RemoteGPIO Communication Module Integration Guide

This document describes the RGPIO driver architecture and explains how to develop an external communication module to integrate non-natively supported Input/Output (I/O) devices.

1. General Architecture

The system is designed in two distinct parts to ensure stability and modularity:

1. **The Core Engine (dbus-rgpio.py)**
    - This is the heart of the system. It runs continuously.
    - **Responsibilities:**
        - Manage the lifecycle of the kernel module (rgpio_module.ko) which creates the virtual GPIOs.
        - Manage the creation, update, and deletion of D-Bus services for relays (com.victronenergy.switch.XXX).
        - Read its configuration file (conf/config.ini) to know the **structure** of the devices (how many inputs, how many relays, what serial number).
        - Expose a simple internal API via the local MQTT broker to receive input states and send commands to relays.
    - **It knows nothing about specific communication protocols** (Dingtian MQTT, Modbus, etc.).
2. **Communication Modules** (e.g., modules/dingtian_mqtt_bridge.py)
    - These are independent scripts, each running in its own service.
    - **Responsibilities:**
        - Manage communication with a specific type of device, using its protocol (MQTT, Modbus TCP, HTTP, etc.).
        - Read its own configuration file (e.g., conf/dingtian_mqtt.ini) for connection details (IP address, MQTT topics, Modbus registers, etc.).
        - Act as a **translator**:
            - Convert messages from the device into messages understandable by the Core Engine's API.
            - Convert commands from the Core Engine's API into commands for the device.

2. The Internal API (via MQTT)

Communication between the Core Engine and the communication modules is done exclusively via the local MQTT broker, accessible at localhost:1883.

A. Digital Inputs Communication

This communication is one-way: from the communication module to the Core Engine.

- **Objective:** Inform the Engine that a physical input has changed state.
- **MQTT Topic:** rgpio/api/input/state/<serial>/<input_number>
- **Payload (Message):** 0 (for OFF) or 1 (for ON).

How it works:

1. Your communication module detects that an input on your hardware has changed state (e.g., input 3 of device RGPIO_001 turns ON).
2. It must then publish a message to the topic: rgpio/api/input/state/RGPIO_001/3
3. The payload of this message must be 1.
4. The Core Engine, which is subscribed to rgpio/api/input/state/#, will receive this message, find the virtual GPIO corresponding to RGPIO_001_input_3, update its value in sysfs, and trigger the interrupt to notify dbus-digitalinputs.

Important:

- The <serial> must exactly match the one defined in the Core Engine's conf/config.ini file.
- The <input_number> is 1-based (the first input is 1, the second is 2, etc.).

B. Relays (Switches) Communication

This communication is two-way.

i. Command from Engine to Module

- **Objective:** Order the communication module to change the state of a physical relay.
- **MQTT Topic:** rgpio/api/relay/set/<serial>/<relay_number>
- **Payload (Message):** ON or OFF.

How it works:

1. Your communication module must subscribe to the topic rgpio/api/relay/set/# to receive all relay commands.
2. When a user changes a relay's state in the Venus OS interface, the Core Engine publishes a command. For example, to turn off relay 2 of device RGPIO_001, it will publish to: rgpio/api/relay/set/RGPIO_001/2
3. The payload will be OFF.
4. Your module receives this message and must translate it into a specific command for your hardware (send a Modbus frame, publish to another MQTT topic, etc.).

ii. State Feedback from Module to Engine (Optional but recommended)

- **Objective:** Inform the Engine of the current state of a relay (useful on startup or if the state is changed locally on the device).
- **MQTT Topic:** The topic is defined by the device's protocol itself (e.g., dingtian/1/relay/1/state). The Core Engine subscribes to these external topics.
- **Payload (Message):** ON or OFF.

How it works:
The Core Engine reads the topic_base from config.ini and automatically subscribes to the relay state topics (<topic_base>/relay/+/state). Your module doesn't need to do anything special for this state feedback, other than ensuring the device publishes its state on these topics.

3. Configuring Your Module

Your communication module must have its own configuration file, stored in /data/RemoteGPIO/conf/.

- **Example (my_module_config.ini):**
  [modbus_settings]
  ip_address = 192.168.1.50
  port = 502

  [device_1]
  serial = RGPIO_003
  modbus_unit_id = 10

- **Link with the Engine:** The serial key is essential. It must match a serial defined in the Core Engine's config.ini file so that the engine knows how many inputs/relays it needs to create for this device.

4. Integration as a daemontools Service

To have your module launch automatically on startup, you must create a service, just as we did for dingtian_mqtt.

1. **Create the service directory:** mkdir -p /service/my-module
2. **Create the /service/my-module/run script** that waits for the Core Engine to be ready before launching your script:
   #!/bin/bash
   echo "Waiting for dbus-rgpio service to be available..."

   count=0

```
while [ ! -f /service/dbus-rgpio/supervise/ok ]; do
    sleep 1
    count=$((count+1))
    if [ $count -gt 60 ]; then
        echo "Timed out waiting for dbus-rgpio service. Exiting."
        exit 1
    fi
done

echo "dbus-rgpio service is up. Starting my bridge."
exec /usr/bin/python3 /data/RemoteGPIO/modules/my_module_bridge.py
```

3. **Make it executable:** chmod +x /service/my-module/run
4. **Create the log structure** just like for the other services.