LOAD,Nb);
Crée une instance de ledcontrol en utilisant un identificateur associé. Les trois broches qui génèrent les signaux vers la ligne SPI du *(des)* MAX7219 sont listées dans l'ordre. Le nombre de modules Nb chaînés avec Dout sur la SPI termine la liste des paramètres. Si plusieurs multiplexeurs sont utilisés, il est plus logique de les chaîner par la SPI et utiliser un seul objet ledcontrol pour tous les contrôler.

Mon_afficheur.shutdown(Num,État);
Par défaut, à la mise sous tension le MAX7219 est en mode veille. Cette instruction permet de le réveiller avec false ou de le rendormir avec true. Num désigne le module concerné dans le chaînage éventuel, le premier étant le 0. Chaque circuit doit être commandé séparément.

Mon_afficheur.setIntensity(Num,Clarté);
Ajuste l'intensité lumineuse de chaque module. Clarté : De 0 à 15.

Mon_afficheur.clearDisplay(Num);
Efface *(Éteint)* l'intégralité de l'afficheur piloté.

Mon_afficheur.setScanLimit(Num,Limite); Par défaut le balayage se fait sur les huit "DIGIT". Mais si la luminorité est trop faible, on peut restreindre le balayage aux seuls afficheurs effectifs. Attention : Risque de destruction du MAX7219 par surintensité ! *(Voir notice)*

Modes d'affichage sur 7 segments ou sur Matrice de LED.

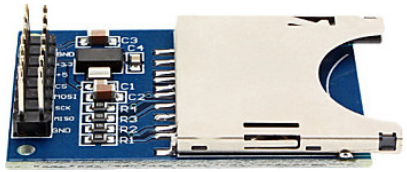
Mon_afficheur.setDigit(Num,Position,Chiffre,Dp);
S'utilise implicitement avec des afficheurs 7 segments sur lesquels on désire visualiser des chiffres. Le paramètre Dp qui vaudra true ou false allumera ou éteindra le point décimal du "DIGIT" associé. La valeur de Position est comprise entre zéro et sept.

Mon_afficheur.setLed(Num,Ligne,Colonne,État);
Instruction spécifique à la mise en œuvre de matrices de LED. Elle permet d'allumer ou d'éteindre une LED individuelle de l'afficheur en spécifiant ses coordonnées. Les valeurs de Ligne et de Colonne vont de 0 à 7. État conditionne la LED avec true ou false.

Mon_afficheur.setRow(Num,Rangée,Octet);
Instruction qui place directement les états précisés par Octet dans l'alignement Rangée *(Entre 0 et 7)* de l'objet ledcontrol Num.

La bibliothèque SD.h.

Dérivée de la bibliothèque sdfatlib.h de *William Greiman* qui propose davantage de fonctions, la bibliothèque SD.h est fournie en standard dans l'environnement **IDE** et s'avère déjà largement suffisante pour couvrir des applications très étoffées. Le tableau ci-dessous précise les branchements vers ARDUINO imposés par la bibliothèque, qui libèrent les deux lignes TX et RX ainsi qu'un maximum de sorties de type PWM.



$\overline{CS}$	Sélection de la carte	10
DI	Entrée série SPI. *(MOSI)*	11
VSS	Masse. *(GND)*	GND
VDD	+ Alimentation. *(+3.3)*	*Non branché*
CLK	Horloge de la SPI. *(SCK)*	13
DO	Données séries de la SPI. *(MISO)*	12

La librairie SD.h permet de lire et d'écrire sur des cartes mémoires SD et SDHC. Supportant les systèmes de fichier FAT16 et FAT 32 sur les cartes mémoires SD standard et les cartes SDHC. L'ouverture simultannée de plusieurs fichiers est possible. Les noms de fichiers doivent respecter le format **8.3** au standard assez ancien. Les noms de fichiers peuvent inclure des chemins séparés par des slash "/", le répertoire de travail étant toujours la racine de la carte SD, un nom se réfère au même fichier, qu'il inclue ou non un slash. *(Exemple :"/Mon_Fichier.txt" est équivalent à "Mon_Fichier.txt".)*

Cette librairie ouvre des possibilités de stockage de données considérables puisque les cartes SD disposent facilement de 8Go à 16Go d'espace. Elles peuvent être lues directement sur un PC. En enregistrant les données dans un fichier texte au format CSV, on rend possible un traitement des données enregistrées directement dans un tableur par exemple, avec réalisation de graphiques, etc...

Outre la directive incontournable #include <SD.h> on doit également compléter avec appel à une bibliothèque de gestion des protocoles de communication Arduino sur la ligne de dialogue I2C. Par exemple on ajoutera un #include <SPI.h> en tête de programme.

Méthodes de la classe SD.

Sd2Card Carte_SSD ; Procédure qui instancie la classe SD.
Sd2Volume Contenu ; Identifie la "directory".
Sd2Card Racine ; Identifie la racine de la "directory".
const byte ChipSelect = 10; Déclaration de l'identificateur utilisé.

Carte_SSD.init(Rapidité,ChipSelect);

Fonction qui précise par les identificateurs réservés la vitesse de communication sur la SPI. Rapidité peut prendre les valeurs :
SPI_FULL_SPEED : Fréquence CLK = Fréquence CPU / 2.
SPI_HALF_SPEED : Fréquence CLK = Fréquence CPU / 4.
QUARTER_SPEED : Fréquence CLK = Fréquence CPU / 8.
Retourne true si l'initialisation a réussie, false si non réalisée.

SD.begin(ChipSelect)

Fonction booléenne qui initialise la librairie SD et la carte SD et active l'utilisation du bus SPI *(Sorties binaires 11,12 et 13)* et initialise la broche ChipSelect qui par défaut est la broche matérielle SS. *(Broche 10 sur la plupart des cartes Arduino.)* Même si une broche de sélection différente est utilisée, la broche matérielle SS doit être laissée en SORTIE, sinon les méthodes de la librairie SD ne fonctionneront pas correctement. Le paramètre ChipSelect est optionnel si l'on conserve la broche 10 par défaut. La fonction retourne true si réussite et false si échec. Exemple d'utilisation :

```
if (!SD.begin(ChipSelect))
  { Serial.println("Echec de l'initialisation."); return; }
    Serial.println("Initialisation OK.");
```

SD.mkdir("DOSSIER/Prov")

Fonction qui crée **un répertoire** sur la carte mémoire SD ainsi que les répertoires intermédiaires qui n'existent pas encore. Retourne true si la création du chemin a réussi, et false s'il y a échec.

SD.rmdir("DOSSIER")

Fonction booléenne qui efface tout **un répertoire**, incluant les sous-répertoires délimités par des slashs "/". **Le répertoire doit être vide**. Retourne true si le répertoire a bien été effacé, et false s'il n'a pas pu être effacé. La valeur renvoyée n'est pas spécifiée si le répertoire n'existe pas.

.../...

File Mon_Fichier;

Le constructeur File permet de déclarer un objet de type File pour la librairie SD, objet qui représente un fichier. Un tel objet pourra recevoir notamment l'objet File renvoyé par la fonction SD.open().

Mon_Fichier = SD.open("ESSAI.TXT", Option)

Fonction booléenne qui ouvre **un fichier** sur la carte mémoire SD. Si le fichier est ouvert pour écriture, il sera créé s'il n'existe pas déjà. *(Le chemin de destination doit cependant exister.)*
Plusieurs fichiers peuvent être ouvert : *Voir note en bas de page.*
Les noms de fichiers présenteront huit lettres au maximum, et éventuellement un point suivi au maximum de trois lettres. Le mode par défaut est FILE_READ qui ouvre le fichier en positionnant le pointeur au début. FILE_WRITE ouvre le fichier en lecture et écriture, en démarrant au début du fichier. La fonction retourne true si l'ouverture a réussie, et false dans le cas contraire. Mon_Fichier peut donc être testé comme un booléen classique.

SD.exists("ESSAI.TXT")

Fonction booléenne qui teste la présence d'un répertoire *(Délimité par des slash "/".)* ou d'un fichier dont on passe en paramètre le nom.

SD.remove("DOSSIER/Avider.txt")

Fonction booléenne qui efface **un fichier** de la carte mémoire. Retourne true si le fichier a été effacé, et false s'il n'a pas pu être effacé. La valeur renvoyée n'est pas spécifiée si le fichier n'existe pas. Le nom du fichier peut inclure un chemin délimité par des slashs.

Méthodes de la classe FILE.

Mon_Fichier.available()

Fonction booléenne qui vérifie si des octets sont disponibles en lecture dans le fichier. Retourne le nombre d'octets disponibles ou zéro si le fichier est vide ou fermé.

Mon_Fichier.close();

Cette procédure ferme le fichier, et s'assure que toute donnée écrite dans ce fichier est physiquement enregistrée sur la carte mémoire SD. Comme la librairie SD.h ne peut ouvrir qu'un seul fichier à la fois, il faut fermer un fichier avant d'en ouvrir un autre.

NOTE : *Bien que non signalé sur les documentations mises en ligne, il semble parfaitement possible de travailler simultanément sur deux*

…/…

ou plusieurs fichiers. Il suffit d'utiliser plusieurs fois le constructeur File avec des identificateurs différents.

Mon_Fichier.flush();

Cette procédure s'assure que les données écrites dans un fichier ont été physiquement enregistrées sur la carte mémoire SD. Ceci est fait automatiquement également lorsque le fichier est fermé avec la fonction close();

Mon_Fichier.read();

Fonction qui lit un octet dans un fichier et retourne sa valeur sous forme d'un byte. Positionne le pointeur sur le prochain octet du fichier. Retourne -1 si octet indisponible.

Mon_Fichier.peek();

Fonction qui lit un octet dans un fichier et retourne sa valeur sous forme d'un byte. N'avance pas le pointeur sur le prochain octet du fichier. Retourne -1 si octet indisponible. *(Plusieurs appels successifs à fonction peek(); renverront la même valeur.)*

Mon_Fichier.size();

Fonction qui retourne la valeur de la taille du fichier en nombre d'octets sous forme d'un long. Le fichier doit être ouvert au moment où la fonction size() est invoquée.

Mon_Fichier.position();

Fonction qui retourne la valeur du pointeur de position dans le fichier sous forme d'un long. *(Correspond à la position où le prochain octet sera lu ou écrit.)*

Mon_Fichier.seek();

Cette fonction permet de placer le pointeur de lecture/écriture à une nouvelle position, qui doit être comprise entre 0 et la taille du fichier. *(Taille déclarée inclusive.)* Retourne true si réussite et false s'il y a échec. Le fichier doit être ouvert.

NOTE : Pour **effacer entièrement le contenu d'un fichier**, il suffit de l'effacer avec l'instruction SD.remove puis de le recréer avec SD.open.

…/…

```
Mon_Fichier.write(Data);
Mon_Fichier.write(Tableau, Nb);
```

Procédure prévue pour écrire des données Data dans un fichier préalablement ouvert. Data peut être un byte, un char ou une "Chaine de caracteres". Exemples :

```
①  const byte Octet = 65; // 'A' en ASCII.
②  char TABLEAU[5] {'5','4','3','2','1'};
③  Mon_Fichier.write(Octet);
④  Mon_Fichier.write('B');
⑤  Mon_Fichier.write("Texte");
⑥  Mon_Fichier.write(TABLEAU, 3);
⑦  Mon_Fichier.write(TABLEAU, 2);
```

La directive ① on crée une donnée de type byte. En ② on crée un tableau de cinq caractères ASCII. En ③ on écrit la valeur d'Octet dans le fichier. Puis en ④ on y ajoute le caractère 'B' suivi en ⑤ d'une chaîne de caractères. En ⑥ on copie les 3 premiers éléments du tableau. Comme cette syntaxe lit toujours à partir du début du tableau, en ⑦ on réécrit les 2 premiers éléments. Chaque écriture dans Mon_Fichier incrémente le pointeur de Lecture/Écriture.

```
Mon_Fichier.print(Data);
Mon_Fichier.print(Data, BASE);
```

Procédure pour pour écrire des données Data dans un fichier préalablement ouvert. Data peut être un byte, un char, un int, un long, un float ou une "Chaine de caracteres". Les nombres sont écrits comme des séquences de chiffres textuels ASCII. L'option BASE spécifie éventuellement la base dans laquelle est transformée le nombre : BIN, *(Base 2)* DEC, *(Base 10 par défaut)* OCT, *(Base 8)* et HEX. *(hexadécimal)*

```
Mon_Fichier.println();
Mon_Fichier.println(Data);
Mon_Fichier.println(Data, BASE);
```

Procédure totalement analogue à Mon_Fichier.print(Data), qui écrit des données Data dans un fichier préalablement ouvert et qui ajoure deux octets à la suite : "Retour chariot" + "passage à la ligne". Les formats d'écriture sont strictement identiques.

… / …

Afficher le contenu d'un fichier.

Faire afficher le contenu d'un fichier n'est pas prévu, car la présentation sera fonction de la nature des données qui y sont contenues. L'écriture d'une procédure dédiée est assez simple :

```
// On ouvre le fichier, ou on repositionne le pointeur au début :
Mon_Fichier.seek(0); // 0 ou plus pour commencer plus loin.
while (Mon_Fichier.available()
      {Serial.write(Mon_Fichier.read()); };
```

Il faut également vérifier l'existence du fichier. *(Voir ci-dessous.)*

Afficher le contenu de la carte mémoire.

Issue de la bibliothèque sdfatlib.h de ***William Greiman*** l'exemple de programme qui suit en utilise certaines fonctionnalités facilitant le traitement de la "directory" et, en particulier, permet l'exploration des sous-dossiers.

```
   // Vérification de la présence carte :
   if (!Carte_SSD.init(SPI_HALF_SPEED, ChipSelect)))
①     {Serial.println("Pas de carte SD."); return;}
   else {Serial.println("Carte presente dans le lecteur.");
   // Vérification des données :
   if (!Contenu.init(Carte_SSD))
②     {Serial.println("Carte non pas formatee."); return;}
   // Afficher le contenu de la carte SD :
   Serial.println("\nFichiers présents sur la carte :");
   Serial.println("Nom / date / Heure / taille");
   Racine.openRoot(&Contenu);   } Deux écritures
   Racine.openRoot(Contenu);    } possibles.
③  Racine.ls(LS_R | LS_DATE | LS_SIZE);
```

En cas d'échec en ① ou en ② il y a sortie anticipée de la procédure par le return, tout ce qui suit sera ignoré. En ③ les paramètres sont optionnels. LS_R si l'on désire les sous-dossiers, LS_DATE si l'on veut la date et l'heure et LS_SIZE si l'on veut la taille des fichiers.

COMPLÉMENTS : Aller sur http://www.mon-club-elec.fr/pmwiki_reference_arduino/pmwiki.php?n=Main.LibrairieSD pour avoir des informations complémentaires sur les fonctions name(), isDirectory(), openNextFile() et rewindDirectory().

Méthodes de la bibliothèque Servo.h.

Cette librairie permet à une carte Arduino de contrôler plusieurs servomoteurs de modélisme. En standard ils permettent de positionner et de maintenir l'axe à différents angles, habituellement entre 0° et 180°. Sur les cartes autres que la Mega, **l'utilisation de la librairie** Servo.h **désactive l'instruction** analogWrite() **sur les broches 9 et 10**, qu'il y ait ou non un servomoteur branché sur ces sorties.

- Le constructeur Servo déclare une variable de type Servomoteur. On peut déclarer autant de variable Servo qu'on utilise de moteurs.
  Ex : Servo Moteur_roulis ;
- L'instruction attach précise la broche de commande.
  Ex : Moteur_roulis.attach(3) ;
  On peut définir les limites de l'impulsion de commande correspondant aux rotations limites du moteur :
  NomMoteur.attach(broche, impls_min, impus_max); Par défaut les valeurs sont de 544 pour impuls_min et 2400 pour impuls_max.
  Ex : Moteur_roulis.attach(3,1000,2000) ;
- L'instruction detach désassocie la variable Servomoteur de sa broche. Si toutes les variables Servo sont libérées, alors les broches 9 et 10 peuvent être utilisées pour générer de la PWM avec analogWrite().
  Ex : Moteur_roulis.detach() ;
- L'instruction write provoque sur un moteur standard son positionnement à l'angle donné en paramètre. L'angle exprimé en degrés est compris entre 0° et 180°. La largeur de l'impulsion envoyée au servomoteur est éventuellement limitée par les paramètres imposés au préalable avec l'instruction attach(broche, impls_min, impls_max)
  Ex : Moteur_roulis.write(90) ; *(Poritionne le moteur au neutre)*
- L'instruction writeMicroseconds permet d'imposer la durée de l'impulsion au lieu de l'angle de positionnement.
  Ex : Moteur_roulis.writeMicroseconds(1500) ;
  Sur les moteurs standards, un paramètre de valeur 1000 µs est la position extrême dans un sens, 2000 est la position extrême dans le sens inverse et 1500 est au centre. De nombreux fabricants dépassent ces valeurs entre 700 et 2300 raison pour laquelle l'instruction attach() initialise par défaut entre 544 et 2400. Repousser éventuellement ces valeurs limites tant que le servomoteur continue à se positionner.

Méthodes de la bibliothèque Stepper.h.

Cette librairie ne fournit que trois procédures, mais ces dernières sont tout à fait suffisantes pour **gérer un moteur pas à pas** dans un programme pour Arduino. Elles permettent de configurer les paramètres relatifs au moteur utilisé, et d'imposer à ce dernier des mouvements aussi bien dans le sens direct que dans le sens rétrograde.

Initialisations des paramètres relatifs au moteur utilisé.

Stepper Mon_moteur(Resolution, S1, S2, S3, S4);

Cette procédure crée une instance de la classe Stepper qui identifie Mon_moteur à un moteur pas à pas connecté aux sorties binaires S1, S2, S3 et S4. Si seulement deux broches sont définies, c'est que le moteur est de type unipolaire. Cette procédure doit être rencontrée dans setup() ou dans loop() avant toute utilisation du moteur. Le paramètre Resolution définit le nombre de pas pour effectuer un tour sur le moteur. Il ne faut pas tenir compte de la présence d'un éventuel réducteur. La Fig.1 montre dans quel ordre doivent être réalisées les liaisons électriques si le moteur utilisé est un modèle de type bipolaire comme le 28BYJ-48 par exemple.

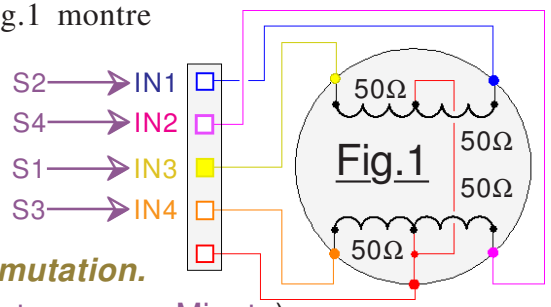


Fig.1

Vitesse possible de commutation.

Mon_moteur.setSpeed(Nb_tours_par_Minute);

Cette procédure précise la vitesse de rotation acceptable par le moteur sans risquer de perdre des pas. Elle ne fait pas mouvoir le moteur, mais définit simplement la vitesse à laquelle il tournera lors de l'appel à la procédure step(). Nb_tours_par_Minute est un long.

Faire tourner le moteur.

Mon_moteur.step(Nb_pas);

Active le moteur pour lui faire réaliser Nb_pas à la vitesse de rotation déterminée par l'appel le plus récent à setSpeed() ce qui permet au programme de pouvoir modifier à convenance la rapidité de la rotation. Cette fonction est "suspensive", c'est à dire qu'il faut attendre que le moteur ait fini de réaliser les Nb_pas avant de passer à l'instruction suivante dans le programme. Le paramètre Nb_pas est un nombre entier signé. Si le signe est positif le moteur tourne dans le sens direct. Si le signe est négatif la rotation sera rétrograde.

Méthodes de la bibliothèque AFMotor.h.

Cette librairie est prévue pour gérer des moteurs à courant continu en ***utilisant le Shield ADAFRUIT*** montré sur la Fig.1 et spécifiquement dédiée à l'interfaçage de puissance de petites motorisations. Après l'avoir déclarée, AFMotor.h permet de contrôler jusqu'à quatre moteurs à courant continu en sens et en vitesse sur le Shield de d'ADAFRUIT.

Gestion des moteurs à courant continu.

AF_DCMotor Mon_moteur(Port, Fréquence PWM);

Ce constructeur pour un moteur à courant continu doit être utilisé pour chaque moteur inclus dans le projet. *(Invoqué une seule fois par moteur)* Chaque instance doit avoir un nom différent. Le paramètre Port peut prendre les valeurs de 1 à 4 et précise le port de branchement du moteur. L'alimentation du moteur se fait en PWM, le paramètre Fréquence PWM définit la fréquence du signal rectangulaire généré. On peut utiliser les quatre options suivantes :

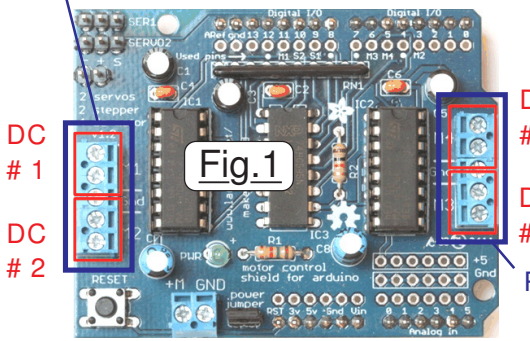
- MOTOR12_64kHZ
- MOTOR12_8kHZ
- MOTOR12_2kHZ
- MOTOR12_1kHZ

☞ 2kHZ n'est possible que sur #1 et #2 mais pas sur #3 et #4.

Si le paramètre Fréquence PWM n'est pas précisé, la fréquence de 1Khz sera utilisée par défaut pour le découpage du signal.
Cette instruction s'utilise après la déclaration de la bibliothèque, hors de la procédure d'initialisation void**setup**{}.

Exemple :

```
#include <AFMotor.h>
AF_DCMotor Mon_moteur_DC(1, MOTOR12_64KHZ);
// Création du moteur #1 avec PWM à 64KHz.
```



Remarque : Les fréquences les plus élevées produisent souvent un sifflement moins audible au fonctionnement, mais peuvent entraîner un couple inférieur avec certains moteurs.

Mon_moteur.setSpeed(Vitesse);

Cette procédure peut définir à tout moment la vitesse à laquelle tournera le moteur. La valeur du paramètre Vitesse sera comprise entre les entiers 0 à 255. *(0 : Plein ralenti / 254 : Vitesse maximale)*

Exemple :

```
Mon_moteur_1.setSpeed(254); // La vitesse du n°1 sera maximale.
Mon_moteur_3.setSpeed(127); // La vitesse du n°3 sera moyenne.
```

Remarque : La réponse d'un moteur à courant continu n'est pas forcément linéaire et dépend du couple résistant. La vitesse de rotation réelle ne sera pas nécessairement proportionnelle à la vitesse programmée.

Mon_moteur.run(Mode);

Cette procédure peut définir à tout moment le mode de fonctionnement du moteur. Les paramètres Mode valides sont :
FORWARD : Rotation dans le sens direct.
BACKWARD : Rotation dans le sens inverse.
RELEASE : Coupe l'alimentation du moteur. *(Équivalent à l'instruction SetSpeed (0).)* Le circuit d'ADAFRUIT ne gère pas un freinage dynamique, le moteur entrainé par inertie peut prendre un certain temps avant de s'immobiliser.

Exemple :

```
Mon_moteur.run(FORWARD); // Le moteur tourne dans un sens.
delay(1000); // Durant une seconde.
Mon_moteur.run(BACKWARD); // Le moteur tourne dans l'autre sens.
```

Gestion des moteurs PAS À PAS.

AF_Stepper Mon_pas_a_pas(Nb_Pas, Port);

Ce constructeur pour un moteur PAS À PAS doit être utilisé pour chaque moteur inclus dans le projet. *(Invoqué une seule fois par moteur)*. Chaque instance doit avoir un nom différent. Le paramètre Port peut prendre les seules valeurs 1 et 2 et précise le port de branchement du moteur. *(1 pour #1 et #2 et 2 pour #3 et #4 : Voir la Fig.1)* Le paramètre Nb_Pas précise le **nombre de pas par tour**.

Cette instruction s'utilise après la déclaration de la bibliothèque, hors de la procédure d'initialisation void**setup**{}.

Exemple :

```
#include <AFMotor.h>
AF_Stepper Moteur1(48, 1);   // 48 pas par tour / branché sur #1 & #2.
AF_Stepper Moteur2(200, 2);  // 200 pas par tour / branché sur #3 & #4.
```

…/…

Méthodes de la bibliothèque AFMotor.h. *(Suite)*

Mon_pas_a_pas.step(Nb_Pas, Sens, Mode);

Cette instruction commande la rotation du moteur pour Nb_Pas. Le sens de rotation est précisé par le paramètre Sens qui peut prendre les deux valeurs FORWARD et BACKWARD. Le paramètre Mode défini le type de commutation des bobines. Les options valides sont :

SINGLE : Une seule bobine est excitée à la fois.

DOUBLE : Deux bobines sont alimentées pour avoir plus de couple.

INTERLEAVE : Alterne entre simple et double pour générer un demi-pas "entre les deux". Généralement le fonctionnement est plus doux, mais le demi-pas réduit la vitesse et la rotation de moitié.

MICROSTEP : Les bobines adjacentes sont alimentées ou libérées par des rampes pour engendrer un certain nombre de "micro-étapes" entre deux pas complets. Il en résulte une résolution plus fine et une rotation plus lisse, mais le mouvement s'accompagne d'une perte de couple.

NOTE : L'instruction est "mobilisée" dans une boucle de programme tant que le moteur n'a pas effectué les Nb_Pas. Si on désire un mouvement simultané des deux moteurs possibles, il faut générer la chronologie par utilisation de l'instruction onestep() insérée dans une boucle qui traite les deux moteurs en gérant le nombre de pas effectués par chacun d'eux. *(Voir aussi l'encadré en bas de **p11**)*

Exemple :

```
Moteur1.step(100, FORWARD, DOUBLE);
// Faire 100 pas dans le sens direct avec couple augmenté.
Moteur2.step(300, BACKWARD, MICROSTEP);
// Faire 300 pas dans le sens rétrograde avec rotation lissée.
```

Mon_pas_a_pas.setSpeed(RPM);

Cette instruction qui peut être invoquée à tout moment et à convenance dans le programme définit la vitesse de rotation qui sera imposée au moteur pas à pas. La valeur indiquée est **exprimée en tours par minute**. Elle ne sera respectée que si :

- Le nombre de pas par tout du moteur a été correctement définie.
- Le couple résistant ne dépasse pas le couple moteur.

Exemple :

```
Moteur1.setSpeed(10); // Vitesse de 10 tours par minutes pour Moteur1.
Moteur2.setSpeed(20); // Vitesse de 20 tours par minutes pour Moteur2.
```

... / ...

Mon_pas_a_pas.release();

Cette instruction sans paramètre permet de supprimer le courant de maintien du moteur dans le but de réduire la température dissipée dans les bobines. Mais le moteur ne fournit aucun couple et la charge peut dans ce cas l'entrainer facilement.

Exemple :

```
Moteur1.release(); // Coupe l'alimentation des bobines du Moteur1.
```

Mon_pas_a_pas.onestep(Sens, Mode);

Cette instruction commande la rotation du moteur pour un seul pas. Le sens de rotation est précisé par le paramètre Sens qui peut prendre les deux valeurs FORWARD et BACKWARD. Le paramètre Mode défini le type de commutation et n'accepte pas ~~MICROSTEP~~. De plus **la vitesse n'est plus gérée** par la procédure setSpeed(RPM). Il faut imposer un petit délai entre chaque pas pour imposer la vitesse.

Exemple :

```
Moteur1.onestep(FORWARD, DOUBLE); delay(1);
// Un pas en mode DOUBLE effectué sur Moteur1.
```

Gestion des SERVOMOTEURS.

Le circuit imprimé de gestion des moteurs ne fait que déporter sur deux connecteurs dédiés aux servomoteurs la masse et le +5Vcc d'Arduino. Ces deux connecteurs ont trois broches. Deux pour l'alimentation du servomoteur et une pour son pilotage. Il faut donc utiliser la bibliothèque Servo.h décrite en page 6, usant en pilotage les sorties PWM 9 et 10.

La bibliothèque complémentaire AccelStepper.h.

Cette librairie qui inclut Stepper.h de la page 7 fournit une interface orientée objet pour gérer deux, trois ou quatre moteurs pas à pas. Elle améliore la bibliothèque de base de plusieurs façons :

- **Gère les accélérations et les ralentissements.**
- **Gère simultanément et indépendamment plusieurs moteurs.**
- Pilote des moteur de structures différentes.
- Prend en charge des pilotes de moteurs PAS À PAS comme le "Sparkfun EasyDriver" basé sur le circuit intégré spécialisé 3967.
- Les vitesses très lentes sont prises en charge.

Proposant des fonctions très élaborées, l'auteur de cette librairie donne un lien pour télécharger une documentation complète sur :

http://www.airspayce.com/mikem/arduino/AccelStepper/

Méthodes de la bibliothèque LiquidCrystal.h.

LiquidCrystal.h permet à une carte Arduino de contrôler un afficheur LCD piloté par un "chipset" de type Hitachi HD44780 ou un compatible qui équipe sur la plupart des écrans LCD alphanumérique. Elle autorise en outre aussi bien la commande avec 4 bits que par le mode 8 bits.

LiquidCrystal lcd(RS, RW, Enable, d0, d1, d2, d3, d4, d5, d6, d7);

Déclaration globale pour créer un objet lcd de type LiquidCrystal.
RW : Paramètre facultatif. Dans ce cas relier RW à la masse d'Arduino.
d0, d1, d2, d3 sont également facultatifs. Si omis pilotage quatre bits. Les diverses syntaxes possibles qui sont différenciées par le nombre de paramètres donnés à la fonction sont :

```
LiquidCrystal lcd(RS, Enable, d4, d5, d6, d7);
LiquidCrystal lcd(RS, RW, Enable, d4, d5, d6, d7);
LiquidCrystal lcd(RS, Enable, d0, d1, d2, d3, d4, d5, d6, d7);
LiquidCrystal lcd(RS, RW, Enable, d0, d1, d2, d3, d4, d5, d6, d7);
```

(Dans cet exemple lcd est l'identificateur de l'objet déclaré)

lcd.begin(Cols,Lignes); ***(Déclaration dans void setup())***

Précise les dimensions de la matrice de caractères de l'écran.
Doit être appelée avant les autres instructions de la bibliothèque.

lcd.setCursor(Col,Ligne);

Définit la position du curseur où commencera la prochaine écriture sur l'écran. *(0 est la première ligne ou la première colonne)*

lcd.cursor(); lcd.noCursor();

Valide ou supprime l'affichage du curseur de position d'écriture.

lcd.home();

Place le curseur de la prochaine écriture en haut et à gauche.

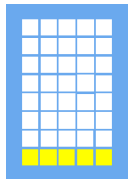
lcd.clear();

Efface tous les caractères de l'écran LCD.
Comme pour home positionne le curseur en haut et à gauche de l'écran du module d'affichage.

lcd.blink(); lcd.noBlink();

Valide ou supprime l'affichage du gros curseur clignotant qui remplit intégralement la matrice 5/8. Le curseur simple reste visible en permanence s'il est validé.

Curseur de Blink



lcd.noDisplay(); lcd.display();

noDisplay efface l'écran et le curseur mais conserve l'état en mémoire.
display rétablit l'affichage du texte et du curseur s'il est validé.

lcd.scrollDisplayLeft(); lcd.scrollDisplayRight();

Décale l'intégralité de l'affichage, y compris le curseur d'une position à gauche ou à droite. Les caractères "entrants sont des "espaces".

lcd.leftToRight(); lcd.rightToLeft();

Définit le sens de décalage du curseur de la gauche vers la droite *(Valeur par défaut)* ou de la droite vers la gauche lors d'un affichage.

lcd.autoscroll();

Chaque caractère sera écrit à la position actuelle du curseur, immédiatement suivi d'un décalage d'une position pour tout l'écran. Le décalage de l'écran est fonction du sens imposé par scrollDisplay Left ou Right. Le recyclage interne des caractères n'est pas spécialement évident. L'instruction lcd.noAutoscroll(); annule ce mode de fonctionnement et retourne au défilement classique.

lcd.print();

Cette fonction est analogue au print de la librairie Serial que l'on pourra consulter et présente un comportement similaire pour les divers formats reconnus. Elle permet d'afficher des textes et des nombres représentatif des valeurs de variables sur un module LCD.

```
lcd.print("Texte"); ou lcd.print('Caractère');
lcd.print(Data); Affiche la valeur numérique de la variable Data.
lcd.print(Data, BASE);
```

L'option BASE est facultative.

BIN : Binaire pur.
DEC : Décimal.
HEX : Hexadécimal.
OCT : Octal.

Data : char, byte, int, long, float ou string

lcd.write();

Cette fonction est orientée caractère. Elle fait afficher au module LCD le caractère **dont on fournit le code ASCII en décimal**. Si la valeur du paramètre est précédée de B le code est alors exprimé en binaire, les zéros en tête peuvent être omis. Exemples de codage :

... / ...

```
lcd.write(57);  (Affiche "9" car par défaut paramètre en décimal)
lcd.write(B111101);  (Affiche "=", paramètre en Binaire)
lcd.write(B01000110);  (Octet complet, affiche "F")
```

Il est tout à fait possible de passer comme paramètre à afficher un caractère issu de la ligne série USB :

```
void loop() { // Attendre l'arrivée d'un caractère sur le port série.
  if (Serial.available()) {
    lcd.clear(); delay(100);
   while (Serial.available() > 0)  // Lire tous les caractères reçus.
      {lcd.write(Serial.read); } } }
```

lcd.createChar(Numéro,Tableau);

Les modules LCD permettent de se créer jusqu'à huit caractères personnalisés désignés par Numéro dont la plage va de 1 à 8. L'apparence de chaque caractère se définit par un tableau de huit octets chacun relatif à l'une des lignes. Seuls les cinq bits de poids faibles décrivent les pixels de la matrice du caractère. Pour afficher un caractère personnalisé sur l'écran, utiliser l'instruction :

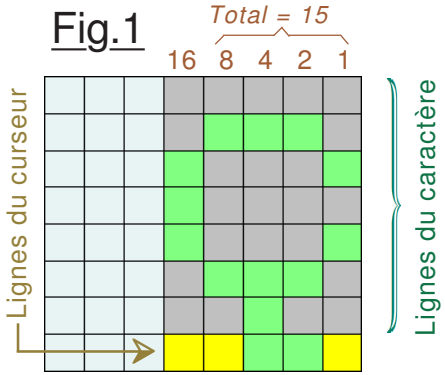
lcd.write(Numéro);

L'exemple de construction d'un caractère spécifique donné en Fig.1 est relatif à la génération du caractère "ç". On peut coder le tableau de définition en binaire ou en décimal. *(Et mélanger les deux)* En binaire il est inutile de donner les zéros en tête. Exemple pour le "ç" :

```
byte Ccedille[8] = {0,14,17,16,17,14,4,6};
byte Ccedille[8] = {0,B1110,B10001,B10000,B10001,14,4,6};
```

Le petit programme Caracteres_Speciaux_LCD.ino donne un exemple d'utilisation de génération de caractères personnalisés. Surtout il intègre les tableaux relatifs aux accentués qui n'existent pas dans la liste initiale, et donne une meilleure présentation aux caractères minuscules "g", "j", "p", "q" et "y" qui d'origine ne sont pas très beaux.

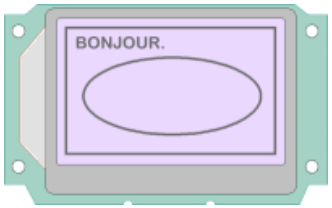
Fig.1



NOTE : "Sur Internet" Numéro est supposé varier entre 0 et 7 et non entre 1 et 8.

… / …

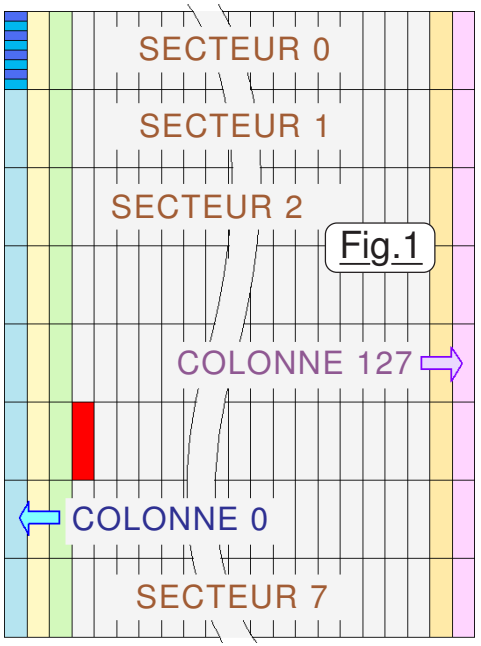
Méthodes de la bibliothèque Perso_ST7565.h.



Spécifique à l'afficheur LCD avec rétroéclairage trichrome ST7565, cette bibliothèque gère globalement des tracés géométriques et l'affichage de textes et de variables numériques.

Gestion par pavés verticaux de l'écran.

Géométriquement la matrice de points est constituée de 128 COLONNEs et de huit SECTEURs nommé également PAGEs dans certaines documentations. Pour modifier ce qui sera affiché à l'écran, il faut intervenir directement sur les BITs des OCTETs dont on doit indiquer les "coordonnées". Le traitement des coordonnées n'est pas immédiat, mais la bibliothèque s'en charge. Toutefois, comme la procédure Envoyer_une_DONNEE(Octet); dépose son paramètres aux coordonnées gérées par la bibliothèque, il importe de savoir comment la répartition géométrique logicielle est effectuée : Considérons la Fig.1 sur laquelle on constate que les COLONNEs sont numérotées de 0 à 127, et les SECTEURs entre 0 et 7. Le tout premier octet qui constitue le "pavé vertical" colorié en bleu se trouve en position [0,0]. Le rectangle rouge est contenu dans l'octet de coordonnées [3,5] et ainsi de suite. Le dernier octet en bas tout à droite se trouve en position [7,127]. Pour modifier un octet dans la mémoire d'écran, il faut au préalable positionner le curseur aux coordonnées désirées dans "la surface de l'écran" avec la procédure Positionne_curseur (Colonne, Secteur). Notez qu'il existe des afficheurs à logique d'illumination positive, et des composants de type négatif. "Noir" et "Allumé" seront fonction du type utilisé.



… / …

Créer une instance de l'objet Perso_ST7565.h .

Le programme doit commencer par la directive :

#include <PERSO_ST7565.h>

Si l'on désire pouvoir traiter librement le rétro-éclairage il faut définir les broches de pilotage par des identificateurs. Par exemple :

```
#define Retro_Bleu 9
#define Retro_Rouge 10
#define Retro_Vert 11
```

NOTE : Ici 9,10 et 11 sont choisies pour pouvoir les traiter en PWM.

Ensuite on doit créer une instance *(Nommée ici LCD.)* de l'objet :

```
PERSO_ST7565 LCD = PERSO_ST7565
                   (6,7,8,Retro_Bleu,Retro_Rouge,Retro_Vert,12);
```

Les paramètres dans l'ordre désignent la broche qui pilotera RST, A0, SCLK, Le Bleu, Le Rouge, le Vert et SID. Dans cet exemple A0 est pilotée par la sortie binaire 6, SCLK par 7 ... SID par 12. *(Dans cet exemple l'utilisateur regroupe toutes les broches de commande en laissant libre la LED Arduino 13 et en permettant de commander les trois sorties du rétroéclairage en analogique PWM.)*

À partir d'ici l'objet instancié est complètement initialisé, les broches sont configurées, l'écran est mis en service et effacé et l'on peut à convenance utiliser les diverses méthodes proposées par la bibliothèque. Par exemple LCD.ELLIPSE(0,0,60,30,1);

Méthodes de base.

LCD.Envoyer_une_COMMANDE(Octet);

Procédure qui envoie la commande précisée par Octet :
B11001000 : Balayage vertical normal du Haut vers Bas.
B11000000 : Balayage vertical inversé du BAS vers le Haut.
B10100000 : Balayage horizontal normal de Gauche à Droite.
B10100001 : Balayage horizontal inversé de Droite à Gauche.
B10100101 : Allume tous les points.
B10100100 : Retour au mode normal.
B11001000 : Passer en mode de contraste dynamique. Doit être suivi de la valeur entre 32 et 41 en décimal. *(0 à 63 dans la doc.)*
Exemple : LCD.Envoyer_une_COMMANDE(41);
B10100110 : Ecran NORMAL. *(Écriture allumée, fond noir.)*
B10100111 : Inverse l'état de l'intégralité de l'écran.

…/…

LCD.Afficheur_en_veille();
Procédure quiPasse l'afficheur en veille et éteint les trois couleurs. *(L'afficheur passe en consommation minimale.)*

LCD.Afficheur_en_service;
Réactive l'afficheur et allume les trois couleurs.

LCD.Envoyer_une_DONNEE(Octet);
Procédure qui Affiche le pavé vertical Octet aux coordonnées [Colonne, Secteur] actuelles. *(Voir Fig.1 en page 23)* Comme l'index horizontal interne de l'afficheur est automatiquement décalé d'une position à droite, Colonne est incrémentée pour pointer le prochain "pavé vertical" sur la mosaïque de l'écran.

LCD.Impose_PAGE(Secteur); *(0 < Secteur < 8)*
Positionne verticalement la prochaine écriture sur Secteur.

LCD.Impose_COLONNE(Colonne); *(0 < Secteur < 128)*
Positionne horizontalement la prochaine écriture sur Colonne.

LCD.Positionne_curseur(Colonne, Secteur);
Positionne l'emplacement d'écriture du prochain **"pavé vertical"** aux coordonnées Colonne, Secteur dans la mosaïque de l'écran. Les limites sont précisées ci-dessus ou sur la Fig.1 en *page 23*.

LCD.Contraste(Contraste);
Procédure qui impose la valeur du paramètre byte Contraste en contraste dynamique. La documentation en ligne précise des valeurs comprises entre et 0x63. *(Je conseille entre 27 pour le Noir et 50 Maxi pour le Blanc, valeurs données en décimal.)*

NOTE : L'afficheur ST7565 ne dispose pas d'une mémoire interne accessible par l'utilisateur. Pour palier cette faiblesse un tableau [MEMOIRE] est créé dans l'instance de l'objet LCD. En utilisation du programme, certaines méthodes agissent directement sur l'écran et mettent à jour [MEMOIRE]. D'autres permettent de modifier directement le contenu de [MEMOIRE] sans en visualiser le changement sur l'écran. Dans la description de la bibliothèque, faire référence à l'Écran correspond à l'affichage, alors que [MEMOIRE] correspond à la modification du tableau sans incidence sur l'écran sauf si c'est précisé.

LCD.Effacer_Ecran();
Efface entièrement l'écran sans modifier [Colonne / Secteur].

…/…

LCD.Blancking_Ecran();
Blanchi *(Ou noirci en fonction du type d'afficheur.)* entièrement l'écran sans modifier [Colonne / Secteur].

LCD.Inverser_ecran();
Inverse l'état des PIXELs de l'écran et conserve[Colonne / Secteur].

LCD.Remplir_et_affiche_Ecran(Octet); *(**Octet** : Huit pixels.)*
Remplit l'écran de pavés sans modifier [Colonne / Secteur].

LCD.Afficher_MEMOIRE();
Rafraichi la mosaïque de l'écran avec la totalité du contenu de [MEMOIRE] sans modifier [Colonne / Secteur].

LCD.Decale_MEMOIRE();
Décale la [MEMOIRE] d'un Secteur vers le haut sans afficher le résultat. *(Sera surtout utile pour du texte.)*

LCD.Decale_ECRAN();
Décale la mosaïque de l'écran d'un Secteur vers le haut sans modifier [Colonne / Secteur]. *(Sera surtout utile pour du texte.)*

Les Méthodes GRAPHIQUES.

NOTE : La notion de "Allumer" ou "Éteindre" sera directement fonction du type d'afficheur ST7565 utilisé. Pour rendre compatible les effets obtenus par rapport aux informations de ce document, il suffit dans void **setup** de faire appel à la procédure LCD.Inverser_ecran(); si le modèle utilisé dans l'application présente une logique inversées. Pour toutes les méthodes le paramètre Allume précise si l'élément sera tracé en "Blanc" si la valeur passée à la méthode est true ou 1, ou en "Noir" si elle vaut false ou 0.

LCD.PIXEL(int X, int Y, boolean Allume);
Procédure qui modifie un PIXEL individuellement. Les coordonnées du PIXEL X de 0 à 127 et Y de 0 à 63 peuvent être quelconques, *(Car utilisées pour tracer les autres entités géométriques.)* mais il n'y aura pas d'action si elles sont hors de la mosaïque d'écran.

LCD.LIGNE(int x, int y, int X, int Y,boolean Allume);
Trace la ligne dont on spécifie les coordonnées du début avec x et y et de la fin avec X et Y. L'ordre début et fin peut être quelconque, et les extrémités peuvent être hors de la mosaïque d'écran. Seule la partie "visible" sera tracée les débordements d'écran étant entièrement gérés. Les coordonnées peuvent être négatives sans problème.

… / …

LCD.CADRE(int x, int y, int X, int Y,boolean Allume);
Trace un cadre *(Rectangle creux.)* dont on définit dans un ordre quelconque les extrémités de l'une de ses deux diagonales. Traite entièrement l'ordre dans lequel on donne ces "angles" ainsi que les débordements d'écran. *(Début avec x et y et Fin avec X et Y.)*

LCD.RECTANGLE_plein(int x, int y, int X, int Y,boolean Allume);
Trace un rectangle "plein" dont on définit dans un ordre quelconque les extrémités de l'une de ses deux diagonales. Traite entièrement l'ordre dans lequel on donne ces "coins" ainsi que les débordements d'écran. *(Début avec x et y et Fin avec X et Y.)*

LCD.CERCLE(int X, int Y, int R, boolean Allume);
Trace un cercle dont on indique les coordonnées du centre avec X et Y et la valeur du rayon avec R. Seule la partie "visible" sera tracée les débordements d'écran étant entièrement gérés. Les coordonnées du centre peuvent être négatives sans problème.

LCD.ELLIPSE(int X, int Y, int RX, int RY, boolean Allume);
Trace une ellipse dont on indique les coordonnées du centre avec X et Y, le "rayon" horizontal avec RX et le "rayon" vertical avec RY. Seule la partie "visible" sera tracée les débordements d'écran étant entièrement gérés. Les coordonnées peuvent être négatives sans problème.

LCD.ARC(int X, int Y, int R, int Angle_debut, int Angle_fin, boolean Allume); Trace un arc de cercle dont on indique les coordonnées du centre avec X et Y et la valeur du rayon avec R. Les Angle_debut et Angle_fin sont indiqués en partant du "Nord" et doivent être POSITIFS. Seule la partie "visible" sera tracée les débordements d'écran étant entièrement gérés. Les coordonnées du centre peuvent être négatives. **Le tracé est effectué dans le sens horaire** tous les degrés donc, **plus le rayon imposé devient grand, plus les PIXELs représentatifs seront séparés.**

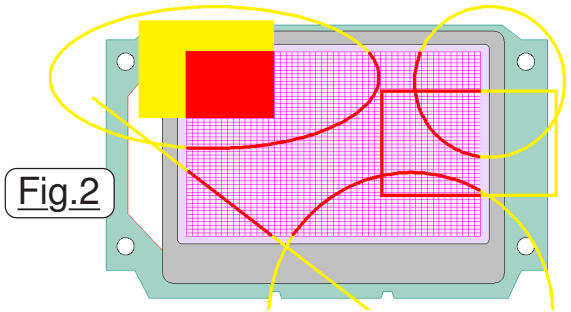


Fig.2

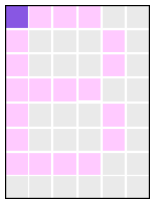
Sur la Fig.2 ci-contre les éléments hors de la mosaïque d'écran qui ne seront pas "tracés" sont représentés en jaune, alors que la partie en rouge est celle qui modifiera les PIXELs.

… / …

Police de caractères en mémoire de programme FLASH.

Actuellement la table de génération des caractères construite pour des matrices 6x8 est logée en mémoire de programme pour libérer totalement l'EEPROM et ne pas consommer de place en RAM dynamique. De plus la bibliothèque devient entièrement "autonome". Si l'on désire modifier des caractères spéciaux, il faut modifier leurs cinq octets génériques dans PERSO_ST7565.ccp avec un quelconque éditeur de texte.

Les Méthodes TEXTUELLES.



LCD.CURSEUR(byte X, byte Y);

Impose les coordonnées logicielles de la prochaine écriture d'un caractère sur l'écran. X et Y indiquent la position du PIXEL situé en haut à gauche de la matrice qui définit le caractère. Contrairement à la méthode LCD.Positionne_curseur(Colonne, Secteur); dédiée au tracé de "pavés verticaux", LCD.CURSEUR() qui gère des "matrices 6 x 8" peut placer le caractère à des coordonnées quelconques sur l'écran. La partie hors écran du caractère n'est pas affichée.

LCD.CRLF();

Fait passer le **curseur textuel** en début de ligne suivante. Si le bas de l'écran est dépassé provoque un "schrolling" écran.

LCD.HOME();

Fait passer le **curseur textuel** en haut à gauche de l'écran.

LCD.Decale_MEMOIRE(); *(Voir Page 26.)*

LCD.Decale_ECRAN(); *(Voir Page 26.)*

LCD.Affiche_TEXTE("Chaîne de caractères");

Affiche la chaîne de caractères à partir de la position actuelle du **curseur textuel**. Gère un semi passage à la ligne si débordement. Un débordement à droite fait passer à la ligne, mais coupe éventuellement le caractère en cours. Si le débordement se fait sur la ligne du bas il y a génération d'un "schrolling" écran.

LCD.Affiche_NOMBRE(Valeur numérique, Nb_decimales);

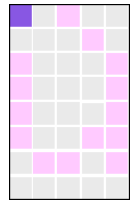
Affichera une valeur numérique quel que soit son type. Le nombre de décimales souhaité est précisé dans Nb_decimales. La valeur sera comprise entre **0** et **2**. Si on propose une valeur supérieure à 2 elle sera ramenée à **2**. Si la consigne est **0**, il n'y aura aucune décimale et le point décimal n'est alors pas affiché.

…/…

LCD.Surbrillance(boolean)

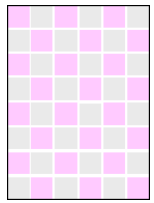
Si "true" écrit et affiche les textes et les nombres en surbrillance.

Modifier des caractères spéciaux.



Fondamentalement il suffit de remplacer les cinq octets qui définissent l'un des caractères spéciaux qui ne sera plus utilisé. L'appel se fait toujours avec le caractère textuel de son code ASCII. Par exemple on désire que 'Flèche Gauche' de code ASCII 127 non utile pour le programme en cours de développement après réécriture de ses cinq octets devienne 'ù'. Le premier octet est relatif à la colonne verticale de gauche. **Attention, le BIT de poids faible correspond au haut de la matrice, et le poids fort au bas.** L'inter caractères sur la 6ième colonne de droite est généré automatiquement par la méthode concernée avec un octet égal à B00000000, raison pour laquelle chaque caractère n'est représenté que par cinq OCTETs.
Pour notre exemple la dernière ligne du tableau devient :
B00111100, B01000000, B01000001, B00100010, B01111100};
La table de génération des matrices de caractères se trouve dans le fichier Perso_ST7565.ccp de la bibliothèque.

Génération d'une matrice de 6 colonnes.



On peut créer un "pavé" complet sans modifier la bibliothèque. La technique consiste à créer une procédure spécifique dans laquelle la matrice 6 x 8 est définie dans un petit tableau local à la procédure.
Chaque nouveau caractère coute 110 octets de programme et 6 octets en mémoire dynamique. Quand on veut afficher le caractère spécial, dans le programme on fait appel à cette procédure. Exemple :

```
Appel dans le programme : Damier();
void Damier() {// Laisse PTR sur le prochain Octet de
   MEMOIRE après de caractère.
   byte CAR_Special1[6] {B01010101, B10101010,
        B01010101,  10101010, B01010101, B10101010, };
   byte BarreV;
   for(byte Colne=0; Colne<6; Colne++)
      {BarreV = CAR_Special1[Colne];
       LCD.Envoyer_une_DONNEE(BarreV);} }
```

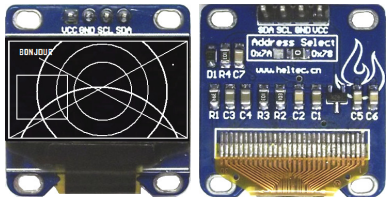
Méthodes de la bibliothèque Adafruit_ssd1306syp.h.

Spécifique à l'afficheur graphique monochrome miniature OLED 128 x 64, cette bibliothèque est suffisante pour mettre facilement en œuvre ce produit aussi bien pour du tracé géométrique que pour de l'affichage de textes.

Initialisations diverses.

Adafruit_ssd1306syp display(broche_SDA, broche_SCL);
Cette directive doit être placée avant void **setup**() et précise les deux sorties binaires d'Arduino qui seront utilisées pour interfacer l'I2C. Il n'est pas utile de configurer ces deux broches en sorties.

display.initialize(); delay(100);
Instruction à placer dans void **setup**() avant toute utilisation de l'afficheur. Cette instruction se charge de définir les deux broches affectées à la ligne I2C en sorties. Les broches A0 à A5 sont parfaitement utilisables en sorties. *(Déclarations 14 à 19.)*

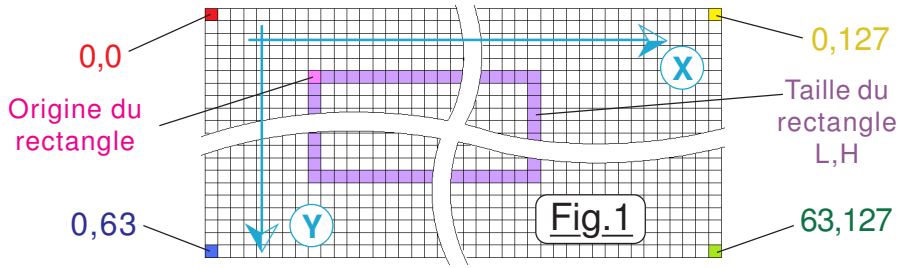


Principe des d'affichages.

Qu'il soit graphique ou texte, un affichage se fait toujours en deux phases. La première action consiste à placer les "pixels" dans la mémoire tampon 128 x 64 de l'afficheur. La deuxième action consiste à transférer l'affichage sur l'écran LCD avec l'instruction display.update(); cette ligne étant indispensable même pour l'instruction display.clear(); d'effacement de l'écran.
La Fig.1 précise les valeurs des coordonnées pour les pixels.

Affichages graphiques.

display.drawPixel(X, Y, WHITE);
Procédure qui permet de gérer l'afficheur point par point. Elle allume le pixel de coordonnées X et Y si le paramètre de lumière est WHITE ou éteint le point si le paramètre est BLACK.



display.drawLine(Xd, Yd, Xf, Yf,WHITE);
Procédure qui trace une ligne entre les deux points dont on précise les coordonnées. WHITE ou BLACK précise si on trace en blanc ou en noir. L'ordre de définition des extrémités est quelconque.

display.drawRect(Xo, Yo, Largeur, Hauteur,WHITE); *(Voir Fig.1)*
Procédure qui trace un rectangle dont on précise les **coordonnées de l'Origine**, la Largeur et la Hauteur. WHITE ou BLACK précise si on trace en blanc ou en noir. Largeur et hauteur peuvent avoir des valeurs négatives et sont alors tracés vers la gauche et vers le haut.

display.drawCircle(Xo, Yo, Rayon, WHITE);
Procédure qui trace un cercle dont on précise les **coordonnées du centre** et la valeur du Rayon. WHITE ou BLACK précise si on trace en blanc ou en noir. Gère les débordements de 127 x 63.

Affichages des textes.

display.setTextColor(WHITE);
Procédure qui définit la "couleur" du texte. (Noir si BLACK.)

display.setTextColor(TEXTE,FOND);
Procédure qui définit la couleur du texte puis du fond. Par exemple display.setTextColor(BLACK, WHITE); fait passer le texte en "surbrillance" par inversion écriture en noir sur fond blanc.

display.setCursor(X, Y);
Positionne le curseur de la prochaine écriture textuelle.

display.setTextSize(Taille);
Définit la taille des caractères, la plus petite valeur étant **1.**
1 : Matrice 5x7, **2** : Le double, **3** : Le triple etc.

display.print(Donnée); display.println(Donnée);
Affiche la donnée à partir de la position actuelle du curseur. Gère les débordements de la fenêtre 127 x 63, le texte dépassant du domaine affichable sera ignoré. Le format de la Donnée peut être :
display.println("Texte quelconque."); *(1)*
display.print(PI,8); Valeur et en option le nombre de décimales.
display.println(1234, BIN); Valeur affichée en Binaire.
display.print(1234, HEX); Valeur affichée en Hexadécimal.
display.println(1234, OCT); Valeur affichée en Octal.

(1) : Les lettres, les chiffres et la ponctuation standard sont correctement affichés ainsi que 128 autres caractères particuliers. (Matrices standard ASCII au format 5x7 pour le coefficient 1.)

Méthodes de la bibliothèque avr/eeprom.h.

Cette bibliothèque remplace avantageusement EEPROM.h décrite en bas de la page 5. Il n'y a pas besoin de la déclarer avec un ***include*** ce qui résous pas mal de problème dans la rédaction de bibliothèques personnelles. Le code généré est plus compact et fonctionne plus rapidement. On peut lire et écrire des entités de tailles différentes.

Utilisation de la bibliothèque :

Il suffit dans le programme utilisateur d'ajouter les procédures et les fonctions dont on a besoin puis d'y faire appel. Dans les exemples qui suivent les identificateurs coloriés en rose sont ceux de la bibliothèque avr/eeprom.h et il ne faut pas les modifier.

Fonction Lire un octet

```
byte  Lire_Octet_EEPROM(int  ADRESSE)
  {return  eeprom_read_byte((unsigned  char *)  ADRESSE);}
```

Exemple d'appel : OCTET = Lire_Octet_EEPROM(PTR_EPROM);

Procédure Écrire un octet

```
void Ecrire_Octet_EEPROM(int ADRESSE, byte OCTET)
  {eeprom_write_byte((unsigned  char *)  ADRESSE, OCTET);}
```

Exemple d'appel : Ecrire_Octet_EEPROM(PTR,Donnée);

Fonction Lire un entier

```
int  Lire_Entier_EEPROM(int  ADRESSE)
  {return  eeprom_read_word((unsigned  int *)  ADRESSE);}
```

Exemple d'appel : Entier = Lire_Entier_EEPROM(PTR_EPROM);

Procédure Écrire un entier

```
void  Ecrire_Entier_EEPROM(int ADRESSE, int OCTET)
  {eeprom_write_word((unsigned  int *)  ADRESSE,  OCTET);}
```

Exemple d'appel : Ecrire_Entier_EEPROM(PTR,Donnée);

Fonction Lire un long

```
int  Lire_un_Long_en_EEPROM(int  ADRESSE)
  {return  eeprom_read_dword((unsigned  long *)  ADRESSE);}
```

Exemple: Quatre_Octets = Lire_un_Long_en_EEPROM(PTR);

Procédure Écrire un long

```
void Ecrire_un_Long_en_EEPROM(int ADRESSE, long QtrOct)
  {eeprom_write_dword((unsigned long *) ADRESSE, QtrOct);}
```

Exemple d'appel : Ecrire_un_Long_en_EEPROM(PTR,Donnée);

Fonction Lire un float

```
int Lire_un_Float_en_EEPROM(int ADRESSE)
   {return eeprom_read_float((float *) ADRESSE);}
```

Exemple: REEL = Lire_un_Float_en_EEPROM(PTR_EEPROM);

Procédure Écrire un float

```
void Ecrire_un_Float_en_EEPROM(int ADRESSE, Float REEL)
   {eeprom_write_float((float *) ADRESSE, REEL);}
```

Exemple d'appel : Ecrire_un_Float_en_EEPROM(PTR,REEL);

Écrire ou lire un BIT individuel en EEPROM :

Ces méthodes ne sont pas prévues dans les routines proposées en standard par la bibliothèque avr/eeprom.h mais elles ne sont pas spécialement compliquées à coder :

Fonction Lire un BIT individuel

```
boolean Lire_un_BIT_en_EEPROM(int ADR, byte N) {
  return ((eeprom_read_byte((unsigned char *) ADR)) >> N) & 1;}
```

L'ordre du BIT N doit être compris entre [0 et 7].

Exemple d'utilisation :

```
boolean BIT; BIT = 3; PTR = 1020;
BIT = Lire_un_BIT_en_EEPROM(PTR, BIT );
```

Procédure Écrire un BIT individuel

```
void Ecrire_un_BIT_en_EEPROM(int ADR, byte N, boolean Etat)
 { byte Octet, Masque;
   Octet = eeprom_read_byte((unsigned char *) ADR);
   Masque = 1 << N;
   if (Etat) Octet = Octet | Masque;
   else {Masque=~Masque; Octet = Octet & Masque;}
   eeprom_write_byte((unsigned char *) ADR, Octet);}
```

L'ordre du BIT N doit être compris entre [0 et 7].
L'état du BIT peut être true, false, "1" ou "0".

Exemple d'utilisation :

```
Ecrire_un_BIT_en_EEPROM(844, 7, 0);
// L'emplacement relatif n°844 verra son BIT n°7 forcé à zéro.
```

http://www.nongnu.org/avr-libc/user-manual/group__avr__eeprom.html#ga2d4ee8b92a592c764785fb5e4af5662b

Méthodes de la bibliothèque avr/pgmspace.h.

Cette librairie est prévue pour loger en mémoire de programme des constantes dans le but de libérer de la place dans la SRAM. La mémoire Flash dite "de programme" est dédiée au stockage du code objet compilé envoyé au microcontrôleur au moment du "téléversement". Comme toutes les mémoires Flash, sa durée de vie se situe à environ 100000 écritures et effacements. L'IDE en fait donc un usage particulier : On ne peut y enregistrer des données qu'au moment du "téléversement" du programme. Elle fonctionne ensuite en lecture seule, **donc on ne peut que lire des données et plus les modifier**.

Utilisation de la mémoire FLASH pour loger des données.

Au même titre que l'EEPROM, la mémoire FLASH prévue pour loger le programme peut également être utilisée pour y loger des données. D'un accès aussi rapide que celui de la RAM, elle conserve les données sur coupure d'alimentation. La mémoire FLASH doit être considérée comme une mémoire accessible uniquement **en lecture de constantes**. Par précaution le programme doit commencer par la directive classique #include <avr/pgmspace.h> sachant qu'avec les versions actuelle de l'**IDE** ce n'est pas un impératif.

```
PROGMEM const type NomConstante (Valeur);
```

PROGMEM est un modificateur de variable, et ne peut s'utiliser **qu'avec les types restrictifs de données définis dans la librairie** avr/pgmspace.h qui impose au compilateur de placer les données dans la mémoire FLASH", au lieu de la SRAM, où elles devraient résider en standard. *(Voir types de données page suivante.)*

La directive PROGMEM étant un modificateur de variable, donc en principe il n'y a pas d'emplacement obligatoire, et le compilateur d'Arduino devrait accepter toutes les définitions équivalentes. L'expérience montre que dans certaines versions anciennes il y a disfonctionnement si le mot clef PROGMEM est positionné après le type et avant le nom de la variable :

```
Type NomConstante[] PROGMEM = {};   // Accepté.
PROGMEM  Type  NomConstante[] = {}; // Accepté.
Type PROGMEM NomConstante[] = {};   // Ne pas utiliser.
```


Les types de données utilisables.

L'utilisation de certains types de données peuvent conduire à des erreurs énigmatiques. *(Les float ne sont pas correctement traités.)* Il importe donc de n'utiliser que les types de données déclarés dans la bibliothèque avr/pgmspace.h soit char, int et long. *(signed ou unsigned)*

Quelques exemples de déclaration :

```
PROGMEM const char CaractereA ('A');
const unsigned char Car_F PROGMEM ('f');
const byte Vingt PROGMEM = 20;
PROGMEM const int Moins32000 (-32000);
PROGMEM const unsigned int Plus65000 (+65000);
PROGMEM const long Moins12345678 = -12345678;
PROGMEM const unsigned long Plus4Giga = +4000000000;
```

Dans ces exemples les deux positions de PROGMEM dans la déclaration sont utilisées. Pour affecter une valeur à la constante, on peut utiliser deux écritures équivalentes : Soit on fait suivre la déclaration par le "=" de façon classique, soit on indique la constante entre parenthèses. L'indication const avant le type est obligatoire. Noter que déclarer un caractère ASCII en unsigned char il devient un nombre, une instruction comme Serial.print affichera son code.

Utilisation des données placées en mémoire programme.

Intimement liée à la notion de pointeur et de passage de paramètre par adresse, l'utilisation des constantes logées en mémoire FLASH en reprend plus ou moins directement la syntaxe d'écriture. Dans les exemples qui suivent, les méthodes spécifiques à avr/pgmspace.h sont coloriées en rose et sont associées au type déclaré pour la variable :

```
Serial.println(CaractereA);
Serial.println(Car_F); // Affiche 102 le code ASCII de 'f'.
Serial.println(Vingt);
int ENTIER;
ENTIER = pgm_read_word(&Moins32000);
unsigned int EntierPositif;
EntierPositif = pgm_read_word(&Plus65000);
long DoubleOctet;
DoubleOctet = pgm_read_dword(&Moins12345678);
unsigned long DoubleOctetPositif;
DoubleOctetPositif = pgm_read_dword(&Plus4Giga);
```

... / ...

Traitement des chaînes de caractères.

Impose pour la déclaration des chaînes la syntaxe avec le "=" :

```
PROGMEM const char Bonjour[] = "Bonjour ";
const char Les_Amis[] PROGMEM = "les Amis.";
```

L'utilisation des constantes chaîne oblige à écrire une procédure spécifique avec pour paramètre un pointeur :

```
void Affiche_TEXTES_en_FLASH(char *PTR) {
  char I=0; char CarASCII=1; // Dif de 0 pour entrer en boucle.
  while(CarASCII != '\0') { // Tant que non fin de chaîne :
        CarASCII = pgm_read_byte(PTR + I);
        if (CarASCII != 0) Serial.print(CarASCII); I++; } }
```

Traitement des tables d'OCTETs.

Bien que PROGMEM fonctionne avec des variables élémentaires, il est plus judicieux de s'en servir avec des blocs de données à stocker, comme des tableaux ou d'autres structures de données en langage C. L'exemple typique est celui d'un dessin de type LOGO, d'une table pour générer des caractères en matrice 5 x 7 etc.

Exemple de déclaration pour des OCTETs :

```
PROGMEM const byte TABLE_ASCII[] =
 {B00000000, B00000000, B00000000, B00000000, B00000000, // ESPACE
  B00000000, B00000000, B01001111, B00000000, B00000000, // !
  B00000000, B00000011, B00000000, B00000011, B00000000, // "
   ...    ...    ...    ...    ...    ...    ...    ...    ...    ...    ...    ...
  B00011100, B10100000, B10100000, B10100000, B01111100, // y
  B01000100, B01100100, B01010100, B01001100, B01000100, // z
   ...    ...    ...    ...    ...    ...    ...    ...    ...    ...    ...    ...
  B00000000, B01000001, B00110110, B00001000, B00000000, // }
  B00001000, B00001000, B00101010, B00011100, B00001000, // Flèche D
  B00001000, B00011100, B00101010, B00001000, B00001000 }; // Flèche G
```

Exemple de lecture des OCTETs :

Impose d'écrire une procédure pour "balayer" la matrice de caractère.

```
void Affiche_Caractere(byte ASCII) {
  int PTR = ASCII * 5;
  for(byte Colonne=0; Colonne<5; Colonne++)
    {  Aff_Colonne (pgm_read_byte(&TABLE_ASCII[PTR]));
       PTR++;} Aff_Colonne(0); }
```

…/…

CRÉER UNE BIBLIOTHÈQUE PERSONNELLE :

Préservés dans un dossier commun, les bibliothèques Arduino sont composées d'au moins deux fichiers : Un fichier d'en tête *(Le header.)* d'attribut .h et un fichier de code source ayant pour attribut .cpp les deux ayant le même nom identique à celui du dossier qui héberge les modules. *(.c en langage C standard et .cpp en version C++.)* Le fichier d'en tête contient la liste des fonctions disponibles et le fichier source contient l'implémentation du code. À la compilation la totalité du code sera analysée.

Le fichier d'entête.h *(h comme Head)*

Fondamental, il sert à déclarer l'intégralité des modules qui seront impératif à son bon fonctionnement : Les procédures et les fonctions, les variables sans oublier la gestion des #include nécessaires à cette bibliothèque. Pour que cette dernière puisse utiliser des fonctions propres à Arduino il faut une déclaration #include <Arduino.h> *(Version > 1.00)* qui fait partie du "core Arduino" et n'a pas à être téléchargée sur Internet. On peut également inclure des modules qui sont spécifiques à la bibliothèque et situés dans son dossier. Dans ce cas les délimiteurs sont des guillements. Par exemple #include "Ma_police_de_caracteres". Le fichier d'entête comprendra au minimum :

```
①  #ifndef PERSO_ST7565_h  // Si non déjà définie. (1)
②  #define PERSO_ST7565_h // On la défini.
③  #include <Arduino.h> // probablement indispensable.
④  class PERSO_ST7565 {
⑤    public :
⑥    void Envoyer_une_COMMANDE(byte OCTET);
⑦   };
⑧  #endif
```

(1) Attention : "_" et non le "." pour désigner l'identificateur.

Avant tout contenu, il faut en ① et ② éviter que le fichier d'en tête ne soit victime ***d'inclusions multiples***. Donc on ajoute une protection avec des directives de préprocesseur. Il est probable que l'on utilisera des procédures de l'IDE ce qui impose la déclaration ③. Comme pour une variable quelconque, il faut déclarer un objet : En ④ on définit la classe avec le **même nom que celui de la bibliothèque**. En publicon va déclarer en ⑥ au moins une méthode.

…/…

Toutes les déclarations relatives à la classe concernée seront regroupées entre les délimiteurs **{ }** suivis en ⑦ d'un point-virgule. Enfin en ⑧ le fichier d'entête termine ① par la directive #endif.
Le corps des directives peut comporter d'autres déclarations comme les constantes et les variables utilisées dans la bibliothèque.
Dans private :

```
boolean Surbrillance;
byte OCTET, BIT, PAGE;
byte MEMOIRE[1024]; // Ici on déclare un grand tableau.
```

Variables déclarées en général en private pour les protéger des "contaminations" externes.

Entrées/Sorties définies par l'utilisateur.

Laisser à l'utilisateur la possibilité de répartir à sa guise les broches de l'ATmega328 qui seront utilisées pour piloter son afficheur est un incontournable. Dans ce but, on doit déclarer dans le fichier d'entête une instance logicielle de l'objet afficheur :
Dans public :

```
PERSO_ST7565(byte broche_RST, ... byte broche_SID);
```

Une telle déclaration commence par le nom de l'objet qui est identique à celui de la bibliothèque, suivi entre parenthèses de la liste des identificateurs pour les broches qui seront utilisées. Comme il s'agit d'un "passage de paramètre", les valeurs sont typées.
NOTE : Plus généralement, l'utilisateur de la bibliothèque passera en paramètres diverses constantes, et pas forcément le numéro des broches utilisées par la classe décrite dans le constructeur.

Protéger ou accéder à une variable dans une classe.

D'une façon générale il est fortement recommandé de déclarer en **private :** les variables d'une classe pour les protéger. La visibilité de ces dernières sera alors limité au "domaine" de la classe instanciée. Le mot clef **public :** précise que les éléments dont la liste fait suite sont accessibles par toutes les fonctions et procédures y compris depuis le ***programme.ino*** qui inclus la classe. Naturellement les méthodes "utilisateur" seront publiques.
On peut si on désire pouvoir les modifier déclarer des entités dans le domaine **public :** leur utilisation imposant de faire référence à la classe pour leur affecter ou y lire des valeurs. Exemple :
Dans le fichier.h on écrit : public : byte TEST;
Et pour utiliser dans le croquis.ino : LCD.TEST = TEST + 25;

…/…

Le fichier du code source.ccp

Commençant comme le fichier d'entête par des directives pour le préprocesseur, il contiendra l'ensemble du code d'implémentation qui conduit aux méthodes exécutables et commence normalement par #include "PERSO_ST7565.h" et en hérite des directives #include qui s'y trouvent. Par exemple du #include <Arduino.h> qu'il serait néfaste de réitérer provoquant des définitions multiples. Ensuite, on peut éventuellement trouver des directives #define qui permettent de définir des constantes.

```
#define Allume_Tout B10100101  // Allume tous les points.
```

On doit obligatoirement trouver la déclaration de la classe avant l'écriture des méthodes. L'architecture ressemble à :

```
PERSO_ST7565::PERSO_ST7565(Les paramètres) {
  ...
Déclarations et instructions
  ... }
```

La première occurrence de PERSO_ST7565 définit un **type classe** et *Les paramètres* sont strictement identiques en type, ordre et identificateurs à ceux de la déclaration de la classe dans public :

```
PERSO_ST7565(byte broche_RST, ... byte broche_SID);
```

du fichier d'entête. Ensuite, entre les délimiteurs **{** et **}** on trouvera tout ce que le compilateur devra faire quand dans le programme utilisateur il y aura appel au constructeur de PERSO_ST7565 :
Généralement, quand on crée la classe, on doit configurer les Entrées/Sorties, les initialiser, envoyer des consignes au périphérique piloté pour une initialisation de base etc.

> **L'opérateur de résolution de portée ::** *(Appelé aussi double deux points.)* pour simplifier, est utilisé pour référencer des éléments appartenant à une classe et non à un objet quelconque.

La deuxième occurrence de PERSO_ST7565 définit le constructeur PERSO_ST7565 qui par l'entremise de **::** appartient à cette classe. Les procédures et fonctions qui suivent commencent naturellement par void, mais il faut établir leur appartenance à la classe. La description d'une méthode doit donc respecter l'écriture :

```
void PERSO_ST7565::MAJ_PTR(Paramètres) { Code source }
```

…/…

Passage des paramètres utilisateur au constructeur.

Bien que les paramètres représentant les broches affectées aux Entrées/Sorties choisies par l'utilisateur de la bibliothèque ne vont pas changer, et sont généralement des constantes, il faut obligatoirement leur réserver des emplacements pour qu'elle puissent exister dans le contexte. Dans le fichier d'entête :

```
private : byte RST, A0, SCLK, ... Retro_Vert, SID;
```

Il faut alors, dans le corps { ... } de PERSO_ST7565 établir le lien entre ces variables et les paramètres de l'instance logicielle :

```
RST=broche_RST;
A0=broche_A0;
SCLK=broche_SCLK;
Retro_Bleu=Retro_eclairage_Bleu;
```

Dans tout le code source **on utilisera ensuite les identificateurs des variables** du genre A0 et non des paramètres tels que broche_A0. La relation Paramètre/Variable étant établie, on peut coder le programme source avec la syntaxe habituelle :

```
pinMode(SCLK, OUTPUT); digitalWrite(SCLK, LOW);
```

Résumé de l'architecture de base du fichier.ccp

```
#include "PERSO_ST7565.h" // Par filiation "récupère" les #includes de l'entête.
#define Contraste_Dynamique B10000001
#define LED_Arduino 13

PERSO_ST7565::PERSO_ST7565(byte broche_RST,  ... ) {
 RST=broche_RST;
 A0=broche_A0;
 SCLK=broche_SCLK;
        ...
 pinMode(Retro_Bleu, OUTPUT);
 pinMode(SCLK, OUTPUT); digitalWrite(SCLK, LOW);
 Envoyer_une_COMMANDE(B10100010);
        ...
 Afficheur_en_service(); Effacer_Ecran(); Surbrillance(0); }
```

Initialisations et configurations de la classe.

Puis suivent les diverses fonctions et procédures :

```
void PERSO_ST7565::Envoyer_une_COMMANDE(byte OCTET)
  { ... traitement relatif à la procédure ...}
```

... / ...

Le fichier KEYWORDS.txt

Strictement optionnel, ce fichier permet à l'IDE d'apprendre la table des mots clef spécifiques à la bibliothèque pour activer la coloration syntaxique. Quand les identificateurs sont spécifiés dans le fichier **keyword.txt** *(Lettres majuscules ou minuscules.)* ces symboles seront alors affichés en couleur ocre dans l'éditeur de l'**IDE** comme pour tous les autres mots du langage. La syntaxe est :

```
LCD_ST7565[TAB]KEYWORD1
Envoyer_une_COMMANDE[TAB]KEYWORD2
Envoyer_une_DONNEE[TAB]KEYWORD2
Contraste[TAB]KEYWORD2
Afficheur_en_veille[TAB]KEYWORD2
```

Chaque ligne commence par le nom du mot-clef, **suivi d'une TABULATION**, *(Pas d'espaces, et **UNE SEULE** TABULATION.)* suivi par le type de mot-clé. KEYWORD1 prévu pour les classes qui seront de couleur orange. KEYWORD2 pour les procédures et les fonctions qui seront affichées en Ocre. *(En version > 1.0 il semble que 1 et 2 se traduisent tous les deux de la couleur ocre.)* Il faut sortir puis redémarrer l'éditeur Arduino pour que la table des symboles soit prise en compte. Les lignes de commentaires doivent commencer par le caractère # dans le fichier keyword.txt.

Conventions sur les identificateurs : Les classes sont désignées par des caractères majuscules. Les fonctions par des minuscules "titrées" par des majuscules. Les constantes par conventions d'écriture seront avantageusement écrites en majuscules.

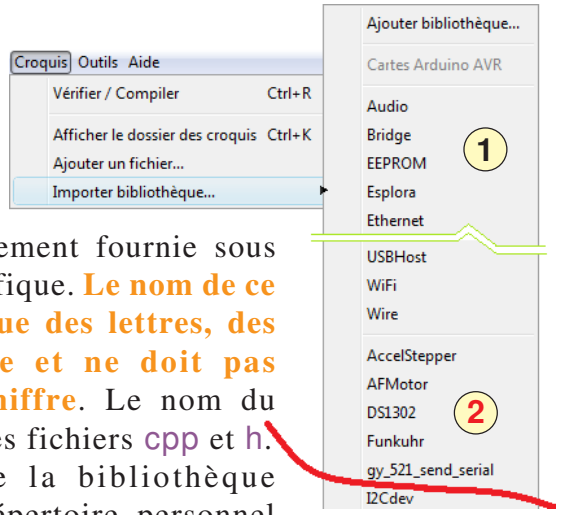
Choix du répertoire pour développer la bibliothèque :
Le plus simple consiste à placer le dossier dans le répertoire
*MonPC**Documents\Arduino\Librairies**\NomBibliotheque*

Le format pde pour les fichiers source.

Souvent, des exemples de programme Arduino sont donnés au format.pde qu'il faut transformer en textes compatibles avec Arduino. La méthode est simple. Il suffit de renommer le fichier pour remplacer le type pde par ino.

IMPORTER UNE BIBLIOTHÈQUE :

Les bibliothèques sont constituées d'une collection de code pour faciliter l'interfaçage d'un module ou d'un capteur spécifique sur une carte ARDUINO. Il existe des centaines de bibliothèques supplémentaires disponibles gratuitement sur Internet en téléchargement. Pour utiliser des bibliothèques supplémentaires, il faut les installer. Elles sont généralement fournies sous la forme d'un fichier ZIP ou d'un dossier. **Le nom du dossier est le nom de la bibliothèque**. Dans le dossier se trouve un fichier de type cpp, un fichier de type h et souvent un fichier keywords.txt, un dossier d'exemples et éventuellement d'autres fichiers requis par la bibliothèque.



1) Télécharger la bibliothèque dans un répertoire quelconque,
2) Décompacter le fichier ZIP dans ce répertoire.
   La librairie est généralement fournie sous forme d'un dossier spécifique. **Le nom de ce dossier ne comporte que des lettres, des chiffres, pas d'espace et ne doit pas commencer par un chiffre**. Le nom du dossier est aussi celui des fichiers cpp et h.
3) Placer le dossier de la bibliothèque décompressé dans le répertoire personnel <C:\! ARDUINO\! Bibliothèques importées>.
4) Ajouter la nouvelle bibliothèque dans la liste disponible :
   Croquis > Importer bibliothèque.. > Add Library... >
   Indiquer le chemin du dossier personnel sur le H.D. *(Précisé en (2).)* La bibliothèque est alors ajoutée dans le dossier personnel :
   <C:\Users\MICHEL\Documents\Arduino\libraries>
   ou éventuellement dans
   <C:\Users\user\Documents\Arduino\libraries>.
5) Recharger un croquis quelconque. Cliquer sur Croquis puis placer le curseur de la souris sur Importer bibliothèque.. : En **1** on trouve les bibliothèques par défaut et en **2** les bibliothèques importées.

Déclarer les bibliothèques utilisées dans un programme.

Deux syntaxes équivalentes sont possibles pour indiquer les délimiteurs d'une bibliothèque dont on veut utiliser les procédures :

```
#include <Wire.h>
#include <I2Cdev.h>
#include <MPU6050.h>
```

OU

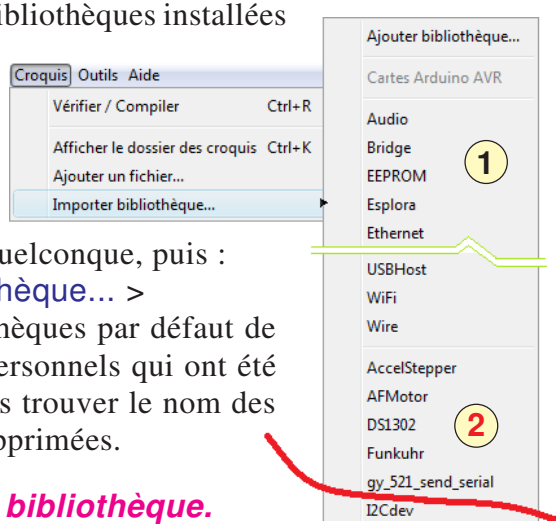
```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050.h"
```

REMARQUE : Avec l'utilisation des délimiteurs "**<**" les bibliothèques par défaut faisant partie de l'environnement d'Arduino passent en couleur marron dans l'éditeur de l'**IDE**.

Enlever une bibliothèque importée de l'IDE.

Aller dans le dossier :
C:\Users\MICHEL\Documents\Arduino\libraries.
On y trouve l'ensemble des bibliothèques installées par l'utilisateur.

Y effacer les dossiers des bibliothèques que l'on désire supprimer.



Ouvrir un ***programme.ino*** quelconque, puis :
Croquis > Importer bibliothèque... >
En **1** nous avons les bibliothèques par défaut de l'IDE et en **2** les modules personnels qui ont été ajoutés. En **2** on ne doit plus trouver le nom des bibliothèques qui ont été supprimées.

Utiliser facilement une bibliothèque.

Pour importer une bibliothèque dans un programme, on peut directement écrire les lignes de déclaration indiquées dans l'encadré ci-dessus. Il est toutefois bien plus commode d'utiliser la suite de commande listée ci-dessous :

- Cliquer sur Croquis >
- Déplacer le curseur sur Importer bibliothèque... >
- **BGS** sur le nom de la bibliothèque à incorporer au programme.

Quelle que soit la position actuelle du curseur dans l'éditeur, la déclaration sera placée en tête de programme, avec éventuellement des bibliothèques associées.