Recherche sur le principe d'une trame GPS

Protocole NMEA :

Dans le cas d'un GPS, les trames sont transmises en boucle via une simple transmission série à la vitesse de 4800 baud, sous la forme d'une suite de caractères ASCII, tous imprimables, ainsi que les caractères [CR] Retour Chariot et [LF] Retour à la ligne.

Chaque trame commence par le caractère $, suivi par un groupe de 2 lettres pour l'identifiant du récepteur (par exemple GP pour Global Positioning System). La virgule est utilisée pour séparer les différents champs de donnée.

Suit un groupe de 3 lettres pour l'identifiant de la trame, comme par exemple :

`GGA` : pour GPS Fix et Date,
`GLL` : pour Positionnement Géographique Longitude – Latitude,
`GSA` : pour DOP et satellites actifs,
`GSV` : pour Satellites visibles,
`VTG` : pour Direction (cap) et vitesse de déplacement (en nœuds et km/h),
`RMC` : pour données minimales exploitables spécifiques.

En queue de trame, un champ nommé *checksum*, précédé du signe `*`, représente le OU exclusif de tous les caractères compris entre `$` et `*` (exclus).

Parmi les messages émis par les GPS on rencontre les données d'acquisition du FIX (trame `GGA`) :

`123519` = Acquisition du FIX à 12:35:19 UTC
`4807.038,N` = Latitude 48°07.038′ N
`01131.324,E` = Longitude 11°31.324′ E
`1` = Fix qualification : (0 = non valide, 1 = Fix GPS, 2 = Fix DGPS)
`08` = Nombre de satellites en poursuite.
`0.9` = DOP (*horizontal dilution of position*) Dilution horizontale.
`545.4,M` = Altitude, en Mètres, au-dessus du MSL (*mean see level*) niveau moyen des Océans.
`46.9,M` = Correction de la hauteur de la géoïde en Mètres par rapport à l'ellipsoïde WGS84 (MSL).
(Champ vide) = nombre de secondes écoulées depuis la dernière mise à jour DGPS.
(Champ vide) = Identification de la station DGPS.
`*42` = *checksum*

Recherche et développement pour décoder une trame GPS

Étape 1 : Configuration du projet Qt

Assure-toi d'avoir installé Qt sur ton système. Ensuite, crée un nouveau projet Qt en utilisant Visual studio 2017 ou 2022. Sélectionne "Application Qt Console" pour ce tutoriel. (Assure toi d'avoir suivis avant le tuto port série QT sur pearltrees)

Étape 2 : Configuration de la classe de ta communication Série

Dans visual, tu vas créer une classe communication ou tu vas regrouper toutes les informations pour la communication série.

Tout d'abords il te faut inclure dans ton fichier Communication.h les bibliothèques et le code suivant

```
4   #include <QtSerialPort/QSerialPort>
5   #include <QDebug>
6   #include <QSqlDatabase>
7   #include <QSqlQuery>
8   #include <QSqlError>
9   #include <QDate>
10
11  class Communication : public QObject
12  {
13      Q_OBJECT
14  public:
15      Communication(QObject *parent = 0);
16      ~Communication();
17
18      QSerialPort portSerie; // Création du port serie
19      QString nmeaDataBuffer; // Tampon pour collecter les caractères entrants
20
21  public slots:
22      void onReadyRead(); // Lire les données reçu du port COM (GPS)
23  };
24
25
```

Suite à la page suivante :

Étape 3 : Implémentation de ta méthode lorsque le signal readyRead de ton port série est émis.

Pour cette étape, tu vas aller dans le fichier communication.cpp et dans ton slot (méthode) et copier coller ce code que j'ai expliqué ligne par ligne. (assure-toi d'avoir configurer ta classe et ton port série).

Il ne faut pas non plus que tu oublies de prendre une carte arduino et copier le code sur pearltrees afin de le mettre dans celle-ci, cela te permettra de simuler l'envoie de trames et donc de tester ton code en allant.

```
QByteArray newData = portSerie.readAll(); // On prend tout les données dispo
nmeaDataBuffer += QString(newData); // On additionne les données reçues

while (nmeaDataBuffer.contains("$GPGGA"))
{
    int start = nmeaDataBuffer.indexOf("$GPGGA"); // Recherche le début d'une trame
    nmeaDataBuffer = nmeaDataBuffer.mid(start); // Éliminer tout avant la trame

    int end = nmeaDataBuffer.indexOf("\r\n"); // Rechercher un séparateur de ligne (fin d'une trame)
    if (end != -1)  // Si c'est pas la fin
    {
        QString nmeaString = nmeaDataBuffer.left(end); // On va prendre tout le reste de la trane à  part de end donneés
        nmeaDataBuffer = nmeaDataBuffer.mid(end + 2); // Éliminer la trame traitée et le séparateur de ligne (\r\n)

        // Maintenant, nmeaString contient une trame GPS complète
        QStringList fields = nmeaString.split(','); // Séparer toutes les infos avec une virgule

        // La taille de tout les données qui nous intéressent
        if (fields.size() >= 15) {
            QString time = fields[1]; // Heure (HHMMSS)

            QString latitude = fields[2]; // Latitude (DDMM.MMMM)
            QString latitudeDirection = fields[3]; // Direction de la latitude (N/S)

            QString longitude = fields[4]; // Longitude (DDDMM.MMMM)
            QString longitudeDirection = fields[5]; // Direction de la longitude (E/W)

            // Ensuite on les affiches dans la console
            qDebug() << "longitude : " << longitude << longitudeDirection;
            qDebug() << "latitude : " << latitude << latitudeDirection;
            qDebug() << "heure : " << time << endl;
        }
    }
    else
    {
        // La trame n'est pas encore complète, attendez plus de données.
        break;
    }
}
```

Étape 4 : Créer ton objet afin de créer la communication et recevoir les données sur ton terminal (QT console) lorsque tu lances ton programme.

Pour cela, il te faut aller dans ton fichier ou il y a ta fonction main et ajouter cette ligne entre les deux déjà présente (oublier d'inclure ton fichier .h).

```
Communication Com1;
```

Recherche pour enregistrer en base des données (Application C++)

Étape 1 : Configuration du projet Qt

Pour cela tu peux reprendre la classe et le projet tu avais créer sur le tuto pour décoder les trames GPS. Il va juste te falloir aller dans les propriétés de ton projet et ajouter dans QT Project Setting “sql”

Étape 2 : Configuration de la base de données

Dans visual, tu peux utiliser la classe QSqlDatabase pour gérer la base de données. Voici comment configurer une base de données SQLite (bien que tu puisses utiliser d'autres bases de données également).

Tout d’abords il te faut inclure dans ton fichier .h les bibliothèques suivantes :

```
#include <QSqlDatabase>
#include <QSqlQuery>
```

Ensuite, il faut que tu déclares ton objet BDD dans la classe qu’il y a dans ton .h (dans la section private):

```
QSqlDatabase db; // Base de donnée
```

Il faut que tu ailles maintenant dans le fichier de déclaration .cpp de ta classe (dans le constructeur), et tu devras initier ton objet BDD, ensuite tu le configures avec les informations pour se connecter à la base de donnée et enfin tu fais une condition pour vérifier si tu as bien réussi à te connecter ou non.

```
// [Connexion BDD]
database = QSqlDatabase::addDatabase("QMYSQL");

database.setHostName("127.0.0.1");
database.setUserName("root");
database.setPassword("");
database.setDatabaseName("lawrence");

if (database.open())
{
    qDebug() << "Connexion reussie a " + QString::fromStdString(database.hostName().toStdString());
}
else
{
    qDebug() <<  "Probleme de connexion a " + QString::fromStdString(database.hostName().toStdString());
}
```

Suite à la page suivante :

Étape 3 : Insérer des données dans la base de données

Pour insérer des données dans la base de données, tu peux utiliser des requêtes SQL. Voici un exemple d'insertion de données dans une table :

```
QSqlQuery query;
query.prepare("INSERT INTO ma_table (colonne1, colonne2) VALUES (:valeur1, :valeur2)");
query.bindValue(":valeur1", "Valeur1");
query.bindValue(":valeur2", "Valeur2");

if (query.exec()) {
    qDebug() << "Données insérées avec succès !";
} else {
    qDebug() << "Échec de l'insertion : " << query.lastError().text();
}
```

Étape 4 : Gestion des erreurs et de la fermeture

Il est important de gérer les erreurs lors de l'interaction avec la base de données. Utilise query.lastError() pour obtenir des informations sur les erreurs potentielles. N'oublie pas de fermer la base de données lorsque tu as terminé en appelant db.close().