Supervision et contrôle d'un onduleur Fronius GEN24 avec un ESP32

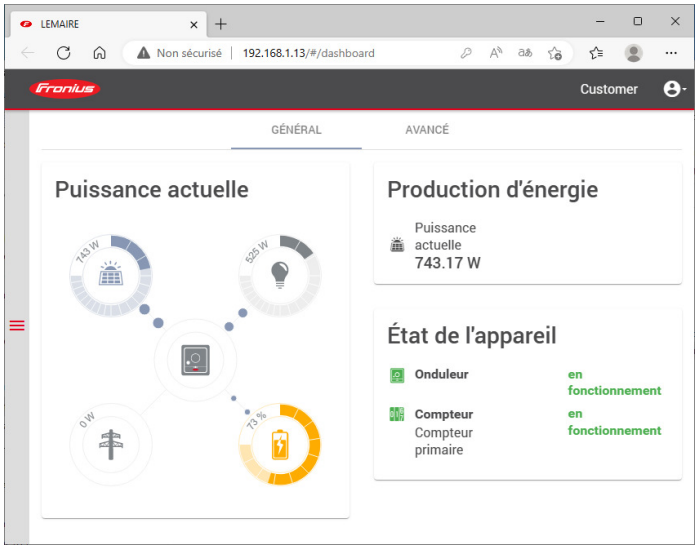
J. Lemaire[1]

(Pierrefeu – février 2023)

1 Objectifs

Dans le domaine du photovoltaïque résidentiel, la société Fronius propose des gammes complètes d'onduleurs hybrides polyvalents, notamment celle des **Primo GEN24** en monophasé et celle des **Symo GEN24** en triphasé[2] :

Ils peuvent se connecter comme **station Wifi** dans un réseau local et implémentent un serveur Web qui permet leur paramétrage, la supervision et le contrôle en local du système photovoltaïque, avec un simple navigateur Internet :

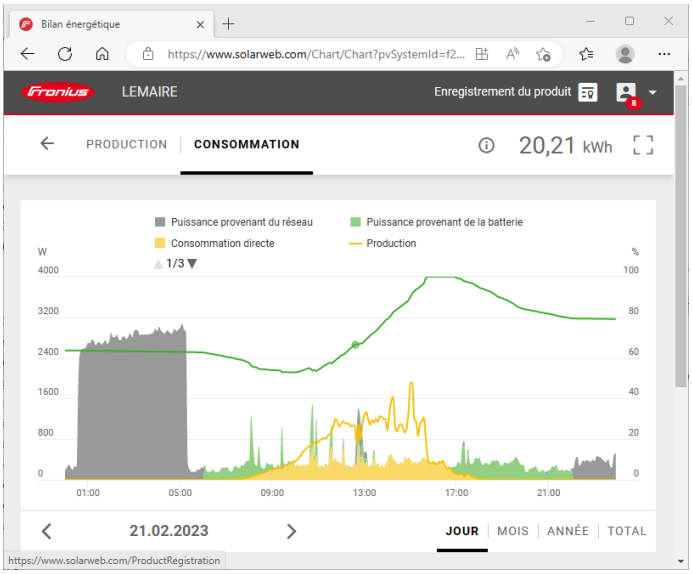


---

[1] jacques_lemaire@orange.fr

[2] https://www.fronius.com/fr-fr/france/energie-solaire/installateurs-et-partenaires/produits-et-solutions/solutions-energetiques-residentielles/gen24plus-onduleur-hybride-avec-mode-secours-integre

Si ce réseau local possède un routeur d'accès Internet, ils peuvent communiquer en temps réel leur état au service Fronius **Solar.web**, qui permet là encore de superviser le fonctionnement du système avec un simple navigateur Internet, mais cette fois à distance et avec beaucoup plus de possibilités, graphiques par exemple :



Mais il peut être utile de pouvoir accéder à ces données à partir de logiciels tiers souvent utilisés dans le domaine du photovoltaïque, **Home Assistant** par exemple. Et bien que déjà très complets et parfaitement ergonomiques, ces outils Fronius de supervision et de contrôle peuvent être complétés : par exemple, le pilotage d'une charge non réactive ne fonctionne pas de manière optimale en mode 0-injection et il est n'est pas possible de modifier automatiquement les contraintes de charge ou de décharge de la batterie en fonction des prévisions météorologiques du lendemain.

Fronius propose plusieurs interfaces permettant des accès externes à ses systèmes :

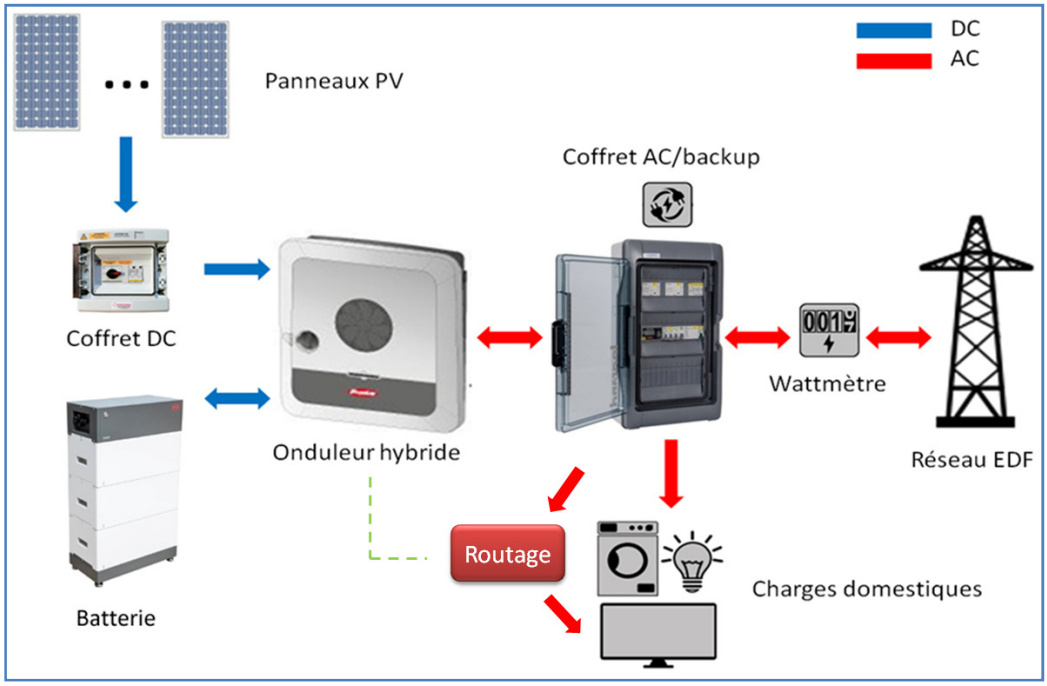
1. l'API **Solar.API** du serveur Web de l'onduleur ;
2. l'API **Solar.Web Query API** du service Solar.web ;
3. l'API **Modbus RTU** de l'onduleur ;
4. l'API **Modbus TCP** de l'onduleur.

Les 2 premières permettent uniquement la supervision du système alors que les 2 dernières en permettent la supervision et le contrôle.

Les API numéro 1 et 4 ont été testées ici avec un microcontrôleur de type ESP32, connecté en Wifi, comme station, au réseau local dont fait partie l'onduleur : la carte **ESP32 DevkitC V4 avec antenne externe** a été utilisée ici, mais on peut aussi faire ces tests avec d'autres cartes ESP32.

Quelques exemples très simples de supervision et de contrôle ont été mis en œuvre et testés sur le système photovoltaïque installé ici, comprenant entre-autres :

- un onduleur Fronius Primo GEN24 Plus 6.0 ;
- une batterie BYD HVS 7.7 ;
- un wattmètre Fronius Smart Meter TS 100A 1.


DC
AC
Panneaux PV
Coffret AC/backup
Coffret DC
Onduleur hybride
Wattmètre
Réseau EDF
Batterie
Routage
Charges domestiques

2 GEN24 comme station Wifi dans un réseau local

Pour que les techniques ici mises en œuvre fonctionnent, l'onduleur doit d'abord être intégré comme **station** dans un réseau local défini par exemple avec une box Internet, une **Livebox Orange** dans notre cas. Et bien évidemment, il sera plus simple de le faire en établissant une connexion **Wifi**.

L'intégration se fait généralement quand on commence à paramétrer l'onduleur, par exemple avec l'application **Solar.start** sur un smartphone, connecté en Wifi à l'onduleur démarré comme point d'accès Wifi.

Dans notre cas, voici les paramètres installés :

Dès ce paramétrage effectué, on peut redémarrer l'onduleur, cette fois comme station Wifi d'adresse IP 192.168.1.13 ; pour continuer son paramétrage, il suffira de se connecter à son serveur Web, en entrant cette adresse dans un navigateur Internet.

Mais auparavant, il est important de réserver cette adresse IP, attribuée dynamiquement par la box Internet, de manière à ce qu'elle ne puisse plus changer ensuite. Dans le cas d'une Livebox, il suffit de se connecter comme administrateur[3] à son serveur Web, en entrant l'adresse IP 192.168.1.1 dans un navigateur Internet. Dans notre cas, c'est la page **Configuration avancée/DHCP** de ce serveur qui permet de le faire, en ajoutant l'onduleur dans la liste des adresses **statiques** attribuées :

[3] ID = admin, PWD = 8 premiers caractères de sa clé Wifi par défaut.

DHCP | NAT/PAT | DNS | UPnP | DynDNS | DMZ | NTP

le serveur DHCP de la Livebox permet d'attribuer une adresse IP à chaque appareil de votre réseau local.

configuration DHCP

serveur DHCP IPv4	(•) activer ( ) désactiver
adresse IP de la Livebox	192.168.1.1
masque de sous-réseau du LAN	255.255.255.0
adresse IP de début	192.168.1.10
adresse IP de fin	192.168.1.150

annuler enregistrer

Vous pouvez visualiser les adresses IP dynamiques attribuées par le serveur DHCP de la Livebox.

adresse IP dynamique

nom	adresse IP	adresse MAC
chuangmi.camera.ipc019	192.168.1.10	78:8b:2a:d9:fd:90
Galaxy-S20-FE	192.168.1.18	2a:a7:ec:be:a5:0f
Android	192.168.1.11	d8:ce:3a:e8:be:2d
Fronius-33097736	192.168.1.13	78:c4:0e:b2:b1:b2
PC-JAC	192.168.1.21	00:21:5d:14:9c:52
MX720 series (UPnP)_93A707EC2A23	192.168.1.41	f4:81:39:93:a7:07

Vous pouvez réserver une adresse IP statique à chaque équipement de votre réseau local. L'équipement aura donc systématiquement la même adresse sur votre réseau local.

adresse IP statique

nom	adresse IP	adresse MAC	
chuangmi.camera.ipc019	192.168.1.10	78:8b:2a:d9:fd:90	ajouter
Fronius-33097736	192.168.1.13	78:c4:0e:b2:b1:b2	supprimer

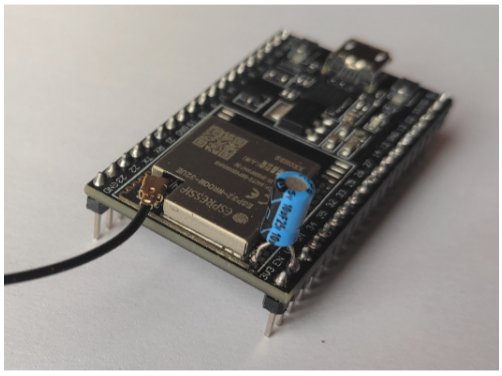
3 Carte ESP32 DevkitC V4[4]

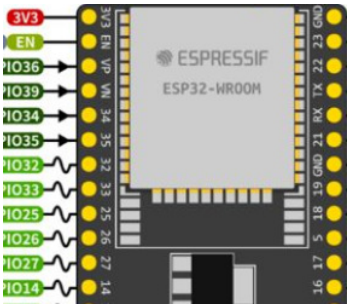
Basée ici sur le module **Espressif ESP32-WROOM-32UE**, on peut y connecter une antenne externe 2,4GHz pour améliorer la qualité de la transmission Wifi :

Elle a été programmée avec un PC sous Windows 10, en utilisant la **version 1.8.19 de l'IDE Arduino**[5], complété par l'ajout des cartes basées sur un ESP32[6]. On peut alors choisir « ESP32 Dev Module » pour la déclarer dans le menu Outils de l'IDE.

Pour y transférer un programme (sketch), il suffit de la connecter au PC via son port micro-USB qui permet également de l'alimenter[7].

Par contre, ce transfert n'est possible que si la carte est un mode « bootloader » et il faut a priori maintenir enfoncé le bouton « Boot » pendant tout le transfert pour installer ce mode. On peut éviter cela en soudant un condensateur électrolytique de 10µF entre la broche EN de la carte et la patte GND de son module Espressif[8] :





[4] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/hw-reference/esp32/get-started-devkitc.html#get-started-esp32-devkitc-board-front

[5] https://www.arduino.cc/en/software

[6] Ajouter l'URL https://dl.espressif.com/dl/package_esp32_index.json dans le fichier **preferences.txt** puis utiliser le **Gestionnaire de carte** pour installer **esp32**.

[7] Outre l'utilisation du port USB, on peut aussi l'alimenter en utilisant ses broches 5V et GND ou bien 3V3 et GND, mais à condition de ne choisir qu'une seule de ces 3 possibilités.

[8] https://randomnerdtutorials.com/solved-failed-to-connect-to-esp32-timed-out-waiting-for-packet-header/

4 Supervision avec l'API Solar API[9]

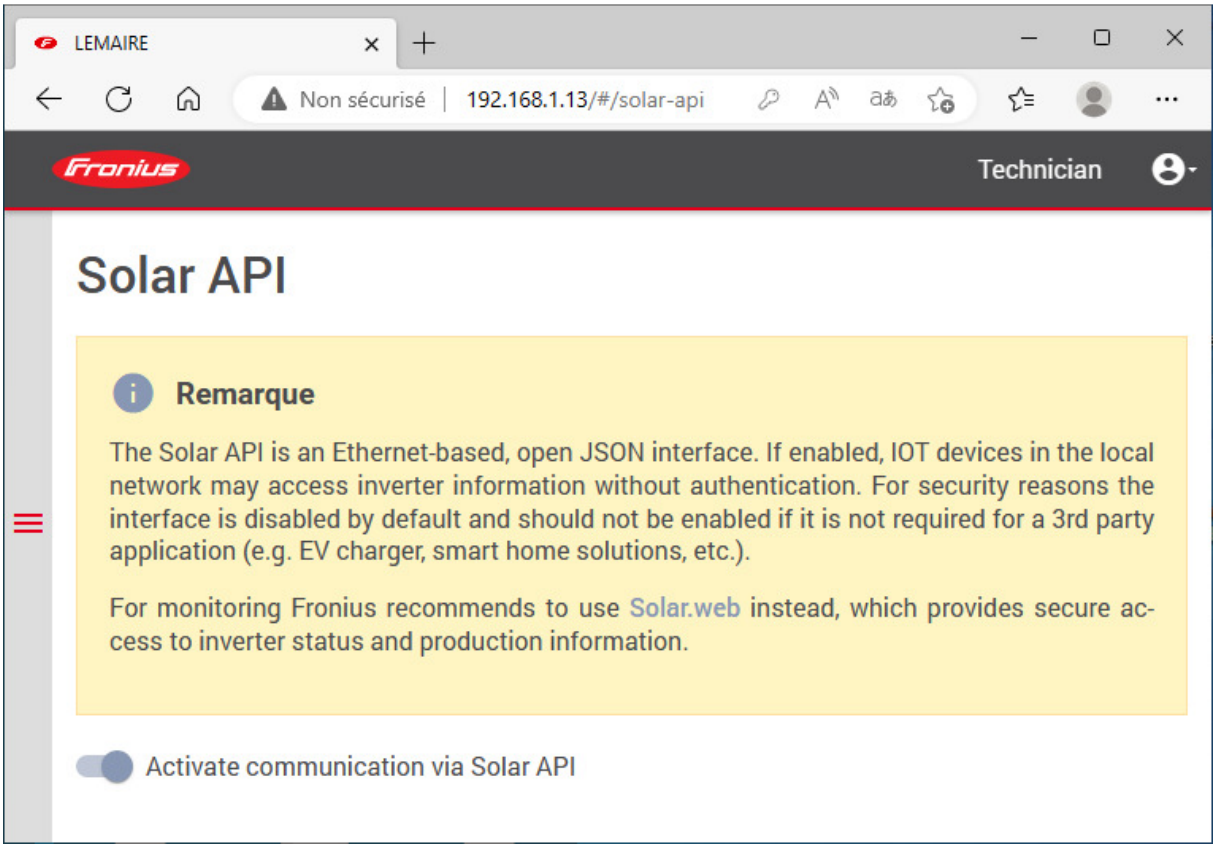
4.1 Description et activation

Cette API expose plusieurs scripts CGI, permettant d'accéder aux données de l'onduleur, de la batterie, du wattmètre et d'autres équipements gérables par l'onduleur[10] :

- **GetAPIVersion.cgi**
- **GetInverterRealtimeData.cgi**
- **GetMeterRealtimeData.cgi**
- **GetStorageRealtimeData.cgi**
- **GetPowerFlowRealtimeData.fcgi**
- etc.

Ceux-ci retournent toujours des données textuelles en format **JSON**[11].

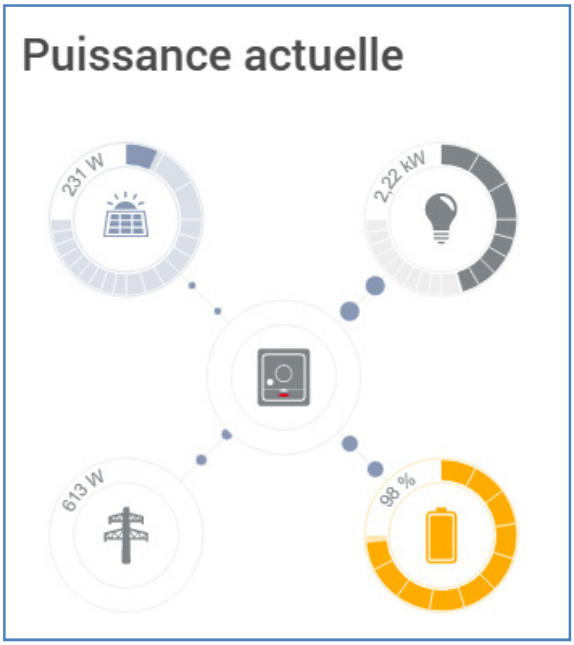
L'API doit d'abord être activée en se connectant avec le compte **Technician** à la page http://192.168.1.13/#/solar-api du serveur Web de l'onduleur :

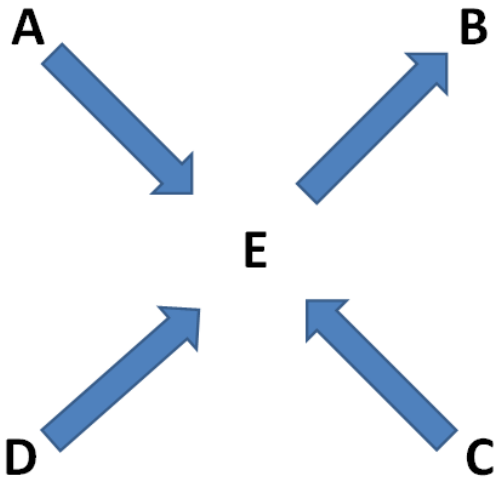


[9] https://www.fronius.com/~/downloads/Solar%20Energy/Operating%20Instructions/42,0410,2012.pdf
[10] OhmPilot ou WattPilot par exemple.
[11] https://fr.wikipedia.org/wiki/JavaScript_Object_Notation par exemple.

4.2 Exemple

On se propose d'afficher les 4 flux de puissance gérés par l'onduleur :





Les flux de puissance sont évalués en Watts (W) :

- P_A = puissance de la **production PV** (toujours ≥ 0)
- P_B = puissance de la **consommation** (toujours ≥ 0)
- P_C[12] :
  - P_C^+ = puissance de la **décharge**
  - P_C^- = puissance de la **charge**
- P_D :
  - P_D^+ = puissance du **soutirage**
  - P_D^- = puissance de l'**injection**

On observera que $P_B < P_A + P_C + P_D$ à cause des pertes dans les conversions DC⇔AC.

On peut les obtenir en appelant le script **GetPowerFlowRealtimeData.fcgi** dans un navigateur, avec l'URL http://192.168.1.13/solar_api/v1/GetPowerFlowRealtimeData.fcgi :


```
{
   "Body" : {
      "Data" : {
         "Inverters" : {
            "1" : {
               "Battery_Mode" : "normal",
               "DT" : 1,
               "E_Day" : null,
               "E_Total" : 1757314.723888889,
               "E_Year" : null,
               "P" : 2182.44970703125,
               "SOC" : 97.599999999999994
            }
         },
         "SecondaryMeters" : {},
         "Site" : {
            "BackupMode" : false,
```


[12] Pour connaître sa valeur, il suffit de cliquer sur l'icône de la batterie.

```
            "BatteryStandby" : true,
            "E_Day" : null,
            "E_Total" : 1757314.723888889,
            "E_Year" : null,
            "Meter_Location" : "grid",
            "Mode" : "bidirectional",
            "P_Akku" : 2045.85791015625,
            "P_Grid" : 42.200000000000003,
            "P_Load" : -2224.6497070312498,
            "P_PV" : 238.76676940917969,
            "rel_Autonomy" : 98.103072143599874,
            "rel_SelfConsumption" : 100.0
         },
         "Smartloads" : {
            "Ohmpilots" : {}
         },
         "Version" : "12"
      }
   },
   "Head" : {
      "RequestArguments" : {},
      "Status" : {
         "Code" : 0,
         "Reason" : "",
         "UserMessage" : ""
      },
      "Timestamp" : "2023-02-22T15:36:16+00:00"
   }
}
```


Ici[13] :

- P_PV ➔ $P_A = 238{,}77\text{W}$
- P_Load ➔ $-P_B = -2224{,}65\text{W}$
- P_Akku = ➔ $P_C = 2045{,}86\text{W}$
- P_Grid ➔ $P_D = 42{,}20W$

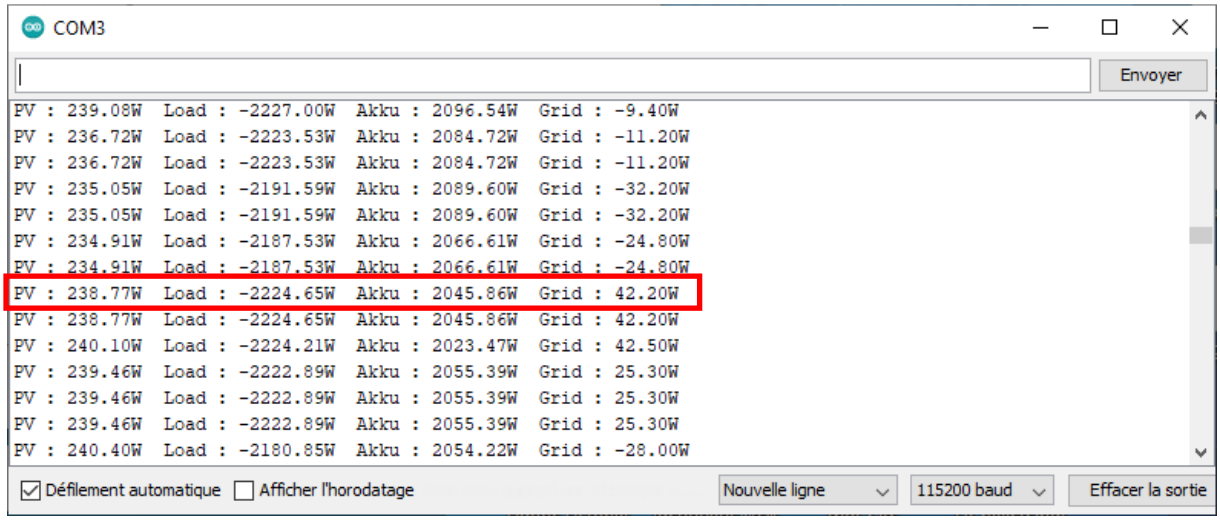
Code Arduino

```
// ClientReadPowerFlow.ino

/*****************************************************************************
  Test HTTP Client on ESP32 as Wifi station in a local network.
  Decode JSON data sent by the Fronius API of a GEN24 inverter.

  References :
  - https://randomnerdtutorials.com/esp32-http-get-post-arduino/
  - https://github.com/arduino-libraries/Arduino_JSON
  - https://github.com/pfeerick/elapsedMillis

*****************************************************************************
  Libraries and structures
*****************************************************************************/

// Wifi
#include <WiFi.h>

// HTTPClient
#include <HTTPClient.h>

// JSON
#include <Arduino_JSON.h>

// Timer
#include <elapsedMillis.h>

/*****************************************************************************
  Constants
*****************************************************************************/
```

[13] Les flux sont orientés positivement vers l'onduleur.

```
// Local network access point (to complete)
const char *SSID = "Livebox-XXXX";
const char *PWD = "YYYYYYYYYYYYYYYY";

// Wifi period (ms)
#define WIFI_PERIOD 1000

/*****************************************************************************
  Global variables
*****************************************************************************/

// Parameters
double pv;   // PV production
double load; // Consommation
double akku; // Battery
double grid; // Grid


// Timer
elapsedMillis tmrWifi;

/*****************************************************************************
  Functions
*****************************************************************************/

void setup()
{
  // Connect to the Wifi access point
  WiFi.begin(SSID, PWD);
  while (WiFi.status() != WL_CONNECTED) delay(500);

  // Open serial port
  Serial.begin(115200);
  while (!Serial) {}
}

void loop()
{
  if (tmrWifi > WIFI_PERIOD)
  {
    // Reset timer
    tmrWifi = 0;

    // Reconnect Wifi if necessary
    if (WiFi.status() == WL_CONNECTED)
    {
      // GET request
      HTTPClient http;
      http.begin("http://192.168.1.13/solar_api/v1/GetPowerFlowRealtimeData.fcgi");
      if(http.GET() == HTTP_CODE_OK)
      {
        // Decode data
        String s = http.getString();
        pv = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_PV"]);
        load = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_Load"]);
        akku = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_Akku"]);
        grid = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_Grid"]);


        // Display data
        Serial.print("PV : "); Serial.print(pv); Serial.print("W  ");
        Serial.print("Load : "); Serial.print(load); Serial.print("W  ");
        Serial.print("Akku : "); Serial.print(akku); Serial.print("W  ");
        Serial.print("Grid : "); Serial.print(grid); Serial.println("W");
      }
      http.end();
    }
    else WiFi.reconnect();
  }
}
```

Valeurs affichées sur le moniteur série



Commentaires

Avant de lancer le programme, il faut installer les bibliothèques non déjà incluses dans l'IDE Arduino + ESP32 : **Arduino_JSON** et **elapsedMillis**[14] ici.

Le programme commence par inclure les entêtes des bibliothèques utilisées :

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <Arduino_JSON.h>
#include <elapsedMillis.h>
```

Dans la fonction **setup**, il connecte l'ESP32 comme **station Wifi** au réseau local de la Livebox :

```
WiFi.begin(SSID, PWD);
while (WiFi.status() != WL_CONNECTED) delay(500);
```

Dans la fonction **loop**, toutes les secondes, si cette connexion est valide, il envoie une **requête http GET** et si le serveur http (ici l'onduleur) y répond correctement :

```
HTTPClient http;
http.begin("http://192.168.1.13/solar_api/v1/GetPowerFlowRealtimeData.fcgi");
if(http.GET() == HTTP_CODE_OK)
```

il récupère la chaîne de caractères transmise en format JSON :

```
String s = http.getString();
```

et en extrait les valeurs intéressantes :

```
pv = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_PV"]);
load = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_Load"]);
akku = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_Akku"]);
grid = (double)(JSON.parse(s)["Body"]["Data"]["Site"]["P_Grid"]);
```

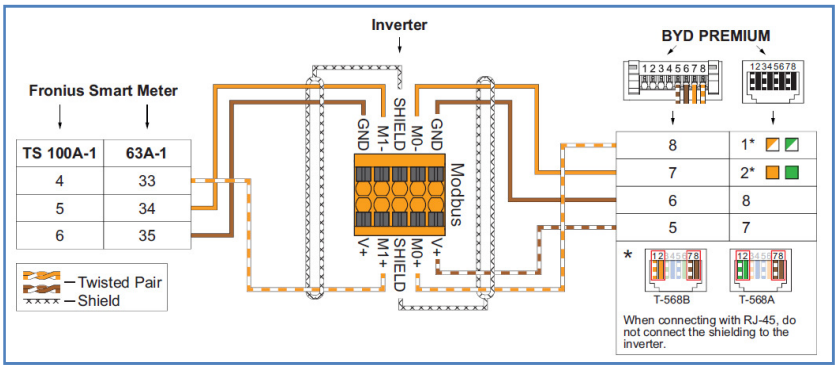
Le reste est évident.

[14] On peut le faire avec la commande **Inclure une bibliothèque/Ajouter la bibliothèque.ZIP…**, après avoir téléchargé les fichiers .zip de ces bibliothèques, disponibles sur Github.

5 Supervision et contrôle avec l'API Modbus TCP[15] [16]

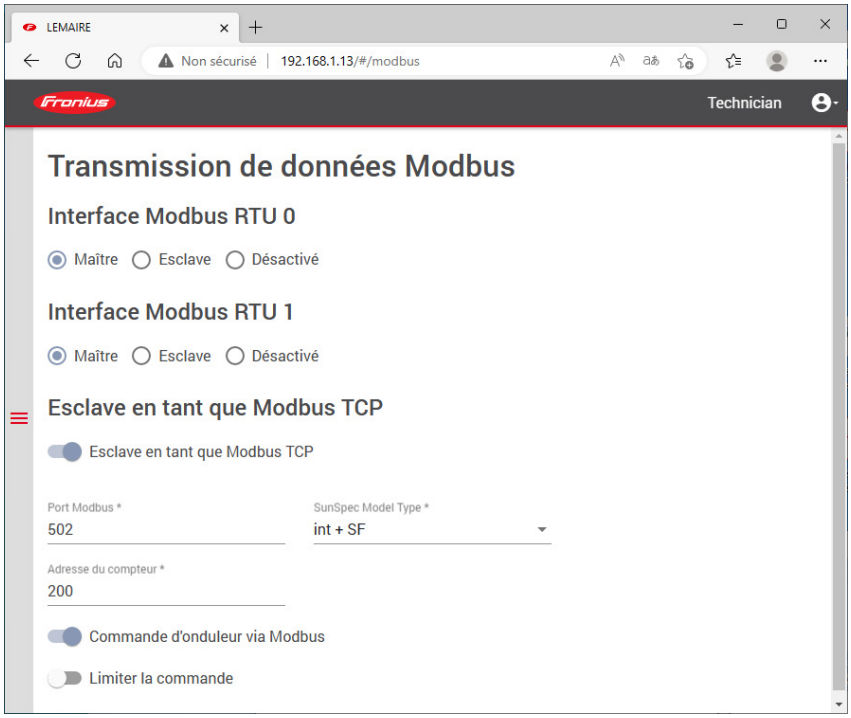
5.1 Description et activation

L'onduleur communique avec les autres composants du système photovoltaïque (wattmètre et batterie ici) en utilisant le protocole **Modbus RTU**[17] via des liaisons filaires[18] :



Il dispose de 2 interfaces RTU où il joue le rôle de **Maître**, la batterie et le wattmètre jouant un rôle d'**Esclave**[19].

Mais on peut aussi activer un second mode de communication, qui utilisera le protocole **Modbus TCP**[20] et dans lequel l'onduleur pourra jouer parallèlement un rôle de d'**Esclave** :



---

[15] https://www.fronius.com/~/downloads/Solar%20Energy/Operating%20Instructions/42,0410,2649.pdf

[16] https://www.youtube.com/watch?v=UMR_4rqxZts (Webinar- Third-party monitoring and control via Sunspec Modbus RS485 TCP-IP interface41).

[17] https://modbus.org/specs.php et https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf par exemple.

[18] Protocole RS-485.

[19] Il peut même ainsi piloter d'autres onduleurs.

[20] https://view.officeapps.live.com/op/view.aspx?src=https%3A%2F%2Fmodbus.org%2Fdocs%2FObject_Messaging_Protocol_ExtensionsVers1.1.doc&wdOrigin=BROWSELINK

On notera au passage que ce serveur Modbus TCP de l'onduleur répondra cette fois sur le port TCP **502**[21], que les données de ce serveur Modbus TCP suivront la norme **SunSpec**[22] **int+SF**[23] et qu'il faudra utiliser l'adresse **200**[24] pour le wattmètre, l'adresse de l'onduleur et de la batterie étant égale à **1** (valeur par défaut). Dans la suite, ces adresses seront appelées **ID d'unité** pour ne pas les confondre avec les adresses de registres.

Si l'onduleur est connecté en Wifi comme station dans un réseau local, ce nouveau mode de communication Modbus s'effectuera en Wifi.

Il est hors de propos d'expliquer ici en détail le fonctionnement du **protocole Modbus**[25], mais on va quand-même en préciser quelques concepts, pour bien comprendre la formulation des requêtes.

Dans le **protocole Modbus TCP**, on parle plutôt de **communication Client-Serveur**, l'Esclave étant ici le Serveur. Là encore, on va donc construire un client qui formulera des requêtes Modbus TCP, en s'appuyant sur le protocole TCP/IP.

Dans le protocole Modbus (RTU ou TCP), le serveur doit définir un certains nombre de **registres** auxquels le client peut accéder en lecture ou lecture/écriture. Dans notre cas, une application externe ne pourra accéder qu'à un seul type de registres, les **Holding Registers**, chaque registre occupant 16 bits de mémoire (1 word) et il n'y aura que 3 opérations possibles pour le client :

1. **Read Holding Registers** (x03) pour lire un ou plusieurs registres contigus ;
2. **Write Single Register** (0x06) pour écrire dans un seul registre ;
3. **Write Multiple Registers** (0x10) pour écrire dans un ou plusieurs registres contigus.

Les valeurs hexadécimales indiquées ici (3, 6 et 16 en décimal) sont les **Codes fonction** de ces opérations.

Dans tous les cas, il faudra préciser l'**adresse du registre de départ**, pour les opérations multiples le **nombre de registres traités** et, dans le cas des opérations d'écritures, la ou les valeurs à écrire.

Avant de mettre en œuvre tout ceci, il faut donc connaître les adresses et la signification de ces **Holding Registers** pour l'onduleur Fronius GEN24 : on les trouvera dans l'archive **gen24-modbus-api-external-docs.zip**[26] et, compte-tenu des options retenues pour activer Modbus TCP, dans les classeurs Excel suivants :

- Feuille **Complete Map** de **Gen24_Primo_Symo_Inverter_Register_Map_Int&SF_storage.xlsx** pour l'onduleur et la batterie ;
- Feuille **Complete Map de Smart_Meter_Register_Map_Int&SF** pour le wattmètre.

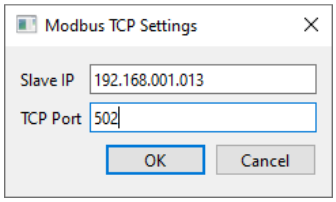
Par exemple, dans la première, on peut voir que les 16 registres dont l'adresse est comprise entre 40021 et 40036 contiennent le nom de l'onduleur : « Primo GEN24 6.0 » ici.

---

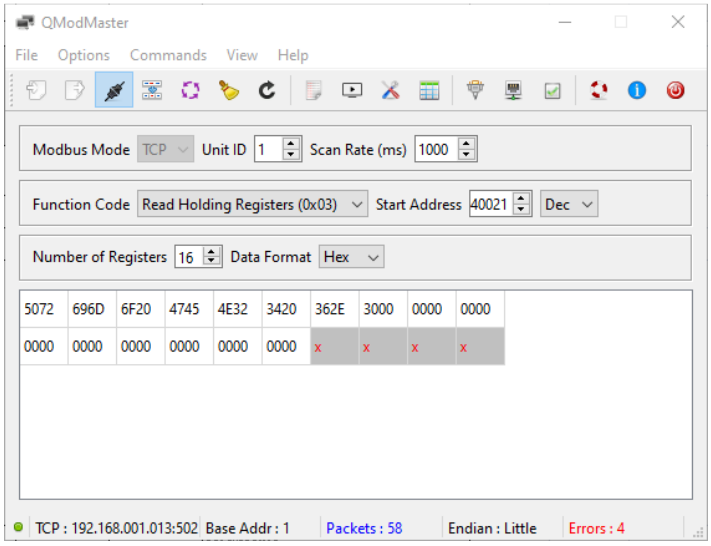
[21] Alors que le serveur Web de l'onduleur répond sur le port 80.
[22] https://sunspec.org/wp-content/uploads/2015/06/SunSpec-Information-Models-12041.pdf et https://mschoeffler.com/2019/11/02/sunspec-tutorial-part-i-modbus/ par exemple.
[23] Les valeurs numériques sont stockées en mémoire comme des entiers, avec indication éventuelle d'un facteur d'échelle (Scale Factor) : par exemple 42,38 sera stockée 4238 avec -2 comme facteur d'échelle.
[24] Appelée **ID d'unité** dans la suite.
[25] https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf par exemple.
[26] https://www.fronius.com/~/downloads/Solar%20Energy/Operating%20Instructions/gen24-modbus-api-external-docs.zip

On peut le voir avec un générateur de requêtes clientes Modbus TCP : **QModMaster**[27].

Il sera lancé avec les options suivantes :



Avec les paramètres suivant, on obtient ceci en exécutant les commandes **Connect** puis **Read/Write** :



La réponse est fournie par le server sous la forme d'une chaîne de caractères[28] codés en ASCII, **0x50 0x72 0x69 0x6D … 0x36 0x2E 0x30 0x00 …** ; elle correspond bien à « **Primo GEN24 6.0** » comme on peut le constater en utilisant la table suivante :

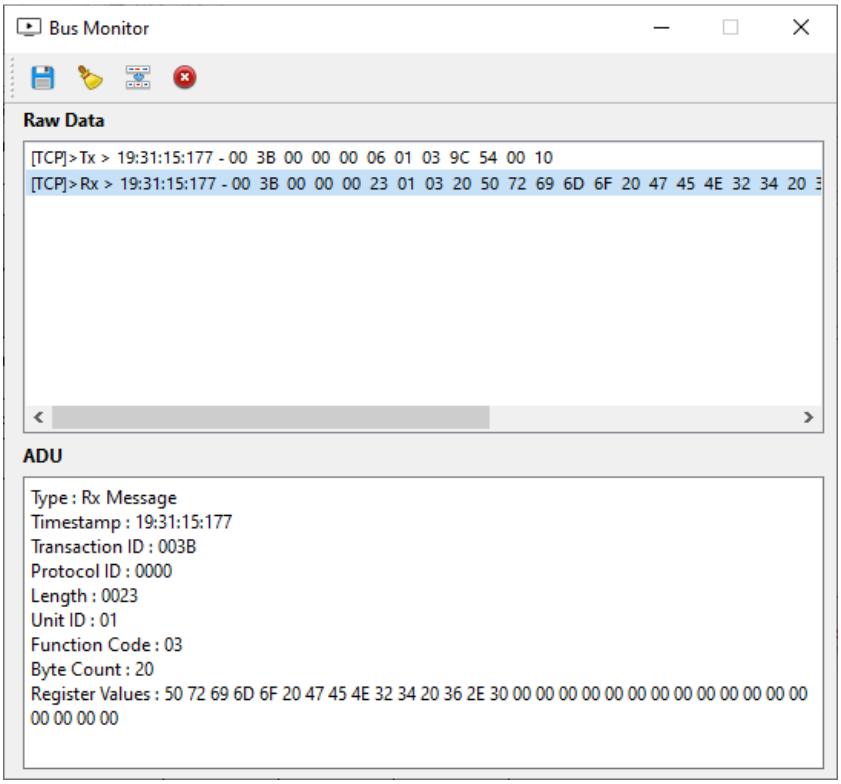
most significant nibble

least significant nibble	0_	1_	2_	3_	4_	5_	6_	7_
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL

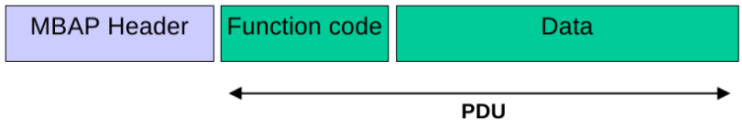
Cette application permet aussi de bien comprendre au passage le contenu des trames TCP échangées entre le client (le PC) et le serveur (l'onduleur), en affichant son **Bus Monitor** :

---

[27] https://sourceforge.net/projects/qmodmaster/
[28] Le code ASCI 0x00 indiquant la fin de la chaîne.



La **requête du client** contient un en-tête de 5 octets (MBAP) suivi d'une unité de données de protocole (PDU), avec le code fonction et les données :



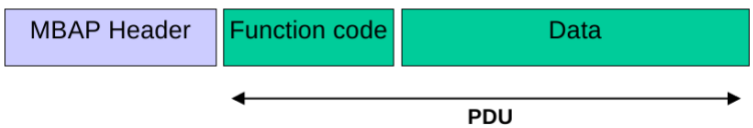
Le MBAP (7 octets) est structuré comme suit :

- Numéro de transaction = 0x003B (2 octets)
- ID du protocole = 0x0000 (2 octets)
- Longueur du PDU + l'ID d'unité = 0x0006 = 6 (2 octets)
- ID d'unité = 0x01 = 1 (1 octet)

Le PDU (5 octets ici) est structuré comme suit :

- Code fonction = 0x03 (1 octet)
- Data :
  - Adresse du premier registre – 1[29] = 0x9C54 = 40020 (2 octets)
  - Nombre de registres = 0x10 = 16 (2 octets)

La **réponse du serveur** a la même structure :



[29] Car, dans les modèles SunSpec, les adresses de registres démarrent à 1 et non à 0.

Le MBAP (7 octets) est structuré comme suit :

- Numéro de transaction = 0x003B (2 octets)
- ID du protocole = 0x0000 (2 octets)
- Longueur du PDU + l'ID d'unité = 0x0023 = 35 (2 octets)
- ID d'unité = 0x01 (1 octet)

Le PDU (34 octets ici) est structuré comme suit :

- Code fonction = 0x03 (1 octet)
- Data :
    - Nombre d'octets (2 x nb de registres lus) = 0x20 = 32 (1 octet)
    - Valeurs = 0x50 0x72 0x69 0x6D … 0x2E 0x30 0x00 … 0x00 (32 octets)

On constate donc au passage quelques détails qui permettront de bien comprendre les codes suivants, notamment en ce qui concerne le décalage de 1 dans l'adressage des registres SunSpec.

5.2 Exemple

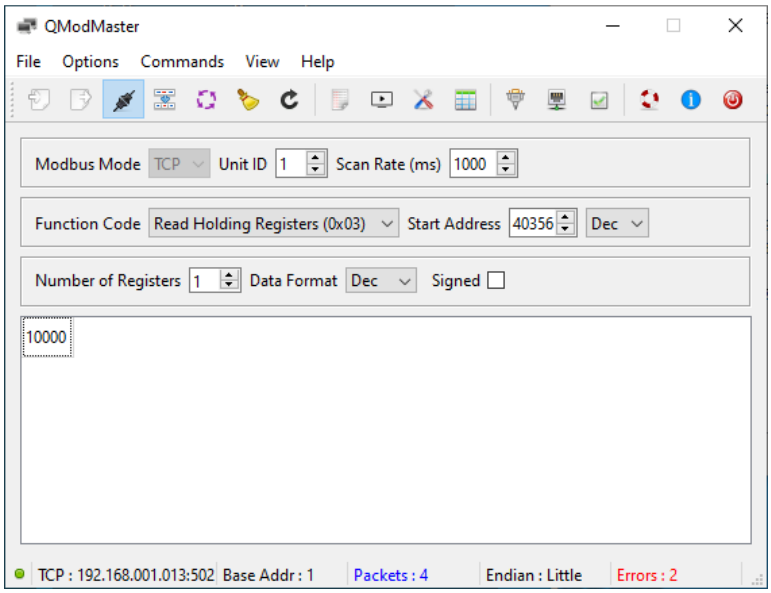
On va ici voir comment stopper la décharge de la batterie.

Le document **Operating Instructions – Fronius GEN24 Modbus TCP & RTU**[30] propose d'écrire la valeur **0** dans le registre **OutWRte** (Percent of max discharge rate[31], **#40356**) et la valeur **2** dans le registre **StorCtl_Mod** (Activate hold/discharge/charge storage control mode, **#40349**) du modèle SunSpec.

Au départ il n'y a pas de limitation dans la décharge de la batterie qui couvre toute la consommation[32] :

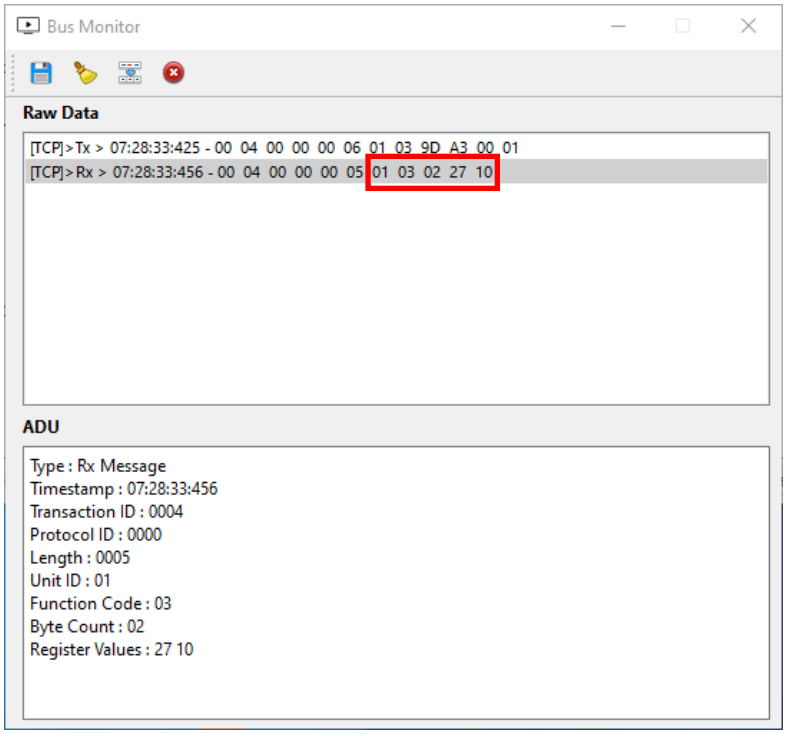
[30] https://www.fronius.com/~/downloads/Solar%20Energy/Operating%20Instructions/42,0410,2649.pdf
[31] Stocké dans WChaMax (#40346) qui contient ici la valeur 7680.
[32] Ces mesures ont été réalisées le soir, à une heure de très faible production du système photovoltaïque.

Avec **QModMaster**, on peut lire la valeur du registre **OutWRte** :



Elle est égale à **10000** = 0x2710, ce qui correspond à un taux de 100% car ce registre utilise le facteur d'échelle contenu dans **InOutWRte_SF** (#40369) où se trouve la valeur **-2**.

Là encore, on peut observer les trames échangées :



L'ID de l'unité + PDU est donc ici égal à 0x01 0x03 0x02 0x27 0x10 où :

- ID d'unité = 0x01 (1 octet)
- Code fonction = 0x03 (1 octet)
- Data :
    - Nombre d'octets = 0x02 = 2 (1 octet)
    - Valeurs = 0x27 0x10 (2 octets)

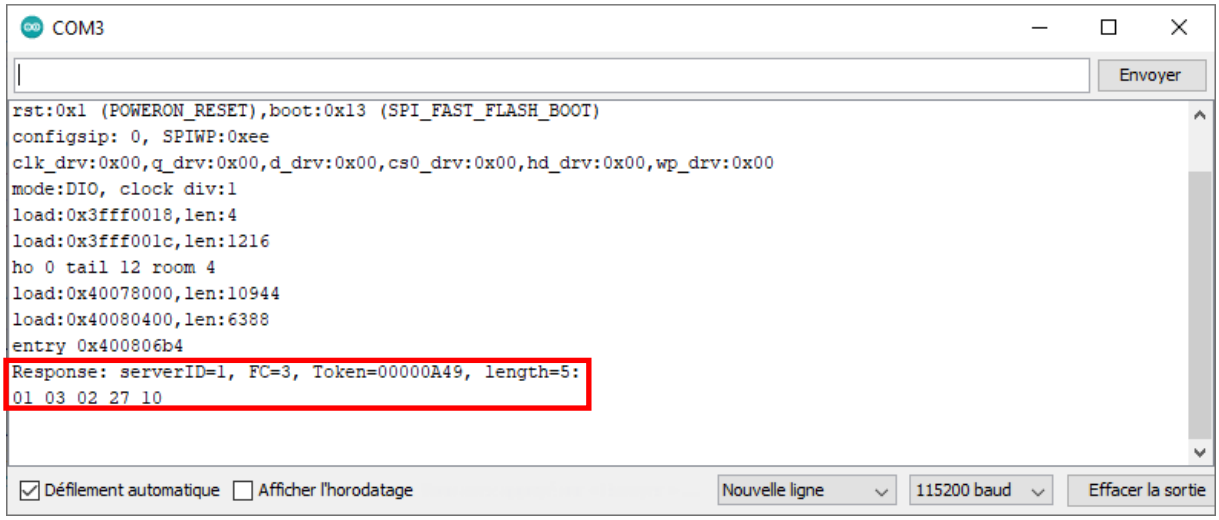
On va maintenant réaliser cette lecture avec le microcontrôleur.

Code Arduino

```
// ClientReadHoldingRegisters.ino

/*****************************************************************************
  Modbus TCP client (master) to read asynchronously contiguous holding registers in a
  Fronius GEN24 inverter (slave), using an ESP32 as Wifi station in a local network.

  References :
  - https://github.com/eModbus/eModbus

*****************************************************************************/

/*****************************************************************************
  Libraries and structures
*****************************************************************************/

// Wifi
#include <WiFi.h>

// Modbus Client TCP
#include <ModbusClientTCP.h>

/*****************************************************************************
  Constants
*****************************************************************************/

// Local network access point (to complete)
const char *SSID = "Livebox-XXXX";
const char *PWD = "YYYYYYYYYYYYYYYY";

/*****************************************************************************
  Global variables
*****************************************************************************/

// Wifi client
WiFiClient theClient;

// Create a ModbusTCP client instance
ModbusClientTCP MB(theClient);

/*****************************************************************************
  Functions
*****************************************************************************/

// Define an onData handler function to receive the regular responses
void handleData(ModbusMessage response, uint32_t token)
{
  Serial.printf("Response: serverID=%d, FC=%d, Token=%08X, length=%d:\n", response.getServerID(),
response.getFunctionCode(), token, response.size());
  for (auto& byte : response) Serial.printf("%02X ", byte);
  Serial.println("");
}

// Define an onError handler function to receive error responses
void handleError(Error error, uint32_t token)
{
  ModbusError me(error);
  Serial.printf("Error response: %02X - %s\n", (int)me, (const char *)me);
}

void setup()
{
    // Open serial port
  Serial.begin(115200);
  while (!Serial) {}

  // Connect to the Wifi access point
  WiFi.begin(SSID, PWD);
  while (WiFi.status() != WL_CONNECTED) delay(500);

  // Set up ModbusTCP client
  MB.onDataHandler(&handleData);
  MB.onErrorHandler(&handleError);
  MB.setTimeout(2000, 200); // Message timeout and interval between requests to the same host
```

```
  MB.begin();

  // Issue a request
  MB.setTarget(IPAddress(192, 168, 1,13), 502); // Modbus Server address and port
  uint8_t serverID = 1;
  uint8_t functionCode = READ_HOLD_REGISTER; // 0x03
  uint16_t startAddress = 40355; // => OutWRte (#40356) = percent of max decharge rate
  uint16_t nbRegisters = 1;  // Nb of registers
  Error err = MB.addRequest((uint32_t)millis(), serverID, functionCode, startAddress, nbRegisters);
  if (err!=SUCCESS)
  {
    ModbusError e(err);
    Serial.printf("Error creating request: %02X - %s\n", (int)e, (const char *)e);
  }
}

void loop() {}
```

Valeurs affichées sur le moniteur série



Commentaires

Avant de lancer le programme, il faut installer la bibliothèque **eModbus**[33] qui n'est pas incluse dans l'IDE Arduino + ESP32[34].

Le programme commence par inclure les entêtes des bibliothèques utilisées :

```
  #include <WiFi.h>
  #include <ModbusClientTCP.h>
```

Ensuite, il définit une variable globale de type ModbusClientTCP, essentielle ici :

```
  WiFiClient theClient;
  ModbusClientTCP MB(theClient);
```

Après, il implémente les 2 fonctions de rappel (callback) nécessaires pour le traitement des réponses du serveur en mode asynchrone :

```
void handleData(ModbusMessage response, uint32_t token)
{
  …
```

[33] Cette bibliothèque Arduino Modbus pour ESP32 (entre-autres), réalisée par Bert Melis, a été retenue car elle est très complète et permet un fonctionnement en mode asynchrone non bloquant des échanges entre le client et le serveur. Elle est illustrée par de nombreux exemples bien documentés (https://emodbus.github.io/). On notera également le caractère générique de son implémentation, notamment en ce qui concerne les fonctions de rappel, de manière à n'avoir que très peu de modifications à faire quand on change de requêtes.

[34] On peut le faire avec la commande **Inclure une bibliothèque/Ajouter la bibliothèque.ZIP…**, après avoir téléchargé les fichiers .zip de ces bibliothèques, disponibles sur Github.

```
}

void handleError(Error error, uint32_t token)
{
  …
}
```

Dans la fonction **setup**, il connecte l'ESP32 comme **station Wifi** au réseau local de la Livebox :

```
  WiFi.begin(SSID, PWD);
  while (WiFi.status() != WL_CONNECTED) delay(500);
```

puis démarre le client ModbusTCP en installant les fonctions de rappel précédentes :

```
  MB.onDataHandler(&handleData);
  MB.onErrorHandler(&handleError);
  MB.setTimeout(2000, 200); // Message timeout and interval between requests to the same host
  MB.begin();
```

et enfin lance la requête de lecture avec **READ_HOLD_REGISTER** comme code function :

```
  MB.setTarget(IPAddress(192, 168, 1,13), 502); // Modbus Server address and port
  uint8_t serverID = 1;
  uint8_t functionCode = READ_HOLD_REGISTER; // 0x03
  uint16_t startAddress = 40355; // => OutWRte (#40356) = percent of max decharge rate
  uint16_t nbRegisters = 1;  // Nb of registers
  Error err = MB.addRequest((uint32_t)millis(), serverID, functionCode, startAddress, nbRegisters);
```

Le reste est évident.

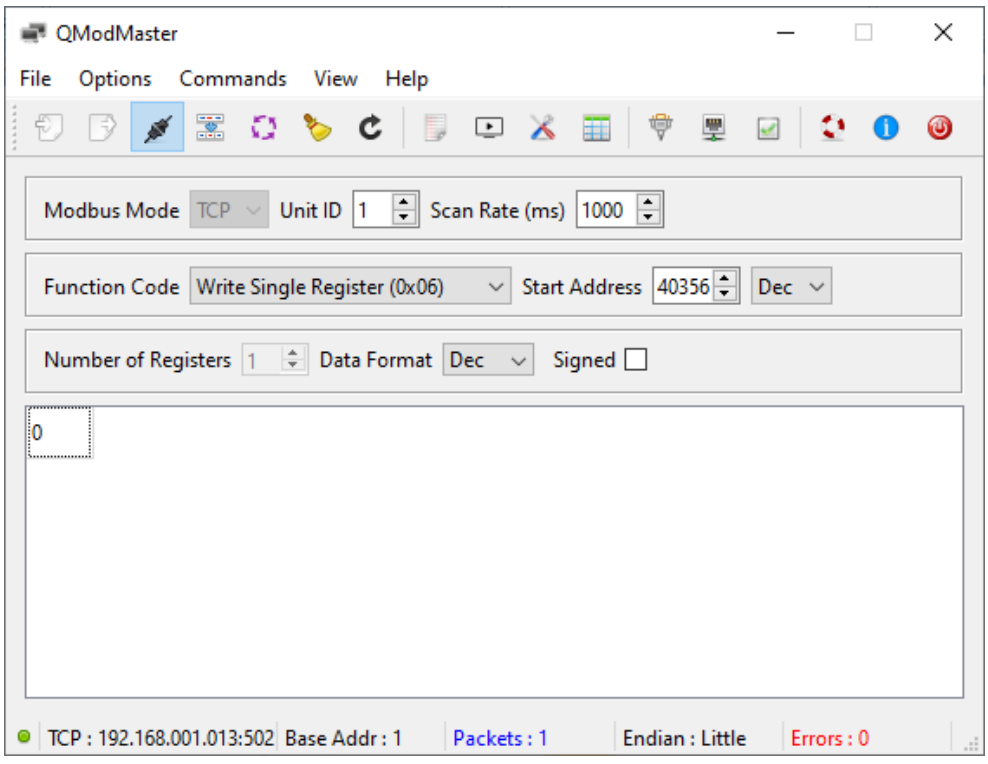
De manière analogue, on pourrait constater que le registre **StorCtl_Mod** contient la valeur **0**.

****

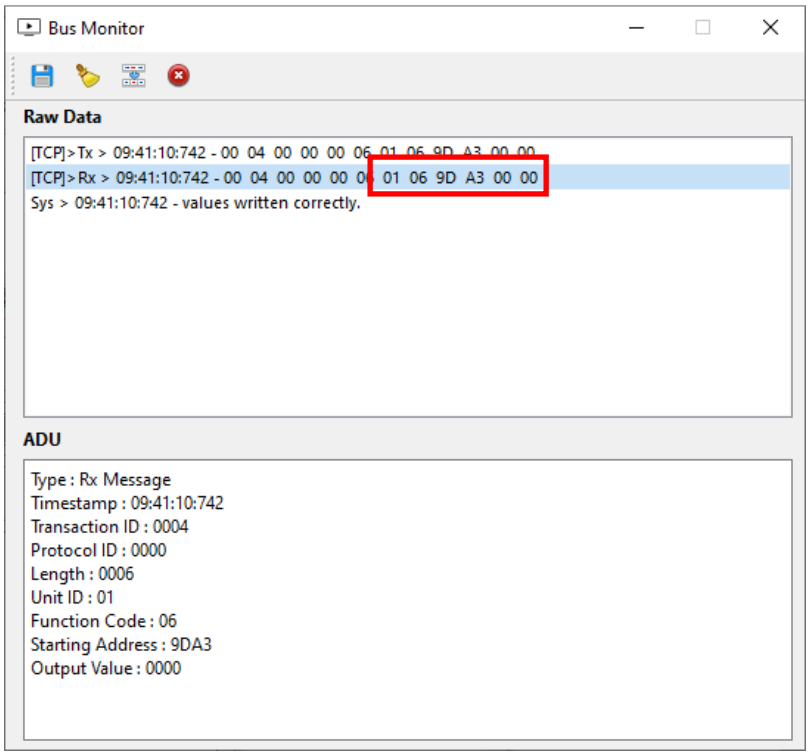
On va maintenant changer ces valeurs comme indiqué dans le document Fronius précité :

- 0 => OutWRte
- 2 => StorCtl_Mod

Là encore on peut le faire avec **QModMaster** , en utilisant cette fois le code fonction 0x06 :



Trame reçue :



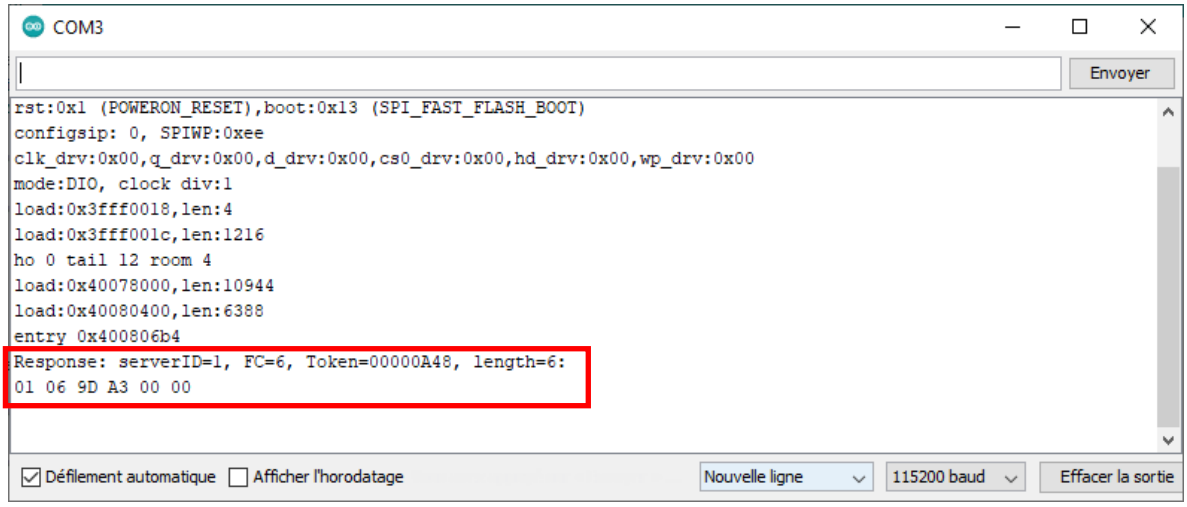
L'ID de l'unité + PDU est donc ici égal à 0x01 0x06 0x9D 0xA3 0x00 0x00 où :

- ID d'unité = 0x01 (1 octet)
- Code fonction = 0x06 (1 octet)
- Data :
    - Adresse du registre – 1 = 0x9DA3 = 40355 (2 octets)
    - Valeur = 0x0000 (2 octets)

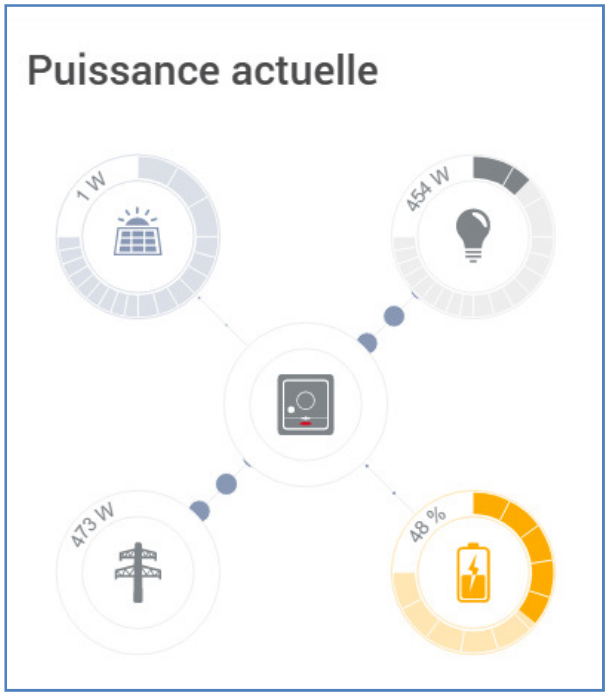
Avec l'ESP32, pour réaliser cette écriture, il suffit juste de changer les instructions suivantes dans la requête du précédent programme :

```
uint8_t functionCode = WRITE_HOLD_REGISTER; // 0x06
uint16_t regAddress = 40355; // => OutWRte (#40356) = percent of max decharge rate
uint16_t value = 0; // 0%
Error err = MB.addRequest((uint32_t)millis(), serverID, functionCode, regAddress, value);
```

Valeurs affichées sur le moniteur série



Si on procède de même pour le registre **StorCtl_Mod** où on écrit la valeur 2, on peut observer que la décharge de la batterie a bien été interrompue et que la consommation est désormais couverte en soutirant le réseau de distribution :



Cas d'écritures multiples

Il suffira, là encore, de modifier uniquement les instructions de la requête ; par exemple :

```
uint8_t functionCode = WRITE_MULT_REGISTERS; // 0x10
uint16_t regAddress = 40232; // => WMaxLimPct (#40233) = power output limit (%)
uint16_t nbRegisters = 5; // Nb of registers
uint8_t nbBytes = 10; // 2xnbRegisters
uint16_t values[5] = {10000, 0, 0, 0, 0};
Error err = MB.addRequest((uint32_t)millis(), serverID, functionCode, regAddress, nbRegisters, nbBytes, values);
```

6 Conclusion

Si l'on fait l'effort de se plonger dans les arcannes des processus utilisés ici, notamment Modbus TCP qui n'est pas évident, on peut superviser et contrôler avec un logiciel externe un système photovoltaïque basé sur un onduleur Fronius GEN24, ceci grâce aux API proposées.

Les microcontrôleurs offrent ici un moyen très souple pour les exploiter et en particulier ceux basés sur des ESP32 dont le rapport qualité/prix est excellent, notamment grâce à leurs capacités mémoire importantes, leur rapidité et leurs fonctionnalités Wifi.

Les clés d'accès et les références utiles ont été communiquées : elles peuvent servir de point de départ pour créer des logiciels ergonomiques.

Evidemment, il faudra le faire avec prudence en ce qui concerne le contrôle !