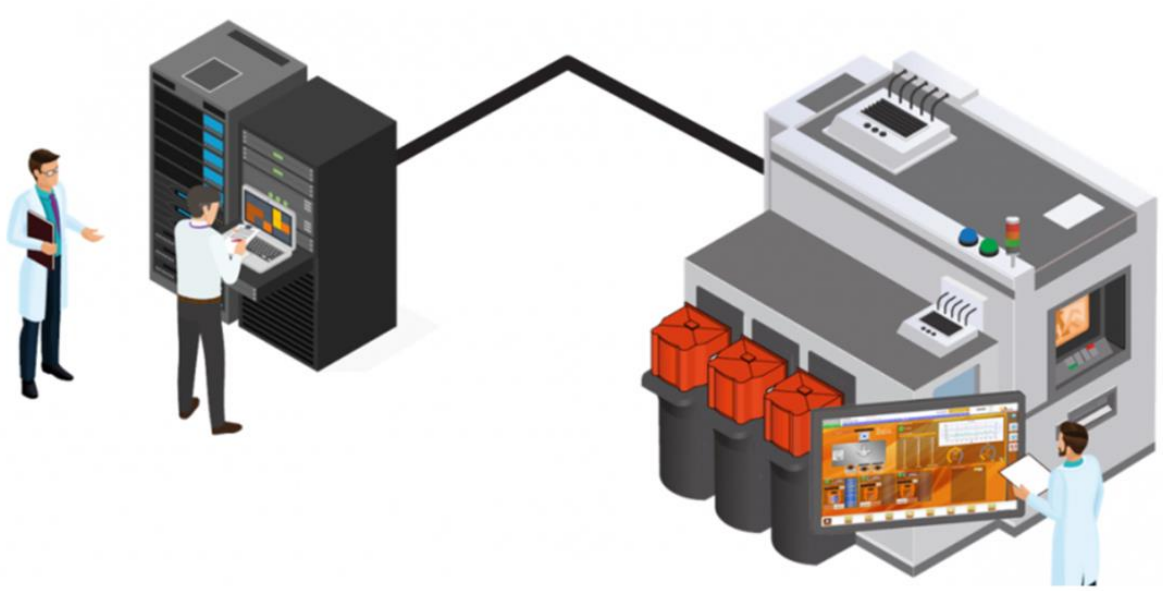


Projet Solar Panel (Huawei SUN2000)
Category

Date : 15/03/2025
www.agileo-automation.com

Purpose

This document describes the "SolarPanel" project, a C# WPF application developed to communicate with a Huawei SUN2000 inverter via the Modbus TCP protocol. The objective is to promote a responsible approach to energy by visualizing the performance of the solar installation in real time.

Ce document décrit le projet "SolarPanel", une application C# WPF développée pour communiquer avec un onduleur Huawei SUN2000 via le protocole Modbus TCP. L'objectif est de promouvoir une approche responsable de l'énergie en visualisant les performances de l'installation solaire en temps réel.

Scope

This document covers the technical aspects of the project, including the MVVM architecture, use of the NModbus library, Modbus register management, and the WPF user interface.

Ce document couvre les aspects techniques du projet, y compris l'architecture MVVM, l'utilisation de la bibliothèque NModbus, la gestion des registres Modbus, et l'interface utilisateur WPF.

Out Of Scope

This document does not cover deployment or maintenance aspects of the application.

Ce document ne couvre pas les aspects de déploiement ou de maintenance de l'application.

Disclosure and Confidentiality

This document is Agileo Automation Company private. It contains the intellectual property of Agileo Automation and may not be copied, modified, or otherwise duplicated, reproduced, or changed without the expressed written permission of Agileo Automation.

This document may only be distributed on a need-to-know basis and may not be disclosed to any other party without the written permission of Agileo Automation.

Intellectual Property

Agileo Automation Component xxx use is subject to a license

Contact information

If there are any questions, clarifications, or requests for additional information regarding this document, please contact the Agileo Automation Support Team: *support@agileo-automation.com*

Electronic mail is the preferred communication method.

Revision History

Release	Modified parts	Comments	Date	Author
01.00	All	Initial Document	15/03/25	Agileo

Glossary

Term or Abbreviation	Description
Modbus TCP	Protocole de communication utilisé pour interagir avec l'onduleur SUN2000.
MVVM	Modèle d'architecture logicielle (Model-View-ViewModel).
WPF	Windows Presentation Foundation, framework pour les interfaces utilisateur.
NModbus	Bibliothèque .NET pour la communication Modbus.

Table of Contents

1 Introduction

Dans le cadre de mon stage de 6 semaines au sein de l'entreprise **AGILEO Automation**, j'ai travaillé sur le projet « **SolarPanel** ». Ce projet consiste en le développement d'une **application C#** WPF avec une architecture **MVVM** (Model-View-ViewModel) permettant la communication avec un onduleur Huawei SUN2000 via le protocole **Modbus TCP**.
Ce projet a pour objectif de promouvoir une approche **responsable** de l'énergie en mettant en avant l'efficacité de l'énergie photovoltaïque auprès des collaborateurs d'AGILEO. Également, la visualisation continue des performances de l'installation solaire en affichant la consommation énergétique ainsi que des données essentielles en temps réel permet d'assurer **le suivi et la garantie** des équipements solaires.

2 Diagramme de classe

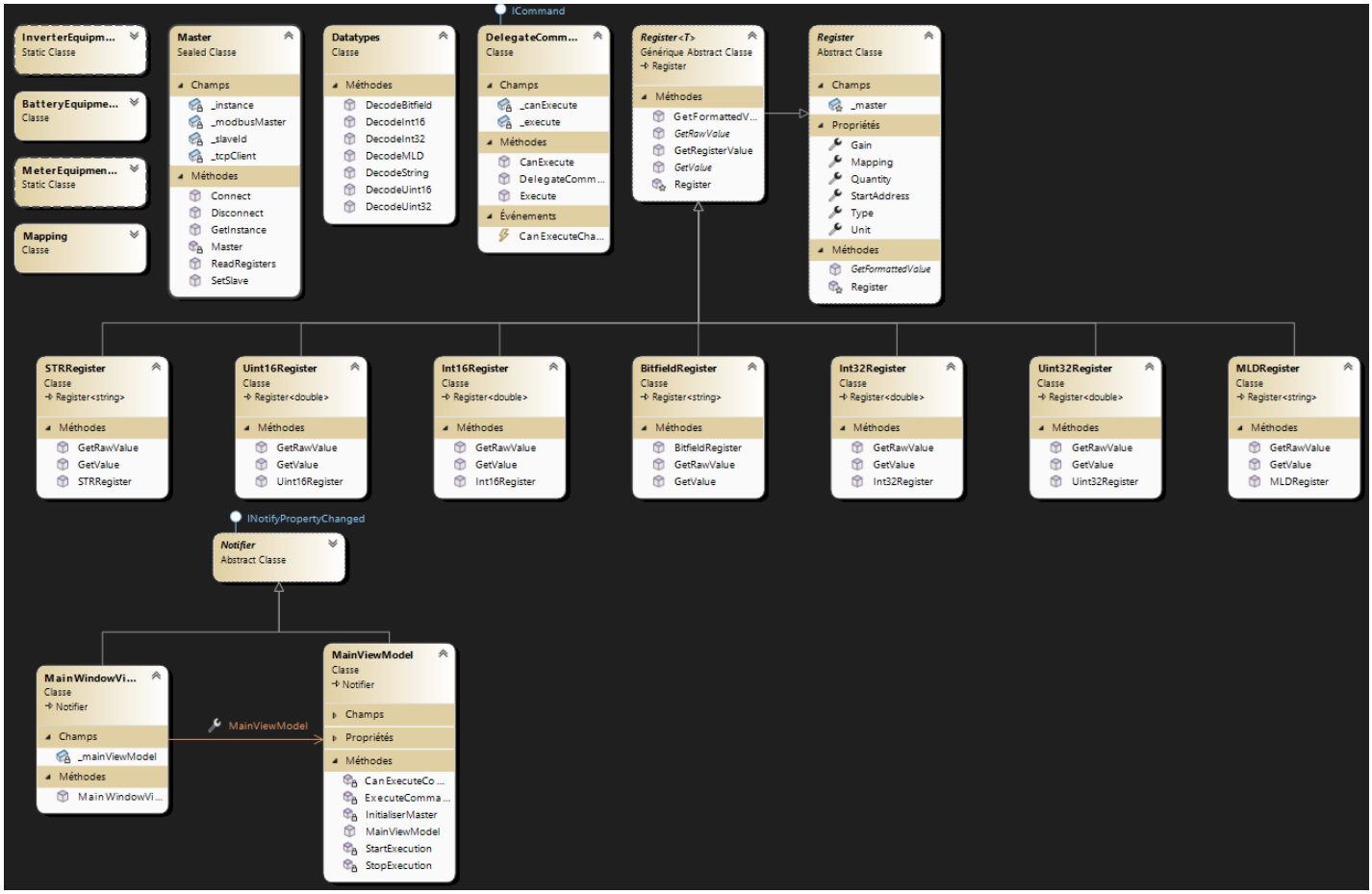
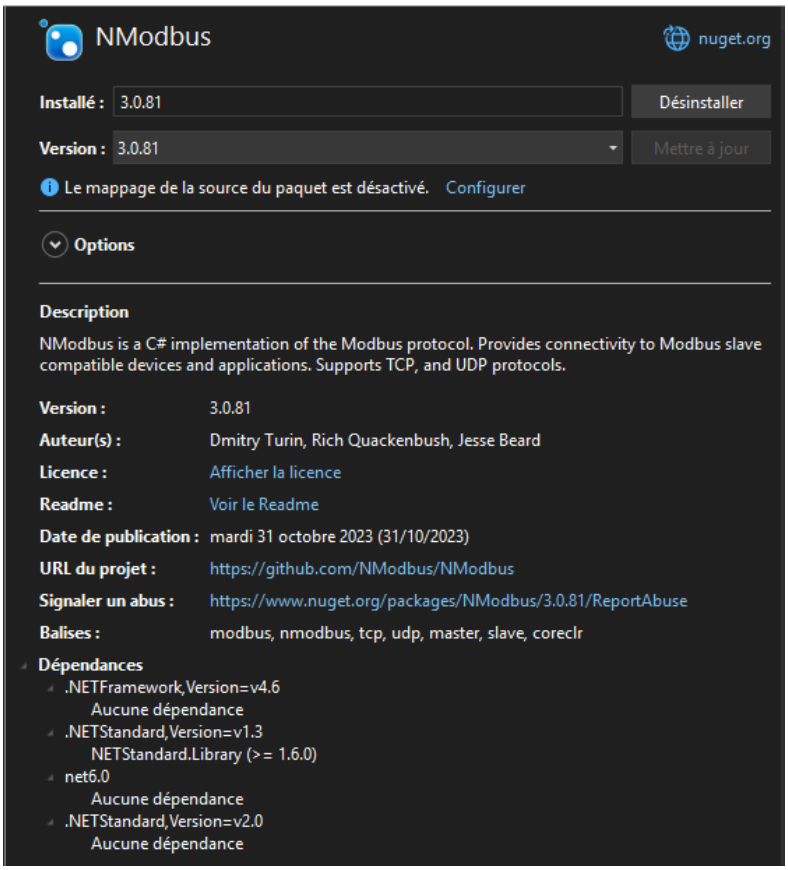


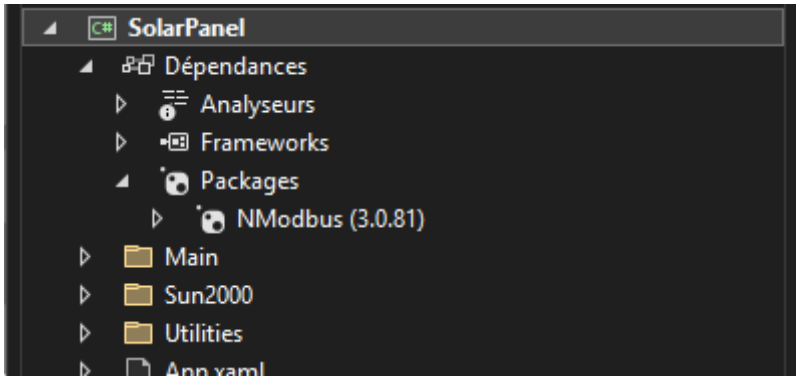
Diagramme de classe du projet SolarPanel représentant les différentes classes permettant de récupérer, traiter et afficher les valeurs de mesure de l'onduleur SUN2000.

3 Package NuGet : NModbus



NModbus est une bibliothèque (librairie) .NET permettant la communication via le protocole Modbus. Elle est utilisée pour interagir avec des équipements industriels comme les automates programmables (PLC), les onduleurs, les compteurs d'énergie et autres dispositifs prenant en charge Modbus.

Voir documentation NModbus => https://nmodbus.github.io/api/NModbus.html
Voir GitHub NModbus => https://github.com/NModbus/NModbus



4 Master

```
using System.Net.Sockets;
using NModbus;
using Sun2000_modbus.Registers;

namespace Sun2000_modbus.ModbusCommunication
{
    public sealed class Master
    {
        private IModbusMaster _modbusMaster;
        private byte _slaveId;
        private TcpClient _tcpClient;
        private static Master? _instance;

        //singleton master
        private Master() { }
        //Set the instance of master
        public static Master GetInstance()...

        //Set the slave number
        public void SetSlave(byte slaveId)...

        //Create the connection to the hôst
        public void Connect()...

        //Set the disconnection
        public void Disconnect()...

        //Execute the reading of register according to the startAdress and the quantity
        public ushort[] ReadRegisters(ushort startAdress, ushort quantity)...

        //Execute the reading of register according to an Register object
        public ushort[] ReadRegisters(Register register)...
    }
}
```

La classe Master permet de se connecter/déconnecter à l'onduleur Huawei SUN2000 en créant une connexion CP et de pouvoir communiquer avec celui-ci en interrogeant les registres Modbus.

```
        public void SetSlave(byte slaveId)...

        //Create the connection to the hôst
        public void Connect()
        {
            _tcpClient = new TcpClient("192.168.1.28", 502);
            Thread.Sleep(1000);
            var factory = new ModbusFactory();
            _modbusMaster = factory.CreateMaster(_tcpClient);
        }

        //Set the disconnection
        public void Disconnect()...
```

La méthode ***Connect()*** permet de créer la connexion. C'est ici que l'on indique l'adresse IP et le port de l'hôte distant auquel nous voulons nous connecter.

```
        public void Connect()...

        //Set the disconnection
        public void Disconnect()
        {
            _tcpClient.Close();
        }

        //Execute the reading of register according to the startAdress and the quantity
        public ushort[] ReadRegisters(ushort startAdress, ushort quantity)...
```

La méthode ***Disconnect()*** permet de fermer la liaison.

```
        public void Disconnect()...

        //Execute the reading of register according to the startAdress and the quantity
        public ushort[] ReadRegisters(ushort startAdress, ushort quantity)...

        //Execute the reading of register according to an Register object
        public ushort[] ReadRegisters(Register register)...
    }
}
```

4.1 ReadRegisters()

Les méthodes ***ReadRegisters()*** permettent de récupérer les données des registres selon l'adresse Modbus de départ et la quantité de registres. Il y a une surcharge de la méthode permettant de rentrer un type de paramètre différent.

```
        //Execute the reading of register according to the startAdress and the quantity
        public ushort[] ReadRegisters(ushort startAdress, ushort quantity)
        {
            try
            {
                return _modbusMaster.ReadHoldingRegisters(_slaveId, startAdress, quantity);
            }
            catch
            {
                return [0];
            }
        }

        //Execute the reading of register according to an Register object
        public ushort[] ReadRegisters(Register register)
        {
            try
            {
                return _modbusMaster.ReadHoldingRegisters(_slaveId, register.StartAddress, register.Quantity);
            }
            catch
            {
                return [0];
            }
        }
```

Ces méthodes exécutent ***ReadHoldingRegisters()*** de la bibliothèque importée NModbus qui interroge les registres de l'hôte distant et retourne le résultat dans un tableau d'entiers de type ushort. En cas de problèmes lors de l'exécution, renvoie une valeur de [0].

```
public ushort[] ReadRegisters(ushort startAdress, ushort quantity)
{
    try
    {
        return _modbusMaster.ReadHoldingRegisters(_slaveId, startAdress, quantity);
    }
    catch
    {
        return [0];
    }
}
```

```
public override double GetRawValue()
{
    return Datatypes.DecodeInt16(_master.ReadRegisters(StartAddress, Quantity));
}
```

```
PVAll = master.ReadRegisters(InverterEquipmentRegister.PVAll), //
RatePower = InverterEquipmentRegister.RatePower.GetRawValue(),
ModelValue = InverterEquipmentRegister.Model.GetFormattedValue(),
```

ReadRegisters() est appelée sur l'objet « Rate Power » pour directement retourner une valeur de donnée lisible.

ReadRegisters(Register registre) est appelé sur l'objet « master » et prend en paramètre un objet de type

```
public ushort[] ReadRegisters(Register register)
{
    try
    {
        return _modbusMaster.ReadHoldingRegisters(_slaveId, register.StartAddress, register.Quantity);
    }
    catch
    {
        return [0];
    }
}
```

Register pour retourner plusieurs valeurs sous forme d'un tableau de type ushort non lisible.

```
PhaseTotal = master.ReadRegisters(InverterEquipmentRegister.PhaseTotal),
EnergyYield = master.ReadRegisters(InverterEquipmentRegister.EnergyYeld),
PVAll = master.ReadRegisters(InverterEquipmentRegister.PVAll), //Read PV
```

4.2 Groupe de registres

L'objet de type Register entré en paramètre est un objet qui recense plusieurs registres formant un groupe. Cela permet de récupérer les valeurs des différents registres en une seule requête afin d'éviter de multiples exécutions de requêtes et de récupérer toutes les données sans décalage temporel.

```
//Read PhaseAVoltage to InsulationResistance registers
private static STRRegister? _phaseTotal;
1 référence
public static STRRegister PhaseTotal => _phaseTotal ??= new(32069, 20, "STR", null, null, null);
```

L'objet « PhaseTotal » recense les adresses à partir du registre « PhaseAVoltage » jusqu'au registre « InsulationResistance ».
Tous ces registres recensés peuvent être appelés individuellement ou bien être regroupés en un seul registre

```
//41-45 InverterEquipment registers
private static Uint16Register? _lineVoltageBetweenPhasesCAndA;
private static Uint16Register? _phaseAVoltage;
private static Uint16Register? _phaseBVoltage;
private static Uint16Register? _phaseCVoltage;
private static Int32Register? _phaseACurrent;
0 références
public static Uint16Register LineVoltageBetweenPhasesCAndA => _lineVoltageBetweenPhasesCAndA ??= new Uint16Register(32068, 1, "U16", 10, "V", null);
1 référence
public static Uint16Register PhaseAVoltage => _phaseAVoltage ??= new Uint16Register(32069, 1, "U16", 10, "V", null);
1 référence
public static Uint16Register PhaseBVoltage => _phaseBVoltage ??= new Uint16Register(32070, 1, "U16", 10, "V", null);
1 référence
public static Uint16Register PhaseCVoltage => _phaseCVoltage ??= new Uint16Register(32071, 1, "U16", 10, "V", null);
1 référence
public static Int32Register PhaseACurrent => _phaseACurrent ??= new Int32Register(32072, 2, "I32", 1000, "A", null);

//46-50 InverterEquipment registers
private static Int32Register? _phaseBCurrent;
private static Int32Register? _phaseCCurrent;
private static Int32Register? _eakActivePowerOfCurrentDay;
private static Int32Register? _activePower;
private static Int32Register? _reactivePower;
1 référence
public static Int32Register PhaseBCurrent => _phaseBCurrent ??= new Int32Register(32074, 2, "I32", 1000, "A", null);
1 référence
public static Int32Register PhaseCCurrent => _phaseCCurrent ??= new Int32Register(32076, 2, "I32", 1000, "A", null);
1 référence
public static Int32Register PeakActivePowerOfCurrentDay => _eakActivePowerOfCurrentDay ??= new Int32Register(32078, 2, "I32", 1000, "kW", null);
2 références
public static Int32Register ActivePower => _activePower ??= new Int32Register(32080, 2, "I32", 1000, "kW", null);
1 référence
public static Int32Register ReactivePower => _reactivePower ??= new Int32Register(32082, 2, "I32", 1000, "kVar", null);

//51-55 InverterEquipment registers
private static Int16Register? _powerFactor32084;
```

pour ne faire qu'un seul appel.
« PhaseTotal » est un tableau d'éléments de types « ushort », constitués de plusieurs valeurs de registre. Lorsque l'on découpe PhaseTotal, on obtient ces registres :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	c	d	d	e	e	f	f	g	g	h	h	i	i	j	k	l	m	n

Les lettres correspondent aux valeurs des registres ci-dessous.

PhaseAVoltage : **a**	ActivePower : **h**
PhaseBVoltage : **b**	ReactivePower : **i**
PhaseBVoltage : **c**	PowerFactor32084 : **j**
PhaseACurrent : **d**	GridFrequency : **k**
PhaseBCurrent : **e**	Efficiency : **l**
PhaseCCurrent : **f**	InternalTemperature : **m**
PeakActivePowerOfCurrentDay : **g**	InsulationResistance : **n**

La valeur du registre « ActivePower » est située aux indices 11 et 12, elle est composée d'une quantité de 2 adresses.

```
#region
ActivePower = InverterEquipmentRegister.ActivePower.GetFormattedValue([result.PhaseTotal[11], result.PhaseTotal[12]]);
```

Ici, « ActivePower » se constitue donc d'un tableau représenté comme ceci : [PhaseTotal[11], PhaseTotal[12]]

4.3 Singleton

```
namespace Sun2000_modbus.ModbusCommunication
{
    public sealed class Master
    {
        private IModbusMaster _modbusMaster;
        private byte _slaveId;
        private TcpClient _tcpClient;
        private static Master? _instance;

        //singleton master
        private Master() { }
        //Set the instance of master
        public static Master GetInstance()
        {
            if (_instance == null)
            {
                _instance = new Master();
            }
            return _instance;
        }

        //Set the slave number
        public void SetSlave(byte slaveId)...
```

Utilisation du singleton pour l'instance master :

- Le Singleton est un design pattern qui garantit qu'une classe n'a qu'une seule instance et fournit un point d'accès global à cette instance.
- Il est implémenté en utilisant un constructeur privé, une instance statique et une méthode d'accès.
- Ce modèle est utile pour des objets partagés comme les logs, les connexions aux bases de données ou les gestionnaires de configurations.

```
public class MainViewModel : Notifier
{
    //Get the same instance of master by the singleton
    private Master master = Master.GetInstance();
    //Set timer for a loop
    private System.Timers.Timer _timer;
    //State of ExecuteCommand()
    private bool _isRunning = false;

    private ICommand _stopCommand;
    private ICommand _startCommand;
```

Récupération de l'instance master dans le ViewModel.

5 Register

```
using Sun2000_modbus.ModbusCommunication;
using static Sun2000_modbus.Mapping;

namespace Sun2000_modbus.Registers
{
    public abstract class Register
    {
        public ushort StartAddress { get; }
        public ushort Quantity { get; }
        public string Type { get; }
        public ushort? Gain { get; }
        public string? Unit { get; }
        public string? Mapping { get; }

        //get the master instance
        protected Master _master = Master.GetInstance();

        protected Register(ushort StartAddress, ushort Quantity, string Type, ushort? Gain, string? Unit, string? Mapping)
        {
            this.StartAddress = StartAddress;
            this.Quantity = Quantity;
            this.Type = Type;
            this.Gain = Gain;
            this.Unit = Unit;
            this.Mapping = Mapping;
        }
    }

    //Classe generic type Register
    public abstract class Register<T> : Register
    {
        protected Register(ushort StartAddress, ushort Quantity, string Type, ushort? Gain, string? Unit, string? Mapping) : base(StartAddress, Quantity, Type, Gain, Unit, Mapping)
        {
        }
```

La classe Register est une classe abstraite qui représente un registre de base. Elle contient des propriétés et un constructeur permettant de définir les informations nécessaires pour interagir avec un registre Modbus.

```
public abstract class Register
{
    public ushort StartAddress { get; }
    public ushort Quantity { get; }
    public string Type { get; }
    public ushort? Gain { get; }
    public string? Unit { get; }
    public string? Mapping { get; }
```

Propriétés :

- **StartAddress** (ushort) : Adresse de départ du registre.
- **Quantity** (ushort) : Quantité d'adresses associées à ce registre.
- **Type** (string) : Type de la valeur du registre (string, entier, …).
- **Gain** (ushort?) : Facteur d'échelle appliqué aux données du registre (nullable).
- **Unit** (string?) : Unité associée à la valeur du registre (nullable).
- **Mapping** (string?) : Mapping supplémentaire lié au registre (nullable).

5.1 Généricité <T>

```
//Classe generic type Register
public abstract class Register<T> : Register
{
    protected Register(
        ushort StartAddress,
        ushort Quantity,
        string Type,
        ushort? Gain,
        string? Unit,
        string? Mapping) :
        base(StartAddress, Quantity, Type, Gain, Unit, Mapping)
    {
    }
```

La classe Register<T> est une extension générique de la classe Register. Elle permet de spécialiser les registres avec un type de données spécifique, défini par le paramètre générique « T ». Le constructeur hérite des paramètres et du comportement du constructeur de la classe Register, tout en introduisant la possibilité d'ajouter un type spécifique au registre.
« T » est un paramètre générique utilisé pour représenter un type défini au moment de l'utilisation dans une classe, une méthode ou une interface.

```
    public override double GetRawValue()...
    //Overload param ushort
    public override double GetRawValue(ushort register)...
    //Overload param ushort[]
    public override double GetRawValue(ushort[] register)...
```

Cela permet d'écrire du code plus générique et d'éviter les répétitions.
Ici, « T » est défini à l'utilisation des méthodes et des surcharges en tant que type « double ». Ces méthodes abstraites retournent donc des valeurs d'décimaux.

```
    public override string GetRawValue()...
    //Overload param ushort
    public override string GetRawValue(ushort register)...
    //Overload param ushort[]
    public override string GetRawValue(ushort[] register)...
```

Ainsi, « T » étant défini en tant que « string », ces méthodes retournent dans ces cas-là des chaînes de caractère.

5.2 Recensement des registres de l'onduleur SUN2000

```
using Sun2000_modbus.Registers;
namespace Sun2000_modbus
{
    public static class InverterEquipmentRegister...

    public static class BatteryEquipmentRegister...

    public static class MeterEquipmentRegister...
}
```

3 catégories de registre :

- InverterEquipment : concerne les registres de l'équipement de l'onduleur.
- BatteryEquipment : concerne les registres de l'équipement des batteries.
- MeterEquipment : concerne les registres de l'équipement des appareils de mesures.

```
using Sun2000_modbus.Registers;
namespace Sun2000_modbus
{
    public static class InverterEquipmentRegister
    {
        //1-5 InverterEquipment registers
        private static STRRegister? _model;
        private static STRRegister? _sN;
        private static STRRegister? _pN;
        private static Uint16Register? _modelId;
        private static Uint16Register? _numberOfPvStrings;
        public static STRRegister Model => _model ??= new STRRegister(30000, 15, "STR", null, null, null);
        public static STRRegister SN => _sN ??= new STRRegister(30015, 10, "STR", null, null, null);
        public static STRRegister PN => _pN ??= new STRRegister(30025, 10, "STR", null, null, null);
        public static Uint16Register ModelId => _modelId ??= new Uint16Register(30070, 1, "U16", null, null, null);
        public static Uint16Register NumberOfPvStrings => _numberOfPvStrings ??= new Uint16Register(30071, 1, "U16", 1, null, null);

        //6-10 InverterEquipment registers
        private static Uint16Register? _numberOfMppTrackers;
        private static Uint32Register? _ratePower;
        private static Uint32Register? _maximumActivePower;
        private static Uint32Register? _maximumApparentPower;
        private static Int32Register? _maximumReactivePowerFedToTheGrid;
        public static Uint16Register NumberOfMppTrackers => _numberOfMppTrackers ??= new Uint16Register(30072, 1, "U16", 1, null, nul
        public static Uint32Register RatePower => _ratePower ??= new Uint32Register(30073, 2, "U32", 1000, "kW", null);
        public static Uint32Register MaximumActivePower => _maximumActivePower ??= new Uint32Register(30075, 2, "U32", 1000, "kW", nu
        public static Uint32Register MaximumApparentPower => _maximumApparentPower ??= new Uint32Register(30077, 2, "U32", 1000, "kVA
        public static Int32Register MaximumReactivePowerFedToTheGrid => _maximumReactivePowerFedToTheGrid ??= new Int32Register(30079
```

```
        private static STRRegister? _model;
        private static STRRegister? _sN;
```

« _model » est une variable statique nullable, qui stocke un objet unique de « STRRegister » pour toute la classe.

```
public static STRRegister Model => _model ??= new STRRegister(30000, 15, "STR", null, null, null);
public static STRRegister SN => _sN ??= new STRRegister(30015, 10, "STR", null, null, null);
```

« Model » est une propriété statique qui initialise « _model » uniquement si nécessaire. Cela permet de garantir qu'une seule instance est utilisée à travers toute l'application (singleton), ainsi qu'optimiser les performances en évitant d'instancier un objet tant qu'il n'est pas nécessaire.
L'objet « Model » est à l'adresse de départ **30000**, la quantité (correspondant à la taille de ce registre) est de **15** adresses, le type de la valeur de ce registre est **STR** (correspondant à string ou chaîne de caractères), ainsi ce registre ne possède aucun gain ou échelle à appliquer, aucune unité de mesure et ne possède aucun traitement de mapping.

Voir la documentation des interfaces Modbus de l'onduleur Huawei SUN2000.
=> https://support.huawei.com/enterprise/fr/doc/EDOC1100387581?section=k004

5.3 Récupération des valeurs des registres de l'onduleur SUN2000

```
//Get values contained in the inverter registers
public string GetRegisterValue()...
//Overload param ushort
public string GetRegisterValue(ushort register)...
//Overload param ushort[]
public string GetRegisterValue(ushort[] register)...


//Get raw values according to their data types
public abstract T GetRawValue();
//Overload param ushort
public abstract T GetRawValue(ushort register);
//Overload param ushort[]
public abstract T GetRawValue(ushort[] register);


//Get values according to their data types and their gain
public abstract T GetValue();
//Overload param ushort
public abstract T GetValue(ushort register);
//Overload param ushort[]
public abstract T GetValue(ushort[] register);


//Get the processed values with their unit
public string GetFormattedValue()...
//Overload param ushort
public string GetFormattedValue(ushort register)...
//Overload param ushort
public string GetFormattedValue(ushort[] register)...
```

Plusieurs méthodes permettant de récupérer les valeurs des registres de l'onduleur SUN2000 sous différentes formes.

5.3.1 GetRegisterValue()

Récupère la valeur contenue dans le registre sous forme de chaîne de caractères concaténées.

```
//Get values contained in the inverter registers
public string GetRegisterValue()
{
    string data = "[";
    ushort[] register = _master.ReadRegisters(StartAddress, Quantity);
    if (register.Length == 1)
    {
        foreach (var value in register)
        {
            data += value.ToString();
        }
    }
    else
    {

        foreach (var value in register)
        {
            data += $"{value.ToString()},";
        }
    }
    return data += "]";
}
```

- Le tableau « register » récupère les valeurs du registre de l'appel à la méthode ***ReadRegisters().***
- Contrôle si le registre a une longueur de 1 élément.
- Concatène la valeur dans la chaine « data ».

```
GetRegisterValue() de ActivePower :
[0,3029,]
```

```
//Overload param ushort[]
public string GetRegisterValue(ushort[] register)
{
    string data = "[";
    if (register.Length == 1)
    {
```

La surcharge de la méthode prend en paramètre un tableau ushort mais n'appelle pas la méthode ***ReadRegisters().***

5.3.2 GetRawValue()

Récupère la valeur brute de la donnée selon son typage (entier, double, chaîne, …).

```
//Get raw values according to their data types
public abstract T GetRawValue();
//Overload param ushort
public abstract T GetRawValue(ushort register);
//Overload param ushort[]
public abstract T GetRawValue(ushort[] register);
```

Méthodes abstraites implémentées par les classes filles et ayant pour paramètre générique « T ».

```
public override string GetRawValue()
{
    return Datatypes.DecodeString(_master.ReadRegisters(StartAddress, Quantity));
}
//Overload param ushort
public override string GetRawValue(ushort register)...
//Overload param ushort[]
public override string GetRawValue(ushort[] register)
{
    return Datatypes.DecodeString(register);
}
```

- Retourne la valeur de l'appel de la méthode ***DecodeString()*** prenant en paramètre l'appel de la méthode ***ReadRegisters()***.
- Retourne la valeur de l'appel de la méthode ***DecodeString()*** prenant en paramètre un tableau d'éléments d'un registre.

```
GetRawValue() de ActivePower :
3406
```

5.3.3 GetValue()

Récupère la valeur du registre selon son typage en appliquant son échelle (Gain : /1, /10, /100, /1000).

```
//Get values according to their data types and their gain
public abstract T GetValue();
//Overload param ushort
public abstract T GetValue(ushort register);
//Overload param ushort[]
public abstract T GetValue(ushort[] register);
```

Méthodes abstraites implémentées par les classes filles et ayant pour paramètre générique « T ».

```
public override double GetValue()
{
    if (Gain != null)
    {
        return (double)(GetRawValue() / Gain);
    }
    else
    {
        return GetRawValue();
    }
}
//Overload param ushort
public override double GetValue(ushort register)
{
    if (Gain != null)
    {
        return (double)(GetRawValue(register) / Gain);
    }
    else
    {
        return GetRawValue(register);
    }
}
//Overload param ushort[]
public override double GetValue(ushort[] register)...
```

- Vérifie si le gain du registre n'est pas nul.
- Si oui, retourne la valeur de l'appel de la méthode ***GetRawValue()*** en appliquant le gain du registre.
- Sinon, retourne seulement la valeur de l'appel de la méthode ***GetRawValue()***.

```
GetValue() de ActivePower :
3,209
```

5.3.4 GetFormattedValue()

Récupérer la valeur traitée et concaténée à son unité de mesure (kW, kWh, V, A, %, …).

```
        //Get the processed values with their unit
        public string GetFormattedValue()
        {
            if (Mapping == null)
            {
                return $"{GetValue()} {Unit}";
            }
            else
            {
                return GetMapping(Mapping, Convert.ToInt32(GetValue()));
            }
        }
        //Overload param ushort
        public string GetFormattedValue(ushort register)
        {
            if (Mapping == null)
            {
                return $"{GetValue(register)} {Unit}";
            }
            else
            {
                return GetMapping(Mapping, Convert.ToInt32(GetValue(register)));
            }
        }
        //Overload param ushort
        public string GetFormattedValue(ushort[] register)...
    }
```

- Vérifie si le mapping du registre est nul.
- Si oui, retourne la valeur de l'appel à la méthode ***GetValue()*** concaténée à l'unité du registre.
- Sinon, retourne la valeur de l'appel à la méthode ***GetMapping()*** prenant comme paramètre :
    - Le mapping du registre
      Voir Mapping => Mapping
    - La valeur de l'appel à la méthode ***GetValue()***

```
GetFormattedValue() de ActivePower :
3,509 kW
```

Puissance de sortie PV

3,576 kW

5.4 Surcharge des méthodes

```
        //Get the processed values with their unit
        public string GetFormattedValue()...
        //Overload param ushort
        public string GetFormattedValue(ushort register)...
        //Overload param ushort
        public string GetFormattedValue(ushort[] register)...
    }
```

Ces méthodes possèdent les mêmes noms, mais des paramètres différents permettant de traiter différents types de données d'entrée selon le besoin d'utilisation.

```
return new
{
    //Retrieve values
    ActivePowerPercentageDerating = InverterEquipmentRegister.ActivePowerPercentageDerating.GetFormattedValue(),
    PhaseTotal = master.ReadRegisters(InverterEquipmentRegister.PhaseTotal); //Read PhaseAVoltage to InsulationResistance registers
```

L'appel de la méthode ***GetFormattedValue()*** sur l'objet « ActivePowerPercentageDerating » permet de récupérer seulement la valeur formatée de celui-ci.

```
InsulationResistance = InverterEquipmentRegister.InsulationResistance.GetFormattedValue(result.PhaseTotal[19]);
InternalTemperature = InverterEquipmentRegister.InternalTemperature.GetFormattedValue(result.PhaseTotal[18]);
```

L'appel de la méthode ***GetFormattedValue(result.PhaseTotal[19])*** sur l'objet « InsulationResistance » prend en paramètre l'élément à l'indice « 19 » du tableau « PhaseTotal » récupéré.

```
});
//Set values to display on the view
#region
ActivePower = InverterEquipmentRegister.ActivePower.GetFormattedValue([result.PhaseTotal[11], result.PhaseTotal[12]]);
ActivePowerMeterEquipment = Math.Abs(MeterEquipmentRegister.ActivePower.GetValue([result.MeterRegisters[12], result.Met
EnergyConsumption = Math.Round(InverterEquipmentRegister.ActivePower.GetValue([result.PhaseTotal[11], result.PhaseTotal[
DailyEnergyYield = InverterEquipmentRegister.DailyEnergyYield.GetFormattedValue([result.EnergyYield[8], result.EnergyYie
AccumulatedEnergyYield = InverterEquipmentRegister.AccumulatedEnergyYield.GetFormattedValue([result.EnergyYield[0], resu
```

L'appel de la méthode ***GetFormattedValue([result.PhaseTotal[11], result.PhaseTotal[12]])*** sur l'objet « ActivePower » prend en paramètre un tableau contenant les éléments « 11 » et « 12 » du tableau « PhaseTotal » récupéré.

Voir Groupe de registres => Groupe de registres

6 Décodage des types

```
namespace Sun2000_modbus
{

    public class Datatypes
    {
        //Decode the registers values to return a data string
        public static string DecodeString(ushort[] registers)...

        //Decode register values to return 16-bit unsigned integer data
        public static uint DecodeUint16(ushort[] registers)...
        //Overload param ushort
        public static uint DecodeUint16(ushort register)...

        //Decode the registers values to return 16-bit signed integer data
        public static int DecodeInt16(ushort[] registers)...
        //Overload param ushort
        public static int DecodeInt16(ushort register)...

        //Decode the registers values to return 32-bit unsigned integer data
        public static uint DecodeUint32(ushort[] registers)...
        //Decode the registers values to return 32-bit signed integer data
        public static int DecodeInt32(ushort[] registers)...

        //Decode the registers values to return a string of binary data
        public static string DecodeBitfield(ushort[] registers)...
        //Decode the registers values to return a string of multible bytes
        public static string DecodeMLD(ushort[] registers)...
    }
}
```

La classe Datatypes contient des méthodes statiques permettant de décoder les données brutes des registres Modbus en différents types ou formats utilisables. Ces méthodes facilitent l'interprétation des données extraites des registres de l'onduleur SUN2000.

6.1 DecodeString()

Convertir un tableau ushort en chaîne de caractères pour le type STR.

```
//Decode the registers values to return a data string
public static string DecodeString(ushort[] registers)
{
    string data = "";
    foreach (var register in registers)
    {
        //convert to binary 16 bits
        string binaire16Bits = Convert.ToString(register, 2).PadLeft(16, '0');
        //Struct the 16 bits in 2 octets and
        //convert octets to ASCII
        char asciiChar1 = (char)Convert.ToInt32(binaire16Bits.Substring(0, 8), 2);
        char asciiChar2 = (char)Convert.ToInt32(binaire16Bits.Substring(8, 8), 2);
        //concat ASCII characters in string data
        data = data + asciiChar1 + asciiChar2;
    }
    return data;
}
```

- Parcours chaque élément du tableau « registers ».
- Convertir l'élément en une représentation binaire sur 16 bits.
  PadLeft(16, '0') garantit que la chaîne a toujours 16 bits en ajoutant des zéros en tête si nécessaire.
- Scinde cette chaîne de 16 caractères en 2 chaînes d'octets.
  Convertir les 2 chaînes d'octets en caractères ASCII.
- Concatène chaque caractère ASCII pour former la valeur du registre.

6.2 DecodeUint16()

Convertir un tableau ushort en entier non signé pour le type U16, surcharge de la méthode pour prendre seulement en paramètre un entier ushort.

```
//Decode register values to return 16-bit unsigned integer data
public static uint DecodeUint16(ushort[] registers)
{
    string data = "";
    foreach (var register in registers)
    {
        //concat registers values in string data
        data += register;
    }
    return uint.Parse(data);
}
//Overload param ushort
public static uint DecodeUint16(ushort register)...
```

- Parcours chaque élément du tableau « registers ».
- Concatène l'élément à la chaîne « data » pour le convertir en chaîne.
- Retourne cette chaîne convertie en entier non signé.

6.3 DecodeInt16()

Convertir un tableau ushort en entier signé pour le type I16, surcharge de la méthode pour prendre seulement en paramètre un entier ushort.

```
//Decode the registers values to return 16-bit signed integer data
public static int DecodeInt16(ushort[] registers)
{
    string data = "";
    foreach (var register in registers)
    {
        //concat registers values in string data
        data += register;
    }
    return int.Parse(data);
}
//Overload param ushort
public static int DecodeInt16(ushort register)...
```

- Parcours chaque élément du tableau « registers ».
- Concatène l'élément à la chaîne « data » pour le convertir en chaîne.
- Retourne cette chaîne convertie en entier signé.

6.4 DecodeUint32()

Convertir un tableau ushort en entier signé pour le type U32.

```
//Decode the registers values to return 32-bit unsigned integer data
public static uint DecodeUint32(ushort[] registers)
{
    string binaire32Bits = "";
    foreach (var register in registers)
    {
        //convert to binary 16 bits
        string binaire16Bits = Convert.ToString(register, 2).PadLeft(16, '0');
        //concatenate 2 binary 16 bits to get 32 bits
        binaire32Bits += binaire16Bits;
    }
    //convert 32 bits binary to int32
    //double dataInt32 = Convert.ToInt32(binaire32Bits, 2);
    return Convert.ToUInt32(binaire32Bits, 2);
}
//Decode the registers values to return 32-bit signed integer data
public static int DecodeInt32(ushort[] registers)...
```

- Parcours chaque élément du tableau « registers ».
- Convertir l'élément en une représentation binaire sur 16 bits.
- Le concatène à la valeur précédente de la chaîne « binaire32Bits » pour le convertir en chaîne.
- Retourne la chaîne « binaire32Bits » convertie en entier non signé 32 bits.

6.5 DecodeInt32()

Convertir un tableau ushort en entier signé pour le type I32.

```
//Decode the registers values to return 32-bit signed integer data
public static int DecodeInt32(ushort[] registers)
{
    int i = 0;
    double data = 0;
    foreach (var register in registers)
    {
        //traitement for negative values
        //if the first register data value is not empty
        //we substract the first register data value to the second register data value
        data = (i == 0) ? (data - register) : (data + register);
        i++;
    }
    return Convert.ToInt32(data);
}
```

- Initialise une variable locale « i ».
- Parcours les éléments du tableau « registers ».
- Opération ternaire : ***(condition ? (Faire si vrai) : (Faire si faux)).***
  Si la condition « i » est égale à 0, alors soustrait la valeur de l'élément du tableau à data.
  Sinon, la condition « i » n'est pas égale à 0, additionne la valeur de l'élément du tableau à data.
- Incrémente la variable « i » de 1.
- Retourne le résultat converti en entier signé 32-bit.

6.6 DecodeBitfield()

Convertir un tableau ushort en chaîne de représentation binaire pour le type Bitfield.

```
//Decode the registers values to return a string of binary data
public static string DecodeBitfield(ushort[] registers)
{
    string data = "";
    foreach (var register in registers)
    {
        //convert to binary 16 bits
        string binaire16Bits = Convert.ToString(register, 2).PadLeft(16, '0');
        //concat registers values in string data
        data = data + binaire16Bits;
    }
    return data;
}
```

- Parcours les éléments du tableau « registers ».
- Convertir l'élément en une représentation binaire sur 16 bits.
- Concatène les valeurs des éléments pour former une chaîne binaire.

6.7 DecodeMLD()

Convertir un tableau ushort en chaine pour le type MLD.

```
//Decode the registers values to return a string of multible bytes
public static string DecodeMLD(ushort[] registers)
{
    string data = "";
    foreach (var register in registers)
    {
        //concat registers values in string data
        data += register;
    }
    return data;
}
```

- Parcours les éléments du tableau « registers ».
- Concatène les valeurs des éléments pour former une chaîne.

7 Mapping

```
namespace Sun2000_modbus
{
    class Mapping
    {
        public static string GetMapping(string mapping, int register)...
        #region
        //InverterEquipment register
        public static string DeviceStatus(int register)...
        public static string LicenseStatus(int register)...
        public static string ModuleStatus4G(int register)...
        public static string PinVerificationStatus4G(int register)...
        public static string ChargeDischargeMode(int register) //erreur exception non gérer levé...
        public static string QUCharacteristicCurveMode(int register)...
        public static string TLSEncryption(int register)...
        public static string WLANWakeup(int register)...
        public static string FastPowerScheduling(int register)...
        public static string RemoteChargeDischargeControlMode(int register)...
        public static string ScheduledTask(int register)...
        public static string PeakShaving(int register)...
        public static string AIOpticalStorage(int register)...
        public static string BackupBoxModel(int register)...
        public static string PhaseToGroundShortCircuitProtection(int register)...
        //BatteryEquipment register
        public static string RunningStatus(int register)...
        public static string WorkingMode(int register)...
        public static string ProductMode(int register)...
        public static string WorkingModeSettings(int register)...
        public static string ChargeFromGridFunction(int register)...
        public static string ForcibleChargeDischargeSettingMode(int register)...
        public static string ExcessPVEnergyUseInTOU(int register)...
        public static string ActivePowerControlMode(int register)...
        public static string SwitchToOffGrid(int register)...
        public static string VoltageInIndependentOperation(int register)...
        //MeterEquipment register
        public static string MeterStatus(int register)...
        public static string MeterType(int register)...
        public static string MeterModelDetectionResult(int register)...
        #endregion
    }
}
```

Le mapping de registre consiste à associer une valeur d'un registre à une signification compréhensible par l'utilisateur ou le système.

7.1 GetMapping()

```
else
{
    return GetMapping(Mapping, Convert.ToInt32(GetValue(register)));
```

La méthode ***GetMapping()*** prend en paramètre une chaîne qui correspond à l'attribut « Mapping » d'un objet de type Register et à sa valeur du registre.

```
public static string GetMapping(string mapping, int register)
{
    switch (mapping)
    {
        //InverterEquipment register
        case "DeviceStatus":
            return DeviceStatus(register);
        case "LicenseStatus":
            return LicenseStatus(register);
        case "ModuleStatus4G":
            return ModuleStatus4G(register);
```

GetMapping() traite le mapping de l'objet à travers un « switch case », cela permet d'appeler la sous-méthode adéquate en donnant en paramètre la valeur du registre.

Si un registre contient une valeur qui peut être 1, 2, 3 ou autre, on effectue un mapping comme suit :

```
#region
//InverterEquipment register
public static string DeviceStatus(int register)
{
    switch (register)
    {
        case 0:
            return "Standby: initializing";
        case 1:
            return "Standby: detecting insulation resistance";
        case 2:
            return "Standby: detecting irradiation";
        case 3:
            return "Standby: Grid detecting";
        case 256:
            return "Starting";
        case 512:
            return "On-grid";
        case 513:
```

8 MainViewModel

8.1 Classe MainViewModel

```
using System.Windows.Input;
using SolarPanel.Utilities;
using Sun2000_modbus;
using Sun2000_modbus.ModbusCommunication;
using System.Windows;

namespace SolarPanel.Main
{
    public class MainViewModel : Notifier
    {
        //Get the same instance of master by the singleton
        private Master master = Master.GetInstance();
        //Set timer for a loop
        private System.Timers.Timer _timer;
        //State of ExecuteCommand()
        private bool _isRunning = false;

        private ICommand _stopCommand;
        private ICommand _startCommand;

        //Button Start and stop commande
        public ICommand StartCommand => _startCommand ??= new DelegateCommand(async () => await StartExecutionAsync(), CanExecuteStartCommand);
        public ICommand StopCommand => _stopCommand ??= new DelegateCommand(StopExecution, CanExecuteCommand);

        public MainViewModel()...

        //Connect the master to inverter
        private void InitialiserMaster()...

        //Start async execution on a loop
        private async Task StartExecutionAsync()...

        //Close the programme
        private void StopExecution()...

        //execute the program to recover the inverter values
        private async Task ExecuteLoopAsync()...

        //Set visible TextBlock according to Active Power value;
        public bool IsTextVisible => MeterEquipmentRegister.ActivePower.GetValue() < 0;
        public bool IsNotTextVisible => MeterEquipmentRegister.ActivePower.GetValue() >= 0;

        //Execute the task who get inverter register values
        private async Task ExecuteCommand()...

        private void SetDateTime()...
        //Set seed values at the beginning
        public void SetStartValue()...

        //return if the start button can be execute
        private bool CanExecuteStartCommand()...
        private bool CanExecuteCommand()...

        //Inverter interface instance
        //Display values
        #region
    }
```

En WPF (Windows Presentation Foundation), le ViewModel fait partie du modèle MVVM (Model-View-ViewModel), qui est une architecture utilisée pour séparer la logique de l'interface utilisateur de la logique métier et des données.
La ViewModel agit comme un intermédiaire entre la View (Vue) et le Model (Modèle) :

- Elle contient la logique de présentation et transforme les données du modèle en un format que la vue peut afficher.
- Elle implémente l'interface INotifyPropertyChanged pour notifier la vue lorsqu'une donnée change.
- Elle gère les commandes (ICommand) pour permettre l'interaction entre la vue et la logique métier.

```
public class MainViewModel : Notifier
{
    //Get the same instance of master by the singleton
    private Master master = Master.GetInstance();
    //Set timer for a loop
```

Récupération de l'instance master.
Voir singleton => Singleton

8.1.1 ICommand

```
private ICommand _stopCommand;
private ICommand _startCommand;

//Button Start and stop commande
public ICommand StartCommand => _startCommand ??= new DelegateCommand(async () => await StartExecutionAsync(), CanExecuteStartCommand);
public ICommand StopCommand => _stopCommand ??= new DelegateCommand(StopExecution, CanExecuteCommand);
```

Instanciation des commandes de démarrage et d'arrêt utilisées pour exécuter des actions à partir de la vue. Exécute la méthode asynchrone ***StartExecutionAsync()*** selon la condition ***CanExecuteStartCommand().*** Exécute la méthode ***StopCommand()*** selon la condition ***CanExecuteCommand()***.

```
private bool CanExecuteStartCommand()
{
    return _canStart; // Retourne false disable the start button
}
private bool CanExecuteCommand()
{
    return true;
}
```

8.1.2 MainViewModel

```
public MainViewModel()
{
    InitialiserMaster(); //Set the connection
    _timer = new System.Timers.Timer(200); //Set timer of 0,2 seconds
    _timer.Elapsed += (sender, e) => SetDateTime(); //Execute SetDatetime every 0,2 seconds
    SetDateTime();
    _timer.Start();
}
```

- Appel de la méthode ***InitialiserMaster()***.
- Initialiser un chronomètre « _timer » à 0,2 seconde.
- Déclenche l'exécution de la méthode ***SetDateTime()*** selon le chronomètre.
- Appel de la méthode ***SetDateTime()***.
- Lance le chronomètre.

```
//Connect the master to inverter
private void InitialiserMaster()
{
    master.Connect(); //Connect the communication
    master.SetSlave(1); //Set slave number 1
}
```

Initialise la connexion vers l'hôte distant en établissant la liaison et en attribuant le numéro de l'esclave pour interroger les registres.
Voir Master => Master

8.1.3 SetDateTime()

```
private void SetDateTime()
{
    DateTime LocalDate = DateTime.Now;
    LocalDateTime = LocalDate.ToString();
}
```

Affecte la valeur de la date et l'heure locale à la propriété LocalDateTime permettant de l'afficher sur l'interface graphique.

8.1.4 StartExecutionAsync()

```
//Start async execution on a loop
private async Task StartExecutionAsync()
{
    CanStart = false; //Disable the start button
    SetStartValue(); //Set all values to "-"
    await ExecuteCommand(); //First data recovery
    await ExecuteLoopAsync(); //Start a continuous loop
}
```

- Définit l'instance « canStart » à « false » empêchant d'exécuter plusieurs fois la commande ***StartExecutionAsync().***
- Appel de la méthode ***SetStartValue()***.
- Appel de la méthode ***ExecuteCommand()***.
- Appel de la méthode ***ExecuteLoopAsync()***.

8.1.5 SetStartValue()

```
//Set seed values at the beginning
public void SetStartValue()
{
    ModelValue = "-";
    DailyEnergyYield = "-";
    AccumulatedEnergyYield = "-";
    ActivePower = "-";
    OutputPower = "-";
    ReactivePower = "-";
    RatePower = "-";
    Metertype = "-";
```

Affecte toutes les valeurs à afficher de base à « - » en attendant de récupérer une première fois toutes les valeurs des registres.

8.1.6 ExecuteLoopAsync()

```
//execute the program to recover the inverter values
private async Task ExecuteLoopAsync()
{
    _isRunning = true;
    //ExecuteCommand while stop button is not press
    while (_isRunning)
    {
        await ExecuteCommand(); //Retrieve and display values
        //await Task.Delay(5000); //Set a wait delay
    }
}
```

La méthode ***ExecuteLoopAsync()*** exécute en boucle la méthode ***ExecuteCommand()***.

8.1.7 ExecuteCommand()

```
//Execute the task who get inverter register values
private async Task ExecuteCommand()
{
    var result = await Task.Run(() =>
    {
        return new
        {
            //Retrieve values
            ActivePowerPercentageDerating = InverterEquipmentRegister.ActivePowerPercentageDerating.GetFormattedValue(),
            PhaseTotal = master.ReadRegisters(InverterEquipmentRegister.PhaseTotal), //Read PhaseAVoltage to Insulationresistance registers
            EnergyYield = master.ReadRegisters(InverterEquipmentRegister.EnergyYeld), //Read AccumulatedEnergyYield to DailyEnergyYield registers
            PVAll = master.ReadRegisters(InverterEquipmentRegister.PVAll), //Read PV1Voltage to PPV6Current registers
            RatePower = InverterEquipmentRegister.RatePower.GetFormattedValue(),
            ModelValue = InverterEquipmentRegister.Model.GetFormattedValue(),
            MetterRegisters = master.ReadRegisters(MeterEquipmentRegister.MetterRegisters), //Read GridAPhaseVoltage to MeterType registers
        };
    });
```

La méthode asynchrone ***ExecuteCommand()*** exécute une tâche en arrière-plan via ***Task.Run()***, afin d'éviter de bloquer le thread principal de l'UI lors de la récupération des valeurs en arrière-plan. « await » est utilisé pour attendre la fin de l'exécution avant d'en relancer une.
Les valeurs sont récupérées depuis l'onduleur et un compteur d'énergie :

- via ***GetFormattedValue()***
  Voir méthode => GetFormattedValue()
- via master.***ReadRegisters()***
  Voir méthode => ReadRegisters()

Toutes les valeurs sont stockées dans l'objet « result ». Par la suite, l'objet est exploité pour ressortir et traiter les valeurs des registres.

8.1.8 Affectation des valeurs

```
//Set values to display on the view
#region
ActivePower = InverterEquipmentRegister.ActivePower.GetFormattedValue([result.PhaseTotal[11], result.PhaseTotal[12]]);
ActivePowerMeterEquipment = Math.Abs(MeterEquipmentRegister.ActivePower.GetValue([result.MetterRegisters[12], result.MetterRegisters[13]])) + " kW";
EnergyConsumption = Math.Round(InverterEquipmentRegister.ActivePower.GetValue([result.PhaseTotal[11], result.PhaseTotal[12]]) - MeterEquipmentRegister.Activ
DailyEnergyYield = InverterEquipmentRegister.DailyEnergyYield.GetFormattedValue([result.EnergyYield[8], result.EnergyYield[9]]);
AccumulatedEnergyYield = InverterEquipmentRegister.AccumulatedEnergyYield.GetFormattedValue([result.EnergyYield[0], result.EnergyYield[1]]);

ReactivePower = InverterEquipmentRegister.ReactivePower.GetFormattedValue([result.PhaseTotal[13], result.PhaseTotal[14]]);
OutputPower = InverterEquipmentRegister.PeakActivePowerOfCurrentDay.GetFormattedValue([result.PhaseTotal[9], result.PhaseTotal[10]]);
InsulationResistance = InverterEquipmentRegister.InsulationResistance.GetFormattedValue(result.PhaseTotal[19]);
InternalTemperature = InverterEquipmentRegister.InternalTemperature.GetFormattedValue(result.PhaseTotal[18]);
PowerFactor = InverterEquipmentRegister.PowerFactor32084.GetFormattedValue(result.PhaseTotal[15]);
ActivePowerPercentageDerating = result.ActivePowerPercentageDerating;

RatePower = result.RatePower;
PhaseAVoltage = InverterEquipmentRegister.PhaseAVoltage.GetFormattedValue(result.PhaseTotal[0]);
PhaseBVoltage = InverterEquipmentRegister.PhaseBVoltage.GetFormattedValue(result.PhaseTotal[1]);
PhaseCVoltage = InverterEquipmentRegister.PhaseCVoltage.GetFormattedValue(result.PhaseTotal[2]);
PhaseACurrent = InverterEquipmentRegister.PhaseACurrent.GetFormattedValue([result.PhaseTotal[3], result.PhaseTotal[4]]);
PhaseBCurrent = InverterEquipmentRegister.PhaseBCurrent.GetFormattedValue([result.PhaseTotal[5], result.PhaseTotal[6]]);
PhaseCCurrent = InverterEquipmentRegister.PhaseCCurrent.GetFormattedValue([result.PhaseTotal[7], result.PhaseTotal[8]]);

PV1Voltage = InverterEquipmentRegister.PV1Voltage.GetFormattedValue(result.PVAll[0]);
PV2Voltage = InverterEquipmentRegister.PV2Voltage.GetFormattedValue(result.PVAll[2]);
PV3Voltage = InverterEquipmentRegister.PV3Voltage.GetFormattedValue(result.PVAll[4]);
PV4Voltage = InverterEquipmentRegister.PV4Voltage.GetFormattedValue(result.PVAll[6]);
PV5Voltage = InverterEquipmentRegister.PV5Voltage.GetFormattedValue(result.PVAll[8]);
PV6Voltage = InverterEquipmentRegister.PV6Voltage.GetFormattedValue(result.PVAll[10]);

PV1Current = InverterEquipmentRegister.PV1Current.GetFormattedValue(result.PVAll[1]);
PV2Current = InverterEquipmentRegister.PV2Current.GetFormattedValue(result.PVAll[3]);
PV3Current = InverterEquipmentRegister.PV3Current.GetFormattedValue(result.PVAll[5]);
PV4Current = InverterEquipmentRegister.PV4Current.GetFormattedValue(result.PVAll[7]);
PV5Current = InverterEquipmentRegister.PV5Current.GetFormattedValue(result.PVAll[9]);
PV6Current = InverterEquipmentRegister.PV6Current.GetFormattedValue(result.PVAll[11]);

ModelValue = result.ModelValue;
Metertype = MeterEquipmentRegister.MeterType.GetFormattedValue(result.MetterRegisters[24]);
ReactivePowerMeterEquipment = MeterEquipmentRegister.ReactivePower.GetFormattedValue([result.MetterRegisters[14], result.MetterRegisters[15]]);
PowerFactorMeterEquipment = MeterEquipmentRegister.PowerFactor.GetFormattedValue(result.MetterRegisters[16]);
PositiveActiveElectricity = MeterEquipmentRegister.PositiveActiveElectricity.GetFormattedValue([result.MetterRegisters[18], result.MetterRegisters[19]]);
ReversActivePower = MeterEquipmentRegister.ReversActivePower.GetFormattedValue([result.MetterRegisters[20], result.MetterRegisters[21]]);
AccumulatedReactivePower = MeterEquipmentRegister.AccumulatedReactivePower.GetFormattedValue([result.MetterRegisters[22], result.MetterRegisters[23]]);

GridFrequency = InverterEquipmentRegister.GridFrequency.GetFormattedValue(result.PhaseTotal[16]);
GridAPhaseVoltage = MeterEquipmentRegister.GridAPhaseVoltage.GetFormattedValue([result.MetterRegisters[0], result.MetterRegisters[1]]);
GridBPhaseVoltage = MeterEquipmentRegister.GridBPhaseVoltage.GetFormattedValue([result.MetterRegisters[2], result.MetterRegisters[3]]);
GridCPhaseVoltage = MeterEquipmentRegister.GridCPhaseVoltage.GetFormattedValue([result.MetterRegisters[4], result.MetterRegisters[5]]);
GridAPhaseCurrent = MeterEquipmentRegister.GridAPhaseCurrent.GetFormattedValue([result.MetterRegisters[6], result.MetterRegisters[7]]);
GridBPhaseCurrent = MeterEquipmentRegister.GridBPhaseCurrent.GetFormattedValue([result.MetterRegisters[8], result.MetterRegisters[9]]);
GridCPhaseCurrent = MeterEquipmentRegister.GridCPhaseCurrent.GetFormattedValue([result.MetterRegisters[10], result.MetterRegisters[11]]);
#endregion
}
```

```
RatePower = result.RatePower;
PhaseAVoltage = InverterEquipmentRegister.PhaseAVoltage.GetFormattedValue(result.PhaseTotal[0]);
PhaseBVoltage = InverterEquipmentRegister.PhaseBVoltage.GetFormattedValue(result.PhaseTotal[1]);
PhaseCVoltage = InverterEquipmentRegister.PhaseCVoltage.GetFormattedValue(result.PhaseTotal[2]);
PhaseACurrent = InverterEquipmentRegister.PhaseACurrent.GetFormattedValue([result.PhaseTotal[3], result.PhaseTotal[4]]);
PhaseBCurrent = InverterEquipmentRegister.PhaseBCurrent.GetFormattedValue([result.PhaseTotal[5], result.PhaseTotal[6]]);
PhaseCCurrent = InverterEquipmentRegister.PhaseCCurrent.GetFormattedValue([result.PhaseTotal[7], result.PhaseTotal[8]]);
```

Assigne une valeur formatée à la variable d'instance PhaseAVoltage en utilisant la méthode ***GetFormattedValue()*** sur l'objet « result.PhaseAVoltage ». ***GetFormattedValue()*** prend en paramètre l'élément 0 de l'objet « result.PhaseTotal », qui correspond à la valeur de la PhaseAVoltage du groupe de registres PhaseTotal.

Voir Groupe de registres =>Groupe de registres

8.1.9 Propriété des registres

```
//inverter phase voltage instance

private string _phaseAVoltage;
public string PhaseAVoltage
{
    get => _phaseAVoltage;
    set
    {
        if (_phaseAVoltage != value)
        {
            _phaseAVoltage = value;
            OnPropertyChanged();
        }
    }
}
```

Stocke la valeur de la propriété PhaseAVoltage. La propriété publique vérifie si la nouvelle valeur est différente lorsqu'elle est modifiée, met à jour le champ et déclenche ***OnPropertyChanged()*** pour notifier l'interface graphique du changement.
Cela permet de mettre à jour automatiquement l'affichage lorsqu'une nouvelle valeur est assignée.

8.1.10 StopExecution()

```
//Stop and close the programme
private void StopExecution()
{
    if (MessageBox.Show("Voulez-vous vraiment quitter ?", "Confirmation", MessageBoxButton.YesNo) == MessageBoxResult.Yes)
    {
        _isRunning = false; //Stop the loop execution
        _timer.Stop(); //Stop the DateTime timer
        master.Disconnect(); //Disconnect the communication
        Application.Current.Shutdown(); //Close application
    }
}
```

- La méthode affiche une fenêtre de dialogue indiquant la confirmation de quitter l'application.
- Arrêt de la boucle exécutant indéfiniment la méthode ***ExecuteCommand()***.
- Arrêt du chronomètre exécutant ***SetDateTime()***.
- Clôture la connexion du master à l'hôte distant.
- Fermeture de l'application.

8.2 Notifier

La classe MainViewModel hérite de la classe Notifier pour pouvoir appeler la méthode ***OnPropertyChanged()***.

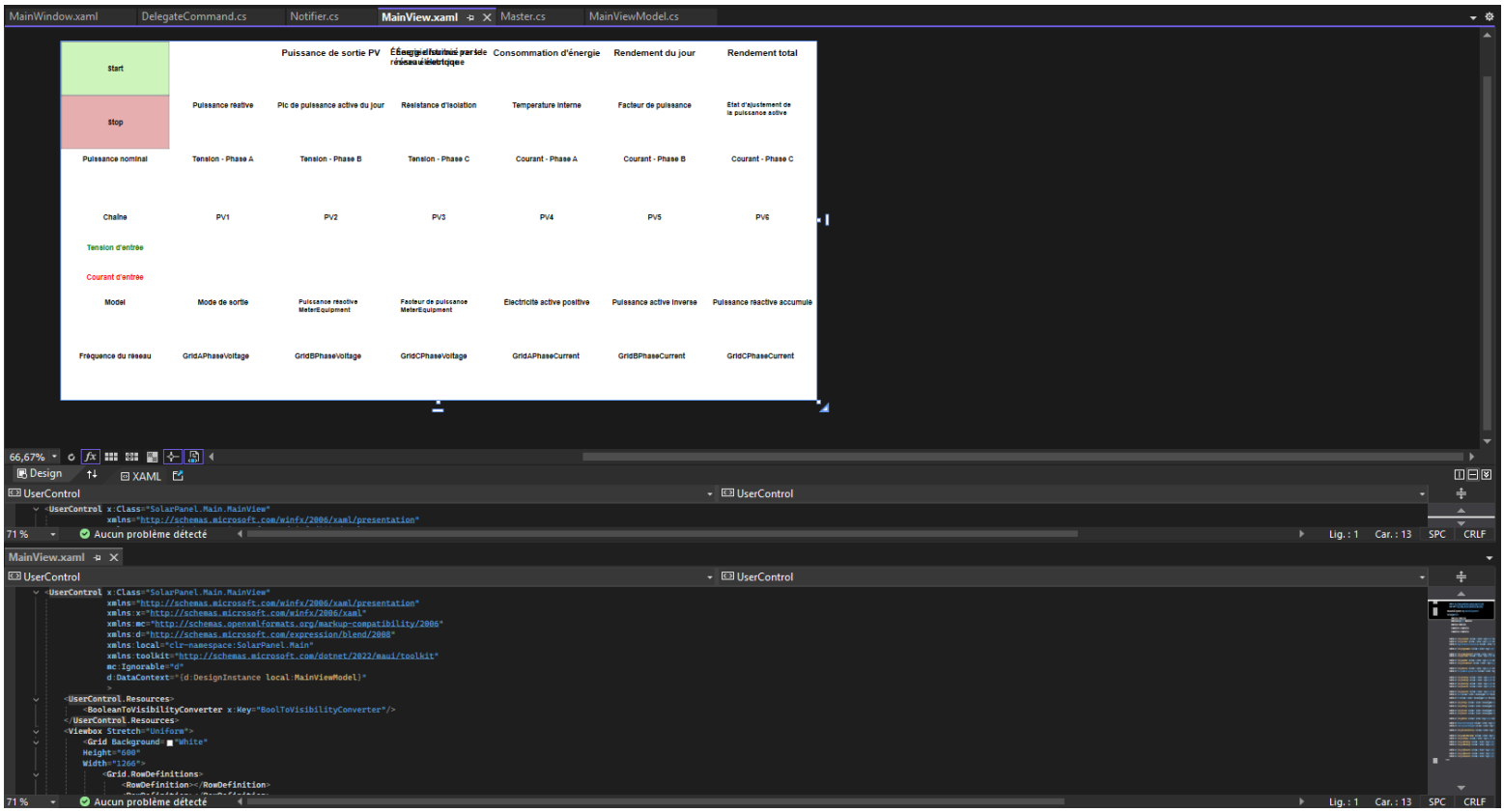
```
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace SolarPanel.Utilities
{
    public abstract class Notifier : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

8.3 DelegateCommand

La classe DelegateCommand permet de créer et d'instancier les commandes Start et Stop des boutons.

```
using System.Windows.Input;

namespace SolarPanel.Utilities
{
    public class DelegateCommand : ICommand
    {
        private readonly Action _execute;
        private readonly Func<bool> _canExecute;

        public DelegateCommand(Action execute, Func<bool> canExecute = null)
        {
            _execute = execute ?? throw new ArgumentNullException(nameof(execute));
            _canExecute = canExecute;
        }

        public bool CanExecute(object parameter)
        {
            return _canExecute == null || _canExecute();
        }

        public void Execute(object parameter)
        {
            _execute();
        }

        public event EventHandler CanExecuteChanged
        {
            add { CommandManager.RequerySuggested += value; }
            remove { CommandManager.RequerySuggested -= value; }
        }
    }
}
```

9 MainView



XAML (Extensible Application Markup Language) est un langage de balisage utilisé principalement pour définir l'interface utilisateur dans les applications WPF, UWP et Xamarin.Forms. XAML a une syntaxe qui s'appuie sur XML. Il permet de décrire la structure, la disposition et l'apparence des éléments visuels d'une application de manière déclarative, en séparant la logique métier (C#) de la présentation. Grâce à XAML, les développeurs peuvent facilement créer des interfaces complexes, en exploitant des fonctionnalités comme la liaison de données (data binding) et les styles pour personnaliser l'affichage.

9.1 XAML MainView

Ce UserControl sert à afficher les données que le programme relève de l'onduleur :

- Il possède un Viewbox pour adapter dynamiquement la taille des composants à la taille de la fenêtre.
- Une Grid organise l'affichage des composants en lignes et colonnes.
- Des TextBlock affichent des données liées à l'énergie et à la consommation relevées.
- Deux boutons permettant de démarrer ou d'arrêter le processus de collecte de données.

```
</UserControl.Resources>
<Viewbox Stretch="Uniform">
    <Grid Background="White"
    Height="600"
    Width="1266">
        <Grid.RowDefinitions>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition Height="50"></RowDefinition>
            <RowDefinition Height="50"></RowDefinition>
            <RowDefinition Height="50"></RowDefinition>
            <RowDefinition></RowDefinition>
            <RowDefinition></RowDefinition>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
            <ColumnDefinition></ColumnDefinition>
        </Grid.ColumnDefinitions>
        <Button Grid.Column="0" Content="Start" Command="{Binding StartCommand}" FontFamily="Arial" FontWeight="Bold" B
        <Button Grid.Column="0" Grid.Row="1" Content="Stop" Command="{Binding StopCommand}" FontFamily="Arial" FontWeig

        <TextBlock Text="{Binding LocalDateTime}" Grid.Column="1" Grid.Row="0" Margin="0,0,0,0" TextAlignment="Center"

        <TextBlock Text="Puissance de sortie PV" Grid.Column="2" Grid.Row="0" Margin="0,10,0,0" HorizontalAlignment="Ce
```

```
<TextBlock Text="Puissance de sortie PV" Grid.Column="2" Grid.Row="0" Margin="0,10,0,0" HorizontalAlignment="Center"
<TextBlock Text="{Binding ActivePower}" Grid.Column="2" Grid.Row="0" Margin="0,26,0,0" TextAlignment="Center" Vertica
```

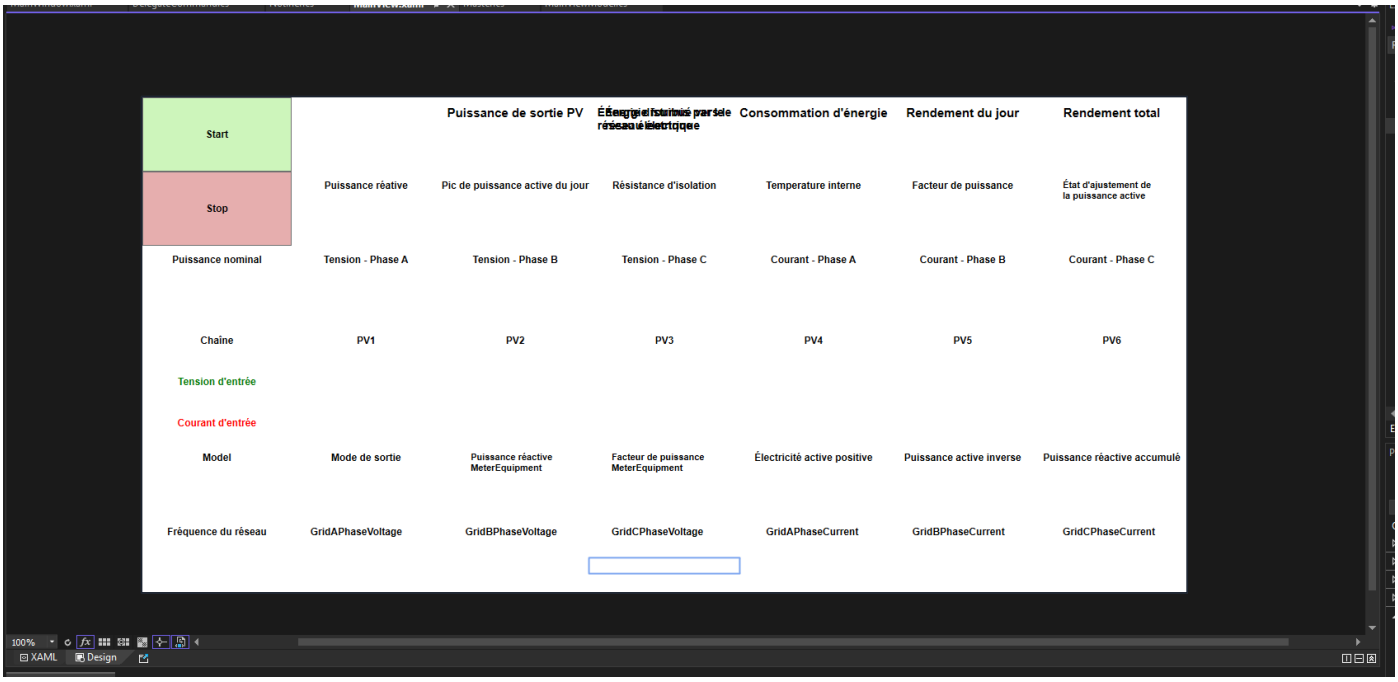
On peut définir des composants en spécifiant des attributs et des styles tels que :

- Text : permettant d'afficher du texte statique comme étiquette.
- Grid.Column et Grid.Row : positionner le composant à une certaine place dans la grille.
- Margin : ajouter des espacements de marges.
- Etc…

Binding permet d'affecter la valeur de Text à une propriété définie dans la ViewModel.

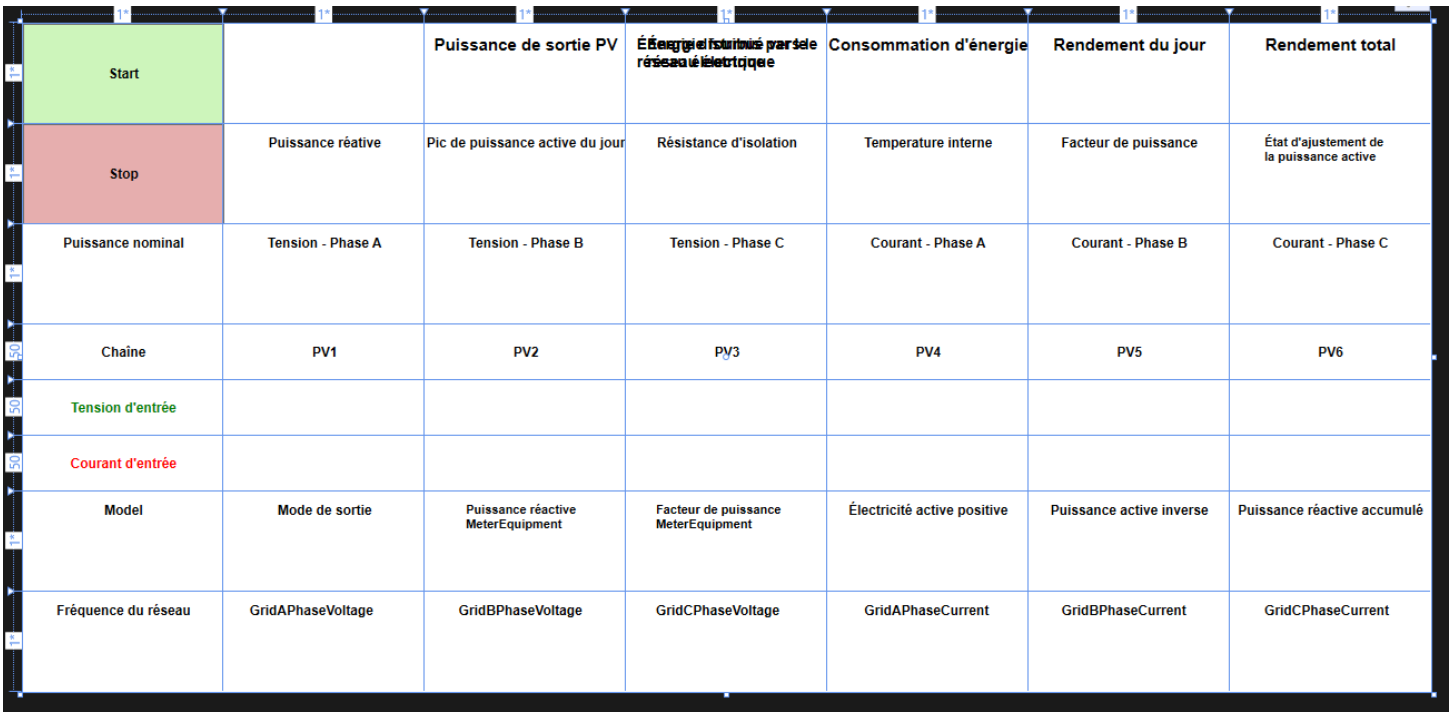
```
#region
ActivePower = InverterEquipmentRegister.ActivePower.GetFormattedValue([result.PhaseTotal[11], result.PhaseTotal[12]]);
```

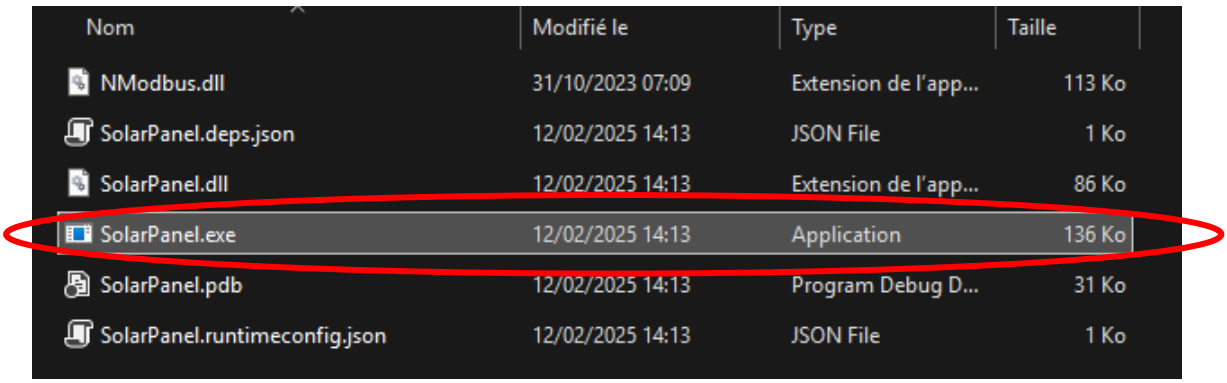
Voir introduction à XAML => https://pimo-wpf.netlify.app/cours/introduction/xaml/



9.2 XAML graphique MainView

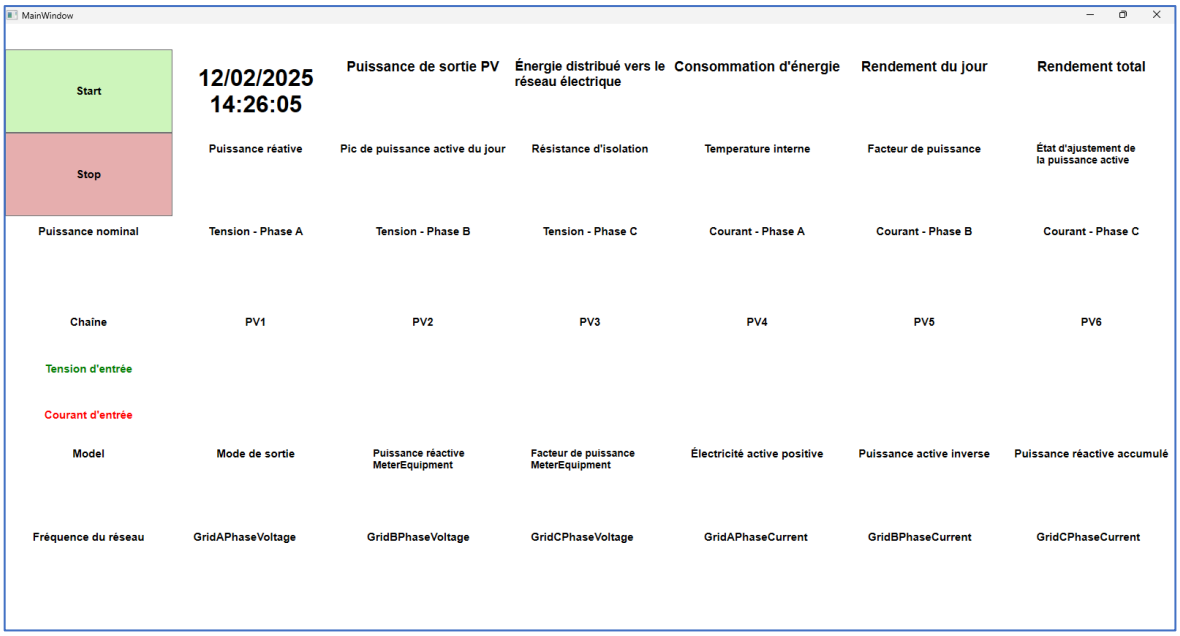
Une représentation graphique des composants définis dans le code XAML.



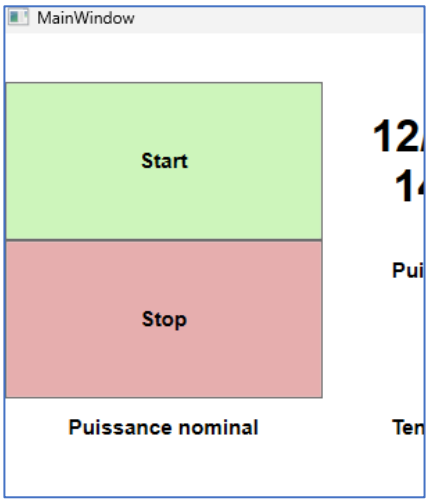


10 Guide d'utilisation

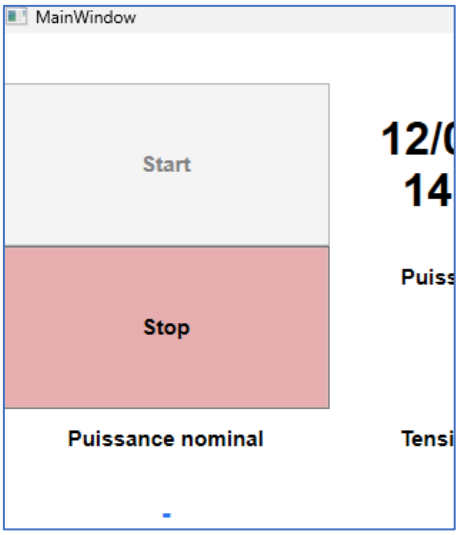
Ouvrir l'application « SolarPanel.exe ».



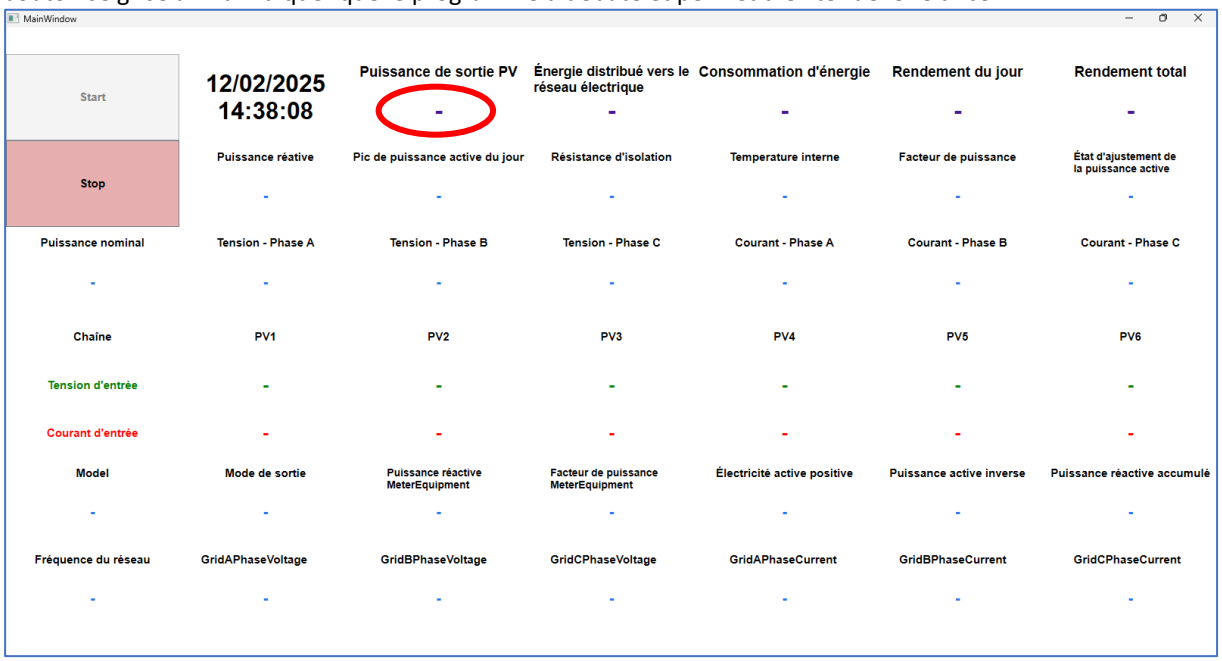
La fenêtre de l'application s'ouvre ainsi. Nous pouvons voir plusieurs informations affichées sur cette fenêtre, telles que deux boutons, la date, l'heure et les indications de ce que représenteront les valeurs affichées.



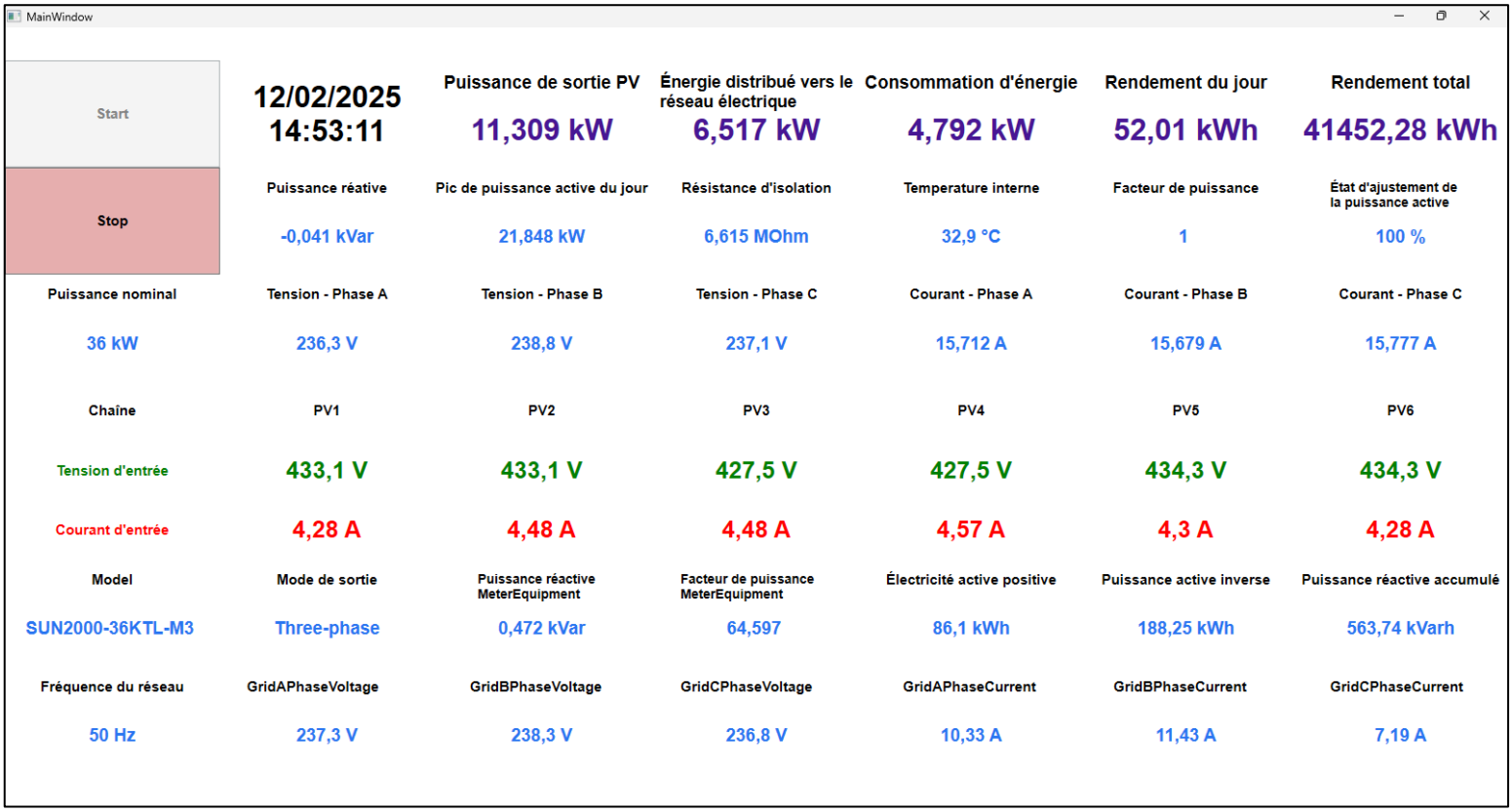
Le bouton Start lance l'exécution du programme permettant d'interroger les registres de l'onduleur, de traiter



ces données et de les afficher sur la fenêtre.
Lorsque l'on appuie sur le bouton Start, le programme initialise par défaut toutes les valeurs affichées à « - », le bouton se grise afin d'indiquer que le programme a débuté et permet d'éviter de le relancer.



Quelques instants après, les valeurs des registres s'affichent sur l'interface graphique indiquant les valeurs relevées par les équipements solaires en temps réel.

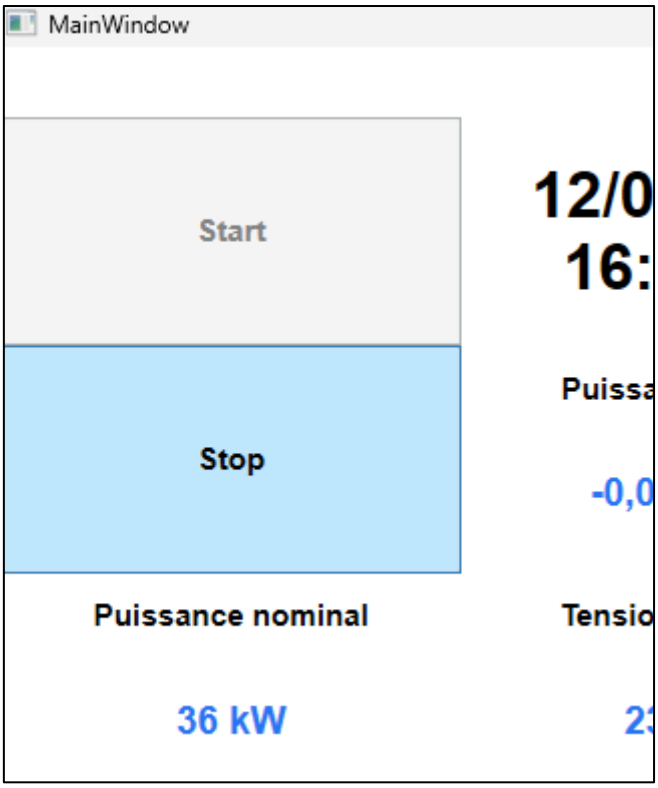


Correspondance des valeurs affichées :

- **Puissance de sortie PV** (ActivePower) : puissance active générée par les panneaux photovoltaïques alimentant le bâtiment, mesurée en kilowatts.
- **Énergie distribuée vers le réseau électrique / énergie fournie par le réseau électrique** (ActivePowerMeterEquipment) : puissance énergétique distribuée vers ou fournie par le réseau électrique, mesurée en kilowatt.

- **Consommation d'énergie** (EnergyConsumption) : puissance énergétique consommée par le bâtiment, mesurée en kilowatts.
- **Rendement du jour** (DailyEnergyYield) : rendement énergétique journalier produit, mesuré en kilowattheures.
- **Rendement total** (AccumulatedEnergyYield) : rendement énergétique cumulé depuis l'installation de l'équipement, mesuré en kilowattheures.
- **Puissance réactive** (ReactivePower) : puissance générée « non utile » ou « invisible » de l'électricité, mesurée en kilovoltampère.
- **Pic de puissance active du jour** (PeakActivePowerOfCurrentDay) : pic de puissance énergétique journalier généré par l'équipement, mesuré en kilowatts.
- **Résistance d'isolation** (InsulationResistance) : résistance d'isolation électrique entre les parties sous tension et la terre, permettant de détecter d'éventuels défauts d'isolement dans le système PV, mesurée en mégaohms.
- **Température interne** (InternalTemperature) : interne des composants électroniques de l'équipement, mesurée en degrés Celsius.
- **Facteur de puissance** (PowerFactor) : rapport entre la puissance active et la puissance apparente.

- **État d'ajustement de la puissance active** (ActivePowerPercentageDerating) : pourcentage de réduction ou de modulation de la puissance active de l'onduleur par rapport à sa capacité nominale, mesurée en pourcentage.
- **Puissance nominale** (RatedPower) : puissance maximale que peut fournir l'équipement en fonctionnement normal, mesurée en kilowatts.
- **Tension – Phase A** (PhaseAVoltage) : tension mesurée sur la phase A en volts.
- **Tension – Phase B** (PhaseBVoltage) : tension mesurée sur la phase B en volts.
- **Tension – Phase C** (PhaseCVoltage) : tension mesurée sur la phase C en volts.
- **Courant – Phase A** (PhaseACurrent) : courant mesuré sur la phase A en ampères.
- **Courant – Phase B** (PhaseBCurrent) : courant mesuré sur la phase B en ampères.
- **Courant – Phase C** (PhaseCCurrent) : courant mesuré sur la phase C en ampères.
- **Tension d'entrée – PV1** (PV1Voltage) : tension fournie par la chaine de panneaux photovoltaïque 1.
- **Tension d'entrée – PV2** (PV2Voltage) : tension fournie par la chaine de panneaux photovoltaïque 2.
- **Tension d'entrée – PV3** (PV3Voltage) : tension fournie par la chaine de panneaux photovoltaïque 3.
- **Tension d'entrée – PV4** (PV4Voltage) : tension fournie par la chaine de panneaux photovoltaïque 4.
- **Tension d'entrée – PV5** (PV5Voltage) : tension fournie par la chaine de panneaux photovoltaïque 5.
- **Tension d'entrée – PV6** (PV6Voltage) : tension fournie par la chaine de panneaux photovoltaïque 6.
- **Courant d'entrée – PV1** (PV1Current) : courant fourni par la chaine de panneaux photovoltaïque 1.
- **Courant d'entrée – PV2** (PV2Current) : courant fourni par la chaine de panneaux photovoltaïque 2.
- **Courant d'entrée – PV3** (PV3Current) : courant fourni par la chaine de panneaux photovoltaïque 3.
- **Courant d'entrée – PV4** (PV4Current) : courant fourni par la chaine de panneaux photovoltaïque 4.
- **Courant d'entrée – PV5** (PV5Current) : courant fourni par la chaine de panneaux photovoltaïque 5.
- **Courant d'entrée – PV6** (PV6Current) : courant fourni par la chaine de panneaux photovoltaïque 6.
- **Model** (ModelValue) : modèle de l'onduleur.
- **Mode de sortie** (Metertype) : type de compteur d'énergie.
- **Puissance réactive MeterEquipment** (ReactivePowerMeterEquipment) : puissance « non utile » ou « invisible » de l'électricité mesurée par le compteur électrique, mesurée en voltampère.
- **Électricité active positive** (PositiveActiveElectricity) : électricité nourrie par l'onduleur au réseau électrique.
- **Puissance active inversée** (ReversActivePower) : puissance fournie au système de distribution par le réseau électrique.
- **Puissance réactive accumulée** (AccumulatedReactivePower) : quantité totale d'énergie réactive qui a été produite ou consommée par l'onduleur, mesurée en kilo voltampère heure.
- **Fréquence du réseau** (GridFrequency) : fréquence à laquelle l'onduleur injecte de l'énergie dans le réseau électrique, mesurée en hertz.
- **GridAPhaseVoltage** (GridAPhaseVoltage) : tension mesurée sur la phase A du réseau électrique en volts.
- **GridBPhaseVoltage** (GridBPhaseVoltage) : tension mesurée sur la phase B du réseau électrique en volts.
- **GridCPhaseVoltage** (GridCPhaseVoltage) : tension mesurée sur la phase C du réseau électrique en volts.
- **GridAPhaseCurrent** (GridAPhaseCurrent) : courant mesuré sur la phase A du réseau électrique en ampères.
- **GridBPhaseCurrent** (GridBPhaseCurrent) : courant mesuré sur la phase B du réseau électrique en ampères.
- **GridCPhaseCurrent** (GridCPhaseCurrent) : courant mesuré sur la phase C du réseau électrique en ampères.



Lorsque l'on appuie sur le bouton Stop, une fenêtre de dialogue s'ouvre demandant la confirmation de quitter l'application. Une fois avoir confirmé, exécute l'arrêt du programme et la fermeture de l'application.