Cadenne Simon 1^{E4}

Présentation (Projet Surveillance Plantes)

Le projet consiste à déterminer si les facteurs vitaux d'une plante (humidité, ensoleillement, température) sont corrects à son développement, son bien-être. En effet, les plantes doivent avoir un environnement suffisamment proche de leurs besoins. Ainsi, ce système de surveillance autonome permet d'améliorer et d'adapter un environnement adéquat pour la plante.

Matériel à disposition :

- Une carte micro :bit
- Un pc avec le logiciel MU
- Un capteur d'humidité

Besoins et contraintes liées au projet

Le projet repose sur 3 principes essentiels, qui correspondent : aux 3 besoins d'une plante ; à la gestion de l'affichage ; et à un stockage des options :

Le stockage des options et paramètres : Le projet nécessite beaucoup de variables (état de l'écran, affichage des paramètres …), ainsi un dictionnaire *option* regroupe toutes les variables nécessaires :

- A l'affichage :
    1- Paramètre actuel.
    2- Liste des noms de paramètres : « Luminosité », « Température » et « Humidité ».
    3- Les fonctions correspondant à chaque paramètre (lien vers la fonction).
    4- Etat de l'écran : « allumé » ou « éteint ».

- Aux besoins de la plantes :
    1- Possibilité de prendre en compte différents types de plantes.
    2- Type de plante sélectionné.

```
option = {
    "affichage":{
            "id":0, #identifiant du paramètre en cours d'affichage
            #Liste des noms des paramètres, l'id 0 correspond à l'indice 0
            #du tableau donc à luminosité
            "option":('Luminosite', 'Temperature','Humidite'),
            #Liste des fonctions correspondantes à l'id, idem,
            #l'id 0 correspond à l'indice 0 donc à la luminosité
            "option_def":[get_lum, get_temp, get_hum],
            "ecran": False
    },
    "types_plantes":{ #dictionnaire contenant les différents types de plantes prises en charge
        "base":[(12,25),1900] #paramètres généraux pour tout types de plantes,
    },
    "types_plantes_id":0
}
```

Luminosité : La plante a besoin d'une bonne luminosité, ni trop grande ni trop faible. On doit donc la mesurer.

Cela est possible en inversant la polarisation des LEDs, on a alors une valeur comprise entre 0 et 255 (peu exhaustive). En m'appuyant sur le tableau suivant :

*Table 1: micro:bit and phone app lux values on a scale of healthy light**

Micro:bit value	Phone app lux value	Healthy light?
12	41	Too dark
17	57	Too dark
35	190	Too dark
44	230	Too dark
53	307	Still too dark
56	345	Still too dark
84	429	Still too dark
88	471	Still too dark
102	592	Light level OK (boundary level)
224	2017	Light level OK
255	5000	Light level OK

Lien du travail original : https://www.australiancurriculum.edu.au/media/6746/classroom-ideas-5-8-microbit-environmental-measurement.pdf

Une conversion de la valeur du micro :bit en LUX est possible. En utilisant les LUX, on obtient une valeur plus parlante et beaucoup plus intéressante pour une gestion de différents types de plante (voir plus bas). Le principe de fonctionnement :

Le tableau ci-dessus associe 11 valeurs issus du micro :bit à des valeurs expérimentales converties en LUX. Ainsi, lors de la mesure, le programme obtient la valeur mesurée par la carte, il prend dans ce tableau la valeur la plus proche et récupère sa conversion en LUX. Il y a un écart entre la valeur réelle et celle expérimentale, pour réduire le taux d'erreur, on utilise la proportionnalité pour équilibrer la valeur.

Cet écart existe bel et bien mais on peut remarquer que les 9 premières valeurs n'augmentent que de quelques dizaines à chaque fois tandis que les 2 dernières augmentent considérablement. Or ces deux dernières sont trop importantes pour les besoins de la plante et peu prise en compte (en général).

```
def get_lum(type_plante):

    #On supprime tout ce qui peut être afficher par la matrice
    #Car sinon nous n'aurons pas la bonne luminosité, les LEDs allumées
    #influenceront la mesure

    display.clear()
    #Récupération valeur microbit
    lum = display.read_light_level()
    #Tableau de correspondance entre (valeur microbit, valeur_lux)
    lum_correspondance = [(12, 41),(17, 57),(35,190),(44,230),(53,307),(56,345),(84,429),(88, 471), (102,592),(224,2017),(255,5000)]

    #Fonction permettant de déterminer la valeur d'une liste (liste) la plus proche d'une valeur demandée (recherche)
    def near(liste, recherche):
        diff = []
        for i in liste:
            diff.append(abs(recherche - i[0]))
        diff_sort = list(diff)
        diff_sort.sort()


        for n in range(len(diff)):
            if diff[n] == diff_sort[0]:
                return lum_correspondance[n]

    #Détermination d'une valeur plus précise par proportionnalité
    lux = lum * near(lum_correspondance, lum)[1] / near(lum_correspondance, lum)[0]

    #Récupération du seuil nécessaire, voir variable : option (plus bas)
    seuil = option["types_plantes"][type_plante][1]
    seuil_max = seuil + (seuil * 2 / 100)
    seuil_min = 250

    #On vérifie si la valeur mesurée se situe dans l'intervalle définie
    #Puis affichage de la réponse
    if lux >= seuil_min and lux <= seuil_max:
        display.show(Image.YES)
    else:
        display.show(Image.NO)
```

Température : La température est également un critère essentiel. Elle est récupérable par l'intermédiaire du micro :bit et de la température de son processeur qui ne chauffe pas énormément (une différence existe, mesure : de 4°C). En prenant compte de l'écart, on vérifie alors si la température suit les seuils requis.

```
def get_temp(type_plante):
    #On prend compte d'un écart, ici de 4
    temp = temperature() - 4

    #On récupère les seuils de température définis plus bas dans les options, voir variable : option (plus bas)
    seuil_min = option["types_plantes"][type_plante][0][0]
    seuil_max = option["types_plantes"][type_plante][0][1]

    #On vérifie si la valeur mesurée se situe dans l'intervalle définie
    #Puis affichage de la réponse
    if temp >= seuil_min and temp <= seuil_max:
        display.show(Image.YES)
    else:
        display.show(Image.NO)
```

Humidité : Pour l'humidité, la carte n'a pas la capacité de la mesurer, on a alors besoin d'un capteur d'humidité du substrat. Il suffit de le monter directement sur la carte pour mesurer l'humidité présente dans le sol : les deux électrodes conductrices sont placées dans le sol, toute eau ou humidité dans le sol entraine une modification de la résistivité du sol et donc de la résistance entre les deux électrodes. Le capteur fournit une tension analogique proportionnelle à la résistance entre les deux électrodes donc proportionnelle à l'humidité du sol. On utilise alors la fonction read_analog et on prend en considération la valeur retournée. Ainsi, d'après les essais et les données du fabricant, en dessus d'une valeur de 400, l'humidité est insuffisante.

```
"""----------------------------------------------------------------
Fonction permettant de déterminer si l'humidité est statisfaisante
----------------------------------------------------------------"""
def get_hum(type_plante):
    #On récupère la valeur du capteur
    hum = pin1.read_analog()

    #En dessous de 400, l'humidité est insuffisante
    if hum <= 400:
        display.show(Image.NO)
    else:
        display.show(Image.YES)
```

Comme vu précédemment le projet repose sur de nombreuses caractéristiques uniques à la plante, ainsi, les besoins peuvent variés.

La gestion des différents types de plantes :

(A noter : cette partie ne fonctionne qu'en partie : reste sur les valeurs de base, car il y a un problème de mémoire)

Les paramètres étant stocké dans le dictionnaire *option*, une partie est destinée à la gestion des différents types de plantes : il s'agit du dictionnaire *types_plantes.*

Fonctionnement :

Le dictionnaire contient des éléments sous forme clé : valeur avec valeur = [(temp_min, temp_max), lux_max]. L'autre partie du code s'exécute lorsque les boutons A et B sont appuyés, il change le type de plante et affiche son nom.

```
#le bouton b et a sont pressés
if button_a.is_pressed() and button_b.is_pressed():
    a = "a"
    #problème de mémoire
    # if option["types_plantes_id"] == len(list(option["types_plantes"].keys()))-1:
          option["types_plantes_id"] = 0
      else:
          option["types_plantes_id"] = option["types_plantes_id"] + 1
      display.scroll(str(option["types_plantes"][list(option["types_plantes"].keys())[option["types_plantes_id"]]]))
else:
```

Gestion de l'affichage : Tous ces points essentiels reposent sur un autre très essentiel : l'affichage. La gestion de l'affichage se découpe ainsi en 2 parties :

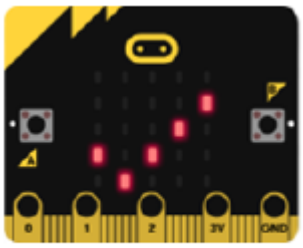
- Affichage des paramètres : Il est nécessaire de pouvoir changer de paramètres en appuyant sur le bouton b de la carte. En voici le principe de fonctionnement :

  Une première fonction gère l'affichage des paramètres et l'affichage du paramètre en cours :

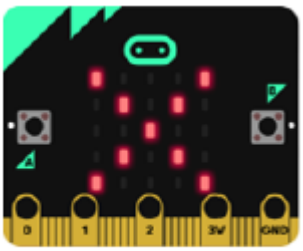
<u>Avant de continuer :</u> à chaque paramètre (« Luminosité », « Température », « Humidité ») est associé un ID. Chaque nom de paramètre est enregistré dans un tableau qui contient donc 3 éléments. L'ID de chaque paramètre correspond donc à son indice dans le tableau. Une variable contient l'ID du paramètre à afficher. Un ultime tableau contient lui un lien vers la fonction correspondant au paramètre.

```
"affichage":{
        "id":0, #identifiant du paramètre en cours d'affichage
        #Liste des noms des paramètres, l'id 0 correspond à l'indice 0
        #du tableau donc à luminosité
        "option":('Luminosite', 'Temperature','Humidite'),
        #Liste des fonctions correspondantes à l'id, idem,
        #l'id 0 correspond à l'indice 0 donc à la luminosité
        "option_def":[get_lum, get_temp, get_hum],
        "ecran": False
```

La fonction récupère l'identifiant du paramètre à afficher, cherche ensuite à partir de l'identifiant son nom. Celui-ci est affiché puis au bout de 2 secondes le résultat est affiché : est-ce que le paramètre est correct ? ou non ?



Valeur du paramètre satisfaisante



Valeur du paramètre non satisfaisante

```
def affichage_menu():
    #On récupère l'id du paramètre à afficher, chaque paramètre
    #a un id qui correspond à son indice dans le tableau (voir variable option, plus haut)
    id = option['affichage']['id']

    #On récupère le nom du paramètre
    nom_option = option['affichage']["option"][id]

    #On affiche le paramètre demandé
    display.scroll(str(nom_option), wait=False)
    sleep(300)
    sleep(9000)

    #On appelle la fonction correspondant au paramètre
    option['affichage']["option_def"][int(id)](list(option["types_plantes"].keys())[option["types_plantes_id"]])

    #On met à jour l'id pour afficher le prochain paramètre
    if id == len(option['affichage']["option"])-1:
        option['affichage']["id"] = 0
    else:
        option['affichage']['id'] = id + 1

    sleep(2000)
    display.clear()
```

- Gestion de l'état de l'écran : Deuxième point important, il est important de pouvoir éteindre ou allumer l'écran pour réduire la consommation et économiser la batterie.

  Principe de fonctionnement : Une variable booléenne prend en charge cela. Si la variable est égale à *True*, cela signifie que l'écran est actuellement allumé donc on l'éteindra. Et inversement, dans le cas de *False*.

  Cette gestion se base sur les fonctions : « display.on() » et « display.off() ». A savoir : lors du démarrage l'écran est éteint.

```
def ecran_etat():
    option["affichage"]["ecran"] = not(option["affichage"]["ecran"])

    if option["affichage"]["ecran"] == True:
        #On allume l'écran
        display.on()
        #Animation pour savoir quand l'écran s'allume
        display.show(Image.SQUARE_SMALL)
        sleep(150)
        display.show(Image.SQUARE)
        sleep(150)
        display.clear()
    else:
        #Animation pour savoir quand l'écran s'éteint
        display.show(Image.SQUARE)
        sleep(150)
        display.show(Image.SQUARE_SMALL)
        sleep(150)
        display.clear()
        #On éteind l'écran
        display.off()
```

Problèmes rencontrés :

Un des problèmes a été de choisir les bonnes valeurs pour la luminosité, en effet, les valeurs du micro :bit entre 0 et 255 n'étaient pas exhaustive, avec un flash de téléphone ou encore la lumière du jour, l'orientation de la carte, … les valeurs variés énormément. La conversion en LUX, m'a permis d'avoir une meilleure vision de la luminosité.

Un des autres problèmes a été de gérer l'affichage : les display.show() et les display.scroll() ne sont pas liés notamment au niveau de l'option *wait.* On doit alors mettre des sleep() pour découper l'exécution (pour ne pas que le résultat s'affiche : display.show() avant que le nom du paramètre ne soit affiché). Cela implique notamment d'avoir à attendre que le résultat s'affiche avant de pouvoir rappuyé sur le bouton B.

Démonstration : voir vidéo du dossier (démonstration.mp4)

Sources :

- https://www.australiancurriculum.edu.au/media/6746/classroom-ideas-5-8-microbit-environmental-measurement.pdf
- https://viedejardin.pagesperso-orange.fr/la_lumi%C3%A8re.htm#:~:text=Besoin%20minimal%20d'une%20plante%20%3A%20250%20lux&text=%2D%2015%20W%2Fm2%20en%20fluorescence%20%C3%A0%201%20m%20des%20plantes.
- https://www.kitronik.co.uk/pdf/5647-prong%20moisture-sensor-microbit-datasheet.pdf