Zhengyu Wang & Domingos Bravo

GE5

**Conception et validation des commandes SCPI *:*** Méthodologie de communication multi-analyseurs (USB et LAN), organisation des données et fiabilité industrielle

**Table des matières**

[Introduction 3](#_Toc189501026)

[1. Construction et validation des commandes SCPI 5](#_Toc189501027)

[1.1. Contexte général : pourquoi choisir SCPI ? 5](#_Toc189501028)

[1.2. Lien avec les fichiers .sta : ce qui est déjà implémenté & pistes d’extension 5](#_Toc189501029)

[1.3. Aperçu général : où retrouver ces commandes dans le code ? 6](#_Toc189501030)

[1.4 Commandes IEEE générales (Common Commands) et interactions de base avec l’appareil 6](#_Toc189501031)

[1.4.1 \*CLS 6](#_Toc189501032)

[1.4.2 \*OPC? 6](#_Toc189501033)

[1.4.3 \*IDN? 6](#_Toc189501034)

[1.5 Réglage de fréquence (Frequency Subsystem) 7](#_Toc189501035)

[1.6 Réglages d’amplitude / gain (Amplitude Subsystem) 7](#_Toc189501036)

[1.7 Réglage de la largeur de bande (Bandwidth Subsystem) 8](#_Toc189501037)

[1.8 Balayage / temps de balayage (Sweep Subsystem) 8](#_Toc189501038)

[1.9 Configuration Trace (Trace Subsystem) 9](#_Toc189501039)

[1.9.1 Activation / configuration d’une Trace 9](#_Toc189501040)

[1.9.2 Sauvegarde des données :TRACe:DATA? {N} 9](#_Toc189501041)

[1.10 Limite (Limit Subsection) 9](#_Toc189501042)

[1.11 Sous-système Peak (Peak Searching) 10](#_Toc189501043)

[1.12 Sous-système Math (Trace Math) 10](#_Toc189501044)

[1.13 Configuration EMI (EMI Subsystem) 10](#_Toc189501045)

[1.14 Sauvegarde / chargement (Memory Subsystem) 11](#_Toc189501046)

[1.15 Transfert / acquisition de données (transfer\_csv\_to\_pc) 11](#_Toc189501047)

[1.16 Autres commandes : lien avec operation\_complete() 12](#_Toc189501048)

[1.17 Adaptation aux fichiers .sta : formats XML et binaire 12](#_Toc189501049)

[1.17.1 Des formats différents (Siglent SSA3032X vs SSA3021X) 12](#_Toc189501050)

[1.17.2 Différences entre SiglentSSA3000XPlus et HMS1000 13](#_Toc189501051)

[1.17.3 Logique de parsing et “factory driver” 16](#_Toc189501052)

[1.17.4 Bilan 17](#_Toc189501053)

[1.18 Vérification avant et après envoi 17](#_Toc189501054)

[1.18.1 Contrôles avant l’envoi 17](#_Toc189501055)

[1.18.2 Validation après l’envoi 18](#_Toc189501056)

[1.19 Points clés 18](#_Toc189501057)

[1.20 En résumé 19](#_Toc189501058)

[2. Méthodes de connexion USB et LAN 20](#_Toc189501059)

[2.1 Connexion USB 20](#_Toc189501060)

[2.1.1 Détection et initialisation du périphérique 20](#_Toc189501061)

[2.1.2 Timeouts et indisponibilité 20](#_Toc189501062)

[2.2 Connexion LAN 20](#_Toc189501063)

[2.2.1 Paramétrage de l’adresse IP 20](#_Toc189501064)

[2.2.2 Cohabitation de plusieurs appareils 21](#_Toc189501065)

[2.3 Gestion des temps de réponse et détection d’erreurs 21](#_Toc189501066)

[2.3.1 Timeouts dans les échanges SCPI 21](#_Toc189501067)

[2.3.2 Valeurs hors plage et appareils indisponibles 21](#_Toc189501068)

[2.4 Cas d’usages simultanés (plusieurs instruments) 22](#_Toc189501069)

[2.4.1 Identification des différentes ressources 22](#_Toc189501070)

[2.4.2 Conflits et priorités 22](#_Toc189501071)

[2.5 Conclusion sur la connexion et la gestion d’erreurs 22](#_Toc189501072)

[3. Centralisation de la gestion des erreurs 23](#_Toc189501073)

[3.1 Principes de robustesse en milieu industriel 23](#_Toc189501074)

[3.2 Module dédié aux codes d’erreurs 23](#_Toc189501075)

[3.2.1 Architecture globale 23](#_Toc189501076)

[3.2.2 Gestion des exceptions et affichage contrôlé 24](#_Toc189501077)

[3.3 Éviter la prolifération de pop-ups 24](#_Toc189501078)

[3.4 Gestion des déconnexions et reconnexions 24](#_Toc189501079)

[3.4.1 Exemple de connexion USB 24](#_Toc189501080)

[3.4.2 Exemple de connexion LAN 25](#_Toc189501081)

[3.5 Conclusion sur la centralisation de la gestion des erreurs 25](#_Toc189501082)

[4. Organisation et traçabilité des données 26](#_Toc189501083)

[4.1 Structure standardisée des données 26](#_Toc189501084)

[4.2 Le fichier de log : pierre angulaire de la traçabilité 27](#_Toc189501085)

[4.3 Principes de traçabilité et avantages pour la conformité 28](#_Toc189501086)

[5. Conclusion & Perspectives 28](#_Toc189501087)

# Introduction

Dans l’environnement actuel de mesures en Compatibilité Électromagnétique (CEM), il est fréquent de devoir gérer des besoins de mesure très variés ainsi que des analyseurs provenant de fabricants ou de gammes différentes. Par exemple, pour les tests d’émissions conduites ou d’émissions rayonnées, on utilise couramment des analyseurs tels que les Siglent SSA3032X / SSA3021X ou la série HMS1000. Face à la multiplicité de ces modèles, comment **unifier** la construction et la gestion de leurs commandes SCPI ? Comment **s’adapter avec flexibilité** aux connexions USB et LAN ? Comment **centraliser** le traitement des anomalies (comme la déconnexion d’un appareil, des paramètres hors plage ou des dépassements de délai importants) et **enregistrer** tous les paramètres de mesure et l’état de l’instrument, afin de satisfaire aux exigences d’un environnement industriel en matière de traçabilité ? Ce sont précisément les questions au cœur de cette *note d’application*.

Le présent rapport s’articule comme suit :

1. **Construction et validation des commandes SCPI**

* Nous expliquerons comment convertir les paramètres XML/binaires des fichiers .sta en commandes SCPI unifiées, et comment mettre en place un “factory driver” pour réutiliser le code entre différents modèles (Siglent, HMS, etc.).
* Nous détaillerons les commandes SCPI pour les sous-systèmes relatifs à la fréquence, la largeur de bande, l’amplitude, la limite (Limit), les traces (Trace), la recherche de pics (Peak), l’EMI, ainsi que les points clés de configuration et de validation en programmation.

1. **Méthodes de connexion USB/LAN**

* Nous décrirons comment, dans un environnement Windows, écouter les événements de branchement/débranchement USB, initialiser le gestionnaire VISA, configurer les adresses IP et détecter plusieurs appareils LAN.
* Nous mettrons l’accent sur la **gestion des délais (timeout)**, la réaction aux déconnexions réseau ou USB, et l’accès séquentiel en présence de plusieurs instruments.

1. **Centralisation de la gestion des erreurs**

* Nous présenterons la logique d’un module dédié (tel que “StatusManager”) pour regrouper toutes les erreurs via des codes spécifiques (par ex. ERROR\_USB\_CONNEXION\_FAILED, ERROR\_OPC\_FAILED), afin d’éviter une prolifération de pop-ups et de renforcer la robustesse.
* Nous insisterons sur la prise en compte des scénarios industriels de **déconnexion/reconnexion** et sur l’idée d’une **auto-réinitialisation**, avec un unique dialogue ou un journal centralisé pour informer l’opérateur de toutes les erreurs.

1. **Organisation des données et traçabilité**

* Nous proposerons une structure de répertoires recommandée (LOG/, Measures\_data/, LTSpice/, report/, etc.) pour stocker séparément les journaux (logs), les données brutes, les résultats de traitement et les rapports.
* Nous donnerons un exemple de fichier de journal (log\_file.txt) détaillant les informations enregistrées : signaux mesurés, fréquences de début/fin, RBW/VBW, mode Trace, avertissements de correction automatique, etc., de manière à garantir facilité de vérification et exactitude a posteriori.

À travers ces différentes sections, le lecteur verra comment, dans un processus de mesure CEM, il est possible de gérer **efficacement** et **évolutivement** l’automatisation de plusieurs marques d’instruments tout en préservant la **stabilité** et la **traçabilité** requises en milieu industriel.

# 1. Construction et validation des commandes SCPI

Dans le cadre de l’automatisation des essais CEM, l’une des tâches essentielles consiste à **concevoir, envoyer et valider** les commandes adressées à l’analyseur de spectre. Grâce à la **norme SCPI (Standard Commands for Programmable Instruments)**, on peut piloter différentes gammes d’appareils (par exemple Siglent SSA3032X, SSA3021X, HMS1000, etc.) au moyen d’un langage de commande unifié. La présente section aborde :

* La **construction pratique** des commandes SCPI (gestion de la fréquence, RBW/VBW, mode crête ou moyen, etc.).
* Les **caractéristiques de format** des fichiers .sta (XML et binaire) et l’adaptation grâce à une approche “factory driver”.
* Les **vérifications nécessaires** visant à garantir la cohérence et la validité des paramètres.

## 1.1. Contexte général : pourquoi choisir SCPI ?

La plupart des analyseurs de spectre modernes prennent en charge des commandes SCPI permettant de configurer leurs paramètres internes. Par exemple :

* :SENSe:FREQuency:STARt 150000 définit la fréquence de départ à 150 kHz,
* :SENSe:BWIDth:RESolution 1kHz règle la bande passante de résolution (RBW) à 1 kHz,
* :SENSe:DETector:FUNCtion POS active le mode de détection “Peak positif”, etc.

Dans un logiciel d’automatisation, **ces commandes SCPI sont généralement générées automatiquement à partir d’un fichier de configuration (par ex. \*.sta)**. Ainsi, si un fichier .sta comporte les champs Start\_Freq, Stop\_Freq, RBW, VBW, ceux-ci seront “traduits” en commandes SCPI compréhensibles par l’analyseur.

**Remarque** :  
Toutes les commandes SCPI mentionnées ci-après figurent dans la documentation de programmation (Programming Guide PG0703P\_E02A) où l’on retrouve leur signification, leurs paramètres et leurs limites. À travers l’exemple de code SiglentSSA3000XPlus, vous verrez :

1. Comment nous faisons correspondre les paramètres des fichiers .sta aux commandes SCPI,
2. Le rôle de chacune de ces commandes dans la logique d’automatisation.

## 1.2. Lien avec les fichiers .sta : ce qui est déjà implémenté & pistes d’extension

* Dans notre exemple, les principaux tags de **fichiers .sta** (<Frequency>, <Ampt>, <BW>, <Sweep>, <Trace>, <Limit>, <Peak>, <EMI\_\*>, <Math>, etc.) ont **déjà un équivalent** côté SCPI.
* En revanche, des tags comme <Marker>, <Meas>, <DEMOD> ne sont que **partiellement pris en compte** pour l’instant (création d’un parseur minimal). Si vous souhaitez aller plus loin (par ex. automatiser ACP/OBW, gérer des marqueurs plus avancés, etc.), il suffit d’ajouter les commandes SCPI correspondantes (par ex. :CALCulate:MARKer:NDB {ON|OFF}, :CALCulate:OBWidth:STATe {ON|OFF}, etc.).
* Globalement, la stratégie est la suivante : **“lire les paramètres dans un fichier XML (.sta) → les mapper vers SCPI → envoyer les commandes à l’instrument”**, en veillant à ce que les évolutions futures de projet restent simples à maintenir.

## 1.3. Aperçu général : où retrouver ces commandes dans le code ?

Dans la classe d’exemple SiglentSSA3000XPlus, on utilise souvent self.inst.write("...") et self.inst.query("...") :

* **inst.write(...)** envoie une commande d’écriture à l’instrument,
* **inst.query(...)** interroge directement l’appareil (souvent pour lire une valeur mesurée ou un état, p. ex. :SENSe:...:READ?, \*OPC?).

Nous présenterons ci-après, **sous forme de catégories fonctionnelles** (Frequency, Amplitude, BW, Sweep, Trace, Limit, Peak, EMI, etc.), les principales commandes SCPI, leur utilité, leur emplacement dans le code, et les chapitres de la documentation auxquels elles se réfèrent.

## 1.4 Commandes IEEE générales (Common Commands) et interactions de base avec l’appareil

### 1.4.1 \*CLS

* **Signification** : Clear Status. Efface les registres d’événements de l’instrument et réinitialise la file d’erreurs, les registres de statut, etc.
* **Référence** : manuel de programmation [3.1 IEEE Common Commands – entrée \*CLS].
* **Emploi dans le code** : dans la méthode transfer\_csv\_to\_pc, via self.inst.write("\*CLS").
* **Rôle** : avant de lancer une nouvelle mesure, on purifie les éventuels statuts/erreurs résiduels.

### 1.4.2 \*OPC?

* **Signification** : Operation Complete Query. Renvoie “1” lorsque la tâche en cours est terminée.
* **Référence** : manuel de programmation [3.1 IEEE Common Commands – entrée \*OPC?].
* **Emploi dans le code** :
  + Dans la méthode operation\_complete : opc\_response = self.inst.query('\*OPC?'),
  + À divers endroits, self.inst.query('\*OPC?') sert à vérifier si l’instrument a fini d’exécuter la commande.
* **Rôle** : réaliser une synchronisation pour éviter de lire ou d’envoyer la prochaine commande trop tôt.

### 1.4.3 \*IDN?

* **Signification**  
  Interroge l’appareil pour obtenir la chaîne d’identification complète (format habituel : “Siglent Technologies, , , ”, etc.).
* **Référence**  
  Une commande standard IEEE 488.2, présente sur la plupart des instruments programmables. Cf. manuel [3.1 IEEE Common Commands – entrée \*IDN?].
* **Emploi dans le code**  
  Dans la classe SiglentSSA3000XPlus, on **n’utilise pas** directement \*IDN?. Néanmoins, en phase de test ou de vérification de connexion (USB/LAN), on s’en sert fréquemment :
* response = inst.query("\*IDN?")
* if expected\_response in response:
* print("Appareil reconnu")
* else:
* print("La réponse n’est pas conforme aux attentes")

**Rôle** : confirmer que l’appareil détecté correspond bien au modèle attendu et garantir que les commandes envoyées sont valides.

## 1.5 Réglage de fréquence (Frequency Subsystem)

Dans la méthode load\_config(...), le bloc if 'Frequency' in parameters: appelle :

1. :SENSe:FREQuency:SPAN {valeur}
2. :SENSe:FREQuency:STARt {valeur}
3. :SENSe:FREQuency:STOP {valeur}
4. :SENSe:FREQuency:CENTer {valeur}
5. :SENSe:FREQuency:CENTer:STEP {valeur}
6. :SENSe:FREQuency:OFFSet {valeur}
7. :SENSe:FREQuency:CENTer:STEP:AUTO {ON|OFF}

* **Signification / référence**
  + :SENSe:FREQuency:STARt/STOP : définit la fréquence de départ/fin du balayage (en Hz).
  + :SENSe:FREQuency:CENTer / :SENSe:FREQuency:SPAN : réglage sous forme fréquence centrale + étendue.
  + :SENSe:FREQuency:CENTer:STEP : pas de la fréquence centrale.
  + :SENSe:FREQuency:OFFSet : offset en fréquence.
  + :SENSe:FREQuency:CENTer:STEP:AUTO : activation/désactivation de l’auto-calcul du pas.
* **Rôle dans le code** : appliquer les réglages de balayage (fréquences de départ/fin, etc.) issus du tag <Frequency> d’un fichier .sta.
* **Chapitre correspondant dans le manuel** : 4.1 Frequency Subsection.

## 1.6 Réglages d’amplitude / gain (Amplitude Subsystem)

Toujours dans load\_config(...), sous if 'Ampt' in parameters:, on envoie :

1. :DISPlay:WINDow:TRACe:Y:RLEVel {valeur}DBM
2. :DISPlay:WINDow:TRACe:Y:SCALe:RLEVel:OFFSet {valeur}
3. :DISPlay:WINDow:TRACe:Y:SCALe:PDIVision {valeur}
4. :SENSe:POWer:RF:ATTenuation {valeur}
5. :SENSe:POWer:RF:ATTenuation:AUTO {ON|OFF}
6. :SENSe:POWer:RF:GAIN:STATe {ON|OFF}
7. :UNIT:POWer {DBM|DBUV...}
8. :DISPlay:WINDow:TRACe:Y:SCALe:SPACing {LIN|LOG}
9. :SENSe:CORRection:OFF
10. :SENSe:CORRection:CSET{1..4}:STATe OFF
11. [:SENSe]:CORRection:IMPedance[:INPut][:MAGNitude] {OHM50|OHM75}

* **Utilité** :
  + :DISPlay:WINDow:TRACe:Y:RLEVel définit le niveau de référence (Ref Level).
  + :SENSe:POWer:RF:ATTenuation règle l’atténuation d’entrée (0–51 dB) ; AUTO bascule l’atténuation automatique.
  + :SENSe:POWer:RF:GAIN:STATe active/désactive l’amplificateur d’entrée (préampli).
  + :UNIT:POWer sélectionne l’unité de puissance (DBM, DBUV, W...).
  + :DISPlay:WINDow:TRACe:Y:SCALe:SPACing LOG ou LIN fixe l’échelle verticale (logarithmique/linéaire).
  + :SENSe:CORRection:OFF et :SENSe:CORRection:CSETn:STATe OFF désactivent les tables de correction d’amplitude.
  + :SENSe:CORRection:IMPedance paramètre l’impédance d’entrée (50 Ω ou 75 Ω).
* **Référence** : manuel de programmation 4.2 Amplitude Subsection.

## 1.7 Réglage de la largeur de bande (Bandwidth Subsystem)

Dans le bloc if 'BW' in parameters: :

1. :SENSe:BWIDth:RESolution {valeur}
2. :SENSe:BWIDth:RESolution:AUTO {ON|OFF}
3. :SENSe:BWIDth:VIDeo {valeur}
4. :SENSe:BWIDth:VIDeo:AUTO {ON|OFF}
5. :SENSe:BWIDth:VIDeo:RATio {valeur}
6. :SENSe:AVERage:TYPE {LOGPower|POWer|VOLTage}

* **Référence** : manuel de programmation 4.5 Bandwidth Subsection
* **Utilité** :
  + :SENSe:BWIDth:RESolution / VIDeo déterminent la RBW et la VBW.
  + AUTO ON|OFF indique si l’on utilise un couplage automatique.
  + :SENSe:BWIDth:VIDeo:RATio définit le rapport VBW/RBW (typiquement 3.0).
  + :SENSe:AVERage:TYPE choisi le type d’avg (puissance, log, tension).

## 1.8 Balayage / temps de balayage (Sweep Subsystem)

Dans if 'Sweep' in parameters: :

1. :SWEep:TIME {valeur}
2. :SWEep:TIME:AUTO {ON|OFF}
3. :SWEep:SPEed {NORMal|ACCUracy}
4. :SWEep:COUNt {valeur}
5. :SWEep:MODE {AUTO|SWEep|FFT}
6. :INITiate:CONTinuous {ON|OFF}

* **Référence** : manuel 4.3 Sweep Subsection
* **Explications** :
  + :SWEep:TIME définit le temps de balayage (souvent réglé automatiquement par l’appareil).
  + :SWEep:TIME:AUTO active/désactive la détermination automatique de la durée.
  + :SWEep:SPEed {NORM|ACCU} choisit la vitesse de balayage (précision vs normal).
  + :SWEep:COUNt {n} pour répéter le balayage en mode simple ou continu.
  + :SWEep:MODE détermine si l’on opère en balayage classique ou en mode FFT, etc.
  + :INITiate:CONTinuous ON|OFF bascule entre un balayage en continu ou un seul balayage.

Dans la méthode launch\_measurement, on voit un appel à self.inst.query('\*OPC?') pour savoir quand le balayage est terminé.

## 1.9 Configuration Trace (Trace Subsystem)

Dans if 'Trace' in parameters: :

### 1.9.1 Activation / configuration d’une Trace

* :TRACe{N}:MODE {WRIT/VIEW/AVER/MAXH/MINH/BLAN...}
* :SENSe:DETector:TRACe{N}:FUNCtion {POS|NEG|AVER|SAMP|NORMAL|QUAS}
* :SENSe:AVERage:TRACe{N}:COUNt {avg\_times}
* :SENSe:AVERage:TRACe{N}:STATe ON
* **Référence** : manuel 4.6 Trace Subsection
* **À savoir** :
  + :TRACe{N}:MODE WRIT : mode d’écriture standard,
  + :TRACe{N}:MODE AVER : affichage avec moyenne,
  + :TRACe{N}:MODE MAXH/MINH : maintien du maximum/minimum,
  + :SENSe:DETector:TRACe{N}:FUNCtion : détection (crête+, crête–, échantillonnage, etc.).
  + En mode AVER, on précise :SENSe:AVERage:TRACe{N}:COUNt (nombre de moyennes).

### 1.9.2 Sauvegarde des données :TRACe:DATA? {N}

* Dans transfer\_csv\_to\_pc, on emploie response = self.inst.query(f':TRACe:DATA? {index\_trace}').
* **Correspondance** : :TRACe[:DATA]? / :FORMat[:TRACe][:DATA] {ASCii|REAL}
* **Référence** : manuel 4.6.2 :TRACe[:DATA]?
* **Fonction** : lire la forme d’onde sur une trace donnée pour export en CSV ou en texte brut.

### 1.10 Limite (Limit Subsection)

Dans if 'Limit' in parameters: :

1. :CALCulate:LLINe{N}:STATe {ON|OFF}
2. :CALCulate:LLINe{N}:TYPE {UPPer|LOWer}
3. :CALCulate:LLINe{N}:MODE {LINE|POINt}
4. :CALCulate:LLINe{N}:Y {valeur}
5. :CALCulate:LLINe:FAIL:STOP {ON|OFF}
6. :CALCulate:LLINe:CONTrol:BEEP {ON|OFF}
7. :CALCulate:LLINe:CONTrol:DOMain {FREQuency|TIME}

* **But** : dans un balayage, définir des courbes de limites (ligne haute/basse), configurer des alertes sonores en cas de dépassement, etc.
* **Référence** : 4.8 Limit Subsection

## 1.11 Sous-système Peak (Peak Searching)

Dans if 'Peak' in parameters: :

1. :CALCulate:MARKer:PEAK:SEARch:MODE {MAX|MIN}
2. :CALCulate:MARKer:PEAK:THReshold {valeur}
3. :CALCulate:MARKer:PEAK:EXCursion {valeur}
4. :CALCulate:MARKer:PEAK:TABLe {ON|OFF}
5. :CALCulate:MARKer{n}:CPEak:STATe {ON|OFF}

* **Explications** :
  + :CALCulate:MARKer:PEAK:THReshold : seuil de détection de pics,
  + :CALCulate:MARKer:PEAK:EXCursion : largeur de résolution pour distinguer plusieurs pics,
  + :CALCulate:MARKer:PEAK:TABLe ON : affichage de la table de pics.
* **Référence** : 4.7 Marker Subsection → commandes PEAK.

## 1.12 Sous-système Math (Trace Math)

Dans if 'Math' in parameters: :

* :CALCulate:MATH:FUNCtion {OFF|PDIF|PSUM|LOFF|LDIF}
* :CALCulate:MATH:X TRACE{n}
* :CALCulate:MATH:Y TRACE{n}
* :CALCulate:MATH:Z TRACE{n}
* :CALCulate:MATH:OFFSet {valeur}
* **Référence** : manuel 4.6 Trace Subsection → :CALCulate[:SELected]:MATH:...
* **Utilité** : réaliser différentes opérations (différence de puissance, somme, différences log, etc.) ou désactiver (OFF). On spécifie X, Y comme sources de trace, le résultat étant en Z ; on peut aussi appliquer un offset.

## 1.13 Configuration EMI (EMI Subsystem)

Dans if 'EMI' in parameters:, if 'EMI\_BW' in parameters:, if 'EMI\_Sweep' in parameters: :

* :SENSe:FILTer:TYPE {EMI|GAUSS}
* :SENSe:METer1:DETector:DWELl {valeur}ms
* :SENSe:FSCan:SCAN:BWIDth:RESolution {valeur}
* :SENSe:FSCan:SCAN:BWIDth:RESolution:AUTO {ON|OFF}
* :SENSe:BANDwidth:VIDeo {valeur}
* :SENSe:BANDwidth:VIDeo:AUTO {ON|OFF}
* :SENSe:BWIDth:VIDeo:RATio {valeur}
* :SENSe:AVERage:TYPE {LOGPower|POWer|VOLTage}
* :SENSe:SWEep:TIME {valeur}
* :INITiate2:CONTinuous {ON|OFF}
* [:SENSe]:SWEep:COUNt {valeur}
* **Référence** : manuel 9. EMI Measurement
* **Explications** : en mode EMI, on configure le filtre EMI (:SENSe:FILTer:TYPE EMI), la bande passante de scan (FSCan:SCAN:BWIDth), le temps de mesure (dwell time), etc.

## 1.14 Sauvegarde / chargement (Memory Subsystem)

Dans la méthode save\_data :

self.inst.write(f':MMEM:STOR CSV, "{self.csv\_file}.csv"')

self.inst.write(f':MMEM:STOR STA, "{self.sta\_file}.sta"')

self.inst.write(f':MMEM:LOAD STA, "{self.sta\_file}.sta"')

* **Commandes** :
  + :MMEMory:STORe CSV, "fichier.csv"
  + :MMEMory:STORe STA, "fichier.sta"
  + :MMEMory:LOAD STA, "fichier.sta"
* **Référence** : manuel 3.3 Memory Subsystem
* **Commentaires** : on enregistre l’état courant de l’instrument au format CSV/STA, ou on recharge un fichier .sta.

## 1.15 Transfert / acquisition de données (transfer\_csv\_to\_pc)

Cette méthode utilise également :

* :INITiate:IMMediate
* :INITiate:CONTinuous?
* :TRACe:DATA? {n}
* :SENSe:FREQuency:STARt?
* :SENSe:FREQuency:STOP?
* :SWEep:TIME?
* :SWEep:MODE?
* etc. (sous forme de *query*)

**Note** : La commande :INITiate:IMMediate (se reporter au manuel 4.3 Sweep ou 4.3.5 INITiate Subsystem) provoque un nouveau balayage ou une nouvelle mesure par l’analyseur.

## 1.16 Autres commandes : lien avec operation\_complete()

* self.inst.query('\*OPC?') (mentionné plus haut)
* La gestion de timeout et d’erreurs relève de la logique applicative et ne modifie pas la configuration de l’instrument.

## 1.17 Adaptation aux fichiers .sta : formats XML et binaire

Dans un processus de test automatisé, nous récupérons depuis un fichier de configuration externe (\*.sta) un ensemble de paramètres (plages de fréquences, RBW, VBW, Traces, Peak, Limit, etc.), à traduire en commandes SCPI destinées à l’analyseur de spectre. Toutefois, **selon le modèle** (voire différentes gammes d’un même constructeur), le format du fichier .sta peut varier. Il faut donc un parseur suffisamment **souple**. Nous examinerons d’abord deux exemples Siglent SSA3032X et SSA3021X, puis nous comparerons la série HMS1000 (autre constructeur).

### 1.17.1 Des formats différents (Siglent SSA3032X vs SSA3021X)

**a) Siglent SSA3032X**

1. Format du fichier
   * XML quasiment “pur” (ex. <configuration mode="Analyzer">…).
2. Structure typique
   * On y voit <Analyzer>, <FREQ>, <BW>, <AMPT>, <TRACE>, etc., avec des attributs (p. ex. start="150000", rbw="8") ou des sous-balises pour les valeurs.
3. Exemple de parsing
   * En Python, on peut faire :
   * import xml.etree.ElementTree as ET
   * tree = ET.parse("P12\_PK\_test1\_SSA3032Xplus.sta")
   * root = tree.getroot()
   * freq\_node = root.find(".//Analyzer/FREQ")
   * start\_freq = freq\_node.get("start") # "150000"
   * Si le fichier contient des caractères nuls (\x00) ou des retours à la ligne superflus, un nettoyage **préalable** est parfois nécessaire.
4. Points de vigilance
   * Certains modèles SSA3032X proposent des fonctionnalités EMI, Meas, etc., représentant <EMI> ou <MEAS>. Si votre projet n’en a pas l’utilité, il suffit d’ignorer ces champs.
   * Des tags tels que <Marker>, <TG> peuvent être absents si la fonction n’est pas activée, d’où l’importance de gérer les cas de balises vides.

**b) Siglent SSA3021X**

1. Format du fichier
   * On retrouve un schéma XML, mais sur certains anciens appareils, il peut s’agir d’un format **hybride (XML/binaire)**, avec beaucoup de caractères parasites, ce qui peut faire échouer ElementTree.
2. Structure typique
   * La balise racine est <nsp\_config>, contenant <Frequency>, <Ampt>, <BW>, <Trace>, etc., parfois imbriqués dans <Marker>, <Meas>, <Peak>.
   * Des nœuds comme <Trace1\_Info>1,0,100</Trace1\_Info> imposent de découper la chaîne en virgules pour en extraire trace\_type=1, detect=0, avg\_times=100.
3. Exemple de parsing
   * Retirer d’abord les \x00 ou caractères illégaux, puis faire un ElementTree.parse().
   * Les champs <TraceX\_Info> se traitent par split sur la virgule.
4. Points de vigilance
   * Dans d’anciennes versions, le .sta peut être davantage binaire, avec très peu d’infos XML.
   * Si parse() échoue, on peut tenter la voie des expressions régulières (ex. détecter <Start\_Freq>(\d+\.?\d+)</Start\_Freq>).
   * Les sections <Marker>, <Meas>, <Peak> sont parfois **inutilisées** dans le projet ; on peut ignorer leur contenu ou définir des valeurs par défaut.

### 1.17.2 Différences entre SiglentSSA3000XPlus et HMS1000

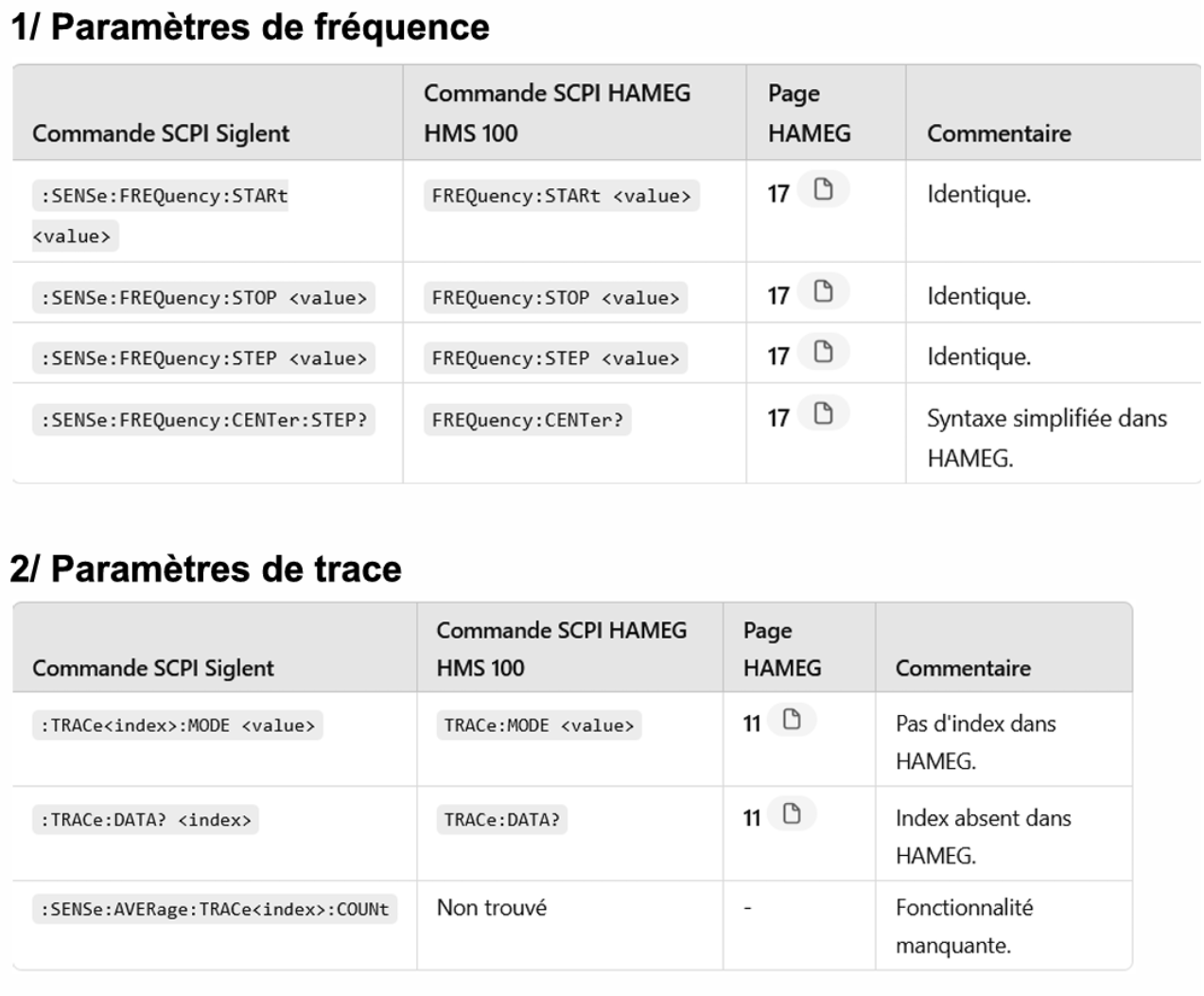
Pour gérer **plusieurs constructeurs** (ou différentes gammes) dans un même projet, on constate aussi que les **commandes SCPI** peuvent varier sensiblement. Voyons brièvement la comparaison entre deux drivers déjà disponibles : SiglentSSA3000XPlus.py et HMS1000.py.

1. Style de commande
   * **Siglent** (série 3000X Plus / 3021X) : on utilise des commandes hiérarchisées comme :SENSe:FREQuency:STARt, :SENSe:BWIDth:RESolution.
   * **HMS** (HMS1000) : plutôt :FREQuency:STARt, :BANDwidth:RBW, :AMPLitude:ATTenuation, etc.
   * Les traces diffèrent aussi : chez Siglent on a :TRACe{n}:MODE, :SENSe:DETector:TRACe{n}:FUNCtion, alors que chez HMS c’est :TRACe{n}:MODE CLR/PEAK/RMS.
2. Couverture fonctionnelle
   * Le code Siglent intègre souvent des éléments avancés (EMI, QPD, etc.).
   * HMS possède des commandes spécifiques (ex. :HCOPy:DATA:FORMat CSV) pour exporter les données.
3. Différences d’implémentation
   * **SiglentSSA3000XPlus.py**
     + Dans load\_config(), on mappe <Frequency> en :SENSe:FREQuency:STARt …, <BW> en :SENSe:BWIDth:RESolution …, etc.
     + Les fonctions Trace, Peak, Limit suivent un style SCPI plus standard chez Siglent.
   * **HMS1000.py**
     + On gère aussi <Frequency>, <Ampt>, <BW>, mais on génère des commandes différentes (par ex. :FREQuency:STARt … au lieu de :SENSe:FREQuency:STARt).
     + Pour les modes de détection, on peut trouver :TRACe:DETector:MODE?, etc., propres aux HMS.
4. Points communs
   * Les deux incluent load\_config(), save\_data(), transfer\_csv\_to\_pc() pour prendre en charge le flux “chargement config → sauvegarde → transfert de fichiers”.
   * On peut utiliser operation\_complete() ou \*OPC? pour synchroniser la fin d’une opération.
   * Les deux s’appuient sur un dictionnaire de paramètres issu de la couche de parsing, de manière à éviter le “hard-coding” dans chaque driver.

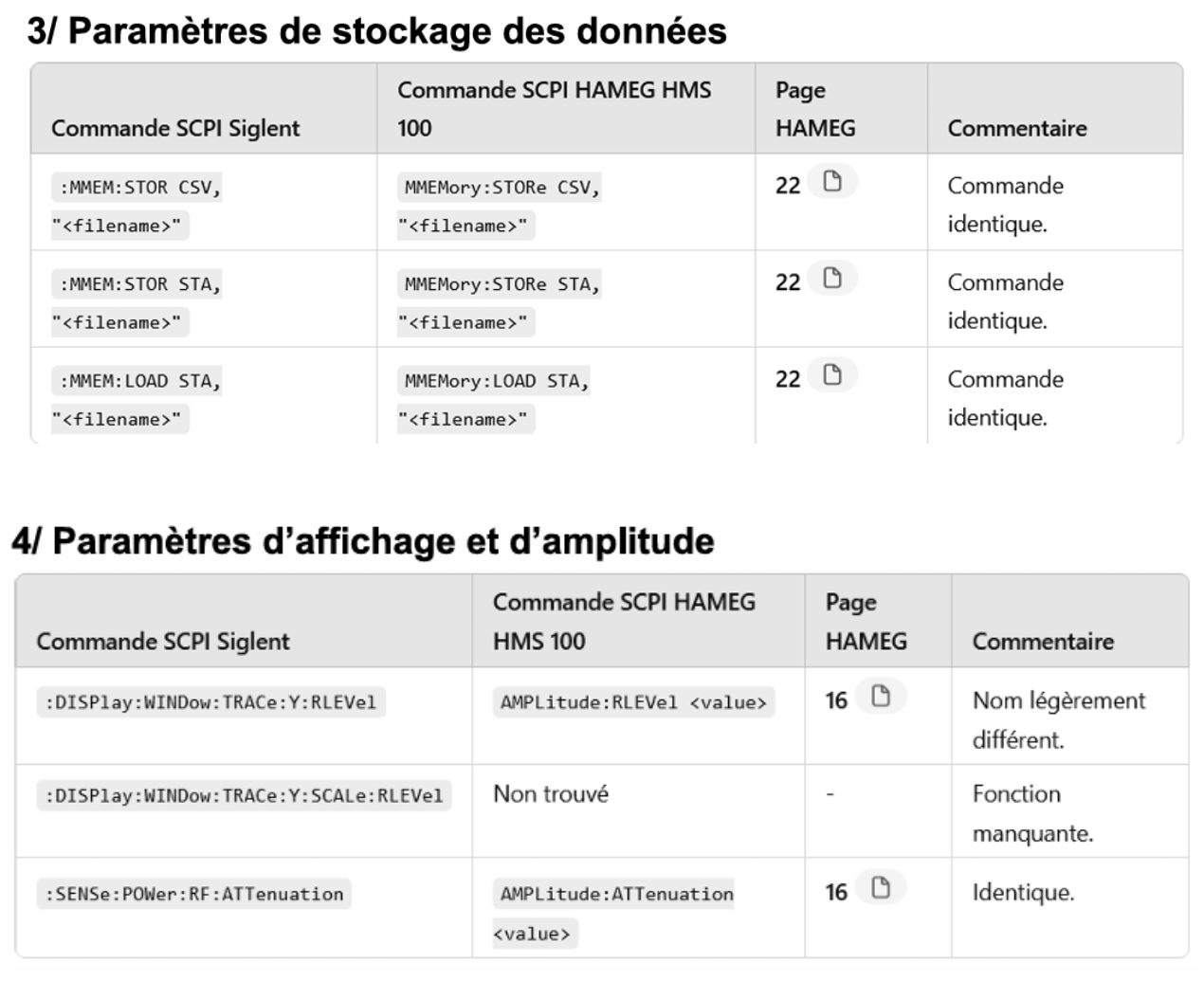
**En résumé** :

* Les commandes SCPI chez **Siglent** et **HMS** diffèrent notablement.
* Mixer ces commandes dans une seule fonction générerait du code illisible (ex. if brand=="Siglent": ... else if brand=="HMS": ...).
* D’où l’approche “**factory driver**” : un fichier driver par instrument, conservant les spécificités de chacun, et réutilisable dans un workflow commun (parsing → config → mesure → export).

Dans les images ci-dessous, nous comparons en détail les deux classes de pilote que nous avons déjà implémentées : SiglentSSA3000XPlus.py et HMS1000.py.



**Figure 1 :** Comparaison partielle des paramètres entre SiglentSSA3000XPlus.py et HMS1000.py



**Figure 2 :** Comparaison partielle des paramètres entre SiglentSSA3000XPlus.py et HMS1000.py



**Figure 3 :** Comparaison partielle des paramètres entre SiglentSSA3000XPlus.py et HMS1000.py

### 1.17.3 Logique de parsing et “factory driver”

Le **concept “factory driver”** repose sur :

1. **Couche parsing** : pour chaque gamme ou format de fichier, on définit une fonction parse\_sta\_file() (ou similaire) qui va extraire les champs d’un <nsp\_config> ou <configuration> et remplir un dictionnaire unique parameters (par ex. parameters['Frequency']['Start\_Freq'] = 150000).
2. **Couche driver** : dans load\_config(), save\_data(), transfer\_csv\_to\_pc(), on traduit ensuite les valeurs de parameters en commandes SCPI appropriées **au modèle d’instrument** (SiglentSSA3000XPlus, HMS1000, etc.).

**Exemple** :

* Si le parseur trouve <Trace1\_Info>1,0,100</Trace1\_Info>, il en déduit trace\_type=1, detect=0, avg\_times=100 qu’il stocke dans parameters["Trace"]["Trace1\_Info"].
* Dans le driver Siglent, on émettra self.inst.write(':TRACe1:MODE WRIT'), alors que dans le driver HMS, ce sera :TRACe1:MODE CLR.

**Robustesse** :

* Si le .sta est binaire ou incomplet, on supprime d’abord les \x00, on tente éventuellement des regex pour isoler certains champs, on signale les erreurs de parsing avant de passer à l’étape SCPI.

Grâce à ce découplage “**couche parsing** + **couche driver**”, on peut appliquer un même flux aux appareils Siglent et HMS, mais en n’adaptant qu’un petit segment de code par type d’instrument.

### 1.17.4 Bilan

Que l’on ait un **SSA3032X** très “XML friendly” ou un **SSA3021X** semi-binaire, il faut une stratégie de parsing **adaptable** pour extraire des infos comme <Frequency>, <Ampt>, <BW>. Concrètement :

1. **D’abord**, utiliser les bibliothèques standard (ex. xml.etree.ElementTree) pour tout ce qui est parseable,
2. **Ensuite**, si nécessaire, nettoyer les caractères spéciaux ou compléter avec des regex,
3. **Enfin**, générer un dictionnaire unifié parameters, que l’on enverra à chaque driver (SiglentSSA3000XPlus, HMS1000, etc.) pour la traduction SCPI.

Ainsi, même si l’on doit gérer Siglent **et** HMS ou, à l’avenir, ajouter Keysight, R&S, etc., on se contente d’ajouter un “factory” côté parsing et côté driver, sans casser l’architecture existante. On gagne donc en **maintenabilité** et en **extensibilité**.

## 1.18 Vérification avant et après envoi

Une fois le fichier .sta correctement interprété et les commandes SCPI générées, le code intègre des mécanismes de **contrôle avant envoi** et de **validation post-envoi**, décrits ci-dessous.

### 1.18.1 Contrôles avant l’envoi

1. **Correspondance fichiers / signaux**
   * Quand l’opérateur clique “Mesure”, on vérifie d’abord la présence effective du fichier .sta, son chemin, et la concordance entre les noms de signaux et les fichiers. En cas de problème, on arrête tout et on affiche une erreur.
2. **Test de connexion (USB/LAN)**
   * Le code appelle verify\_usb\_connexion ou lan\_instrument.verify\_connection\_dynamique et compare la réponse à \*IDN? pour s’assurer de la disponibilité de l’appareil. Si la connexion n’est pas valide, aucune commande n’est émise.
3. **Cohérence des paramètres**
   * Dans load\_config() de chaque driver, on contrôle la validité (RBW, VBW, pas de fréquence, etc.). Par exemple, format\_bandwidth\_value() peut corriger une valeur hors plage, en avertissant l’utilisateur ou en ajustant automatiquement vers la valeur la plus proche.
   * Si le ratio VBW/RBW est hors limites, on peut l’indiquer et forcer un ajustement.
4. **Lecture / mapping des Traces**
   * Si <TraceX\_Info> indique 1,4,100, on convertit ces valeurs en SCPI (POS/NEG/AVER, etc.), en tenant compte du type de détection et du nombre d’averages.
   * En cas de format inconnu ou de champ manquant, on signale l’anomalie et on stoppe l’envoi.

### 1.18.2 Validation après l’envoi

1. **Synchronisation du processus de mesure**
   * Dans transfer\_csv\_to\_pc, si l’instrument est en balayage unique (continuous=OFF), on boucle sur \*OPC? jusqu’à ce que l’appareil rende “1”. S’il est en balayage continu (continuous=ON), on attend un délai fixe avant de récupérer les données. De cette façon, on évite de lire trop tôt.
2. **Lecture et archivage des valeurs réelles**
   * Le code interroge l’analyseur (self.inst.query(...)) pour connaître la fréquence de départ (STARt?), la fréquence de fin (STOP?), la RBW (RESolution?), la VBW (VIDeo?), le temps de balayage (SWEep:TIME?), etc. Ces valeurs sont consignées dans un fichier de log, afin de tracer les réglages effectifs utilisés.
3. **Données de Trace et journaux**
   * Une fois les traces configurées (mode, détecteur, etc.), on lance :TRACe:DATA? {index} pour extraire les points de mesure, puis on les sauvegarde au format texte (.txt). En parallèle, le journal (log) est mis à jour pour tracer la fréquence de départ, la BW, etc.
   * Cela facilite la traçabilité et la relecture des mesures.

Grâce à ce dispositif, on **évite** les erreurs de configuration (fichier absent, connectivité défaillante, etc.) **avant** l’envoi des commandes et on **assure** la bonne exécution des mesures (lecture des vrais paramètres, récupération des traces) **après** l’envoi, constituant ainsi un processus de contrôle et de validation assez complet.

## 1.19 Points clés

Au fil des explications sur l’ordonnancement des commandes SCPI et leur validation, notre projet couvre les aspects clés suivants :

1. **Intégration entre .sta et SCPI**
   * Les tags <Frequency>, <Ampt>, <BW>, <Sweep>, <Trace>, <Limit>, <Peak>, <EMI> d’un fichier .sta sont correctement reliés aux commandes SCPI.
   * L’approche “**parsing + driver**” génère et envoie automatiquement les commandes, sans inscription en dur d’un modèle précis dans le code.
2. **Factory driver pour plusieurs appareils**
   * Grâce au pattern “Factory”, SiglentSSA3000XPlus et HMS1000 ont chacun leurs méthodes load\_config(), save\_data(), transfer\_csv\_to\_pc(), couvrant leurs SCPI respectifs.
   * Le même flux de mesure (parse → load → measure → save) reste réutilisable pour d’autres gammes, en créant simplement un nouveau driver.
3. **Contrôles avant l’envoi**
   * On vérifie l’existence du fichier .sta, le nombre de signaux, la liaison USB/LAN (verify\_usb\_connexion, verify\_connection\_dynamique). Toute incohérence provoque un arrêt et un message.
   * Lorsque la RBW ou d’autres paramètres sortent de la plage disponible, le code peut corriger et alerter l’utilisateur.
4. **Vérifications après l’envoi**
   * \*OPC? ou un temporisateur assure que le balayage est réellement terminé avant la lecture des résultats,
   * Lecture et stockage dans le journal des fréquences de départ/fin, de la RBW, de la VBW, etc., pour la traçabilité,
   * Extraction des données de trace (TRACe:DATA?) par mode de détection, puis sauvegarde en CSV ou TXT.
5. **Journalisation et traçabilité**
   * Une fois la mesure finie, le code enregistre la configuration courante de l’instrument (fréquences, largeurs de bande, détecteur, etc.) dans un log, de même que les données de mesure (CSV/TXT). Ainsi, chaque campagne de test est **traçable**.

Sur la base de ces éléments, le système offre un cycle complet et cohérent : **lecture du .sta → construction des commandes SCPI → contrôle en amont → mesure + validation → archivage**.

## 1.20 En résumé

La construction et la validation des commandes SCPI sont présentes à chaque étape de l’automatisation des essais CEM. Dans notre projet, nous avons non seulement géré différents formats de fichiers .sta et mis en correspondance les ordres SCPI pour plusieurs gammes d’appareils, mais également implémenté plusieurs niveaux de vérification (fichier, paramétrage, connectivité) et, post-mesure, un archivage structuré (logs, données).

Sur cette base, il est encore possible de renforcer la fiabilité des connexions USB/LAN et de gérer un plus grand nombre d’appareils simultanément, afin d’offrir une **robustesse** et une **extensibilité** accrues dans un contexte industriel exigeant.

# 2. Méthodes de connexion USB et LAN

Dans cette seconde partie, nous nous intéressons aux différentes façons de communiquer avec l’analyseur de spectre (ou d’autres appareils CEM) via USB et LAN, ainsi qu’à la gestion pratique de la disponibilité du réseau ou du périphérique. Nous aborderons également la gestion des temps de réponse (*timeouts*), la détection d’erreurs (valeurs hors plage, appareil indisponible, etc.), et la cohabitation de plusieurs instruments sur le même réseau.

## 2.1 Connexion USB

### 2.1.1 Détection et initialisation du périphérique

* **Identification du matériel** Une fois le câble USB branché, le logiciel interroge la ressource via un \*IDN?. L’analyseur doit renvoyer un ID compatible (ex. Siglent Technologies, SSA3...).
* **Ouverture de la ressource** Le gestionnaire VISA (ou équivalent sous Python, par ex. *pyvisa*) liste les ressources disponibles. Si l’adresse USB correspondante est trouvée, le code appelle la fonction usb\_instrument.initialize().
* **Gestion des événements Windows** Dans l’interface graphique, il existe un **listener** (“device arrival/removal”) qui permet de savoir si un appareil USB vient d’être branché ou déconnecté. Lorsqu’un nouveau périphérique est inséré, on tente automatiquement la connexion ; s’il est retiré, le logiciel met à jour l’état pour refléter la perte de l’instrument.

### 2.1.2 Timeouts et indisponibilité

* **Configuration du timeout** Lorsque l’on envoie une commande à l’analyseur, on définit un temps maximum de réponse (ex. 10 secondes). Si l’appareil met trop de temps ou ne répond pas, on lève une exception et signale “Instrument Timeout” au logiciel.
* **Réinitialisation** En cas de non-réponse prolongée, il est possible de réinitialiser l’état interne (usb\_instrument.reset\_internal\_state()) et de retenter usb\_instrument.initialize().
* Cas d’erreurs
  + Si l’analyseur n’est pas allumé ou si le driver USB est indisponible, la fonction de connexion échoue immédiatement.
  + Si un paramètre hors plage provoque une erreur SCPI, le code peut renvoyer une exception “Command parameter error” que l’on journalise.

## 2.2 Connexion LAN

### 2.2.1 Paramétrage de l’adresse IP

* **Adresses statiques** Dans notre scénario, on considère que chaque analyseur a une IP fixe (ex. 192.168.100.2, 192.168.100.3, etc.). Le logiciel propose à l’utilisateur de renseigner ces IP dans l’UI, puis stocke ces informations (ex. Nom\_Appareil + Adresse\_IP) dans une liste.
* **Vérification de la ressource** Une fois l’IP connue, la séquence d’initialisation ouvre la ressource VISA du type TCPIP0::192.168.100.X::inst0::INSTR. Comme en USB, on exécute ensuite un \*IDN? pour confirmer qu’il s’agit bien du bon appareil.
* Déconnexion / Reconnexion
  + Si l’utilisateur supprime un appareil de la liste ou si le LAN n’est plus accessible, on appelle lan\_instrument.close() pour fermer proprement la ressource.
  + Si la connexion échoue, on réinitialise le gestionnaire VISA et on réessaie.

### 2.2.2 Cohabitation de plusieurs appareils

* **Adresses IP distinctes** Pour connecter plusieurs analyseurs en même temps, chacun doit avoir une IP unique dans le même sous-réseau (ex. 192.168.100.2, 192.168.100.3, etc.). Le logiciel maintient en mémoire une liste de (Nom\_Appareil, IP).
* **Gestion des priorités** En principe, les commandes s’exécutent de façon séquentielle. Si plusieurs instruments doivent être commandés simultanément, le code peut interroger chaque IP dans un ordre prédéfini.
* **Détection de panne réseau** Si l’un des appareils est indisponible (ping infructueux, pas de réponse \*IDN?), la logique de connexion renvoie un statut d’erreur. L’utilisateur peut décider de retenter ou d’ignorer l’appareil indisponible.

## 2.3 Gestion des temps de réponse et détection d’erreurs

### 2.3.1 Timeouts dans les échanges SCPI

* **Commandes bloquantes** Lorsqu’on envoie un SCPI du type :INITiate:IMMediate, on peut soit attendre le sweep en effectuant un \*OPC?, soit définir un délai (timeout). Si l’appareil met plus longtemps que prévu à répondre, le logiciel lève une exception.
* **Boucles de vérification** Dans certains cas (mesures longues, par ex. FFT avec un grand nombre de points), on interroge l’instrument à intervalles réguliers avec \*OPC? pour voir quand l’opération est terminée. Si la réponse ne revient jamais, on notifie “Timeout”.

### 2.3.2 Valeurs hors plage et appareils indisponibles

* **Limites en local** Dans nos fonctions load\_config(), si l’utilisateur demande une RBW ou un Span hors tolérance, le code le détecte (ex. calcul de la valeur la plus proche, ou génération d’erreur).
* **Détection d’erreurs SCPI** Après envoi de certaines commandes critiques, il est possible d’interroger :SYSTem:ERRor?. Si la réponse n’est pas +0, on signale l’erreur rencontrée (ex. “Settings conflict”, “Parameter error”, etc.).
* Gestion “indisponible”
  + Si l’appareil est éteint, hors ligne ou débranché, la requête VISA échoue immédiatement (exception). Le code capture l’exception et place l’instrument en statut “indisponible”.
  + Si plusieurs tentatives échouent, on propose de supprimer ou de reconfigurer l’appareil concerné.

## 2.4 Cas d’usages simultanés (plusieurs instruments)

### 2.4.1 Identification des différentes ressources

* **USB** Chaque analyseur connecté en USB apparaît comme une ressource distincte, souvent notée USB0::...::INSTR. On associe un “Nom de l’appareil” pour l’afficher dans l’UI.
* **LAN** Pour le réseau, chaque IP donne un accès unique ; la commande lan\_instrument.initialize(ip\_address) ouvre la ressource correspondante. Dans le logiciel, on peut avoir plusieurs entrées (Nom\_Appareil, IP).

### 2.4.2 Conflits et priorités

* **Planification de l’accès** En mode séquentiel, on envoie les ordres SCPI à l’un, on attend la fin, puis on passe au suivant. Les ressources VISA sont indépendantes, donc pas de conflit direct.
* **Partage du même LAN** Du moment que les IP ne se chevauchent pas, il n’y a pas de problème. Cependant, si la bande passante réseau est faible ou si plusieurs balayages se lancent en parallèle, la réception des données CSV peut prendre plus de temps.
* **Surveillance** Dans l’interface, on affiche l’état de chaque instrument (connecté, en attente, mesure en cours, etc.). Si l’un des instruments répond plus lentement, on signale une possible congestion ou un timeout.

## 2.5 Conclusion sur la connexion et la gestion d’erreurs

En intégrant ces méthodes de connexion (USB et LAN) et en gérant soigneusement les *timeouts*, la détection d’erreurs SCPI, ainsi que la cohabitation de multiples appareils, notre application couvre un large spectre de scénarios :

1. **Connexion USB** : branchement “plug & play”, détection automatique, écoute d’événements (arrivée / retrait).
2. **Connexion LAN** : configuration IP fixe, vérification via \*IDN?, gestion d’un ensemble d’appareils sur le réseau.
3. **Temps de réponse** : time.sleep() fixe ou boucle \*OPC? pour les mesures longues, avec un *timeout* clair.
4. **Erreurs** : interception des réponses hors plage, statut SYSTem:ERRor? et exceptions VISA.
5. **Multi-appareils** : identification par IP distinctes ou par canaux USB différents, planification de l’accès, affichage dans l’UI.

Ce dispositif garantit une **robustesse** adéquate dans les environnements de test CEM, où l’on peut être amené à brancher/débrancher régulièrement des analyseurs, ou à enchaîner plusieurs mesures sur différentes configurations réseau.

# 3. Centralisation de la gestion des erreurs

Pour renforcer la robustesse d’une application en environnement industriel, il est nécessaire d’établir un mécanisme centralisé chargé d’enregistrer et de gérer tous les types d’erreurs possibles (problèmes de connexion, paramètres hors tolérances, timeouts, etc.). L’objectif est d’éviter une profusion de pop-ups ou d’exceptions non interceptées, tout en offrant à l’utilisateur une vue claire et unifiée des problèmes survenus.

## 3.1 Principes de robustesse en milieu industriel

1. **Déconnexions prévisibles**
   * Dans un contexte de production CEM, le câble USB peut être retiré par inadvertance, ou un appareil LAN peut devenir indisponible.
   * La solution doit “tolérer” ces évènements sans faire planter le programme : les erreurs sont consignées, mais le logiciel continue de fonctionner. Il peut tenter une reconnexion, avertir l’opérateur ou, selon les cas, ignorer l’appareil en panne.
2. **Gestion fluide de la reconnexion**
   * Si un instrument est reconnecté, le module de gestion d’erreurs doit permettre d’amorcer un nouveau cycle d’initialisation, sans rester bloqué dans un état d’erreur.
   * On réinitialise l’objet représentant l’instrument ou la ressource VISA, puis on contrôle de nouveau l’état du matériel en envoyant \*IDN?.
3. **Traçabilité des incidents**
   * Même si certains problèmes sont traités de manière “silencieuse” (pas d’affichage intrusif), ils doivent être inscrits dans un journal, avec date/heure, code d’erreur et brève description, de façon à ce que l’équipe de maintenance puisse reconstituer l’historique des incidents.

## 3.2 Module dédié aux codes d’erreurs

### 3.2.1 Architecture globale

* **Classe ou module “StatusManager”**
  + Contient un champ status (masque de bits ou entier) représentant le cumul des erreurs (p. ex. ERROR\_USB\_CONNEXION\_FAILED, ERROR\_MEASUREMENT\_FAILED, ERROR\_OPC\_FAILED, etc.).
  + Tient à jour un dictionnaire error\_messages indexé par code d’erreur.
  + Fournit des méthodes comme set\_error(code, message), clear\_error(code), reset\_error(), etc.
* **Catégories courantes d’erreurs**
  + **Erreurs de connexion** : *appareil non détecté, IDN non conforme, driver indisponible, etc.*
  + **Erreurs de paramétrage SCPI** : *valeurs hors plage, mode incompatible, paramètre manquant, etc.*
  + **Erreurs de temps de réponse** : *timeout, balayage trop long, pas de réponse de l’appareil, etc.*
  + **Erreurs de transfert** : *fichier CSV introuvable, fichier STA invalide, etc.*

Chaque catégorie se voit associer un code d’erreur unique (par ex. ERROR\_USB\_CONNEXION\_FAILED = 0x400, ERROR\_CONFIG\_LOAD\_FAILED = 0x4, etc.).

### 3.2.2 Gestion des exceptions et affichage contrôlé

* **Accumulation et lecture**  
  Lorsqu’une anomalie survient, on appelle status\_manager.set\_error(...) au lieu de générer une exception non gérée. Cela évite que l’application ne se bloque.
* **Centralisation de l’information**  
  En fin de processus, on peut n’afficher qu’une seule fenêtre récapitulative ou regrouper l’ensemble des erreurs dans un panneau d’interface.
* **Filtrage des erreurs**  
  Selon le niveau de gravité, on adopte des mesures différentes :
  + *Erreur mineure* (correction automatique d’un paramètre) : journaliser un avertissement, mais ne pas forcément alerter l’opérateur.
  + *Erreur critique* (appareil indisponible) : bloquer l’opération et demander une reconnexion ou l’abandon.

## 3.3 Éviter la prolifération de pop-ups

1. **Agrégation des messages**
   * Lorsque survient une erreur, le module “StatusManager” l’ajoute au champ status au lieu de faire surgir immédiatement une boîte de dialogue.
   * À la fin de la séquence de mesure, on peut afficher un **rapport global** des erreurs détectées.
2. **Boîte de dialogue unique**
   * Cette fenêtre mentionne tous les soucis rencontrés (p. ex. “Timeout lors de l’analyse du fichier .sta”, “Atténuation hors plage”, “Instrument en timeout”), évitant ainsi au technicien de devoir cliquer sur une multitude de pop-ups “OK”.
   * L’opérateur voit l’ensemble des problèmes et choisit comment réagir (retenter, ignorer, déconnecter l’appareil, etc.).
3. **Journal en lieu et place d’une fenêtre**
   * De nombreux avertissements (p. ex. “la RBW a été ajustée de 2500 Hz à 3000 Hz”) n’exigent pas l’affichage d’une boîte de dialogue. Une simple ligne dans le log suffit pour les retrouver par la suite.

## 3.4 Gestion des déconnexions et reconnexions

### 3.4.1 Exemple de connexion USB

1. **Connexion** : Windows envoie un événement “DEVICEARRIVAL”, le code appelle Connexion\_USB(...) puis envoie \*IDN?.
2. **Déconnexion inopinée** : si le câble USB est retiré pendant une mesure, la commande SCPI échoue et le module d’erreurs reçoit ERROR\_USB\_CONNEXION\_FAILED.
3. **Traitement logiciel** :
   * Le code interrompt la mesure en cours, marque l’instrument comme “indisponible”.
   * Si l’utilisateur rebranche le câble, un nouvel événement “DEVICEARRIVAL” relance la séquence d’initialisation.
4. **Éviter un crash** : grâce à la centralisation des erreurs, aucune exception fatale ne reste en suspens, et l’application demeure stable.

### 3.4.2 Exemple de connexion LAN

1. **Appareil hors tension ou IP erronée** :
   * Un ping ou un \*IDN? échoue, déclenchant ERROR\_LAN\_CONNEXION\_FAILED.
2. **Réinitialisation** :
   * Le module “StatusManager” enregistre l’erreur et invite l’opérateur à vérifier l’IP ou le réseau.
3. **Reconnexion** :
   * Si l’appareil est rallumé (en conservant la même IP), on rouvre la liaison. Une fois la connexion établie, le code efface l’erreur (status\_manager.clear\_error(...)).

## 3.5 Conclusion sur la centralisation de la gestion des erreurs

Grâce à un module dédié tel que “StatusManager” :

* Les erreurs sont **catégorisées** (connexion, paramètres SCPI, délais de réponse, etc.) et **cumulées** au lieu de se traduire par des exceptions disparates ou des pop-ups multiples.
* L’utilisateur bénéficie d’une vue **unifiée** des anomalies via une boîte de dialogue globale ou un panneau sur l’interface.
* Les scénarios industriels de **déconnexion / reconnexion** sont **anticipés** : si l’appareil devient “indisponible”, on peut le réintégrer facilement et poursuivre le fonctionnement sans interrompre l’ensemble du système.

Cette approche confère une **robustesse** accrue et une expérience utilisateur plus fluide, même face aux aléas de câbles débranchés, de pannes réseau ou de redémarrages d’appareils.

# 4. Organisation et traçabilité des données

En matière de Compatibilité Électromagnétique (CEM), la **traçabilité** est cruciale : il est indispensable de pouvoir identifier clairement chaque mesure, chaque paramètre de configuration et chaque résultat associé. L’application présentée s’appuie sur une **arborescence de répertoires** dédiée, des **fichiers de journal détaillés**, et une **distinction entre logs techniques et erreurs opérationnelles**, afin de satisfaire pleinement à cet impératif.

## 4.1 Structure standardisée des données

Lors de l’exécution des essais, l’application crée ou utilise divers **sous-répertoires** au sein d’un **répertoire principal** défini par l’utilisateur. Cette organisation permet de repérer immédiatement la nature des différents fichiers :

1. **LOG/**
   * Contient les journaux décrivant la configuration des essais : informations sur le dispositif sous test (DUT), configuration matérielle et logicielle, paramètres de mesure (fréquence, détecteur, etc.).
   * Le principal fichier de log (souvent nommé log\_file.txt) y recense de manière détaillée la liste des signaux mesurés et les réglages effectivement appliqués à l’analyseur (fréquences de départ / d’arrivée, RBW/VBW, temps de balayage, détecteur activé, etc.).
2. **Measures\_data/**
   * Renferme les fichiers de données brutes extraits de l’analyseur, sous forme de texte (par ex. P12\_AV.txt, P12\_PPK.txt) ou de CSV.
   * Chaque fichier correspond aux résultats de mesure d’un **signal** sur une **trace** (ou un mode de détection) particulier.
3. **LTSpice/**
   * Après l’acquisition, certains fichiers peuvent être **fusionnés** (p. ex., plusieurs traces similaires) en un jeu de courbes simplifiées.
   * Le répertoire LTSpice recueille ces fichiers agrégés (p. ex. P12\_AV\_merged.txt), exploitables par des outils de simulation ou de post-traitement.
4. **report/**
   * Regroupe le rapport final d’essai (généralement un .xlsx), qui inclut les résultats retraités, les limites comparées et le niveau de conformité.
   * À l’issue des mesures, ce dossier contient par exemple Report\_file.xlsx, présenté au service qualité ou aux clients.

Avec cette **organisation par sous-répertoires**, l’opérateur sait facilement où chercher :

* Les **journaux texte** (paramètres de config),
* Les **données brutes** (pour réexaminer une mesure),
* Les **fichiers consolidés** (pour des simulations ou analyses complémentaires),
* Le **rapport final** (pour la validation des résultats).

## 4.2 Le fichier de log : pierre angulaire de la traçabilité

Le fichier log\_file.txt, placé dans le répertoire LOG/, constitue le noyau de la traçabilité. Au fil de chaque campagne de mesures, l’application y ajoute :

1. **Informations sur le DUT et son environnement**
   * Numéro ou nom du DUT (p. ex. “Inverter IV3”),
   * Configuration matérielle (gammes de tension, puissance, version hardware...),
   * Configuration logicielle (version firmware, AutoSAR...),
   * Liste des équipements associés (analyseur de spectre, sonde RF, etc.).
2. **Paramètres d’essai**
   * Type de test (ex. “émissions conduites CEM”),
   * Signaux mesurés (p. ex. “P12” ou “M12”),
   * Fichier de configuration chargé (.sta).
3. **Paramètres de mesure effectivement appliqués**
   * Fréquences de départ / d’arrivée, pas de fréquence,
   * Temps de balayage (automatique ou manuel),
   * RBW, VBW, ratio VBW/RBW, atténuation, préamplification,
   * Traces activées, mode de détection (Peak, Average, Quasi-peak), nombre de moyennes,
   * Ajustements ou corrections automatiques (p. ex. “RBW modifiée de 2500 Hz à 3000 Hz”).
4. **Informations complémentaires**
   * Écart entre la commande et la valeur réellement appliquée (p. ex. “le temps de balayage a été recalculé en mode Auto”).

**Exemple de log** :

DUT id: Inverter IV3

HW config: 270-450V, 150 kW, 500 Arms 30s

SW config: AutoSAR 4.2.2

Equipment: Analyseur de spectre SSA 3032X Plus

Test - type de mesure: CEM Émission Conduite

Signal: P12

Measurement Settings from F:/.../test11111.sta

Start Frequency: 80000000.000000 Hz

Stop Frequency: 110000000.000000 Hz

Step Frequency: 40000.000000 Hz

Sweep Time: 0.0372 seconds (Auto)

Note: The Sweep Time value shown above is generated by the system in Auto mode and may not represent the actual user-defined value.

Sweep Mode: AUTO

Attenuation: 20.00 dB

Pre-Amplifier: OFF

RBW: 100000.00 Hz

VBW: 300000.00 Hz

VBW/RBW Ratio: 3.00

Filter Type: GAUSS

Enabled Traces:

- Trace1\_Info: Display, Detector Mode: POS, Average Count: 100

- Trace2\_Info: Display, Detector Mode: NEG, Average Count: 100

- Trace3\_Info: Display, Detector Mode: QUAS, Average Count: 100

- Trace4\_Info: Display, Detector Mode: AVER, Average Count: 100

On voit ainsi la correspondance entre l’intention de configuration de l’opérateur et les valeurs **réellement** appliquées par l’analyseur.

## 4.3 Principes de traçabilité et avantages pour la conformité

Grâce à ces mécanismes, la solution répond aux **exigences de traçabilité** en contexte CEM :

1. **Chaque mesure** est associée :
   * à un **fichier de configuration** clairement identifié (.sta),
   * à un ou plusieurs **fichiers de données brutes** (dans Measures\_data/),
   * à un **journal** synthétique, consignant les paramètres chargés et les mesures réalisées.
2. **L’architecture de stockage** (dossiers hiérarchisés) garantit :
   * la capacité à répertorier facilement les fichiers générés,
   * une séparation nette entre logs, données brutes, fichiers consolidés et rapports.
3. **La cohérence entre commande et état réel de l’instrument** est vérifiée :
   * le fichier .sta contient les consignes,
   * le fichier de log, via SCPI, confirme les paramètres chargés (fréquences de départ/fin, RBW effective, etc.),
   * l’historique des erreurs facilite la recherche de cause et les audits.
4. **Le rapport final** (au format Excel, PDF, etc.) reprend les données clés (limites, marges, conformité) tout en renvoyant vers **le log détaillé** pour un contrôle plus approfondi.

**En résumé**

L’application atteint une **gestion de la traçabilité** efficace grâce à :

* **Une structure de stockage normalisée**,
* **Un fichier de log centralisé et exhaustif**,
* **Une gestion claire et cohérente des erreurs**.

Ces caractéristiques offrent non seulement une facilité d’utilisation pour l’opérateur, mais aussi une **robustesse** et une **conformité** indispensables en contexte industriel.

## 5. Conclusion & Perspectives

À travers les quatre volets abordés (**parsing et génération des commandes SCPI**, **pilotage USB/LAN**, **traitement centralisé des erreurs**, **organisation et journalisation des données**), nous disposons d’une solution fiable et efficace pour automatiser les mesures sur plusieurs analyseurs de spectre en environnement industriel.

* **Bilan** :
  + Un modèle en **factory driver** pour adapter différents formats .sta (XML/binaire) et produire un dictionnaire parameters unifié,
  + Une logique d’envoi de commandes isolée dans chaque classe driver (Siglent/HMS, etc.), évitant le “hard-coding” et préservant la maintenabilité,
  + Un module de **gestion des erreurs** limitant l’apparition intempestive de pop-ups, autorisant la reconnexion automatique et la correction des paramètres, et offrant une synthèse claire des problèmes,
  + Un système de **logging et d’arborescence** clairement défini, pour garantir la traçabilité des résultats de mesure.
* **Perspectives** :
  + **Étendre à d’autres constructeurs et fonctions** : en reprenant la même approche, on peut intégrer des pilotes pour Keysight, R&S, Rigol, etc., et gérer plus de champs .sta (EMI, ACPR, OBW, Marker2...).
  + **Renforcer la robustesse en production** : proposer une stratégie de récupération automatique en cas de timeout ou de conflit de ressources, avec plus de flexibilité.

Ainsi, cette *note d’application* a passé en revue la **construction et la validation des commandes SCPI**, la **gestion USB/LAN**, la **centralisation des erreurs** et la **traçabilité des données**, offrant une base technique solide pour l’industrialisation des essais CEM. Nous espérons qu’elle saura guider les lecteurs dans la mise en place de systèmes de mesure automatisée performants et évolutifs.