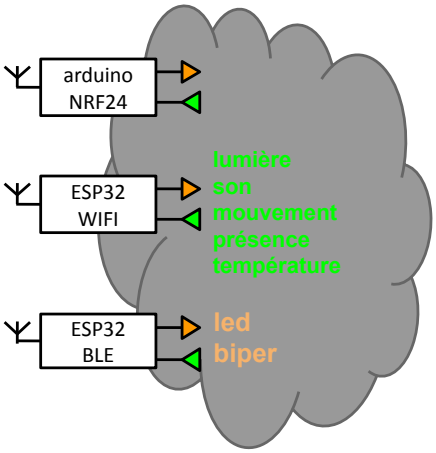
ESP32
programmation multi-tâches

Module IOC — MU4IN109

Franck Wajsbürt

Problème

Les modules permettant de faire des mesures ou d'agir sur l'environnement ont peu de mémoire et une faible puissance de calcul, mais ils doivent gérer un grand nombre d'événements asynchrones et potentiellement un grand nombre d'actions périodiques avec les contraintes du temps réel.



Il n'est pas raisonnable*, voire il est souvent impossible*, d'installer un système d'exploitation proposant naturellement la programmation multi-threads. Pour autant, la programmation multi-threads (multi-tâches) est nécessaire pour avoir des programmes évolutifs.

Dans cette séance, nous allons voir :

1. une présentation du SoC ESP32 (micro-contrôleur évolué)
2. une proposition de programmation multi-tâches sans OS

** Pourquoi ?*

ESP 32

Qu'est ce que l'ESP32

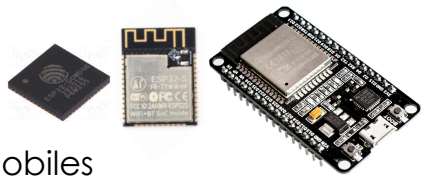
C'est un SoC (micro-contrôleur)

- Conçu par Espressif une société fabless chinoise (Shanghai)
- Fabriqué par TSMC en 40 nm
- Processor Xtensa dual-core 32-bit LX6 (ou single core)
- 160 MHz ou 240 MHz *(les modules des TME sont à 80 Mhz)*
- 512 kiB SRAM + 448 kiB ROM + 0 à 4 MiB Flash
- Fonctionnel entre -40°C et +125°C
- Ultra-basse consommation (5µA deep sleep)
- Intégrant des transceivers WIFI et Bluetooth
- Intégrant des accélérateurs de cryptographie
- Bon marché (qqs €)
- Destiné au marché IoT
- Site : https://en.wikipedia.org/wiki/ESP32 & http://esp32.net

Espressif

- Espressif Systems société chinoise fabless créée en 2008
  Composants IoT WiFi et Bluetooth, low power, sécurisé et robuste.
  https://www.espressif.com/en/company/about-us/who-we-are
- Conception
  - SoC : ESP8266 et ESP32
  - Module : SoC + flash + antenne + devices
  - Board : SoC | Module + devices
  - Logiciel : SDK, projets open-sources, app mobiles

  https://www.espressif.com/en/company/about-us/what-we-do



- Dates clés
  - 2008 : création de Espressif Systems
  - 2013 : premier circuit ESP8089 transceiver WIFI
  - 2014 : premier SoC ESP8266EX WiFi
  - 2016 : premier SoC ESP32 WiFi + BT
  - 2018 : Plus de 100 Millions de chip vendus.

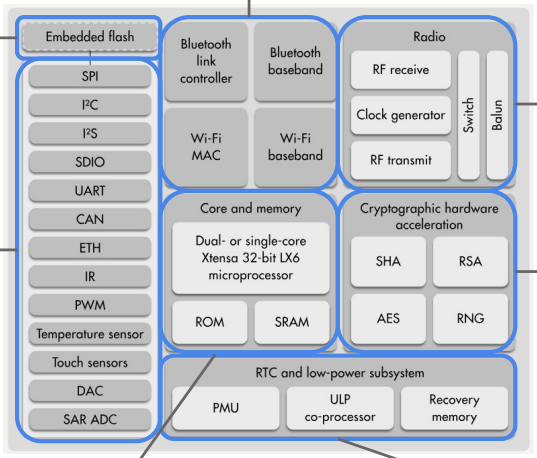
  https://www.espressif.com/en/company/about-us/milestones

Contenu d'un ESP32

flash interne
0, 2MiB ou 4 MiB
La mémoire externe est interfacée par cache (16 kiB ?)

Device Controller

couches physiques
WiFi et BT



Modules Radio
WIFI, BT et BLE

- **Wi-Fi:** 802.11 b/g/n/e/i (802.11n @ 2.4 GHz up to 150 Mbit/s)
- **Bluetooth:** v4.2 BR/EDR and Bluetooth Low Energy (BLE)

Security

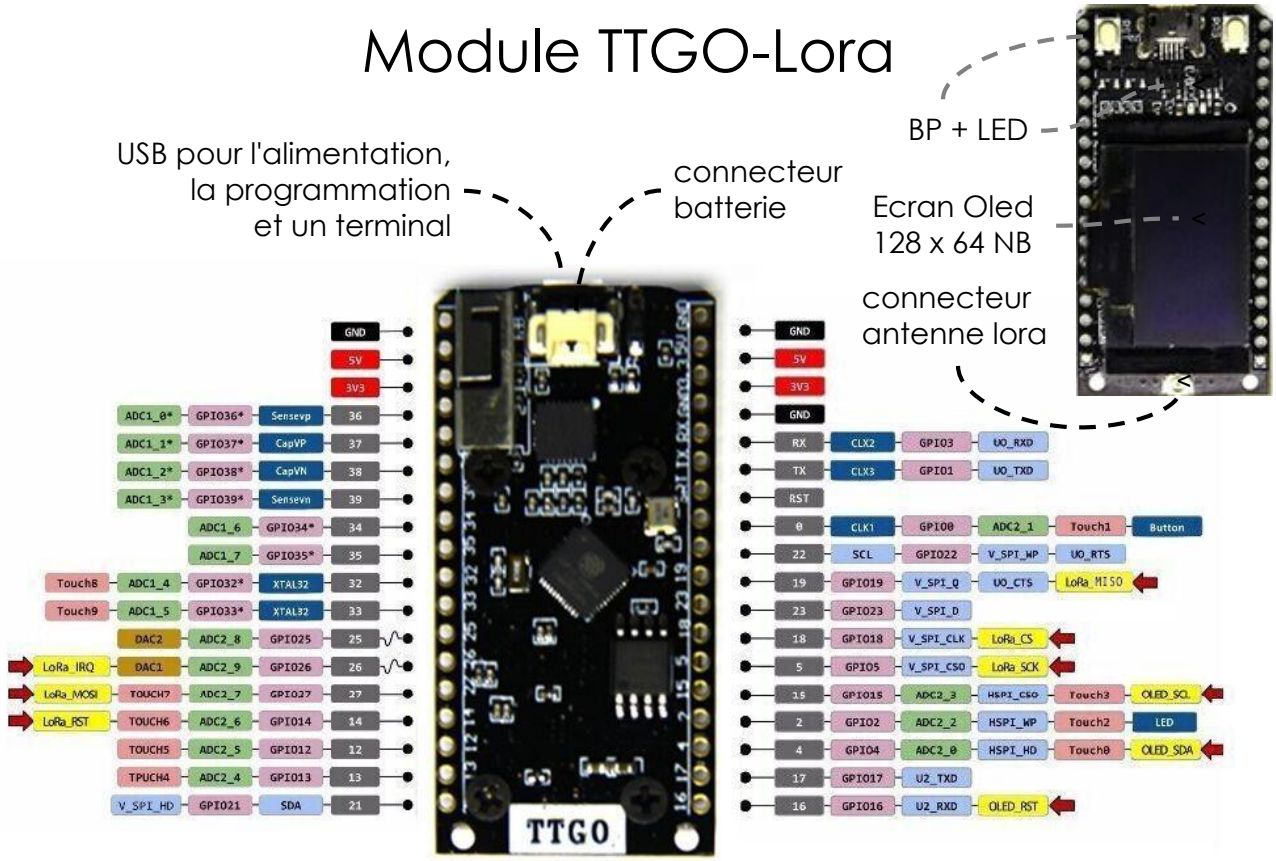
AES, SHA-2, RSA,
elliptic curve cryptography (ECC),
random,number generator (RNG)

Single ou dual core
Rom (boot) + RAM

Processeur et RAM
en Deep Sleep

http://esp32.net/

Module TTGO-Lora



https://primalcortex.wordpress.com/2017/11/24/the-esp32-oled-lora-ttgo-lora32-board-and-connecting-it-to-ttn/

Caractéristiques techniques

- Microprocesseur dual core à 240 MHz (en TP 80MHz)
- 16 MiB de mémoire flash
- Connectivité
  - WiFi 802.11 b/g/n conforme à la norme IEEE 802.11 compatible avec les sécurités WFA, WPA/WPA2 et WAPI
  - Bluetooth 4.0 LE et BR/EDR
  - Lora 433MHz (SX1278) (*long-range wide-area network*)
- Entrées/Sorties (48 sur le chip mais moins sur le module)
  - 26x E/S numériques (3.3V)
  - 12x entrées analogiques (SAR - Successive Approximation Register)
  - 4× SPI, 2× I²S*, 2× I²C, 3× UART, CAN 2.0, IR, Touch Sensor
- Capteur de température
- Cryptographie :
  - AES, SHA-2, RSA, ECC, random number generator (RNG)

* *Integrated Interchip Sound*

Programmation

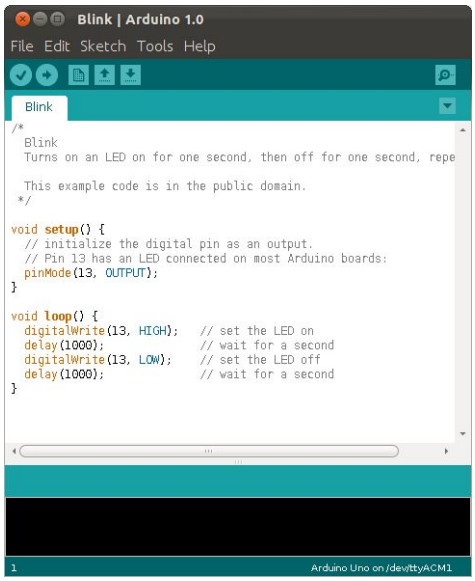
L'ESP32 se programme de plusieurs manières :

- IDF Internet Development Framework developpé par Espressif
- MicroPython
- C FreeRTOS
- **C++ Arduino**

Le chargement (bootloader) peut être :

- **Par liaison série via USB**
- Par Wifi OTA (Over The Air)

Environnement de développement logiciel



L'environnement de développement (IDE) : http://arduino.cc/en/main/software

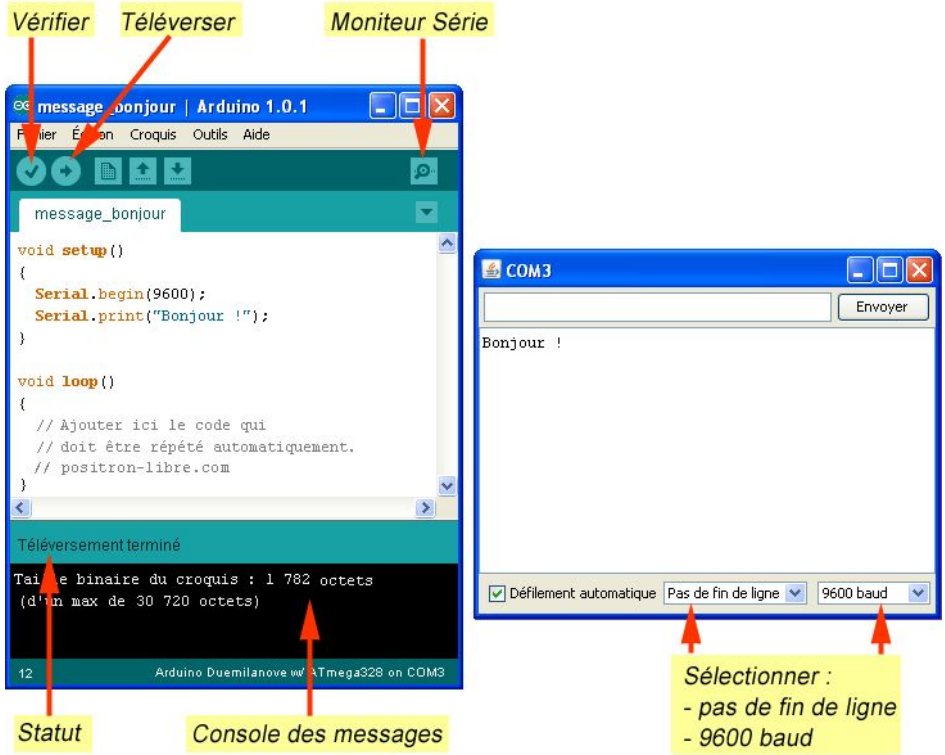
- écrit en Java (linux, windows, macos)
  - éditeur de code
  - compilateur
  - programmateur
  - terminal de commande
- Il est possible de compiler et de charger les programmes en lignes de commande.

Langage de programmation

- C++, compilé avec avr-g++
- bibliothèque de développement Arduino nommée Wiring pour le contrôle des composants internes du microcontrôleur.
- Un programme Arduino se nomme **sketch**, composé au minimum de 2 fonctions
  - `setup()` exécutée une fois pour initialiser les composants et les variables
  - `loop()` exécutée en boucle jusqu'à l'extinction de la carte

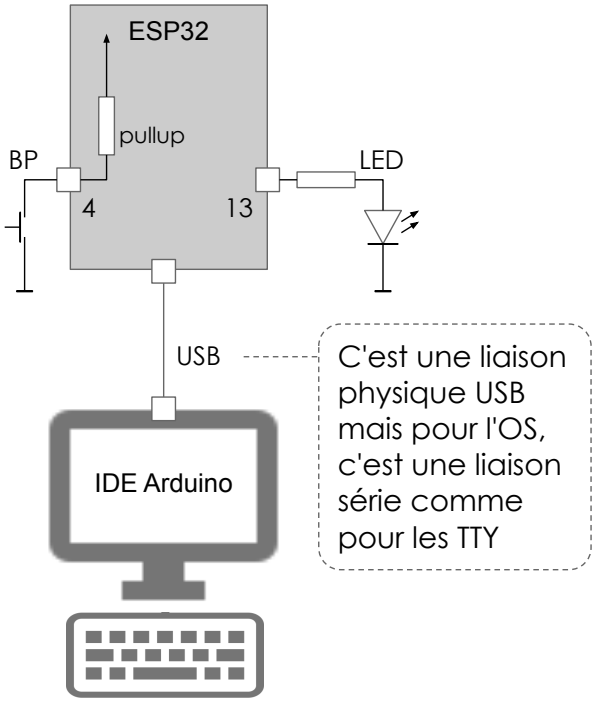
Menus de la fenêtre Arduino

http://www.positron-libre.com/robotique/robots/boe-shield-bot/notice/premier-programme.php



Exemple de programme

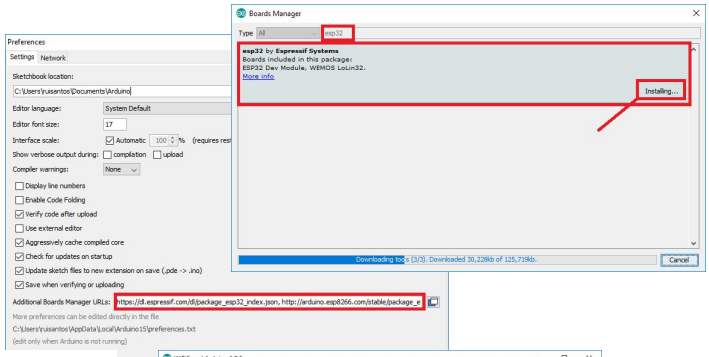
```
// Pas d'include par défaut
// Pas de fonction main()

// Il faut connaître les pins et l'architecture
const int buttonPin = 4;
const int ledPin =  13;
int buttonState = 0;

// Fonction exécutée une seule fois au reset
void setup() {
      Serial.begin(115200);
      pinMode(buttonPin, INPUT_PULLUP);
      pinMode(ledPin, OUTPUT);
}

// Fonction exécutée en boucle sans arrêt
void loop() {
      buttonState = digitalRead(buttonPin);
      Serial.println(buttonState);
      if (buttonState == HIGH) {
            digitalWrite(ledPin, HIGH);
      } else {
            digitalWrite(ledPin, LOW);
      }
}
```



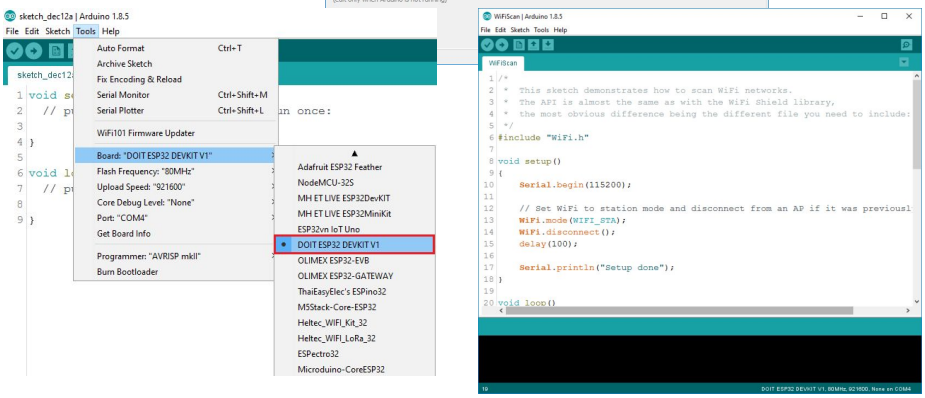
Installation sur Arduino

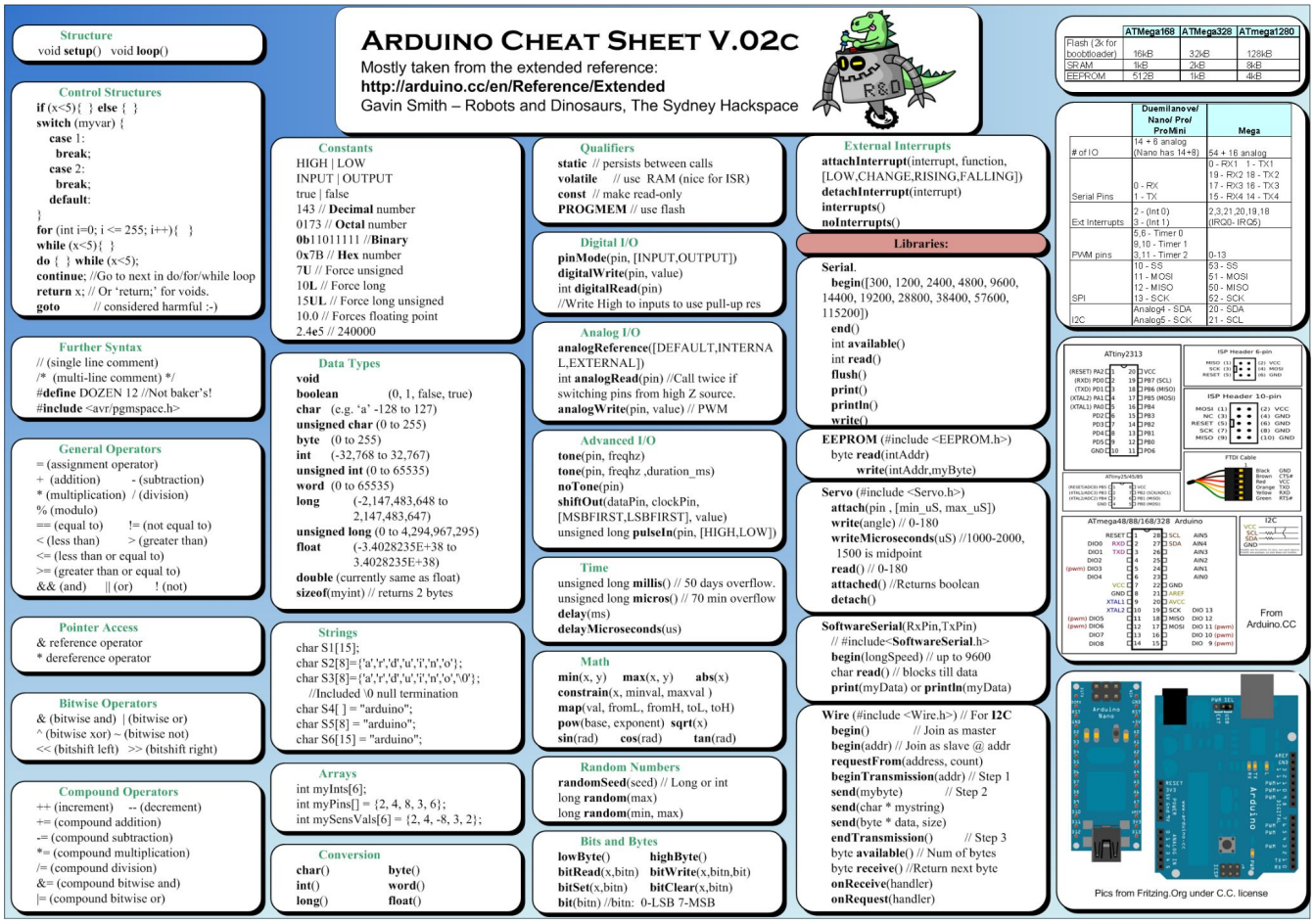
https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/

1. Installer les cartes ESP32 installation de la base de données des cartes ESP32 existantes



2. Tester

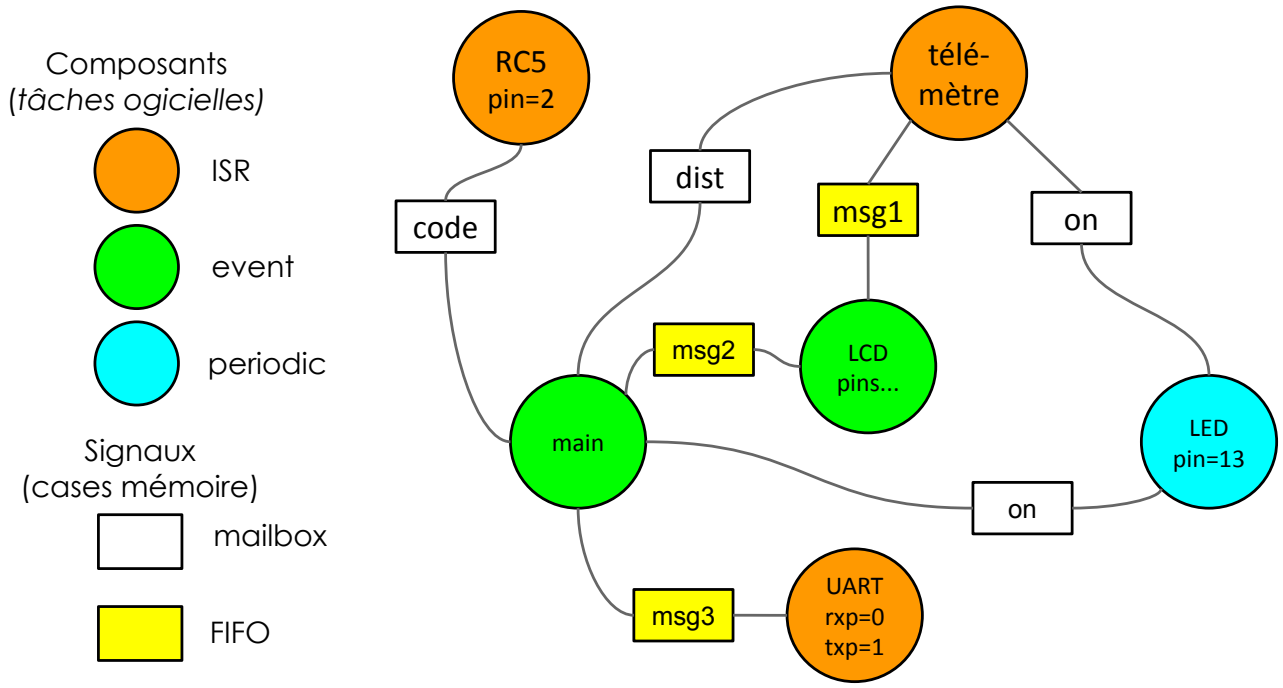




Programmation par automates

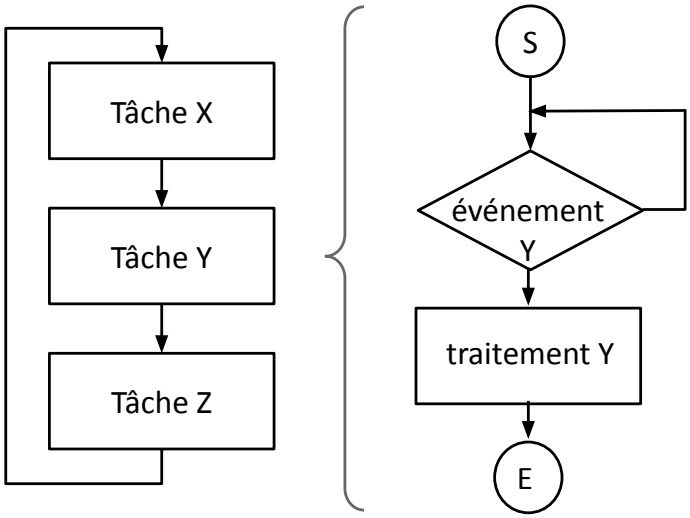
Applications

Les applications peuvent être vues comme des composants matériels communiquant par des signaux. Chaque composant étant décrit par un automate d'états finis.



Problème : exécution de plusieurs tâches

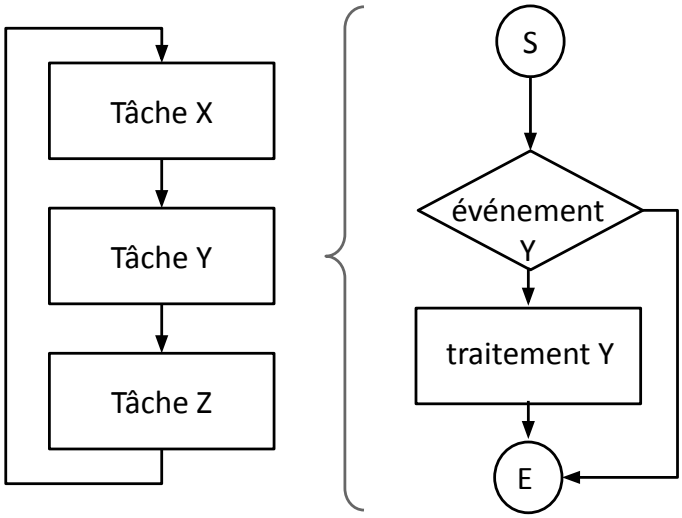
Un grand nombre des tâches destinées à un microcontrôleur dépendent d'événements externes périodiques ou non.



Si l'événement tarde à venir cela bloque l'exécution de toutes les autres tâches.

→ si les tâches coopèrent, on attend pas

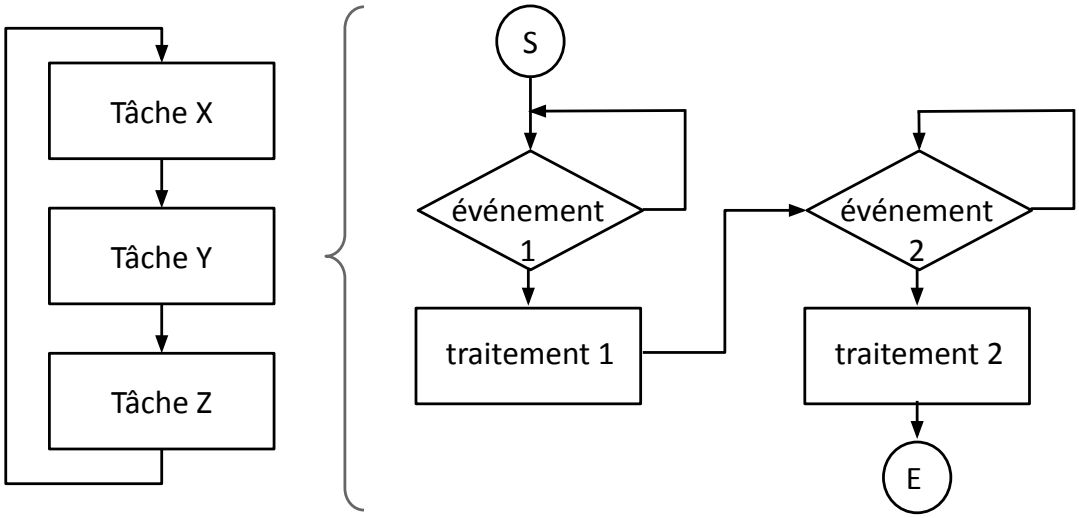
Si on peut garantir que les traitements sont bornés, alors il n'est pas obligatoire d'attendre l'événement.



Les tests des tâches doivent être multiplexées.
Les tâches ne se gênent pas.

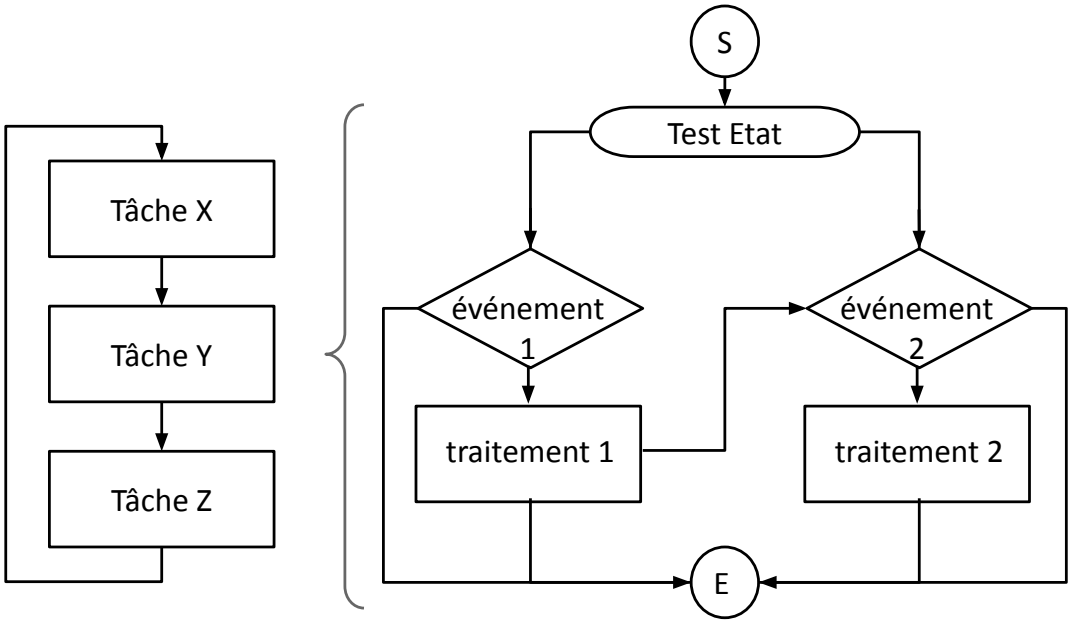
Une tâche peut avoir plusieurs traitements

Quand l'événement 1 arrive, il faut exécuter le traitement 1
Quand l'événement 2 arrive, il faut exécuter le traitement 2



→ les tâches doivent avoir des états

Si on n'attend pas les événements, il faut se souvenir
où en était la tâche.



Ordonnancement des tâches

- Une tâche est représentée par une fonction dont les arguments sont des entrées/sorties (adresses de cases mémoire)

  `void tache_1 (types entrées_sorties, ...)`
- Une instance d'exécution est l'exécution de la fonction de la tâche, cette instance est bornée dans le temps par construction
- L'ordonnancement consiste à exécuter les fonctions en boucle dans la fonction **loop()**

```
int x, y, z, t;
void loop() {
     tache_1(&x, &y, &z);
     tache_2(&x, &t);
     tache_3(&y, &z);
}
```

- La fonction setup va servir à initialiser un état interne.

Représentation de l'état interne

- Le comportement de la tâche est représenté par une fonction
- Cette fonction peut avoir des variables locales (temporaires) pour ses calculs
- Une tâche peut avoir un état interne (p.ex. le registre Etat) qui *pourrait* être représenté dans des variables locales **`static`**, mais cela pose deux problèmes :
  1. Il n'est pas possible d'initialiser ce registre d'état dans `setup()`
  2. Si une tâche a plusieurs exemplaires, il faut autant de fonctions que d'exemplaires.
- Donc on utilise une variable globale comme contexte d'exécution de la tâche

```
typedef struct ctx_tache_st {
       int etat;
       int x, y;
} ctx_tache_t;

ctx_tache_t ctx_tache;
void setup_tache(ctx_tache_t *ctx_tache, int ETAT) {
       ctx_tache→etat = ETAT;
       …
}
void loop_tache(ctx_tache_t *ctx_tache, int * in_out, ...) {
       switch (ctx_tache→etat) {
       case ETAT:
              …
       }
}
```

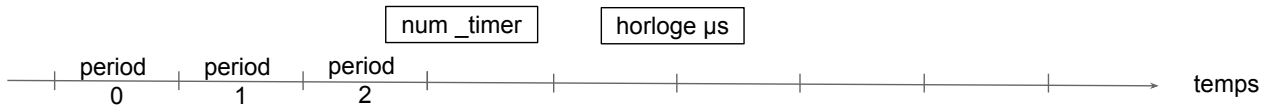
```
typedef struct ctx_tache_st {
            int etat;
} ctx_t0, ctx_t1;
int x;
setup() {
      setup_t0(&ctx_t0, ETAT0);
      setup_t1(&ctx_t1, ETAT1);
}
loop() {
      loop_t0(&ctx_t0, &x);
      loop_t1(&ctx_t1, &x);
}
```

Tâches périodiques

- Pour beaucoup de tâches, l'événement attendu est le temps. Par ex: on doit exécuter une tâche toutes les 20ms.
- Il y a deux solutions :
  1. Configurer un timer pour envoyer une interruption périodique
  2. Utiliser une horloge qui compte le temps et la consulter souvent

```
Soit H une horloge qui s'incrémente automatique
Soit T un registre qui servira à enregistrer le temps (date)
Soit P une période
Initialisation T ← H + P
tâche :
    si (H > T)        // événement attendu (on attend une date)
        exécution
        T ← T+P // mise à jour de la date
    fsi
```


Fonction waitFor



`waitFor(num_timer, period)`

rend le nombre de périodes écoulées depuis le dernier appel

`micros()`

rend la valeur de l'horloge (sur 32bits donc retour à 0 toutes les 1h11m)

```
// Configuration :
// - MAX_WAIT_FOR_TIMER : nombre maximum de timers utilisés
// arguments :
//  - timer  : numéro de timer entre 0 et MAX_WAIT_FOR_TIMER-1
//  - period : période souhaitée exprimée en microseconde
// valeur de retour :
//  - nombre de périodes écoulées depuis le dernier appel (normalement c'est 1)
#define MAX_WAIT_FOR_TIMER 2
unsigned int waitFor(int timer, unsigned long period){
  static unsigned long waitForTimer[MAX_WAIT_FOR_TIMER];
  unsigned long newTime = micros() / period;          // numéro de la période modulo 2^32
  int delta = newTime - waitForTimer[timer];          // delta entre la période courante et celle enregistrée
  if ( delta < 0 ) delta += 1 + (0xFFFFFFFF / period);  // en cas de dépassement du nombre de périodes possibles
  if ( delta ) waitForTimer[timer] = newTime;         // enregistrement du nouveau numéro de période
  return delta;
}
```

Tâche ISR

- Les tâches invoquées par loop ont un ordonnancement fifo
- Arduino permet facilement de configurer des ISR
  ISR = Interrupt Service Routine

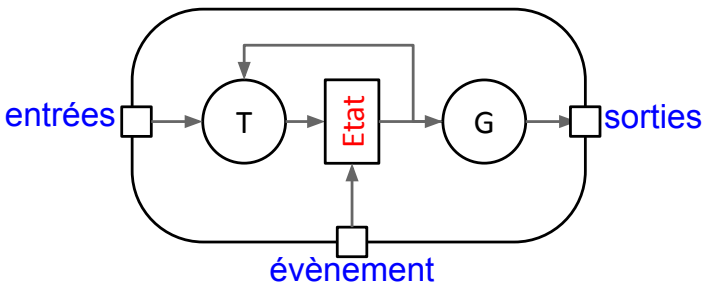
source:

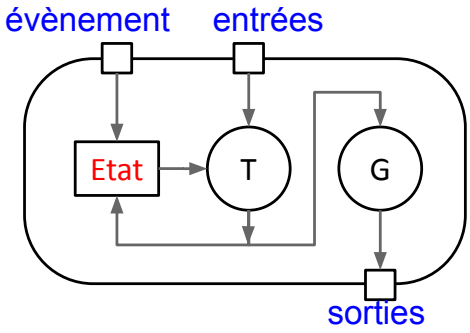
https://www.arduino.cc/reference/en/language/functions/external-interrupts/attachinterrupt/

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
	pinMode(ledPin, OUTPUT);
	pinMode(interruptPin, INPUT_PULLUP);
	attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
	digitalWrite(ledPin, state);
}

void blink() {
	state = !state;
}
```

- ici blink est un type de tâche ISR

Une tâche est une machine d'états finis

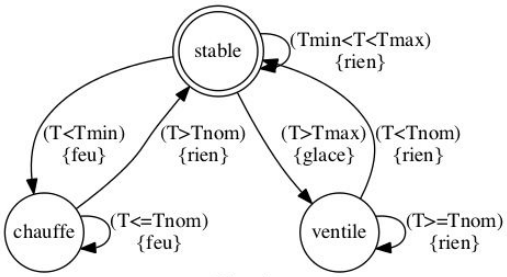





```
Aller à l'état courant (p.ex. E0)
Etat=E0:
    Si événement attendu
    Alors
        G : sorties ← traitement
        T : en fonction des entrées
             Etat ← état suivant
    Fsi
Etat=E1:
    [...]
```

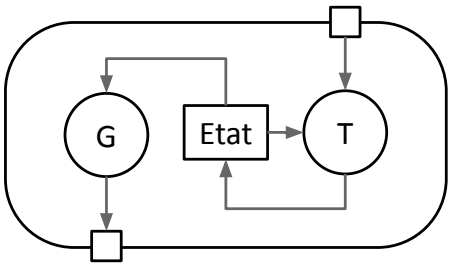
```
Aller à l'état courant (p.ex. E0)
Etat=E0:
    Si événement attendu
    Alors
        en fonction des entrées
           T : Etat ← état suivant
           G : sorties ← traitement
    Fsi
Etat=E1:
    [...]
```

Automate en C




```
enum {STABLE, CHAUFFE, VENTIL} state;
void setup_clim(ctx_clim_t * ctx, int timer, int period, int *T, int pin_V, int pin_C) {
        ctx->state=STABLE; ctx->ventil=OFF; ctx->radiateur=OFF;
        ctx->timer = timer;
        ctx->period = period;
        ctx->T = T; ctx->pin_v = pin_V ; ctx->pinc = pin_C;
}
void loop_clim(ctx_clim_t * ctx) {
        if (!waitfor(ctx->timer, ctx->period) return;
        switch (ctx->state) {
        case STABLE:
                if (ctx->T>Tmax)        {ctx->state=VENTIL; ctx->ventil=ON; ctx->radiateur=OFF;}
                else if (ctx->T<Tmin)   {ctx->state=CHAUFFE; ctx->ventil=OFF; ctx->radiateur=ON;}
                else                    {ctx->ventileur=OFF; ctx->radiateur=OFF;}
                break;
        case CHAUFFE:
                if (ctx->T>Tnom)        {ctx->state=STABLE; ctx->ventil=OFF; ctx->radiateur=OFF;}
                break;
        case VENTIL:
                if (ctx->T<Tnom)        {sctx->tate=STABLE; ctx->ventil=OFF; ctx->radiateur=OFF;}
                break;
        }
        digitalWrite(ctx->pin_v, ctx->ventil);
        digitalWrite(ctx->pin_c, ctx->radiateur);
}
```

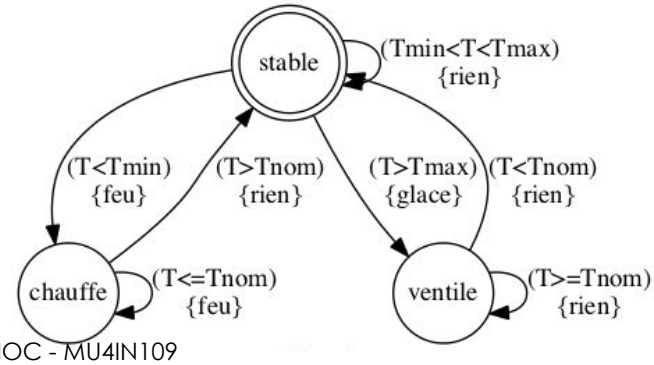

FSM : Machine à états finis



Une FSM (automate) est défini par :

- un nombre d'états finis
- un état initial
- un état courant
- une fonction de transition d'états définissant l'état futur à partir des entrées et de l'état courant
- une fonction de génération définissant les sorties à partir de l'état courant ou futur

On décrit le comportement d'uns FSM avec son graphe de transition d'états

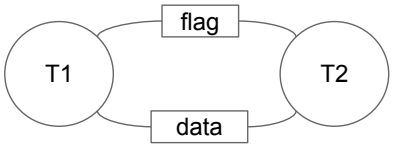


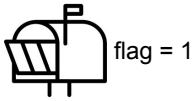
Climatiseur

INPUTS
3 températures : Tmin - Tnom - Tmax (p.ex. 18° 19° 20°)
la tempéraure T

ACTIONS
feu : {ventilateur = OFF; radiateur = ON;}
glace : {ventilateur = ON; radiateur = OFF;}
rien : {ventilateur = OFF; radiateur = OFF;}

Communication des tâches : boîte à lettre





- Les tâches communiquent
  - une T1 produit une donnée qu'utilise une tâche T2
  - le plus simple est d'utiliser une variable globale comme boîte à lettre

```
typedef struct boite_st {
        int flag;
        int data;
} boite_t;

boite_t b;

void T1 (..., boite_t *b,...)  {
        if (b→flag == 0) {
                utiliser b →data;
                b →data = VALUE;
                b→flag = 1;
        }
}
```

```
dans setup ()
        b→flag = 0;
        b→data = valeur_intiale;

void T2 (..., boite_t *b,...){
        if (b→flag == 1) {
                utiliser b→data;
                b→data = RETOUR;
                b→flag = 0;
        }
}
```

- On peut utiliser la boîte dans les deux sens
- On peut faire une communication avec et sans perte

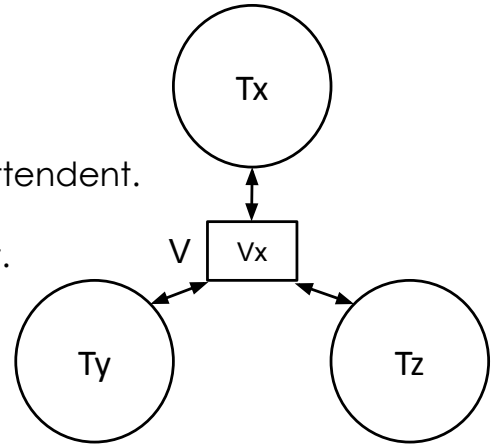
Set-Reset Flag

Pour la synchronisation des tâches on utilise une variable partagée V.

- On associe à chaque tâche T_i une valeur propre V_i.
- Une tâche T_x ne peut changer la valeur de la variable que si elle contient sa valeur propre V_x

A l'initialisation:

- V ← Vx
  Tx peut donc travailler, les autres tâches attendent.
- Puis Tx place Vy dans V
  Ty peut alors travailler, les autres attendent.
- Puis Ty place Vz dans V
  C'est alors Tz qui peut avancer



Buffer + flags (Set-Reset-Flag)

Si deux tâches T1 et T2 s'échangent des données par BAL

- La valeur du drapeau désigne le propriétaire du buffer
  0. le buffer appartient à T1
    - T1 peut lire ce que contient le buffer
    - T1 peut remplir le buffer
    - T1 informe T2 que le buffer est plein en mettant 1 dans le drapeau

  1. le buffer appartient à T2
    - T2 peut lire ce qu'à écrit T1
    - T2 peut écrire une réponse
    - T2 informe T1 que le buffer est lu (et qu'une réponse est mise) en mettant 0 dans le drapeau

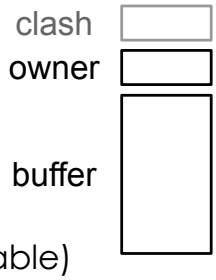
Si T1 s'interdit d'écrire si le drapeau est à 1

⇒ communication sans perte

Si T1 écrit même quand le drapeau owner est à 1

⇒ communication à perte
Il faut un drapeau de collision en plus (clash)

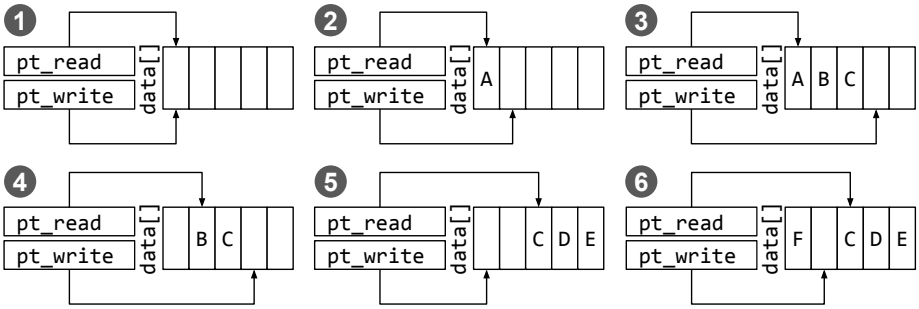
Le buffer peut être utilisée dans les deux sens (entente préalable)



FIFO

Un dernier mécanisme de communication simple est la FIFO

Suivant le comportement de **tty_fifo_push**() on peut avoir une FIFO avec ou sans perte

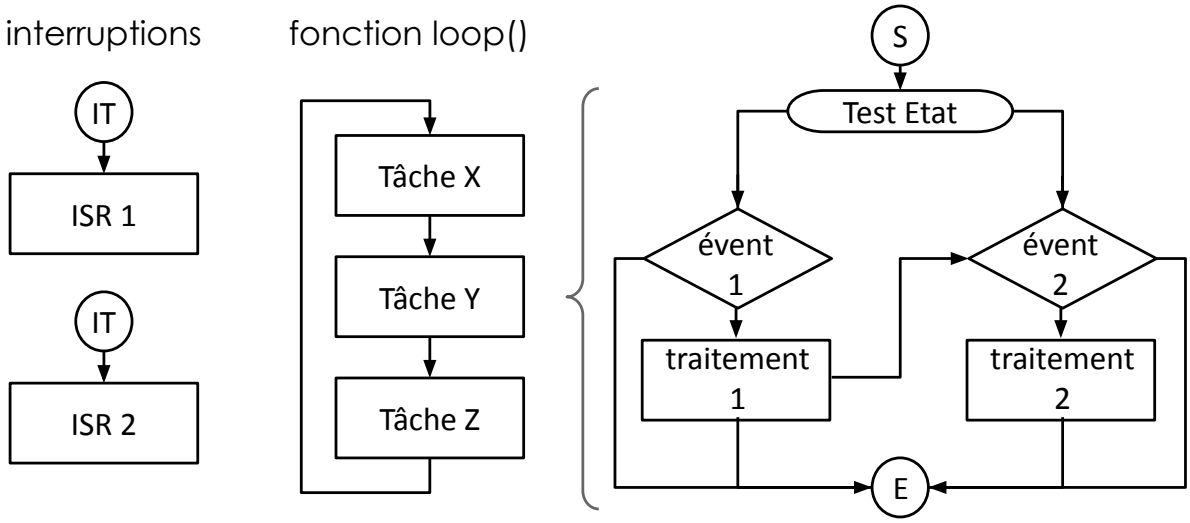



```
struct tty_fifo_s {
        char data [20];
        int  pt_read;
        int  pt_write;
};

static int tty_fifo_push (struct tty_fifo_s *fifo, int c) {
        int pt_write_next = (fifo->pt_write + 1) % sizeof(fifo->data);
        if (pt_write_next != fifo->pt_read) {
                fifo->data [fifo->pt_write] = c;
                fifo->pt_write = pt_write_next;
                return 1;
        }
        return 0;
}
```

```
static int tty_fifo_pull (struct tty_fifo_s *fifo, int *c) {
        if (fifo->pt_read != fifo->pt_write) {
                *c = fifo->data [fifo->pt_read];
                fifo->pt_read = (fifo->pt_read + 1)% sizeof(fifo->data);
                return 1;
        }
        return 0;
}
```

En résumé 1/2

Les tâches sont des fonctions exécutées périodiquement ou après une interruption.
Ces fonctions réalise un traitement unitaire borné dans le temps.



En résumé 2/2

- Une interface vers les signaux
- Des paramètres de configuration globaux
- Un contexte
  - mémoire d'état interne
  - configuration d'instance
- Des fonctions de comportement
  - initialisation : setup
  - cas normal : loop ou ISR
  - *exception : cas d'erreur optionnel*
- mailbox
  - un buffer de données
  - un drapeau d'état
    - ready / busy
    - full / empty
- FIFO
  - un tableau de buffer
  - un état
    - un pointeur de lecture
    - un pointeur d'écriture
  - API push - pull

TME

- Programmation de l'ESP32 en multitâches
  - installer l'environnement ESP32
  - faire clignoter la led
  - faire clignoter la led en fonction de la lumière reçue
  - afficher l'état du bouton poussoir sur l'écran oled
  - configurer le comportement en fonction de commandes reçu sur le terminal
  - …