Documentation TO52

1. Mise en contexte

Dans le cadre de l'uv TO52 nous avons été amenés à travailler sur un datatlogger, ayant pour finalité de recueillir différentes données d'un véhicule (vitesse, coordonnées gps…), permettant par la suite la possibilité d'effectuer un post traitement et d'étudier les performances du véhicule en question.

Au début de notre projet, différentes consignes nous furent assignées pour le travail sur le datalogger. Elles se découpent en 3 gros points :

- La récupération des données bruts
- Le pré-traitement des données
- Développement d'un enregistrement intelligent.

La récupération des données brutes consiste à récupérer l'ensemble des trames OBD, GPS et IMU, qu'un véhicule envoie lors de son fonctionnement. A savoir que selon le véhicule utilisé, les trames envoyées peuvent différer.

Suite à la récupération de ces données, un pré-traitement est effectué, il nous sert notamment au calcul de données telles que la pente, la vitesse.. ainsi que la mise en place de l'horodatage des données. Cette étape nous sert également pour effectuer un filtrage des aberrations rencontrées.

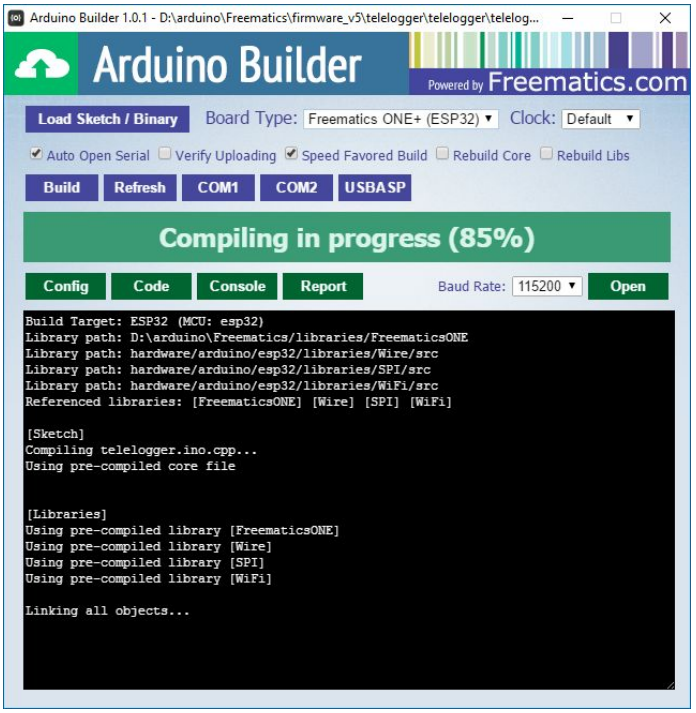
Enfin, la dernière partie concerne l'enregistrement des données sur la carte SD en fonction de la situation rencontrée. Par exemple, avec l'allègement des données enregistrées lors de situations non intéressantes pour le post traitement des données.

❖ Outils utilisés

Au commencement du projet, un outil nous a été confié, le freematic one+ model A. c'est un boitier conçu pour être branché sur la prise OBD d'un véhicule et muni d'un gps ayant pour but d'enregistrer différentes données et de les implémenter au sein d'un outil de stockage telle qu'une carte SD.

Boitier Freematic One+ Model A

Après avoir effectué différents tests de découverte sur ce boitier, l'outil s'étant démarqué comme étant le plus facile à manipuler pour utiliser ce boitier fut le logiciel freematic builder. C'est donc lui que nous avons utilisé pour la totalité de nos actions pour finaliser le projet.



Interface du logiciel freematic builder

Ce logiciel nous permet de charger un code au sein du boîtier, par la suite celui-ci utilisera la dernière version du code qui lui a été fourni pour fonctionner (à noter que certains ordinateurs nécessitent un driver particulier pour supporter la connexion au boîtier, on le retrouve directement sur le site freematics).
Freematics met également à disposition différentes ressources ayant pour but d'appréhender plus facilement leur boîtier, et de le prendre en main plus facilement. C'est sur ces ressources que nous avons basé le développement de notre projet.

2. Décomposition de l'algorithme

❖ **La récupération des données bruts**

Nous avions pour consigne de récupérer 3 grandes types de données de différentes fréquence :

- les données OBD: c'est-à-dire les différentes informations pouvant être fournies par le véhicule (vitesse, régime moteur…).
- les données GPS: Plus particulièrement la longitude, latitude et altitude, ces données sont renseignées par le gps rattaché au boîtier
- les données IMU: accéléromètre, gyroscope et magnétomètre selon les 3 axes (X, Y, Z).

Ayant chacune des fréquences voulues différentes, nous avons dû les regrouper suivant différentes tâches chargées de leur récupération.
On retrouve ci-dessous un tableau représentant les différentes données voulues, ainsi que la fréquence à laquelle elles sont reliées.

Description de la donnée	PID du OBD2	Fréquence
Vitesse	PID_SPEED, 0x0D	Fmax
Température	PID_AMBIENT_TEMP, 0x46	1éch/min
Régime moteur	PID_RPM, 0x0C	Fmax
Couple moteur	PID_ENGINE_REF_TORQUE, 0x63	Fmax
Temps depuis allumage du véhicule	PID_RUNTIME, 0x1F	1éch/min
Pression atmosphérique	PID_BAROMETRIC, 0x33	1éch/min
Niveau d'essence (pour véhicule thermique)	PID_FUEL_LEVEL, 0x2F	1éch/min
Etat de charge de la batterie (pour véhicule électrique)	PID_HYBRID_BATTERY_PERCENTAGE, 0x5B	1éch/min

Les différentes données récupérées selon l'OBD et leur fréquence associée

Concernant les données GPS et IMU, la fréquence demandée est de 1éch/100ms.

Afin de mener à bien notre projet, nous avons eu recours à l'OS Freertos. L'utilisation de cet OS nous a notamment permis d'implémenter la récupération des différentes données souhaitées selon les fréquences demandées.
Afin d'organiser les différentes tâches récupérées et de les écrire dans la carte SD, nous avons dû trouver une solution.

Pour cette étape, nous avons choisi d'opter pour une méthode de récupération de données basée sur une tâche daemon.

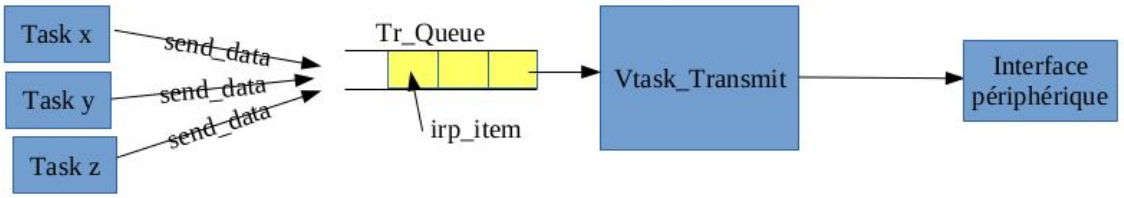


Image représentant le principe d'une tâche daemon.

La tâche daemon s'articule comme le montre le schéma ci-dessus. On récupère tout d'abord les différentes tâches de récupération des données. Chacune de ces tâches est placée par la suite dans une file d'attente, avant d'être renseignée dans la carte SD.

Récupération des données:

Nous avons au sein du code trois tâches principales récupérant les données. Deux tâches récupérant les données OBD (1ère et 3ème tâche) et une tâche récupérant les données GPS et IMU (2ème tâche).

Pour la première et la troisième tâche, on appelle la méthode OBD_looping() chargée de récupérer les données OBD à leur fréquence maximale ou minimale selon la valeur du paramètre fr_max de la fonction. Si le paramètre est à 1, on l'appelle pour récupérer des données à une fréquence maximale, si elle est à 0, on l'appelle pour la fréquence minimale.
Pour fonctionner, cette fonction prend comme paramètres: un pointeur sur un tableau de ObdInfoPid et le nombre de ses éléments et l'entier décrit auparavant, une fois l'information récupérée elle renverra un entier déterminant le temps selon lequel la tâche devrait s'endormir avant la prochaine récupération de données.

Pour la deuxième tâche, on appelle les méthodes AddGPSData() et AddMEMSData(). Ces deux dernières prennent comme paramètre: le temps actuel comme chaîne de caractères, un pointeur sur le tableau construisant le message à envoyer à la tâche daemon et un pointeur sur un entier décrivant le nombre de caractères écrits dans le tableau du message après le retour de la fonction. Les deux fonctions renvoient 1 si les données sont bien récupérées et 0 sinon. De plus, la méthode AddMEMSData() demande un 4ème argument étant la valeur de retour du AddGPSData().

```
int OBD_looping(ObdInfoPid* PIDs, int nbr_elem,int fr_max); //customized_OBD.cpp
int AddMEMSData(char* isoTime,char* p,int* nbr,int GPS_Stat); //customized_MEMS.cpp
int AddGPSData(char* isoTime,char* p, int* nbr); //customized_GPS.cpp
```

Ecriture dans la carte SD:

Une fois ces données récupérées, il faut les écrire dans la carte sd.

Au niveau du code, nous retrouvons la récupération des tâches ainsi que leur écriture dans la carte SD au niveau des fichiers “simple_OBD_test.ino” et “customized_SD.cpp”.

Pour cette étape on appelle les méthodes print_data() et appendFile(). La méthode print_data() est chargée d’insérer un message dans la queue. Une fois que le message est traité par la tâche daemon on envoie un acquittement de réception à la tâche appelante de print_data().

La méthode appendFile() (appelée par la tâche Daemon) nous permet ensuite de concaténer les différentes valeurs de données recueillies au sein du fichier courant dans la carte SD.

```
void print_data(char* data, uint32_t len);
void appendFile(fs::FS &fs, const char * path, const char * message);
```

L’écriture des données s’effectue s’effectue sous le format CSV. Ce format est caractérisé par une virgule pour un changement de colonne et un saut de ligne pour un changement de ligne. C’est directement au niveau des tâches que l’on formate cette structure, qui servira pour la création du fichier final.

- **Le pré-traitement des données**

Pour organiser les données recueillies et les structurer, il fallait ensuite ajouter l’horodatage. Un simple appel à la fonction UpdateTime permet de mettre à jour la date et l’heure actuelles, grâce au périphérique GPS. Ce qui permettra d’avoir pour chacune des tâches l’heure précise à laquelle les données sont récupérées.

```
void UpdateTime(void); //customized_DateAndTime.cpp
```

L’étape du pré-traitement des données se compose également du filtrage des aberrations GPS. Pour chaque nouvelle valeur (Latitude, Longitude ...) on fait la moyenne glissante de la grandeur GPS sur les 10 dernières valeurs déjà récupérées et sur un intervalle de 3 secondes. Par ailleurs, on superpose un deuxième filtre qui élimine les points incohérents en se basant sur la vitesse du véhicule. Ceci permet de prendre en considération seulement les points inclus dans un cercle de rayon R=V*dt, évitant ainsi l’ajout de point non cohérent.
La méthode FilterGpsData() effectue le filtrage décrit ci dessus. Elle prend comme paramètre le temps en chaîne de caractères et un pointeur sur un tableau des différentes données GPS (Latitude, Longitude ...). Elle renvoie -1 si le point est aberrant afin de ne pas le prendre en considération, 1 si c’est possible de calculer la pente selon le gps et 0 sinon.
Ceci parvient si le nombre de points est insuffisant ou les points sont trop éloignés sur l’axe de temps (plus de 3 secondes).

```
int FilterGpsData(char* isoTime, double* res); //customized_GPS.cpp
```

La pente est calculée comme suit:

pente = dela_Altitude/delta_Distance

- **Développement d’un enregistrement intelligent.**

Cette partie consistait à rendre l'enregistrement de manière à ce qu'il soit le plus optimal, et afin d'avoir des fichiers CSV moins lourds. Nous avons découpé cette partie selon différents points.

Tout d'abord, on arrête l'enregistrement des données lorsque le moteur est arrêté ou en cas d'embouteillage. Dans le premier cas, on teste si le boîtier détecte bien l'OBD du véhicule, si il n'est pas détecté alors nous n'enregistrons aucune donnée. De plus, si la vitesse du véhicule est nulle durant plus de 3 secondes, alors nous arrêtons également l'enregistrement des données dans la carte SD.
On effectue également un test sur la stabilité de la vitesse, si la vitesse du véhicule est stable pendant plus de 3 secondes, on arrête d'enregistrer les données du véhicule, ce qui permet d'éviter une saturation superflue du fichier final CSV.

La méthode init_OBD_dev() permet l'initialisation et la détection de l'OBD, c'est ainsi que l'on détecte si le moteur est allumé ou non.
Concernant l'arrêt du véhicule, on teste la vitesse via les données de l'OBD. C'est la fonction CheckObdSpeed() qui intervient. Elle prend en paramètres l'étiquette de la structure ObdInfoPid qui correspond à la vitesse, le temps actuel en chaîne de caractères, l'ancienne valeur de la vitesse et un pointeur sur un entier déterminant si la vitesse est stable ou pas.

```
void init_OBD_dev(void); //customized_OBD.cpp
void CheckObdSpeed(char* label,char* isoTime,int val, int* res) //customized_OBD.cpp
```

La deuxième partie du développement de l'enregistrement intelligent concerne la création de fichiers. En effet, nous faisons en sorte de ne créer qu'un seul fichier par trajet. On incrémente son nom, à chaque nouveau trajet. Une fois créé, le nom de fichier est enregistré dans les registres de l'RTC. A chaque réveil, on décide de créer un nouveau fichier ou chercher l'ancien en regardant le drapeau renvoyé par esp_sleep_get_wakeup_cause() dans la méthode init_SD_Card() qui prend en paramètres un pointeur sur un tableau d'ObdInfoPid et son nombre d'éléments.

```
void init_SD_Card(ObdInfoPid* PIDs, int nbr_elem); //customized_SD.cpp
```

Enfin, la dernière étape de l'enregistrement intelligent concernait la calibration de l'IMU.
Comme nous l'avons fait pour l'étape précédente, la méthode init_MEMS_dev() permet d'initialiser et de calibrer le périphérique IMU. La calibration est effectuée une fois par power cycle et donc par trajet.

```
void init_MEMS_dev(void) //customized_MEMS.cpp
```

3. Points sensibles

Durant le processus de création du projet, nous avons pu rencontrer différents points sensibles.

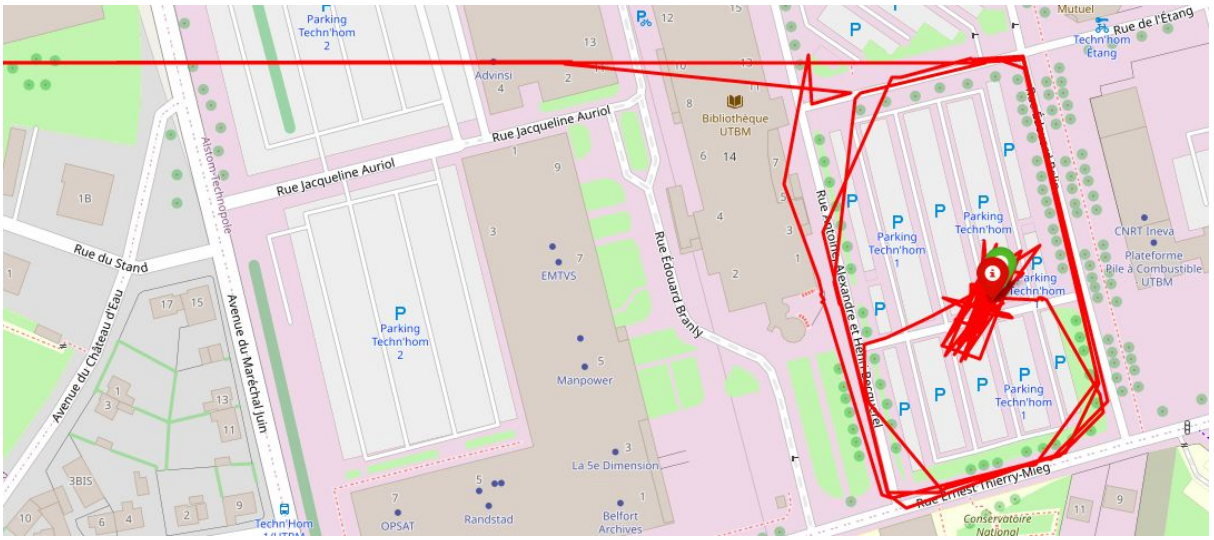
La plus grosse difficulté quant à l'utilisation du boîtier freematic, et de la création du code pour le faire fonctionner, est la capacité limitée pour déboguer le code.
Effectivement, pour pouvoir tester le programme et sa capacité de fonctionnement, il faut forcément que le boîtier soit relié à un véhicule, afin d'avoir des tests réels. Ne possédant pas de véhicules personnels, nous ne pouvions nous permettre de réaliser des tests qu'environ une fois par semaine, ce qui est peu lorsque que nous sommes face à des erreurs qui nous sont inconnues.

Le deuxième point sensible est la sensibilité du matériel. En effet, nous avons déjà été confrontés au fait que le matériel nous fasse défaut dans certaines situations. Par exemple, même si le programme fonctionne à la perfection (utilisation du gps entre autres) à un point A, il pourra rencontrer différents problèmes pour un point B en fonction de la capacité du signal reçu.

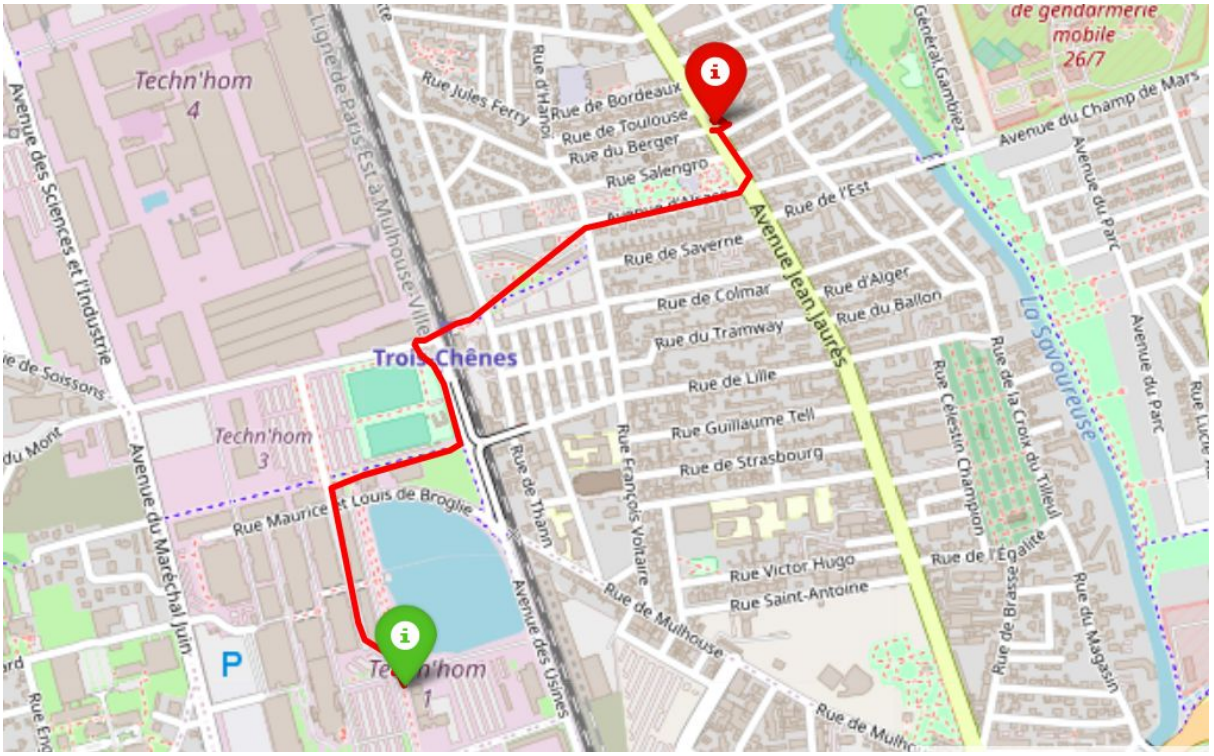
D'autre part, nous avons travaillé durant la période finale du projet, avec un câble très court, ce qui n'est pas pratique pour le débogage (câble de connexion entre le boîtier et le pc). Il nous fallait donc créer un fichier log dans la carte sd afin de déboguer le code.

Mis à part les deux gros points sensibles, le déroulement du développement se passe sans encombre une fois le processus bien intégré, et les outils bien compris.
Il faut donc faire preuve de rigueur.

4. Exemples d'exploitation des données récupérées



Exemple de trajet non filtré



Exemple de trajet filtré

5. Voies d’amélioration

Différentes perspectives d’ouverture pourraient être développées sur la base de ce projet.

Parmi elles, nous pourrions par exemple citer l’envoie des données directement sur un serveur. Ce qui permettrait ainsi d'avoir une analyse en temps réel sur les données du véhicule pour optimiser la conduite de l’utilisateur.

Nous pourrions également songer à une synchronisation des horloges désolidarisée du gps, en utilisant une connexion directement sur un téléphone par exemple.

Il pourrait également être envisageable de mettre en place un processus de type MQTT, ce qui permettrait d’obtenir un dashboard des données pour ce projet.