# BZO PDU v1.0 — Guide du firmware pour intervenir (humain ou IA)

Guide du firmware - novembre 2026

> **À qui s'adresse ce document** : à toute personne, ou à son assistant IA, qui doit modifier ou étendre le firmware de la BZO PDU v1.0. Fournissez ce fichier **et** le dossier `Code_PDU_v1.0` complet à votre assistant : il y trouvera tout ce qu'il faut pour intervenir sans poser de questions de base. Vous n'avez qu'à décrire votre projet.
>
> **Consigne pour l'IA** : lisez ce guide en entier avant de toucher au code, puis respectez la section « Conventions » à la lettre. Le code doit rester lisible par un électronicien qui code en Arduino : pas d'astuces de langage.

## 1. La carte en bref

La BZO PDU v1.0 est un distributeur de puissance 12 V à 16 voies, piloté par un processeur principal (ce firmware).

| Bloc | Rôle côté firmware |
|---|---|
| Processeur principal | ce firmware |
| Voies 1 à 12 | sorties « petites puissances », mesure de courant jusqu'à ~11 A |
| Voies 13 à 16 | sorties « fortes puissances », mesure de courant jusqu'à ~67 A |
| Mesures analogiques | processeur de mesure (esclave ADC) : envoie 19 mesures 8 bits au processeur principal par liaison série ( `SerialADC` , PA3/PA2, 115200 bauds) |
| Surveillance d'alimentation | superviseur d'alim : coupe la logique si la ligne `MCU_OK` (PA13) tombe |
| Entrées | 16 entrées logiques, niveau actif réglable (masse ou +12 V) |
| LEDs de voie | LED bicolore par voie : **vert** = piloté directement par la sortie de puissance ; **rouge** = piloté par le processeur principal |
| LED status | D1 (PB2) : état de fonctionnement |
| CAN | bibliothèque `STM32_CAN` (pazi88) |
| Liaison PC | `SerialTS` (PA10/PA9, déclaré dans le `.ino` ) : TunerStudio via la bibliothèque `BZO_TS.h` . Ne pas utiliser `Serial` : sur cette carte il tombe sur PA3/PA2, la liaison ADC |

**Chaîne d'alimentation logique** : le +12 V passe par un interrupteur commandé par le superviseur d'alim, puis par les régulateurs 5 V et 3,3 V. Si le processeur principal met `MCU_OK` à LOW, le superviseur d'alim coupe toute la logique pendant 2 s, puis relance. Au démarrage, le processeur principal doit remonter `MCU_OK` en moins de 2 s (c'est la première ligne de `setup()` ).

**Processeur de mesure et superviseur d'alim** : ils ont leur propre code (hors de ce dossier), sont programmés avant montage et **ne doivent pas avoir besoin d'être reprogrammés** : tout ce qui est réglable se fait dans ce firmware.

## 2. Outils

- Arduino IDE **1.8.19**, core **STM32 2.4.0**.
- Flash de la carte : logiciel **BZO Programmer**, mode **« Protocole BZO générique »**.
- Bibliothèques : `STM32_CAN` (pazi88), `EEPROM` (fournie par le core), `BZO_TS.h` (fournie dans le dossier).
- TunerStudio avec le fichier `BZO_PDU_v1.0.ini` (protocole CRC imposé par le `.ini` , rien à régler).
- Les réglages d'une carte neuve sont tous à 0 : on charge une **base tune (.msq)** dans TunerStudio. Il n'y a pas de valeurs par défaut codées dans le firmware.

## 3. Les fichiers et leur ordre

L'ordre des `#include` dans le `.ino` est **obligatoire** : chaque fichier utilise ce qui a été déclaré avant lui.

| Ordre | Fichier | Contenu |
|---|---|---|
| 1 | `BZO_TS.h` | bibliothèque de communication TunerStudio (pages de réglages, temps réel, commandes, sauvegarde). **Ne pas modifier** pour un projet : elle est commune à plusieurs cartes. |
| 2 | `globals.h` | macros `BIT_SET / BIT_CLEAR / BIT_CHECK`, structures de réglages `config1` (page 1) et `config2` (page 2), structure temps réel `statuses` (`currentStatus`), liste des octets envoyés à TS (`getTSLogEntry()`) |
| 3 | `Outputs_management.h` | broches des 16 sorties, modes des sorties, **fonction mère** `ManageOutputs()` |
| 4 | `TS_Commands.h` | boutons TunerStudio : mode test, ON/OFF manuels, effacement d'erreur, réarmement des fusibles |
| 5 | `DigitalReadings.h` | lecture et filtrage des 16 entrées |
| 6 | `AnalogReadings.h` | réception des mesures du processeur de mesure et conversion en unités réelles |
| 7 | `Security.h` | seuils absolus, fusibles par voie, LEDs rouges, LED status, mémoire de la dernière erreur, ligne `MCU_OK` |
| 8 | `CAN_Commands.h` | bac à sable CAN : modes, ID des trames, réception, émission, sniffer |
| — | `Code_PDU_v1.0.ino` | `setup()` et `loop()` uniquement |
| — | `BZO_PDU_v1.0.ini` | fichier TunerStudio (doit rester synchronisé avec `globals.h`) |

### Boucle principale (`loop()`)

```
TS.update();        // communication TunerStudio (+ mise à jour de secl, statutTS, erreursTS, boucles)
ReadDigitals();     // entrées -> currentStatus.DigitalStatus1/2
ReadAnalogs();      // mesures -> courants, températures, tension (+ newADC = true à chaque trame)
CheckSecurity();    // fusibles, seuils absolus, LED status (calculs une fois par trame ADC, ~50 ms)
ReadCAN();          // une trame CAN par tour maximum
ManageOutputs();    // fonction mère : décide de l'état de chaque voie
SendCAN();          // trames de sortie CAN en continu (si Broadcast)
```

## 4. Les données

### 4.1 Unités utilisées partout

| Grandeur | Unité stockée | Exemple |
|---|---|---|
| Courant | 0,1 A (`uint8_t` voies 1-12, `uint16_t` voies 13-16) | 35 = 3,5 A |
| Température carte | °C (0 si négatif) | 72 = 72 °C |
| Tension +12P | 0,1 V (`uint8_t`, bloquée à 25,5 V) | 138 = 13,8 V |
| Seuil de fusible | A | 5 = 5 A |
| Temps de fusible | 0,1 s | 20 = 2 s |
| Temporisation de voie | 0,5 s | 20 = 10 s |
| Période PWM | 0,2 s | 5 = 1 s |
| Rapport cyclique | % | 50 = 50 % |

## 4.2 Page 1 — `configPage1` (`struct config1`, 25 octets)

| Offset | Champ | Rôle |
|---|---|---|
| 0 | `DigitalLevel1` | niveau actif des entrées 1-8 (1 bit par entrée) — lu au démarrage seulement |
| 1 | `DigitalLevel2` | niveau actif des entrées 9-16 |
| 2 | `CANspeed` | 0 désactivé, 1 = 125k, 2 = 250k, 3 = 500k, 4 = 1M — lu au démarrage seulement |
| 3 | `CANMode` | trames d'entrée : 0 = Demande/Réponse, 1 = Broadcast, 2+ = modes ajoutés dans le code |
| 4 | `CANTimeout` | sécurité CAN en s (0 = désactivée) |
| 5 | `CANOutMode` | trames de sortie : 0 = Demande/Réponse, 1 = Broadcast |
| 6 | `CANTxRate` | fréquence des trames de sortie en Broadcast (Hz) |
| 7-22 | `Fuse_ch1` à `Fuse_ch16` | seuil de fusible de chaque voie (A, 0 = désactivé) |
| 23 | `FuseDelay` | temps toléré au-dessus du seuil avant coupure (0,1 s) |
| 24 | `FuseRearm` | réarmement automatique après coupure (0,1 s, 0 = pas de réarmement auto) |

## 4.3 Page 2 — `configPage2` (`struct config2`, 104 octets)

| Offset | Champ | Rôle |
|---|---|---|
| 0-15 | `OutMode[16]` | mode de chaque voie (voir §5) |
| 16-31 | `OutSource[16]` | bits 0-5 : 0 = aucune, 1-16 = entrée physique, 17-32 = entrée CAN virtuelle 1-16 ; bit 7 : sens inversé |
| 32-47 | `OutTimeout[16]` | coupure au bout de (0,5 s, 0 = sans limite) |
| 48-63 | `OutPWMPeriod[16]` | période PWM (0,2 s, 0 = allumage fixe) |
| 64-79 | `OutPWMDuty[16]` | rapport cyclique (%) |
| 80-87 | `GroupInput[8]` | entrée qui active le groupe (0 = aucune, 1-16 = entrée physique, 17-32 = entrée CAN virtuelle) |
| 88-95 | `GroupOutA[8]` | 1re voie du groupe (0 = aucune) |
| 96-103 | `GroupOutB[8]` | 2e voie du groupe (0 = aucune) |

Dans les tableaux, **la case 0 correspond à la voie 1** : `configPage2.OutMode[voie - 1]`.

## 4.4 Temps réel — `currentStatus` et log TunerStudio (156 octets)

`struct statuses` est libre : ce qui part vers TunerStudio est décidé octet par octet dans `getTSLogEntry()` (`globals.h`). Le numéro en commentaire de chaque champ est son offset dans le log.

| Offset | Valeur |
|---|---|
| 0 | `secl` — secondes depuis le démarrage (rempli par la bibliothèque) |
| 1 | `statutTS` — bit0 mémoire OK, bit1 sauvegarde en attente, bit2 protocole CRC, bit3 EEPROM trop petite, bit4 RAM insuffisante (rempli par la bibliothèque) |
| 2 | `erreursTS` — erreurs de communication (rempli par la bibliothèque) |
| 3 | `TestMode` — 0 normal, 1 mode test |
| 4-5 | `boucles` — tours de `loop()` par seconde (rempli par la bibliothèque) |
| 6-7 | `DigitalStatus1/2` — entrées actives (1 bit par entrée) |
| 8-19 | `Current_ch1` à `ch12` |
| 20-27 | `Current_ch13` à `ch16` (2 octets chacun) |
| 28-29 | `BoardTemp1/2` |
| 30 | `Battery10` |
| 31 | `SafetyStatus` — 0 normal, 1 proche d'un seuil absolu (≥ 80 %, sauf batterie), 2 seuil dépassé |
| 32 | `LastError` — 0 aucune, 1-16 courant de la voie N, 17-18 BoardTemp 1/2, 19 batterie |
| 33-34 | `LastErrorValue` — valeur mesurée au moment de l'erreur |
| 35-36 | `FuseStatus1/2` — fusibles coupés (1 bit par voie) |
| 37-38 | `OutputStatus1/2` — état réel des voies (1 bit par voie) |
| 39-40 | `CANInputs1/2` — entrées CAN virtuelles |
| 41 | `CANGroups` — groupes activés par CAN |
| 42-43 | `CANTraffic` — trames CAN reçues par seconde |
| 44-153 | `CANSniff[10]` — 10 dernières trames CAN reçues, 11 octets chacune (ID 2, longueur 1, données 8), la plus récente en premier |
| 154 | `SniffPause` — 1 = sniffer en pause (affichage figé) |
| 155 | `CANError` — 1 = trame refusée sur un ID BZO (tenu 2 s) |

Les valeurs brutes de l'ADC (`IntensityADC_chN`, `BoardTempADC1/2`, `BatteryADC`) existent dans `currentStatus` mais ne sont pas envoyées à TS.

## 5. La fonction mère des sorties (`Outputs_management.h`)

`ManageOutputs()` est appelée à chaque tour. Elle ne fait **rien** si `currentStatus.TestMode == 1`. Sinon, elle appelle `DriveChannel(voie, broche)` pour chacune des 16 voies.

Pour chaque voie, `DriveChannel()` :

1. calcule la **demande** : la source de la voie est active (inversée si le bit 7 de `OutSource` est à 1), OU un groupe actif contient la voie ;
2. applique le **mode** (`switch`) ;
3. applique la **temporisation** (sauf en ON et OFF) ;
4. force OFF si le **fusible** de la voie est coupé ;
5. écrit la broche et met à jour `OutputStatus1/2`.

### Modes existants

| Valeur | Nom | Comportement |
|---|---|---|
| 0 | `MODE_OFF` | toujours coupée |
| 1 | `MODE_ON` | toujours allumée (seul le fusible peut la couper) |
| 2 | `MODE_INPUT` | suit la demande |
| 3 | `MODE_LATCH` | un appui de plus de `LATCH_PRESS` (100 ms) bascule ON/OFF |
| 4 | `MODE_PWM` | tant que la demande est active : clignote à `OutPWMPeriod` / `OutPWMDuty`. Calé sur `millis()`, donc deux voies réglées pareil clignotent **en phase** (clignotants + warning) |
| 5 | `MODE_LATCH_PWM` | un appui lance le clignotement, l'appui suivant l'arrête |
| 20+ | `MODE_PERSO_x` | réservés aux fonctions personnalisées |

### Temporisation ( `TimeoutReached()` )

Quand la voie est « en service » (demande active en INPUT/PWM, état ON en LATCH) depuis plus de `OutTimeout × 0,5 s`, elle est coupée quel que soit l'état de l'entrée. Elle ne repart qu'après une nouvelle demande : la source repasse inactive puis active, ou un nouvel appui en LATCH (qui revient à OFF).

### Groupes (« commandes multiples »)

8 groupes. Un groupe est actif si son entrée `GroupInput` est active **ou** si son bit est à 1 dans `CANGroups`. Un groupe actif fait une demande sur ses voies `GroupOutA` et `GroupOutB`. Exemple : warning = groupe dont l'entrée est le bouton warning et les voies sont les deux clignotants.

### → Ajouter un mode de sortie

1. `#define MODE_PERSO_1 20` dans la liste en haut de `Outputs_management.h` ;
2. écrire `bool Mode_Perso1(uint8_t voie, bool demande){ ... }` dans la section « UNE FONCTION PAR MODE » (retourne `true` si la voie doit être allumée) ;
3. ajouter `case MODE_PERSO_1: enService = demande; etat = Mode_Perso1(voie, demande); break;` dans le `switch` de `DriveChannel()` ;
4. dans le `.ini`, remplacer le choix « INVALID » de rang 20 des listes `outModeN` par le nom du mode.

## 6. Mode test ( `TS_Commands.h` )

- **Start Test** ( `E\x04\x01` ) : coupe les 16 voies, `TestMode = 1`. La fonction mère ne touche plus aux sorties.
- **Voie N ON / OFF** ( `E\x02\xNN` / `E\x03\xNN`, NN = 01 à 10 en hexa) : pilotent la voie à la main. Sans effet hors mode test. ON est refusé si le fusible de la voie est coupé.
- **End Test** ( `E\x04\x02` ) : coupe les 16 voies, `TestMode = 0`, la fonction mère reprend la main.
- `ManualStatus1/2` gardent l'état demandé à la main (1 bit par voie).

### Tous les boutons TunerStudio

| Commande | Code | Fonction |
|---|---|---|
| Effacer la dernière erreur | `E\x01\x10` | `ClearError()` |
| Réarmer tous les fusibles | `E\x01\x11` | `ResetFuses()` (Security.h) |
| Sniffer CAN pause / reprise | `E\x01\x12` / `E\x01\x13` | `SniffStop()` / `SniffGo()` |
| Start Test / End Test | `E\x04\x01` / `E\x04\x02` | `StartTest()` / `EndTest()` |
| Voie N ON | `E\x02\x01` à `E\x02\x10` | `OutON_N()` |
| Voie N OFF | `E\x03\x01` à `E\x03\x10` | `OutOFF_N()` |

### → Ajouter un bouton TunerStudio

1. Dans `TS_Commands.h` : le prototype en haut ( `void MaFonction();` ), puis `TS.addCommand(0xGGNN, MaFonction);` dans `InitCommands()`, puis la fonction elle-même sous le commentaire « déclarer ensuite chaque fonction

commandes ici ». Si la fonction est écrite dans un autre fichier inclus plus loin, ne mettre que le prototype et le `addCommand`.

2. Dans le `.ini` : `cmdMonBouton = "E\xGG\xNN"` dans `[ControllerCommands]`, et `commandButton = "Texte", cmdMonBouton` dans une fenêtre de `[UserDefined]`.
3. Codes libres : éviter le groupe `0xFF` (réservé à `BZO_TS.h`). Pas de limite de nombre de commandes.

## 7. Sécurité (`Security.h`)

### 7.1 Seuils absolus (survie de la carte, non réglables dans TS)

| Mesure | Seuil | Filtre avant coupure |
|---|---|---|
| Courant voies 1-12 | `MAX_CURRENT_1_12` = 10,0 A | `FRAMES_CURRENT` = 10 trames (~500 ms) |
| Courant voies 13-16 | `MAX_CURRENT_13_16` = 50,0 A | 10 trames |
| Température carte | `MAX_BOARD_TEMP` = 120 °C | `FRAMES_SLOW` = 20 trames (~1 s) |
| Tension +12P | `MAX_BATTERY` = 18,0 V | 20 trames |

Au dépassement (après le filtre), **si TunerStudio n'est pas connecté** :

1. la dernière erreur est mémorisée en EEPROM (3 derniers octets, hors zone TS, une écriture par nouvelle erreur) ;
2. les 16 voies sont coupées (`AllOutputsOFF()`) ;
3. les 16 LEDs rouges s'allument (signal de détresse) pendant `DISTRESS_DELAY` (500 ms) ;
4. `MCU_OK` passe à LOW : le superviseur d'alim coupe toute la logique 2 s, puis relance.

**Si TunerStudio est connecté**, rien n'est coupé (diagnostic) : seuls `SafetyStatus` et la mémoire d'erreur sont mis à jour.

`SafetyStatus` = 1 dès qu'un courant ou une température dépasse `SAFETY_WARNING` (80 %) de son seuil : voyant « ALERTE SEUIL » dans TS et LED status rapide. La batterie n'a pas de niveau « proche » : seulement au seuil.

> Ces seuils protègent le matériel. Ne les relever qu'en connaissance de cause (limites des composants de puissance et tenue du PCB).

### 7.2 Fusibles par voie (réglables)

Pour chaque voie : si le courant reste au-dessus de `Fuse_chN` (A) pendant `FuseDelay`, la voie est coupée, sa LED rouge s'allume et son bit passe à 1 dans `FuseStatus1/2`. Au bout de `FuseRearm`, le fusible se réarme tout seul (bit à 0, LED rouge éteinte) : la voie repart et recoupe si le défaut est toujours là. `FuseRearm = 0` : pas de réarmement automatique (bouton TS ou redémarrage). Les fusibles agissent même quand TS est connecté.

La logique d'une voie est dans `FuseChannel()`, appelée une fois par voie dans `CheckFuses()`.

### 7.3 LED status D1

| Priorité | Clignotement | Signification |
|---|---|---|
| 1 | rapide (100 ms) | une mesure est proche ou au-dessus d'un seuil absolu |
| 2 | double flash par seconde | TunerStudio connecté |
| 3 | lent (500 ms) | fonctionnement normal |

## 8. CAN (`CAN_Commands.h`) — un bac à sable

### 8.1 Principe

- Tous les **ID sont fixes**, en `#define` en haut du fichier. Ils ne se règlent pas dans TS.
- `ReadCAN()` lit **une seule trame par tour de** `loop()` (pas de blocage si le réseau est saturé), la passe au sniffer, compte le trafic, puis l'envoie au décodeur du mode choisi par `configPage1.CANMode` (`switch`).

- Les modes, temporisations et réglages PWM reçus par CAN sont écrits **en RAM** dans `configPage2`, sans sauvegarde.

## 8.2 Les deux modes standards

Deux réglages indépendants :

| | Demande/Réponse (0) | Broadcast (1) |
|---|---|---|
| Trames d'entrée (`CANMode`) : commandes reçues | action puis confirmation `0x610` | action, aucune réponse |
| Trames de sortie (`CANOutMode`) : état `0x610`, courants `0x611`-`0x613` | envoyées en réponse aux demandes `0x60E` / `0x60F` | envoyées en continu à `CANTxRate` Hz |

## 8.3 Trames reçues (format BZO)

Une valeur `0xFF` veut toujours dire « inchangé ».

Contrôle de syntaxe : une trame sur ces ID doit avoir 8 octets, chacun à `0xFF` ou dans sa plage (ON/OFF et groupes 0-1, modes 0-5, tempos et périodes 0-254, duty 0-100). Sinon elle est ignorée en entier et le voyant CAN du tableau de bord passe au rouge pendant 2 s (`CANError`, octet 155).

| ID | Rôle | Les 8 octets |
|---|---|---|
| `0x600` / `0x601` | ON/OFF virtuels voies 1-8 / 9-16 → `CANInputs1/2` | 1 octet par voie : 0 = OFF, 1 = ON |
| `0x602` / `0x603` | modes voies 1-8 / 9-16 → `OutMode[]` | 1 octet par voie |
| `0x604` / `0x605` | temporisations voies 1-8 / 9-16 → `OutTimeout[]` | 1 octet par voie (0,5 s) |
| `0x606` / `0x607` | périodes PWM voies 1-8 / 9-16 → `OutPWMPeriod[]` | 1 octet par voie (0,2 s) |
| `0x608` / `0x609` | rapports cycliques voies 1-8 / 9-16 → `OutPWMDuty[]` | 1 octet par voie (%) |
| `0x60A` | groupes 1-8 → `CANGroups` | 1 octet par groupe : 0 = OFF, 1 = ON |
| `0x60E` | demande d'état | sans données |
| `0x60F` | demande des courants | sans données |

Une entrée CAN virtuelle N n'agit que si une voie a `OutSource = 16 + N` (ou via un groupe).

## 8.4 Trames émises

| ID | Contenu |
|---|---|
| `0x610` | [0] `OutputStatus1`, [1] `OutputStatus2`, [2] `FuseStatus1`, [3] `FuseStatus2`, [4] `SafetyStatus`, [5] `TestMode`, [6] température carte max (°C), [7] `Battery10` |
| `0x611` | courants voies 1-8 (0,1 A, 1 octet chacun) |
| `0x612` | courants voies 9-12 (4 octets) |
| `0x613` | courants voies 13-16 (0,1 A, 2 octets chacun, poids faible d'abord) |

Trames de sortie : en Demande/Réponse → réponse aux demandes `0x60E` / `0x60F` ; en Broadcast → envoi de `0x610` à `0x613` à `CANTxRate` Hz.

## 8.5 Sécurité CAN et diagnostic

- `CANTimeout` (s) : si plus aucune trame n'est reçue pendant ce temps, `CANInputs1/2` et `CANGroups` repassent à 0. 0 = désactivée (conseillé quand les trames d'entrée sont en Demande/Réponse, où les commandes ne sont envoyées qu'une fois). Toute trame reçue, même hors format BZO, relance le délai.
- `CANTraffic` : trames reçues par seconde.
- Sniffer : les 10 dernières trames reçues (toutes, même celles qu'aucun mode n'utilise) dans `currentStatus.CANSniff[0..9]`, la plus récente en `[0]`. Boutons Pause / Reprendre dans TS (le trafic reste compté). TS les affiche en **décimal** (un ID `0x600` s'affiche 1536).
- Détail complet des trames : `FICHE_TRAMES_CAN_BZO_PDU.md`.

### → Ajouter un mode CAN (boîte à boutons, protocole d'un autre équipement…)

1. `#define CANMODE_PERSO_1 2` dans la liste « MODES CAN » ;
2. ses ID en `#define` (nommés clairement, par exemple `CAN_ID_MONBOITIER_BOUTONS`) ;
3. une fonction `void CAN_Perso1(){ ... }` qui lit `CAN_RX_msg.id`, `CAN_RX_msg.len`, `CAN_RX_msg.buf[0..7]` et agit sur `currentStatus.CANInputs1/2`, `currentStatus.CANGroups` ou `configPage2` ;
4. `case CANMODE_PERSO_1: CAN_Perso1(); break;` dans le `switch` de `ReadCAN()` ;
5. dans le `.ini`, remplacer le 3e choix de `CANMode` (« INVALID », valeur 2) par le nom du mode.

Pour émettre : remplir `CAN_TX_msg.id`, `CAN_TX_msg.len`, `CAN_TX_msg.buf[]`, puis `Can.write(CAN_TX_msg);` (modèle : `SendState()`). Les fonctions d'émission doivent être écrites **avant** les fonctions qui les appellent (ou avoir un prototype en haut du fichier).

## 9. Mesures (`AnalogReadings.h`)

Le processeur de mesure renvoie 19 octets sur demande (`'0'`) toutes les 50 ms environ. Référence ADC = 5 V, 8 bits (1 pas = 19,6 mV).

| Mesure | Formule (dans le code) | Coefficients |
|---|---|---|
| Courant voies 1-12 | I = V × K / Rsense | `SENSE_K_1_12` 2700 (typique), `SENSE_RS_1_12` 1200 Ω |
| Courant voies 13-16 | I = V × K / Rsense | `SENSE_K_13_16` 52100 (typique), `SENSE_RS_13_16` 3900 Ω (R11-R14) |
| Température carte | T = (1569 mV − V) / 8,2 mV/°C | bloquée à 0 °C en négatif |
| Tension +12P | V × (82k + 10k) / 10k | bloquée à 25,5 V |

Les coefficients K sont des valeurs typiques : ils se recalent au banc avec une charge connue (modifier `SENSE_K_1_12` / `SENSE_K_13_16`). Tous les calculs sont en entiers (`UL` = `unsigned long` pour éviter les débordements).

À chaque trame reçue : `newADC = true` (lu par `CheckSecurity()`, qui travaille une fois par trame).

## 10. Conventions de code (à respecter)

1. **Pas de boucle `for`** : les répétitions sont écrites ligne par ligne, ou via une petite fonction appelée une fois par ligne (exemples : `DriveChannel()`, `FuseChannel()`, `CopyByte()`).
2. **Bits** : toujours `BIT_SET(octet, rang)`, `BIT_CLEAR(octet, rang)`, `BIT_CHECK(octet, rang)` (définies dans `globals.h`). Jamais de `|=` / `&= ~` écrits à la main.
3. **Pas d'opérateur ternaire** (`a ? b : c`) : écrire un `if / else`.
4. Code explicite, une action par ligne, **commentaires en français** abondants. Pas d'astuces de langage, pas de pointeurs quand une référence (`&`) suffit.
5. Une fonction appelée avant d'être écrite dans le fichier a besoin d'un **prototype** en haut du fichier.
6. Fichiers en fins de ligne **CRLF** (Windows).
7. Nouveaux réglages : à la **fin** de `config1` ou `config2`, et en même temps dans le `.ini` (même offset, même taille).
8. **`TS_EEPROM_VERSION`** (dans le `.ino`) : c'est le propriétaire du projet qui décide quand l'incrémenter (à chaque changement de structure de page, sinon les réglages sauvegardés seront relus de travers). Ne pas le changer sans le lui dire.
9. **Signature** : `TS_SIGNATURE` dans le `.ino` et `signature` dans le `.ini` doivent être identiques au caractère près.
10. Les seuils de `Security.h` protègent le matériel : ne pas les rendre réglables dans TS.

## 11. Synchroniser firmware et `.ini`

À chaque fois qu'on ajoute une valeur :

| On ajoute… | Dans le firmware | Dans le `.ini` |
|---|---|---|
| un réglage | champ à la fin de `config1` / `config2` | ligne dans `page = 1` / `page = 2` au bon offset, `pageSize` mis à jour, champ dans une fenêtre `[UserDefined]` |
| une valeur temps réel | champ dans `statuses` + `case` dans `getTSLogEntry()` + `TS_LOG_TAILLE` | ligne dans `[OutputChannels]` au même offset, `ochBlockSize` = `TS_LOG_TAILLE`, éventuellement une jauge et une entrée `[Datalog]` |
| un bouton | `addCommand` (voir §6) | `[ControllerCommands]` + `commandButton` |

`TS_LOG_TAILLE` est actuellement **156** (offsets 0 à 155) : une nouvelle valeur commence à l'offset 156.

## 12. Pièges connus

- **PA13 / PA14 / PA15 / PB3 / PB4** sont des broches de débogage (SWD/JTAG) utilisées ici comme GPIO (`MCU_OK`, voies 7, 8, 12, LED rouge de la voie 16) : le débogage SWD n'est pas disponible. La carte se flashe avec le logiciel BZO Programmer, mode « Protocole BZO générique ».
- **Ordre des `#include`** : un « 'xxx' was not declared in this scope » vient presque toujours d'un fichier inclus avant celui qui déclare `xxx`.
- **Arduino IDE ouvert** : si un fichier est modifié en dehors de l'IDE pendant que le croquis est ouvert, l'IDE peut réécrire son ancienne version à l'enregistrement. Fermer et rouvrir le croquis après une modification externe.
- **Démarrage** : `InitSecurity()` doit rester la première ligne de `setup()` (le superviseur d'alim ne laisse que 2 s pour monter `MCU_OK`).
- **Batterie faible au démarrage du moteur** : sous ~7 V sur le +12P, la logique peut redémarrer (chute du 5 V), ce qui fait tomber `MCU_OK`.
- **Mesures** : sous ~1 A, la mesure des voies 1-12 est peu précise (dispersion du ratio K à faible courant).
- **Réglages modifiés par CAN** : ils vivent en RAM ; si TS lit la page 2 à ce moment, il voit les valeurs du CAN. Une sauvegarde (burn) depuis TS les écrirait en mémoire.