Lycée TOUCHARD WASHINGTON

Développement logiciel

Introduction informatique embarquée

OBJECTIFS

Découvrir la programmation dans le domaine de l'informatique embarquée en utilisant un ESP32. Ce document se concentre sur des exemples concrets basés sur les composants de la carte ESP32-SNIR développée dans la section Informatique et Réseaux.

Ce cours vise également à introduire les notions de programmation orientée objet (POO) sans créer de nouvelles classes, mais plutôt en utilisant les objets disponibles dans les différentes bibliothèques ou librairies de l'environnement de développement de l'ESP32. Les principales fonctions de ces librairies sont décrites dans ce document.

Préambule

L'ensemble des exemples présentés dans ce cours est rédigé en **langage C++**. Le code est développé au moyen de l'IDE **NetBeans**, utilisant le framework **PlatformIO**. Aucune nouvelle classe n'est créée dans ce document. Seule, les classes disponibles dans les bibliothèques dédiées à la gestion des composants intégrés sur la carte **ESP32-SNIR**, sont utilisées.

Vocabulaire :

Une **classe** représente une catégorie ou un ensemble d'objets partageant des attributs ou des données similaires, ainsi que des méthodes communes. Elle agit comme un modèle, définissant la structure et le comportement que les objets de cette classe vont posséder. Ce concept équivaut à la notion de type en langage C.

Les **attributs** font référence aux données encapsulées dans une classe. Généralement, ils ne sont pas accessibles directement depuis l'extérieur de la classe.

Les **méthodes** représentent les opérations ou fonctions qui agissent sur ces attributs. Elles constituent l'interface utilisable par le programmeur pour manipuler les attributs de la classe.

Une **instance** est une occurrence spécifique d'une classe, semblable à une variable dans le langage C. Elle est créée en utilisant le modèle fourni par la classe.

Instanciation d'une classe de manière automatique :

```
SSD1306 afficheurOLED(0x3C,21,22);
```

Accès à une méthode de la classe :

```
afficheurOLED.init();
```

Ce code crée une **instance** nommée **`afficheurOLED`** de la classe **`SSD1306`**, avec les paramètres spécifiés (adresse et broches). Ensuite, la méthode **`init()`** de l'objet **`afficheurOLED`** est appelée pour initialiser cet écran OLED spécifique.

Sommaire

1. Définition

L'informatique embarquée est l'intégration d'un système informatique dans un appareil ou un système qui n'a pas pour fonction principale d'être un ordinateur. Les systèmes informatiques embarqués sont typiquement conçus pour être compacts, économes en énergie et fiables. Ils répondent généralement à des besoins spécifiques tels que le contrôle de processus, la collecte de données ou la communication.

Les systèmes embarqués sont utilisés dans une large gamme d'applications, notamment :

- **Les appareils électroménagers**, tels que les réfrigérateurs, les fours et les lave-linge.
- **Les appareils électroniques grand public**, tels que les téléviseurs, les lecteurs multimédias.
- **Les automobiles**, avec les systèmes de contrôle du moteur, les systèmes de navigation et les systèmes de sécurité.
- **Les appareils médicaux**, tels que les stimulateurs cardiaques et les appareils d'imagerie médicale.
- **Les machines industrielles**, telles que les robots, les machines-outils et les systèmes de contrôle de la production.

2. Architecture des systèmes embarqués

Les systèmes informatiques embarqués sont généralement composés des éléments suivants :

- **Un microprocesseur**, qui est le cerveau du système.
- **De la mémoire**, qui est utilisée pour stocker le code et les données.
- **Des périphériques d'entrées/sorties**, qui permettent d'interagir avec le monde extérieur par l'intermédiaire de capteurs et d'actionneurs.
- En option, **une interface homme-machine** réalisée à l'aide d'un afficheur pour l'interaction avec les utilisateurs.

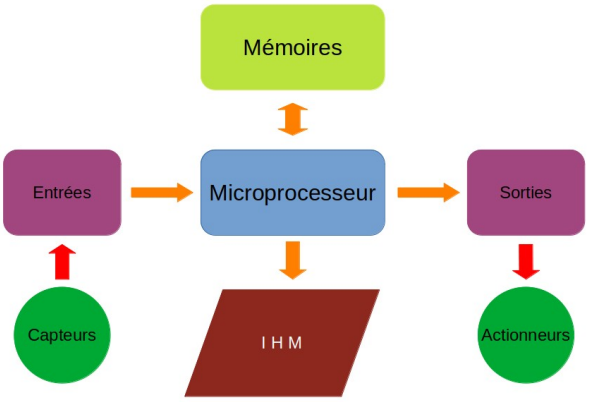


Figure 1: Architecture des systèmes embarqués

2.1. Le microcontrôleur

Au cœur du système embarqué se trouve le microprocesseur, responsable des calculs arithmétiques et logiques ainsi que du séquencement des opérations.

Ce type de processeur n'est pas forcement très puissant dans un système embarqué. Par contre, il est souvent associé sur la même puce électronique à de multiples broches d'entrées/sorties, formant ainsi un microcontrôleur.

La production à grande échelle de ce type composant permet de le rendre économique.

2.2. Les entrées

Les entrées sont chargées de capturer les grandeurs physiques du monde réel, pouvant être soit des informations analogiques, soit numériques.

Des circuits spécialisés permettent au processeur de les traiter numériquement.

Par exemple :

Les convertisseurs analogiques/numériques, connus également sous l'acronyme CAN ou ADC en anglais, sont utilisés pour obtenir des grandeurs analogiques comme la température ou la luminosité.

Les entrées tout ou rien (TOR) détectent des présences ou l'état de boutons-poussoirs et d'interrupteurs.

Enfin, les entrées de comptage servent à mesurer des vitesses ou des distances.

Parfois, l'utilisation de circuits spécialisés est nécessaire pour adapter les tensions et rendre les signaux d'entrée compatibles avec le microcontrôleur.

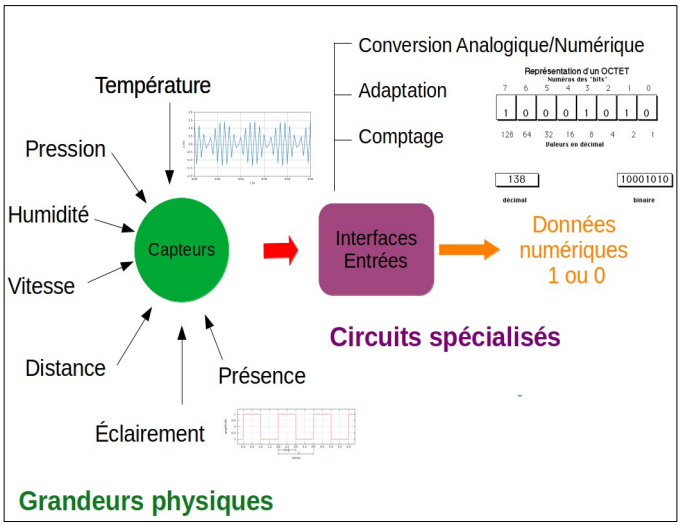


Figure 2: Les entrées des microcontrôleurs

2.3. Les sorties

Les sorties des systèmes embarqués influent sur le monde extérieur en contrôlant divers éléments tels que les moteurs, les vérins, les lampes… Elles sont responsables de l'actionnement de ces composants.

Les données numériques peuvent être converties en signaux analogiques par le biais de convertisseurs numériques / analogiques, connus sous l'acronyme CNA ou DAC en anglais.

Les sorties tout ou rien (TOR) sont utilisées pour commander des relais, des vannes... offrant ainsi un contrôle binaire en activant ou désactivant ces dispositifs.

Certains systèmes nécessitent des trains d'impulsions pour fonctionner, ils peuvent être générés soit par des sorties TOR, soit par des broches spécialisées associées à un **timer** ou minuterie programmable pour assurer la gestion du temps.

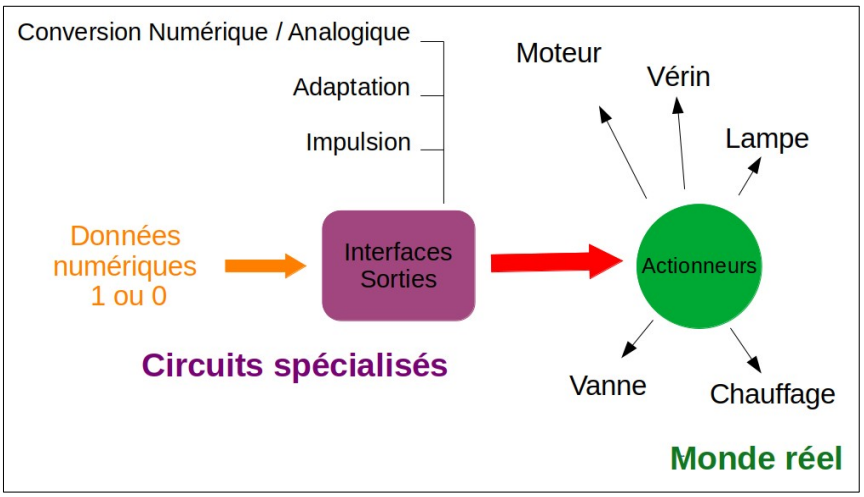


Figure 3: Les sorties des microcontrôleurs

Il est fréquemment nécessaire d'ajuster les tensions de sortie pour contrôler les actionneurs, car le microcontrôleur ne fournit généralement pas la puissance nécessaire au bon fonctionnement des composants du monde réel. Cela implique donc souvent l'utilisation de circuits d'adaptation de tension.

2.4. Les bus de communication

Les bus de communication sur les microcontrôleurs sont des canaux utilisés pour permettre aux différents composants d'un système embarqué de communiquer entre eux. Ces bus facilitent les échanges de données et de commandes entre le microcontrôleur et les périphériques qui peuvent lui être adjoints.

Il y a deux méthodes de communication avec un périphérique : la communication en parallèle, où tous les bits sont transmis simultanément, et la communication en série, où les bits sont envoyés un par un. La méthode en parallèle permet une transmission plus rapide, mais elle requiert de nombreuses broches du processeur. En revanche, la méthode en série utilise moins de broches, mais peut être moins performante, car elle transporte les bits un par un.

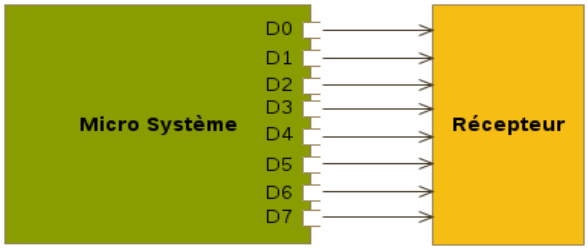


Figure 5: Communication parallèle

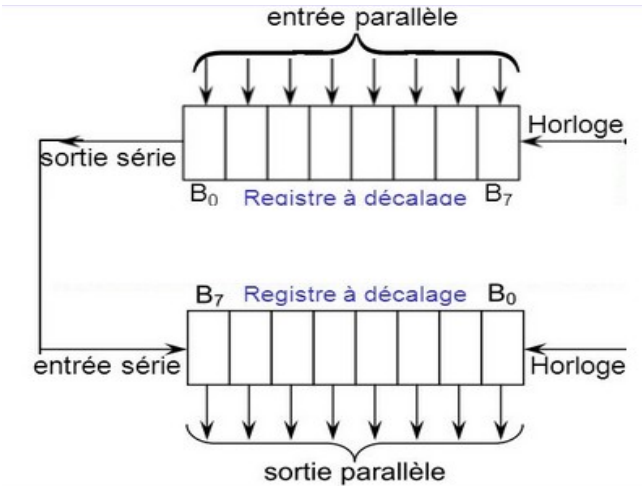


Figure 4: Communication série

Pour la communication en série, il en existe principalement deux formes. La liaison série, dite synchrone dans ce cas l'horloge de l'émetteur est fournie au récepteur. La liaison série, dite asynchrone ou seule les données sont transmises ce qui nécessite des mécanismes de synchronisation, souvent au niveau du caractère. La première étant plus performante et plus précise. La seconde est cependant plus populaire et est communément utilisée pour dialoguer avec un module GPS, un lecteur de code à barres...

Il existe également des bus spécialisés, principalement pour communiquer sur de très courtes distances avec des composants en périphérie du microcontrôleur. Voici quelques-uns des types de bus de communication couramment utilisés sur les microcontrôleurs :

- **Bus I2C (Inter-Integrated Circuit)** : Ce bus permet une communication série bidirectionnelle entre plusieurs périphériques à l'aide de seulement deux fils, un fil de données (SDA) et un fil d'horloge (SCL). Il est utilisé pour connecter des capteurs, des écrans, des mémoires EEPROM...
- **Bus SPI (Serial Peripheral Interface)** : Ce bus est également utilisé pour la communication série entre le microcontrôleur et des périphériques externes, mais il utilise généralement plus de fils que l'I2C. Le SPI implique des fils dédiés pour les données (MOSI - Master Out Slave In, MISO - Master In Slave Out) et une ligne d'horloge (SCLK - Serial Clock). Il est souvent utilisé pour des applications nécessitant des transferts de données rapides comme les communications avec des écrans TFT, des cartes SD…
- **Bus 1-Wire (1 fil) :** Ce bus est utilisé pour connecter des capteurs numériques.
- **Bus CAN (Controller Area Network)** : Ce bus est particulièrement utilisé dans les systèmes embarqués automobiles. Il permet la communication entre les différents contrôleurs présents dans un véhicule, comme les capteurs, les calculateurs… Il permet une transmission robuste et fiable des données.

Ces différents bus offrent des méthodes de communication adaptées à différents besoins en fonction de la vitesse, du nombre de périphériques connectés, du type de transferts de données, de la fiabilité, ce qui permet une grande flexibilité dans la conception des systèmes embarqués.

3. L'ESP32

L'ESP32, conçu par Espressif Systems, une entreprise basée en Chine, est une série de microcontrôleurs qui tire parti de l'architecture Xtensa LX6 de Tensilica une société basée en Californie. Il intègre la gestion du Wi-Fi et du Bluetooth pour les communications externes, ainsi qu'un grand nombre de broches d'entrée/sortie. De plus, il dispose d'un module DSP pour le traitement du signal et les calculs rapides.

Habituellement présenté sous forme de carte ou de "board", l'ESP32 inclut non seulement le microcontrôleur lui-même, mais également son alimentation et les connexions nécessaires pour interagir avec des composants externes. Parfois, la carte dispose de périphériques intégrés tels qu'un afficheur ou un module GSM.

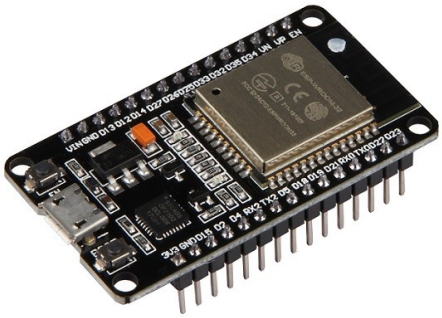


Figure 6: NodeMCU ESP32

Figure 7: Esp32 LILYGO - Lora

Figure 8: ESP32 - WROOM-32 chip set module

L'ESP32 est un microcontrôleur puissant et polyvalent qui peut être utilisé dans une large gamme d'applications. Il offre les fonctionnalités suivantes :

- **Processeur Dual-Core Xtensa LX6** cadencé à 160 ou 240 MHz et performant jusqu'à 600 millions d'instructions par secondes, unité de mesure DMIPS (Dhrystone MIPS)
- **Mémoire SRAM:** 520 ko, mémoire volatile (1 ko = 1024 octets).
- **Mémoire Flash:** 448 ko, mémoire non volatile, les données et programmes sont conservés même sans alimentation.
- **Connectivité sans fil:** Wi-Fi 802.11 b/g/n et Bluetooth 5.0.
- **Interfaces périphériques:** I2C, SPI, UART, ADC, DAC, RTC (horloge temps réel).
- **Gestion de l'alimentation:** faible consommation.

Il est compatible avec les outils de développement Arduino.

3.1. Le brochage de l'ESP32

Les cartes de développement se présentent avec plus ou moins de broches. Voici la version 30 broches, elle est utilisée couramment dans la section.

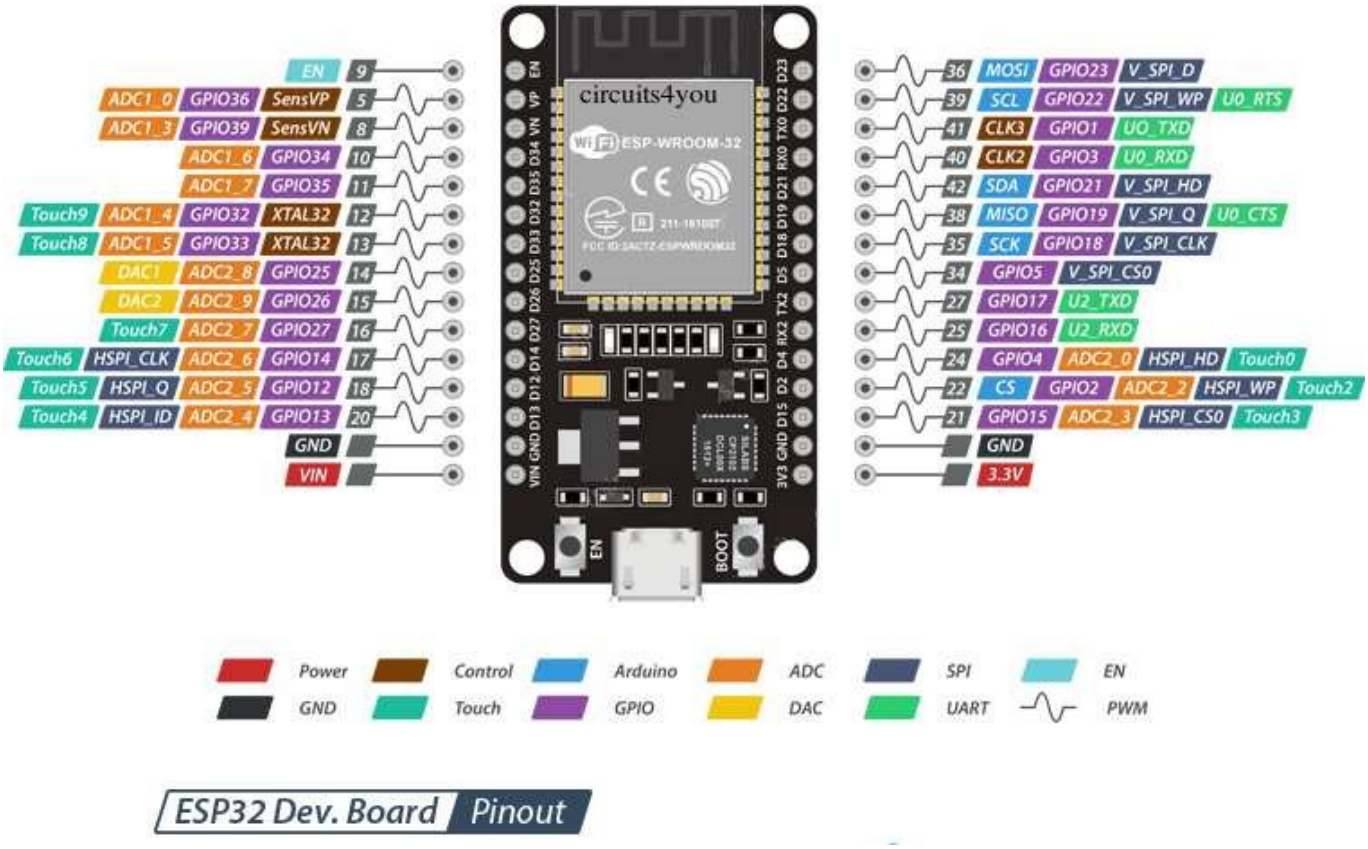


Figure 9: Brochage de la carte à base d'ESP32 version 30 broches

Toutes les broches de la puce ne sont pas câblées sur la carte de développement. Elle se compose de 25 broches d'entrée/sortie GPIOx pouvant être affectées à un usage particulier à la fois :

- •15 Canaux pour la conversion analogique numérique (ADC)**.**
- • 3 Interfaces SPI.
- • 2 Interfaces série de type UART (liaison série asynchrone).
- • 1 Interfaces I2C.
- •16 Canaux de sortie pour la modulation de largeur d'impulsion en anglais « Pulse Width Modulation » PWM.
- • 2 Sorties de convertisseur numérique analogique (DAC).
- • 2 Interfaces I2S.
- • 9 Broches d'entrée/sortie capacitive.

Les autres broches sont utilisées pour l'alimentation et le contrôle du système.

Elle dispose également d'une LED indiquant la mise sous tension de la carte et une LED reliée à la broche GPIO2.

Le port Micro-USB est utilisé pour la programmation du composant ou comme liaison série au travers de l'interface USB.

Le bouton-poussoir BOOT est utilisé éventuellement pour le téléchargement de nouveaux programmes

Le bouton-poussoir EN redémarre le microprocesseur.

3.2. Restrictions sur les broches de l'ESP32

Les broches de l'ESP32 sont uniquement utilisables en 3,3V. Il sera nécessaire de mettre un circuit d'adaptation de tension avec certains composants périphériques fonctionnant en 5V.

Certaines broches ont un usage particulier :

Broches uniquement en entrée

GPIO34	GPIO35	GPIO36	GPIO39

Broches disposant d'une résistance de tirage au +3,3V intégrée

GPIO14	GPIO16	GPIO17	GPIO18
GPIO19	GPIO21	GPIO22	GPIO23

Ce rappel au +3,3V fixe le potentiel de la broche, cela permet de définir clairement l'état logique attendu, évitant ainsi les ambiguïtés. Cela peut également aider à réduire les interférences électromagnétiques et le bruit qui pourrait perturber le signal. Une entrée en l'air peut faire antenne. Fixer le potentiel de la broche peut également contribuer à économiser l'énergie en évitant des courants de fuite.

Broches ne disposant d'une résistance de tirage au +3,3V intégrée

GPIO13	GPIO25	GPIO26	GPIO27
GPIO32	GPIO33		

Broches de détection capacitive

GPIO02	GPIO04	GPIO12	GPIO13
GPIO14	GPIO15	GPIO27	GPIO32
GPIO33			

L'ESP32 dispose de 9 capteurs capacitifs internes. Ces capteurs peuvent détecter les variations d'une charge électrique. Par exemple, la peau humaine peut retenir une charge électrique, ainsi, ils peuvent détecter les variations induites lorsqu'on touche les broches GPIOs avec un doigt. Ces broches peuvent être facilement intégrées en tant que pads capacitifs et remplacer les boutons mécaniques.

Broches d'entrées pour la conversion analogique numérique

L'ESP32 dispose en interne de 2 convertisseurs analogiques numériques 12 bits nommés ADC1 et ADC2. Avec la version 30 broches de la carte, toutes les broches ne permettent pas la conversion analogique numérique. Ils convertiront les tensions d'entrées comprises entre 0 et 3,3V en une valeur allant de 0 à 4095. La résolution du convertisseur est donc 3,3 / 4096 soit 0,8 mV.

ADC1	ADC1_0 (GPIO36)	ADC1_3 (GPIO39)	ADC1_4 (GPIO32)
	ADC1_5 (GPIO33)	ADC1_6 (GPIO34)	ADC1_7 (GPIO35)

ADC2	ADC2_0 (GPIO04)	ADC2_2 (GPIO02)	ADC2_3 (GPIO15)
	ADC2_4 (GPIO13)	ADC2_5 (GPIO12)	ADC2_6 (GPIO14)
	ADC2_7 (GPIO27)	ADC2_8 (GPIO25)	ADC2_9 (GPIO26)

Broches de sortie pour la conversion numérique analogique

L'ESP32 dispose en interne de 2 convertisseurs numériques analogiques 8 bits, ce qui signifie que les valeurs allant de 0 à 256 seront converties en une tension analogique allant de 0 à 3,3V.

DAC1 (GPIO25)	DAC2(GPIO26)

4. Utilisation de PlatformIO

PlatformIO est un ensemble d'outils open source. Ils sont conçus pour le développement de logiciels dans les domaines de l'embarqué et de l'Internet des Objets (IoT). Il est multiplateforme et peut être utilisé en ligne de commande ou intégré sous forme d'une extension ou plug-in en anglais dans divers environnements de développement tels que **NetBeans**, **QtCreator** ou **Visual Studio Code**. Cette plateforme offre des possibilités de développement pour de nombreuses cartes appelées **board** avec l'outil, incluant celles basées sur l'ESP32, ainsi que pour divers autres microcontrôleurs tels que la gamme des STM32 de STMicroelectronics ou la série ATmega utilisée dans les cartes Arduino.

La liste des cartes compatibles avec l'outil est accessible via la commande en ligne de commande, dont la syntaxe commence toujours par "**platformio**" ou "**pio**", ces deux termes étant synonymes.

```
pcruchet@b108tu4prof:~$ platformio boards
ou
pcruchet@b108tu4prof:~$ pio boards
```

La liste est impressionnante, pour restreindre à une famille, il est nécessaire de le préciser comme le montre l'exemple ci-dessous :

```
pcruchet@b108tu4prof:~$ pio boards lolin
Platform: espressif32
================================================================================
ID              MCU      Frequency  Flash    RAM    Name
--------------  -------  -----------  -------  -----  ------------------
lolin_c3_mini   ESP32C3  160MHz       4MB      320KB  WEMOS LOLIN C3 Mini
lolin_d32       ESP32    240MHz       4MB      320KB  WEMOS LOLIN D32
lolin_d32_pro   ESP32    240MHz       4MB      320KB  WEMOS LOLIN D32 PRO
lolin_s2_mini   ESP32S2  240MHz       4MB      320KB  WEMOS LOLIN S2 Mini
lolin_s2_pico   ESP32S2  240MHz       4MB      320KB  WEMOS LOLIN S2 PICO
lolin_s3        ESP32S3  240MHz       16MB     320KB  WEMOS LOLIN S3
lolin_s3_mini   ESP32S3  240MHz       4MB      320KB  WEMOS LOLIN S3 Mini
lolin32         ESP32    240MHz       4MB      320KB  WEMOS LOLIN32
lolin32_lite    ESP32    240MHz       4MB      320KB  WEMOS LOLIN32 Lite
```

4.1. Création d'un projet pour NetBeans

Pour commencer un projet avec NetBeans, quelques étapes préliminaires sont essentielles, tel que la mise en place du répertoire de travail et la création du projet via la ligne de commande. Le répertoire peut être réalisé soit avec un explorateur soit en ligne de commande.

```
pcruchet@b108tu4prof:~/Esp32$ mkdir TestEsp32
pcruchet@b108tu4prof:~/Esp32$ cd TestEsp32/
pcruchet@b108tu4prof:~/Esp32/TestEsp32$
```

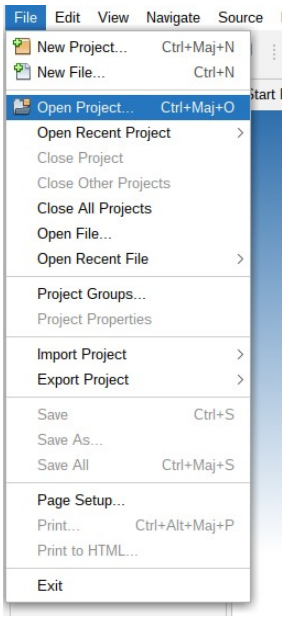
Dans le répertoire ainsi créé, la commande pio project init va initialiser un nouveau projet, il est nécessaire de préciser en arguments environnement et la carte utilisée pour le développement.

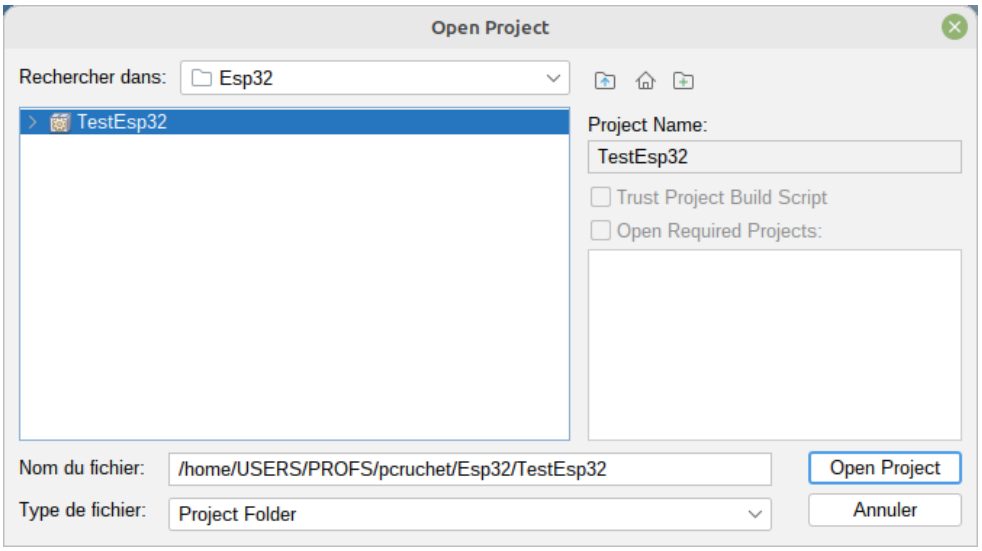
```
pcruchet@b108tu4prof:~/Esp32/TestEsp32$ pio project init --ide netbeans --board lolin32
The following files/directories have been created in pcruchet/Esp32/TestEsp32
include - Put project header files here
lib - Put project specific (private) libraries here
src - Put project source files here
platformio.ini - Project Configuration File
Resolving lolin32 dependencies...
Already up-to-date.
Updating metadata for the netbeans IDE...
Project has been successfully initialized!
```

Le projet est initialisé avec un dossier **include** destiné à accueillir les fichiers d'en-tête, un dossier **lib** prévu pour les bibliothèques spécifiques au projet, tandis que le répertoire **src** est dédié aux sources du projet. Le fichier **platformio.ini** sert de fichier de configuration. La commande vérifie toutes les dépendances pour la carte **lolin32** et les mises à jour nécessaires pour la plateforme de développement.

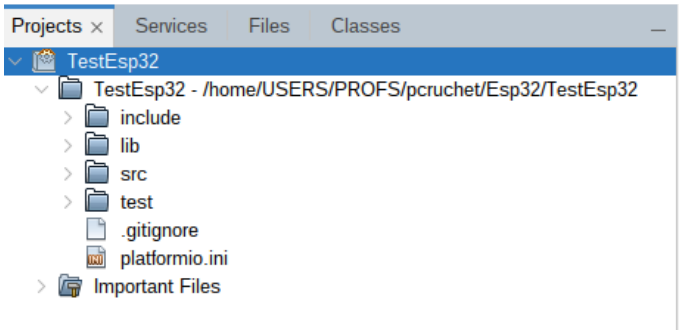
4.2. Ouverture du projet avec NetBeans

Le projet s'ouvre classiquement avec NetBeans :



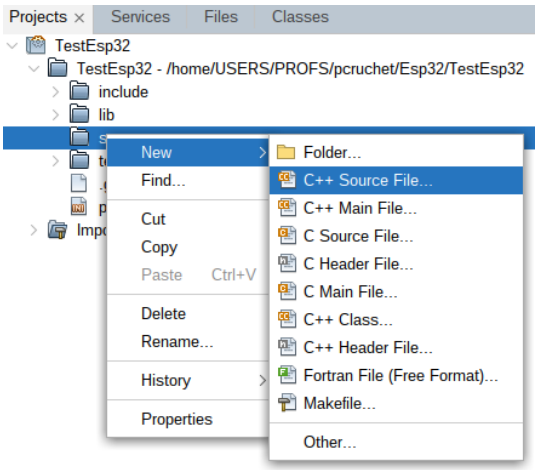


Les différents répertoires apparaissent dans l'onglet **Projects**. Le fichier plateformeio.ini contient les caractéristiques définies pour le projet.

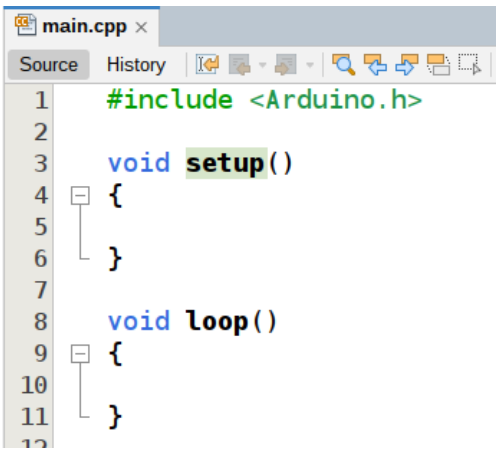



```
[env:lolin32]
platform = espressif32
board = lolin32
framework = arduino
```

Reste maintenant à créer un premier fichier pour recevoir le code source. Ce fichier est placé dans le dossier **src**, ce sera un **fichier source cpp**, en effet les bibliothèques disponibles pour le développement sont écrites en **C++**.



La structure d'un programme pour l'ESP32 avec Netbeans est toujours la même, elle reprend le schéma des fonctions utilisées pour l'Arduino :




```
1   #include <Arduino.h>
2
3   void setup()
4   {
5
6   }
7
8   void loop()
9   {
10
11  }
```

Inclusion du fichier **<Arduino.h>** et présence des deux fonctions **setup()** et **loop()**. La première est destinée à initialiser les périphériques et les variables, elle est exécutée une fois au démarrage du programme. La seconde constitue une boucle sans fin chargée du traitement attendu.

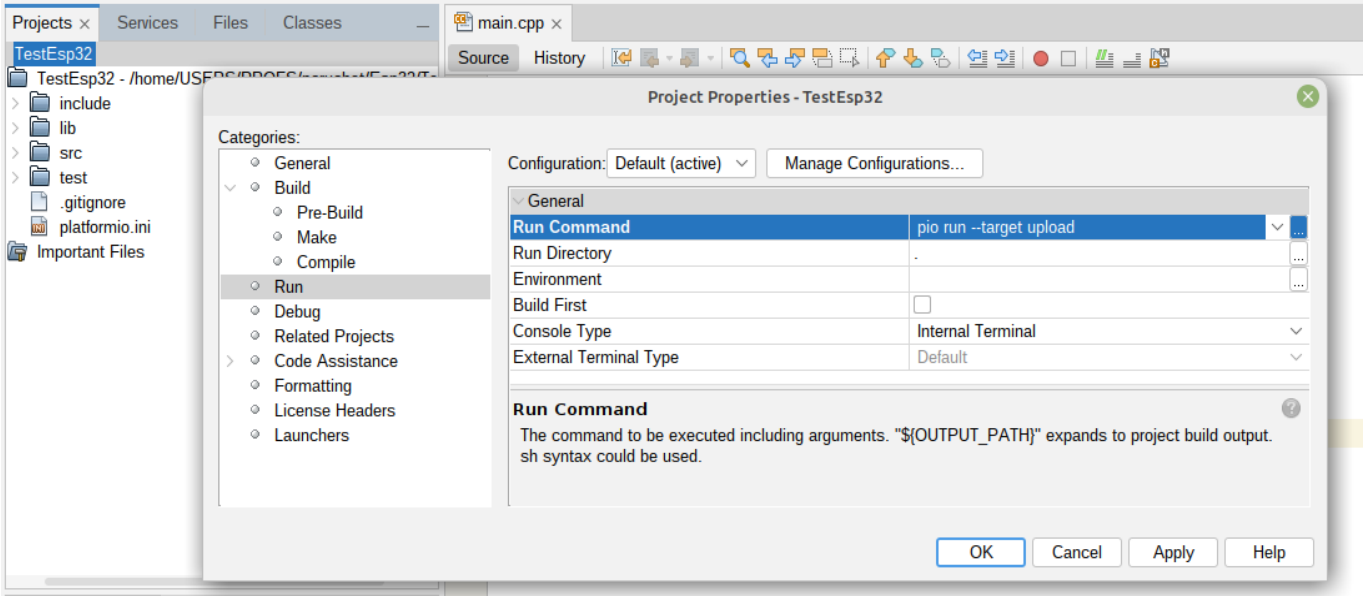
4.3. Premier programme avec l'ESP32

Ce premier programme va initialiser la broche GPIO02 en sortie. Cette dernière est connectée une LED précâblée sur la carte. Ensuite dans la boucle, cette LED est alternativement allumée puis éteinte avec une temporisation de 500ms entre chaque changement d'état.

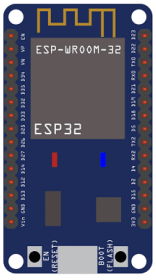
```
#include <Arduino.h>
uint8_t LED = 2;
void setup()
{
    pinMode(LED,OUTPUT);
}
void loop()
{
    digitalWrite(LED,HIGH);
    delay(500);
    digitalWrite(LED,LOW);
    delay(500);
}
```

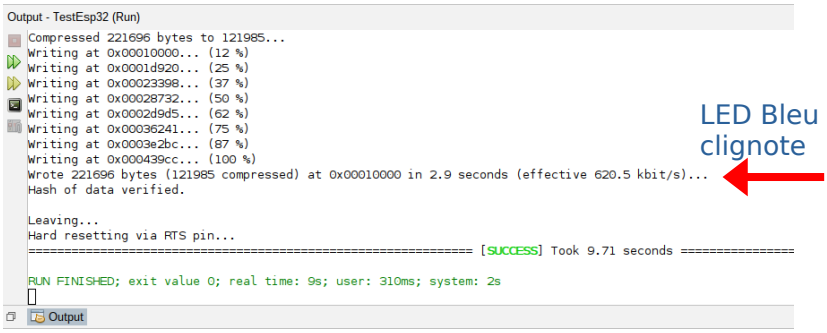
La compilation est effectuée de manière similaire à celle d'un programme classique en C ou C++. La différence principale réside dans la nécessité de téléverser le programme compilé dans l'ESP32 avant son exécution. Pour ce faire, il est nécessaire d'ajouter la commande **pio run** avec l'option de téléversement dans la cible dans les propriétés du projet.

```
pio run -target upload
```



Une fois la cible connectée via USB, le programme peut être lancé. S'il n'a pas déjà été compilé, il le sera, puis téléversé dans l'ESP32 pour être exécuté.





5. Programmation de l'ESP32

Les exemples de programme proposés dans ce document sont basés sur la carte ESP32-SNIR. Elle dispose de :

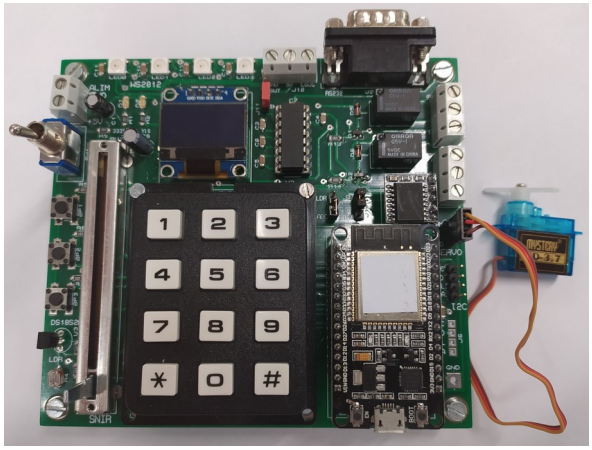


Figure 10: Carte ESP32-SNIR

- 3 boutons-poussoirs,
- 1 interrupteur.
- 4 LED RGB,
- 2 LED classique, une rouge une verte.
- 1 potentiomètre à glissière relié sur une entrée analogique
- 1 LDR pour capter la luminosité
- 1 capteur de température DS18S20
- 2 relais
- 1 afficheur graphique 128x64 pixels 0,96 pouce - SSD1306
- 1 clavier matricé 12 touches
- 1 Liaison série RS232
- 3 connecteurs pour I2C, dont un pour connecter une RTC
- 1 connecteur pour un servomoteur

5.1. Gestion des entrées/sorties numériques

Trois fonctions de la bibliothèque Arduino permettent de gérer simplement les broches de l'ESP utilisées en tant qu'entrées/sorties numériques.

```
void pinMode(uint8_t _pin, uint8_t _mode);
```

Paramètres :

_pin : représente le numéro de la broche GPIOx.

_mode : peut prendre les valeurs : **INPUT**, **OUTPUT** ou **INPUT_PULLUP**.

Rôle :

Cette fonction initialise le port dans un des trois modes disponibles : en tant qu'**entrée**, en tant que **sortie,** ou en tant qu'**entrée avec résistance interne activée** pour un tirage au +3,3V. Habituellement, elle est employée à dans la fonction **setup()**.

```
void digitalWrite(uint8_t _pin, uint8_t _value);
```

Paramètres :

_pin : représente le numéro de la broche GPIOx.

_value : peut prendre les valeurs : **HIGH** ou **LOW**.

Rôle :

Cette fonction permet de définir l'état logique d'une broche GPIOx configurée en sortie, la mettant soit à l'état haut avec **HIGH**, soit à l'état bas avec **LOW**.

```
uint8_t digitalRead(uint8_t _pin);
```

Paramètres :

_pin : représente le numéro de la broche GPIOx.

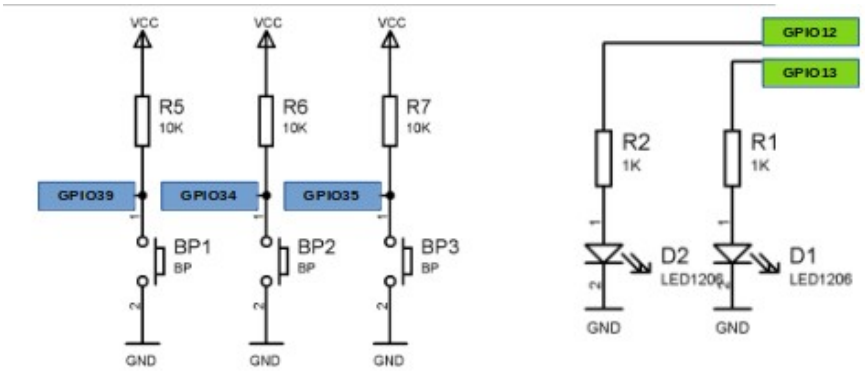
valeur de retour : état logique de la broche GPIOx **HIGH** ou **LOW**

Rôle :

Cette fonction permet de lire l'état logique d'une broche GPIOx configurée en entrée ou en sortie, en effet il est possible de relire l'état d'une sortie. Les valeurs possibles sont HIGH pour état logique haut et LOW pour l'état logique bas.

Exemple d'utilisation :

Dans cet exemple, les boutons-poussoirs BP1, BP2 et BP3 vont être mis en œuvre ainsi que les LED D1 et D2. Les entrées/sorties utilisées sont indiquées sur le schéma ci-dessous.



Objectif du programme :

BP1	Allume la LED **D1**
BP2	Allume la LED **D2**
BP3	Éteint les 2 LED

Figure 11: Boutons-poussoir et LED de la carte ESP32-SNIR

include/esp32_snir.h

```
#ifndef ESP32_SNIR_H
#define ESP32_SNIR_H
#include <Arduino.h>
#define BP1 39
#define BP2 34
#define BP3 35
#define LED  2
#define D1 13
#define D2 12
#endif /* ESP32_SNIR_H */
```

src/main.cpp

```
#include <Arduino.h>
#include "esp32_snir.h"
void setup()
{
    pinMode(BP1,INPUT);
    pinMode(BP2,INPUT);
    pinMode(BP3,INPUT);
    pinMode(D1,OUTPUT);
    pinMode(D2,OUTPUT);
    digitalWrite(D1,LOW);
    digitalWrite(D2,LOW);
}
void loop()
{
    if(digitalRead(BP1) == LOW)
        digitalWrite(D1,HIGH);
    if(digitalRead(BP2) == LOW)
        digitalWrite(D2,HIGH);
    if(digitalRead(BP3) == LOW)
    {
        digitalWrite(D1,LOW);
        digitalWrite(D2,LOW);
    }
}
```

Le schéma de câblage des boutons-poussoirs indique qu'ils sont activés lorsque le niveau de tension est bas **LOW**. Ainsi, le test « **if**(digitalRead**(BPx)** == LOW**)** » vérifie si le bouton-poussoir **x** est enfoncé.

5.2. Utilisation de la liaison série

Les interactions avec l'utilisateur peuvent s'effectuer au travers d'une des liaisons série disponibles sur l'ESP32. La classe **HardwareSerial** est responsable de la gestion de ces liaisons, à la fois dans l'environnement Arduino et pour des microcontrôleurs tels que l'ESP32. Un objet global nommé **`Serial`** est automatiquement instancié pour le premier port, offrant un accès universel dans l'ensemble de votre programme pour communiquer via la liaison série.

Par défaut, sur les cartes ESP32, ce premier port utilise les broches **GPIO1** (TX) et **GPIO3** (RX). Il est physiquement connecté au port USB de la carte. Cette même liaison étant utilisée pour la programmation du composant, il est donc primordial de l'initialiser dans votre application et de fermer tout utilitaire, par exemple **Putty**, qui permet le dialogue avec l'utilisateur, avant de lancer un téléversement d'un nouveau programme.

5.2.1. Description des principales méthodes de la classe HardwareSerial

Voici quelques méthodes utiles pour l'utilisation de **Serial** :

```
void begin(unsigned long _baud);
```

Paramètres :

_baud : représente la vitesse du port série.

Rôle :

Cette fonction ouvre le port série et initialise sa vitesse. Elle est appelée généralement dans la fonction **setup()**. Elle possède plusieurs autres paramètres non présentés ici par souci de simplification avec des valeurs par défaut. Ils ne sont généralement pas modifiés. La liaison série possède ainsi par défaut les caractéristiques suivantes : **8 bits de données**, **pas de parité** et **1 bit de stop**.

La méthode **print()**, utilisée pour afficher, existe suivant différentes formes. Une partie est présentée ici :

```
size_t print(char);
size_t print(const char []);
size_t print(const String &);
```

Paramètres :

La méthode print est surchargé pour recevoir un caractère, une chaîne de caractères ou une String
valeur de retour : nombre de caractères écrits.

```
size_t print(unsigned char, int = DEC);
size_t print(int, int = DEC);
size_t print(unsigned int, int = DEC);
size_t print(long, int = DEC);
size_t print(unsigned long, int = DEC);
size_t print(long long, int = DEC);
size_t print(unsigned long long, int = DEC);
```

Paramètres :

La méthode print est surchargée pour recevoir en 1er paramètre un entier, quelque soit sa taille.
Le 2e paramètre représente la base DEC, HEX, OCT, BIN, pour décimal, hexadécimal, octal ou binaire.
Sans préciser le 2e paramètre, l'entier est affiché en décimal.
valeur de retour : nombre de caractères écrits.

```
size_t print(double, int = 2);
```

Paramètres :

La méthode print est surchargée pour recevoir en 1er paramètre un réel.
Le 2e paramètre représente le nombre de décimales après la virgule
Sans préciser le 2e paramètre, le réel est affiché avec 2 décimales après la virgule.
valeur de retour : nombre de caractères écrits.

Rôle :

Ces fonctions écrivent sur la liaison série la valeur reçue en 1er paramètre et retourne le nombre de caractères réellement écrits. Des options sont possibles en fonction d'un éventuel 2e paramètre.

Il existe une variante à toutes ces méthodes nommée **println()**. Cette fonction ajoute un retour chariot '\n' à la fin de l'affichage.

```
int available(void);
```

Paramètres :

valeur de retour : nombre de caractère disponible dans le tampon du port série

Rôle :

Cette fonction permet de savoir si des caractères ont été reçus sur le port série.

Voici les deux principales méthodes pour la lecture du port série :

```
int read(void);
```

Paramètres :

valeur de retour : le code ASCII du caractère reçu

```
size_t read(uint8_t *_buffer,size_t _size);
```

Paramètres :

_buffer : adresse d'un tampon (tableau d'octets ou de caractères)
_size : taille du tampon.
valeur de retour : nombre d'octets lu réellement

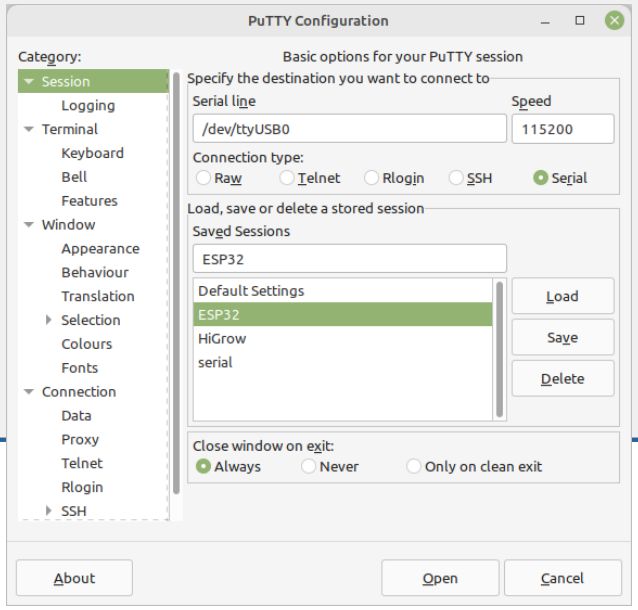
Rôle :

Cette fonction lit l'octet ou les octets reçus sur le port série en fonction de sa forme avec ou sans paramètre.

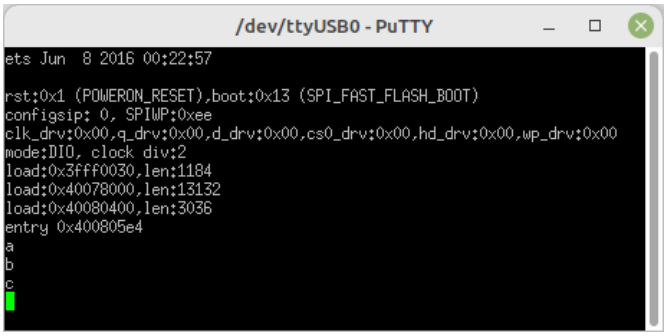
5.2.2. Exemple d'application :

Le programme suivant lit un caractère en provenance de la liaison série via le port USB et le renvoie sur le même port. La liaison est initialisée à 115200 bauds.

```
#include <Arduino.h>

void setup()
{
    Serial.begin(115200);
}
void loop()
{
    char carLu;
    if(Serial.available()>0)
    {
        carLu = Serial.read();
        Serial.println(carLu);
    }
}
```



Une fois le programme téléversé sur l'ESP32, le port /dev/ttyUSB0 de l'ordinateur est ouvert, la vitesse de transfert est fixée à 115200 bauds.



Au clavier, les touches 'a', 'b', 'c' ont été enfoncées, elles sont renvoyées à l'écran.

5.3. Utilisation de l'afficheur graphique

La carte ESP32-SNIR dispose d'un afficheur OLED du type SSD1306 piloté en I2C par l'ESP32 à l'adresse **0x3C**. Il s'agit d'un afficheur graphique avec une géométrie de 128 x 64 pixels. Les broches utilisées sont **SDA** GPIO21 et **SCL** GPIO22.

Cet afficheur utilise une librairie externe par exemple « **ESP8266 and ESP32 OLED driver for SSD1306 displays** » développé par ThingPulse – Fabrice Weinberg.

5.3.1. Installation de la librairie

Vérification des librairies installées de manière globale sur l'ordinateur.

```
pcruchet@b107PCT:~$ pio pkg list -g
Platforms
└── espressif32 @ 6.0.0 (required: platformio/espressif32)
Tools
├── framework-arduinoespressif32 @ 3.20006.221224
|   (required: platformio/framework-arduinoespressif32)
├── tool-esptoolpy @ 1.40400.0 (required: platformio/tool-esptoolpy)
├── tool-mkfatfs @ 2.0.1 (required: platformio/tool-mkfatfs)
├── tool-mklittlefs @ 1.203.210628 (required: platformio/tool-mklittlefs)
├── tool-mkspiffs @ 2.230.0 (required: platformio/tool-mkspiffs)
├── tool-scons @ 4.40400.0 (required: platformio/tool-scons)
└── toolchain-xtensa-esp32 @ 8.4.0+2021r2-patch5 (required: espressif/toolchain-
xtensa-esp32)
Libraries
└── ESP8266 and ESP32 OLED driver for SSD1306 displays @ 4.3.0
    (required: thingpulse/ESP8266 and ESP32 OLED driver for SSD1306 displays)
```

La librairie pour l'afficheur est bien disponible. Elle possède, entre autres, la classe **SSD1306** qui sera utilisée pour piloter le composant. Si ce n'est pas le cas, il est nécessaire de la rechercher dans la liste des librairies disponibles.

La commande **pkg search** de PlatformIO suivit d'un mot clé permet cette recherche exemple ici avec la référence de l'afficheur OLED : **SSD1306**

```
pcruchet@b107PCT:~$ pio pkg search SSD1306
Found 96 packages (page 1 of 10)
mbed-turkishp/SSD1306
Library • 0.0.0+sha.4b5e234b4f3e • Published on Thu Jun 13 01:08:01 2019
A buffered display driver for the SSD1306 OLED controller. Please note that this is
a work-in-progress; only very rudimentary drawing support is provided.

codewitch-honey-crisis/htcw_ssd1306
Library • 1.2.4 • Published on Sun Jan 22 00:51:34 2023
Provides support for the SSD1306 display device w/ GFX

somhi/ESP8266 SSD1306
Library • 1.0.0 • Published on Tue May 17 21:27:37 2016
SSD1306 oled driver library for 'monochrome' 128x64 and 128x32 OLEDs!

adafruit/Adafruit SSD1306
Library • 2.5.9 • Published on Wed Nov 15 12:58:05 2023
SSD1306 oled driver library for monochrome 128x64 and 128x32 displays

thingpulse/ESP8266 and ESP32 OLED driver for SSD1306 displays
Library • 4.4.0 • Published on Wed Mar 22 21:31:07 2023
I2C display driver for SSD1306 OLED displays connected to ESP8266, ESP32, Mbed-OS.
The following geometries are currently supported: 128x64, 128x32, 64x48. The init
sequence was inspired by Adafruit's library for the same display.
```

Le choix se fait en fonction du processeur, des dates de publications des caractéristiques de la librairie et des essais qu'il est nécessaire de faire. Dans notre cas, la dernière fonctionne correctement.

Pour son installation, la commande **pkg install** de PlatformIO réalise le traitement. Les librairies peuvent être installées uniquement pour le projet courant, si on ne précise rien, soit de manière globale avec le paramètre **-g** derrière la commande **install**, c'est-à-dire pour les différents projets réalisés avec votre profil utilisateur. C'est cette solution qui est retenue ici pour disposer de la librairie dans tous les futurs projets. Le nom de la librairie doit être entre guillemets à cause des espaces.

```
pcruchet@b107PCT:~$ pio pkg install -g --library "thingpulse/ESP8266 and ESP32 OLED
driver for SSD1306 displays"
Library Manager: Installing thingpulse/ESP8266 and ESP32 OLED driver for SSD130>
Downloading  [####################################]  100%
Unpacking  [####################################]  100%
Library Manager: ESP8266 and ESP32 OLED driver for SSD1306 displays@4.4.0 has b>
```

5.3.2. Description des principales méthodes de la classe SSD1306

Les deux méthodes suivantes permettent d'instancier et d'initialiser l'afficheur OLED SSD1306. Les paramètres du constructeur doivent tenir compte des caractéristiques de la carte ESP32-SNIR.

```
SSD1306(uint_8 _add,int _sda,int _scl);
```

Paramètres :

_add : adresse sur le port I2C (la valeur doit être à 0x3C pour la carte ESP32_SNIR)
_sda : numéro de la broche SDA (la valeur doit être 21 pour la carte ESP32_SNIR)
_scl : numéro de la broche SCL (la valeur doit être 21 pour la carte ESP32_SNIR)
Le constructeur possède plusieurs autres paramètres, non présentés ici par souci de simplification.
les valeurs par défaut correspondent aux besoins de la carte ESP32_SNIR

Rôle :

Le constructeur de la classe initialise la communication I2C de l'ESP32 et initialise l'afficheur OLED en fonction de ses caractéristiques. Le constructeur est appelé lors de l'instanciation de la classe SSD1306.

```
int init(void);
```

Paramètres :

aucun

Rôle :

Cette méthode termine l'initialisation de l'afficheur OLED, elle est appelée dans la méthode **setup()** généralement.

La méthode suivante permet de retourner l'affichage par rapport à l'implantation de l'afficheur sur la carte ESP32-SNIR.

```
int flipScreenVertically(void);
```

Paramètres :

aucun

Rôle :

Cette méthode retourne l'écran verticalement afin de s'adapter à l'orientation de l'afficheur sur la carte.

La méthode suivante permet de choisir la police de caractères utilisée par l'afficheur, par défaut la valeur est **ArialMT_Plain_10.** Les autres polices connues sont **ArialMT_Plain_16**, et **ArialMT_Plain_24**

```
void setFont(const uint8_t *_fontData);
```

Paramètres :

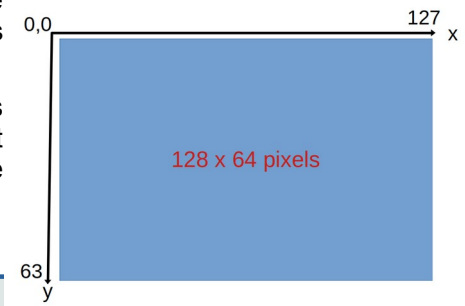
_fontData : pointeur sur le tableau contenant la police de caractères.

Rôle :

Cette méthode modifie l'affichage de la police (taille et forme) en fonction du paramètre reçu.

Il s'agit d'un dispositif affichage graphique. En ce qui concerne l'affichage, différentes méthodes sont donc utilisées pour "dessiner", généralement chaque méthode commence par le terme "**draw**...". Typiquement, les deux premiers paramètres dans ces méthodes représentent les coordonnées en x et y du point de départ de la forme ou du texte.

La disposition de l'afficheur graphique est structurée comme une matrice de 128 pixels sur 64. Les valeurs de x varient de 0 à 127, tandis que celles de y vont de 0 à 63.



Le programme ne manipule pas directement l'écran de l'afficheur, mais opère plutôt sur un tampon mémoire. Une fois la figure complètement créée dans ce tampon, une méthode spécifique permet d'afficher le contenu sur l'écran graphique.

Les méthodes graphiques sont les suivantes :

```
void drawLine(int16_t _x0,int16_t _y0,int16_t
_x1,int16_t _y1);
```

Paramètres :

- **_x0 :** abscisse du point 0.
- **_y0 :** ordonnée du point 0.
- **_x1 :** abscisse du point 1.
- **_y1 :** ordonnée du point 1.

Rôle :
Cette méthode trace une ligne entre les points 0 et 1

```
void drawRect(int16_t _x,int16_t _y,int16_t _width,int16_t _height);
```

Paramètres :

- **_x :** abscisse du point d'origine.
- **_y :** ordonnée du point d'origine
- **_width :** largeur du rectangle
- **_height :** hauteur du rectangle

Rôle :
Cette méthode trace un rectangle à partir des paramètres fournis. Attention à l'orientation de l'afficheur.

```
void drawCircle(int16_t _x,int16_t _y,int16_t _radius);
```

Paramètres :

- **_x :** abscisse du centre.
- **_y :** ordonnée du centre.
- **_radius :** rayon du cercle.

Rôle :
Cette méthode trace un cercle à partir de son centre et de son rayon

```
void drawTriangle(int16_t _x0,int16_t _y0,int16_t _x1,int16_t _y1,
                  int16_t _x2,int16_t _y2);
```

Paramètres :

- **_x0 :** abscisse du point 0.
- **_y0 :** ordonnée du point 0.
- **_x1 :** abscisse du point 1.
- **_y1 :** ordonnée du point 1.
- **_x2 :** abscisse du point 2.
- **_y2 :** ordonnée du point 2.

Rôle :
Cette méthode trace un triangle passant par les trois points dont les coordonnées sont fournies en paramètres.

```
void drawProgressBar(int16_t _x,int16_t _y,int16_t _width,int16_t _height,
                     int18_t _progress);
```

Paramètres :

_x : abscisse du point d'origine.
_y : ordonnée du point d'origine
_width : largeur du rectangle
_height : hauteur du rectangle
_progress : état de la progression, valeur entre 0 et 100 %

Rôle :
Dessine une barre de progression arrondie avec des dimensions externes données par la largeur (**width**) et la hauteur (**height**). La progression est une valeur non signée (**unsigned byte**) comprise entre 0 et 100.

```
void drawHorizontalLine(int16_t _x,int16_t _y,int16_t _length);
void drawVerticalLine(int16_t _x,int16_t _y,int16_t _length);
```

Paramètres :

_x : abscisse du point de départ.
_y : ordonnée du point de départ.
_length : longueur du segment.

Rôle :
Ces méthodes tracent une ligne respectivement horizontale ou verticale de longueur définie par le paramètre **_length** à partir du point de départ.

```
void drawFastImage(int16_t _x,int16_t _y,int16_t _width,int16_t _height,
                   int18_t *_image);
void drawXbm(int16_t _x,int16_t _y,int16_t _width,int16_t _height,
                   int18_t *_xbm);
```

Paramètres :

_x : abscisse du point d'origine.
_y : ordonnée du point d'origine
_width : largeur de l'image
_height : hauteur de l'image
_image : pointeur désignant un tableau de pixels
ou **_xbm :** pointeur désignant une image au format xbm

Rôle :
Ces méthodes affichent une image sur l'écran (attention, l'écran ne possède que 128x64 pixels).

```
void drawIco16x16(int16_t _x,int16_t _y,const uint8_t *_ico,bool _inverse=false) ;
```

Paramètres :

_x : abscisse du point d'origine.
_y : ordonnée du point d'origine
_ico : pointeur désignant l'icône 16x16 pixels
_inverse : true affiche en inversion vidéo, par défaut la valeur est **false** affichage normal

Rôle :
Dessine un icône sur l'écran, avec les trois premiers paramètres l'icône s'affiche tel qu'il a été conçu. Si le 4e possède la valeur **true**, l'affichage s'inverse.

En ce qui concerne l'affichage de texte, les méthodes de la classe commencent également par **draw**…

```
uint16_t drawString(int16_t _x,int16_t _y,const String &_text);
```

Paramètres :

_x : abscisse du point d'origine.
_y : ordonnée du point d'origine.
_text : texte à afficher sous la forme d'une String (chaîne de caractères).
Valeur de retour : nombre de caractères qui ont pu être écrits.

Rôle :
Écrit le texte passé, à partir du point d'origine. Attention, l'origine dépend de la méthode **setTextAlignment()** et en particulier, du paramètre qui lui est associé. Par défaut alignement en haut à gauche (TEXT_ALIGN_LEFT).

```
 uint16_t drawStringMaxWSSD1306 afficheurOLED(0x3C,21,22);idth(int16_t _x,int16_t
_y,int16_t _maxLineWidth,
                              const String &_text);
```

Paramètres :

_x : abscisse du point d'origine.
_y : ordonnée du point d'origine.
_maxlineWidth : Longueur maxi de la chaîne exprimée en pixels.
_text : texte à afficher sous la forme d'une String (chaîne de caractères).

Rôle :
Dessine une chaîne de caractères avec une largeur maximale à l'emplacement donné. Si, la chaîne de caractères donnée est plus large que la largeur spécifiée, le texte sera enveloppé à la ligne suivante à un espace ou un tiret. La méthode renvoie 0 si tout s'adapte à l'écran ou retourne le nombre de caractères dans la première ligne sinon.

```
void setTextAlignment(OLEDDISPLAY_TEXT_ALIGNMENT _textAlignement);
```

Paramètres :

_textAlignement : mode d'alignement

Rôle : Spécifie par rapport à quel point d'ancrage le texte est rendu.
Constantes disponibles : TEXT_ALIGN_LEFT (alignement à gauche), TEXT_ALIGN_CENTER (alignement au centre), TEXT_ALIGN_RIGHT (alignement à droite), TEXT_ALIGN_CENTER_BOTH (centré sur les deux axes).

Deux méthodes permettent de connaître la largeur exprimée en pixels par une chaîne de caractères en fonction de la fonte choisie avec la méthode **setFont() ;**

```
uint16_t getStringWidth(const String &_text);
uint16_t getStringWidth(const char *_text, uint16_t _length, bool _utf8 = false) ;
```

Paramètres :

_text : le texte dont on recherche la largeur (sous la forme d'une String ou d'un tableau de caractères)

pour la deuxième forme :

_length : nombre de caractères dans la chaîne
_utf8 : la chaîne est au format utf8 si le paramètre possède la valeur **true**. Si la valeur est **false** ou non précisée la chaîne n'utilise pas de caractère utf8.
Valeur de retour : largeur en pixels de la chaîne.

Rôle : retourne la largeur en pixels de la chaîne de caractères en fonction de la taille de la police sélectionnée.

Il existe également des méthodes chargées du remplissage des figures géométriques dessinées avec les méthodes **drawRect()**, **drawCircle()** et **drawTriangle()** vues précédemment :

```
void fillRect(int16_t _x,int16_t _y,int16_t _width,int16_t _height);
```

Paramètres :

_x : abscisse du point d'origine.
_y : ordonnée du point d'origine
_width : largeur du rectangle
_height : hauteur du rectangle

Rôle : Cette méthode remplit un rectangle.

```
void fillCircle(int16_t _x,int16_t _y,int16_t _radius);
```

Paramètres :

_x : abscisse du centre.
_y : ordonnée du centre.
_radius : rayon du cercle.

Rôle : Cette méthode remplit un cercle.

```
void fillTriangle(int16_t _x0,int16_t _y0,int16_t _x1,int16_t _y1,
                  int16_t _x2,int16_t _y2);
```

Paramètres :

_x0 : abscisse du point 0.
_y0 : ordonnée du point 0.
_x1 : abscisse du point 1.
_y1 : ordonnée du point 1.
_x2 : abscisse du point 2.
_y2 : ordonnée du point 2.

Rôle : Cette méthode remplit un triangle.

Voici deux méthodes pour travailler au niveau du pixel :

```
void setPixel(int16_t _x,int16_t _y);
```

Paramètres :

_x : abscisse du pixel.
_y : ordonnée du pixel.

Rôle : Dessine un pixel aux coordonnées indiquées.

```
void clearPixel(int16_t _x,int16_t _y);
```

Paramètres :

_x : abscisse du pixel.
_y : ordonnée du pixel.

Rôle : Efface un pixel aux coordonnées indiquées.

Pour que l'affichage soit effectif sur l'écran, il est indispensable d'appeler la méthode **display()** :

```
void display(void);
```

Paramètres :

Aucun

Rôle : Transfert le contenu du tampon mémoire sur l'écran

```
void clear(void);
```

Paramètres :

Aucun

Rôle : Efface le tampon mémoire courant.

```
void displayOn(void);
```

Paramètres :

Aucun

Rôle : Allume l'afficheur

```
void displayOff(void);
```

Paramètres :

Aucun

Rôle : Éteins l'afficheur

5.3.3. Exemple d'application

Ce programme initialise l'afficheur par rapport à la carte ESP32-SNIR trace un rectangle sur le pourtour de l'écran et affiche le message « Hello World ! » avec la police ArialMT_Plain_16 à la position 20,5.

```
#include <Arduino.h>
#include <SSD1306.h>

SSD1306 afficheurOLED(0x3C,21,22);
void setup()
{
    afficheurOLED.init();
    afficheurOLED.flipScreenVertically();
    afficheurOLED.setFont(ArialMT_Plain_16);
    afficheurOLED.drawRect(0,0,127,63);
    afficheurOLED.drawString(20,5,"Hello world !");
    afficheurOLED.display();
}
void loop()
{
}
```

Rappel : Sans la ligne **afficheurOLED.display()**, rien ne se produit sur l'écran.

AfficheurOLED représente l'instance de la classe **SSD1306**.

Les différentes méthodes de la classe sont disponibles en ajoutant un point derrière le nom de l'instance, suivi du nom de la méthode. Le fait de déclarer l'objet **afficheurOLED** en dehors des fonctions **setup()** et **loop()** le rend accessible dans les deux fonctions. On parle alors d'une variable globale. Elle est visible dans l'ensemble du fichier.

Les broches **SDA** GPIO21 et **SCL** GPIO22 du bus **I2C** sont déjà définies en incluant le fichier **Arduino.h** avec les bonnes valeurs. La déclaration peut également s'écrire :

```
SSD1306 afficheurOLED(ADD_OLED,SDA,SCL); // ADD_LED = 0x3C voir esp32_snir.h
```

5.4. Utilisation du capteur de température

Sur la carte ESP32-SNIR, on trouve un capteur de température DS18S20, un composant numérique fabriqué par Maxim Integrated, auparavant connu sous le nom de Dallas Semiconductor. L'ESP32 utilise le bus 1-Wire pour échanger des données avec ce capteur. Le bus 1-Wire, comme son nom l'indique, fonctionne avec un seul fil, tout en permettant la communication avec plusieurs capteurs connectés à ce même fil. L'ordre de grandeur est de 10 à 20 capteurs pour un même bus.

5.4.1. Librairies à installer

Le capteur nécessite l'utilisation de 2 classes pour fonctionner la première, **OneWire** est chargée de la gestion du bus et **DallasTemperature** prend en charge le capteur.

```
DallasTemperature @ 3.11.0 (required: milesburton/DallasTemperature)
|   └── OneWire @ 2.3.7 (required: paulstoffregen/OneWire @ ^2.3.5)
OneWire @ 2.3.7 (required: paulstoffregen/OneWire)
```

Pour la lecture d'une température, il est uniquement nécessaire d'instancier la classe OneWire en précisant le numéro de GPIO en charge du bus. Sur la carte ESP32-SNIR, le bus 1-Wire est relié à la broche GPIO18.

```
OneWire(uint_8 _pin);
```

Paramètres :

_pin : broche utilisée pour créer un bus 1-Wire

Rôle : constructeur de la classe **OneWire**, il paramètre la broche pour qu'elle supporte le bus et initialise les éléments utiles à la communication.

D'autres méthodes sont bien entendu présentes dans la classe **OneWire**, mais ne sont pas utiles directement pour obtenir la température lue par le capteur. Il n'en sera donc pas fait mention dans ce document.

5.4.2. Description des principales méthodes de la classe DallasTemperature

La classe **DallasTemperature**, elle aussi contient de nombreuses méthodes pour communiquer avec le capteur. Seules les méthodes utiles dans le cadre de ce document feront l'objet d'une description.

```
DallasTemperature(OneWire *_oneWire);
```

Paramètres :

_**oneWire :** adresse de l'instance responsable du bus 1-Wire.

Rôle : constructeur de la classe **OneWire**, il paramètre la broche pour qu'elle soit utilisée comme bus de communication et initialise les éléments nécessaires à cette communication.

```
void begin(void);
```

Paramètres :

aucun

Rôle : Initialise le ou les capteurs sur le bus.

```
void requestTemperatures(void);
```

Paramètres :

aucun

Rôle : Envoie la commande pour la mesure de température à tous les capteurs sur le bus 1-Wire.

```
uint8_t getDS18Count(void);
```

Paramètres :

aucun

Rôle : Renvoie le nombre de capteurs du type DS18xxx présents sur le bus.

```
float getTempCByIndex(uint8_t _index);
float getTempFByIndex(uint8_t _index);
```

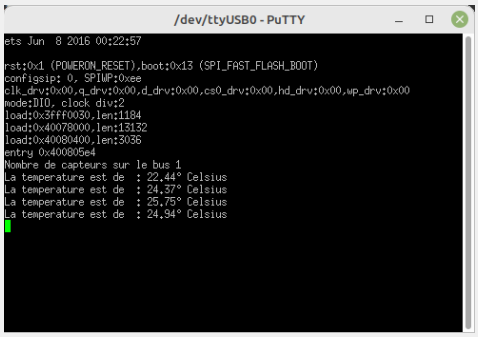
Paramètres :

_**index :** numéro du capteur sur le bus, 0 pour le premier.

Rôle : Renvoie la température obtenue par le capteur dont l'index est passé en paramètre pour la première méthode en degrés Celsius pour la deuxième en degrés Fahrenheit.

5.4.3. Exemple d'application

```
#include <Arduino.h>
#include <OneWire.h>
#include <DallasTemperature.h>

#include "esp32_snir.h"

OneWire oneWire(TEMP);
DallasTemperature capteurTemp(&oneWire);
void setup()
{
    Serial.begin(115200);
    capteurTemp.begin();
    uint8_t nbCapteurs = capteurTemp.getDS18Count();
    Serial.printf("Nombre de capteurs sur le bus %d\n\r",nbCapteurs);
}
void loop()
{
    capteurTemp.requestTemperatures();
    float temperature = capteurTemp.getTempCByIndex(0);
    Serial.print("La température est de  : ");
    Serial.print(temperature);
    Serial.println("° Celsius");
    delay(1000);
}
```



Le programme dans un premier temps initialise le capteur de température, puis chaque minute demande la mesure de la température et l'affiche en degrés Celsius.

5.5. Utilisation du clavier matricé, 12 touches

La carte ESP32-SNIR est équipée d'un clavier matricé 12 touches, soit modèle A (connecteur en bas sous le * 0 #), soit modèle B (connecteur en haut sous 1 2 3).

Les broches utilisées pour les lignes sont :

	L0	L1	L2	L3
Modèle A	GPIO32	GPIO33	GPIO25	GPIO26
Modèle B	GPIO26	GPIO25	GPIO33	GPIO32

Les broches utilisées pour les colonnes sont :

	C0	C1	C2
Modèle A	GPIO4	GPIO5	GPIO15
Modèle B	GPIO15	GPIO5	GPIO4

5.5.1. Librairie à installer

La librairie **Keypad** est utilisée pour gérer ce type de clavier. Si elle n'est pas installée, il est nécessaire de le faire.

```
Keypad @ 3.1.1 (required: chris--a/Keypad)
```

5.5.2. Description des principales méthodes de la classe Keypad

```
Keypad(char *_userKeymap, byte *_row, byte *_col, byte _numRows, byte _numCols);
```

Paramètres :

_ **userKeymap :** Tableau contenant le nom des touches du clavier
_ **row :** Tableau contenant le numéro de GPIO pour les lignes
_ **col :** Tableau contenant le numéro de GPIO pour les colonnes
_numRows : nombre de lignes
_numCols : nombre de colonnes

Rôle : Constructeur de la classe Keypad, initialise les broches pour la lecture du clavier et définit les touches du clavier ainsi que le câblage des lignes et de colonnes.

```
char getKey();
```

Paramètres :

Valeur de retour : Caractère correspondant à la touche enfoncée. **NO_KEY** si aucune touche n'est enfoncée.

Rôle : Scrute le clavier pour savoir si une touche est enfoncée. La méthode n'est pas bloquante et renvoie la constante **NO_KEY** si aucune touche n'est enfoncée.

```
char waitForKey();
```

Paramètres :

Valeur de retour : Caractère correspondant à la touche enfoncée.

Rôle : méthode bloquante, scrute le clavier jusqu'à ce qu'une touche soit enfoncée.

```
bool getKeys();
```

Paramètres :

Valeur de retour : true si une ou plusieurs touches sont enfoncées

Rôle : Scrute le clavier pour savoir si une ou plusieurs touches sont enfoncées. Met à jour la liste des touches enfoncées.

```
bool isPressed(char _keyChar);
```

Paramètres :

_ **keyChar :** touche dont on souhaite savoir si elle est enfoncée.
Valeur de retour : true si la touche est enfoncée, **fasle** sinon

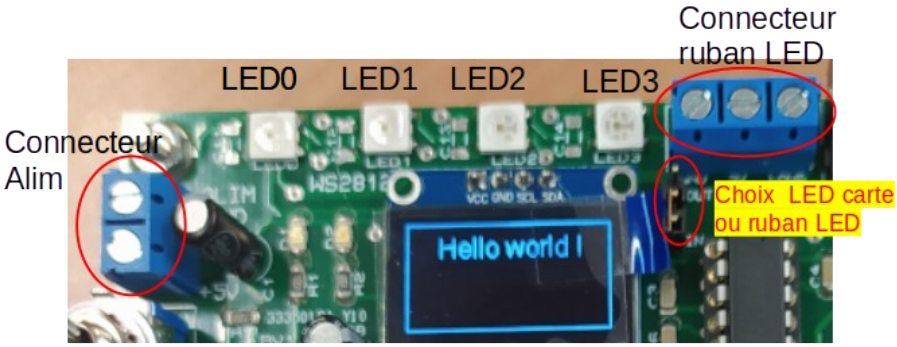
Rôle : Cette méthode vérifie si la touche passée en paramètre fait partie de la liste des touches enfoncées. La méthode **getKeys()** doit être appelée avant.

5.5.3. Exemple d'application

```
#include <Arduino.h>
#include <Keypad.h>

static const uint8_t LIGNES = 4;
static const uint8_t COLONNES = 3;

char touches[LIGNES * COLONNES] = {
    '1', '2', '3',
    '4', '5', '6',
    '7', '8', '9',
    '*', '0', '#'
    };

//affectation pour un connecteur en bas
//affectation des E/S GPIO aux lignes L0,L1,L2,L3 du clavier
uint8_t brochesEnLigne[LIGNES] = {32, 33, 25, 26};
//affectation des E/S GPIO aux colonnes C0,C1,C2 du clavier
uint8_t brochesEnColonne[COLONNES] = {4, 5, 15};

Keypad clavier(touches,brochesEnLigne,brochesEnColonne,LIGNES,COLONNES);

void setup()
{
    Serial.begin(115200);
}

void loop()
{
    char touche = clavier.getKey();
    if( touche != NO_KEY)
    {
        Serial.println(touche);
    }
}
```

5.6. Utilisation des LED RGB

Les LED RGB, pour ***Light Emitting Diode - Red Green Blue***, sont des diodes électroluminescentes capables de produire une gamme de couleurs en combinant différentes intensités de lumière rouge, verte et bleue. Ces LED sont constituées de trois LED individuelles, une rouge, une verte et une bleue intégrées dans un seul boîtier. La carte ESP32-SNIR est dotée de 4 de ces LED. Un connecteur externe permet de relier un ruban de LED RGB. Dans ce cas, il est nécessaire de prévoir une alimentation supplémentaire et modifier la position du cavalier.



Les LED RGB utilisent la librairie :

```
Adafruit NeoPixel @ 1.12.0 (required: adafruit/Adafruit NeoPixel)
```

5.6.1. Description des principales méthodes de la classe Adafruit_NeoPixel

Il existe plusieurs types de LED RGB, il est donc nécessaire de préciser l'ordre des couleurs au niveau du constructeur, sachant que les autres méthodes utilisent toujours l'ordre rouge, vert , bleu. Le dernier paramètre sera à fixer en fonction de la carte ESP32_SNIR que vous utilisez.

```
Adafruit_NeoPixel(uint16_t _nbLed, uint16_t _pin, neoPixelType _type);
```

Paramètres :

_ **nbLed :** nombre de LED à commander.
_ **pin :** numéro de broche GPIO pour transmettre les instructions de couleur ou de luminosité aux LED.
_ **neoPixelType :** type de LED parmi NEO_RGB, NEO_GBR, NEO_BRG, il est également possible de combiner avec la valeur NEO_KHZ400 ou NEO_KHZ800 qui représente la fréquence de transmission exprimée en kHz par l'intermédiaire d'un OU binaire.

Rôle : Constructeur de la classe Adafruit_NeoPixel définit la broche pour le pilotage des LED ainsi que le type de LED RGB en fonction du nombre de LED disponibles.

```
void begin(void);
```

Paramètres :

aucun

Rôle : Initialise la broche pour la transmission de données.

```
void setPixelColor(uint16_t _numLed, uint8_t _red, uint8_t _green, uint8_t _blue);
```

Paramètres :

_ **numLed :** numéro de la LED à piloter.
_ **red :** valeur de la couleur rouge [0 -255]
_ **green :** valeur de la couleur rouge [0 -255]
_ **blue :** valeur de la couleur rouge [0 -255]

Rôle : sélectionne une couleur en utilisant les composantes séparées rouge, verte et bleue pour une LED donnée en paramètre. Les 3 valeurs à 0 entrainent l'extinction du pixel. La méthode **show()** doit être appelée ensuite.

```
void setPixelColor(uint16_t _numLed, uint32_t _couleur);
```

Paramètres :

_ **numLed :** numéro de la LED à piloter.
_ **couleur :** valeur de la couleur sur 32 bits xxrrggbb. Le premier octet est ignoré. Les octets suivants représentent la valeur des couleurs sur 8 bits

Rôle : sélectionne une couleur en utilisant les composantes rouge, verte et bleue dans un même mot. La méthode **show()** doit être appelée ensuite.

```
uint32_t getPixelColor(uint16_t _numLed);
```
Paramètres :

_ **numLed :** numéro de la LED à piloter.

Valeur de retour : Couleur du pixel dont le numéro est passé en paramètre sur 32bits xxrrggbb.

Rôle : Permet de connaître la couleur d'un pixel.

```
void fill(uint32_t _couleur,uint16_t _first, uint16_t _count);
```
Paramètres :

_ **couleur :** valeur de la couleur sur 32 bits xxrrggbb. Le premier octet est ignoré. Les octets suivants représentent la valeur des couleurs sur 8 bits

_ **first :** numéro de la première LED

_ **count :** nombre de LED à allumer

Rôle : Sélectionne la couleur pour une série de LED. La méthode **show()** doit être appelée ensuite.

```
void clear(void);
```
Paramètres :

Aucun

Rôle : Cette méthode prépare l'extinction de toutes les LED. La méthode **show()** doit être appelée ensuite.

```
void show(void);
```
Paramètres :

Aucun

Rôle : Cette méthode rafraichit la couleur des LED. Elle est indispensable après tout changement de couleur.

D'autres méthodes permettent d'agir sur la brillance de chaque LED, d'autres encore permettent de piloter les LED en HSV pour Teinte (Hue en anglais), Saturation et Valeur , elles ne sont pas évoquées ici pour raison de simplification.

Dans le mode HSV la teinte représente la couleur elle-même de 0 à 360 degrés (360° le rouge, 240° le bleu et 120° le vert), la saturation indique l'intensité ou la pureté de la couleur, et la valeur contrôle sa luminosité.

5.6.2. Exemple d'application

L'application suivante allume progressivement chaque LED en rouge en faisant une pause de 500 ms entre chaque allumage. Lorsque les 4 LED sont allumées, il y a extinction du ruban et reprise du cycle.

```
#include <Arduino.h>
#include <Adafruit_NeoPixel.h>
#include "esp32_snir.h"

Adafruit_NeoPixel pixels(NB_PIXELS, DATALEDS, NEO_RGB);

void setup()
{
    pixels.begin();
}
void loop()
{
    pixels.clear();
    pixels.show();
    delay(500);
    for(int i = 0 ; i < NB_PIXELS ; i++)
    {
        pixels.setPixelColor(i,16,0,0);
        pixels.show();
        delay(500);
    }
}
```

Dans esp32_snir.h

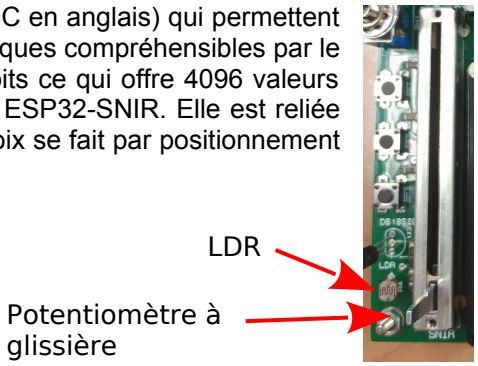
```
#define DATALEDS 19
#define NB_PIXELS 4
```

Vous pouvez essayer ce programme en modifiant les couleurs des LED en agissant sur les trois derniers paramètres de la méthode **setPixelColor()**.

5.7. Conversion analogique numérique

L'ESP32 dispose de convertisseurs analogiques numériques (CAN ou ADC en anglais) qui permettent de mesurer des signaux analogiques et de les convertir en valeurs numériques compréhensibles par le microcontrôleur. Le convertisseur utilisé possède une résolution de 12 bits ce qui offre 4096 valeurs possible [0..4095]. La broche **GPIO36** est dédié à cet usage sur la carte ESP32-SNIR. Elle est reliée soit au potentiomètre à glissière soit à la LDR captant la luminosité, le choix se fait par positionnement du cavalier J13 (ADC ou LDR).

Choix entre le potentiomètre (position ADC) et la LDR.



La fonction **analogRead()** de la bibliothèque Arduino permet de lire simplement les broches de l'ESP utilisées en tant qu'entrées analogiques. Il n'est pas nécessaire de configurer leur fonction par ailleurs.

```
int analogRead(uint8_t _pin);
```

Paramètres :

_pin : représente le numéro de la broche GPIOx.

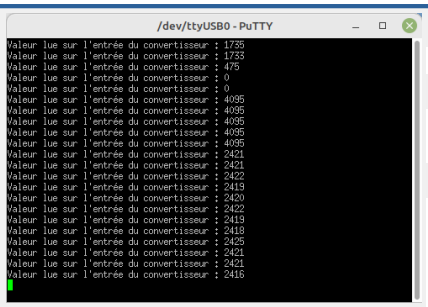
Valeur de retour : une valeur entre 0 et 4095, résultat de la conversion.

Rôle :
Cette fonction lit la grandeur analogique présente en entrée sur la broche indiquée en paramètre et retourne le résultat sous la forme d'un entier après la conversion sur 12 bits.

5.7.1. Exemple d'application

Voici un simple programme permettant de relever la position du potentiomètre :

```
#include <Arduino.h>
#include "esp32_snir.h"
void setup(){
  Serial.begin(115200);
}
void loop() {
  int valeurADC = analogRead(ADC_LDR);
  Serial.print("Valeur lue sur l'entrée du convertisseur : ");
  Serial.println(valeurADC);
  delay(1000);
  //La lecture de l'entrée analogique se fait ici toutes les secondes.
}
```



Dans esp32_snir.h

```
#define ADC_LDR 36
```

5.7.2. Mesure de l'intensité lumineuse

La LDR pour **Light Dependent Resistors**, également appelée résistances photo-dépendantes ou photocellules, sont des composants électroniques dont la résistance varie en fonction de l'intensité lumineuse qu'ils reçoivent. Pour étalonner une LDR, c'est-à-dire calibrer ses valeurs de résistance en fonction de la lumière, il est nécessaire de définir des seuils faible lumière, lumière ambiante, lumière directe, etc. Les valeurs obtenues peuvent être utilisées pour définir des plages d'utilisation en fonction des exigences précises de l'application.

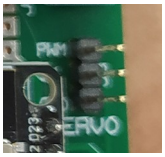
La LDR elle-même ne fournit pas directement une mesure en lux (unité de mesure de l'éclairement lumineux). La LDR est plutôt utilisée pour détecter les changements d'intensité lumineuse et convertir ces variations en changements de résistance électrique. Pour estimer la luminosité en lux à partir d'une LDR, il est nécessaire de faire une corrélation entre la résistance de la LDR et l'intensité lumineuse qu'elle reçoit en utilisant des capteurs de lumière calibrés pour fournir des mesures de référence.

Il est nécessaire de déplacer le cavalier (J13) de la carte ESP32-SNIR en **position LDR**, pour effectuer la mesure de l'intensité lumineuse.

5.8. La commande du servomoteur

Les servomoteurs sont des dispositifs utilisés pour contrôler avec précision la position d'un axe. Ils sont souvent utilisés dans les projets d'ingénierie, de robotique et de modélisme. La carte ESP32-SNIR dispose d'un connecteur (J3) dédié à ce type de moteur. Le signal de commande est réalisé sur la broche **GPIO2**

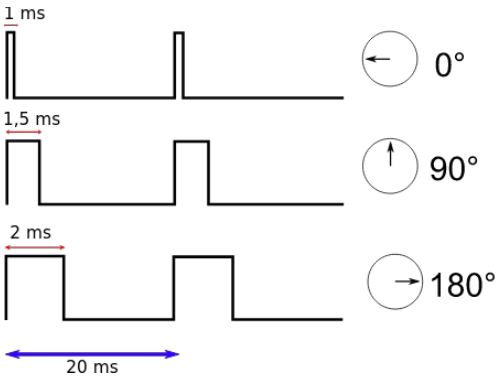
Signal PWM	orange	Commande moteur **GPIO2**
5V	rouge	Alimentation +
GND	marron	Alimentation -



Les servomoteurs fonctionnent sur la base d'un asservissement pour contrôler précisément la position de l'axe de sortie. Voici le principe de fonctionnement simplifié d'un servomoteur :

1. **Moteur électrique :** À l'intérieur du servomoteur se trouve un moteur électrique qui génère le mouvement. Ce moteur est souvent un moteur à courant continu avec des engrenages pour fournir un couple et une précision de mouvement accrus.
2. **Potentiomètre (capteur de position) :** C'est le cœur du système de rétroaction, il sert de capteur de position. Il est connecté à l'axe de sortie du servomoteur. Le potentiomètre fournit un retour sur la position actuelle de l'axe en mesurant la tension ou la résistance qui change en fonction de cette position.
3. **Circuit de commande et électronique interne :** Le servomoteur contient un circuit électronique qui compare le signal de commande reçu avec la position actuelle du potentiomètre. Ce circuit utilise cette information pour déterminer dans quelle direction et à quelle vitesse le moteur doit tourner pour atteindre la position souhaitée.
4. **Signal de commande (PWM) :** Les servomoteurs reçoivent des signaux de commande sous forme de signaux PWM (modulation de largeur d'impulsion). Ce signal contient des informations sur la position cible souhaitée pour le servomoteur.
5. **Boucle de rétroaction :** Le servomoteur ajuste la rotation de son axe jusqu'à ce que la position du potentiomètre corresponde à la position spécifiée par le signal de commande. La boucle de rétroaction est continue, ce qui signifie que le servomoteur ajuste constamment sa position pour correspondre à la commande reçue, assurant ainsi un contrôle précis de la position de sortie.

Ce système de boucle de rétroaction permet aux servomoteurs de maintenir des positions précises et de résister aux forces extérieures qui pourraient tenter de les déplacer de leur position souhaitée. Ils sont largement utilisés dans les applications où la précision de positionnement est essentielle, comme dans les bras robotisés, les drones, les systèmes de direction des modèles réduits, etc.



Ce type de servomoteur peut être contrôlé en utilisant un signal modulé en largeur d'impulsion (PWM) de 50 Hz, qui produit une impulsion toutes les 20ms. On peut ajuster la position du servomoteur en modifiant la durée des impulsions, pars exemple entre 1ms et 2ms. Ces valeurs sont liés au servomoteur et aux paramètres fournis lors de la configuration. Pour fonctionner, le servomoteur utilise la librairie :

```
ESP32Servo @ 1.1.1 (required: madhephaestus/ESP32Servo)
```

La classe **Servo** de cette bibliothèque est responsable du contrôle du servomoteur connecté au **GPIO 2** de la carte ESP32-SNIR et peut en gérer jusqu'à 16. Comme il n'y a qu'un seul port disponible, généralement, seule l'allocation du **timer 0** est nécessaire pour fabriquer le signal modulé en largeur d'impulsion. Cela est réalisé par la méthode **static** suivante de la classe ESP32PWM :

```
ESP32PWM::allocateTimer(0);
```

Le constructeur de cette classe ne nécessite pas de paramètre spécifique. Les attributs de la classe sont chargés avec des valeurs par défaut, incluant une période de rafraichissement de 50 Hz. Ainsi, il n'est généralement pas nécessaire de modifier cette valeur.

5.8.1. Description des principales méthodes de la classe Servo

```
int attach(int _pin, int _min, int _max);
```

Paramètres :

_pin : représente le numéro de la broche GPIOx.
_min : représente la largeur d'impulsion minimum (valeur supérieure ou égale à 500 µs)
_max : représente la largeur d'impulsion maximum (valeur inférieure ou égale à 2500 µs)
Valeur de retour : numéro du canal PWM utilisé par le servomoteur.

Rôle :
Cette méthode associe le générateur de signal PWM à la broche indiqué en paramètre fixe les largeurs d'impulsions minimales et maximale et retourne le numéro de **timer** à allouer à cette tâche.

```
void detach(void);
```

Paramètres :

Aucun

Rôle :

Libère la ressource **timer** pour un autre usage éventuel. Cette méthode est utile si le servomoteur n'est plus utilisé dans l'application.

```
void write(int _value);
```

Paramètres :

_value : détermine l'angle à obtenir. Si la valeur est inférieure à la largeur minimale d'impulsion, la valeur est exprimée en degré [0..180] sinon elle représente la largeur d'impulsion en µs.

Rôle :

Permet de définir la largeur d'impulsion. Pour exprimer l'angle en degré, il est nécessaire que les valeurs mini et maxi de la largeur d'impulsion soit correctement définies avec la méthode **attach()**.

```
int read(void);
```

Paramètres :

Valeur de retour : position du moteur en degrés [0..180]

Rôle :

Permet d'obtenir la position du moteur exprimé en degrés.

5.8.2. Exemple d'application

L'exemple suivant est réalisé avec un servomoteur type : **Mystery 3.7G mini servo**, la largeur d'impulsion minimale est de 700 µs et sa valeur maximale est de 2500 µs.

```
#include <Arduino.h>
#include <ESP32Servo.h>

Servo leServoMoteur;

void setup()
{
    int canal = leServoMoteur.attach(PWM,700, 2500);
    ESP32PWM::allocateTimer(canal);
    Serial.begin(115200);
}
```

Dans esp32_snir.h

```
#define PWM 2
```

La fonction **setup()** initialise la sortie pour piloter le servomoteur. Les valeurs mini et maxi de largeur d'impulsion sont à définir en fonction du servomoteur. Le moteur ne doit pas produire de bruit pour les positions 0° et 180°. Elle initialise le port console pour définir les positions à l'aide de **Putty** par exemple. Comme le montre la fonction **loop()** à la page suivante.

```
void loop()
{
    char carLu;
    if(Serial.available()>0)
    {
        carLu = Serial.read();
        Serial.println(carLu);
        switch(carLu)
        {
            case '1':
            leServoMoteur.write(0);
            break;
            case '2':
            leServoMoteur.write(45);
            break;
            case '3':
            leServoMoteur.write(90);
            break;
            case '4':
            leServoMoteur.write(135);
            break;
            case '5':
            leServoMoteur.write(180);
            break;
        }
    }
}
```

6. Classes et fonctions de l'environnement Arduino

Ce chapitre présente des fonctions et des classes propres à l'environnement Arduino, qui facilite la programmation de l'ESP32 et qui ne font pas partie des standards du C ou du C++.

6.1. La fonction map()

La fonction **map()** permet de transférer une valeur d'une plage vers une nouvelle autre plage de valeur spécifiée par les paramètres qui lui sont transmis. Par exemple, si la fonction reçoit une valeur comprise entre 0 et 4095 et que l'on souhaite obtenir une valeur entre 0 et 255 proportionnelle à cette valeur reçue, elle réalise cette opération. Il est important de noter que la fonction ne travaille qu'avec des nombres entiers. La partie fractionnaire du nombre obtenu est simplement ignorée, et aucune opération d'arrondi au sens mathématique n'est effectuée sur la partie entière.

```
long map(const long _valeur,const long _deMin,const long _deMax,const long _versMin,const long _versMax) ;
```

Paramètres :

_ **valeur :** valeur à transférer
_ **deMin :** limite inférieure de la plage de départ
_ **deMax :** limite supérieure de la plage de départ
_ **versMin :** limite inférieure de la plage d'arrivée
_ **versMax :** limite supérieure de la plage d'arrivée
Valeur de retour : valeur obtenue dans la nouvelle plage

Rôle :

Convertit **_valeur** allant d'une plage [**_deMin** .. **_deMax**] dans la plage de valeur [**_versMin** .. **_versMax**] proportionnellement.

6.2. La fonction delay()

La fonction **delay()** permet de mettre en pause l'exécution du programme en cours. Il reprend une fois le temps spécifié écoulé. Cependant, pendant cette pause, le microcontrôleur ne peut effectuer aucune autre tâche, ce qui peut limiter la réactivité globale du programme, surtout dans les situations nécessitant un traitement continu ou des actions simultanées.

```
void delay(unsigned long _tempo) ;
```

Paramètres :

_tempo : nombre de millisecondes à attendre

Rôle :

Suspend le programme pendant un temps exprimé en millisecondes (1000 ms = 1s)

6.3. La fonction millis()

La fonction **millis()** est extrêmement utile dans les situations où le programme ne peut pas être mis en pause à l'aide de la fonction **delay()**. En comparant les résultats de deux appels distincts à cette fonction, on obtient une mesure de temps. Il est possible de mesurer des durées allant jusqu'à environ 50 jours avant que la valeur ne dépasse sa capacité et ne revienne à zéro.

```
unsigned long millis() ;
```

Paramètres :

Valeur de retour : nombre de millisecondes écoulées depuis la mise en route de l'ESP32.

Rôle :

Renvoie le nombre de millisecondes écoulées depuis le démarrage de la carte Arduino avec le programme actuel.

6.4. La classe String

Les chaînes de caractères peuvent toujours être gérées à la manière du C avec un tableau de caractères avec une marque de fin de chaîne, le caractère '\0'. Il existe cependant une classe **String** qui facilite la programmation.

La classe **String** est présente dans un environnement C++ classique gcc sous Linux ou autre. Cependant son fonctionnement est quelque peu différent dans l'environnement Esp32 – Arduino.

L'un des avantages de la classe **String** est qu'elle s'adapte automatiquement à la taille nécessaire de la chaîne, évitant ainsi la nécessité de se préoccuper de sa dimension. Cela permet une gestion flexible des données sans contraintes de taille fixe, s'ajustant dynamiquement en fonction des besoins de l'application.

Chaque caractère reste accessible individuellement à l'aide de l'**opérateur [ ]**, fonctionnant comme un accès à un tableau. L'affectation se réalise aisément grâce à l'**opérateur =**, à la différence avec un tableau de caractères où le contenu de l'objet String est automatiquement dupliqué. De plus, les **opérateurs +** ou **+=** autorisent la concaténation simple et efficace de chaînes de caractères.

Cette classe possède un certain nombre de **constructeurs surchargés** ce qui permet de transformer tous les types courants en chaîne de caractères sous la forme d'une instance de la classe **String**.

```
String(const char *_cstr = "");
String(const char *_cstr, unsigned int _length);
```

Paramètres :

_cstr : chaîne de caractères sous la forme d'un tableau, terminé par '\0' pour la première forme.

_length : indication du nombre de caractères pour la seconde forme.

Rôle :

Construit un objet de type String à partir d'un tableau de caractères.

```
String(const String &_str);
```

Paramètres :

_str : autre objet de type String.

Rôle :

Construit un objet de type String à partir d'un autre objet de même type, constructeur de recopie.

```
String(char _c);
```

Paramètres :

_c : Un caractères.

Rôle :

Construit un objet de type String à partir d'un caractère.

```
String(unsigned char _c, unsigned char _base = 10);
String(int _val, unsigned char _base = 10);
String(unsigned int _val, unsigned char _base = 10);
String(long _val, unsigned char _base = 10);
String(unsigned long _val, unsigned char _base = 10);
```

Paramètres :

_c : entier ou non signé sur 8 bits

ou **_val :** entier ou non signé

_base : précise la base DEC, HEX, OCT

Rôle :

Transforme toutes les formes d'entiers signés ou pas en objet String,

```
String(float _val, unsigned char _decimal = 2);
String(double _val,unsigned char _decimal = 2);
```

Paramètres :

_val : réel simple ou double précision

_decimal : précise le nombre de chiffres après la virgule

Rôle :

Transforme un réel en objet String,

Les principaux opérateurs de comparaison ont été également surchargés pour cette classe ainsi :

opérateur	Description	Exemple String S1, S2
= =	Comparaison, les deux chaînes sont-elles identiques ?	s1 == s2
>	La chaîne s1 est-elle après la chaîne s2 dans l'ordre lexicographique	s1 > s2
>=	La chaîne s1 est-elle après la chaîne s2 dans l'ordre lexicographique ou identique	s1 >= s2
<	La chaîne s1 est-elle avant la chaîne s2 dans l'ordre lexicographique	s1 < s2
<=	La chaîne s1 est-elle avant la chaîne s2 dans l'ordre lexicographique ou identique	s1 <= s2
!=	Comparaison, les deux chaînes sont-elle différentes ?	s1 != s2

Des méthodes permettent la conversion de **String** en entier ou en réel si cela est possible dans les 3 cas la méthode retourne la valeur 0 si c'est impossible.

toInt()	Conversion en entier
toFloat()	Conversion en réel simple précision
toDouble()	Conversion en réel double précision

```
String s2(152);
int val2 = s2.toInt();
```

Il est également possible de forcer la case de la police de caractères soit en majuscules soit en minuscules.

toLowerCase()	Conversion de la chaîne en minuscules
toUpperCase()	Conversion de la chaîne en majuscules

```
String s1("Bonjour");
s1.toUpperCase();
```

La méthode suivante permet de retrouver une chaîne de caractère classique du C

```
toCharArray(char *_buf, const unsigned int _taille);
```

Paramètres :

_buf : tableau de caractères

_taille : indication du nombre de caractères maxi que peut contenir le tableau.

Rôle :

Recopie une le contenu d'un objet **String** dans un tableau de caractères sans dépacer la dimension imposée par le deuxième paramètre.

Les méthodes suivantes permettent de vérifier la présence d'une sous chaîne de caractères dans un objet String.

```
bool startsWith(const String &_s);
bool endsWith(const String &_s);
```

Paramètres :

_s : chaîne recherchée

valeur de retour : true si l'objet String commence pour la première ou se termine par la deuxième par la la chaîne recherchée, **false** sinon.

Rôle :

Recopie une le contenu d'un objet String dans un tableau de caractères sans dépacer la dimension imposée par le deuxième paramètre.

Les méthodes suivantes remplace un caractère ou une chaîne de caractères par un/une autre.

```
void replace(char _find, char _replace);
void replace(const String &_find, const String &_replace);
```

Paramètres :

_find : caractère ou objet **String** recherché

_replace : caractère ou objet **String** de remplacement

Rôle :

Recherche le caractère ou la chaîne de caractères dans l'objet String et le remplace par un nouveau caractère ou une nouvelle chaîne de caractères.

Voici quelques méthodes qui permettent de vérifier la position d'un caractère ou d'une chaîne de caractère dans l'objet String.

```
int indexOf(const char _val);
int indexOf(const String &_val);
int indexOf(const char _val, const unsigned int _position);
int indexOf(const String &_val, const unsigned int _position);

int lastIndexOf(const char _val);
int lastIndexOf(const String &_val);
int lastIndexOf(const char _val, const unsigned int _position);
int lastIndexOf(const String &_val, const unsigned int _position);
```

Paramètres :

_val : caractères ou chaîne de caractère recherché

_position : position à partir de laquelle la recherche est effectuée

valeur de retour : index dans la chaîne ou l'élément a été trouvé, ou -1 si l'élément n'est pas présent.

Rôle :

Ces méthodes permettent de rechercher un caractère ou une chaîne de caractères dans l'objet **String** existant. **indexOf** trouve la première occurrence à partir du début de la chaîne ou d'une position spécifiée, tandis que **lastIndexOf** trouve la dernière occurrence, également à partir du début ou d'une position déterminée.

De même, il est possible supprimer des caractères :

```
void remove(unsigned int _index);
void remove(unsigned int _index, unsigned int _nombre);
```

Paramètres :

_index : index à partir duquel les caractères doivent être supprimés

_nombre : nombre de caractères à supprimer

Rôle :

Ces méthodes permettent de supprimer des caractères dans un objet String à partir d'une position donnée par l'index, en allant jusqu'à la fin si aucun nombre n'est spécifié, ou en supprimant uniquement le nombre demandé s'il est précisé.

```
void trim(void);
```

Paramètres :

Aucun

Rôle :

Supprime tous les espaces et tabulations au début de la chaîne contenue dans l'objet String.

Il est bien sûr également possible de connaître le nombre de caractères contenus dans l'objet String.

```
unsigned int length(void);
```

Paramètres :

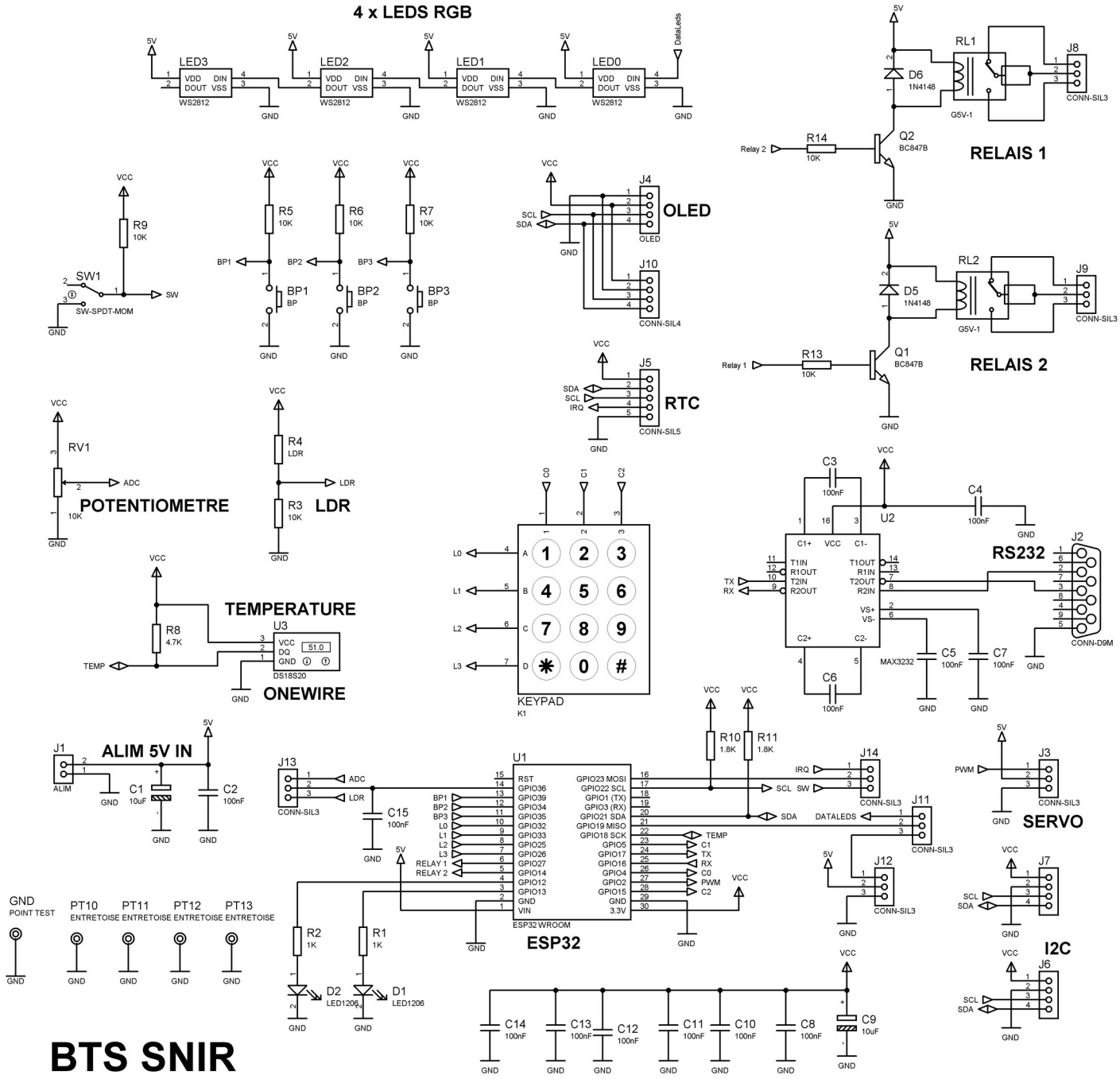
valeur de retour : la longueur de la chaîne

Rôle :

Fournit la longueur de la chaîne contenue dans l'objet String.

Annexes

A. Schéma de principe de la carte ESP32-SNIR



B. Résumé des commandes de PlatformIO

Création du projet

```
pio project init --ide netbeans --board lolin32
```

Initialisation du projet dans netBeans

```
pio run --target upload
```

Liste des cartes « boards » disponibles

```
pio boards [nom d'une famille de cartes]
```

Recherche des librairies disponibles pour installation

```
pio pkg search [Mot clé]
```

Liste des librairies installées pour tous les projets

```
pio pkg list -g
```

Installation d'une nouvelle des librairies pour tous les projets

```
pio pkg install -g -l "Nom de la librairie"
```

Remarques :

La commande **`pio`** peut être remplacée par la commande **`platformio`**.
Les options **`-g`** et **`-l`** peuvent être remplacées par **`--global`** ou **`--library`**.

C. Librairies à installer pour la carte ESP32-SNIR

Voici la liste des librairies supplémentaires à installer pour les composants présent sur la carte.

```
Platforms
└── espressif32 @ 6.0.0 (required: platformio/espressif32)

Tools
├── framework-arduinoespressif32 @ 3.20006.221224 (required: platformio/framework-
    arduinoespressif32)
├── tool-esptoolpy @ 1.40400.0 (required: platformio/tool-esptoolpy)
├── tool-mkfatfs @ 2.0.1 (required: platformio/tool-mkfatfs)
├── tool-mklittlefs @ 1.203.210628 (required: platformio/tool-mklittlefs)
├── tool-mkspiffs @ 2.230.0 (required: platformio/tool-mkspiffs)
├── tool-scons @ 4.40502.0 (required: platformio/tool-scons)
└── toolchain-xtensa-esp32 @ 8.4.0+2021r2-patch5 (required: espressif/toolchain-
xtensa-esp32)

Libraries
├── Adafruit NeoPixel @ 1.12.0 (required: adafruit/Adafruit NeoPixel)
├── DallasTemperature @ 3.11.0 (required: milesburton/DallasTemperature)
│   └── OneWire @ 2.3.7 (required: paulstoffregen/OneWire @ ^2.3.5)
├── ESP32Servo @ 1.1.1 (required: madhephaestus/ESP32Servo)
├── ESP8266 and ESP32 OLED driver for SSD1306 displays @ 4.3.0 (required:
    thingpulse/ESP8266 and ESP32 OLED driver for SSD1306 displays)
├── Keypad @ 3.1.1 (required: chris--a/Keypad)
```

L'ensemble des exemples présents dans ce document sont fonctionnels dans le contexte décrit ci-dessus. Attention lors de l'installation, il est obligatoire d'encadrer le nom de librairie par des guillemets, si elle contient des espaces.

D. Contenu du fichier esp32_snir.h

```
/*
 * File:   esp32_snir.h
 * Author: Philippe Cruchet
 *
 * Created on 29 novembre 2023, 13:56
 */

#ifndef ESP32_SNIR_H
#define ESP32_SNIR_H

#define BP1 39 // définition des boutons-poussoirs
#define BP2 34
#define BP3 35
#define SW 23  // définition de l'interrupteur

#define LED 2 // définition de la LED bleue sur le board ESP32
#define D1 13 // définition de la LED rouge
#define D2 12 // définition de la LED verte

#define ADD_OLED 0x3C // adresse de l'afficheur OLED sur le bus I2C

#define DATALEDS 19 // définition du GPIO pour la commande des LED RGB
#define NB_PIXELS 4 // nombre de pixels sur la carte.

#define ADC_LDR 36 // définition du GPIO pour la conversion analogique numérique

#define RELAY_1 27 // définition du GPIO pour le relais 1
#define RELAY_2 14 // définition du GPIO pour le relais 2

#define TX_RS232 17  // définition de la broche de transmission pour la RS232
#define RX_RS232 16  // définition de la broche de réception pour la RS232

#define TEMP 18 // définition du bus 1-wire pour le capteur de température

#define PWM 2 // définition du GPIO pour le servomoteur

#endif /* ESP32_SNIR_H */
```