Robot - TP4 - Embarqué - Suiveur de ligne

Installation

- Mergez la branche `tp3` dans la branche `main` et créez une branche `tp4`
- Dans le fichier `idf_component.yml` :
  - Dans le fichier `idf_component.yml`, vous devrez ajouter les dépendances suivantes :

```
espressif/asio: "*"
bblanchon/arduinojson: "^6.21.3"
```

- Pour ne pas à avoir à mettre le SSID et le mot de passe dans le dépôt git, vous devrez les ajouter en passant par le menu config. Ajoutez le fichier `Kconfig.projbuild` dans le dossier `main` avec le contenu suivant (en faisant attention à l'indentation) :

```
menu "Robot ESP IDF Configuration"

    config ESP_WIFI_SSID
        string "WiFi SSID"
        default "myssid"
        help
            SSID (network name) for the example to connect to.

    config ESP_WIFI_PASSWORD
        string "WiFi Password"
        default "mypassword"
        help
            WiFi password (WPA or WPA2) for the example to use.

endmenu
```

- Supprimez les fichiers ou dossiers générés : `git clean -ixd` (`dependencies.lock`, `build`, `sdkconfig`, `sdkconfig.old` et `managed_components`)
- Ouvrez VS code et lancez un `menuconfig` (>ESP-IDF: SDK configuration editor (menuconfig) ou icône roue crantée) :
  - Remettez les 3 paramètres du premier TP (4MB, C++ exception et C++ RTTI)
  - Ajoutez les valeurs des paramètres Wifi suivant le mode de connexion (`ESP_WIFI_SSID` et `ESP_WIFI_PASSWORD`)
  - Mettez la valeur `Single factory app(large), no OTA` pour le champ `Partition Table`. Sinon vous aurez une erreur de taille de binaire trop grande pour la partition principale.
- Lancez un build
- Vérifiez votre `.gitignore`
- Commitez cet état dans la branche `main` et poussez le sur le gitlab.

Commun

Pour simplifier la mise au point des paramètres, pour toutes les étapes suivantes, vous devrez définir le plus possible de valeur en tant que constante (constexpr) juste en dessous des includes.

Au lieu de mettre tout le code dans le fichier main.cpp, vous pouvez créer autant de fichier source qu'il vous semble utile. De plus, si vous voulez créer des classes contenant des objets non copiables (copy constructor et operator en delete) pensez que la copie de valeur n'est pas le seul moyen de copier un objet en C++.

Etape 1 : Moteur électrique

En utilisant la classe idf::BdcMotor avec comme paramètres :

- pwmFreqHz : 50 Hz
- timerResolutionHz : 500000 Hz

Faire :

- Avancer le robot à 50% de puissance pendant 1 seconde
- Puis tourner le robot sur lui-même dans le sens des aiguilles d'une montre à 25%

Etape 2 : Serveur Json RCP TCP en Wifi

1. En vous inspirant de l'exemple `esp-idf-cxx/examples/wifi_cxx` vous devrez au début du programme activer le Wifi pour pouvoir établir une connexion TCP/IP entre l'ESP et votre ordinateur.

2. En ajoutant les 2 fichiers sources de l'archive `robot-embedded-TP2-jsonrpctcpserver.tar.gz` dans le dossier main (et en ajoutant le fichier source dans le `CMakeLists.txt`, vous devrez instancier le serveur json RCP TCP puis lancez la fonction listen() qui fera office de boucle `while(true)` dans le main. Voici un exemple de code :

```
// Create an instance of Json RCP server on TCP port 6543
JsonRpcTcpServer jsonRpcTcpServer(6543);

// When a new client connect send it a jsonRCP notification
// with name "setIsReady" and empty param
jsonRpcTcpServer.bindOnConnectSendNotification("setIsReady",
        [](){return ArduinoJson::DynamicJsonDocument(10);});

// Wait and serve new client
jsonRpcTcpServer.listen();
```

   Vous devez connecter l'ESP32 et l'ordinateur sur le même réseau Wifi, il y a 3 méthodes :

   - L'ESP32 en mode AP (accès point) et le PC se connecte en Wifi à l'AP de l'ESP32.
   - L'ordinateur en mode accès point et l'ESP32 en mode station pour se connecter dessus.
   - La borne d'accès Wifi de l'armoire, ou un téléphone mobile en mode partage de connexion par le Wifi, et l'ESP32 et l'ordinateur se connecte dessus.

   Depuis l'ordinateur, si tout est bien configuré, vous devrez pouvoir recevoir le message `setIsReady` à l'aide de la commande netcat. Voici un exemple d'utilisation :

```
$ netcat 192.168.1.2 6543
{"jsonrpc":"2.0","method":"setIsReady","params":null}
```

Etape 3 : Ajout du contrôle des moteurs

Le but de cette étape est de traiter la réception des messages Json RCP de commande de puissance des moteurs `setMotorsPower` contenant les paramètres :

- `leftValue` : avec comme valeur un float entre -1.0 et 1.0
- `rightValue`: avec comme valeur un float entre -1.0 et 1.0

Il faut utiliser la fonction `bindNotification` de la classe `JsonRpcTcpServer` qui s'utilise un peu comme la méthode `bindOnConnectSendNotification` de l'étape 2 avec en argument :

- Le nom de la méthode du message que vous voulez associer
- La fonction (qui peut être une lambda) permettant de traiter ce message. Dans cette fonction vous devez avoir en paramètre les paramètres du message Json (décrit au début de cette étape).

Attention vous ne pouvez pas modifier la vitesse du moteur directement dans la lamda du `bindNotification` car il y a des problèmes de contrainte de temps réel (le module réseau de l'ESP32 lève une erreur si on ne rend pas la main rapidement). Vous devrez donc utiliser la classe `idf::Queue` pour vous poster un message (en utilisant une version de la fonction send qui ne bloque pas) et le traiter dans un thread dédié au contrôle des moteurs.

Sur le projet `robot-command` (utilisé par votre binôme) il y a une branche pour contrôler le robot au clavier. Voici les commandes pour le récupérer et vous avez dans le `README.md` les instructions pour le compiler.

```
git clone git@gitlab.cri.epita.fr:jeremie.graulle/ssie-robot-command.git
git checkout tp2-motorKeyboard
```

Vous devrez le lancer en ajoutant en 1er argument l'IP de l'ESP32 et en 2ème argument le port TCP de votre serveur Json RCP.

Si besoin, vous pouvez suivre les étapes suivantes, en testant chaque étape avec le programme robot-command :

1. Vous pouvez essayer d'afficher un message en console dans cette fonction mais il me semble que l'ESP 32 redémarre à cause de problèmes de temps réel, vous pouvez donc à la place incrémenter une variable globale, que vous afficherez dans un autre thread.
2. Vous passerez cette variable globale en variable capturée par la lambda (attention à passer une référence ou un pointeur) et pareil pour la passer dans le thread.
3. Vous convertirez cette variable en une "idf::Queue" en postant un message de type int depuis la lambda et en le lisant et en l'affichant dans le thread.
4. Vous changerez le type de message pour contenir suffisamment d'informations pour traiter la commande `setMotorsPower` avec leurs paramètres (il vaut mieux éviter de copier la partie Json mais plutôt de la convertir) et vous afficherez ces informations dans le thread de réception.
5. Vous ajouterez en plus de l'affichage l'envoi de la commande aux moteurs.

Etape 4 : Ajout du capteur suiveur de ligne

A l'aide de la fonction `JsonRpcTcpServer::sendNotification` vous devrez, depuis un thread dédié, envoyer toutes les 100ms, envoyer les 2 messages du capteur de suivi de ligne :

- `lineTrackValue` avec les valeurs :
  - `index` : l'index du capteur, ici un seul capteur donc toujours 0.
  - `value` : valeur brute lu par le capteur de 0 à 255.
  - `changedCount` : un entier qui s'auto-incrémente à chaque message.
- `lineTrackIsDetected` avec les valeurs :
  - `index` : l'index du capteur, ici un seul capteur donc toujours 0.
  - `value` : booléen vrai si la ligne est détecté et faux si non détecté (ajout du seuil en constante).
  - `changedCount` : un entier qui s'auto-incrémente à chaque message.

Attention :

- On ne peut pas instancier 2 fois la classe ADC sur la même unité. Comme on doit l'utiliser dans les 2 threads différents, il va falloir le créer qu'une seule fois et le passer en paramètre aux 2 threads.
- L'ADC est une ressource critique et si on l'utilise dans 2 threads différents il faut la protéger par un mutex. En c++ on utilise "std::mutex" qu'il faudra donc utiliser dans les 2 threads et donc également passer en paramètre aux 2 threads.
- Comme on a activé les exceptions, on ne devra pas utiliser directement la fonction "lock()" et "unlock()" de la classe "std::mutex", mais utiliser la version RAII "std::lock_guard" (voir sur cppreference il y a un exemple good et bad). Ce qu'il faut comprendre c'est que le lock se fait au moment du constructeur (déclaration de la variable) et que le unlock se fait au moment du destructeur (fin de portée de la variable c'est à dire au prochain "}").
- Comme l'ADC et "std::mutex" sont des classes non copiable (le constructeur de copie et l'opérateur d'affectation sont marqués "delete") il faudra les passer par référence et non par valeur et donc utiliser "std::ref()" dans les paramètres de la création des threads.

Vous devrez tester le contrôle de votre robot à l'aide du programme `robot-command` de l'étape 1 du TP2.

Etape 5 : Ajout des capteurs roues codeuses

A l'aide de la fonction `JsonRpcTcpServer::sendNotification` vous devrez, depuis un thread dédié, envoyer à chaque passage d'une fente de la roue codeuse en créant une interruption à l'aide de `idf::GPIOInput`, le message `encoderWheelValue` en json avec les valeurs :

- `index` : l'index du capteur, ici 0 pour droite et 1 pour gauche.
- `value` : un entier qui s'auto-incrémente à chaque message.
- `changedCount` : un entier qui s'auto-incrémente à chaque message.

Attention : comme pour l'étape 4, vous ne devrez pas envoyer le message json depuis l'interruption pour ne pas avoir des erreurs de temps réel, mais vous devrez passer par une file de message `idf::Queue`.

Vous devrez tester le contrôle de votre robot à l'aide du programme `robot-command` de l'étape 2 du TP2.