Modbus TCP Server For Victron Network With CCGX Gateway

1. Disclaimer

This document describes a piece of software which I have written for my personal use within another application. It is not meant as a generic tool for Modbus applications or a considered a finished product. It is merely a set of classes which can be used to implement the communication with the Modbus via TCP usimg a definitions file. The software is by no means completed or robust in all aspects, however it can be a useful base for further development.

I do not take responsibility for wrong use or any damage resulting from the use of the software to any connected components. This software does not write to any fields in the Victron components.

The software comes without warranty and you can use as is or modify and redistribute as you wish, however I would like to have a link included to my website teutanic.com.

The sofware and this documentation is not meant as an end user product, it is puerely meant for enthusiasts with programming knowledge who want to get started a boit easier with Victron Modbus in Java than I did.

1. **Introduction**

The software is written in Java and uses and object orientated approach based on the obeservable/observer pattern in Java.

One class controlls all access to the Modbus to avoid conflicts.

The available devices are hardcoded in the ModbusDefinitions class and probably do not need to be changed unless you install additional devices.

*uidList.add( new ModbusUid("MPPT", 0, 0)); // Ve.can on CCGX starts with device id 0  
uidList.add( new ModbusUid("MPPT", 1, 1)); // index 1 is the second MPPT controller at aId 1  
uidList.add( new ModbusUid("BMV", 0, 245)); // the BMV on Ve-direct 1  
uidList.add( new ModbusUid("BMV", 1, 247)); // the BMV on Ve-direct 2  
uidList.add( new ModbusUid("Multi",0, 246)); // the pair of multis appears as one*

The IP address of the CCGX is also hardcoded in ModbusInterface. This needs to be changed for other installations. I use fixed ip’s in my network.

The port for the TCP connection is 502, the standard port for Modbus.

The software does not support Strings as a result type of a read, only signed and unsigned numbers.

1. **Definition File**

The software reads a definition file, which contains information for each Modbus field to be read. The file must be in a comma delimeted textfile called **fieldlist.csv**.

Example:

Multi,0,c,Input voltage phase 1,3,uint16,10,0,1  
Multi,0,c,Input current phase 1,6,int16,10,0,5  
Multi,0,c,Input frequency 1,9,int16,100,0,0.01  
Multi,0,c,Input power 1,12,int16,0.1,0,5  
Multi,0,c,Output voltage phase 1,15,int16,10,0,0.2  
Multi,0,c,Output current phase 1,18,int16,10,0,5  
Multi,0,c,Output frequency,21,int16,100,0,0.01  
Multi,0,c,Active input current limit,22,int16,10,1,0.1  
Multi,0,c,Output power 1,23,int16,0.1,0,5  
Multi,0,c,Battery voltage,26,uint16,100,0,0.5

Colum description:

String fieldType = fields[0]; // the device type - MPPT, BMV, MULTI in my installation  
int deviceIndex = Integer.parseInt(fields[1]); // the device index if more than one device is connected of the same type  
String sampleType = fields[2]; // c = continuous (change notification with each sample,  
 d = discrete (change notification only when changed)   
String registerName = fields[3]; // the meaning of the field as defined by Victron  
int registerNumber = Integer.parseInt(fields[4]); // the fieldnumber as per xcel file from Victron   
String valueType = fields[5]; // uint16 and int16 are supported, no string yet  
float scaleFactor = Float.parseFloat(fields[6]); // the scale factor of the result as per xcel field list  
boolean isWritable = Boolean.parseBoolean(fields[7]); // flag if you can write the field as per xcel filefloat sampleFactor = Float.parseFloat(fields[8]); // a factor based on timeBase for the sampling of the register

See class Modbus Definitions.

If there is more than one of the same devices (BMV or MPPT) connected the second one gets the index 1 in the second field and all fields are repeated.

1. **The application**

The application comprises 7 classes.

* 1. **The Modbus class**

This is only for test and represents the main entry point for the program. You would replace this with your own. It only shows how to instantiate the controller class

new ModbusController(); // this activates all classes for the modbus server  
TestWindow window = new TestWindow(); // this is the application part  
window.setVisible(true); // show the little test input window

Instantiating the ModbusController class is all that is needed to create the server. The TestWindow is created and made visible.

* 1. **Testwindow Class**



The class creates the window as pictured above and also demonstrates how to implement an observer for a specific value.

The window allows to enter a field number and a device index. The device index is only used when more than one BMV or Bluesolar MPPT would be connected. My parallel Multis appear as one device.

Since Victron defined all fields of all devices without overlap the fieldnumber alone identifies the device and the server identifies the device by index and maps the slaveId (unitId) of the device on the Modbus. See above.

Once the sampling is started the window displays the field name, the value received, the ms between samples and the sample count to monitor progress. Only one field is meant to be sampled at any one time. The process can be stopped using the button or by entering another field number and pressing READ. The previous field sampler will be stopped.

If the field is not known, the message :unknown: is displayed in the field name.

* 1. **Creating a sampling entry**

The example of how to start a sampling process for a specific field can be found in the TestWindow class.

public void readField(int register,int index) {  
 ModbusRegisterObject anObject = ModbusController.mbController.findRegisterObject(register,index);  
 stop(); // stop any current observers  
 if (anObject != null) {  
 anObject.addObserver(this);  
 }else {  
 valueDisplayField.setText("unknown");  
 }  
}

The first red statement defines a sampling task with the register number and index read from the input field. The second red statement lists the TestWindow object as an observer to the created sample object for all change notifications. This is how the link is initiated.

@Override  
public void update(Observable observable, Object arg) {

This method establishes the observer and gets called everytime the sampling object dispatches a notification. Here the fields are updated or calculations can be made. This is all that is needed to read a specific value and get notified when it changes (for discrete sampling type=d) or when the sampling interval is over (for continuous sampling type=c).

* 1. **ModbusController class**

The ModbusController is the central mechanism to organise access to the Modbus and the CCGX. The field definitions are read from a .csv file and the reader object is instantiated. The controller is the schedulerfor all modbus access and has a task list which is sequentially worked through for all the field requests. The controller has a timer handling the highest sampling rate and all other fields will be handled depending on their individual sampling rate.

Some call the task list also a stack and the type would be a fifo - first in first out.

* 1. **ModbusDefinitions class**

As mentioned above the ModbusDefinitions class contains the device list and reads the csv file for all fields to be processed. It is used to look up a specific field object. The inventory of all field objects is kept here.

* 1. **ModbusInterface class**

Contains the code to actually read the data from the Modbus. It uses the library j2mod.jar which can be downloaded from the internet.

* 1. **ModbusRegisterObject class**

This class holds the information for one field and is the actual observable object. It sends notifications for change to all observers. It is also responsible to identify if the value field needs another sample depending on the samplingFactor. A sampling factor of 10 would create a sample every 100 milliseconds.

The occurences of this classes is intantiated by the ModbusDefinitions class when reading the csv file.

* 1. **ModbusUid class**

This is a helper class holding data about each device. Defined by hardcoded statements in ModbusDefinitions.

For more information please refer to the program sources.