Quelques principes de programmation en automatisme :

1 - Le microcontrôleur (CPU) ne doit pas être monopolisé par une tâche.

On va donc structurer le programme comme des cycles successifs où le passage d'une étape à la suivante se fera par la mise à jour d'un registre (variable) qui va prendre en charge l'ordre d'ordonnancement des tâches pour un même processus.

2 – Une étape trop longue ou une boucle d'attente = Risque → *Considéré comme un plantage*

Par mesure de précaution on utilise très fréquemment le WathDog au cas où il y aurait un 'plantage'. Le WatchDog Timer [(1)] est un compteur qui doit être remis à zéro régulièrement sinon il va ré-initialiser le système (reset → redémarrage).

3 – Registres d'état et registres de contrôle

Un principe de base consiste donc à utiliser une fonction spécifique dont le but est de mettre à jour l'état du système environnant et, le cas échéant, prévenir le microcontrôleur si un nouvel évènement a eu lieu.

Pour limiter la quantité de mémoire nécessaire on utilise généralement une variable que l'on va utiliser comme un **registre =>** chaque bit de cette variable aura une signification particulière. On pourra donc tester un bit de la variable pour savoir si un capteur à changé d'état et agir en conséquence.

De même, si on a modifié l'état d'un actionneur, on met à jour la variable qui correspond au registre de sortie pour en informer les autres processus qui peuvent être exécutés par le microcontrôleur (utilisation d'une ressource, configuration en mode spécifique (manuel / automatique), ...).

Exemple :
On va déclarer trois variables sur 8 bits qui vont nous servir de registres → **chaque bit à une fonction** :

Bits :		...	3	2	1	0
Registre : **capteurs**	*Etat actuel*	*lecture de l'état des broches*			fdc_open[(2)]	fdc_close
	Mémoire	*état du registre an cycle précédent*			0 / 1	0 / 1
Indicateur de changement →		*indique si une broche a changé d'état*			0 / 1	0 / 1

Bits :	...	3	2	1	0
Registre : **actionneurs** →				mot_OpenClose	mot_OnOff
				0 / 1	0 / 1

voir la fonction `def mise_a_jour_capteurs ():` sur la page suivante …

1 - Class WDT (WatchDog Timer) :
https://micropython-docs-esp32.readthedocs.io/en/esp32_doc/library/machine.WDT.html

2 – fdc → Interrupteur de fin de course *(interrupteur pour détecter si un dispositif est arrivé en butée)*

4 – Interruption → service prioritaire

Dans certains cas, des situations peuvent être considérées comme prioritaires (bouton d'arrêt d'urgence, détection de passage, …) et l'information transmise doit absolument être prise en compte aussi brève soit-elle.
On utilise alors des fonctions spécifiques dites d'interruption. Lorsqu'il y a un changement d'état électrique sur la broche reliée au capteur qui gère la fonction prioritaire. Le CPU suspend la tâche en cours et exécute la routine de gestion de l'interruption *(petite fonction de quelques lignes par exemple pour mettre un bit à 1 dans un registre (flag))* puis il reprend ensuite la tâche en cours, là où il s'était arrêté.

Exemple de code pour utiliser des registres bit à bit où chaque bit correspond à un capteur.

```
from machine import Pin
from time import sleep

fdc_close = Pin(0, Pin.IN )
fdc_open = Pin (4, Pin.IN )

capteurs = 0 # Registre d'état des capteurs
capteurs_mem = 0 # Etat des capteurs au cycle précédent
capteurs_chg = 0 # Indicateur de changement d'état

def mise_a_jour_capteurs ( ):
  global capteurs
  global capteurs_mem
  global capteurs_chg
  capteurs = (capteurs & 0xFE) | fdc_close.value ( ) # Màj du bit d'index 0
  capteurs = (capteurs & 0xFD) | ( fdc_open.value ( ) << 1 ) # Màj bit index 1
  capteurs_chg = capteurs ^ capteurs_mem # Indicatr de chgmnt d'état
  capteurs_mem = capteurs # Mémoriser l'état pour la proch. boucle
  if capteurs_chg :
    sleep ( 0.3 ) # tempo d'anti-rebond pour les interrupteurs

# ========= Boucle principale ============================
# ========= Utilisation du registre de changement d'état des capteurs :
while True:

  mise_a_jour_capteurs ( )

  if capteurs_chg & 1 and capteurs_mem & 1 :
    print("Close !")
  elif capteurs_chg & 2 and capteurs_mem & 2 :
    print("Open !")
  elif capteurs_chg :
    print("Sait pas !")
```

Temporisation d'anti-rebond :

Lors d'un appui sur un bouton poussoir ou un interrupteur, il arrive très souvent que la lamelle rebondissent sur les contacts électriques. Il s'en suit une succession très rapide de 0 et de 1.

Cet état transitoire avant que la lamelle ne soit positionnée peut être interprété comme plusieurs appuis successifs ! Avec une temporisation dite d'anti-rebond de 0.3 s, normalement, on s'affranchit de ce type de problème.

Pour information :

Ce principe de registre est très utilisé dans les capteurs numériques où les informations de configuration ou les résultats de conversion analogique numérique sont stockés dans des espaces mémoires intégrés au capteur. Chaque registre est situé à un emplacement de la mémoire.

Registre de configuration → en fonction de la valeur qu'il contient, le capteur va adopter un mode de fonctionnement particulier.

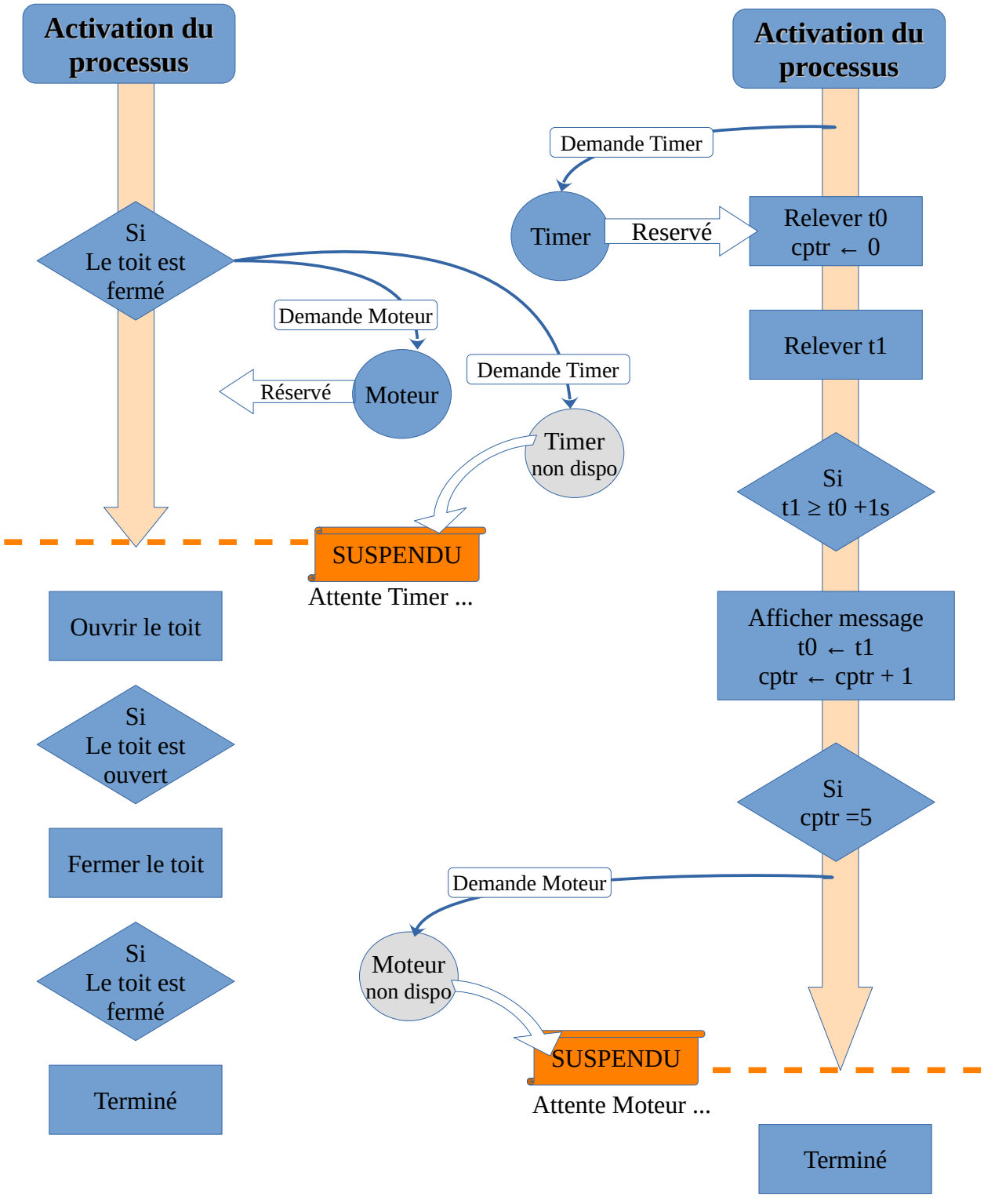
Voir les registres du capteur de température et humidité de la mini-serre : BME280 en annexe

5 – Situation d'interblocage

Lorsqu'on développe un noyau multitâches pour que le CPU puisse exécuter plusieurs tâches simultanément [3], il se peut qu'on soit confronté à une situation d'interblocage où deux processus se bloquent mutuellement.

Ci dessous est représenté l'organigramme du code en annexe. On programme deux tâches : [ouvrir_toit] et [métronome]. Ces tâches sont découpées en étapes qui vont s'exécuter les unes après les autres.

Le fait de placer un processus en attente lorsque la ressource demandée n'est pas disponible, peut provoquer une situation d'interblocage.



Exemple de capteur numérique avec une configuration bit à bit à travers des registres stockés dans un

Table 18: Memory map

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
hum_lsb	0xFE	hum_lsb<7:0>								0x00
hum_msb	0xFD	hum_msb<7:0>								0x80
temp_xlsb	0xFC	temp_xlsb<7:4>				0	0	0	0	0x00
temp_lsb	0xFB	temp_lsb<7:0>								0x00
temp_msb	0xFA	temp_msb<7:0>								0x80
press_xlsb	0xF9	press_xlsb<7:4>				0	0	0	0	0x00
press_lsb	0xF8	press_lsb<7:0>								0x00
press_msb	0xF7	press_msb<7:0>								0x80
config	0xF5	t_sb[2:0]				filter[2:0]			spi3w_en[0]	0x00
ctrl_meas	0xF4	osrs_t[2:0]				osrs_p[2:0]		mode[1:0]		0x00
status	0xF3					measuring[0]			im_update[0]	0x00
ctrl_hum	0xF2						osrs_h[2:0]			0x00
calib26..calib41	0xE1…0xF0	calibration data								individual
reset	0xE0	reset[7:0]								0x00
id	0xD0	chip_id[7:0]								0x60
calib00..calib25	0x88…0xA1	calibration data								individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Chip ID	Reset
Type:	do not change	read only	read / write	read only	read only	read only	write only

5.4 Register description

5.4.1 Register 0xD0 *"id"*

The *"id"* register contains the chip identification number chip_id[7:0], which is 0x60. This number can be read as soon as the device finished the power-on-reset.

5.4.2 Register 0xE0 *"reset"*

The *"reset"* register contains the soft reset word reset[7:0]. If the value 0xB6 is written to the register, the device is reset using the complete power-on-reset procedure. Writing other values than 0xB6 has no effect. The readout value is always 0x00.

5.4.3 Register 0xF2 *"ctrl_hum"*

The *"ctrl_hum"* register sets the humidity data acquisition options of the device. **Changes to this register only become effective after a write operation to "*ctrl_meas*".**

espace mémoire intégré au capteur. *(ici un BME280 : capteur d'humidité et de température)*

Table 19: Register 0xF2 *"ctrl_hum"*

Register 0xF2 *"ctrl_hum"*	Name	Description
Bit 2, 1, 0	osrs_h[2:0]	Controls oversampling of humidity data. See Table 20 for settings and chapter 3.4.1 for details.

Table 20: register settings *osrs_h*

osrs_h[2:0]	Humidity oversampling
000	Skipped (output set to 0x8000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, others	oversampling ×16