Objets connectés et robotique

L3 IFA, S5

Octobre 2026

Vincent Lannurien [1]

vincent.lannurien@univ-brest.fr

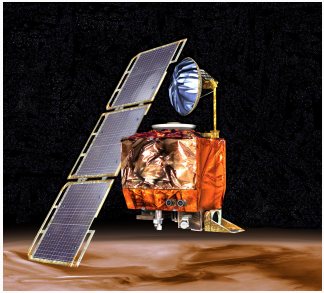
[1] Lab-STICC, CNRS, UMR 6285, Université de Bretagne Occidentale, Brest

Lab-STICC

UBO
Université de Bretagne Occidentale

Introduction

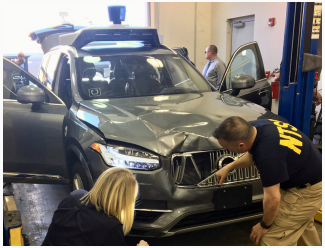
Quelques systèmes embarqués...

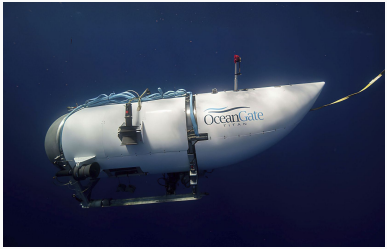










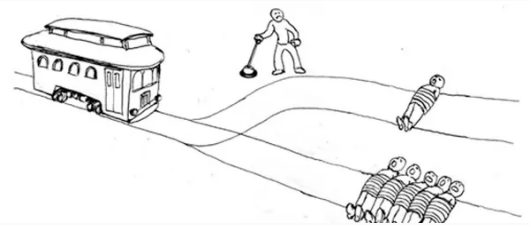


… devenus tristement célèbres

- https://en.wikipedia.org/wiki/Mars_Climate_Orbiter
  - problème d'unités (système métrique vs système impérial)
- https://en.wikipedia.org/wiki/Ariane_flight_V88
  - erreur de conversion (*overflow*)
- https://en.wikipedia.org/wiki/14th_Quartermaster_Detachment
  - bug d'arrondi (*overflow*)
- https://en.wikipedia.org/wiki/Therac-25
  - erreur de programmation concurrente (*race condition*)
- https://en.wikipedia.org/wiki/Death_of_Elaine_Herzberg
  - détection d'obstacle inopérante
- https://en.wikipedia.org/wiki/Titan_submersible_implosion
  - certifications de sécurité insuffisantes

Commençons par un test...

- *Trolley problem* (https://en.wikipedia.org/wiki/Trolley_problem) : le dilemme du tramway

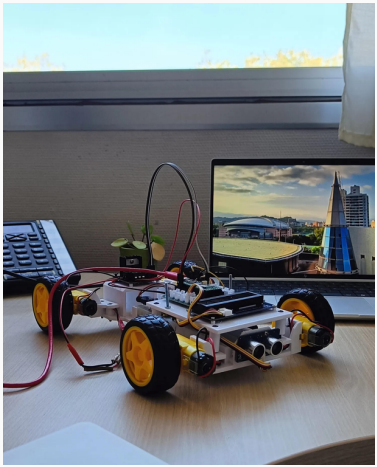


🤔

Que feriez-vous ? → https://www.moralmachine.net/hl/fr

Plan du cours

1. Introduction
2. Organisation de l'UE
3. Projet à réaliser
4. Contexte
5. Informatique embarquée
   - Microcontrôleurs
   - Raspberry Pi Pico et écosystème
   - Programmation et gestion des Entrées/Sorties
   - Communications
   - Chaîne de compilation
6. Robotique
   - Généralités
   - Commande à distance
   - Interface utilisateur
   - Missions
   - Comportement autonome et contrôle
   - Calcul déporté
7. Présentation des TP



Organisation de l’UE

Organisation de l'UE – Contenu

- Objectifs :
  - Savoir concevoir des systèmes cyberphysiques simples, autonomes
  - Comprendre les compromis entre performance (latence, débit) et consommation (énergie, mémoire)
- Méthode :
  - Le plus de **manipulations** possible !
  - Dimension **matérielle** (un peu d'électronique, un peu de mécanique)
  - Dimension **logicielle** (de l'informatique !)

Organisation de l'UE – Logistique

- Volume horaire :
  - CM : 8h
  - TP : 14h
- Travail en groupes :
  - Un robot par personne
  - Binômes possibles dans les groupes de TP
- Examen :
  - Démo **individuelle** lors des TP
  - Rendu de projet sur Moodle
- Intervenants :
  - Vincent Lannurien – `vincent.lannurien@univ-brest.fr`
  - Kilian Le Gall – `kilian.legall@univ-brest.fr`

Projet à réaliser

Portée du projet

Objectifs :

- Prise en main de l'environnement de développement
- Implantation d'algorithmes simples sur le robot
- Étude des compromis performances/ressources

Tout ce que l'on ne va pas faire :

- Électronique : montages complexes, conception de carte...
- Traitement du signal : radio, interférences...
- Sécurité : authentification, détection d'intrusion...

Portée du projet

Objectifs :

- Prise en main de l'environnement de développement
- Implantation d'algorithmes simples sur le robot
- Étude des compromis performances/ressources

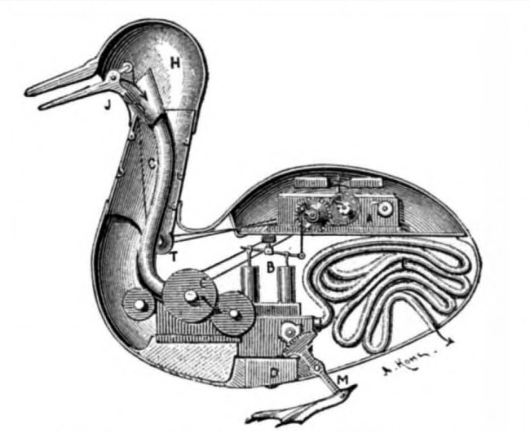
Tout ce que l'on ne va pas faire :

- Électronique : montages complexes, conception de carte...
- Traitement du signal : radio, interférences...
- Sécurité : authentification, détection d'intrusion...

Étapes du projet

- Commande : Fonctionnement de base d'un robot
  - Envoi de messages : avancer, reculer, tourner, etc.
  - Communications : filaire (série), sans-fil (radio)
  - Interface web de commande : Node.js, JavaScript
- Environnement : capteurs et actionneurs
  - Utilisation d'un capteur de distance
  - Fonctionnement des moteurs
- Contrôle : Programmation de missions simples
  - Comportement autonome :
    - Avancer jusqu'à trouver un obstacle
    - Avancer jusqu'à trouver une ouverture
    - Faire un quart de tour
    - Contourner un objet
- En option :
  - Plus de capteurs (second capteur de distance, caméra monochrome…)
  - Plus de cartes (calcul déporté, communication entre robots…)

Contexte



INTERIOR OF VAUCANSON'S AUTOMATIC DUCK.

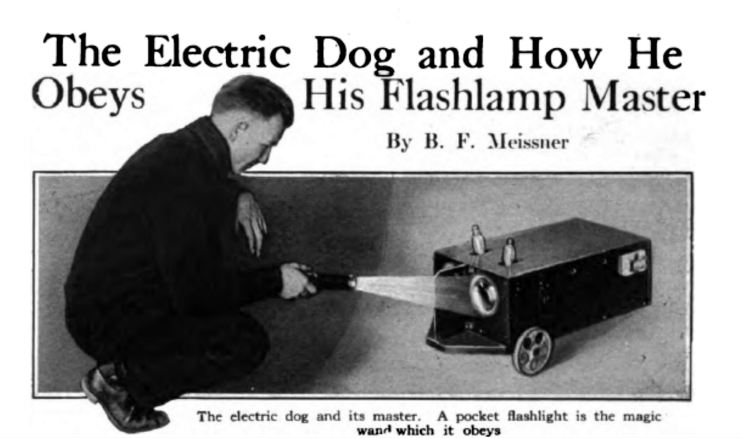
A, clockwork; *B*, pump; *C*, mill for grinding grain; *F*, intestinal tube; *J*, bill; *H*, head; *M*, feet.

Canard digérant (Jacques de Vaucanson, 1739)

Histoire de la robotique – Automates



Serinette de Jacotel (famille Husson-Jacotel, début du XIX^{e})



Chien électrique (John Hammond et Benjamin Miessner, 1915)

Histoire de la robotique – Automates



Le Défenseur du Temps (Jacques Monestier, 1979)

Histoire de la robotique – Robots



Goliath, mine filoguidée activable à distance (Normandie, 1944)

- Accident de Tchernobyl (1986) :
  - **Liquidateurs** : personnel civil et militaire intervenu immédiatement sur les lieux de la catastrophe nucléaire
  - 600 000 personnes ont reçu des doses équivalentes à Hiroshima ou Nagazaki
    - Reconnaissance, détection de la radioactivité
    - Démantèlement des débris du réacteur
    - Documentation (photographie)
    - Échantillonnage des matériaux
    - Enfouissement des déchets



Stèle à la mémoire des liquidateurs (Kharkiv, Ukraine)

Histoire de la robotique – Robots

- Développement de robots télécommandés pour l'inspection et le nettoyage **à distance**
  - **Télédétection** : caméra, capteurs de radiation
  - **Télémétrie** : transmission des données vers un poste de contrôle
  - **Télémanipulation** : bras robotique pour manipuler les débris



Robot **Joker** (1986)

Histoire de la robotique – Robots



Ligne 14 du métro de Paris

Automates ou robots ?

- **Automate** :
  - répète une séquence fixe de mouvements mécaniques
  - ne s'adapte pas à son environnement
- **Robot** :
  - possède des capacités de *perception*
  - peut *interagir* avec son environnement
  - peut prendre des décisions *autonomes*

Définitions – Systèmes cyberphysiques

Definition

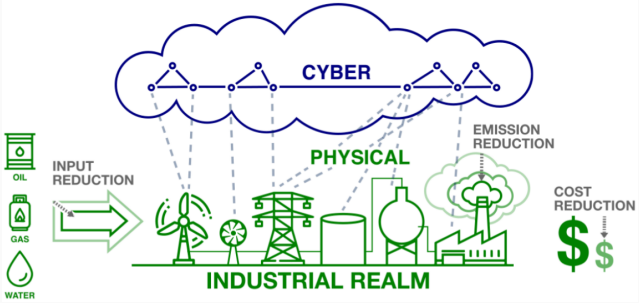
Cyber-Physical Systems (CPS) are integrations of computation with physical processes. [1]

- Énergie : *smart grids*
- Transport : systèmes ferroviaires, aviation civile, automobile autonome…
- Santé : dispositifs médicaux « connectés »
- Industrie : usines automatisées, robots industriels
- Ville « intelligente »

1. Edward A. Lee. « Cyber Physical Systems : Design Challenges ». *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008.

Definition

Cyber-Physical Systems (CPS) are integrations of computation with physical processes. [2]

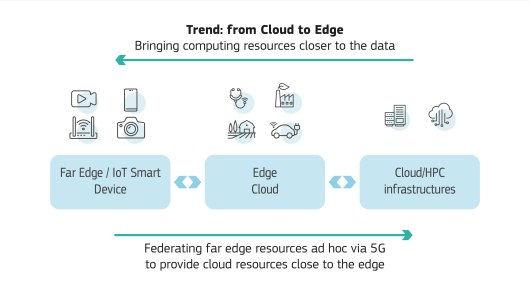


2. Edward A. Lee. « Cyber Physical Systems : Design Challenges ». *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. 2008.

3. [3] Oliver Inderwildi et al. « The Impact of Intelligent Cyber-Physical Systems on the Decarbonization of Energy ». *Energy & Environmental Science* (2020)

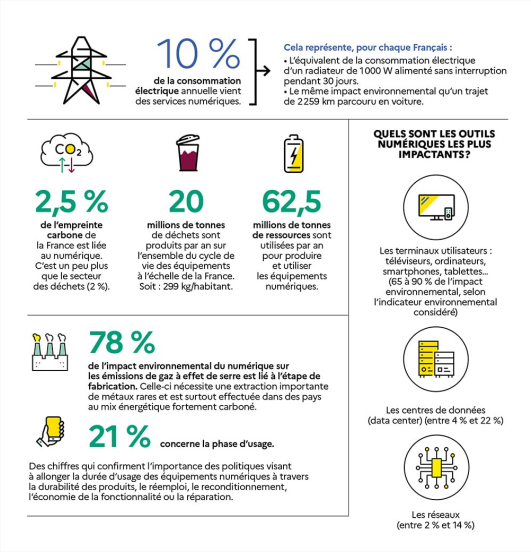
Quelques tendances [ ]

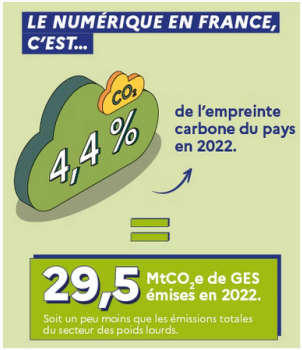
- 41 milliards d'appareils *IoT* déployés en 2025
- Intelligence artificielle de plus en plus déployée en périphérie de réseau (Apple Intelligence, Google AI Edge)
- Défis : interopérabilité, durabilité

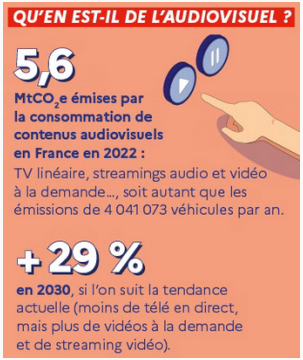


Pour chaque Français, le numérique représente :

- **Consommation électrique** : 10%
- **Empreinte carbone** : 2.5%
- **Déchets** : 299 kg/habitant/an
- **Fabrication** : 78% de l'impact
- **Utilisation** : 21% de l'impact









- Hausse de l'impact des **centres de données** : 46% de l'empreinte carbone du numérique

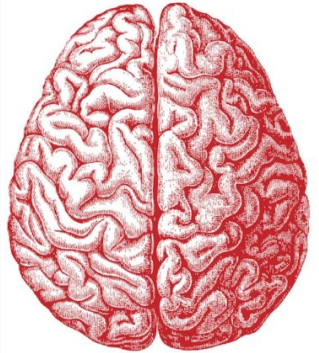
Informatique embarquée

Informatique embarquée

Microcontrôleurs

Microcontrôleurs : Utilisations et limites





- **Circuit intégré** : calcul (processeur), mémoire (mémoire vive et non-volatile), interfaces d'Entrées/Sorties
- **Non-programmable** : son comportement est gravé dans le silicium
- **Utilisation** : partout où un processeur généraliste n'est pas nécessaire
  - + : Plus simple à comprendre, à vérifier
  - + : Consommation d'énergie minimale
  - - : Peu flexible
- **Coût** : en 2018, en moyenne, \$0.03 pour un µc 8-bit, \$1 pour un µc 32-bit $\rightarrow$ ratio $\frac{1}{100}$ par rapport à un PC grand public

- **Énergie** (VCC, GND, tension d'alimentation)
- **Architecture** (horloge, mémoire)
- **Signal** et **interfaces** (GPIO, ADC, PWM)
- **Capteurs** (communications série, UART, I²C, SPI)
- **Actionneurs** (étage de puissance, moteurs, servomoteurs)
- **Mécanique** (moteur + réducteur, roues, châssis)

Principes de câblage

- Tensions, masse et référence commune
- Différence entre **tension d'alimentation** et **tension logique**
- Transporter l'**énergie** : une différence de tension provoque la circulation d'un courant lorsque le circuit électrique est fermé (exemple : LED branchée entre $3.3V$ et *GND*)
- Transporter l'**information** : la tension n'est pas forcément constante ; le microcontrôleur peut la faire varier pour transmettre un signal (tension proche de $0V$ = niveau logique à 0, tension proche de $3.3V$ = niveau logique à 1)

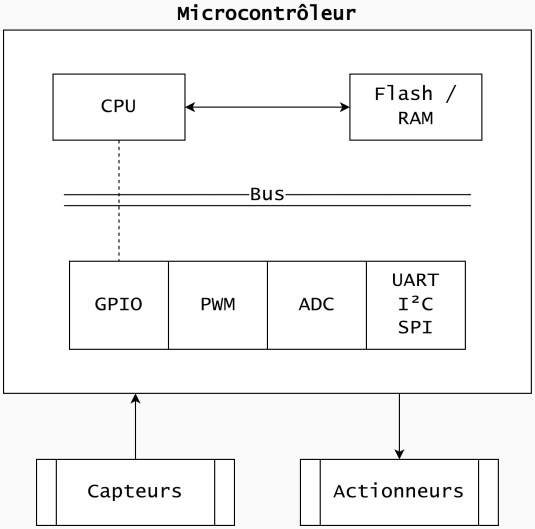
Principes de câblage

- Commander l'allumage d'une LED, le démarrage d'un moteur, lire l'état d'un bouton, envoyer des données sur un bus de communication, etc.
- **ADC** (*Analog to Digital Converter*) : mesure la tension et la transforme en valeur numérique utilisable par le programme
- Tension = différence de potentiel entre deux points
  - $U_{AB} = V_A - V_B$
- Il faut donc toujours une **référence**
  - On choisit un point comme $0V$: *GND*
  - *GND* sert également de chemin de retour au courant du signal
- GPIO à $3.3V$ signifie « GPIO à $3.3V$ au-dessus de *GND* »

Principes de câblage

- Pull-up et pull-down : résistance de rappel pour forcer un état logique par défaut
- Exemple : bouton poussoir connecté entre *GND* et un GPIO configuré en entrée avec pull-up
  - Bouton relâché : GPIO = $3.3V$ (niveau logique à 1)
  - Bouton appuyé : GPIO = $0V$ (niveau logique à 0)
- Nécessaire lorsque le circuit n'est pas fermé (bouton relâché) pour éviter que le GPIO « flotte » et prenne des valeurs aléatoires

Architecture



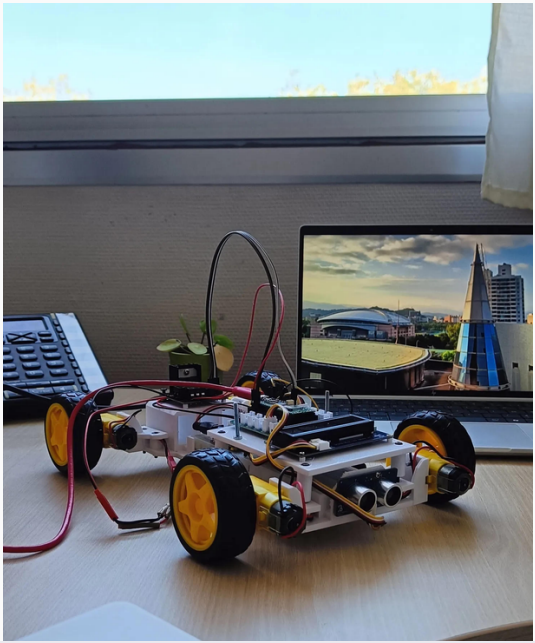
Architecture des ordinateurs

Le *programme* est stocké en **flash** ; le **processeur** en exécute les *instructions* ; l'**horloge** *cadence* son fonctionnement ; la **RAM** conserve les *données* nécessaires à l'exécution.

Quelle différence entre un ordinateur et un microcontrôleur ?

Informatique embarquée

Raspberry Pi Pico et écosystème

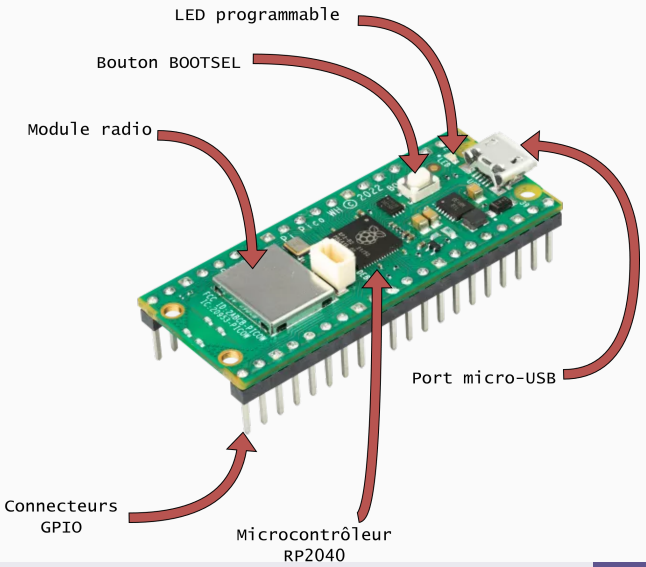


Raspberry Pi Pico WH : caractéristiques (RP2040)

Plateforme de *prototypage*

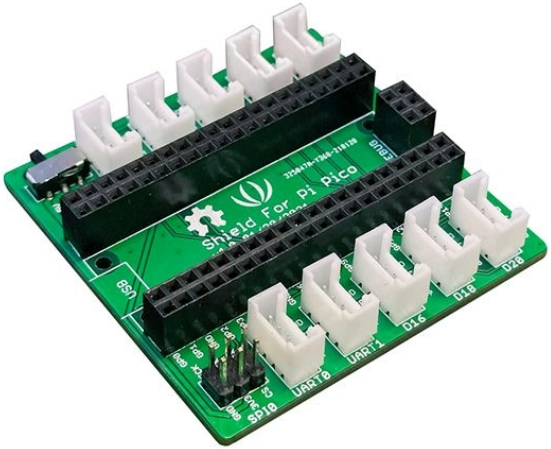
Permet de reprogrammer le système !

- Microcontrôleur RP2040
- CPU bicœur Arm Cortex-M0+ @ 133 MHz
- 264 kB SRAM
- 2 MB flash
- Wi-Fi 2.4GHz 802.11n
- Bluetooth 5.2
- 26 broches GPIO
- 2x UART, 2x SPI, 2x I²C, 16x PWM
- Tension supportée : 1.8 à $5.5V$



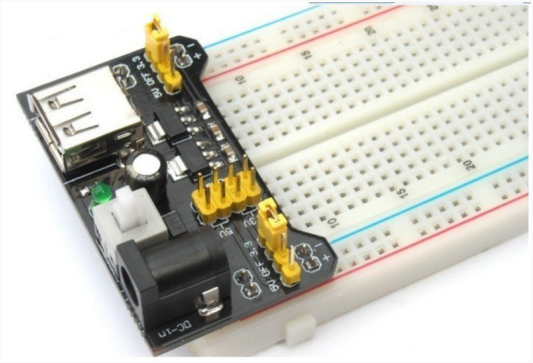
Shield générique

- Simplifie le câblage des capteurs et actionneurs
  - Redistribue les broches (GPIO, VCC, GND) du Raspberry Pi Pico sur des connecteurs standards à 4 fils : *GND*, *VCC*, $signal_2$, $signal_1$
- Fournit une masse commune sur les connecteurs (I²C, SPI…)
- Fournit une tension d'alimentation régulée



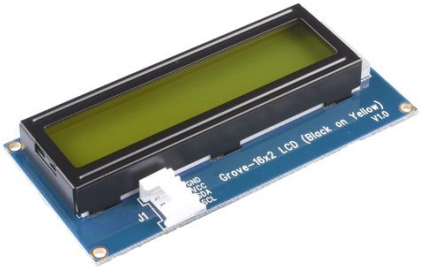
Breadboard et abaisseur de tension

- *Breadboard* : permet de réaliser des montages électroniques sans soudure
- Abaisseur de tension : permet d'alimenter le robot en $5V$ à partir d'une source à une tension plus haute
  - *e.g.*, une pile $9V$



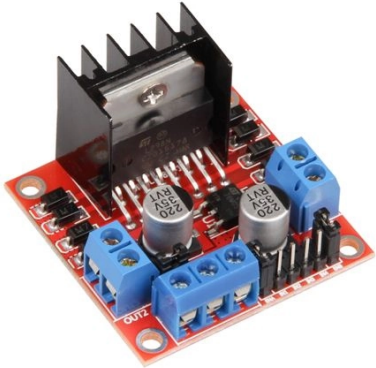
Afficheur LCD

- Afficheur LCD 16x2 caractères
- Interface I²C
- Permet d'afficher des informations sur le robot



Shield de commande de moteurs

- L298N : 2x « Pont en H »
- Interface numérique sur quatre broches de signal
  - IN1, IN2 : commande des deux moteurs gauches
  - IN3, IN4 : commande des deux moteurs droits
- Applique une quantité réglable de puissance à une charge
- Permet de commander deux moteurs à courant continu
- Permet de faire varier la vitesse et le sens de rotation des moteurs



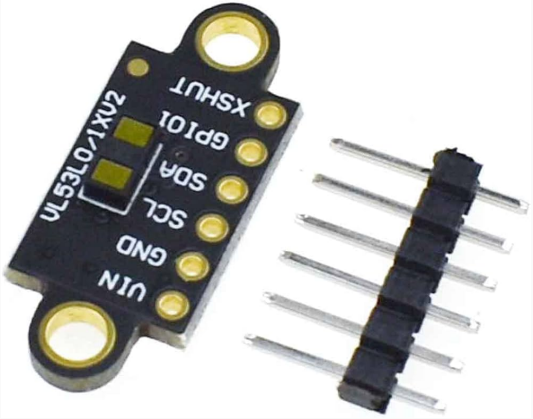
Télémètre à ultrasons

- Capteur de distance à ultrasons
- Interface numérique sur une broche
  - 3 fils : **SIG** (signal numérique bidirectionnel) + VCC + GND
- Permet de mesurer la distance entre le robot et un obstacle
- Principe : émettre une impulsion sonore et mesurer le temps de retour de l'écho
  - Distance = $\frac{v_{son} \times t_{echo}}{2}$, avec $v_{son} \approx 343 m/s$ à $20°C$
- Précision : 1 mm, portée : 30 mm à 700 mm



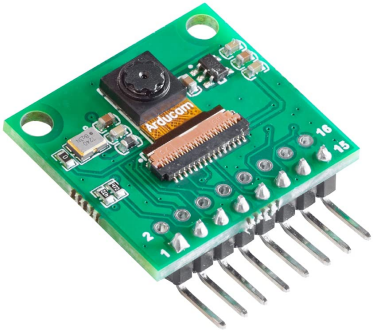
Télémètre à infrarouges

- Capteur de distance à infrarouges
- Interface I²C
- Permet de mesurer la distance entre le robot et un obstacle
- Principe : émettre une impulsion lumineuse et mesurer le temps de retour de l'écho
  - Distance = $\frac{v_{lumiere} \times t_{echo}}{2}$, avec $v_{lumiere} \approx 3 \times 10^8 m/s$
- Précision : 1 mm, portée : environ 2500 mm



Caméra monochrome

- Caméra monochrome, focus fixe, angle de vision à 60°
- Interface I²C
- Permet de capturer des images et de les transmettre au microcontrôleur
- Résolution : QVGA (320x240 pixels), format : RAW6/RAW8, taille : 76 800 octets (75 KB)
- Débit : 30 fps $\times$ 76 800 octets = 2250 KB/s
- Consommation d'énergie : $\lesssim$ 2 mW

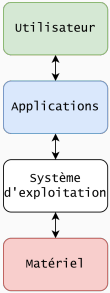


Informatique embarquée

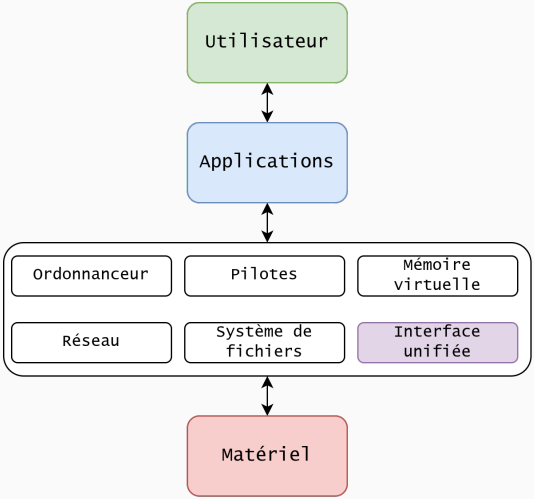
Programmation et gestion des Entrées/Sorties

Quizz

Qu’est-ce que fournit un système d’exploitation ?



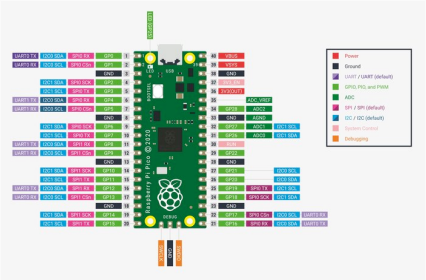
Rôle d'un système d'exploitation



Programmation d'un microcontrôleur

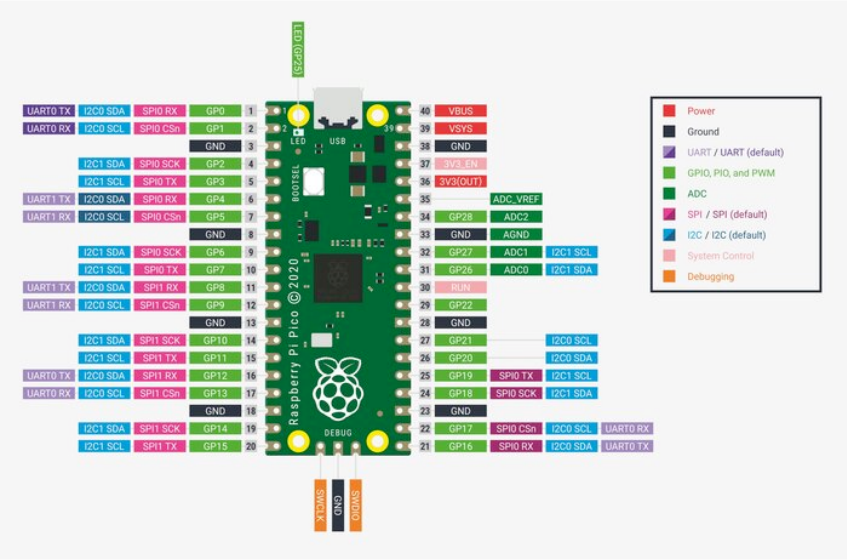
- **Pas de système d'exploitation !**
- Le programme est exécuté directement par le microcontrôleur
- Pas de gestion de la mémoire, pas de processus
- Accès direct aux ressources matérielles
- Programmation en langage C/C++ ou MicroPython

- **GPIO** : *General-Purpose Input Output*
- Lecture **analogique** : mesure d'une *tension* sur certaines broches
  - Conversion analogique vers numérique (*ADC*)
  - $[0.0, 3.3]$ $V \rightarrow$ `int` $[0, 4095]$
  - Précision : 12 bits, résolution théorique : ≈ 0.81 mV/valeur
- Lecture/Écriture **numérique** : états *logiques* (binaires)
  - 0 (`LOW`, 0V)
  - 1 (`HIGH`, 3.3V ou 5V)



Raspberry Pi Pico WH : broches GPIO**26**, GPIO**27** et GPIO**28** en entrées ADC (peuvent lire la tension en sortie d'un périphérique).

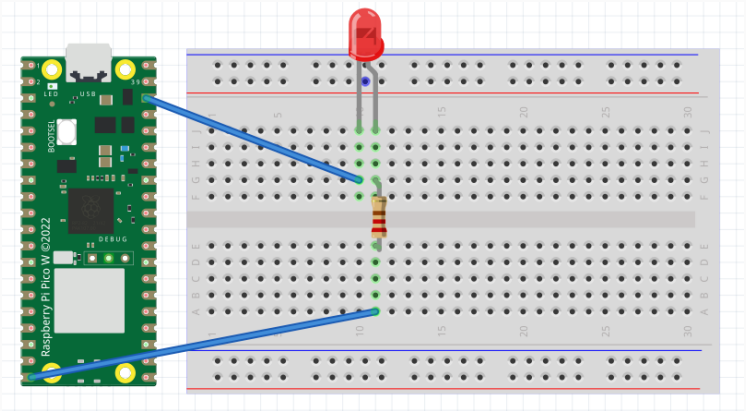
Raspberry Pi Pico – *Pinout* (brochage)



Interface	Niveau d'abstraction	⊕	⊖
MicroPython (Python)	Haut	Simple, interactif	Performance, déterminisme
Pico SDK (C/C++)	Bas à moyen	Contrôle, performances	Lisibilité, compilation
Arduino (C++)	Haut	Portabilité, écosystème	Contrôle, performances

Une même tache dans différent paradigmes

Configurer GP15 en sortie et faire clignoter une LED.



```
from machine import Pin
from time import sleep

led = Pin(15, Pin.OUT)

while True:
    led.toggle()
    sleep(1)
```

```c
#include "pico/stdlib.h"

int main() {
    gpio_init(15);
    gpio_set_dir(15, GPIO_OUT);

    while (true) {
        gpio_put(15, 1);
        sleep_ms(1000);

        gpio_put(15, 0);
        sleep_ms(1000);
    }
}
```

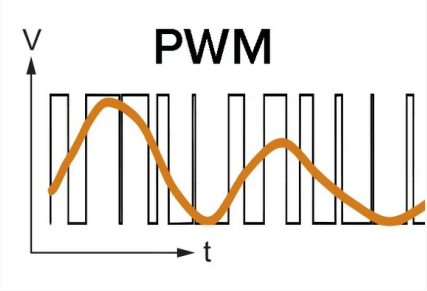
```
void setup() {
    pinMode(15, OUTPUT);
}

void loop() {
    digitalWrite(15, HIGH);
    delay(1000);
    digitalWrite(15, LOW);
    delay(1000);
}
```

Quizz

Comment faire varier l'intensité de la LED pour produire un effet de fondu, en boucle ?

- Écriture analogique ?
  - Le microcontrôleur ne peut pas générer une tension arbitraire…
- Décrire un phénomène continu par des modulations
- Technique : *PWM* (*Pulse Width Modulation*)
  - Générer un signal carré de fréquence fixe
  - Faire varier le rapport cyclique (durée du niveau haut / période)
  - La valeur moyenne du signal est proportionnelle au rapport cyclique



- Assuré par un composant matériel dédié pour ne pas monopoliser le processeur

Mesurer le temps

Dans le but d'évaluer les performances, mesurer la durée d'une action, etc.

```c
#include "pico/stdlib.h"
#include <stdio.h>

int main() {
    stdio_init_all();

    uint64_t start = time_us_64();
    sleep_ms(250);
    uint64_t stop = time_us_64();

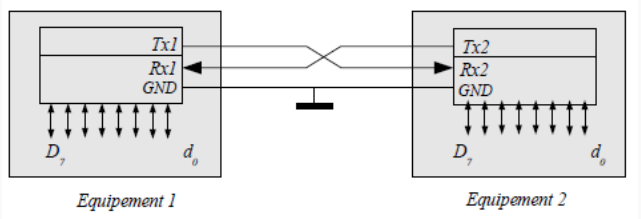
    uint64_t elapsed_us = stop - start;
    printf("Elapsed: %.3f ms\n", elapsed_us / 1000.0);
}
```

Informatique embarquée

Communications

Communications série : UART [ ]

- **UART** : *Universal Asynchronous Receiver Transmitter*
- Convertit des données *parallèles* (un octet en mémoire) en données *série* (des bits sur un fil)
- Liaison avec 3 fils : **Tx** (*transmit* de l'émetteur), **Rx** (*receive* du récepteur), **GND** (masse)
- *Full duplex* : émission et réception simultanées ; point à point : 2 éléments
- Vitesse : en général 9600 bauds (bits/seconde), au maximum 115200 bauds (environ 14 kB/seconde…)
- « Masqué » derrière le port USB du Raspberry !



```c
#include <stdio.h>
#include "pico/stdlib.h"

int main() {
    // UART <-> USB
    stdio_init_all();

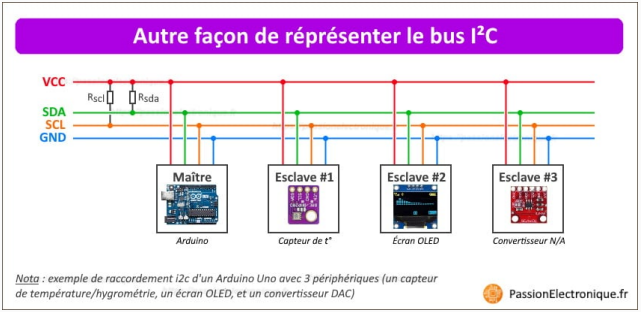
    while (true) {
        // Affichage dans le terminal série
        printf("Hello, World!\n");
        sleep_ms(1000);
    }
}
```

```c
#include <stdio.h>
#include "pico/stdlib.h"

int main() {
    // UART <-> USB
    stdio_init_all();

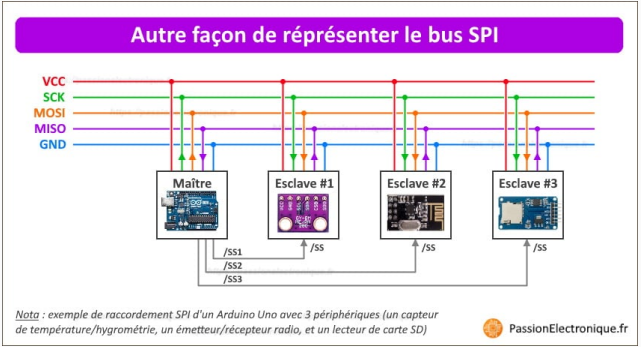
    while (true) {
        // Lecture (non-bloquante !) depuis le terminal série
        int c = getchar_timeout_us(0);
        if (c != PICO_ERROR_TIMEOUT) {
            printf("Reçu : %c\n", c);
        }
        // Reste du programme...
    }
}
```

- **I²C** : *Inter-Integrated Circuit*
- Topologie « maître-esclave », adresses uniques, bits de lecture/écriture
- Liaison avec trois fils : **SDA** (signal de données), **SCL** (signal d'horloge), **GND** (masse)
- *Semi duplex* : communications bidirectionnelles, mais pas simultanées ; multi-points
- Vitesse : en général 100 kbps (10x plus rapide que l'UART), au maximum 3.4 Mbps (environ 425 kB/seconde)



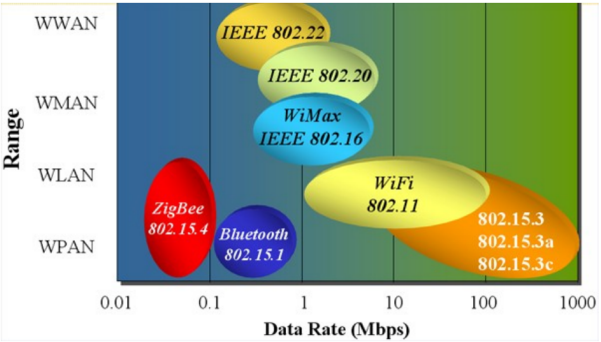
Communications série : SPI [ ]

- **SPI** : *Serial Peripheral Interface*
- Topologie « maître-esclave », pas d'adresses, bits de lecture/écriture
- Liaison avec quatre fils : **MOSI** (signal de données maître vers esclave), **MISO** (signal de données esclave vers maître), **SCK** (signal d'horloge), **CS** (sélection de l'esclave)
- *Full duplex* : communications bidirectionnelles simultanées ; multi-points
- Vitesse : en général 1 Mbps, au maximum 10 Mbps (environ 1.25 MB/seconde)



Communications sans-fil : Wi-Fi

- **Wi-Fi** : norme IEEE 802.11, réseau local sans-fil
- Topologie « point d'accès » (AP) ou « ad-hoc »
- Vitesse : de quelques MB/s à environ 1 GB/s selon la norme (802.11a/b/g/n/ac/ax)
- Portée : de quelques mètres à quelques dizaines de mètres selon l'environnement



Communications sans-fil : comparaison

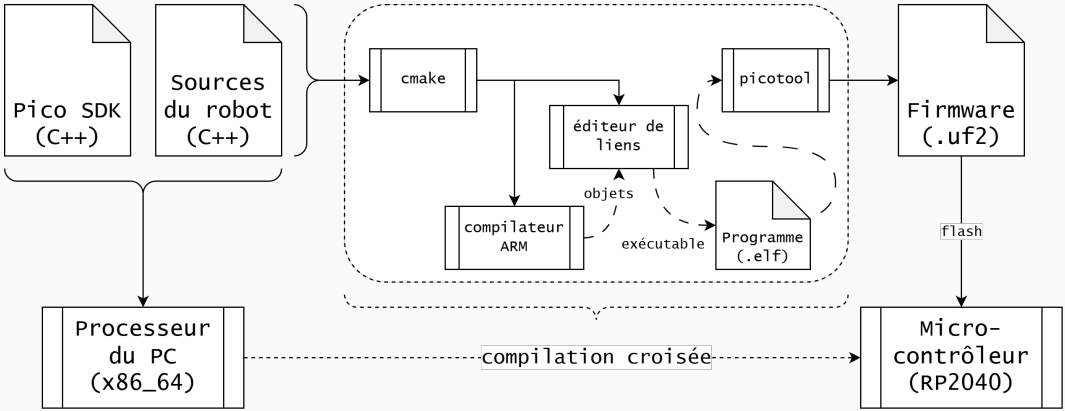
Comparaison des protocoles Zigbee, Bluetooth et Wi-Fi

Caractéristique	Zigbee	Bluetooth Low Energy	Bluetooth	Wi-Fi	Z-Wave
IEEE	802.15.4		802.15.1 (BT 1.x)	802.11a/b/g/n/ac/ax/be	
Besoins mémoire	4-32 ko		250 ko +	1 Mo +	
Autonomie avec pile	Années	Années	Mois	Jours	
Nombre de nœuds	65 000+	illimité	255	256+	232
Vitesse de transfert	20-250 kb/s	1 Mb/s	1-3 Mb/s	11-54-108-320-1000 Mb/s	9,6 à 100 kbit/s
Portée (environ)	10 m	10 m	10 m	10 m (802.11a) à > 300 m (802.11b)	50 m

Informatique embarquée

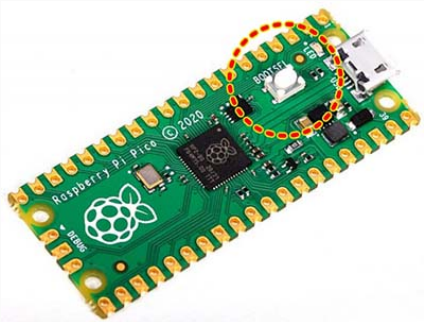
Chaîne de compilation

Chaîne de compilation croisée



Flasher le Pico : chargeur de démarrage

1. Brancher le Pico en USB en maintenant le bouton B00TSEL
2. Le Pico apparaît comme un disque USB nommé `RPI-RP2`
3. Copier le firmware compilé (fichier `.uf2`)
4. Le Pico redémarre et le nouveau programme s'exécute.



En pratique

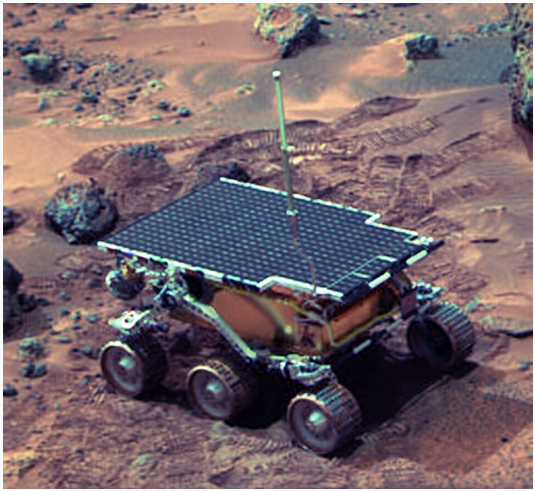
Le programme fonctionne : on peut demander l'entrée en mode BOOTSEL par USB.
S'il ne répond plus : maintenir **BOOTSEL** en rebranchant le câble, puis charger le même fichier UF2.

Robotique

Robotique

Généralités

Qu’est-ce qu’un robot ?



Sojourner sur la surface de Mars, décembre 1997 [réf.]

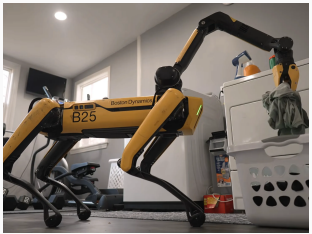
Qu’est-ce qu’un robot ?

Robot = système de systèmes

- **Mobilité** : énergie, déplacement (propulsion), orientation (direction)
- **Charge utile** : capteurs, informations, matériels, etc.
- **Intelligence** : contrôle, commande, communication, décision

Qu’est-ce qu’un robot ? – Mobilité

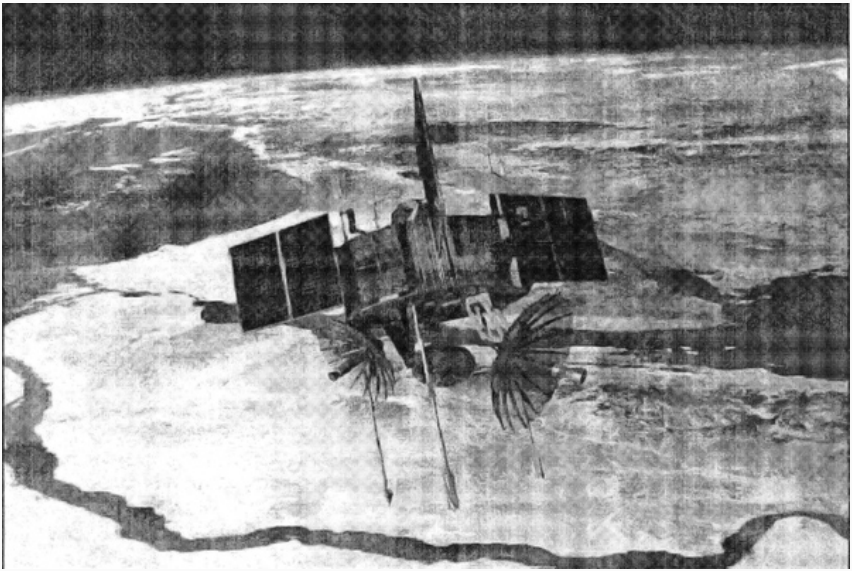








Qu’est-ce qu’un robot ? – Charge utile [ ]



Qu’est-ce qu’un robot ? – Charge utile





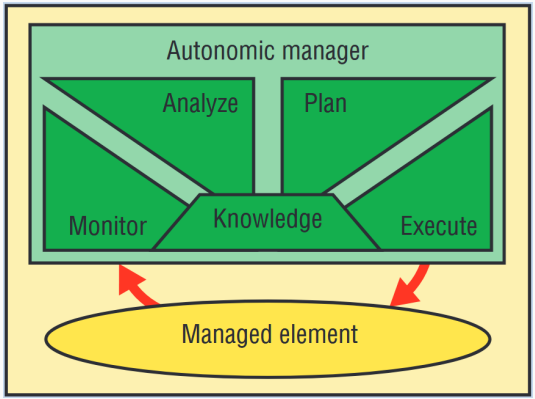
Qu’est-ce qu’un robot ? – Intelligence





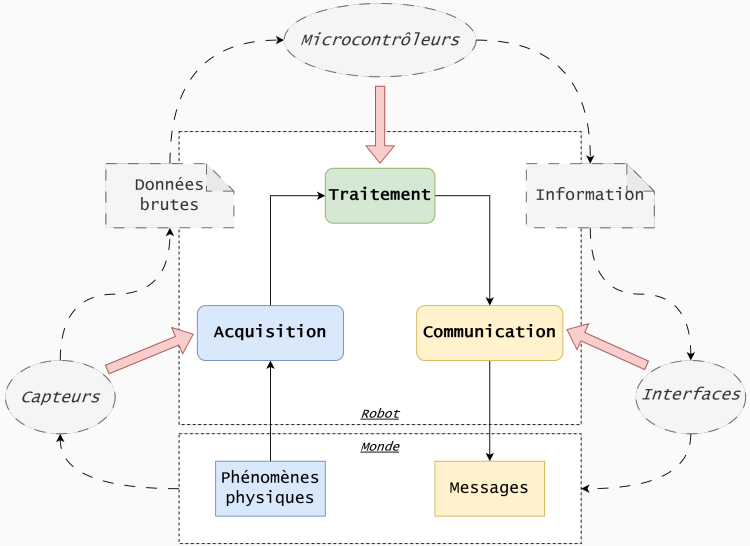
Définitions – Systèmes autonomes

- Robotique, systèmes autonomes[4] :
  - **Capteurs** : produisent une vue de l'environnement du système
  - **Actionneurs** : commandent le système selon son état et sa consigne
- Objets connectés, *Internet of Things* (*IoT*) : contrôle, commande à distance



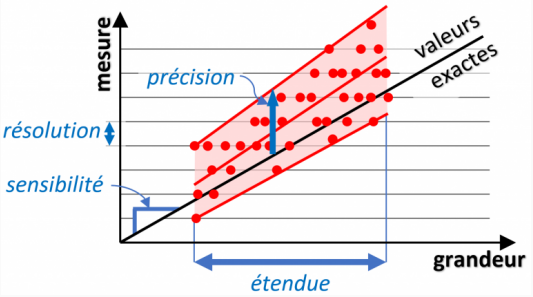
4. [4] J.O. Kephart et D.M. Chess. « The Vision of Autonomic Computing ». *Computer* (2003)

Intelligence – Chaîne de l'information



Caractéristiques d'un capteur :

- Temps de réponse
- Fréquence d'échantillonnage
- Précision, finesse, résolution, stabilité
- Coût
- Taux d'erreur (dispersion, aberrations, etc.)
- Robustesse (environnement)
- Charge de calcul induite
- Poids, consommation d'énergie, dimensions, etc.



Exemple : capteur de distance à ultrasons

distance $\rightarrow$ temps de vol d'une onde ultrasonore $\rightarrow$ durée d'une impulsion électrique $\rightarrow$ mesure du temps par le Pico $\rightarrow$ $d = v \times t/2$

- Éléments contrôlés en boucle fermée : **asservissements**
- Système automatique qui ajuste son comportement en fonction de **consignes** et de **mesures**
- Exemple :

🔁 Boucle d'asservissement

1. 🎯 **Consigne** : objectif à atteindre (ex : avancer à 10 cm du mur)
2. 📊 **Mesure** : capteur qui lit la position (GPS), la distance réelle (LiDAR), etc.
3. ❌ **Erreur** : différence entre consigne et mesure
4. 🧠 **Contrôleur** : calcule la **commande** à partir de l'erreur
5. 🤖 **Commande** : signal envoyé au système (ex : augmenter la vitesse des moteurs)

Problème : le positionnement

- **Cap** = **consigne** + **erreur**
  - Comment savoir où se trouve le robot ?
  - Comment savoir où se dirige le robot ?
  - Comment savoir où se trouve un obstacle ?



The Missile Knows Where It Is [réf.]

The missile knows where it is at all times. It knows this because it knows where it isn't. By subtracting where it is from where it isn't, or where it isn't from where it is (whichever is greater), it obtains a difference, or deviation. The guidance subsystem uses deviations to generate corrective commands to drive the missile from a position where it is to a position where it isn't, and arriving at a position where it wasn't, it now is. Consequently, the position where it is, is now the position that it wasn't, and it follows that the position that it was, is now the position that it isn't.

Systèmes de positionnement – Principes

- **Positionnement global** : localisation dans un référentiel géographique
  - **GNSS** (GPS, Galileo, etc.) : triangulation (trilatération) à partir de signaux satellites
- **Positionnement local** : estimation de la position par rapport à l'environnement
  - **Odométrie** : mesure des déplacements à partir de capteurs (ex : roues, inertie)
  - **LiDAR** : mesure des distances par laser
  - **Radar** : mesure des distances et vitesses par ondes radio
  - **Caméras** : analyse des images par vision par ordinateur
- **Localisation** : exploitation des mesures pour estimer la position
  - SLAM (*Simultaneous Localization and Mapping*) : estimer les déplacements et la position dans un environnement local

Systèmes de positionnement – Comparaison

Technologie	Avantages	Limites
GPS/GNSS	Position absolue, couverture mondiale	Peu fiable en intérieur, précision variable
LiDAR	Mesures précises, cartographie 2D/3D	Coût, sensibilité aux conditions météo
Radar	Robuste aux intempéries, mesure des vitesses	Résolution spatiale souvent inférieure au LiDAR
Caméras	Faible coût, information visuelle riche	Sensibilité à l'éclairage, calcul intensif

→ **Fusion de capteurs** : combiner plusieurs technologies pour compenser leurs limites.

Robotique

Commande à distance

Commande – Communications

- Architecture :
  - PC : Envoi de commandes filaire → USB → Robot
  - Téléphone : Envoi de commandes sans fil → Wi-Fi → Robot
- Peu importe le type de liaison, le même programme s'exécute sur le microcontrôleur
- → Il faut programmer le Raspberry pour **recevoir**, **décoder** et **exécuter** des lignes de commande.

Communications avec le robot

- Objectifs : avancer, reculer, tourner…
- Quels éléments va-t-on commander ?
- Sur la base de quelles informations ?
- $\rightarrow$ De quelles **protocole** avons-nous besoin ?

Commande – Communications – Syntaxe des commandes (JSON)

Commande	Effet
`"left":200,"right":200`	Avancer des deux côtés pendant une seconde par défaut
`"left":200,"right":-200,"duration":3000`	Tourner sur place pendant trois secondes
`"stop":true`	Arrêter immédiatement les roues et annuler la mission du robot
`"telemetry":500` / `"telemetry":0`	Envoyer la télémétrie toutes les 500 ms / désactiver la télémétrie périodique
`"mission":1` / `"mission":0`	Sélectionner la mission 1 / désactiver la mission du robot
`"info":true`	Envoyer une ligne séparée contenant les informations du robot
`"seq":7,"left":100,"right":100`	Associer l'identifiant 7 aux télémétries suivantes

L’étiquette seq

- Choisie par l’émetteur
- Chaque commande qui la fournit remplace la valeur mémorisée
- Toutes les lignes de télémétrie suivantes portent la dernière valeur

Commande – Communications – Syntaxe des réponses (JSON)

Champ	Signification
`seq`	Identifiant de la dernière commande, ou `-1` si aucun n'a été fourni
`ms`	Temps écoulé depuis le démarrage du robot, en millisecondes
`left`, `right`	Vitesses actuellement demandées aux roues, et non vitesses mesurées
`moving`	**`true`** si la vitesse demandée d'au moins une roue est non nulle
`mission`	Numéro de la mission sélectionnée ; `0` signifie qu'aucune mission n'est active
`front_mm`, `side_mm`	Dernière distance mesurée en millimètres, ou `-1` si indisponible
`front_ok`, `side_ok`	Indique si la dernière mesure du capteur est valide ; à vérifier avant d'utiliser la distance
`front_q`, `side_q`	Qualité du signal : `front_q` vaut toujours `-1` ; `side_q` indique le taux de signal du VL53L1X, ou `-1` si indisponible
`done`	Présent avec la valeur **`true`** uniquement lorsque le contrôle de sécurité arrête les roues

Robotique

Interface utilisateur

IHM – Terminal série

- **Terminal** : interface en ligne de commande
- Permet de communiquer avec le microcontrôleur via l'UART
- Le Pico expose un **port série virtuel** *via* USB
  - Ex. : `/dev/ttyACM0` sous Linux
- On peut envoyer des commandes et lire les réponses du microcontrôleur
- On utilise `screen` ou `minicom` pour ouvrir un terminal série
  - `screen /dev/ttyUSB0 115200`
    - `Ctrl-A` puis `Ctrl-D` pour quitter `screen`
  - `minicom -D /dev/ttyUSB0 -b 115200`
    - `Ctrl-A` puis `X` pour quitter `minicom`

```
root ➜ /workspaces/rubo-smol (main) $ python3 tools/pico.py console
/dev/ttyACM0 at 115200 baud -- ctrl-A ctrl-X to quit

picocom v3.1

port is        : /dev/ttyACM0
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
stopbits are   : 1
escape is      : C-a
local echo is  : yes
noinit is      : no
noreset is     : no
hangup is      : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        : lfcrlf,
omap is        : crlf,
emap is        : crcrlf,
logfile is     : none
initstring     : none
exit_after is  : not set
exit is        : no

Type [C-a] [C-h] to see available commands
Terminal ready
{"seq":-1,"ms":6282,"left":0,"right":0,"moving":false,"mission":0,"front_mm":383,"front_ok":true,
"front_q":-1,"side_mm":16,"side_ok":true,"side_q":14568}
{"seq":-1,"ms":6482,"left":0,"right":0,"moving":false,"mission":0,"front_mm":383,"front_ok":true,
"front_q":-1,"side_mm":19,"side_ok":true,"side_q":14320}
{"seq":-1,"ms":6682,"left":0,"right":0,"moving":false,"mission":0,"front_mm":383,"front_ok":true,
"front_q":-1,"side_mm":15,"side_ok":true,"side_q":14480}
{"seq":-1,"ms":6882,"left":0,"right":0,"moving":false,"mission":0,"front_mm":383,"front_ok":true,
"front_q":-1,"side_mm":19,"side_ok":true,"side_q":14392}
```


CONDUITE
vitesse
180
Maintenir pour rouler ; relâcher pour arrêter.
ÉTAT
moteur gauche 0
moteur droit 0
distance avant 483 mm
distance côté 41 mm
mission sur le robot aucune
temps de marche 148 s
requête (moy. / max) 9 ms / 31 ms
commande → télémétrie 86 ms / 181 ms
« requête » : un aller-retour HTTP, le réseau seul. « commande → télémétrie » : du bouton jusqu'au moment où la page voit que le robot a reçu la commande.
MISSIONS SUR LE ROBOT
Aucune mission nommée dans MISSION_NAMES (Robot.h). Un numéro marche quand même.
1
Lancer n°
Arrêter
Écrites dans Robot::mission() : décisions prises directement sur le robot.
MISSIONS SUR LE TÉLÉPHONE
avancer 1 étapes
aller-retour 3 étapes
vitesse 180
durée 1000 ms
arrêt à 250 mm
Écrites dans web/missions.js : décisions prises après réception de la télémétrie.
JOURNAL
11:30:09 mission « avancer » : 1 étapes
11:30:10 1/1 roues 180 180 pendant 1000 ms : vu fini après 1039 ms
11:30:10 terminée en 1040 ms
11:30:12 mission « aller-retour » : 3 étapes
11:30:13 1/3 roues 180 180 pendant 1000 ms : vu fini après 1070 ms
11:30:13 2/3 pause 300 ms
11:30:14 3/3 roues -180 -180 pendant 1000 ms : vu fini après 1103 ms

IHM – Interface web

TODO – JavaScript, Node, etc.

TODO – programmation asynchrone

Quelle différence ?

Missions firmware et *missions téléphone* ?

Robotique

Missions

Missions – Interface de programmation du robot

- **Protocole de commande** : envoyer des commandes JSON au robot et recevoir la télémétrie
- **Interfaces de programmation** :
  - fonctions C++ dans `Robot.h` pour piloter le robot « de l'intérieur »
  - fonctions JavaScript dans `robot.js` pour piloter le robot « depuis l'extérieur »
- **Missions** : séquences de commandes à exécuter sur le robot

Missions – Programmer des déplacements

- **Objectif** : programmer une séquence de mouvements
  - Avancer pendant 1 s, puis reculer pendant 1 s
  - Effectuer un quart de tour à gauche
- **Commande** : vitesse des roues gauche/droite et durée

```
{"left":200,"right":200,"duration":1000}
```

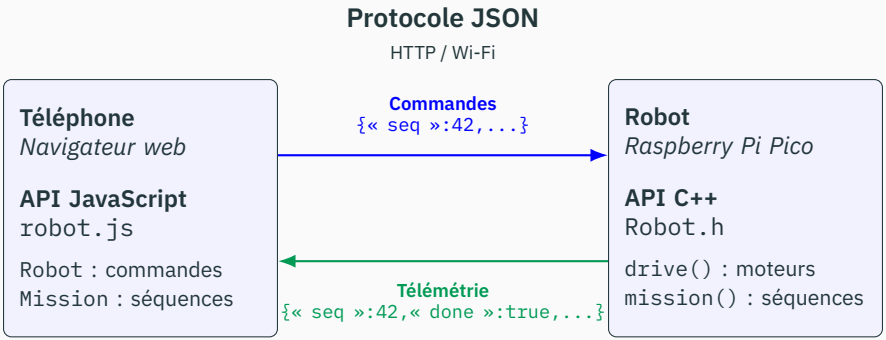
- **Enchaînement** : une nouvelle commande remplace la commande en cours
  - Attendre la ligne `done:true` avant d'envoyer le mouvement suivant
  - Le champ `seq` renvoie le numéro de la dernière commande reçue : attendre un `done` dont le `seq` est $\geq$ à celui envoyé

Missions – Deux architectures

	Téléphone (`robot.js`)	Robot (`Robot.h`)
Exécution	Navigateur du téléphone	Microcontrôleur Pico
Commandes	Envoyées successivement par HTTP	Exécutées localement
Synchronisation	Attente du message done	Surveillance de l'horloge (`nowMs()`)
Conséquence	Délais liés aux échanges réseau	Enchaînement immédiat possible

- **Téléphone** : l'API `Mission` automatise l'enchaînement des commandes.
- **Robot** : la mission s'exécute dans `Robot::update()`, sans attente bloquante (`sleep_ms()`).

Missions – API et protocole de communication



- **Deux possibilités** : séquencer les mouvements depuis le téléphone ou directement sur le robot.

Missions – Tourner de 90°

- **Principe** : faire tourner les roues en sens opposés

```
{"left":-180,"right":180,"duration":600}
```

- **Problème** : sans encodeur ni gyroscope, le robot ne mesure pas son angle de rotation.
- **Solution** : calibrer la durée du mouvement
  1. Effectuer un virage de durée connue
  2. Mesurer l'angle obtenu
  3. Ajuster la durée proportionnellement

$$t_{\text{nouveau}} = t_{\text{ancien}} \times \frac{90^\circ}{\theta_{\text{mesuré}}}$$

- **Limite** : calibration dépendante de la vitesse, du sol et du niveau de batterie.

Robotique

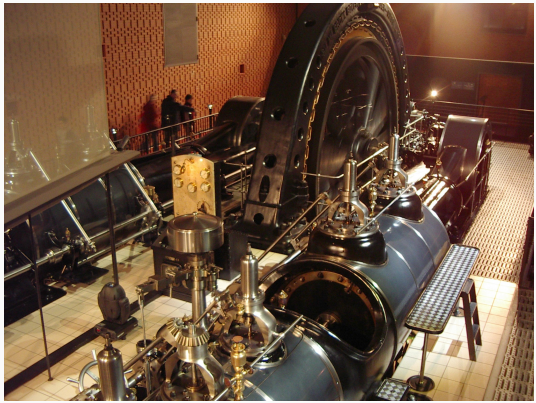
Comportement autonome et contrôle

Un constat : le besoin de calibration

- Les moteurs ne sont pas identiques
- Les roues ne sont pas identiques
- Le robot n'est pas parfaitement symétrique
- Le sol n'est pas parfaitement plat
- $\rightarrow$ Le robot ne se déplace pas *exactement comme prévu*

Comportement autonome – Origines historiques

- Sans régulation : la machine peut **accélérer dangereusement** ou **ralentir**
- Problèmes de sécurité et de fiabilité
- $\rightarrow$ Comment stabiliser la vitesse des machines à vapeur (moulins, locomotives, etc.) ?



Machine à vapeur Sulzer et alternateur Brown Boveri (Electropolis, Mulhouse)



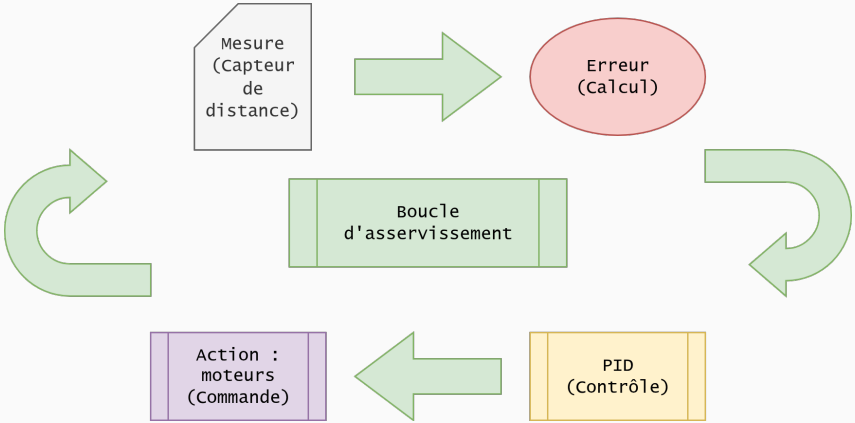
- Comment stabiliser la vitesse des machines à vapeur (moulins, locomotives, etc.) ?
- **Régulateur à boules** pour machines à vapeur (James Watt 🏴) (XVIII ème)
- Plus la machine accélère, plus les boules s'écartent, réduisant l'arrivée de vapeur → diminution de la vitesse
- Initialement proposé par Christian Huygens 🇳🇱 (XVII ème)
- Principe purement mécanique
- Limites :
  - Réponse lente
  - Pas de correction d'erreur persistante
  - Pas d'anticipation des variations rapides
  - Pas adapté aux systèmes numériques !

Comportement autonome – Théorie du contrôle

- **PID** : mis en équation en 1922 par Nicolas Minorsky 🇷🇺
- Testé sur un cuirassé de l'US Navy, l'USS New Mexico
- En pilotage automatique, en partant d'une erreur de 2 degrés, le navire a pu faire mieux qu'un barreur humain en maintenant son cap avec une précision d'1/3 de degré



Feedback loop : stabiliser automatiquement des systèmes en s'appuyant sur les comportements passés



Équation du PID en version discrète (pour une implantation numérique) :

$$u[n] = K_p \cdot e[n] + K_i \cdot \sum_{k=0}^{n} e[k] + K_d \cdot (e[n] - e[n-1]) \tag{1}$$

Équation du PID en version discrète (pour une implantation numérique) :

$$u[n] = K_p \cdot e[n] + K_i \cdot \sum_{k=0}^{n} e[k] + K_d \cdot (e[n] - e[n-1]) \tag{2}$$

- Composante p : correction proportionnelle à l'erreur
  - ❌ Problème : erreur persistante $\rightarrow$ délai de correction très long
- Composante i : correction proportionnelle à la somme des erreurs précédentes
  - ❌ Problème : instabilité de la trajectoire $\rightarrow$ oscillations
- Composante d : correction prédictive basée sur la variation de l'erreur
  - ✔️ L'erreur augmente $\rightarrow$ on « accélère » la correction
  - ✔️ L'erreur diminue $\rightarrow$ on « freine » la correction

Contrôle – Objectifs du PID

Sans PID	Avec PID
Réactions brutales	Réactions fluides
Oscillations	Stabilité
Dépassement	Précision

Robotique

Calcul déporté

Calcul déporté – Une affaire de compromis

- Il est possible de réaliser une partie des calculs sur le PC qui commande le robot
- Mais…

Avantages, inconvénients ?

- Quels sont ici les **compromis** à réaliser ?
- Comment appelle-t-on un tel système ?

Calcul déporté – Une affaire de compromis

- Compromis techniques :
  - **Temps de calcul** : PID simple adapté au Raspberry Pi Pico ?
  - **Mémoire** : quel historique des erreurs peut-on stocker ?
  - **Énergie** : gains avec le PID ?
  - **Latence** : calcul local = réponse rapide

Présentation des TP

Organisation du dépôt

- `robot/` : programme, drivers capteurs et actionneurs, sécurité
  - `main.cpp` : boucle principale du robot
  - `Robot.h` : missions embarquées
  - `Config.h` : configuration matérielle
- `robot/web/` : application web intégrée au firmware
  - `index.html` : interface utilisateur
  - `robot.js` : envoi HTTP des commandes, lecture périodique de la télémétrie, exécution des séquences
  - `missions.js` : missions exécutées sur le téléphone
- `schematics/` : schéma de câblage (Fritzing)

Deux liaisons, deux usages

- **Ordinateur $\longleftrightarrow$ Pico (USB) :**
  - chargement du programme
  - messages sur le port série
  - *en mode série*, commandes et télémétrie échangées sur ce même port
- **Téléphone $\longleftrightarrow$ Pico (WiFi) :**
  - le Pico sert la page HTML
  - le navigateur du téléphone exécute le JavaScript
  - *en mode WiFi*, commandes et télémétrie échangées en HTTP

En pratique

Le WiFi ne remplace pas complètement l'USB : toute modification du C++ *ou de la page web* nécessite de recompiler et de recharger le Pico.

Mission sur le Pico ou sur le téléphone ?

Mission sur le Pico (`Robot.h`)

- ❶ Relevé capteur
- ❷ Décision dans la boucle du Pico
- ❸ Commande des moteurs

Mission sur le téléphone (`missions.js`)

- ❶ Relevé capteur
- ❷ Télémétrie par WiFi
- ❸ Décision dans le navigateur
- ❹ Commande par WiFi
- ❺ Commande des moteurs

En pratique

Le trajet par le téléphone ajoute des délais et des risques de perte de communication.

Progression des séances de TP

TP	Thème	Objectifs
1	Découverte	Compiler, charger, piloter par USB puis WiFi
2	Missions sur le téléphone	Programmer des mouvements depuis la page
3	Interface web	Commandes et mesures échangées en HTTP
4	Deux boucles de décision	Comparer les distances d'arrêt robot / téléphone
5	Suivi de mur	Corriger la trajectoire sur le robot
6	Correcteur PID	Atténuer les oscillations du suivi de mur
7	Mission autonome	Modéliser le robot comme un système à états

Bibliographie

Références i

[1] Michael Barr, Anthony J. Massa et Michael Barr. *Programming Embedded Systems : With C and GNU Development Tools*. 2nd ed. Beijing ; Sebastopol : O'Reilly, 2006. isbn : 978-0-596-00983-0.

[2] James E. Hanson et al. « An Architectural Approach to Autonomic Computing ». *Proceedings of the First International Conference on Autonomic Computing*. ICAC '04. USA : IEEE Computer Society, mai 2004, p. 2-9. isbn : 978-0-7695-2114-5.

[3] Oliver Inderwildi et al. « The Impact of Intelligent Cyber-Physical Systems on the Decarbonization of Energy ». *Energy & Environmental Science* 13.3 (2020), p. 744-771. issn : 1754-5692, 1754-5706. doi : 10.1039/C9EE01919G.

[4] J.O. Kephart et D.M. Chess. « The Vision of Autonomic Computing ». *Computer* 36.1 (jan. 2003), p. 41-50. issn : 0018-9162. doi : 10.1109/MC.2003.1160055.

[5] Edward A. Lee. « Cyber Physical Systems : Design Challenges ». *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. Orlando, FL, USA : IEEE, mai 2008, p. 363-369. isbn : 978-0-7695-3132-8. doi : 10.1109/ISORC.2008.25.

API de programmation

Mission firmware ou mission téléphone

	Firmware	Téléphone
Code	`Robot::mission()`	`Mission.run()`, configurée par `missions.js`
Capteurs	mis à jour avant chaque appel	dernière télémétrie reçue
Réaction	arrêt dans la même passe	télémétrie, puis requête, puis commande d'arrêt
Perte de liaison	`enforceSafety()` arrête et annule	plus de commandes ; échéance et délai de liaison du robot

Les deux passent par les mêmes commandes de déplacement et les mêmes contrôles de sécurité du firmware.

API de programmation

API Robot.h

Boucle principale

`Robot::update()` s'exécute à chaque passe, dans cet ordre :

1. mise à jour des capteurs ;
2. `readCommands();`
3. `mission();`
4. `enforceSafety();`
5. télémétrie ;
6. affichage.

Rien de ce qu'elle appelle ne doit attendre : une passe bloquée retarde aussi les contrôles d'échéance et de liaison.

Se déplacer : `drive()`

```
void drive(int left, int right, uint32_t durationMs = DEFAULT_DURATION_MS);
```

- Fixe les deux vitesses, de -255 à $+255$, et arme une échéance.
- `durationMs` est plafonnée à `MAX_DURATION_MS` (5000).
- Chaque appel remplace le précédent : pour continuer, rappeler `drive()` avant l'échéance.

`enforceSafety()` arrête les roues à l'échéance ou à la perte de liaison.

- `motors.stop()` arrête tout de suite.
- Ne pas appeler `motors.set()` : un mouvement sans échéance est traité comme une faute et arrêté à la passe suivante.

Une mission firmware

```
static constexpr const char* MISSION_NAMES = "wiggle";

void mission() {
  switch (_mission) {
    case 0: break;
    case 1: wiggle(); break;
    default: break;
  }
}

bool     _wiggleLeft = false;
uint32_t _wiggleAt   = 0;

void wiggle() {
  if (nowMs() - _wiggleAt < 400) return;   // not yet
  _wiggleAt   = nowMs();
  _wiggleLeft = !_wiggleLeft;
  drive(_wiggleLeft ? -150 : 150, _wiggleLeft ? 150 : -150, 400);
}
```

Une mission rend la main à chaque appel, garde son état dans des membres, utilise `nowMs()` pour le temps et `drive()` pour bouger.

API Robot.h : une mission firmware

```
void wiggle() {
  if (nowMs() - _wiggleAt < 400) return;   // pas encore
  _wiggleAt   = nowMs();
  _wiggleLeft = !_wiggleLeft;
  drive(_wiggleLeft ? -150 : 150, _wiggleLeft ? 150 : -150, 400);
}
```

- `mission()` est appelée à chaque passe de la boucle et ne doit jamais attendre.
- Une mission est donc une machine à états : `_wiggleLeft` est l'état, `_wiggleAt` la date de la dernière transition. Chaque appel change d'état ou rend la main.
- `drive()` arme une échéance ; `enforceSafety()` arrête les roues quand elle passe.
- Les capteurs sont lus avant chaque appel : la mission peut s'arrêter dans la même passe.

Capteurs et temps

`front` (ultrasons) et `side` (VL53L1X, désactivé par défaut) sont mis à jour au début de chaque passe.

```
if (front.valid() && front.distanceMm() < 200) { ... }
```

- `valid()` : la **dernière** mesure a réussi.
- `distanceMm()` vaut -1 sauf si `valid()`.

`nowMs()` repasse à zéro après environ 49 jours :

```
if (nowMs() - startedAt > 1000)   // juste : survit au passage à zéro
if (nowMs() > startedAt + 1000)   // faux : casse au passage à zéro
```

Ajouter une commande

`execute()` reçoit chaque commande analysée comme un `JsonReader` :

`has(key)`	champ présent ? `null` vaut absent
`integer(key, fallback = 0)`	le champ en `long`
`boolean(key, fallback = false)`	le champ en `bool`

- `has()` distingue un champ absent d'un zéro.
- `execute()` ne doit pas attendre.
- Pour répondre, suivre `sendInfo()` : remplir un `JsonWriter`, puis passer son `finish()` à `_radio.send()`.

API de programmation

API robot.js

Le client Robot

```
import { Robot, Mission } from './robot.js';

const bot = new Robot();
bot.start();                          // GET /api/lines toutes les 100 ms
const seq = bot.forward(200, 1500); // POST /api/command, rend seq aussitôt
await bot.waitUntilDone(seq);
```

`drive`, `forward`, `turnLeft`…	envoient une commande, rendent son `seq` sans attendre
`stop()`, `mission(n)`, `telemetry(ms)`	`{stop}`, `{mission}`, `{telemetry}`
`last`, `info`	dernière télémétrie, dernière ligne `info`
`connected`	le robot a répondu dans la dernière seconde
`on(listener)`	appelé pour chaque ligne reçue

Attendre le robot

```
waitUntil(test, timeoutMs = 10000, aborted = null)
waitUntilDone(seq, timeoutMs = 6000, aborted = null)
```

- `waitUntil()` rend le premier message reçu après l'appel pour lequel `test(message)` est vrai, ou `null` à l'expiration ou si `aborted()` est vrai.
- `waitUntilDone(seq)` attend une ligne done dont le `seq` est **au moins** `seq` : une commande suivante remplace l'étiquette.
- Préférer `waitUntilDone()` à une attente de la durée demandée : `ms` dit quand le robot a arrêté les roues.

Une mission téléphone : `Mission`

```
new Mission('carre')
  .forward(180, 1000)
  .turnRight(180, 600)
  .repeat(4)
  .stopWhen((t) => t.front_ok && t.front_mm < 200, 'quelque chose devant')
  .run(bot);   // -> { completed, stoppedAt, reason }
```

- `run()` règle la télémétrie à 100 ms, envoie chaque étape et attend sa ligne done, au plus sa durée plus 2 s.
- Elle s'arrête plus tôt si `stopWhen` est vrai, si `aborted()` est vrai ou si une étape n'a pas de done.
- Elle laisse toujours le robot arrêté.
- `stopWhen()` teste la dernière télémétrie reçue, pas une mesure en direct.

API `robot.js` : une mission téléphone

```
const bot = new Robot();
bot.start();

await new Mission('carre')
  .forward(180, 1000)
  .turnRight(180, 600)
  .repeat(4)
  .stopWhen((t) => t.front_ok && t.front_mm < 200, 'quelque chose devant')
  .run(bot);
```

- Chaque étape envoie une commande et attend sa ligne done.
- `stopWhen()` teste la dernière télémétrie reçue : réagir demande une requête, puis une commande d'arrêt.
- Même commandes, même sécurité firmware qu'une mission firmware.