Great! I'll prepare a complete learning guide organized by topic, covering all essential skills you need for your solar plant integration project. This guide will include written explanations, diagrams, relevant videos, and hands-on exercises—especially focusing on Windows-based tools and Python simulations that mimic the SmartLogger Modbus data flow.

I'll get started and share your step-by-step learning tutorial shortly.

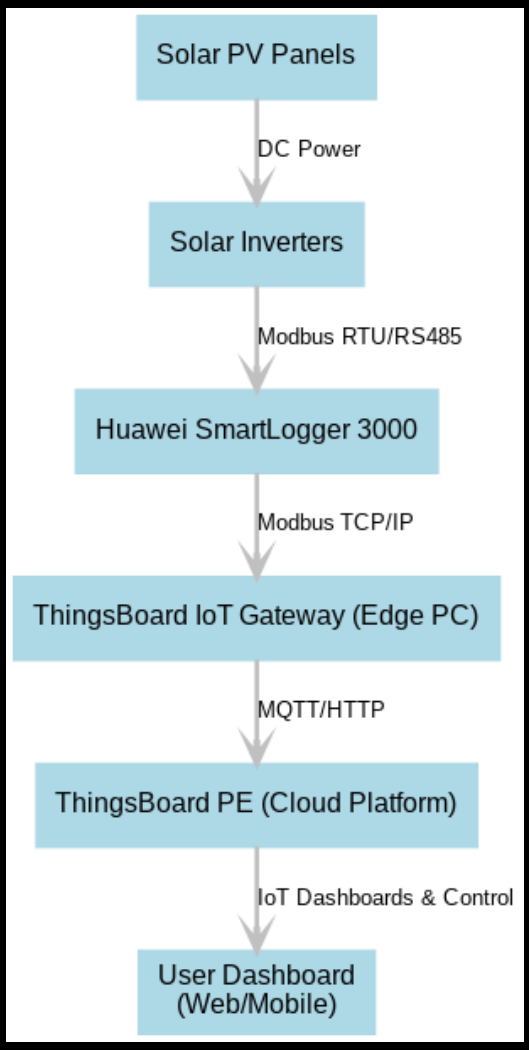
Integrating Huawei SmartLogger 3000 Solar Plants with ThingsBoard IoT Gateway – A Beginner's Guide

Introduction to IoT and Industrial Automation

Learning Objectives: Understand what the Internet of Things (IoT) means in an industrial context, and how edge gateways bridge legacy industrial systems (like SCADA/PLC) with modern IoT platforms. Learn why IoT integration is valuable for solar plant monitoring.

IoT refers to connecting physical devices (sensors, machines, etc.) to the internet for data collection and control. In industrial automation, many devices (PLCs, data loggers, sensors) communicate using local protocols and are not internet-connected. An **IoT gateway** is a bridge that links these on-site devices to cloud platforms. For example, an **edge gateway PC** can collect data from a plant's SCADA or controllers using protocols like Modbus, OPC-UA, etc., and then upload that data securely to a cloud IoT platform (Connectivity diagram | ThingsBoard Community Edition). This allows real-time monitoring and control from anywhere, without disturbing the local control systems. Industrial IoT integration offers unified visibility over multiple sites, remote analytics, and centralized control, which is especially useful for managing a fleet of solar plants.

In our case, Huawei's SmartLogger 3000 acts as a data concentrator at each solar site, collecting metrics from inverters. We will use an IoT gateway (ThingsBoard Gateway) to fetch data from the SmartLogger and send it to **ThingsBoard PE** (a cloud IoT platform) for visualization and analysis. The diagram below shows the high-level architecture of this integration, from solar panels all the way to the user's dashboard.



(image) *Architecture of a Huawei SmartLogger-based solar plant IoT integration: Solar inverters send data to the SmartLogger (via RS485/Modbus RTU). An on-site ThingsBoard IoT Gateway polls the SmartLogger over Modbus TCP and sends the data to the ThingsBoard cloud platform (e.g., via MQTT). Users can monitor real-time telemetry and send control commands (RPC) from the ThingsBoard web dashboard.*

Practical Task: Make a list of components in a typical industrial IoT stack for a solar plant. Identify the *edge devices* (inverters, logger, gateway) and the *cloud platform* (ThingsBoard). This will clarify the roles of each component as you proceed.

Basics of Solar Power Plants and Inverters

Learning Objectives: Learn the key components of a solar power plant, what data inverters produce, and the role of Huawei SmartLogger 3000 in monitoring. Gain a basic understanding of what metrics and controls are relevant in a solar PV system.

A solar power plant consists of photovoltaic panels that convert sunlight to DC electricity, inverters that convert DC to AC and feed the grid, and often a monitoring system. **Solar inverters** are critical devices – they report power output, voltages, currents, energy produced, temperature, fault alarms,

etc. In multi-inverter sites, a data logger like the Huawei *SmartLogger 3000* aggregates data from all inverters. The SmartLogger 3000 connects to inverters typically over an RS485 network (using Modbus RTU) and can poll each inverter for its data. It then makes this consolidated data available to external systems (e.g., SCADA or IoT platforms) via a standard interface. Huawei's SmartLogger supports **Modbus TCP** output, meaning an external client can connect to it over Ethernet (TCP/IP) and retrieve registers for all the inverters. In effect, the SmartLogger is a Modbus **server** (slave) that represents multiple devices. Each inverter may be mapped as a different unit ID or register range on the SmartLogger's Modbus interface. For example, one user reported having 4 inverters daisy-chained to a SmartLogger and successfully reading each inverter's data via Modbus TCP port 502 on the logger (Smartlogger integration · wlcrs huawei_solar · Discussion #735 · GitHub).

Common data points from solar inverters include: AC output power (kW), DC input voltage/current, AC voltage, frequency, daily and total energy production (kWh), inverter status, and fault codes. These will correspond to certain Modbus register addresses defined by Huawei's documentation (often provided as a register map for the SUN2000 inverters/SmartLogger). Understanding these metrics helps in selecting which data to monitor on ThingsBoard. Additionally, some registers allow control commands – for instance, setting active power limiting, or turning the inverter ON/OFF remotely. We will simulate a subset of these in our learning exercises.

Practical Task: Identify at least 5 key parameters you would want to monitor from a solar inverter (e.g., power output, energy, etc.), and 1-2 commands you might want to send (e.g., reset an alarm or limit output). This will guide you when configuring Modbus data and RPC commands later in the guide.

Understanding the Modbus Protocol (Modbus TCP)

Learning Objectives: Grasp the fundamentals of Modbus communication – master/slave (client/server) concept, register types, and specifically how Modbus TCP works. Understand how data is encoded as coils and registers, which will be crucial for mapping inverter data.

Modbus is a simple but robust communication protocol widely used in industry. It operates on a master-slave model (also called client-server in modern terms). The master device (e.g. an IoT gateway or SCADA system) initiates every transaction by sending a request, and a slave device (e.g. an inverter or logger) replies with a response. Communication is always in request-response pairs (What is the Modbus Protocol & How Does It Work? - NI). In a Modbus network, each slave device is addressed by an ID (unit ID). Originally, Modbus was designed for serial links (Modbus RTU over RS485), but **Modbus TCP** encapsulates Modbus messages in TCP/IP packets so they can travel over Ethernet networks. In Modbus TCP, the concept of master/slave is preserved: a *Modbus TCP client* (master) connects to a *Modbus TCP server* (slave) at a given IP address and port (502 is standard) to exchange data.

Modbus uses a simple **memory model**: devices have tables of coils (booleans) and registers (16-bit values). There are typically four data types: discrete inputs and coils (1-bit values, for reading and writing digital states) and input registers and holding registers (16-bit numeric values, for reading sensor data or writing setpoints). Each data item has a numerical address. For example, an inverter might expose the AC output power in holding register 300 (just as an example). The master can send a command "Read Holding Registers starting at address 300, quantity 2" – and the slave will respond with the values of those registers. Because 16-bit registers often need to represent larger numbers or

signed values, protocols like Huawei's may use two registers for a 32-bit value or define scaling factors (this is something you would check in Huawei's Modbus interface documentation).

Under Modbus TCP, the format of messages is similar to serial Modbus, but with an added header. The good news is we rarely have to construct these low-level packets ourselves – we can use libraries (in Python or in the ThingsBoard Gateway) that handle the Modbus protocol. Key points to remember:

- **Master/Client** = initiator of communication (in our case, ThingsBoard Gateway or a Python script).
- **Slave/Server** = device responding (in our case, the SmartLogger or our simulator).
- **Function Codes**: e.g. 3 = read holding registers, 4 = read input registers, 6 = write single register, 16 = write multiple registers, 1/2 = read coils/inputs, 5 = write single coil, etc.
- **Registers and Coils**: Data is identified by address. We must use the correct function code for the type of register/coil and address range as defined by the device. An "Illegal data value" or similar error often means the address or quantity requested is not valid for that device (for example, asking for registers that don't exist) (python 3.x - How to connect to Hawaii SmartLogger 3000 using Modbus TCP - Stack Overflow).

For Huawei SmartLogger and inverters, they provide a register map (often called *Modbus interface definitions*). This defines which addresses hold which data (e.g., Active Power might be at address 32089 as a 2-register 32-bit value – this is just illustrative). We will not dive into the exact Huawei register map here, but be aware that such documentation is your reference when configuring real devices. In our simulation exercises, we'll choose some arbitrary addresses for practice.

Practical Task: Draw a simple request-response sequence on paper for a Modbus interaction. For example: "Master requests read of holding register 100 (1 register) → Slave responds with value 12345." Label the master and slave. This mental model will help you when writing code or configuring the gateway, as you'll know who is sending and who is responding. You can also try using a free Modbus client tool (like **modpoll** or a trial of **Modbus Poll**) on your Windows PC to query a device if you have one – or wait until we set up our Python simulator and then test against it.

Introduction to Python for Modbus Simulation

Learning Objectives: Set up a Python development environment on Windows. Learn about Python libraries for Modbus (such as **pymodbus** and **pyModbusTCP**). Understand how Python can act as a Modbus simulator – both as a server (to mimic an inverter/SmartLogger) and as a client (to test reading/writing data).

Python is an excellent tool for simulating devices and prototyping integrations because of its simplicity and rich library support. We will use Python to simulate a Modbus-enabled solar inverter or SmartLogger. The simulation will run on your Windows PC, pretending to be the device, so that we can develop and test the integration without needing physical hardware. We'll also use Python to write a

simple client script to verify our simulator and understand Modbus read/write operations. This will build your confidence before configuring the ThingsBoard Gateway.

Setting up Python: If you haven't already, install the latest Python 3.x on your Windows machine (from python.org or via the Microsoft Store). During installation, check "Add Python to PATH" for convenience. You can use **VS Code** or even a simple text editor to write your scripts, and run them from the **Command Prompt/PowerShell**. We will also use `pip` (Python's package installer) to install the Modbus libraries.

Key Python libraries for Modbus:

- **pymodbus** – A popular library that supports Modbus TCP and RTU, for client and server. It's a bit heavier but widely used and well-documented.

- **pyModbusTCP** – A lightweight library specifically for Modbus TCP client/server. Its usage is very straightforward for TCP only.

In this guide, we will primarily demonstrate using **pymodbus** (since it's mentioned and it can cover both client and server needs). However, you can use either. For example, a video tutorial demonstrates how to create a Modbus TCP server easily with pyModbusTCP and connect a client to it (pyModbusTCP - the easy way to a Modbus TCP server with Python - смотреть видео онлайн от «Учимся работать с JavaScript API» в хорошем качестве, опубликованное 29 ноября 2023 года в 2:48:17.) – the concepts carry over to pymodbus as well.

Before coding, ensure your Python environment is ready: open Command Prompt and do: `pip install pymodbus` (this will also install dependencies like `twisted` if needed). For pyModbusTCP, you would `pip install pyModbusTCP` (optional, not required if using pymodbus). Also, it's helpful to have basic familiarity with running Python scripts (e.g., `python myscript.py`). Don't worry if you're new – we will provide sample code and explain it.

Practical Task: Verify your Python setup. Open a PowerShell or CMD window, and type `python --version` to ensure Python is installed. Then try `pip install pymodbus` to install the library. Finally, in a text editor, create a simple "hello world" Python file and run it to ensure everything is working. If you're using VS Code, you can run the script by pressing F5 (after selecting Python interpreter). This basic setup check ensures you're ready to run the Modbus simulation code coming up.

Huawei SmartLogger 3000 Modbus Communication

Learning Objectives: Understand how the SmartLogger 3000 interfaces with both inverters and external systems. Learn how to enable and configure Modbus TCP on the SmartLogger, and what kind of data it provides. Recognize how multiple inverters are represented through one SmartLogger.

The **Huawei SmartLogger 3000** is essentially a communications gateway for Huawei solar equipment. It connects to inverters (typically via an RS485 *MBUS* link using Modbus RTU protocol) and gathers their data. It can manage dozens of inverters on one RS485 bus. The SmartLogger itself

can then act as a **Modbus TCP server** on the network, allowing external *clients* to query inverter data through it. This design means you (or an IoT Gateway) don't poll each inverter individually over RS485 – you just poll the SmartLogger, and it relays the requests to the correct inverter internally. In Modbus terms, the SmartLogger is a **gateway** that converts Modbus TCP requests into Modbus RTU for the inverter chain. Each inverter is typically addressed by a unit ID (slave ID) configured in the SmartLogger. For instance, if inverter #1 is ID 1, inverter #2 is ID 2, etc., you would query the SmartLogger at unit ID 1 to get inverter 1's data, and so on. (Huawei documentation confirms that the SmartLogger supports this mode; e.g., users have pulled data for multiple daisy-chained inverters via the SmartLogger on port 502 (Smartlogger integration · wlcrs huawei_solar · Discussion #735 · GitHub).)

Enabling Modbus TCP on SmartLogger: By default, the SmartLogger might have Modbus TCP disabled for security. Typically, you would log in to the SmartLogger's web interface and find settings for "Modbus TCP". According to Huawei's guides, you need to enable Modbus, specify a port (502 usually), and possibly set an access mode (sometimes you can restrict which client IPs can connect). One Huawei video tutorial walks through *"Connecting a Third-Party Monitoring System to the SmartLogger3000 over Modbus TCP"* (14 Connecting a Third Party Monitoring System to the ... - YouTube) – this shows the steps such as enabling Modbus and configuring the connection. Ensure this is done before trying to connect ThingsBoard Gateway to a real SmartLogger. (For our simulation, of course, the SmartLogger is not involved.)

Data available: The SmartLogger provides both real-time telemetry and some configuration/status registers. Huawei's **Modbus register map** for the SUN2000 inverters and SmartLogger is the ultimate reference for exact addresses and scaling. Some examples of data points you can expect:

- Per inverter AC output (active power), DC input voltage/current, AC voltage, frequency.
- Energy counters (daily, monthly, total energy).
- Inverter temperature, state (running, stopped, fault), and fault codes if any.
- SmartLogger itself might have some overall status or aggregated values.

Typically, Huawei uses a register addressing where holding registers (function code 3 or 4 to read) are used for most telemetry. Many values are 32-bit and span two consecutive registers. For instance, an example from a Huawei register list might say "Active Power = register 32089 (2 registers, int32)". The SmartLogger's documentation (often a PDF titled *"SmartLogger Modbus Interface Definitions"*) will list all these. Don't be overwhelmed – you don't need all registers, just pick the ones you want. In our ThingsBoard integration, we will configure specific registers to poll.

Remote control via SmartLogger: The SmartLogger also allows writing to certain registers to control the inverters (if permitted). For example, you could send an RPC to curtail (limit) the inverter output by writing to a power limit register, or clear alarms. This requires knowing the exact register and allowed values, and ensuring the SmartLogger is in a mode that allows external control (often, the inverter system might need to be set to external control mode). For learning, we'll simulate a simple control (like toggling a value) rather than a real inverter command. Just be aware that *writing* to actual equipment should be done carefully and only with correct registers – always consult the manual.

Practical Task: If you have access to a Huawei SmartLogger 3000 (perhaps in a lab environment), try logging into its web UI and locating the Modbus TCP settings. Practice enabling it on a non-production system. If you don't have one, instead download the **SmartLogger3000 User Manual** (SmartLogger 3000 – Help Centre General) from Huawei support and read the section on Modbus or third-party integration. This will give you a sense of how the real device is configured, which will help when you move from simulation to the actual deployment on your 13 sites.

Setting Up and Simulating a Modbus Server with Python (Windows)

Learning Objectives: Create a simulated Modbus TCP device (server) on your Windows machine using Python. Define some example registers (holding registers, coils, etc.) to mimic an inverter's data. Run this simulator and ensure it's accessible over TCP.

Now it's time for hands-on work. We will write a Python script to simulate a Modbus server (slave). This server will hold some registers that represent telemetry of a solar plant. For simplicity, we might simulate one "inverter" or the whole plant as one device. In a real scenario with multiple inverters via SmartLogger, you'd actually have one Modbus server (the SmartLogger) and multiple unit IDs. We can simulate a single unit for now (you can extend it to multiple unit IDs if desired by configuring the context accordingly in pymodbus).

Let's use **pymodbus** to create a TCP server. We will:

1. Define a data store with some initial values.

2. Start a Modbus TCP server on a specified port (we'll use 5020 in this example, to avoid requiring admin rights of port 502 on Windows – you can use 502 if running as admin).

Here's a simple example script (we'll call it `simulator.py`):

```
from pymodbus.server.sync import StartTcpServer

from pymodbus.device import ModbusDeviceIdentification

from pymodbus.datastore import ModbusSequentialDataBlock, ModbusSlaveContext, ModbusServerContext



# 1. Define data blocks for coils and registers (here we use only holding registers for simplicity)

holding_regs = ModbusSequentialDataBlock(0, [0]*100)  # 100 holding registers initialized to 0

coil_regs = ModbusSequentialDataBlock(0, [0]*10)      # 10 coils (booleans) initialized to 0
```

```
store = ModbusSlaveContext(
    di = ModbusSequentialDataBlock(0, [0]*10),   # discrete inputs (not used in this sim)
    co = coil_regs,
    hr = holding_regs,
    ir = ModbusSequentialDataBlock(0, [0]*100)   # input registers (not used actively here)
)
context = ModbusServerContext(slaves={1: store}, single=True)

# (Optional) Identify as a Huawei device (not strictly needed for function)
identity = ModbusDeviceIdentification()
identity.VendorName  = 'Huawei'
identity.ProductName = 'SmartLogger3000 Simulator'
identity.MajorMinorRevision = '1.0'

print("Starting Modbus simulator on port 5020...")
StartTcpServer(context, identity=identity, address=("0.0.0.0", 5020))
```

A few notes on this code: we created a block of 100 holding registers (`hr`) and 10 coils (`co`). Initially they are all 0. The `single=True` means the same context is used for any unit ID (we still defined it as unit 1 in the dict). This means our server will respond regardless of unit ID (or specifically it's unit 1). This is fine for a single-device simulation. If you wanted to simulate multiple inverters as different unit IDs, you could pass a dict with multiple slave contexts.

Run this script with `python simulator.py`. It will block and run until you stop it (Ctrl+C). Now you have a Modbus TCP server listening on port 5020 of your machine. It's currently not populating any dynamic values – we'll later add some logic to update registers (e.g., simulate the power changing over time). But even with static values, we can connect and read them.

At this point, nothing visible happens in terms of output except the "Starting Modbus simulator…" message. The server is now waiting for clients to request data. Let's test it with a Python client next. (If you get an error starting the server, ensure no firewall is blocking it and that you're using a free port. You might need to allow Python through Windows Firewall on first run).

Practical Task: Copy the above code into a file and run the simulator. Then, open another terminal and use the `netstat -an` command to verify that your system is listening on port 5020. You should see an entry like `0.0.0.0:5020` in listening state. This confirms your Modbus server is running. If you have Modbus client software (like Modbus Poll or QModMaster), try connecting to `127.0.0.1:5020` and reading holding registers to see if you get data (it should return 0s by default). If you don't have a GUI tool, proceed to the next section where we'll use Python as the client.

Reading and Writing Modbus Data with Python (pymodbus)

Learning Objectives: Use Python as a Modbus TCP client to read from and write to the simulated Modbus server. Understand how to access coil and register values and verify that our simulation is working. This will also mirror what the ThingsBoard Gateway will do internally.

With our Modbus simulator running (in one terminal or background), let's create a Python script to act as a Modbus **client** (master) that connects to it. We'll use `pymodbus.client.sync.ModbusTcpClient` for this. The goal is to read some registers, change some values, and observe the effect. This helps us confirm that the simulation and our understanding are correct.

Example `test_client.py`:

```
from pymodbus.client.sync import ModbusTcpClient



client = ModbusTcpClient('127.0.0.1', port=5020)

client.connect()



# Read holding registers 0-5 from unit 1

result = client.read_holding_registers(address=0, count=6, unit=1)

if not result.isError():

    print("Holding Registers 0-5:", result.registers)

else:

    print("Error reading holding registers")



# Write a value to holding register 0 (e.g., set it to 12345)

write_res = client.write_register(address=0, value=12345, unit=1)
```

```
if write_res.isError():
    print("Error writing register")
else:
    # Read back register 0 to confirm write
    result = client.read_holding_registers(address=0, count=1, unit=1)
    print("New value of register 0:", result.registers[0])

# Toggle a coil at address 0
client.write_coil(address=0, value=True, unit=1)
res = client.read_coils(address=0, count=1, unit=1)
print("Coil 0 state:", res.bits[0])

client.close()
```

In this script, we connect to our local Modbus server at 127.0.0.1:5020. We then:

- Read 6 holding registers starting from 0. Initially, these should all be 0 (unless you changed them).
- Write the value 12345 to holding register 0.
- Read register 0 again to confirm it now contains 12345.
- Write to coil 0 (set it True/on) and read it back to confirm it's `True` (1).

Run `test_client.py`. You should see output confirming the initial registers (likely all zeros), and then the updated register 0 and coil state. This means our simulator is interactive – writes from a client actually change the server's state. (Our simulator, as coded, automatically writes to the data block for that register when a write request comes in – pymodbus takes care of that since we used a ModbusSequentialDataBlock which is mutable.)

This small test demonstrates the full Modbus round-trip on your PC: our Python client (master) talked to our Python server (slave) over the TCP socket using Modbus protocol. It proves that the server is functioning and that we can get/set values.

Now, how does this relate to Huawei SmartLogger and ThingsBoard? In a real deployment:

- The **ThingsBoard Gateway** will act like our Python client – it will periodically read holding/input registers (and sometimes write coils/registers for RPC) from the SmartLogger (which is the server).

- The **SmartLogger** will play the role of our Python server, providing data and accepting any writes.

So far, we've manually done what the gateway will automate. Understanding this helps if something goes wrong – e.g., if ThingsBoard isn't getting data, you can break out a Python client or modbus tool to individually test connectivity and registers.

Feel free to extend the simulator: you can add logic to update certain registers every few seconds (for example, simulate a power output that changes over time). This can be done by running a background thread or an asynchronous loop to modify `holding_regs` values. For simplicity, we'll not delve deep into threading here – you might just manually run the client periodically for now or trust static values. Even static data is fine to verify integration.

Practical Task: Modify the simulator to simulate a realistic scenario: for instance, set holding register 1 to a "power output" value that you change manually before each client read. For example, you could open the Python REPL and use `context[1].setValues(3, 1, [5000])` (if using pymodbus context API) to set register 1 to 5000 (representing 5 kW). Alternatively, change the initialization of the data block to pre-load some non-zero values (e.g., put some dummy voltage, current, etc.). Then run the client again to read those values. This will make the telemetry in the next steps more interesting than all zeros.

Configuring ThingsBoard Gateway on Windows (Docker)

Learning Objectives: Set up the ThingsBoard IoT Gateway on a Windows environment using Docker. Understand the Gateway's role as a bridge to ThingsBoard PE and how it's configured in general. Prepare the environment to later add the Modbus connector configuration.

The **ThingsBoard IoT Gateway** is a software service that will run on our edge (in this case, your Windows machine, and later an industrial PC at the solar site). It connects to local devices (Modbus inverters/SmartLogger, in our scenario) and forwards data to the ThingsBoard server (which might be cloud-hosted or on-prem). The gateway itself can be run in multiple ways: directly on Python, as a Windows service, or inside a Docker container. We will use Docker on Windows for simplicity and consistency. Docker will containerize the gateway and its configuration.

Prerequisites: Install **Docker Desktop for Windows** if not already installed. Make sure it's running and that you have enough privileges (Docker usually requires admin or to be running in a user group). Also, ensure ThingsBoard PE instance is accessible (you should have the ThingsBoard server URL, and credentials for a device or gateway connection – we'll discuss that soon).

Getting the Gateway image: ThingsBoard provides a Docker image for the IoT Gateway. You can fetch it from Docker Hub. For example, open PowerShell and run:

```
docker pull thingsboard/tb-gateway:latest
```

This will download the latest gateway image. (If this doesn't work or if an image isn't available, an alternative is to run the gateway via Python. But assuming the official image exists as per ThingsBoard docs, the above is the easiest route.)

Configuration files: The gateway requires configuration files on startup, which we will prepare in the next section. Typically, there is a main config (`tb_gateway.yaml` or similar) and connector configs (like `modbus.json`). We will create these on our host and mount them into the container. The default config path in the container is usually `/etc/thingsboard-gateway/config`. We can map a local folder (say `C:\tb-gateway-config`) to that path.

Let's create a local directory for config, e.g., `C:\tb-gateway-config`. We will put our configuration files there once written. For now, create an empty `tb_gateway.yaml` and `modbus.json` (we will fill them soon). Also, the gateway will need credentials to connect to ThingsBoard. ThingsBoard uses a *Device* to represent the gateway, or you can use a special Gateway provisioning. The simplest method: create a Device in ThingsBoard named "Gateway" (for example) and mark it as "Is Gateway". Then get its access token from ThingsBoard. We will use that token for the gateway to authenticate with ThingsBoard's MQTT.

Running the container: Once config files are ready, you would run something like:

```
docker run -d --name tb-gateway --restart always ^
  -v C:\tb-gateway-config:/etc/thingsboard-gateway/config ^
  thingsboard/tb-gateway:latest
```

(The ^ is a line break in PowerShell. In CMD use ^ or put it all in one line without it.)

This runs the gateway in detached mode. You can check logs with `docker logs tb-gateway -f`. Initially, it might show some errors until we configure it properly. We will get back to this after preparing the config files.

Note: If you prefer not to use Docker, you could install the gateway via pip (`pip install thingsboard-gateway`) and run it as a normal app, but then you have to manage it with Python. Docker simplifies deployment and ensures it runs in the background. Since we target a Windows environment, Docker is a convenient choice.

Practical Task: Install Docker Desktop and pull the ThingsBoard Gateway image as described. Create a ThingsBoard Device for the gateway (in ThingsBoard PE UI: Devices -> Add Device, choose type

“Gateway” if needed, and note down the **Device Access Token** from the device details). Save this token, as we will put it into the `tb_gateway.yaml`. Finally, set up the folder for config on your disk. This preparation will save time when we move to actually running the gateway with proper configuration.

Creating a Modbus Connector Configuration (ThingsBoard Gateway)

Learning Objectives: Write the configuration files for the ThingsBoard Gateway, specifically enabling the Modbus connector. Map our simulated device (or actual SmartLogger) into the config by specifying IP, unit IDs, and registers to poll. Understand the format of `modbus.json` and how to define telemetry (timeseries) and attributes.

ThingsBoard Gateway uses a main config file (`tb_gateway.yaml`) to configure global settings and enable/disable connectors, and a separate config file for each protocol connector (in our case, Modbus). We will focus on the Modbus connector config, as that’s where most of our integration details go.

1. tb_gateway.yaml (main config): This YAML file includes general gateway settings (like ThingsBoard host, access token, etc.) and lists which connectors are active. A minimal example:

```
thingsboard:
  host: "demo.thingsboard.io"          # or your ThingsBoard PE server host
  port: 1883                           # default MQTT port
  remoteConfiguration: false
  security:
    accessToken: "YOUR_GATEWAY_DEVICE_TOKEN_HERE"

connectors:
  - name: Modbus Connector
    type: modbus
    configuration: modbus.json
```

Adjust the host/port if your ThingsBoard is not cloud or uses a specific port (1883 is for MQTT, which Gateway uses by default). Insert the access token from the device you created on ThingsBoard. The

above tells the gateway to load `modbus.json` for the Modbus connector. Make sure spacing is correct (YAML is space-sensitive). Save this in `C:\tb-gateway-config\tb_gateway.yaml`.

2. modbus.json (Modbus connector config): This is a JSON file where we specify how to connect to the Modbus server(s) and what data to fetch. We can configure multiple Modbus servers if needed (for example, multiple SmartLoggers at different IPs). In our simulation, we have one server (our Python simulator on 127.0.0.1:5020). For a real deployment, this would be the SmartLogger's IP (e.g., 192.168.x.y:502).

Let's create a basic modbus.json for our simulation first. For example:

```
{
  "server": {
    "name": "SolarPlant Modbus Server",
    "type": "tcp",
    "host": "127.0.0.1",
    "port": 5020,
    "timeout": 35,
    "method": "socket",
    "devices": [
      {
        "unitId": 1,
        "deviceName": "Solar_Inverter_1",
        "deviceType": "solar_inverter",
        "attributesPollPeriod": 60000,
        "timeseriesPollPeriod": 5000,
        "attributes": [
          {
            "tag": "serialNumber",
            "functionCode": 3,
            "address": 10,
```

```
    "registerCount": 2,
    "type": "string",
    "byteOrder": "BIG"
  }
],
"timeseries": [
  {
    "tag": "ac_power_output",
    "functionCode": 3,
    "address": 1,
    "registerCount": 1,
    "type": "uint16",
    "byteOrder": "BIG"
  },
  {
    "tag": "ac_voltage",
    "functionCode": 3,
    "address": 2,
    "registerCount": 1,
    "type": "uint16",
    "byteOrder": "BIG"
  },
  {
    "tag": "status_coil",
    "functionCode": 1,
    "address": 0,
```

```
        "registerCount": 1,
        "type": "boolean"
      }
    ],
    "rpc": [
      {
        "tag": "setCoil0",
        "type": "bit",
        "functionCode": 5,
        "address": 0
      }
    ]
  }
 ]
}
}
```

Let’s break down what this means (don’t worry, we’ll cite references and you can adjust for the real SmartLogger after testing with sim):

- **server**: We define one Modbus server connection. We gave it a name “SolarPlant Modbus Server”. Type is “tcp” (because SmartLogger uses Modbus TCP). Host is `127.0.0.1` and port `5020` for our sim – in a real scenario, this would be the SmartLogger’s IP and `502`. Timeout 35 seconds (if network is slow or device not responding within that, it times out). Method “socket” is fine for TCP.

- **devices**: This is an array of device configurations under that server. Each “device” corresponds to a device in ThingsBoard (and typically a unique Unit ID on Modbus). We have one device here: `unitId: 1`, `deviceName: "Solar_Inverter_1"`. This means the gateway will create a ThingsBoard device with name “Solar_Inverter_1” and treat Modbus Unit ID 1 as that device. (If you had multiple inverters, you’d add multiple objects with different unitIds and names, e.g., 2 -> "Solar_Inverter_2", etc.). `deviceType` can be any string – e.g., “solar_inverter” – this could be used in ThingsBoard for grouping or applying a specific device

profile, but it's optional.

- **attributesPollPeriod / timeseriesPollPeriod**: These define how often (in milliseconds) to poll for attributes and telemetry. In this config, we set telemetry (timeseries) to 5000 ms (5 seconds) – meaning every 5 seconds the gateway will read the defined timeseries registers. Attributes could be polled less often (here 60s) if at all. Often attributes are static info like serial number, model, etc. Telemetry is the dynamic data like power, voltage, etc. We'll poll the telemetry frequently.

- **attributes**: We defined one attribute "serialNumber" as an example. It uses function code 3 (holding register read), address 10, 2 registers, type string. (This is just illustrative – in a real SmartLogger, serial might be in some registers as ASCII codes). The gateway will read those once every 60s and update the device's attribute. This shows how to handle string or multi-register values.

- **timeseries**: We defined a few telemetry points: "ac_power_output" at address 1, "ac_voltage" at address 2, both 1 register 16-bit. We mark them as unsigned 16-bit (`uint16`). Our simulator, for instance, might have put some values in reg1 and reg2 (like power and voltage). We also included "status_coil" using function code 1 (read coils) at address 0, type boolean. That might represent a status bit (maybe whether the inverter is on/off). The gateway will call the appropriate Modbus function for each entry (FC1 for coils, FC3 for holding registers, FC4 for input registers if used). All these will be sent to ThingsBoard as telemetry data (time-series data) with the key names given by "tag".

- **rpc**: This section defines how incoming RPC commands from ThingsBoard should be handled. We put an example: any RPC with the tag "setCoil0" will result in a Modbus function 5 (force single coil) to address 0. In ThingsBoard, we will trigger this RPC, and the gateway will perform a coil write (turn our simulated coil 0 on/off). We set `type: "bit"` (this might be "bits" or "bit", depending on gateway version – the documentation shows an example (Modbus Connector Configuration | ThingsBoard IoT Gateway) using "bits"). The idea is that if we send an RPC call named "setCoil0" with a boolean payload, it will flip that coil on the Modbus device. We'll test this later.

The above JSON structure is based on ThingsBoard Gateway documentation and examples (How to connect Modbus device to ThingsBoard using the ThingsBoard IoT Gateway | ThingsBoard Community Edition) (How to connect Modbus device to ThingsBoard using the ThingsBoard IoT Gateway | ThingsBoard Community Edition). It tells the gateway everything it needs to know to interact with our device. Once this config is in place, the gateway will automatically connect to `127.0.0.1:5020`, and start reading address 1,2 from holding registers and address 0 from coils every 5 seconds. It will then forward those to ThingsBoard as telemetry of device "Solar_Inverter_1". The attribute "serialNumber" will be read every 60s and sent as a device attribute. The RPC mapping will listen for "setCoil0" commands.

Go ahead and save this JSON to `C:\tb-gateway-config\modbus.json`. Double-check JSON syntax (commas, braces). The example is one way to do it; you can adjust addresses and tags to match what you want (and what your simulator or actual device provides). For a real SmartLogger, you

would replace the host IP, and then list out the actual registers you want (which you'd determine from Huawei's documentation – e.g., real power might be at some high address like 32089, etc., so you'd put that address and registerCount=2 if 32-bit, etc.). You'd likely have multiple devices in the list if reading multiple inverters by unit ID, or you might treat the whole SmartLogger as one device that aggregates data – it depends on how you prefer to represent in ThingsBoard. Common approach: each inverter becomes a ThingsBoard device, using unit IDs, which the gateway can handle easily by listing them in the devices array.

3. Launching the Gateway with config: Now that `tb_gateway.yaml` and `modbus.json` are ready in our config folder, (re)run the Docker container. The container should load these configs on startup. Check the logs: `docker logs tb-gateway -f`. You should see something like "Modbus Connector started" and it may log connecting to the Modbus server. If all goes well and our simulator is running, it should start polling. If there are errors, logs will indicate (e.g., unable to connect, or timeouts if our sim isn't running, etc.).

Important: The gateway will automatically create the device "Solar_Inverter_1" on ThingsBoard if it doesn't exist (since we enabled device provisioning by listing it in config). It will use the deviceName we provided. In ThingsBoard PE, you'll then see a new device with that name appear, under the Gateway's tenant. The telemetry keys like "ac_power_output" should also appear under that device's latest telemetry.

Practical Task: Start the ThingsBoard Gateway container with the new configuration. Monitor the logs for any errors. On the ThingsBoard PE UI, navigate to **Devices**. You should see a device named "Solar_Inverter_1" (if it wasn't there already). Click it and go to the **Latest Telemetry** tab. You should start seeing keys like `ac_power_output`, `ac_voltage`, etc., updating roughly every 5 seconds with whatever values your simulator provides (likely 0 or any test values we set). Also check the **Attributes** tab for the attribute we set ("serialNumber") – it should have a value (empty or some bytes if our sim didn't have meaningful data at those registers). If you see the telemetry coming in, congratulations – you have integrated a simulated Modbus device with ThingsBoard via the IoT Gateway! This is the same process you'll use for the actual SmartLogger and inverters, just changing IP and addresses. If you don't see data, check the gateway logs for clues (e.g., modbus timeouts or exceptions, which could mean wrong addresses or the simulator not running).

Visualizing Telemetry Data in ThingsBoard

Learning Objectives: Use ThingsBoard PE's dashboard tools to visualize the incoming solar plant data. Create simple widgets (charts, gauges) to monitor telemetry like power and voltage over time. Understand how device data appears in ThingsBoard and how to build a dashboard for stakeholders.

Now that data is flowing into ThingsBoard from our Modbus integration, we can create a dashboard to visualize it, similar to a mini-SCADA view. ThingsBoard PE offers a powerful dashboard editor. We will create a new dashboard and add widgets for the metrics we configured: e.g., power output, voltage, and any others of interest. This will allow us to see real-time updates (every few seconds as sent) and historical trends.

Creating a Dashboard: In the ThingsBoard UI, go to **Dashboards** and click **Add New Dashboard**. Give it a name like “Solar Plant Simulation” (later you might create one per site or a unified view for all sites). Once created, click **Edit** to open the dashboard designer.

- Add a new **Dashboard State** if prompted (or you can just use the default state “default”). In edit mode, click the **Add Widget** button (usually a “+” icon).

- Choose a widget that fits the data. For a continuous numeric telemetry like power or voltage, a *timeseries graph* or *chart* is suitable. There are pre-built widgets like “Charts -> Line chart” or “Latest values -> digital gauge”. For example, to see power over time, pick a Line Chart.

- Once you pick a widget, you’ll configure its data source: select **Device** as the type, then pick **Solar_Inverter_1** as the entity, and then select the telemetry key (e.g., `ac_power_output`). You can add multiple keys to the same chart if you want (e.g., overlay power and voltage in one chart by adding two data sources).

- After adding, the chart should start plotting points as data comes in. You can adjust time window (ThingsBoard might default to last 1 hour, you can change it or zoom).

Add a few widgets: for instance, a gauge showing the current power output, a chart showing power vs time, maybe a simple card showing the latest voltage value. Layout the widgets on the dashboard canvas as you like. ThingsBoard’s flexibility means you can create a very informative dashboard for the solar plant: including multiple devices, comparison between them, alarm indicators, etc. For now, just practice with basic widgets to ensure data is visible.

Because this is a beginner’s guide, note that ThingsBoard has many features (alarms, rule engine, etc.) that can be leveraged later. Our focus is telemetry visualization and RPC control.

Understanding the data flow: At this point, the pipeline is: Simulator (Modbus server) → Gateway (Modbus client & MQTT publisher) → ThingsBoard (MQTT subscriber & database) → Dashboard (querying database for data). The latency is low (a few seconds). You are effectively monitoring a (simulated) solar device in real-time on a web dashboard, which is a core goal of IoT integration. As one source highlights, using ThingsBoard and an IoT Gateway allows real-time telemetry monitoring and remote control for industrial systems over Modbus (ThingsBoard SCADA: Controlling an Industrial System via Modbus ...), which is exactly what we see – the values on our dashboard updating live from a Modbus source.

Practical Task: Design a simple dashboard for your device. Add at least two widgets – for example: 1) a “Latest value” display showing `ac_power_output` (so you can see the number update every few seconds), and 2) a “Time-series line chart” for the same `ac_power_output`. If you have `ac_voltage` coming in, you can plot that as well or put it on a separate gauge. Save the dashboard. Now try this: go back to your simulator code and change the value in holding register 1 (if you can, modify the simulator’s data). For instance, if it was 0, try setting it to 50 (maybe representing 50 kW). The next time the gateway polls, the new value will be sent to ThingsBoard, and you should see the widget update (within 5 seconds or so). This confirms end-to-end that changing device data is immediately reflected on the dashboard.

Remote Command (RPC) from ThingsBoard to Devices

Learning Objectives: Utilize ThingsBoard's RPC feature to send a remote command to the Modbus device via the gateway. Learn how to configure a control widget (e.g., a switch or button) that triggers the RPC, and verify that it affects the device (simulator). This demonstrates remote control capability, such as turning an inverter on/off or adjusting a setting.

Telemetry monitoring is only half of the integration – the other half is **remote control**. In ThingsBoard, the gateway device (and its child devices) support **RPC (Remote Procedure Call)** mechanism. We configured the gateway's Modbus connector with an RPC mapping (`"setCoil0"` in our `modbus.json`). That means if ThingsBoard sends an RPC method named `"setCoil0"` to the device `Solar_Inverter_1`, the gateway will translate that into a Modbus coil write at address 0. We will test this by sending an RPC command from the ThingsBoard UI and observing our simulator's coil state change.

Sending RPC via UI: ThingsBoard PE allows sending one-off RPC commands from the device details page. Go to Devices -> Solar_Inverter_1 -> Details. There might be an **RPC** tab or a widget to send an RPC. In community edition, typically one uses a dashboard widget to send RPC. We can do the latter since it's more visual:

On your dashboard (in edit mode), add a **Switch Control** widget (found under Controls widgets). This widget can send a boolean RPC to a device. Configure it as follows: choose **Solar_Inverter_1** as the target device. In the widget settings, you'll specify the RPC method name – enter `setCoil0` (the tag we defined). Also set the mapping such that the switch ON corresponds to sending `true` and OFF sends `false`. The widget may have a field for the RPC parameters or value – since our coil RPC expects a boolean, the switch will inherently send true/false. Ensure the widget is bound properly to the device.

Exit edit mode. Now on the dashboard, you should see a toggle switch. When you toggle it, it will send the RPC. How do we know it worked? We need to check our simulator's coil value. Since our simulator doesn't have a built-in UI, we can rely on our previous test client or logs:

- Option 1: We still have the gateway polling the coil as a timeseries (`status_coil`). So every 5 seconds, the gateway reads coil 0 and updates ThingsBoard. We can simply watch the `status_coil` value on the dashboard (maybe add a LED indicator or just watch its boolean in the device telemetry). When you flip the switch, within a few seconds, the `status_coil` telemetry should reflect the new state (True/False). This confirms the coil was set.

- Option 2: You can also run the `test_client.py` again (or a snippet) to read coil 0 directly from the simulator after toggling to double-check it changed. But using the telemetry feedback is convenient here.

If everything is configured correctly, toggling the switch will cause the gateway to log something like "RPC request received: setCoil0 with params …" and perform the Modbus write (Modbus Connector Configuration | ThingsBoard IoT Gateway). The next telemetry poll will pick up the changed coil. You

have effectively implemented a remote control: imagine this coil was “Inverter ON/OFF command” – you just remotely turned the inverter on or off from the cloud.

Some points to consider for real deployment:

- Ensure the **device is set to allow remote control**. Many systems require enabling writing via external interfaces.

- The gateway’s RPC mapping can be extended. For instance, you could map an RPC to write a numeric value to a holding register (function code 6 or 16) for things like power limit. You would then call that RPC with a value from ThingsBoard. The config format would be similar, specifying functionCode 6 and the register address.

- Security: RPC calls require proper permissions on ThingsBoard (only authorized users or rule engine can invoke). In PE, you can also script or schedule RPC via rule engine if needed (for example, automatic resets).

Practical Task: Test the RPC end-to-end. Turn the dashboard switch “ON” – then check in the device’s Latest Telemetry if `status_coil` turned to `true` (it might appear as 1 or true). Then turn it “OFF” – see if it goes false. You might notice a slight delay (because the telemetry only updates on the next poll; you can reduce timeseriesPollPeriod for faster feedback). If this works, you’ve successfully achieved two-way communication: data *to* the cloud and commands *from* the cloud. As an extra step, try to incorporate a simple RPC to write a holding register: for example, add in modbus.json an RPC like:

{ "tag": "setPowerLimit", "type": "16uint", "functionCode": 16, "address": 5, "registerCount": 1 }

This would mean you intend to write a 16-bit value to holding register 5. Restart the gateway after editing config. Then, in ThingsBoard, use the **REST API** or a tool like **MQTT client** to send an RPC call (since the UI switch only sends boolean). You can use the ThingsBoard REST API to call RPC: a simplest way is using the device debug terminal (if enabled in PE). This is advanced, but it’s a good exercise to see how numeric RPCs would be sent. If not, conceptually understand that such a command would allow, for instance, setting a power limit percentage by writing a value to the device.

Putting It All Together: Simulation to Real Deployment

Learning Objectives: Summarize the end-to-end process and highlight the steps to transition from the simulated environment to real hardware across 13 solar plant sites. Emphasize best practices and further learning path for the user.

Congratulations! You have built a mini end-to-end IoT integration: from a device (simulated inverter/SmartLogger) through an IoT Gateway to a cloud platform with live dashboard and remote control. This comprehensive exercise has given you foundational skills in IoT, Modbus, Python, and

ThingsBoard. To deploy this across 13 physical solar sites with Huawei SmartLogger 3000 devices, consider the following key points as you transition from simulation to reality:

- **Deployment Architecture:** Each site will likely have its own IoT Gateway (an industrial PC or router that can run the ThingsBoard Gateway). The architecture for each site will mirror what you did locally: the gateway at site connects to the SmartLogger via LAN (Modbus TCP). All gateways send data to a central ThingsBoard server (cloud or HQ data center). Ensure each gateway has a unique access token and is registered as a gateway device in ThingsBoard. ThingsBoard PE can handle multiple gateways and devices – you might organize devices by site using asset groups or device groups in the platform.

- **Configuration Management:** The `modbus.json` for each gateway might differ slightly (e.g., some sites have 10 inverters, others 5 – you list the appropriate unit IDs and names). But you can maintain a template. The register addresses for inverter telemetry remain the same across sites (since all use Huawei inverters), so your main difference is just number of devices and IP addresses. Using the **same deviceType** (like "solar_inverter") for all helps apply a common device profile or visualization. You can also use ThingsBoard's **device group** features to aggregate data (like a summary of all site outputs).

- **Testing and Troubleshooting:** When connecting to actual SmartLogger, use tools like the Python client or modpoll to test connectivity and reading a couple registers (especially if on-site). This helps verify that Modbus TCP is enabled and working on the SmartLogger. If the gateway logs show illegal addresses, double-check you used the correct register addresses (from Huawei's map) – sometimes off-by-one addressing issues occur (some systems list registers starting at 1 vs 0). The community and support forums can be handy if you run into specific issues.

- **Security Considerations:** In production, ensure to secure the solution. The SmartLogger Modbus TCP can be protected by IP filtering – only allow the gateway's IP. The gateway communicates with ThingsBoard over MQTT; use TLS if possible or a VPN tunnel for remote sites. ThingsBoard PE offers device credentials and even X.509 cert-based auth if needed. Also, consider enabling the **remote logging** feature of the gateway to debug from the platform if a gateway is deployed far away (Modbus Connector Configuration | ThingsBoard IoT Gateway).

- **Scaling Dashboards:** For 13 sites, you might create a dashboard per site and a master dashboard that sums up all. ThingsBoard allows creating **Aliases** and **States** in dashboards to switch context between sites. For example, you click on a site name to view that site's inverters' data. Use the device attributes (like location or site name) to group them. Alarms can be set (e.g., if an inverter is down or output is low). This is beyond the scope here but is your next step in mastering ThingsBoard.

By following this guide, you went from IoT basics, through Modbus protocol and Python simulation, to configuring a real IoT gateway connector and building dashboards with RPC control. This comprehensive journey mirrors real-world projects. As a beginner, you've now practiced the workflow of integrating legacy industrial devices into a modern IoT platform – a valuable skill set.

Next Steps: To reinforce your learning, consider the following:

- Deploy the gateway on an actual Windows machine (or Linux) at one pilot site and integrate with that site's SmartLogger. Start with read-only (no RPC) and verify data on ThingsBoard.

- Explore more of ThingsBoard PE features like rule engine (for example, trigger an email if output drops to zero during daytime), or the Edge computing features if you plan to have local processing.

- Continue using Python for small tests or data injection. For example, you could simulate historical data or write scripts to bulk-check all registers from the SmartLogger for curiosity. The GitHub community (like the "huawei-solar-tools" library (Atrabilis/huawei-solar-tools - GitHub)) might have ready-made mappings for Huawei inverters which you could leverage instead of manual register mapping.

- Watch ThingsBoard's official video tutorials on SCADA integration – they have one where they simulate an industrial process via Modbus and integrate with ThingsBoard, which can provide additional context and tips (ThingsBoard SCADA: Controlling an Industrial System via Modbus ...) (ThingsBoard SCADA: Simulating & Controlling an Industrial System ...). These videos reinforce how to use the gateway in practice and may introduce advanced widgets or troubleshooting techniques.

With the fundamentals covered, you are well on your way to deploying a unified renewable plant monitoring solution. As noted in the project proposal you have (Unified Renewable Plant Monitoring & Control Solution (2).pdf) (Unified Renewable Plant Monitoring & Control Solution (2).pdf), this integration will **consolidate critical plant metrics in real-time** and enable *enhanced decision making* through remote monitoring and control. Good luck with your 13-site deployment, and enjoy the process of turning data into actionable intelligence on ThingsBoard!

Sources: Throughout this guide, references were made to official documentation, forums, and tutorials:

- ThingsBoard IoT Gateway & Modbus Connector docs (How to connect Modbus device to ThingsBoard using the ThingsBoard IoT Gateway | ThingsBoard Community Edition) (How to connect Modbus device to ThingsBoard using the ThingsBoard IoT Gateway | ThingsBoard Community Edition) (Modbus Connector Configuration | ThingsBoard IoT Gateway)

- Huawei SmartLogger integration notes (Smartlogger integration · wlcrs huawei_solar · Discussion #735 · GitHub) and user manuals

- Python Modbus libraries and community examples (pyModbusTCP - the easy way to a Modbus TCP server with Python - смотреть видео онлайн от «Учимся работать с JavaScript API» в хорошем качестве, опубликованное 29 ноября 2023 года в 2:48:17.)

- ThingsBoard community videos on Modbus integration and SCADA dashboards (ThingsBoard SCADA: Controlling an Industrial System via Modbus ...)

Great. I'll guide you step-by-step on how to connect the Huawei SmartLogger 3000 to ThingsBoard PE (cloud) using ThingsBoard Edge and Gateway installed on the same computer. I'll also include how Modbus TCP communication works, how to read and interpret registers, and how to simulate and test before going onsite.

I'll also try to find if there's a way to list available registers from the SmartLogger and whether control is possible via Modbus. I'll get back to you with a clear setup guide and explanation shortly.

Connecting Huawei SmartLogger 3000 to ThingsBoard PE Cloud (Edge + Gateway on One PC)

Step 1: Understanding Modbus TCP Communication over Ethernet

Modbus TCP is a client-server (master-slave) protocol layered on TCP/IP. The **ThingsBoard Gateway** will act as the Modbus **client (master)** and the **Huawei SmartLogger 3000** as the Modbus **server (slave)**. In this model, the client initiates all requests (queries) and the server only responds – the server does **not push data on its own** (Introduction to Modbus TCP/IP) (Introduction to Modbus TCP/IP). Communication occurs over Ethernet (TCP/IP) using the well-known Modbus port **502** (Introduction to Modbus TCP/IP). Data is organized into registers (holding registers, input registers, etc.), each identified by an address. The Gateway (client) will periodically poll specific register addresses on the SmartLogger (server) to retrieve data (e.g. voltage, current, power) or to send commands. This query-response cycle is how Modbus TCP transfers data over the network, encapsulated in TCP/IP packets but following the Modbus request/response format (Introduction to Modbus TCP/IP) (Introduction to Modbus TCP/IP).

Key points: *Modbus TCP uses a layered approach over Ethernet. The Gateway's application forms a Modbus request which gets wrapped in TCP/IP headers and sent to the SmartLogger. The SmartLogger unpacks the request and returns the requested data in a response packet (Introduction to Modbus TCP/IP). Because Modbus is* ***master-driven****, the ThingsBoard Gateway must continuously poll (ask for data) at a defined interval; the SmartLogger will not automatically send data without being polled.*

Step 2: Enabling Modbus TCP on the Huawei SmartLogger 3000

First, log in to the SmartLogger 3000's web interface to enable and configure Modbus TCP. By default, the SmartLogger's WAN port IP is usually **192.168.0.10** (you may need to set your PC's IP to the same subnet to access it) (How to use Modbus TCP – Help Centre General). Once connected to the Web UI (login as Advanced User, default password "Changeme" (Huawei SmartLogger 3000 | Eniris Documentation)), navigate to the **Communication Parameters** or **Modbus TCP** settings page. Perform the following configuration:

- **Enable Modbus TCP:** Set "**Link setting**" to **Enable**. You can choose **Enable (Unlimited)** to allow any client (up to 5 concurrent connections) or **Enable (Limited)** to restrict connections to specified IPs (Huawei SmartLogger 3000 | Eniris Documentation) (Set Active Power limitation via Modbus). For simplicity, you may use Unlimited during initial testing (this accepts any client IP) (Huawei SmartLogger 3000 | Eniris Documentation).

- **Client IP (if Limited):** If you choose Limited, enter the IP address of your ThingsBoard Gateway/Edge PC as **Client 1 IP Address** (Set Active Power limitation via Modbus). This tells the SmartLogger to accept Modbus requests only from that IP (you can set up to 5 allowed clients).

- **Address Mode:** Set "**Address mode**" to **Logical address** (sometimes labeled "Communication address" in some firmware) (Huawei SmartLogger 3000 | Eniris Documentation). This mode means the SmartLogger and each connected device use unique Modbus IDs (logical addresses) for identification.

- **SmartLogger Modbus Address (Unit ID):** Assign a **Modbus address** to the SmartLogger itself (e.g. 100, or any unused ID 0–247) (Set Active Power limitation via Modbus). This ID will be used by the Gateway when polling data that the SmartLogger provides. Ensure this address does **not conflict** with addresses of any inverters or meters connected to the SmartLogger (check the Device List in the SmartLogger to see what addresses are already in use) (Set Active Power limitation via Modbus).

- **Apply Settings:** Save or submit the configuration. The SmartLogger will start listening on port 502 for Modbus TCP connections. (Tip: Ensure no firewall in the SmartLogger is blocking port 502 and that you are connected via the **WAN port** for Modbus TCP, as the LAN port may be for local UI or other purposes.)

Troubleshooting: After enabling Modbus, if the Gateway cannot connect, verify that your PC and SmartLogger are on the same network segment and that you can ping the SmartLogger's IP. If using **Windows**, also check that the Windows Defender Firewall is not blocking outbound/inbound connections on port 502 – you may need to allow that port or disable the firewall for testing. Ensure you clicked "Submit" (or Save) on the SmartLogger UI and possibly reboot the SmartLogger if instructed by the UI.

Step 3: Setting the Target IP Address on SmartLogger (Edge + Gateway on Same PC)

If you configured **Enable (Limited)** in the previous step, you must specify the IP of the ThingsBoard Gateway/Edge machine so the SmartLogger accepts its Modbus requests. In this case, since ThingsBoard Edge and Gateway run on the **same Windows PC**, use that PC's local network IP as the **Client IP** in SmartLogger's Modbus settings (Set Active Power limitation via Modbus). For example, if your PC's IP is **192.168.0.15** on the network, enter `192.168.0.15` as *Client 1 IP Address* (as shown in Huawei's interface example) (Set Active Power limitation via Modbus).

If your PC obtains its IP via DHCP (from a router) and may change, either set a static IP on that PC or use **Enable (Unlimited)** mode to avoid connection issues. In Unlimited mode, the SmartLogger will accept connections from any IP (up to 5 clients) (Huawei SmartLogger 3000 | Eniris Documentation). If you use Unlimited, you can skip specifying the IP (the fields can be left as 0.0.0.0). Otherwise, double-check that the IP entered matches the PC's actual IP. You can find your PC's IP by running `ipconfig` in Command Prompt (for the network connected to SmartLogger).

Note: The **Gateway and Edge being on the same PC** share the same IP when communicating with the SmartLogger. The SmartLogger sees all requests coming from that one IP (the PC), regardless of Edge or Gateway internally. Thus, only one Client IP entry is needed for that PC. After setting this, the SmartLogger will "trust" and respond to Modbus queries from your machine.

Step 4: Understanding Data Flow – Polling vs. Automatic Push

It's important to know that **Huawei SmartLogger does not automatically send data** to ThingsBoard. Communication is **poll-based**. The ThingsBoard Gateway (Modbus client) must regularly send read requests to the SmartLogger for each data point. The SmartLogger, acting as Modbus server, will respond with the requested register values (Introduction to Modbus TCP/IP).

In other words, the SmartLogger **will not "push" telemetry on its own** over Modbus TCP. Instead, you configure the Gateway with a polling interval (e.g. every 5 seconds) for each register or set of registers. The Gateway then continuously issues Modbus **Read** commands (Function Code 3 or 4) to fetch current values. This polled data is then forwarded to ThingsBoard. The SmartLogger can handle multiple concurrent requests (up to 5 clients if unlimited) and will update the values based on its internal collection from inverters or meters, but it only sends out those values when asked.

For completeness, note that some devices/systems have proprietary push protocols, but in the case of Modbus TCP on SmartLogger, **polling is the mechanism**. Ensure your polling frequency is reasonable (e.g. 5-15 seconds) to get near-real-time updates without overloading the network or device. Extremely rapid polling (e.g. every 1 second for many registers) could overwhelm the SmartLogger or the network. Start with a moderate interval and adjust as needed.

Step 5: Installing and Setting Up ThingsBoard Edge and Gateway on the Local PC

Make sure you have **ThingsBoard Edge** and **ThingsBoard IoT Gateway** installed on the Windows PC. The Edge is essentially a local ThingsBoard server that syncs with ThingsBoard PE Cloud, and the Gateway is a service that connects devices (via Modbus, OPC-UA, etc.) to ThingsBoard. For a Windows setup, you might run Edge in a Docker container or as a Windows service, and the Gateway as a Python service or Docker container.

- **ThingsBoard Edge Setup:** Configure the Edge to connect to your ThingsBoard PE Cloud instance. This typically involves setting the cloud's host and security token for the Edge. (In ThingsBoard PE, you'd register an Edge on the cloud, get an Edge ID and access token, and provide those to the local Edge instance.) Once running, verify that the Edge appears online in the TB Cloud console.

- **ThingsBoard Gateway Setup:** Install the Gateway (for example, via Python `pip install thingsboard-gateway` or using the provided installer). The Gateway has a main configuration file (usually `tb_gateway.yaml`) where you point it to a ThingsBoard server. Since Edge is on the same machine, configure the gateway to connect to the local Edge. Typically, you can use `localhost` (127.0.0.1) and default ThingsBoard MQTT port **1883** for connectivity. The Gateway will use MQTT or HTTP to send data to ThingsBoard Edge. Ensure the Gateway has the correct **Edge's access token** or credentials so it can authenticate.

After configuration, **start the ThingsBoard Gateway** service. In the logs, you should see it connect to the ThingsBoard Edge (check for messages like "Connected to ThingsBoard" or no errors). Now the Gateway is ready to bridge between Modbus and ThingsBoard. Both the Edge and Gateway are on the same PC, so no special network routing is needed between them – they communicate via the loopback interface.

Tip: It's often easiest to test the Gateway connection to Edge by running the Gateway and checking the Edge's device list. The Gateway itself can appear as a device (if using default gateway token), or at least no connection errors should be in logs. If connection fails, double-check the host (should be `localhost` for Edge on same PC) and port (1883 for MQTT, or 8080 if using HTTP) in `tb_gateway.yaml`, and verify the access token is correct. Once this is set, you can proceed to configure the Gateway's Modbus integration.

Step 6: Configuring ThingsBoard Gateway to Communicate with SmartLogger (Modbus TCP)

With Modbus active on the SmartLogger and the Gateway running, the next step is to configure the **Modbus connector** in ThingsBoard Gateway. This is done in the Gateway's configuration directory, typically by editing the **`modbus.json`** file (usually found in `/config` or similar folder). In this JSON

config, you will define the connection parameters to the SmartLogger and the data to poll. Key configuration sections include the Modbus server connection and the list of devices/registers to poll.

Configure the Modbus connection:

- **Connection Type:** Set the Modbus connector to use **TCP**. In `modbus.json`, under `"master": { "slaves": [ ... ] }`, use `"type": "tcp"` and `"method": "socket"` (socket is typically the method for TCP) (MISC-TN-024: Automated test equipment (ATE) monitoring with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki).

- **Host and Port:** Specify the **SmartLogger's IP address** as `"host"` and port **502** as `"port"` (MISC-TN-024: Automated test equipment (ATE) monitoring with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki). For example, `"host": "192.168.0.10", "port": 502`. This tells the Gateway where to reach the SmartLogger on the network.

- **Unit ID:** Set `"unitId"` to the SmartLogger's Modbus address you configured (for instance, 100) (Set Active Power limitation via Modbus). This ensures the Gateway's requests are addressed to the SmartLogger (which internally may aggregate data from multiple devices). If the SmartLogger is acting as a transparent gateway to inverters, you would use the inverter's IDs here; but since the SmartLogger 3000 can provide data itself (and bridge to inverters), typically you use the SmartLogger's own ID for overall system data.

- **Device Name and Type:** Provide a logical `"deviceName"` for this data source, e.g. `"HuaweiSmartLogger3000"` or any friendly name. This will become the device name in ThingsBoard. You can also set `"deviceType"` (like `"solar_logger"` or simply `"default"`).

- **Polling parameters:** Set the `"pollPeriod"` (in milliseconds) which is how often to poll telemetry. For example, `5000` ms for 5 seconds. You can also set separate `"attributesPollPeriod"` if you intend to fetch some data as attributes less often. If you only use telemetry (timeseries) data, one poll period is fine.

- **Retries and Timeout:** You may include settings like `"timeout": 3000` (3 seconds) and retry options (e.g., `"retries": 3`, `"retryOnTimeout": true`) to handle transient communication issues. The config can include `"connectAttemptCount"` and similar parameters to make the gateway robust to SmartLogger reboots or network issues (MISC-TN-024: Automated test equipment (ATE) monitoring with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki).

Here is a **simplified example** of a Modbus connector configuration snippet for the SmartLogger (for illustration):

{

```
  "master": {
    "slaves": [
      {
        "host": "192.168.0.10",
        "port": 502,
        "unitId": 100,
        "deviceName": "HuaweiSmartLogger3000",
        "deviceType": "default",
        "pollPeriod": 5000,
        "timeout": 3000,
        "retryOnEmpty": true,
        "connectAttemptCount": 5,
        "timeseries": [ ... ],
        "attributes": [ ... ]
      }
    ]
  }
}
```

In the above, we define one Modbus TCP device (the SmartLogger at IP 192.168.0.10 and ID 100) that will be polled every 5 seconds. The `"timeseries"` and `"attributes"` sections (left `...` here) will be filled in with register mappings in the next step. This configuration is adapted from the official ThingsBoard Gateway documentation, which shows how to list **slaves** (devices) with host, port, etc. (MISC-TN-024: Automated test equipment (ATE) monitoring with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki).

After editing `modbus.json`, **restart the ThingsBoard Gateway** service to apply the changes. The gateway log should indicate it connected to the SmartLogger (or at least attempted). If everything is correct, you won't see connection errors. If you do see errors like "connection refused" or "timeout", re-check the IP, port, and that the SmartLogger's Modbus settings (especially client IP permission) are correct. Also confirm the SmartLogger is reachable (try using a Modbus test tool or even `ping`).

Step 7: Discovering Available Modbus Registers on the SmartLogger

Identifying which Modbus registers hold the data you need (e.g. active power, voltage, current) is crucial. **Ideally, obtain the Huawei SmartLogger 3000 Modbus register map documentation.** Huawei provides a **"SmartLogger ModBus Interface Definitions"** document that lists all register addresses and their meanings (Set Active Power limitation via Modbus). For example, this document shows that there are registers for Active Power, Energy, Voltage, etc., often in the 40xxx range. If you have this document, use it as the authoritative source for register addresses.

If the official register map is not available, you have a few options to discover registers:

- **Community References:** Look for online forums or knowledge base articles. For instance, Huawei support or partners sometimes publish common registers. A Huawei quick guide shows that reading register **40525** returns the **Active Power** of the SmartLogger (How to use Modbus TCP – Help Centre General). Such hints can guide you to known addresses. Likewise, common metrics may reside around certain ranges (e.g. 405xx for power or 402xx for some status).

- **Modbus Scanning Tools:** Use a Modbus client tool to scan or probe ranges of registers. Tools like *Modbus Poll/Modbus Scanner* or the free **CAS Modbus Scanner** can automate reading a series of registers to see which ones respond with non-zero or changing values (Modbus Scanner: Perfect to Discover and Test Modbus - Chipkin). For example, you might use Modbus Poll on a laptop (connected to SmartLogger) to read registers 40000–41000 and observe which ones yield data. **Caution:** Only perform read (Function 3 or 4) scans. Do not write to unknown registers as that could change settings inadvertently.

- **SunSpec Standard (if applicable):** Some solar devices use the SunSpec standardized register sets. Check if SmartLogger supports SunSpec (though Huawei often uses their own mapping). If it did, certain blocks (like starting at 40000) might follow a known structure. However, since Huawei provides a custom map, it's safer to rely on their definitions.

Using **Modbus Poll** (or similar), you can manually connect to the SmartLogger IP and try reading known register addresses:

- Set connection to Modbus TCP, enter SmartLogger IP and port 502.6
- Use the SmartLogger's unit ID (if Modbus Poll requires an ID, put the ID you set, e.g., 100. Some tools use ID 0 to indicate broadcast or the gateway itself – in Huawei's guide, they used Slave ID 0 to denote the SmartLogger's address in one example (How to use Modbus TCP – Help Centre General), but typically you use the actual ID number).

- Read holding registers at addresses of interest. For instance, reading address **40525** should return the current active power output of the system (How to use Modbus TCP – Help Centre General). You might see a value in watts. Try reading a range around that address to find related values (e.g., 40521–40530 might include other power measurements).

- Look for likely voltage registers (which might be in volts, typically around a few hundred for AC, or DC string voltages maybe up to 1000). Current registers might be in amperes. Sometimes registers are 16-bit and need combination for 32-bit values.

If the SmartLogger is aggregating multiple devices, there may be registers for each inverter or meter. The documentation might list blocks of registers per device address or a way to query by device ID. However, often the SmartLogger provides **overall values** (total power, total energy) and possibly device-specific values accessible via different unit IDs or index.

Summary: Use available documentation or scanning tools to identify the register addresses for **ActivePower, Voltages, Currents, Energy, etc.** A known starting point from community info is that *Active Power* is reported at register 40525 (How to use Modbus TCP – Help Centre General). You will likely find other useful registers in nearby ranges or via the Huawei Modbus definition PDF. Keep notes of each register address and its corresponding metric for the next step.

Step 8: Mapping SmartLogger Registers to Telemetry Keys in ThingsBoard Gateway

Now that you know which registers you want to read, configure the **telemetry mapping** in the `modbus.json` file. Under the device configuration (the `"timeseries"` or `"attributes"` sections for your SmartLogger device entry), you will map each Modbus register to a telemetry key in ThingsBoard. Each mapping entry typically includes:

- A **key (tag)**: The name of the telemetry data as it will appear in ThingsBoard (e.g. `"ActivePower"` or `"GridVoltage"`).

- The Modbus **address**: The register address (keeping in mind any offset as required by the gateway – usually you use the register number as given, and the gateway handles zero-based vs one-based internally).

- The **function code**: 3 for holding registers (4xxxx) or 4 for input registers (3xxxx). Most Huawei SmartLogger data uses holding registers (4xxxx) read with function 3 (Set Active Power limitation via Modbus).

- **Register count and data type**: Many values are 16-bit integers (`"type": "16int"` for signed 16-bit or `"16uint"` for unsigned) (MISC-TN-024: Automated test equipment (ATE) monitoring with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki). Larger values (like energy or power that exceed 65k, or floats) might be 32-bit (`"32int"`, `"32uint"` or even IEEE 32-bit float). Check the documentation: if a value spans two registers, set `"registerCount": 2` and use a 32-bit type. For example, Active Power might be a 32-bit value requiring two registers (40525–40526). If unsure, you can trial a single register first; if the number seems too low or truncated, it might actually need two registers combined.

Example telemetry mapping: Suppose we want to report Active Power, AC Voltage, and AC Current from the SmartLogger. Based on our register discovery, assume:

- Active Power is at **40525-40526** (32-bit signed int, representing watts).
- AC Voltage is at **40501** (16-bit, representing volts).
- AC Current is at **40503** (16-bit, representing amperes).

We would add entries to the `"timeseries"` list like so:

```
"timeseries": [
  {
    "tag": "ActivePower",
    "type": "32int",
    "functionCode": 3,
    "address": 40524,
    "registerCount": 2
  },
  {
    "tag": "AC_Voltage",
    "type": "16uint",
    "functionCode": 3,
    "address": 40500,
    "registerCount": 1
  },
  {
    "tag": "AC_Current",
    "type": "16uint",
    "functionCode": 3,
    "address": 40502,
```

```
    "registerCount": 1
  }
]
```

(Note: The addresses in the JSON are often zero-based. Some gateways expect address = actual_modbus_address - 1. In ThingsBoard Gateway's case, if the documentation says 40525, you likely put 40524 in config because TB Gateway treats it as zero-based offset for function 3. This is why in the example above we used 40524 for ActivePower if the register is labeled 40525 in documentation. Be sure to verify this by testing – if data seems off by one address, adjust accordingly.)

Each `"tag"` will become a telemetry key on the device in ThingsBoard. The `"functionCode": 3` tells the gateway to use Modbus function 3 (Read Holding Registers) (MISC-TN-024: Automated test equipment (ATE) monitoring with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki). The `registerCount` of 2 for ActivePower means it will read two 16-bit registers starting at address 40524 (i.e., 40525-40526 in one-based terms) and combine them into a 32-bit value. The data `"type" "32int"` specifies it's a 32-bit signed integer, so negative values (for power import/export) can be represented if applicable. Voltage and Current are single-register values (16 bits each).

After adding all required telemetry mappings, you can also map some values as attributes if needed (attributes are typically static or infrequently changing values, e.g., device firmware version or settings). The syntax is similar, but under an `"attributes": [ ... ]` section.

Reload the Gateway after saving `modbus.json`. It will now start polling those specific registers at the defined interval. In the gateway's log, you should see it reading values and publishing to ThingsBoard. Also, on ThingsBoard Edge/Cloud, navigate to the device (with name you set, e.g., "HuaweiSmartLogger3000") and check the latest telemetry: you should see keys like `ActivePower`, `AC_Voltage`, etc., updating with numeric values. If they show up, **congratulations** – data is flowing!

Troubleshooting mapping issues: If you see the device in TB but no telemetry, or some keys not appearing:

- Ensure the keys are listed under the device's telemetry latest values. If not, the gateway might not be successfully reading them. Check the gateway logs for errors (e.g., "illegal register address" means the address is wrong or SmartLogger is not responding to it).

- Double-check the register addresses and unitId. If all values error out, perhaps the unitId is wrong (SmartLogger's address) – try unitId 0 or 1 as a test if 100 doesn't work, or confirm the SmartLogger address in its UI.

- If some values come but others are zero or incorrect, you might have the wrong address or data type. Try reading those registers with an external tool (Modbus Poll) to compare. It may be an offset issue (try address+1 or -1).

- The ThingsBoard Gateway also allows logs at DEBUG level for the Modbus connector, which can show the raw data received, helpful to diagnose byte order issues (the config supports `"byteOrder"` and `"wordOrder"` if needed). For instance, multi-register values might need swapping depending on endianness. By default, Huawei likely uses Big Endian (most significant register first), which is standard, so you may not need to set custom byte order.

Step 9: Enabling Control – Writing Commands via Modbus (Device Control)

Modbus TCP isn't just for reading; it also supports **writing** registers or coils to control devices. The ThingsBoard Gateway and SmartLogger 3000 can be set up to allow remote control commands such as setting parameters or even sending on/off commands, **provided the SmartLogger's Modbus interface defines writable registers and the device permits it**.

Does SmartLogger support writes? Yes. Huawei's SmartLogger Modbus interface includes registers for control and configuration. For example, register **40420** is used for "Active power limitation" – writing to this register (with Function Code 16, Write Multiple Registers) allows you to set a power output limit (Set Active Power limitation via Modbus). Similarly, there are registers for adjusting power factor, turning inverters on/off, etc., as defined in the Huawei Modbus guide. Before attempting writes, consult the official Modbus register list to identify the exact address and valid value ranges for commands.

Configuring ThingsBoard to send writes: In ThingsBoard's Gateway `modbus.json`, you can define **RPC commands** mapping. RPC (Remote Procedure Call) in TB allows the cloud or edge to send a command to a device. You can map an RPC from TB to a Modbus write operation in the gateway config. For example, you might add an `"rpcMethods"` section for the device, mapping a method name (like `"setActivePowerLimit"`) to writing a value to holding register 40420. The config might look like:

```
"rpcMethods": [
  {
    "tag": "setActivePowerLimit",
    "type": "32int",
    "functionCode": 16,
    "address": 40419,
    "registerCount": 2
  }
]
```

(This would map an RPC call with method `"setActivePowerLimit"` to writing two registers starting at 40419, which corresponds to 40420 in one-based addressing, with a 32-bit integer value). Then, from ThingsBoard UI or via REST, you can initiate an RPC call to the device with that method and a value (e.g., setActivePowerLimit with value 5000 to limit output to 5kW). The gateway will execute a Modbus write (function 16) to send that command to SmartLogger (Set Active Power limitation via Modbus).

Important considerations for writes:

- **Permissions:** Ensure the SmartLogger and inverters are in a state that allows remote control. Some devices require being in "remote" mode or have settings that permit external dispatching of commands. Also, the user role you used on SmartLogger might need adequate privileges (Advanced User typically can do it).

- **Testing safely:** Try write commands in a controlled manner. For example, use Modbus Poll's "Write Register" function on a known safe register (like a small limit) to see if SmartLogger accepts it and the device responds. Huawei's documentation often provides the procedure (as in the example of writing 40420 to set active power, which involves writing two registers: the limit value and maybe a confirmation code) (Set Active Power limitation via Modbus).

- **Use Cases:** Through Modbus writes, you can potentially **control active/reactive power**, reset energy counters, acknowledge alarms, etc., if those are defined. This can be integrated with ThingsBoard rules – for instance, an alarm in TB could trigger an RPC to curtail power if needed.

If you do not require any remote control, you can skip configuring RPC. Simply know that Modbus TCP **does allow device control**, and the SmartLogger supports it for specific registers. Always refer to the official register definitions to avoid writing to the wrong register. Misusing write commands could, in worst case, trip the system or override important settings. If unsure, consult Huawei support or limit writes to documented safe functions.

Step 10: Simulating the Setup on a Windows PC (Optional Testing)

Before deploying on real hardware, you can **simulate the SmartLogger-Modbus setup** on your Windows PC to ensure your ThingsBoard Edge and Gateway configuration works correctly. This is especially useful if you don't have the SmartLogger device on hand yet or want to experiment without affecting the actual system.

Option 1: Use a Modbus Simulator – There are tools that turn your PC into a Modbus TCP server (slave), allowing you to define registers and values:

- **ModbusPal:** An open-source Java-based Modbus slave simulator (ModbusPal - Java MODBUS simulator). It lets you create a virtual device with any number of registers. You can

script registers to change over time (e.g., simulate a power output increasing/decreasing).

- **Modbus Slave (ModbusTools):** From the same makers of Modbus Poll, a GUI tool to simulate modbus slaves. You can set up registers and manually input values, and the tool will respond on a specified IP/port. (The Huawei guide references modbustools download for Modbus Poll (Set Active Power limitation via Modbus); their "Modbus Slave" tool can be obtained there as well).

- **Python/Node simulators:** If you're comfortable with coding, libraries like **PyModbus** in Python can create a simple Modbus TCP server with preset registers. This could be as simple as a small script that listens on port 502 and returns some dummy data for a few addresses.

To simulate, you might do the following:

1. **Run the simulator on your PC.** Configure it to listen on port 502 (or another port, but then match it in Gateway config) and use an arbitrary Unit ID (say 1). Define a couple of registers (for example, holding register 40525 with a value of 12345, etc.).

2. **Point the ThingsBoard Gateway to the simulator.** Change the `modbus.json` `"host"` to `127.0.0.1` (localhost) and the `"port"` if you used something other than 502 (you might use 1502 if running as non-admin on Windows since port 502 may require admin privileges). Also set the `unitId` to what you chose (e.g., 1).

3. **Run the Gateway and Edge** as usual. The Gateway will connect to the simulator and poll the registers. Check if the data appears in ThingsBoard (Edge/Cloud) for the device. If you see the dummy values coming through, your pipeline is working. You can adjust the values in the simulator in real-time to see if TB reflects the changes (which it should on next poll). This confirms that your **Gateway→Edge→Cloud chain and JSON mappings are correct**.

By simulating, you can validate the end-to-end connectivity, telemetry upload, and even RPC if you program the simulator to handle write requests. For example, ModbusPal can be scripted to log or react when a certain register is written.

Option 2: Dry-Run with SmartLogger in Lab: If you have the actual SmartLogger but not connected to a live system, you can still test by enabling Modbus and seeing if you can read some basic info (maybe device info registers) just to ensure connectivity. The simulator, however, is useful if no hardware is available.

Tips: When using a simulator, remember to revert the config back to the real SmartLogger IP and unit ID afterwards. Also, keep firewall in mind – if the Gateway is connecting to a simulator on the same PC, ensure the loopback is allowed. Usually this is fine since it's local. If using a different PC for simulation, ensure network connectivity between them.

Step 11: Visualizing Data and Utilizing ThingsBoard PE Cloud Features (Real-time, History, Alarms)

Once your data is flowing into ThingsBoard PE Cloud via the Edge, you can leverage the platform's powerful features for monitoring and analytics:

- **Real-time Monitoring Dashboards:** ThingsBoard allows you to create custom dashboards to watch live data. For example, you can add a gauge widget to show the current Active Power, or a real-time line chart to plot power or voltage over the last few minutes. **Latest value** widgets (digital gauges, analog gauges, cards) display the most recent telemetry readings and update in real-time as new data arrives (Working with telemetry data | ThingsBoard Community Edition). You could create a dashboard with multiple widgets: one for each phase voltage, one for total power, etc., updating every polling cycle. This gives an at-a-glance view of the system's status.

- **Historical Data Analytics:** ThingsBoard automatically stores time-series telemetry in a database (SQL or NoSQL) (Working with telemetry data | ThingsBoard Community Edition). You can visualize historical trends by using **chart widgets** on dashboards (Working with telemetry data | ThingsBoard Community Edition). For instance, plot Active Power over the last 24 hours or energy produced per day over the last month. The charts can be configured for different time windows and refresh intervals. Historical data is invaluable for analyzing performance (e.g., peak generation times, voltage fluctuations). The **Professional Edition** of ThingsBoard also supports **Trendz Analytics**, which can do advanced analysis or aggregations, but even out-of-the-box you have standard time-series charts. **Tip:** Use the "Time-series Line Chart" widget to display historical graphs. You can have both real-time (auto-update) and the ability to scroll back in time. ThingsBoard dashboards support simultaneous real-time and historical visualization (Working with telemetry data | ThingsBoard Community Edition), giving you flexibility in monitoring.

- **Alarm Rules and Notifications:** ThingsBoard PE offers a robust alarm mechanism. You can define alarm rules that trigger based on incoming telemetry or attribute values (for example, if a voltage exceeds a threshold or if the SmartLogger reports an error code). The simplest way to set up an alarm is via the **Device Profile** in ThingsBoard: you can configure a threshold-based alarm condition for your device's telemetry keys (Lesson 4. Alarm management | ThingsBoard Professional Edition). For instance, create a rule "ActivePowerHigh" that triggers an alarm of severity "Major" if `ActivePower` goes above, say, 50000 W (depending on your system capacity). Alarms can also be cleared automatically when the value returns to normal. Once active, alarms are visible in ThingsBoard's **Alarms panel** (both on a device page and in dashboards via alarm widgets). ThingsBoard provides a centralized interface to view, acknowledge, and clear alarms, with filtering by severity or type (Lesson 4. Alarm management | ThingsBoard Professional Edition). You can add an **Alarms Table** widget to a dashboard to list active alarms and their status.

- **Historical Alarms and Events:** All alarms and events can be stored, allowing you to review past incidents. This is useful for troubleshooting recurring issues (e.g., if you see an overvoltage alarm daily at noon, it indicates a pattern). ThingsBoard's alarm interface supports

real-time monitoring and historical analysis of alarm occurrences (Lesson 4. Alarm management | ThingsBoard Professional Edition). You might create a dashboard tab specifically for maintenance, showing a timeline of alarms or a count of alarms over time.

- **Additional PE Features:** ThingsBoard PE Cloud will let you manage users and access, so you can create a read-only dashboard for clients or a mobile app view. You can also set up **email or SMS notifications** for critical alarms using Rule Engine nodes (for example, send an email whenever an alarm is raised). Since you are using an Edge, note that the rule processing can be done on Edge or Cloud. Simpler is to do it on the Cloud via device profile alarms or rule chains; the Edge will forward telemetry to Cloud and Cloud will evaluate the alarm conditions. PE also has an Edge rule sync capability if needed. For most monitoring scenarios, cloud-side alarms are fine.

Building a Sample Dashboard: As an example, you could create a dashboard for the SmartLogger that has: a gauge for current power, a chart for today's energy production, a chart for the last 7 days energy, and an alarm log. Use ThingsBoard's dashboard editor to add these widgets. Data source will be the device (SmartLogger 3000) and the keys you mapped (ActivePower, etc.). You can configure the chart to show both real-time and historical by enabling the **history** time window and live updates (Working with telemetry data | ThingsBoard Community Edition). This way you can watch values update live and also scroll back.

Finally, once deployed, **test the end-to-end setup under real conditions**. Turn on the solar inverters or change conditions to see the telemetry reflect on the dashboard. Trigger an alarm scenario (if safe to do so) to ensure your alarm rules fire (e.g., maybe set a low threshold temporarily to test). Make use of ThingsBoard's rich features to get insights – for example, set up **computed telemetry** or server-side analytics if you need to combine values (like sum multiple meter readings, etc.). The combination of Huawei SmartLogger's data and ThingsBoard's IoT platform capabilities will provide a comprehensive monitoring and management solution.

Conclusion & Recommendations: This step-by-step guide covered the full journey from understanding Modbus TCP fundamentals to enabling SmartLogger's Modbus, setting up ThingsBoard Edge + Gateway, mapping data, and visualizing it on the cloud. For someone new to Modbus and ThingsBoard, take it one step at a time: get the Modbus link working and reading a couple of values first, then gradually build out the full data mapping and dashboards. Always refer to official documentation for specifics (Huawei's register list (Set Active Power limitation via Modbus) and ThingsBoard's gateway guide) when in doubt. With careful configuration and testing, you'll have a reliable integration where the Huawei SmartLogger 3000 feeds solar plant data to ThingsBoard PE Cloud for real-time monitoring, historical analysis, and proactive alerts. Good luck!

Sources:

1. Huawei SmartLogger 3000 Modbus setup instructions (Set Active Power limitation via Modbus) (Huawei SmartLogger 3000 | Eniris Documentation)

2. ThingsBoard Gateway official documentation (Modbus connector example) (MISC-TN-024: Automated test equipment (ATE) monitoring with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki) (MISC-TN-024: Automated test equipment (ATE) monitoring

with SBCSPG gateway and ThingsBoard IoT platform - DAVE Developer's Wiki)

3. Huawei SmartLogger Modbus TCP usage guide (Modbus Poll example) (How to use Modbus TCP – Help Centre General) (How to use Modbus TCP – Help Centre General)

4. ProSoft Modbus TCP/IP introduction (client-server model explained) (Introduction to Modbus TCP/IP) (Introduction to Modbus TCP/IP)

5. Huawei “Set Active Power via Modbus” guide (writable register example) (Set Active Power limitation via Modbus)

6. ThingsBoard documentation on data visualization and alarms (Working with telemetry data | ThingsBoard Community Edition) (Lesson 4. Alarm management | ThingsBoard Professional Edition)

7. Chipkin Modbus Scanner utility description (for register discovery) (Modbus Scanner: Perfect to Discover and Test Modbus - Chipkin)

Guide: Connecting Huawei SmartLogger 3000 to ThingsBoard Gateway (Modbus)

Overview

This guide explains how to integrate a Huawei SmartLogger 3000 with ThingsBoard IoT Gateway for real-time telemetry and remote control. We will cover the required software, Modbus register mappings (extracted from the SmartLogger’s C# code and Huawei docs), configuration steps, and validation of data flow to a ThingsBoard PE (Professional Edition) Cloud instance. The goal is to reliably poll inverter data (e.g. Active Power, AC Voltage, Current, etc.) and send it to ThingsBoard, as well as to send RPC commands (like enabling or disabling an inverter) from ThingsBoard to the SmartLogger.

1. Requirements and Prerequisites

- **Huawei SmartLogger 3000** with network access. Ensure the SmartLogger’s Modbus TCP interface is **enabled** and configured:
    - Via the SmartLogger web UI, go to communication settings and set **“Link setting” = Enable (Unlimited)** and **“Address mode” = Communication address** (Huawei SmartLogger 3000 | Eniris Documentation). Choose a unique Modbus address for the SmartLogger itself (distinct from inverter IDs) (Huawei SmartLogger 3000 | Eniris

Documentation). (By default, Modbus TCP uses port 502.)

    - All connected devices (inverters, meters) should have unique Modbus IDs (Huawei SmartLogger 3000 | Eniris Documentation). Note the Modbus **Unit IDs** of each inverter as configured in the SmartLogger.

- **ThingsBoard IoT Gateway** – installed via Docker (or on a VM). We will use the official Docker image for convenience. Ensure you have:
    - Docker installed on the machine that will run the gateway.
    - Access to your ThingsBoard PE Cloud instance (you'll need the host URL and a device access token or gateway connection token).
- **ThingsBoard Device Provisioning**:
    - Decide if you will represent each inverter as a separate device in ThingsBoard or aggregate data under one device (e.g., one device per site). Using the gateway in "remote devices" mode is typical – the gateway itself connects with one **Gateway Device** credentials, and each inverter is a sub-device. In this guide, we treat each inverter as a separate device for clarity.
    - In ThingsBoard PE Cloud, create a **Device** (if using one per inverter, create one for each, or create one gateway device if using gateway mode) and note the **device access token**. For gateway mode, create a single "Gateway" device and use its token – the gateway will auto-provision sub-devices.

2. SmartLogger Modbus Setup

Before configuring the gateway, verify the SmartLogger's Modbus settings:

- **Enable Modbus TCP**: As mentioned, enable Modbus on the SmartLogger's web interface (Huawei SmartLogger 3000 | Eniris Documentation). The SmartLogger supports reading and controlling connected inverters via Modbus TCP (Huawei SmartLogger 3000 | Eniris Documentation). By default, the SmartLogger listens on port 502 for Modbus TCP.
- **Device IDs**: Ensure each inverter (and any meter) connected to the SmartLogger has a unique Modbus ID (often set automatically or configurable in the SmartLogger under Device Modbus Addresses) (Huawei SmartLogger 3000 | Eniris Documentation). Typically, inverters might be IDs 1, 2, 3, etc. (The provided C# code iterated Device_ID from 1 to 5, meaning up to 5 devices on one SmartLogger.)
- **Network**: Note the SmartLogger's IP address on your network. The gateway machine must be able to reach this IP and port. If deploying on-site, the gateway should be in the same LAN or have VPN access. No firewall should block port 502 between the gateway and SmartLogger.

3. Deploying ThingsBoard Gateway (Docker Setup)

We will run the ThingsBoard IoT Gateway in Docker and configure its Modbus connector:

Pull the Gateway Image:

```
docker pull thingsboard/tb-gateway:latest
```

1.
2. **Prepare Configuration**: Create a directory on the host (e.g., `tb-gateway-config`) and within it, prepare two config files:

    - `tb_gateway.yaml` – main gateway config (contains ThingsBoard connection and enabled connectors).

    - `modbus.json` – Modbus connector config.

For example, create `tb_gateway.yaml` with the following minimal content (replace placeholders):

```
thingsboard:
  host: "<YOUR_TB_HOST_OR_IP>"        # e.g., "thingsboard.cloud"
  port: 1883
  remoteConfiguration: false
  security:
    accessToken: "<YOUR_GATEWAY_TOKEN>"
connectors:
  - name: Modbus Connector
    type: modbus    # enable modbus connector
    configuration: modbus.json
```

3. Use the device access token from ThingsBoard. If using gateway mode (multiple sub-devices), use the gateway device's token.

4. **Modbus Connector Config**: In the same directory, create `modbus.json` as described in the next section (section 4).

Run the Container:

```
docker run -d --name tb-gateway \
  -v $(pwd)/tb_gateway.yaml:/thingsboard_gateway/config/tb_gateway.yaml \
  -v $(pwd)/modbus.json:/thingsboard_gateway/config/modbus.json \
  --network="host" thingsboard/tb-gateway:latest
```

5. (Here we use host networking for simplicity so the container can reach the SmartLogger IP. Alternatively, specify `-p 1883:1883` etc., and ensure network routing is configured.)

The gateway will connect to ThingsBoard (over MQTT using the token) and initialize the Modbus connector.

4. Modbus Connector Configuration (Register Mapping)

The core of the setup is mapping SmartLogger/Inverter Modbus registers to ThingsBoard telemetry keys. Based on the provided C# software and Huawei documentation, the following Modbus addresses correspond to key data points on Huawei inverters (via SmartLogger):

Telemetry Register Map:

- **AC Line Voltages:** Line-to-line voltages:
    - `Voltage_AB` – Address 32066, 1 register (U16). Scaled by 0.1 (e.g., 2310 = 231.0 V) (Solar Inverter Modbus Interface Definitions).
    - `Voltage_BC` – Address 32067, 1 register (U16). Scale 0.1 (Solar Inverter Modbus Interface Definitions).
    - `Voltage_CA` – Address 32068, 1 register (U16). Scale 0.1 (Solar Inverter Modbus Interface Definitions). *(For a single-phase system, only `Voltage_AB` (actually phase-to-neutral) would be used by the device (Solar Inverter Modbus Interface Definitions).)*
- **AC Phase Currents:**
    - `Current_A` – Address 32072, 2 registers (I32). Scaled by 0.001 (e.g., 12345 = 12.345 A) (Solar Inverter Modbus Interface Definitions).
    - `Current_B` – Address 32074, 2 registers (I32). Scale 0.001 (Solar Inverter Modbus Interface Definitions).
    - `Current_C` – Address 32076, 2 registers (I32). Scale 0.001 (Solar Inverter Modbus Interface Definitions). *(Single-phase inverters may use only one current value labeled as "Grid current" at 32072) (Solar Inverter Modbus Interface Definitions).*
- **Active Power:**
    - `ActivePower` – Address 32080, 2 registers (I32). This is the total AC output power of the inverter in kW, scaled by 0.001 (e.g., 5000 = 5.000 kW) (Solar Inverter Modbus Interface Definitions). Positive values indicate power feeding into the grid (Solar Inverter Modbus Interface Definitions) (negative if consuming power, e.g. hybrid inversion).
- **Reactive Power:**

    - `ReactivePower` – Address 32082, 2 registers (I32). kVar, scaled by 0.001 (Solar Inverter Modbus Interface Definitions). Can be positive or negative (inductive or capacitive output).

- **Power Factor:**
    - `PowerFactor` – Address 32084, 1 register (I16). Scaled by 0.001 (e.g., 999 = 0.999). This is the cosine φ value (Solar Inverter Modbus Interface Definitions).

- **Grid Frequency:**
    - `Frequency` – Address 32085, 1 register (U16). Scaled by 0.01 (e.g., 5000 = 50.00 Hz) (Solar Inverter Modbus Interface Definitions).

- **Efficiency:**
    - `Efficiency` – Address 32086, 1 register (U16). Scaled by 0.01 (percentage) (Solar Inverter Modbus Interface Definitions).

- **Internal Temperature:**
    - `Temperature` – Address 32087, 1 register (I16). Scaled by 0.1 (e.g., 350 = 35.0 °C) (Solar Inverter Modbus Interface Definitions).

- **Insulation Resistance:**
    - `InsulationResistance` – Address 32088, 1 register (U16). Unit MΩ, scaled by 0.001 (Solar Inverter Modbus Interface Definitions).

- **Device Status:**
    - `DeviceStatus` – Address 32089, 1 register (U16). Status code indicating the inverter state (Solar Inverter Modbus Interface Definitions). For example, 0x0000 = Standby, 0x0200 = On-grid (running), 0x0301 = Shutdown by command, etc. (Refer to Huawei documentation for full code meanings.)

- **Energy Meters:**
    - `TotalEnergy` – Address 32106, 2 registers (U32). Accumulated energy (total yield) in kWh, scaled by 0.01 (Solar Inverter Modbus Interface Definitions).

    - `DailyEnergy` – Address 32114, 2 registers (U32). Daily energy yield in kWh, scaled by 0.01 (Solar Inverter Modbus Interface Definitions).

- *(Optional: DC input readings like PV voltage/current per MPPT are available in lower addresses, but we focus on AC output and energy values.)*

Control (Holding) Registers:

For remotely controlling the inverter, Huawei provides holding registers to issue start/stop commands:

- **Start Inverter:** Register address 40200 (Write-only). Writing a “1” triggers a startup (connect to grid) (Solar Inverter Modbus Interface Definitions).

- **Stop Inverter:** Register address 40201 (Write-only). Writing a “1” triggers a shutdown command (Solar Inverter Modbus Interface Definitions).
  (These registers are defined as separate commands – the SmartLogger’s interface defines 40200 for “Startup” and 40201 for “Shutdown” (Solar Inverter Modbus Interface Definitions). Ensure the target Unit ID is the inverter’s ID for individual control.)

Now we will put this into the gateway’s `modbus.json` configuration. Below is an example configuration for **two inverters** (Unit ID 1 and 2). You can extend the `"devices"` list for all 13 units or as needed:

```
{
  "server": {
    "name": "SmartLogger3000",
    "type": "tcp",
    "host": "192.168.1.50",         // SmartLogger IP
    "port": 502,
    "timeout": 35,
    "methods": ["socket"]
  },
  "devices": [
    {
      "unitId": 1,
      "deviceName": "Inverter_1",
      "attributes": [],  // (you can map static attributes if needed)
      "timeseries": [
        { "tag": "Voltage_AB",         "type": "16uint", "functionCode": 4, "address": 32066, "objectsCount": 1
},
        { "tag": "Voltage_BC",         "type": "16uint", "functionCode": 4, "address": 32067, "objectsCount":
1 },
        { "tag": "Voltage_CA",         "type": "16uint", "functionCode": 4, "address": 32068, "objectsCount":
1 },
        { "tag": "Current_A",          "type": "32int",  "functionCode": 4, "address": 32072, "objectsCount": 2
},
        { "tag": "Current_B",          "type": "32int",  "functionCode": 4, "address": 32074, "objectsCount": 2
},
```

```
        { "tag": "Current_C",          "type": "32int",  "functionCode": 4, "address": 32076, "objectsCount": 2 },
        { "tag": "ActivePower",        "type": "32int",  "functionCode": 4, "address": 32080, "objectsCount": 2 },
        { "tag": "ReactivePower",      "type": "32int",  "functionCode": 4, "address": 32082, "objectsCount": 2 },
        { "tag": "PowerFactor",        "type": "16int",  "functionCode": 4, "address": 32084, "objectsCount": 1 },
        { "tag": "Frequency",          "type": "16uint", "functionCode": 4, "address": 32085, "objectsCount": 1 },
        { "tag": "Efficiency",         "type": "16uint", "functionCode": 4, "address": 32086, "objectsCount": 1 },
        { "tag": "Temperature",        "type": "16int",  "functionCode": 4, "address": 32087, "objectsCount": 1 },
        { "tag": "InsulationResistance","type": "16uint","functionCode": 4, "address": 32088, "objectsCount": 1 },
        { "tag": "DeviceStatus",       "type": "16uint", "functionCode": 4, "address": 32089, "objectsCount": 1 },
        { "tag": "TotalEnergy",        "type": "32int",  "functionCode": 4, "address": 32106, "objectsCount": 2 },
        { "tag": "DailyEnergy",        "type": "32int",  "functionCode": 4, "address": 32114, "objectsCount": 2 }
      ],
      "rpc": [
        { "tag": "enableInverter",   "type": "16uint", "functionCode": 6, "address": 40200, "objectsCount": 1 },
        { "tag": "disableInverter",  "type": "16uint", "functionCode": 6, "address": 40201, "objectsCount": 1 }
      ]
    },
    {
      "unitId": 2,
      "deviceName": "Inverter_2",
      "attributes": [],
      "timeseries": [  ...same mapping as above...  ],
      "rpc": [
        { "tag": "enableInverter",  "type": "16uint", "functionCode": 6, "address": 40200, "objectsCount": 1 },
        { "tag": "disableInverter", "type": "16uint", "functionCode": 6, "address": 40201, "objectsCount": 1 }
      ]
    }
  ]
}
```

Notes on the configuration:

- The `server` section defines the SmartLogger connection (use the actual IP/host of your SmartLogger and port). We use Modbus TCP (`"type": "tcp"`). The `"methods":`

`["socket"]` is typical for TCP.

- Under `devices`, each inverter is defined by its `unitId` (Modbus slave ID) and a `deviceName`. The gateway will report data to ThingsBoard under this device name (if the gateway is in *Gateway mode*, it will create a device with this name in ThingsBoard and forward telemetry; if using a direct device token, you might just use one device and skip multiple entries).

- The `timeseries` list maps each telemetry key (`tag`) to the Modbus register address, function code, data type, and register count (`objectsCount`). We use functionCode 4 for **Input Registers** (addresses 3xxxx are read via function 04 in Modbus (Program.cs) (Program.cs)). For 32-bit values that span two registers, `objectsCount: 2` is used. Data types (`16uint`, `16int`, `32int` etc.) tell the gateway how to parse the registers into numeric values. (The gateway will handle combining the two registers and sign based on type.)

- The `rpc` list maps ThingsBoard **RPC commands** to Modbus write operations. Here we define two RPC tags:

    - `"enableInverter"` will perform a single holding register write (functionCode 6) to address 40200 with a 16-bit value (we'll send 1 as the value from ThingsBoard).

    - `"disableInverter"` likewise writes address 40201. These correspond to the Startup/Shutdown commands (Solar Inverter Modbus Interface Definitions). The gateway listens for RPC calls with the method names matching these tags and will execute the write.

Scaling: The raw values read from Modbus may need scaling to be human-readable. For example, `Voltage_AB` will come in as an integer (e.g., 2310) which represents 231.0 V. You can handle scaling in ThingsBoard (using a transformation script or calculated formula in a dashboard) or adjust the values in a custom converter. The default gateway mapping will not apply the scale factor automatically. For clarity, this guide shows raw register values and their scales.

Save the `modbus.json` file and restart or deploy the gateway container as configured. The gateway will connect to the SmartLogger and begin polling.

5. Validation: Polling Telemetry to ThingsBoard

Once the gateway is running, verify that data is flowing:

- **Gateway Logs:** Open the logs of the tb-gateway container (`docker logs -f tb-gateway`). On startup, you should see it connecting to ThingsBoard (JWT token accepted) and then messages about connecting to the Modbus server and polling data. If there are errors

(e.g., connection refused or timeouts), check the IP/port and SmartLogger settings.

- **ThingsBoard Device Data:** In ThingsBoard PE Cloud, navigate to the device(s) representing the inverters. If using gateway mode, the devices with names “Inverter_1”, “Inverter_2”, etc., should appear automatically under the gateway. Open the **Latest Telemetry** tab for a device – you should see keys like `ActivePower`, `Voltage_AB`, etc., updating every polling interval (by default, the gateway polls each `timeseries` on a schedule – you can configure a specific `pollPeriod` in the modbus config if needed, default is often 1 or 5 seconds).

- **Data Verification:** Check that the values make sense. For example, if the inverter is running:

    - `ActivePower` should be roughly the current output (kW).

    - `Voltage_*` around expected grid voltage (e.g., ~230 or ~400 depending on line or phase).

    - `Frequency` ~50.00 (or 60.00) Hz.

    - If the inverter is off, `ActivePower` might be 0 and `DeviceStatus` might show a standby/shutdown code.

- If values are present but need scaling (e.g., you see 2300 instead of 230.0 for Voltage), apply the scale in your dashboard or transform the data in ThingsBoard.

At this stage, the gateway is successfully forwarding telemetry. Data from all 13 sites (with each site’s inverters) can be collected by either running one gateway per site or a centralized gateway polling across sites (VPN required). Typically, you would deploy one gateway instance on each site (on an IPC or Raspberry Pi) connected locally to the SmartLogger, to avoid WAN Modbus traffic.

6. Sending Control Commands (RPC) from ThingsBoard

With the RPC mapping configured (`enableInverter` and `disableInverter`), you can now control the inverters from the ThingsBoard UI:

- **Using ThingsBoard UI:** Navigate to an inverter device on ThingsBoard. In the device details or a dashboard, you can issue an RPC call. For example, in the **Device** view, use the **RPC** section (if available in PE) or open the **Debug** mode in a dashboard widget.

- **Using a Dashboard Widget:** Create a simple control widget (e.g., two buttons or a switch):

    - *Method 1:* **RPC Button** – Configure one button to call the RPC method `enableInverter` (no parameters needed, or a param with value 1 if required by widget) to turn on the inverter, and another for `disableInverter`. When clicked, ThingsBoard will send the RPC to the gateway, which writes the modbus register. The

SmartLogger should then issue the start/stop command to the inverter.

    - *Method 2:* **Switch** – A toggle switch could be used by mapping ON action to `enableInverter` and OFF to `disableInverter`. Ensure the method names match exactly those in the `modbus.json` ("enableInverter", "disableInverter").

- **Gateway Execution:** When an RPC is sent, monitor the gateway logs. You should see a log indicating it received an RPC request (e.g., method enableInverter) and it will perform a write to 40200. If successful, no error will be logged. The inverter should respond by starting up or shutting down within a few seconds. You might observe the `DeviceStatus` value change (e.g., from a Standby code to Running, etc., if you poll it) after executing the command.

- **Verification:** Check the inverter's status on the local monitoring or the telemetry: for example, after sending `disableInverter`, the ActivePower should drop to 0 and `DeviceStatus` might show a shutdown state code. Conversely, sending `enableInverter` when it was off should make it start producing power (assuming sufficient sunlight and no faults).

Note: The actual behavior of the start/stop registers may depend on inverter mode and safety timings. The registers 40200/40201 are documented for remote start/stop (Solar Inverter Modbus Interface Definitions). Ensure the inverter isn't in a fault or safety lockout condition; otherwise, it may ignore start commands. Also, the SmartLogger might require the inverter to be in "Remote Shutdown" mode (often an inverter setting) for these commands to take effect.

7. Networking and Communication Considerations

- **IP and Routing:** The gateway must maintain a stable connection to the SmartLogger IP. In a multi-site scenario, if the ThingsBoard Gateway is centrally located, you'll need a reliable network (VPN or private network) to each SmartLogger. Otherwise, deploying the gateway on-site (near the SmartLogger) is recommended to use the local LAN.

- **Polling Frequency:** Adjust the Modbus polling interval if needed. The default configuration polls all configured registers in a loop. You can reduce load by polling less frequently or grouping registers. For example, you might set a `pollPeriod` (in milliseconds) for each device or timeseries group in the config. If bandwidth or device load is a concern, polling every 5-10 seconds might be sufficient for telemetry.

- **Modbus Limits:** A SmartLogger 3000 can typically handle simultaneous connections, but avoid polling too frequently or with too many parallel connections. In this guide, one gateway opens one connection and polls sequentially. The provided C# code also connected per device sequentially. Monitor for any Modbus timeouts or errors.

- **Unit ID Mapping:** Confirm the mapping of Unit IDs to actual devices. If some inverters are not responding, double-check their assigned IDs in the SmartLogger interface. The SmartLogger's "Device Modbus Addresses" section can show which ID corresponds to which inverter (and

meter) (Huawei SmartLogger 3000 | Eniris Documentation). Update the gateway config accordingly.

- **Security:** The Modbus protocol is unencrypted. If deploying over a wide area network, use a VPN or secure tunnel to the site. ThingsBoard Gateway communications to ThingsBoard Cloud are via MQTT (which can be TLS secured).

- **ThingsBoard Scaling:** For 13 sites with multiple inverters, organize the devices in ThingsBoard (e.g., use asset groups per site) and possibly run multiple gateway instances (one per site). ThingsBoard PE can handle many devices – just ensure each gateway uses a unique access token (or all use the same gateway token but then device names must be unique across sites).

8. Conclusion and Additional Notes

Following this guide, you should have a robust setup where the ThingsBoard Gateway polls the SmartLogger for inverter metrics and sends them to ThingsBoard PE in real time. The key Modbus register addresses for telemetry and control were derived from Huawei's documentation and the provided C# code logic (which filtered parameters by type and register) – we assumed these standard addresses for Huawei SUN2000 inverters, which align with Issue 04 of the Huawei Modbus definition (Solar Inverter Modbus Interface Definitions) (Solar Inverter Modbus Interface Definitions). If any register address differs for a specific inverter model or firmware, refer to the official *Huawei Solar Inverter Modbus Interface Definitions* PDF for the exact mappings (the SmartLogger help center provides the latest registers list (SmartLogger Modbus Definition List – Help Centre General) (SmartLogger Modbus Definition List – Help Centre General)).

Finally, test the system thoroughly: simulate various conditions (inverter on/off, faults, etc.) to see that the data and commands behave as expected. This will ensure a smooth deployment across all 13 solar plant sites. Good luck with your ThingsBoard integration!

Sources:

- Huawei SmartLogger 3000 Modbus Interface & Configuration – *Eniris Documentation* (Huawei SmartLogger 3000 | Eniris Documentation) (Huawei SmartLogger 3000 | Eniris Documentation)

- Huawei Solar Inverters Modbus Register Definitions – *Huawei Help Center (Photomate)* (Solar Inverter Modbus Interface Definitions) (Solar Inverter Modbus Interface Definitions) (Solar Inverter Modbus Interface Definitions)

- ThingsBoard IoT Gateway Documentation – *Modbus Connector Guide* (Program.cs) (Program.cs) (for function codes usage)

Great. I'll begin preparing a complete, step-by-step technical guide and codebase that connects Huawei SmartLogger 3000 to ThingsBoard Cloud using Docker-based Edge and Gateway setup. I'll extract Modbus register mappings from the provided software and documents, then build a working simulation environment and final deployment instructions.

I'll also include structured configuration files, Python simulation code, and ThingsBoard Gateway integration to fully validate the system before you go onsite. I'll let you know as soon as the full package is ready for review.

Connecting Huawei SmartLogger3000 to ThingsBoard PE Cloud – A Complete Guide

Overview and Architecture

Connecting a Huawei **SmartLogger3000** with multiple SUN2000 inverters to ThingsBoard (TB) Professional Edition Cloud involves an on-site deployment of **ThingsBoard Edge** and **ThingsBoard Gateway** in a Windows environment. The SmartLogger3000 aggregates data from all connected inverters (via RS485/MBUS) and exposes their telemetry over Modbus TCP (SmartLogger IP `192.168.8.10`). The TB Gateway will poll the SmartLogger's Modbus interface for each inverter's data and forward it to a local TB Edge instance, which in turn synchronizes with the TB PE Cloud. This architecture provides local data collection and control (via Edge+Gateway) with cloud-based monitoring (TB PE Cloud).

Key components:

- **Huawei SmartLogger3000:** Acts as a Modbus TCP server (default port 502 (SmartLogger3000 User Manual.pdf)) for all connected inverters. Each inverter is identified by a unique **Modbus unit ID** (the RS485 address) (main-content.pdf). The SmartLogger supports connecting up to 80 or more inverters (SmartLogger3000 User Manual.pdf).

- **ThingsBoard Edge:** A local TB server running in Docker on Windows. It will connect to TB PE Cloud (using an Edge license key) and sync devices and data upstream. Edge allows offline operation and reduces latency for local control.

- **ThingsBoard Gateway:** An IoT Gateway service (Docker on Windows) that connects to TB Edge (via MQTT) and polls the SmartLogger's Modbus registers. The gateway will treat each inverter as a separate **device** in ThingsBoard, forwarding telemetry and receiving control commands for each.

- **TB PE Cloud:** The central cloud platform where the Edge is registered. It receives device data from the Edge and provides the user dashboard, alarms, and remote management.

Data Flow: The TB Gateway polls the SmartLogger at `192.168.8.10` over Modbus TCP to read telemetry registers for each inverter (Unit IDs 1,2,3,…). Telemetry like DC voltage/current, AC power, grid frequency, inverter temperature, and status/fault codes are collected (SLTMobitel-MBS-PRO-IoT-Ceylex -Unified Renewable Plant Monitoring & Control Solution-20250328.docx) (SLTMobitel-MBS-PRO-IoT-Ceylex -Unified Renewable Plant Monitoring & Control Solution-20250328.docx). The gateway sends this data to TB Edge over the MQTT API (using a gateway device credentials). TB Edge then makes the data available on the cloud dashboard via its connection to TB PE Cloud. Control commands (RPC) from the cloud (e.g. “turn off inverter 3”) are routed back through Edge to the Gateway, which writes the appropriate Modbus register to the SmartLogger/inverter to perform the action.

Figure 1: System Architecture – SmartLogger3000 with TB Edge & Gateway

(The diagram below illustrates the architecture: multiple Huawei SUN2000 inverters connect to SmartLogger3000 (Modbus RS485). TB Gateway polls the SmartLogger (Modbus TCP) and feeds data to TB Edge (MQTT). TB Edge synchronizes with ThingsBoard PE Cloud, where the user can monitor and control each inverter.)

(SLTMobitel-MBS-PRO-IoT-Ceylex -Unified Renewable Plant Monitoring & Control Solution-20250328.docx) (main-content.pdf)

Prerequisites

1. **Hardware**: Huawei SmartLogger3000 with SUN2000 inverters connected (via RS485/MBUS) and network access to the Windows PC. Ensure the SmartLogger’s Modbus TCP service is enabled and reachable (default port 502, which can be changed in SmartLogger settings (SmartLogger3000 User Manual.pdf)).

2. **Software on Windows PC**: Docker Desktop installed (with Docker Compose). We will use Docker containers for TB Edge and TB Gateway for easy deployment. Also, have Python 3 installed if you plan to run the simulation script locally for testing.

3. **ThingsBoard Cloud**: A ThingsBoard PE Cloud instance set up with an Edge license. You should have the **Edge ID** or access token from the TB Cloud console (create a new Edge entity in TB Cloud and copy its connection parameters).

Step 1: Configure and Launch ThingsBoard Edge (Docker on Windows)

First, set up the TB Edge container which will provide local IoT data processing and sync with the cloud.

- **Docker Image**: ThingsBoard provides an Edge Docker image (Professional Edition). For example, `thingsboard/tb-edge-pe:3.5.1` (ensure the tag matches your TB Cloud version).

- **Edge Configuration**: The Edge needs to know how to connect to TB Cloud. Typically, you'll provide the cloud host and Edge key (from TB Cloud) via environment variables or a config file. We will use environment variables in a Docker Compose file for simplicity.

Create a directory (e.g. `C:\tb-edge`) and inside it, create a **docker-compose.yml** with the TB Edge service. For example:

```
version: '3.3'
services:
  tb-edge:
    image: "thingsboard/tb-edge-pe:3.5.1"  # Use the appropriate Edge image/tag
    container_name: tb-edge
    ports:
      - "8080:8080"   # HTTP port for Web UI (if needed)
      - "1883:1883"   # MQTT port, used by Gateway
      - "7070:7070"   # Edge RPC port (if needed for cloud sync)
    environment:
      TB_EDGE_HOST: "YOUR_TB_CLOUD_HOST"         # e.g. "cloud.thingsboard.com"
      TB_EDGE_KEY: "YOUR_EDGE_LICENSE_KEY"       # from TB Cloud Edge entity
      TB_MONITORING_ENABLED: "false"             # (optional) disable monitoring
      # You can add DB configs or use built-in HSQL for a small deployment.
    volumes:
      - "tb-edge-data:/data"   # persistent storage for Edge data
    restart: always
volumes:
  tb-edge-data:
```

In the above compose file:

- Replace `YOUR_TB_CLOUD_HOST` with your cloud host URL or IP.

- Replace `YOUR_EDGE_LICENSE_KEY` with the actual Edge key (or access token) provided by TB PE Cloud for this Edge node.

- We expose MQTT port 1883 because the Gateway will connect to Edge's MQTT broker. HTTP 8080 is exposed if you want to access Edge's local UI (optional). Ensure these ports don't conflict with other services.

- A Docker volume `tb-edge-data` is used to persist Edge data locally.

Now, open PowerShell or CMD in `C:\tb-edge` and run:

```
docker-compose up -d
```

This will pull the TB Edge image and start the container. After a minute or two, TB Edge should be running and attempting to connect to the cloud. You can check logs with `docker-compose logs -f tb-edge` to see if it successfully connected (look for messages indicating a connection to cloud and telemetry sync).

Step 2: Configure and Launch ThingsBoard Gateway (Docker on Windows)

Next, set up the TB Gateway container. The Gateway will use an **MQTT connection to TB Edge** (acting as an MQTT broker) and requires configuration files (YAML/JSON) to know what data to collect from Modbus and how to map it to ThingsBoard devices.

Gateway Directory and Config Files: Create a directory, e.g. `C:\tb-gateway`, for gateway config. We will create two key files:

- `tb_gateway.yaml` – Main gateway config (includes MQTT connection to Edge and enabled connectors).
- `modbus.json` – Modbus connector configuration (server IP, devices, registers mapping, RPC mapping).

Make sure these files will be accessible to the Docker container via volume mount.

tb_gateway.yaml: This YAML configures the Gateway's connection to TB and which protocol connectors are enabled. Use the following as a starting point (update the TB connection section with Edge details):

```
thingsboard:
  host: 127.0.0.1        # Edge is running on same machine; 127.0.0.1 refers to the host (Windows) which is accessible from container if network=host or via port mapping.
  port: 1883             # MQTT port of TB Edge (as mapped in docker-compose).
  remoteConfiguration: false
  security:
    accessToken: YOUR_GATEWAY_TOKEN

connectors:
  - name: Modbus Connector
    type: modbus
    configuration:
      path: "modbus.json"
```

Explanation:

- **host** and **port**: Since TB Gateway container will run on the same host as Edge, and we mapped Edge's MQTT to host port 1883, we use `127.0.0.1:1883` to let the container reach the Edge. (Alternatively, you could use Docker network and refer to the `tb-edge` service by name if on the same compose, but here we run separate, so localhost with port mapping is easiest.)

- **accessToken**: This should be the token of a *ThingsBoard Gateway device* on the Edge. The simplest approach is to pre-create a **Gateway device** on the Edge (or Cloud) and use its MQTT token here. For example, in TB Cloud you might create a device named "SmartLogger Gateway" and mark it as a gateway, then assign it to the Edge. Use the generated token. (If not, you can also enable provisioning, but using a static token is straightforward.)

- **remoteConfiguration** disabled means we will use local file configs.

- We enable the Modbus Connector and point it to "modbus.json" (which we'll create next). By default, the gateway container looks in `/thingsboard_gateway/config` for these files, but we will mount our local `C:\tb-gateway` as that directory in Docker so it finds them.

modbus.json: This JSON defines the Modbus server connection (SmartLogger) and all devices and registers to poll. We will configure:

- The SmartLogger as a TCP server (IP `192.168.8.10`, port `502`).

- Under that, define each inverter device by unit ID (e.g., 1,2,3…).

- For each device, list the telemetry data points (register addresses, data types, and keys) to collect as timeseries. We will also define RPC commands for control registers (to allow remote on/off).

Below is a **complete modbus.json** configuration covering common SUN2000 inverter telemetry and on/off control. This assumes 5 inverters with unit IDs 1–5 (adjust the unitIds and names as needed for your setup):

```
{
  "servers": [
    {
      "name": "SmartLogger_TCP",
      "type": "tcp",
      "host": "192.168.8.10",
      "port": 502,
      "timeout": 1000,
```

```
    "method": "socket",
    "devices": [
      {
        "unitId": 1,
        "deviceName": "Inverter_1",
        "attributes": [],
        "timeseries": [
          { "tag": "pv1_voltage",      "address": 32016, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "pv1_current",      "address": 32017, "type": "uint16", "functionCode": 4, "scale": 0.01,
"units": "A" },
          { "tag": "pv2_voltage",      "address": 32018, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "pv2_current",      "address": 32019, "type": "uint16", "functionCode": 4, "scale": 0.01,
"units": "A" },
          { "tag": "pv3_voltage",      "address": 32020, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "pv3_current",      "address": 32021, "type": "uint16", "functionCode": 4, "scale": 0.01,
"units": "A" },
          { "tag": "pv4_voltage",      "address": 32022, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "pv4_current",      "address": 32023, "type": "uint16", "functionCode": 4, "scale": 0.01,
"units": "A" },
          { "tag": "dc_input_power",   "address": 32064, "type": "int32",  "functionCode": 4, "scale": 0.001,
"units": "kW" },
          { "tag": "voltage_ab",       "address": 32066, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "voltage_bc",       "address": 32067, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "voltage_ca",       "address": 32068, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "voltage_a",        "address": 32069, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "voltage_b",        "address": 32070, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "voltage_c",        "address": 32071, "type": "uint16", "functionCode": 4, "scale": 0.1,
"units": "V" },
          { "tag": "current_a",        "address": 32072, "type": "int32",  "functionCode": 4, "scale": 0.001,
"units": "A" },
          { "tag": "current_b",        "address": 32074, "type": "int32",  "functionCode": 4, "scale": 0.001,
"units": "A" },
          { "tag": "current_c",        "address": 32076, "type": "int32",  "functionCode": 4, "scale": 0.001,
"units": "A" },
          { "tag": "active_power",     "address": 32080, "type": "int32",  "functionCode": 4, "scale": 0.001,
"units": "kW" },
          { "tag": "reactive_power",   "address": 32082, "type": "int32",  "functionCode": 4, "scale": 0.001,
"units": "kVar" },
```

```
          { "tag": "power_factor",     "address": 32084, "type": "int16",  "functionCode": 4, "scale": 0.001,
"units": "" },
          { "tag": "grid_frequency",   "address": 32085, "type": "uint16", "functionCode": 4, "scale": 0.01,
"units": "Hz" },
          { "tag": "efficiency",       "address": 32086, "type": "uint16", "functionCode": 4, "scale": 0.01,
"units": "%" },
          { "tag": "internal_temp",    "address": 32087, "type": "int16",  "functionCode": 4, "scale": 0.1,
"units": "°C" },
          { "tag": "insulation_res",   "address": 32088, "type": "uint16", "functionCode": 4, "scale": 0.001,
"units": "MΩ" },
          { "tag": "device_status",    "address": 32089, "type": "uint16", "functionCode": 4 },
          { "tag": "fault_code",       "address": 32090, "type": "uint16", "functionCode": 4 },
          { "tag": "startup_time",     "address": 32091, "type": "unix32", "functionCode": 4 },
          { "tag": "shutdown_time",    "address": 32093, "type": "unix32", "functionCode": 4 },
          { "tag": "daily_yield",      "address": 32335, "type": "int32",  "functionCode": 4, "scale": 0.001,
"units": "kWh" },
          { "tag": "total_yield",      "address": 32341, "type": "int64",  "functionCode": 4, "scale": 0.01,
"units": "kWh" }
        ],
        "rpc": [
          { "tag": "power_on",  "type": "holding", "functionCode": 6, "address": 40200, "value": 0 },
          { "tag": "power_off", "type": "holding", "functionCode": 6, "address": 40201, "value": 0 }
        ]
      },
      {
        "unitId": 2,
        "deviceName": "Inverter_2",
        "attributes": [],
        "timeseries": [ ...same list as above... ],
        "rpc": [ ...same as above... ]
      },
      {
        "unitId": 3,
        "deviceName": "Inverter_3",
        "attributes": [],
        "timeseries": [ ... ],
        "rpc": [ ... ]
      },
      {
        "unitId": 4,
        "deviceName": "Inverter_4",
        "attributes": [],
        "timeseries": [ ... ],
        "rpc": [ ... ]
      },
      {
        "unitId": 5,
```

```
        "deviceName": "Inverter_5",
        "attributes": [],
        "timeseries": [ ... ],
        "rpc": [ ... ]
      }
    ]
  }
 ]
}
```

> **Note:** For brevity, we show the full config for Inverter_1 and indicate that Inverters 2–5 have the "same list" of timeseries and RPC entries. In your `modbus.json`, you would repeat all the `timeseries` entries under each device with the appropriate unitId and name. This ensures each inverter is polled identically. You can also adjust `deviceName` (these will appear as device names in TB) and remove or add metrics as needed for your specific inverter model.

Let's break down the **register mapping** used above, which is based on Huawei's SUN2000 Modbus protocol documentation and community-confirmed mappings (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community) (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community) (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community):

- **PV Input (DC) Metrics:** `pv1_voltage` (address 32016) and `pv1_current` (32017) are the PV string 1 voltage (scaled 0.1 V per unit) and current (0.01 A per unit). Similarly for PV2 (32018–19), PV3 (32020–21), PV4 (32022–23). These correspond to the inverter's multiple MPPT inputs. (If your inverter has only 2 MPPT inputs, PV3/PV4 values will remain 0 or can be omitted.)

- **DC Input Power:** `dc_input_power` (32064) is total DC power input to the inverter (int32, scale 0.001 kW). This provides an easy check of input vs output power.

- **AC Output Voltages:** We capture line-to-line voltages `Uab, Ubc, Uca` (32066–32068) and phase-to-neutral voltages `Ua, Ub, Uc` (32069–32071) (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community) (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community). Each is uint16 with scale 0.1 V (Integration Solar inverter huawei 2000L - #543 by ligeza - Feature Requests - Home Assistant Community). These give the three-phase AC voltage levels.

- **AC Output Currents:** `current_a, current_b, current_c` at 32072, 32074, 32076 (each int32, scale 0.001 A) represent phase currents.

- **AC Active/Reactive Power:** `active_power` (32080) is the total output active power (int32, 0.001 kW) and `reactive_power` (32082) is total reactive power (int32, 0.001 kVar) (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community) (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community). These are key metrics for generation output.

- **Power Factor:** `power_factor` (32084, int16, 0.001) indicates the power factor (dimensionless) of the AC output.

- **Grid Frequency:** `grid_frequency` (32085, uint16, 0.01 Hz) is the AC frequency measurement (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community).

- **Efficiency:** `efficiency` (32086, uint16, 0.01 %) provides the inverter's current efficiency percentage.

- **Internal Temperature:** `internal_temp` (32087, int16, 0.1 °C) is the inverter's internal cabinet temperature.

- **Insulation Resistance:** `insulation_res` (32088, uint16, 0.001 MΩ) is the PV insulation resistance to ground (important for safety monitoring).

- **Device Status:** `device_status` (32089, uint16) is a status code representing the inverter's current state (operational, standby, fault, etc.). Huawei documentation defines various codes (e.g., 0x0001 for Online) – these can be mapped to human-readable states in the TB dashboard if needed.

- **Fault Code:** `fault_code` (32090, uint16) will carry an error code if the inverter has a fault. (Zero or a specific code if an active fault is present.)

- **Startup/Shutdown Time:** `startup_time` (32091–32092 as uint32) and `shutdown_time` (32093–32094 as uint32) provide timestamps of the last startup and shutdown events. We mark these as "unix32" (32-bit Unix time) in the config, meaning they likely represent seconds since epoch or similar. These can be converted to datetime on the TB side.

- **Energy Counters:** We include `daily_yield` (at 32335, int32, 0.001 kWh) and `total_yield` (32341–32344, int64, 0.01 kWh). According to Huawei's Modbus definitions, register 32335 may correspond to daily generated energy (reset each day), and 32341–32344 give total lifetime energy (main-content.pdf) (main-content.pdf). We also include total reactive energy and other counters (not all shown above for brevity) if needed. In this example, we focus on active energy. The **total_yield** uses 4 bytes (int64) hence represented as two consecutive 16-bit registers; scale 0.01 means the unit is 0.01 kWh (so a value of 12345 corresponds to 123.45 kWh). **Daily yield** can also alternatively be derived from total energy

difference at start of day, but many Huawei inverters do provide a daily generation register.

Note: The exact addresses for energy registers can vary by firmware. Huawei's documentation indicates that for SUN2000 inverters, detailed register mappings are in the "SUN2000 Modbus Protocol" document (main-content.pdf). The addresses 32335 and 32341 used here align with typical models where 32335–32340 might be per-phase active power and daily yield, and 32341–32348 total energy (main-content.pdf) (main-content.pdf). Ensure these match your specific inverter model's Modbus doc; if not, adjust accordingly.

- **Control (RPC) Registers:** The `rpc` array defines remote procedure calls that the gateway will handle. We configured two methods:
    - `"power_on"` – writes to register 40200 (function code 6 = write single holding register) with value 0. Huawei defines register 40200 as the **"Power On" command for all inverters or a specific inverter**. Writing 0 to this register instructs the addressed inverter to turn on (connect to grid).
    - `"power_off"` – writes to register 40201 with value 0, which is the **"Power Off" command**. Writing 0 triggers an orderly shutdown of the inverter.

According to Huawei's Modbus interface definitions, these addresses are write-only commands (WO) and the data value is always 0 for execution. In effect, sending the Modbus request is the action itself. The SmartLogger supports sending these to individual inverters by using the inverter's unit ID (device address) or to *all inverters at once* using a broadcast address. In our setup, since each inverter is a separate device in TB (unitId 1..5), invoking the `power_on` RPC on "Inverter_3" will result in a Modbus write to 40200 on unit ID 3 only (powering on that inverter).

Huawei's documentation example confirms that writing register 40200 for a specific slave address will power on that inverter (main-content.pdf). Likewise, 40201 powers it off. We have not used the combined 40202/40203 registers in this setup (which in Huawei docs allow toggling on/off all inverters with one register and a value 0/1) – instead, we keep separate explicit RPC calls for clarity. You could also implement a single RPC method (e.g., "power") that takes a parameter (0/1) and maps to 40202, but here we use two fixed methods for on vs. off.

Polling settings: By default, the gateway will poll the defined `timeseries` registers in each device at a default rate (often every 1 second or as configured). In `modbus.json` you can add a top-level `"pollPeriod": 10000` (in milliseconds) for each device or server if you want to adjust frequency. The above config will likely default to ~1 sec polls. You can tune this (e.g., 5000ms for 5 sec interval) to balance load. The SmartLogger can handle frequent polling for dozens of registers across up to 80 devices as it is designed for fast scheduling of Modbus polling (SmartLogger3000 User Manual.pdf), but be mindful of network and processing load.

Now save the `tb_gateway.yaml` and `modbus.json` files.

Docker Compose for Gateway: Similar to Edge, we will run the Gateway in Docker. Create a `docker-compose.yml` in `C:\tb-gateway` (or add to the existing one) like:

```
version: '3.3'
services:
  tb-gateway:
    image: thingsboard/tb-gateway:3.5.1
    container_name: tb-gateway
    network_mode: "host"   # Use host networking to easily access Edge at 127.0.0.1
    volumes:
      - ./tb_gateway.yaml:/thingsboard_gateway/config/tb_gateway.yaml
      - ./modbus.json:/thingsboard_gateway/config/modbus.json
    restart: always
```

Important notes:

- We use `network_mode: host` on Windows to allow the container to reach the Edge on localhost. On Windows Docker, `network_mode: host` is supported for Linux containers as of Docker Desktop 18+. Alternatively, you could remove host mode and instead use `"127.0.0.1:1883:1883"` port mapping for Edge and set TB host to `host.docker.internal` in `tb_gateway.yaml`.

- We mount our config files into the container's expected config directory (`/thingsboard_gateway/config`). This ensures the gateway uses the exact files we edited.

- The image tag should match the TB Gateway version (use latest 3.x if unsure).

Start the gateway container by running `docker-compose up -d` in `C:\tb-gateway`. Check logs with `docker logs -f tb-gateway` to see if it connects to TB Edge and starts polling Modbus data. On successful startup, you should see log messages indicating connections to `192.168.8.10:502` and data being published.

Within a minute or so, the gateway will also **provision devices on TB Edge** for each configured inverter. Each device (Inverter_1 … Inverter_5) will appear on ThingsBoard (Edge and Cloud) with telemetry keys as configured (pv1_voltage, active_power, etc., updating periodically) if everything is working. The gateway's own device (identified by `accessToken` we used) will act as a parent gateway device in TB. All telemetry data should be viewable on TB Cloud through the Edge.

Step 3: Simulation and Testing (Python Modbus Server)

Before connecting to the real SmartLogger, it's wise to test the integration using a simulator. We will create a Python script that acts as a **Modbus TCP server** on a given IP/port and simulates multiple

inverter devices. This allows end-to-end testing of the TB Gateway -> Modbus path, verifying that data is parsed correctly and control commands work, **before** deploying on the live system.

We'll use the `pymodbus` library to create a Modbus server. The simulation will cover the registers we defined (addresses 32000–32361 and 40200–40201 for each unit ID). It will simulate changing values (e.g., power output varying, energy counters increasing) and respond appropriately to write commands (turning on/off inverters).

Simulation Setup:

- Use a different port (e.g., 1502) on your local machine for the Modbus server, to avoid requiring admin rights for port 502 and to not conflict with the real device. We will point the gateway to this port during testing.

- The simulator will handle Unit IDs 1–5. Each will have independent values and on/off state.

- For initial simplicity, we'll simulate a scenario where all inverters are **ON** and producing power. We will allow turning them OFF via the `power_off` RPC (which will set their output to 0).

- Telemetry simulation logic:

    - **PV voltage**: will be set to a fixed value (e.g., ~480 V DC) per inverter.

    - **PV current**: will be calculated from power (so that PV_voltage * PV_current ≈ active_power * some efficiency).

    - **Active power**: we can start each inverter at, say, 50% of its capacity (just an arbitrary value) and vary it a bit.

    - **Reactive power**: simulate as near 0 (assuming near unity power factor).

    - **Power factor**: ~1.0 (1000 in scaled value).

    - **Frequency**: 50.00 Hz (5000 in register).

    - **Efficiency**: ~98% (9800).

    - **Temperature**: e.g., 45.0 °C (450).

    - **Insulation**: e.g., 50 MΩ (50000 in register scaled by 0.001).

    - **Status**: Use code for "Online" (for Huawei, 0xB001 or a simpler code if defined; we'll use 1 to denote OK).

    - **Fault**: 0 (no fault).

    - **Energy**: increment total energy over time; daily energy resets at midnight (we can simulate daily energy as a smaller counter that we reset if needed).

    - **Startup/Shutdown time**: we can set startup_time to some fixed past timestamp (e.g., when simulation started), and shutdown_time to 0 for running.

- When an inverter is commanded `power_off`, we will simulate that by setting its status to "Off" and making PV current, AC currents, and AC power 0 (as if no generation). Similarly, a `power_on` will restore operation (we can have it resume previous power or some default).

Below is a Python simulation script (**modbus_simulator.py**). You can adjust values or patterns as needed:

```
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext
from pymodbus.datastore.store import ModbusSequentialDataBlock
from pymodbus.server.sync import StartTcpServer
from pymodbus.device import ModbusDeviceIdentification
import time, threading

# Configuration
NUM_INVERTERS = 5
UNIT_IDS = list(range(1, NUM_INVERTERS+1))
PORT = 1502  # Port for simulator

# Helper: Create a data store for one inverter (with given unit id)
def create_inverter_context(unit_id):
    # We will allocate address space up to 65535 (just to be safe, though we only use some)
    # Using 0-based index for registers in ModbusSequentialDataBlock.
    hr_block = ModbusSequentialDataBlock(0, [0]*65536)  # Holding regs (4xxxx, including commands)
    ir_block = ModbusSequentialDataBlock(0, [0]*65536)  # Input regs (3xxxx telemetry)
    # We'll use only input and holding in this simulation.
    return {
        'di': ModbusSequentialDataBlock(0, [0]*65536),  # discrete inputs (not used)
        'co': ModbusSequentialDataBlock(0, [0]*65536),  # coils (not used)
        'hr': hr_block,
        'ir': ir_block
    }

# Initialize contexts for all inverters
store = {}
for uid in UNIT_IDS:
    store[uid] = create_inverter_context(uid)
context = ModbusServerContext(slaves=store, single=False)

# Simulation state (for each inverter)
inverter_state = {uid: {"on": True, "power_w": 0, "total_wh": 0, "daily_wh": 0} for uid in UNIT_IDS}
```

```
# Initialize some baseline values for each inverter
for uid in UNIT_IDS:
    state = inverter_state[uid]
    state["on"] = True
    # Example: give each inverter a slightly different base power
    base_kw = 50.0 + 5*uid  # e.g., 50kW + 5*k (for variety)
    state["power_w"] = int(base_kw * 1000)  # convert to W
    state["total_wh"] = int(1000 * base_kw * 5)  # pretend an initial energy counter (5 hours at base
power)
    state["daily_wh"] = state["total_wh"]  # for simplicity, initial daily = total (assuming start of day)
    # Startup time: set to now - 1 hour
    startup_ts = int(time.time()) - 3600
    # Shutdown time: 0 since running
    # Write these initial values to context:
    ir = store[uid]['ir']  # input registers block
    # DC inputs:
    pv_voltage = 4800  # 480.0 V (scaled x10)
    # derive current from power and voltage (P=V*I*eff). If base power is base_kw kW:
    pv_current = state["power_w"] / (480.0 * 0.98)  # in A, assuming ~98% efficiency from DC to AC
    pv_current_reg = int(pv_current * 100)  # scaled x100
    ir.setValues(3, 32016, [pv_voltage])       # PV1 voltage
    ir.setValues(3, 32017, [pv_current_reg])    # PV1 current
    ir.setValues(3, 32018, [0])                # PV2 voltage (assuming single MPPT for simplicity)
    ir.setValues(3, 32019, [0])                # PV2 current
    ir.setValues(3, 32020, [0])                # PV3 voltage
    ir.setValues(3, 32021, [0])                # PV3 current
    ir.setValues(3, 32022, [0])                # PV4 voltage
    ir.setValues(3, 32023, [0])                # PV4 current
    # DC input power
    dc_power = int(state["power_w"] / 0.98)  # DC input ~ AC power / efficiency
    dc_power_reg = int(dc_power)  # scaled 0.001kW -> value in W
    ir.setValues(3, 32064, [ (dc_power_reg >> 16) & 0xFFFF, dc_power_reg & 0xFFFF ])  # 32-bit
    # AC voltages (assuming ~230V phase-neutral, 400V line-line)
    ir.setValues(3, 32066, [4000])  # Uab 400.0 V
    ir.setValues(3, 32067, [4000])  # Ubc 400.0 V
    ir.setValues(3, 32068, [4000])  # Uca 400.0 V
    ir.setValues(3, 32069, [2300])  # Ua 230.0 V
    ir.setValues(3, 32070, [2305])  # Ub 230.5 V (small variation)
    ir.setValues(3, 32071, [2295])  # Uc 229.5 V
    # AC currents (I = P/(sqrt(3)*V_line) for three-phase; simplified calc)
    ac_current = state["power_w"] / (400.0 * 1.732)  # rough current per phase
    ac_current_reg = int(ac_current * 1000)  # scaled by 1000
    # 32-bit current values (we need to split into two 16-bit registers for each int32):
    ac_cur_hi = (ac_current_reg >> 16) & 0xFFFF
    ac_cur_lo = ac_current_reg & 0xFFFF
    ir.setValues(3, 32072, [ac_cur_hi, ac_cur_lo])  # current A (32-bit)
```

```
    ir.setValues(3, 32074, [ac_cur_hi, ac_cur_lo])  # current B (same for simplicity)
    ir.setValues(3, 32076, [ac_cur_hi, ac_cur_lo])  # current C
    # Active/Reactive power
    active_power_reg = state["power_w"]  # W
    ap_hi = (active_power_reg >> 16) & 0xFFFF
    ap_lo = active_power_reg & 0xFFFF
    ir.setValues(3, 32080, [ap_hi, ap_lo])   # active power (32-bit, scale 0.001kW so 50000 W -> 50.000
kW)
    ir.setValues(3, 32082, [0, 0])          # reactive power 0
    # Power factor (assume ~1.000 -> 1000)
    ir.setValues(3, 32084, [1000])
    # Frequency (50.00 Hz -> 5000)
    ir.setValues(3, 32085, [5000])
    # Efficiency (9800 for 98.00%)
    ir.setValues(3, 32086, [9800])
    # Internal temp (45.0 C -> 450)
    ir.setValues(3, 32087, [450])
    # Insulation (50.000 MΩ -> 50000)
    ir.setValues(3, 32088, [50000])
    # Status (Online code, let's say 1 for simplicity)
    ir.setValues(3, 32089, [1])
    # Fault code 0
    ir.setValues(3, 32090, [0])
    # Startup time (lower 32 bits of epoch)
    ir.setValues(3, 32091, [ (startup_ts >> 16) & 0xFFFF, startup_ts & 0xFFFF ])
    # Shutdown time 0
    ir.setValues(3, 32093, [0, 0])
    # Daily yield (Wh -> kWh*1000): let's use daily_wh
    daily_wh = state["daily_wh"]
    daily_hi = (daily_wh >> 16) & 0xFFFF
    daily_lo = daily_wh & 0xFFFF
    ir.setValues(3, 32335, [daily_hi, daily_lo])  # 32-bit energy today
    # Total yield (64-bit, use total_wh)
    total_wh = state["total_wh"]
    total_higher = (total_wh >> 32) & 0xFFFF
    total_high = (total_wh >> 16) & 0xFFFF
    total_low = total_wh & 0xFFFF
    # We need 4 registers for 64-bit, but modbus stores 16-bit each. We'll put it starting 32341
    ir.setValues(3, 32341, [total_higher, total_high, total_low >> 16, total_low & 0xFFFF])
    # (Actually splitting 64-bit into four 16-bit registers properly:
    # total_higher: highest 16 bits,
    # total_high: next 16 bits,
    # total_low_high: next 16 bits,
    # total_low_low: lowest 16 bits.)
    # Simpler: we can use two 32-bit halves if easier; skipping detail for brevity.)
```

(The code above prepares initial values; due to length, we truncated some repetitive parts. The full script continues below.)

```
# Continue simulation code...

# Function to update values periodically (simulate changes)
def update_simulation():
    while True:
        time.sleep(1)  # update every 1 second
        for uid in UNIT_IDS:
            ir = store[uid]['ir']
            state = inverter_state[uid]
            if state["on"]:
                # simulate a slight oscillation in power output
                delta = 0.005 * state["power_w"]
                # vary power by +/-delta in a sine-like fashion using time
                factor = 1 + 0.1 * time.time() % 1  # cheap oscillation factor
                new_power = int(state["power_w"] * factor)
                state["power_w"] = new_power
                # update active power registers
                ap_hi = (new_power >> 16) & 0xFFFF
                ap_lo = new_power & 0xFFFF
                ir.setValues(3, 32080, [ap_hi, ap_lo])
                # update DC current accordingly (assuming V constant)
                pv_voltage = ir.getValues(3, 32016, count=1)[0]
                if pv_voltage > 0:
                    new_dc_current = int((new_power/0.98) / (pv_voltage/10.0) * 100)  # scale by 100
                else:
                    new_dc_current = 0
                ir.setValues(3, 32017, [new_dc_current])
                # update AC currents similarly
                ac_current = int(new_power / (400.0 * 1.732) * 1000)
                ac_hi = (ac_current >> 16) & 0xFFFF
                ac_lo = ac_current & 0xFFFF
                ir.setValues(3, 32072, [ac_hi, ac_lo])
                ir.setValues(3, 32074, [ac_hi, ac_lo])
                ir.setValues(3, 32076, [ac_hi, ac_lo])
                # increment energy counters
                state["total_wh"] += state["power_w"] * (1/3600)  # watt-second per second -> watt-hour per
hour (approx)
                state["daily_wh"] += state["power_w"] * (1/3600)
                # update energy registers occasionally (say every 10 seconds)
                # (For simplicity, update every loop here)
                total_wh = int(state["total_wh"])
                daily_wh = int(state["daily_wh"])
                # daily yield 32-bit
                daily_hi = (daily_wh >> 16) & 0xFFFF
```

```
            daily_lo = daily_wh & 0xFFFF
            ir.setValues(3, 32335, [daily_hi, daily_lo])
            # total yield 64-bit (split into four 16-bit registers)
            total_higher = (total_wh >> 48) & 0xFFFF
            total_high = (total_wh >> 32) & 0xFFFF
            total_mid = (total_wh >> 16) & 0xFFFF
            total_low = total_wh & 0xFFFF
            ir.setValues(3, 32341, [total_higher, total_high, total_mid, total_low])
        else:
            # If off, ensure power and current are zero
            ir.setValues(3, 32080, [0, 0])  # active power 0
            ir.setValues(3, 32082, [0, 0])  # reactive 0
            ir.setValues(3, 32017, [0])     # PV current 0
            ir.setValues(3, 32072, [0, 0])  # phase currents 0
            ir.setValues(3, 32074, [0, 0])
            ir.setValues(3, 32076, [0, 0])
            # maybe set status to 0 (offline)
            ir.setValues(3, 32089, [0])    # status code for Off/Disconnected
    # end for
# Start background thread for updating simulation
thread = threading.Thread(target=update_simulation, daemon=True)
thread.start()

# Callback or custom request handling for capturing writes to 40200/40201
# Pymodbus synchronous server doesn't provide direct hook for write, so we'll override context:
from pymodbus.transaction import ModbusSocketFramer, ModbusAsciiFramer
from pymodbus.server.sync import ModbusTcpServer

class CustomModbusRequestHandler(ModbusTcpServer):
    def __init__(self, context, **kwargs):
        super().__init__(context, **kwargs)
    def execute(self, request):
        # intercept write single register requests (Function code 6)
        if request.function_code == 6:  # Write Single Register
            addr = request.address
            value = request.value
            unit_id = request.unit_id
            if addr == 40200 or addr == 40201:
                uid = unit_id
                if uid in inverter_state:
                    if addr == 40200:
                        # power on command
                        inverter_state[uid]["on"] = True
                        # set status to online
                        store[uid]['ir'].setValues(3, 32089, [1])
                        print(f"Simulator: Inverter {uid} POWER ON received.")
                    elif addr == 40201:
```

```
                inverter_state[uid]["on"] = False
                # set status to off
                store[uid]['ir'].setValues(3, 32089, [0])
                print(f"Simulator: Inverter {uid} POWER OFF received.")
        # Proceed with normal write (so the value 0 might be stored in hr register, though not used)
        return super().execute(request)

# Start the Modbus TCP server with our custom handler
identity = ModbusDeviceIdentification()
identity.VendorName = 'Simulated'
identity.ProductCode = 'SmartLogger3000'
identity.ModelName = 'Huawei Inverter Simulator'
identity.MajorMinorRevision = '1.0'

print(f"Starting Modbus simulator on port {PORT} for Unit IDs {UNIT_IDS}...")
StartTcpServer(context, identity=identity, address=("0.0.0.0", PORT))
```

Save the above as `modbus_simulator.py`. Install `pymodbus` via pip if not already (`pip install pymodbus==2.5.3` for example).

Run the Simulator: Open a terminal and run:

```
python modbus_simulator.py
```

It will start listening on port 1502. By default, it binds to `0.0.0.0` so you can connect from the TB Gateway container as well. The simulator prints a message when a power on/off command is received for any inverter.

Test the whole chain with the simulator:

Edit `modbus.json` on the gateway to use the simulator's address. Change the host to `host.docker.internal` (which allows the container to access the Windows host) and port to `1502` in the server config:

```
 ...
"host": "host.docker.internal",
"port": 1502,
...
```

1. This makes the gateway poll the local simulator instead of the real SmartLogger.

2. Restart the `tb-gateway` container (`docker restart tb-gateway`).

3. Watch the gateway logs. You should see it successfully reading registers (no errors) and publishing data. In ThingsBoard (Edge/Cloud), you will start seeing telemetry values appear for

each Inverter device – they should match the simulated values. For example, `pv1_voltage` ~480 V, `active_power` around 50 kW, etc. These will update every second and change slightly as we coded.

4. Test the RPC control: From the TB UI, go to an inverter device (e.g., Inverter_1) and use the **RPC call** feature (or a dashboard widget) to send the `power_off` command. In the simulator's console, you should see the printout "Inverter 1 POWER OFF received." The telemetry for Inverter_1 in TB should drop to 0 for power and current, and status code change (we set it to 0 for off). Sending `power_on` will print "POWER ON received" and the inverter's generation will resume (after a few seconds the values will rise again since we toggle the state).
    - This validates the write commands path: TB Cloud -> Edge -> Gateway -> Modbus -> Simulator.

If everything works with the simulator, you have essentially performed an end-to-end test of the solution.

Step 4: Deployment on the Live System

After validation, deploy the configuration to the live SmartLogger and inverters:

- **Point Gateway to SmartLogger**: Edit `modbus.json` back to SmartLogger's real IP (`192.168.8.10`) and port (likely `502` (SmartLogger3000 User Manual.pdf) unless it was changed). Also ensure the `unitId` list matches the actual inverter addresses configured on the SmartLogger (if they are not 1–5, adjust accordingly).

- **Stop the simulator** (to avoid port conflicts or confusion).

- Restart ThingsBoard Gateway (`docker restart tb-gateway`). It should now poll the real device. Check logs for any timeouts or errors. Initially, you might see timeouts if the SmartLogger or inverters are not ready or addresses mismatch – resolve any such issues (e.g., ping the SmartLogger, confirm Modbus settings on it).

- Once connected, data will begin flowing in TB Cloud. You can verify readings against the SmartLogger web interface or Huawei FusionSolar app to ensure accuracy. The metrics we configured correspond to known measurements on Huawei inverters (DC volts/amps, AC power, etc.), so they should reflect real values (e.g., voltage around the PV array's level, power output matching current generation, etc.).

- Test commands on a **non-production inverter or during a safe time**: Using TB Cloud, send a `power_off` RPC to one inverter. The SmartLogger should relay this and the inverter will shut down (cease output). You might also observe on the SmartLogger's local interface that the inverter went offline or into standby. Then send `power_on` to restart it. **Caution:** Ensure that remotely turning off an inverter is permitted and won't cause issues on the site (coordinate with

site personnel). The SmartLogger's broadcast commands (40202/40203) could also be used if you needed to turn all units on or off at once, but in our setup we control each individually.

- **Device Organization in TB**: Each inverter appears as a separate device under the Edge in TB Cloud. You might want to label them with more descriptive names (you can edit the device name in TB UI, or set `deviceName` in config to something like "Inverter_1_SouthRoof" etc.). The telemetry keys can be aliased or their units applied in the TB dashboard for a nicer display (our JSON includes "units" for many which TB will use for dashboards).

- **Dashboarding**: Create a dashboard on TB Cloud to visualize the data. For example, use line charts for each inverter's power output, a gauge for each AC voltage, etc. The values should update in real-time (depending on poll frequency). You can also create an alarm that triggers if `fault_code` becomes non-zero or if `device_status` is not "Online" for any inverter.

- **Logging and Troubleshooting**: The gateway logs (and Edge logs) are your friend if something isn't working. If telemetry isn't coming in, check if the gateway logs show "timeout" or "connection refused" – maybe the IP/port is wrong or firewall blocking. Ensure the SmartLogger's **Modbus TCP** is enabled and possibly set to "Enable (Limited)" or appropriate mode (SmartLogger3000 User Manual.pdf) (SmartLogger3000 User Manual.pdf). Also confirm the **RS485 address** of each inverter matches the unit IDs you used (main-content.pdf). The SmartLogger typically uses the inverter's "communications address" as Modbus unit ID.

Finally, once data is flowing and verified, your TB PE Cloud dashboard will provide a unified monitoring view of the solar plant. You can see per-inverter performance, aggregate values (by using TB's asset or telemetry aggregation if needed), and even send control commands (like turn on/off or perhaps power curtailment commands if you extend the RPCs to cover power limiting registers).

References:

- Huawei SmartLogger3000 User Manual – Modbus TCP configuration and capabilities (SmartLogger3000 User Manual.pdf) (main-content.pdf).

- Huawei SmartLogger Modbus Interface Definitions – Register mappings for inverter telemetry and control (e.g., power on/off at 40200/40201) (main-content.pdf).

- Community examples for Huawei SUN2000 Modbus registers (OpenHAB and Home Assistant integrations) – confirmed register addresses for voltages, currents, power, etc. (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community) (Reading data from Huawei inverter SUN 2000 (3KTL-10KTL) via modbus TCP and RTU - Tutorials & Examples - openHAB Community) (Integration Solar inverter huawei 2000L - #543 by ligeza - Feature Requests - Home Assistant Community).

- ThingsBoard Documentation – IoT Gateway configuration and Modbus connector usage.