DESCRIPTION DU CODE MASTER.

F1SSF 12/01/2026

Analyse Globale du Système

Avant de plonger dans les détails de chaque fichier, je vais vous expliquer simplement ce que fait l'ensemble de ce code Arduino, en tenant compte que vous n'êtes pas un expert. Ce système est un "contrôleur maître" (master Controller) pour piloter un rotor (un dispositif qui tourne sur deux axes : azimut horizontal et élévation verticale). Il gère aussi jusqu'à 8 relais (des interrupteurs électroniques pour activer/désactiver des choses comme des moteurs ou des lumières). L'interface utilisateur est un écran tactile Nextion (HMI pour Human-Machine Interface), et la communication avec un module distant (remote) se fait via RS485 (un protocole robuste pour les liaisons série longues distances).

Fonctionnement global :

- Le maître lit des mesures analogiques (positions AZ et EL) du remote via RS485.
- Il gère les commandes des boutons de l'écran Nextion (ex. : UP, DOWN, CW pour clockwise, CCW pour counter-clockwise, STOP, et les relais).
- Il synchronise l'affichage sur l'écran (positions, statuts, couleurs des boutons relais).
- Il permet une calibration des positions AZ/EL, stockée en EEPROM (mémoire non volatile de l'Arduino).
- Les relais fonctionnent en mode "exclusif" : activer un relais désactive les autres, et un toggle sur un relais déjà ON le force à OFF.
- Le code est modulaire : séparé en fichiers pour la config, la gestion RS485, la gestion HMI, et le fichier principal .ino.

Points forts :

- Bien structuré avec des includes et des fonctions inline pour l'efficacité.
- Gestion de la synchronisation entre maître et remote via RS485 (envoi de commandes et parsing des réponses).
- Support pour une conversion en degrés (optionnel via ENABLE_DEG).
- Calibration persistante en EEPROM.

Points à surveiller (sans proposer de mods pour l'instant) :

- Délais fixes (delay(1)) dans RS485 : ça peut causer des problèmes si la communication est lente.
- Buffers fixes (ex. buf[64]) : risque de débordement si les messages sont plus longs.
- Pas de gestion d'erreurs avancée (ex. timeouts sur RS485).
- Les valeurs min/max par défaut (az_raw_min=4, etc.) pourraient ne pas correspondre à votre hardware ; la calibration est là pour ça.

Maintenant, analysons chaque fichier en détail, ligne par ligne où c'est pertinent. Je vais expliquer ce que fait le code, pourquoi c'est fait comme ça, et signaler les pièges classiques pour un non-expert.

1. Analyse de rs485_manager.h

Ce fichier gère la communication RS485 : envoi de commandes au remote et parsing des réponses reçues.

- **#pragma once et #include "config.h"** : Empêche les inclusions multiples et importe les configs (ex. pin RS485_EN=8).
- **static inline void rs485_tx_on() et rs485_rx_on()** : Active le mode transmission (TX) ou réception (RX) en mettant le pin RS485_EN à HIGH ou LOW. "Inline" rend ça rapide, comme une macro.
- **void rs485Send(const char *cmd)** : Envoie une commande au remote.
    - Passe en TX, attend 1ms (delay(1) pour stabiliser).
    - Envoie le cmd suivi de "\r\n" via Serial1 (le port RS485).
    - Flush pour vider le buffer, attend 1ms, repasse en RX.
    - **Explication** : RS485 est half-duplex (pas en même temps TX et RX), d'où le switch. Piège : si le remote est lent, les delays pourraient ne pas suffire ; testez avec un oscilloscope si possible.
- **bool parseStateLine(const char *line, uint8_t &idx, int &val, bool &hasVal)** : Parse une ligne comme "STATE,6" ou "STATE,6,1".
    - Vérifie si ça commence par "STATE,".
    - Extrait l'index (1-8), et optionnellement une valeur (0/1).
    - **Explication** : Utilise strncmp, atoi, strchr pour parser simplement. Retourne false si invalide. Bon pour la sync des relais.
- **void readRS485Debug()** : Lit les données de Serial1 et les parse.
    - Buffer statique buf[64] pour stocker une ligne.
    - Lit char par char : ignore \r\n pour terminer la ligne, stocke les printable (32-126).
    - Affiche en debug via Serial.
    - **Parsing spécifique :**
        - Si "STATE,<n>[,<0/1>]", met à jour relay_state et relay_dirty.
            - **Correction ajoutée** : Mode exclusif. Si relais OFF → ON et éteint les autres. Si déjà ON → OFF (arrêt implicite).
            - Met à jour status_str comme "STATE RL<n>".
        - Si "ANA,<az>,<el>", extrait az_raw et el_raw avec atoi et strchr.
            - Si ENABLE_DEG=1, convertit en degrés avec map et constrain (de 0-360 pour AZ, 0-90 pour EL).
    - **Explication** : C'est le cœur de la réception. Statique pour garder l'état entre appels. Piège : si lignes >63 chars, ça reset (idx=0) ; augmentez la taille si besoin.

2. Analyse de config.h

Fichier de configuration : définit des constantes et déclare des variables globales.

- **#pragma once** : Comme avant.
- **#define RS485_EN 8** : Pin pour enable RS485.
- **EEPROM defs** : Magic number pour vérifier si données valides, adresses pour stocker min/max AZ/EL.
- **Externs** : Déclare des variables partagées (ex. bool up, dw... pour boutons ; az_raw, el_raw pour mesures ; relay_state[9] et relay_dirty[9] pour relais).
- **#define ENABLE_DEG 1** : Active la conversion en degrés (modifiable à 0 pour raw values).

- **Valeurs par défaut** : az_raw_min=4, max=1023, etc. (typique pour analogRead 0-1023).
- **Couleurs Nextion** : Définit RGB565 pour boutons relais (ON rouge, OFF gris, texte blanc/noir).
- **Explication** : Tout est centralisé ici pour facile modification. Piège : Les tableaux relay_*[9] ignorent index 0 (1-8 utilisés) ; c'est courant mais attention aux boucles.

3. Analyse de hmi_manager.h

Gère l'interface Nextion : lecture des commandes HMI, envoi de mises à jour, calibration.

- **#pragma once et includes`** : Importe config et rs485.
- **Fonctions EEPROM** : Save et load calibration (met à jour min/max en mémoire).
- **void handleHMI(const String &cmd)** : Traite les commandes de l'écran (appelée plus bas).
- **Variables** : up/dw/cw/ccw pour boutons ; relay_state/dirty comme dans config.
- **hmiSendCmd(String &s)** : Envoie commande Nextion avec terminaison 0xFF x3.
- **hmiApplyRelayVisual(uint8_t idx)** : Met à jour un bouton relais sur l'écran (val, bco pour fond, pco pour texte). Assume noms comme "bRL1".
- **Fonctions pour relais** : _isDigitStr vérifie si string est numérique ; _parseRelayIndexToken extrait index de "RL1" ou "bRL1".
    - **sendRelayCmd(int idx, char action)** : Envoie "OUT,RL,<idx>,<action>" via RS485.
        - **Correction** : Si toggle sur ON, force OFF immédiat.
        - Met à jour status_str.
    - **handleRelayHMI(String &cmd)** : Parse "RL,<n>,<action>" ou "BTN,RL1,<action>".
        - Mappe actions (ON= '1', etc.) et appelle sendRelayCmd.
- **void readHMI()** : Lit Serial3 (Nextion), parse frames entre < >.
    - Accumule dans frame, appelle handleHMI à la fin.
- **void handleHMI(const String &cmd)** (suite) :
    - Gère relais via handleRelayHMI.
    - Boutons : UP/DW/CW/CCW/STOP → met à jour états, status_str, envoie "OUT,<dir>,<1/0>" via RS485.
        - STOP : Reset tout, force relais OFF et dirty=true.
    - Calibration : "CAL,AZ,MIN" etc. → met à jour min/max, cal_status_str.
        - SAVE : appelle eepromSave ; ABORT : juste statut.
- **void hmiUpdateValues()** : Rafraîchit l'écran toutes les boucles.
    - Envoie AZ/EL (degrés ou raw), status, cal_status via Serial3.print.
    - Met à jour gauges n0/n2.
    - Applique visuels relais si dirty.
- **Explication** : Nextion envoie des frames <CMD> ; ce code les parse et réagit. Piège : String peut consommer mémoire ; utilisez char[] si RAM faible.

4. Analyse de master_controller.ino

Fichier principal : setup et loop.

- **Includes** : Config, rs485, hmi.
- **const char* status_str = "READY";** : Statut initial.

- **Variables** : az_raw, el_raw.
- **void setup()** :
  - Pin RS485_EN en output, passe en RX.
  - Init Serial (debug 115200), Serial3 (Nextion 115200), Serial1 (RS485 9600).
  - Affiche "MASTER READY".
  - Charge calibration EEPROM, met cal_status_str.
- **void loop()** :
  - Toutes les 500ms : envoie "REQ,ANA" via RS485 pour mesures.
  - Appelle readHMI(), readRS485Debug(), hmiUpdateValues().
- **Explication** : Boucle simple, polling-based. Pas de interrupts, ce qui est OK pour ce projet. Piège : Si loop bloque (ex. delays), l'écran peut sembler figé ; optimisez si besoin.

DESCRIPTION DU CODE REMOTE.

Je vais décomposer fichier par fichier, en expliquant ce que fait chaque partie, comment ça s'articule avec le **MASTER**, et en signalant les points forts, les points d'attention et les comportements importants (surtout pour quelqu'un qui n'est pas expert).

1. config.h (côté remote)

C++

```
#define RS485_EN 8


#define RELAY_UP   4
#define RELAY_DW   5
#define RELAY_CW   6
#define RELAY_CCW  7


#define RELAY_RL1 22
#define RELAY_RL2 23
#define RELAY_RL3 24
#define RELAY_RL4 25
#define RELAY_RL5 26
#define RELAY_RL6 27
```

```
#define RELAY_RL7 28
#define RELAY_RL8 29

extern String rxLine;
```

Ce qu'on observe :

- Pin RS485_EN = 8 → identique au master (bien)
- Relais moteurs sur pins 4,5,6,7 (UP, DW, CW, CCW)
- Relais supplémentaires (logiques) sur pins 22 à 29 → probablement une carte Mega ou équivalent (car Uno n'a pas autant de pins numériques)
- extern String rxLine; → la ligne de commande RS485 reçue est partagée entre plusieurs fichiers

Point important : Les relais sont en **logique inversée** (actif à LOW) → c'est très courant avec les modules relais 5 V low-level trigger.

2. rs485_manager.h (remote)

C++

```
static inline void rs485_tx_on() { digitalWrite(RS485_EN, HIGH); }
static inline void rs485_rx_on() { digitalWrite(RS485_EN, LOW);  }

void rs485SendLine(const String &s) {
  rs485_tx_on();
  delay(1);
  Serial1.print(s);
  Serial1.print("\r\n");
  Serial1.flush();
  delay(5);
  rs485_rx_on();
}
```

Différences avec le master :

- delay après flush passé à **5 ms** (au lieu de 1 ms sur le master)
- Utilise String au lieu de const char*

Comportement :

- Le remote prend un peu plus de temps pour repasser en réception (5 ms vs 1 ms)
- Ça peut être utile si le bus RS485 est long ou bruyant, mais ça ralentit un peu les échanges

3. relay_manager.h (cœur de la logique relais)

C’est ici que se trouve la logique la plus intéressante.

Variables globales

C++

```
String rxLine = "";


const uint8_t rlPins[8] = {22,23,24,25,26,27,28,29};


bool rlState[8] = {0};
```

Fonctions importantes

1. notifyRlState(uint8_t idx)

   Envoie au master : "RL,STATE,<numéro>" → **idx = 0** signifie « aucun relais actif » → **idx = 1 à 8** signifie « ce relais est ON »

2. allRelaysOff()

   Met **HIGH** tous les relais moteurs **et** les 8 relais logiques → remet rlState[] à false → très sécuritaire au démarrage

3. setRelayExclusive(uint8_t idx, bool on)
   - Met **tous** les relais à OFF (HIGH)
   - Met uniquement le relais demandé à ON (LOW) si on == true
   - Met à jour rlState[]
   - Envoie la notification au master
4. toggleRelayExclusive(uint8_t idx)

   Inverse l’état du relais demandé → appelle setRelayExclusive

Gestion des commandes (handleCmd)

C++

```
if (cmd == "REQ,ANA") → renvoie ANA,xxx,yyy


if (cmd == "OUT,ALL,0") → tout éteint + notify 0


// Moteurs (classique anti-conflit UP/DW et CW/CCW)
if (cmd == "OUT,UP,1")   { digitalWrite(RELAY_DW, HIGH);
digitalWrite(RELAY_UP, LOW);   }


// Relais logiques
if (cmd.startsWith("OUT,RL,")) {
```

```
    uint8_t idx = ... .toInt() - 1;   // transforme 1..8 → 0..7
    String act = ...

    if (act == "1")  setRelayExclusive(idx, true);
    if (act == "0")  allRelaysOff(); notify(0);
    if (act == "T")  toggleRelayExclusive(idx);
}
```

Comportement global des relais logiques :

- Mode **exclusif** forcé côté remote
- Quand on active un relais → tous les autres s'éteignent
- Quand on envoie "0" → tout s'éteint
- Quand on toggle un relais déjà allumé → il s'éteint (et les autres restent éteints)

C'est cohérent avec la logique que tu avais mise côté master.

4. remote_controller.ino

C++

```
void setup() {
  pinMode(RS485_EN, OUTPUT);
  rs485_rx_on();

  pinMode des relais moteurs + RL1..RL8
  digitalWrite tous les RL → HIGH (éteints)

  allRelaysOff();   // redondant mais très sécuritaire

  Serial.begin(115200);     // debug USB
  Serial1.begin(9600);      // RS485
}

void loop() {
  while (Serial1.available()) {
    char c = Serial1.read();

    if (c == '\r' || c == '\n') {
      if (rxLine.length()) {
        handleCmd(rxLine);
        rxLine = "";
```

```
      }
    }
    else if (c >= 32 && c <= 126) {
      rxLine += c;
      if (rxLine.length() > 40) rxLine = "";   // protection anti-
débordement
    }
  }
}
```

Points positifs :

- Lecture caractère par caractère → robuste
- Protection contre trames trop longues (>40 caractères)
- allRelaysOff() au démarrage → sécurité

Points d'attention :

- Pas de timeout ni de watchdog → si le master envoie n'importe quoi en continu, le remote reste bloqué dans le while(Serial1.available())
- Pas de gestion des erreurs de trame (ex. trame incomplète, checksum, etc.)

Synthèse – Comportement attendu du couple Master + Remote

Action depuis l'écran Nextion (master)	Commande RS485 envoyée	Comportement remote	Réponse RS485 au master
Appui sur RL3	OUT,RL,3,T	toggle exclusif du relais 24	RL,STATE,3
Appui à nouveau sur RL3	OUT,RL,3,T	éteint le relais 24	RL,STATE,0
Appui sur RL5	OUT,RL,5,T	allume RL5, éteint tous les autres	RL,STATE,5
Bouton STOP	OUT,ALL,0	éteint tout (moteurs + relais logiques)	RL,STATE,0
Demande analogique toutes les 500 ms	REQ,ANA	lit A0 & A1, renvoie ANA,xxx,yyy	ANA,512,768 (exemple)