Rapport Mini Projet Arduino

ArdyWeather

UE : Architecture des Ordinateurs

Auteur : Axel PIGEON

Université : INU Champollion

Date : May 4, 2025

Rapport dans le cadre d'un projet de développement en Arduino ayant pour but l'initiation au développement embarqué et la communication entre différentes architectures (Arduino, RaspberryPI, etc...)

Contents

Introduction

Ici est présenté le projet d'Architecture des Ordinateurs 2. Le projet consiste à développer un outil ou un jeu utilisant des connexion arduino/arduino ou arduino/raspberry.
Pour celui-ci, nous allons essayer de faire une station météo composée de deux modules :

- **Le capteur :** composé d'une carte arduino munie de capteurs (ex : température, humidité...) et d'un émetteur/récepteur radio pour envoyer les relevés à la station fixe. Pour le capteur, on utilise un capteur de température et d'humidité Grove permettant une connexion facile à la carte.
- **La station fixe :** Composée d'un arduino recevant les données émises par le capteur et les sauvegarde.

Toutes les informations techniques relatives au projet (code, bibliothèques) sont disponibles sur le Github :

https://github.com/winston2968/ArdyWeather.git

L'objectif est de programmer la station pour qu'elle puisse interragir avec plusieurs stations.

1 Présentation et Principe général

Le projet dispose de deux modules, le `main_station` qui gère le protocole radio côté station et la gestion des données reçues par les capteurs et le `main_sensor` qui gère le protocole radio côté capteur ainsi que la récolte et l'encodage des données. Attardons nous sur le protocole radio.

1.1 Structure des paquets

La bibliothèque `VirtualWire` permet d'envoyer et de recevoir des données via un signal radio. Malgré sa rusticité, elle est capable de différencier le signal radio émis par l'Arduino de celui émis par une autre Arduino. Autrement dit, on ne peut pas recevoir les messages que l'on envoie. Cette bibliothèque ne permet d'envoyer "que" 27 octets par émissions. Nous devons donc dimensionner les datagrammes en fonction. Dans un datagramme (ou paquet) nous devons donc envoyer :

- **Une séquence propre au protocole** pour différencier nos paquets d'autres émissions ambiantes. (*Nous choisirons pour cela deux caractères en début de datagramme : 'A' et 'W'.*)
- **Un identifiant unique pour l'émeteur** du paquet. (*Un nombre entier correspondant au numéro du module. La station (qui reçoit les données des capteurs) aura le numéro '0'.*)
- **Un identifiant unique pour le destinataire** du paquet. (*Idem que pour les émetteurs.*)
- **Le type du paquet** soit des données ou simplement un paquet d'acquittement. (*On prendra 'D' pour un paquet de données et 'A' pour une trame d'acquittement.*)
- **Le type des données envoyées** le cas échéant, pour différencier les données de température et d'humidité. (*De même que précédement, on choisira 'T' pour température ou 'H' pour humidité.*)
- **Le nombre de paquets déjà envoyés** : pour permettre de détecter un problème de réception. (*Un entier encodé avec son caractèse ASCII.*)
- **Le nombre de paquets reçus** : pour informer l'émetteur de la bonne réception des paquets. (*Un entier encodé avec son caractèse ASCII.*)

- **Deux octets de checksum** : pour être capable de détecter des paquets détériorés. (*Calculés à l'aide d'une simple somme modulaire.*)
- **Les données** : température, humidité ou rien si c'est un paquet d'acquittement. (*Encodées d'une certaine façon que nous détaillerons plus tard.*)

Ainsi, un datagramme classique aura donc le format suivant :

```
{'A', 'W', Emetteur, Destinataire, Type de Paquet, Type de Données, Nb
                Envoyés, Nb ACK, CHKSM1, CHKSM2, Datas}
```

Par exemple le paquet `'A','W','1','0','D','H','1','0',...` , est un datagramme envoyé par le module 1 à la station (module 0). C'est un paquet de données d'humidité. Le paquet porte le numéro 1 (donc la station devra acquitter avec une trame d'acquittement et un numéro ACK de 1). On peut voir que la station n'a envoyé aucun paquet au module numéro 1. Les octets de checksum sont envoyés directement sous forme binaire sans formattage ASCII. Cela ne permet pas de les visualiser dans le moniteur série. On peut voir, qu'une fois l'en-tête du paquet remplie, il ne nous reste "que" 16 octets pour les données dans un datagramme.

1.2 Encodage des données

Les données de température et d'humidité relevées par les capteurs des modules sont des nombres flottant. Dans le module relevant les données, on les stocke donc sous forme de tableau de flottant, `float temp_table[]` et `float hum_table[]` dans le code. Pour des données d'humidité et de température, nous n'avons besoin d'une précision à seulement deux décimales. Or les flottant sont encodés sur 8 octets en Arduino. Si on envoyait les flottant directement dans les datagrammes, il nous faudrait envoyer un nombre conséquent de paquets dont le champ de données serait à moitié vide.
Ainsi, nous avons choisis d'encoder ces nombres flottant sur seulement 3 octets. Pour cela, nous les multiplierons par 100 pour obtenir des entiers. Entier, que l'on encodera ensuite sur 3 octets, chaque octet définis par des opérations arithmétiques (voir fonction `convert_float_table_to_ASCII` du fichier `sensors.ino` du module `main_sensor`). Ces opérations ne nous permettent que d'envoyer les flottant entre 0 et 85.54. Donc on ne peut pas relever des températures négatives...

1.3 Principe et fonctionnement

L'émission des datagrammes de données sont déclenchés par les **capteurs** et réceptionnés par la **station**.

Côté capteur, celui-ci relève en permanence la température et l'humidité de son environnement et les stocke dans deux tables `temp_table` et `hum_table` comme énoncé précédement. Une fois ces tables suffisament remplies (on a choisis un nombre arbitraire de 64 relevés correspondant à 24 émissions radio en tout), le capteurs déclenche la séquence d'envoi de données à la station. Pour cela, il dispose de la fonction `update_temp_hum` déclenchée par un timer qui relève les données en permanence et la fonction `build_merged_table` qui permet de construire la `char merged_table[]`. Cette table de données permet de faire un tampon entre l'émission radio et le stockage des données en cours de relevé.
En effet, la séquence d'envoi peut être longue en fonction du temps que met la station à recevoir et acquitter les datagrammes. Il est donc indispensable de libérer les tables de relevés au cours de cet intervalle pour permettre de continuer à enregistrer la température et l'humidité.
Ainsi, le capteur dispose de suffisament de temps (exactement 24 fois l'intervalle de relevés) pour envoyer les données. Passé ce temps là, la `merged_table` est écrasée par les nouvelles données à envoyer.

L'envoi des datagrammes est déclenché par un timer. A chaque activation, le capteur vérifie dans la fonction `send_datagram` s'il est en cours de séquence d'envoi et si tous les datagrammes précédents ont étés acquittés. Si tel est le cas, il récupère les données à envoyer en fonction de la valeur de `nb_tables_stack` pointant sur l'indice de départ de la séquence de `merged_table` à envoyer. Dans le cas contraire (si le précédent datagramme n'a pas été acquitté), il le renvoie via la variable `char last_datagram[]` actualisée à chaque envoi de nouveau datagramme. A noter qu'il n'est pas possible d'avoir envoyé plusieurs trames non acquittées, puisque le capteur attend toujours l'acquittement de la précédente pour envoyer la suivante.

Côté station, celle-ci attend en permanence la réception de datagrammes via son module radio. Une fois un datagramme réceptionné correspondant au protocole développé, elle vérifie que les octets des champs checksum correspondent bien au calcul de la trame reçue. En cas d'égalité, elle acquitte le datagramme à l'émetteur et sauvegarde les données dans `temp_storage` et `hum_storage` en fonction de leur type.

1.4 Fonctions Principales du protocole

Détaillons les principales fonctions du protocole décris ci-dessus.

```
void build_datagram(char destination, char datagram_type, char datas_type) {
    // We will send next datagram
    if (datagram_type == 'D') {
        nb_send += 1;
    }
    if (nb_send == 30) {
        nb_send = 0;
    }

    // Re-init datagram values
    memset(datagram, 0, sizeof(datagram));
    // Define datagram header
    datagram[0] = 'A';
    datagram[1] = 'W';
    datagram[2] = destination;
    datagram[3] = MODULE_NUM;
    datagram[4] = datagram_type;
    datagram[5] = datas_type;
    datagram[6] = '0' + nb_send;
    datagram[7] = '0' + nb_ack;
    // Fill-in datas
    Serial.print("---| Tables Stack : ");
    Serial.println(nb_tables_stack);
    for (int i = 0; i < 16; i++) {
        datagram[i + 10] = merged_table[i + (16 * (nb_tables_stack - 1))];
    }
    // Calculate checksum
    uint32_t sum = 0;
    for (int i = 0; i < datagram_size; i++) {
        sum += (uint8_t) datagram[i];
    }
    uint16_t result = (uint16_t)(sum & 0xFFFF) ;
    datagram[8] = (result >> 8) & 0xFF;
    datagram[9] = result & 0xFF;
    }
```

Cette fonction, présente dans chaque modules (station ou capteurs) permet de construire un datagramme, soit pour l'envoi de données ou l'acquittement d'une trame de données. Un datagramme est stocké dans la variable globale `char datagram[]`. Dans cette fonction, on réduit le compteur de paquets envoyés (et donc d'acquittement) à 30 pour ne pas avoir d'entier trop grand et donc dépasser l'octet alloué dans le datagramme. On réinitialise ensuite le datagramme à envoyer que l'on remplit avec les données d'en-tête. Enfin, en fonction du type de datagramme on remplit ou pas le champ de données. A noter que la valeurs du checksum est calculée sans les octets de checksum dans l'en-tête.

```
void send_datagram() {
    Serial.println("------- Sending Datagram Process Lauched -------");
    // Get/build datagram to send
    if (nb_send == nb_ack_other) {
        // We can send a new datagram
        if (nb_tables_stack < 24) {
            build_datagram(DEST_NUM, 'D', 'T');
        } else {
            build_datagram(DEST_NUM, 'D', 'H');
        }
        Serial.println("---| Sending a new datagram");
    } else {
        // We need to re-send last datagram
        memcpy(datagram, last_datagram, datagram_size);
        Serial.println("Re-sending last datagram");
    }
    // Send datagram
    vw_send((uint8_t *)datagram, datagram_size);
```

```
19      vw_wait_tx();
20      // Keep datagram on buffer
21      memcpy(last_datagram, datagram, datagram_size);
22      Serial.println("---| Datagram Send Succesfully !");
23      }
```

Cette fonction appelée à chaque déclenchement du Timer permet d'initialiser la séquence d'envoi d'un datagramme. En fonction de l'acquittement du datagramme précédent, on construit un nouveau datagramme ou pas. Ici, on sauvegarde bien entendu le datagramme envoyé en cas de non acquittement.

```
1  void convert_float_table_to_ASCII(float table[4][16], char* dest) {
2    int cpt = 0;
3    for (int i = 0; i < 4; i++) {
4      for (int j = 0; j < datas_size; j++) {
5        int value = (int) (table[i][j] * 100);     // Two decimals precision
6
7        char c1 = (value / 91) + 33 ;
8        char c2 = ((value % 91) / 10) + 33;
9        char c3 = (value % 10) + 33 ;
10
11       dest[cpt++] = c1;
12       dest[cpt++] = c2;
13       dest[cpt++] = c3;
14     }
15   }
16 }
```

Ici, on convertit les données de température et d'humidité initialement en `float` en trois caractères ASCII pour l'envoi par signal radio. Cette fonction est utilisée par `build_merged_table`.

2 Développement

Dans cette partie, nous allons détailler brièvement le processus de développement du projet ainsi que les outils utilisés.

2.1 Structure et choix

Tout d'abord, l'utilisation d'Arduino de type MEGA disposant d'un mémoire RAM de 8ko a grandement facilité la gestion de la mémoire pour le stockage d'importantes quantités de données de température et d'humidité. Dans un second temps, le choix de ces modules radio assez peu perfectionnés nous a poussé à repenser complètement un protocole de communication fiable et solide. L'acquittement de chaque datagramme de données n'étant pas la solution la plus optimale, elle reste quand même la plus fiable et simple à implémenter. L'ajout de checksum rajoute une couche de sécurité supplémentaire permettant d'isoler tout paquet endommagé. Enfin, l'ajout d'un dispositif de chiffrement des datagrammes aurait pu être une bonne idée mais les données envoyées n'étant pas sensibles, ce n'était pas une priorité du projet.

Côté techniques et outils de développement, le logiciel ArduinoIDE nous a permis de développer le projet de A à Z. Sa fiabilité et sa facilité d'utilisation en a fait un outil de choix. De plus, sa capacité à inclure plusieurs fichiers dans un même sketch (projet à téléverser dans l'Arduino) a été pratique pour fragmenter le code et ne pas avoir à traiter un fichier de plus de 1000 lignes. Pour ce qui est de la gestion des différentes versions du projet, nous avons utilisé GitHub pour le versionnage et la sauvegarde. Le projet est d'ailleurs toujours disponible sur le site via le lien en introduction.

2.2 Améliorations

Même si le projet final reste assez complet, il est nécessaire d'aborder les différentes fonctionnalités manquantes. Tout d'abord, un lecteur avisé du code remarquera rapidement que les capteurs de température et d'humidité ne sont pas utilisés. En effet, il nous a été impossible de les faire marquer avec la bibliothèque proposée par le fabriquant. Nous avons donc choisi de générer aléatoirement des valeurs pour simuler des relevés et pouvoir tester la gestion des données.

D'autre part, le protocole de communication entre modules n'est pas des plus perfectionnés. Il serait plus optimal que les capteurs soient capables d'envoyer plusieurs datagrammes de suite et de recevoir une seule trame d'acquittement pour tous les paquets envoyés. Cela réduirait beaucoup le flux de données en transit et le nombre d'acquittements à envoyer pour la station principale.

Enfin, il aurait été préférable d'ajouter une option de visualisation des données via un lien station/raspberry pi pour un affichage sur ordinateur. L'ajout d'afficheurs sur les modules ou la station aurait été agréable pour être capable de visualiser ce que font les micro-contrôleurs en temps réel.

2.3 Processus de Test

Dans le GitHub, plusieurs fichiers permettent de tester les différentes fonctionnalités du projet. Tout d'abord, les sketchs `main_sensor` et `main_station` contiennent tout le code des différents modules du projet. Il sont fonctionnels et leur lancement sur deux cartes ArduinoMEGA permettront d'avoir une bonne vision de ce qui est possible. De plus, l'ajout d'un sketch supplémentaire : `testing_station` permet de tester uniquement le module station pour vérifier qu'il est bien capable de recevoir des datagrammes de différents autres modules.

3 Conclusion

Il y a plusieurs choses à dire pour cette conclusion. Tout d'abord, le développement de A à Z d'un protocole radio pour Arduino nous permis de nous poser des questions fondamentales en télécommunication. Questions concernant la gestion de l'acquittement des trames, de la fiabilité via les checksum ainsi que la gestion de la temporisation entre envoyer des données et écouter les datagrammes reçus. La mise en place de toutes ces contraintes nous a poussé à revoir de nombreuses fois la structure des datagrammes, allouant une place importante pour l'en-tête de ces derniers au mépris des données. Enfin, le fait de devoir programmer des choses relativement complexes sur des micro-contrôleurs aux capacités limitées, a rendu le développement plus intéressant. Le langage C, à la base de l'Arduino est un langage parfait pour ce genre de tâches, il permet de manipuler finement les variables ainsi que la gestion de la mémoire.