

Integrations & Automation

Bridging Meshtastic with APRS, external services, and scripting automation via the Python library.

- [Meshtastic APRS Gateway](#)
- [Meshtastic Python Scripting](#)
- [Home Assistant Integration via MQTT](#)
- [Node-RED Flows for Mesh Automation](#)

Meshtastic APRS Gateway

What is APRS?

APRS - **Automatic Packet Reporting System** - is an amateur radio protocol designed for real-time tactical digital communications. Originally developed by Bob Bruninga (WB4APR), it carries position data, weather reports, text messages, and telemetry over RF. Reports are aggregated on publicly accessible maps such as [aprs.fi](https://www.aprs.fi), making it a cornerstone of ham radio situational awareness worldwide.

Why Integrate Meshtastic with APRS?

Meshtastic nodes equipped with GPS can be bridged into APRS-IS (the internet backbone of APRS), causing them to appear on [aprs.fi](https://www.aprs.fi) maps. This provides several advantages:

- Mesh position data becomes visible to the broader amateur radio community without requiring a dedicated APRS radio.
- Enables interoperability with Winlink gateway operators who already monitor APRS.
- During emergency activations, mesh nodes show up alongside APRS-equipped vehicles and portable stations on the same common operating picture.

License Requirement

Transmitting on APRS RF (144.390 MHz) legally requires an amateur radio license - **Technician class or higher** in the United States, under FCC Part 97. An internet-only APRS-IS feed (the software bridge described here) does not transmit on RF, so it is not license-gated by FCC law. However, APRS-IS network policy still requires you to connect with a valid amateur callsign and a callsign-derived passcode; the passcode system exists to keep APRS-IS restricted to amateur radio use. Receive-only IS access does not require a license, but to inject your nodes' positions you connect with your callsign. Check your national regulations.

aprstastic

The open-source **aprstastic** project (GitHub: `afourney/aprstastic`) is a bidirectional Meshtastic-to-APRS gateway that bridges Meshtastic output to APRS-IS. It runs on any Linux system with internet access - a Raspberry Pi is the most common deployment platform. Verify the current repository and its setup instructions before deploying.

Core requirements:

- Python 3.8+
- An MQTT broker already receiving packets from your Meshtastic node(s)
- An APRS-IS account (callsign + APRS-IS passcode - generated from your callsign)

Setup Overview

1. Install the gateway and its Python dependencies (follow the project's own README; for a manual stack: `pip install meshtastic paho-mqtt aprslib`).
2. Clone the gateway repository and copy the example config file.
3. Edit the config: enter your callsign, APRS-IS passcode, APRS-IS server (e.g. `rotate.aprs2.net:14580`), and MQTT broker address.
4. Test manually using the command documented by the gateway project.
5. Install as a systemd service for automatic start on boot:
`/etc/systemd/system/aprstastic.service`
Set `Restart=on-failure` and `RestartSec=10` to recover from transient network drops.

Packet Flow

```
Meshtastic node
↓ (LoRa RF)
Mesh network
↓ (LoRa RF)
MQTT gateway node (USB-connected Pi)
↓ (TCP, localhost)
Mosquitto MQTT broker
↓ (TCP, internet)
aprstastic gateway
↓ (TCP, APRS-IS protocol)
APRS-IS network
↓
aprs.fi (and other APRS clients)
```

Position Filtering and Consent

The gateway config allows you to whitelist specific node IDs or callsigns for forwarding. **Only forward nodes whose operators have consented.** Do not expose the positions of private nodes, shelters, or sensitive installations to the public APRS map without explicit permission from the node owner.

Best practice: maintain a config-controlled allowlist and review it whenever new nodes join your mesh. Note that allowlisting at the APRS gateway only limits APRS exposure - if the source mesh already uplinks to the public MQTT broker, those positions are already public regardless of the APRS allowlist.

Bidirectional Messaging

Some implementations support two-way message bridging - APRS messages addressed to your gateway callsign arrive as Meshtastic messages, and mesh messages can be relayed outbound as APRS messages. This is useful during ARES/RACES exercises where some operators have APRS radios and others have Meshtastic nodes.

Part 97 compliance warning: if you relay Meshtastic content *outbound onto APRS/amateur RF*, FCC Part 97 applies. Encrypted or meaning-obscured traffic is prohibited (§97.113(a)(4)) - so default-encrypted Meshtastic channels cannot lawfully be bridged onto amateur frequencies - and the station must transmit its callsign identification at least every 10 minutes (§97.119). An anonymous or encrypted mesh payload cannot lawfully be retransmitted as amateur traffic.

Implementation complexity is higher than position-only bridging; test thoroughly before deploying in an operational scenario.

Frequency Note

APRS in North America operates on **144.390 MHz** (a 2-meter FM frequency). This is entirely separate from LoRa mesh frequencies (915 MHz in North America). The APRS gateway described here is a *software bridge over IP* - it does not require your Pi to transmit on 144.390 MHz. The RF-to-internet connection is handled by existing APRS-IS infrastructure.

If you want true RF APRS transmission (e.g. for locations without internet), you need a licensed APRS radio and a TNC - that is a separate project beyond this software gateway.

Meshtastic Python Scripting

The meshtastic Python Library

The official **meshtastic** Python library provides programmatic access to any Meshtastic device. It exposes the full device API: sending and receiving messages, reading/writing configuration, querying node lists, requesting telemetry, and subscribing to real-time events.

Install:

```
pip install meshtastic
```

Official documentation: python.meshtastic.org

Source and examples: github.com/meshtastic/python

Connection Types

Interface class	Transport	Typical use
<code>meshtastic.SerialInterface</code>	USB serial (CDC ACM)	Direct connection, most reliable; default port auto-detected or specify <code>/dev/ttyUSB0</code>
<code>meshtastic.TCPInterface</code>	TCP/IP over WiFi	Wireless connection to nodes with WiFi enabled (ESP32 devices); specify host IP (default port 4403)
<code>meshtastic.BLEInterface</code>	Bluetooth Low Energy	Short-range wireless; requires BlueZ on Linux or appropriate BLE stack

Basic Usage Examples

The snippets below assume you have already created an `iface` object as shown in the first example. When copying a single snippet, include that connection setup line first.

Connect and print node info

```
import meshtastic
import meshtastic.serial_interface

iface = meshtastic.serial_interface.SerialInterface()
print(iface.nodes) # dict of all known nodes
iface.close()
```

Send a message to a channel

```
iface.sendText("Hello mesh", channelIndex=0)
# channelIndex=0 is the primary channel; 1 - 7 are secondary channels
```

Subscribe to received messages

```
import meshtastic
import meshtastic.serial_interface
from pubsub import pub

def on_receive(packet, interface):
    print(f"Received: {packet}")

iface = meshtastic.serial_interface.SerialInterface()
pub.subscribe(on_receive, "meshtastic.receive")

# Keep the script alive while the interface thread processes events
import time
try:
    while True:
        time.sleep(1)
except KeyboardInterrupt:
    iface.close()
```

Get node list with last-heard timestamps

```
for node_id, node in iface.nodes.items():
    last_heard = node.get("lastHeard", "unknown")
    name = node.get("user", {}).get("longName", node_id)
    print(f"{name}: last heard {last_heard}")
```

Read and write device config

```
local = iface.getNode('^local')
# Read
print(local.localConfig.lora.hop_limit)
```

```
# Write
local.localConfig.lora.hop_limit = 5
local.writeConfig("lora")
```

Note: `getNode('^local')` returns the local Node object. `localConfig`, `writeConfig()`, `writeChannel()`, `getNodeNum()` and `removeNode()` are methods/attributes of the **Node** object (reached via `getNode(...)`), not of the base interface.

Request position from local node

```
iface.getNode('^local').requestPosition()
```

To set a fixed position programmatically (including altitude), use

```
iface.localNode.setFixedPosition(lat, lon, alt).
```

Automation Use Cases

- **Automated mesh announcements** - scheduled weather, news, or system status broadcasts on a channel.
- **Message bridges** - relay messages between the mesh and Telegram, Discord, Matrix, or SMS (via Twilio). Bidirectional bridging is straightforward with the pub/sub event model.
- **Telemetry logging** - write incoming telemetry packets to CSV, SQLite, or InfluxDB for Grafana dashboards.
- **Position mapping** - forward GPS positions to a self-hosted map (e.g. Traccar, OwnTracks, or a custom Leaflet map).
- **Alert systems** - trigger SMS or email alerts when specific nodes go offline (last-heard threshold exceeded).
- **Config management** - script bulk configuration changes across many nodes connected via TCP.

Privacy note: the bridging and position-mapping patterns export mesh content and node positions to external services. Make sure that exposure is acceptable for your deployment before forwarding messages or locations off the RF mesh.

Common Patterns

```
# Connect (serial, auto-detect port)
iface = meshtastic.serial_interface.SerialInterface()

# Connect (TCP)
```

```
iface = meshtastic.tcp_interface.TCPInterface("192.168.1.50")

# Send text
iface.sendText("Hello mesh", channelIndex=0)

# Send to specific node (DM)
iface.sendText("Private message", destinationId="!a1b2c3d4", channelIndex=0)

# Request position
iface.getNode('^local').requestPosition()

# Write channel config
ch = iface.getNode('^local').channels[0]
ch.settings.name = "MyNet"
iface.getNode('^local').writeChannel(0)
```

Event Loop and Threading

The library spawns a background thread that reads from the serial/TCP connection and dispatches events via the **PyPubSub** pub/sub system. Your `on_receive` callback is called in that background thread - use thread-safe data structures (queues, locks) if you share state with your main thread.

Available topics:

- `meshtastic.receive` - any incoming packet
- `meshtastic.receive.text` - text messages only
- `meshtastic.receive.position` - position updates
- `meshtastic.receive.user` - node user/info packets
- `meshtastic.receive.data.portnum` - data packets by port number (telemetry arrives here under the `TELEMETRY_APP` portnum; there is no separate `meshtastic.receive.telemetry` topic)
- `meshtastic.connection.established` - fired after successful connect
- `meshtastic.connection.lost` - fired on disconnect

Error Handling

- **Serial port busy:** only one process can hold the serial port. Close the [Meshtastic app](#) or CLI before running your script. Use `try/finally: iface.close()` to release cleanly.
- **Reconnection on disconnect:** subscribe to `meshtastic.connection.lost` and re-instantiate the interface with exponential backoff.
- **Timeout handling:** `SerialInterface` has a `noProto` parameter - set it to `True` for raw serial access without waiting for a Meshtastic handshake, useful for debugging hardware.

- **Packet validation:** always guard against missing keys in the packet dict; not all fields are present in every packet type.

Home Assistant Integration via MQTT

Overview

Integrating Meshtastic into Home Assistant unlocks powerful home automation possibilities: track family members on a mesh map, get alerts when a node goes offline, and trigger smart-home actions based on mesh events. The integration uses MQTT as the transport layer, with Mosquitto as the local broker.

Step 1: Install and Configure Mosquitto

In Home Assistant, navigate to **Settings** → **Add-ons** → **Add-on Store** and install the *Mosquitto broker* add-on. After installation, open its configuration tab and add a user:

```
logins: - username: meshuser  
        password: yourpassword
```

Start the add-on and enable *Start on boot*. Note your Home Assistant host IP - your Meshtastic node will connect to this address.

Step 2: Configure Your Meshtastic Node for MQTT

Open the [Meshtastic app](#), go to **Radio Configuration** → **MQTT** and set:

- **MQTT Server Address:** your HA host IP (e.g. `192.168.1.100`)
- **Username / Password:** the credentials you set above

- **Root Topic:** `msh/US` (or your region)
- **JSON output:** enable *JSON enabled* so the node publishes the human-readable JSON used by the sensors below.
- **Uplink Enabled:** on. **Leave Downlink DISABLED** for Home Assistant monitoring - you only need uplink to read data. Downlink lets MQTT inject messages back into your RF mesh, which is an injection risk if the broker is ever reachable by anyone else; only enable it if you specifically intend to send messages from HA into the mesh.

Also ensure your node has WiFi configured under **Radio Configuration → Network**.

Step 3: MQTT Sensor YAML in Home Assistant

Add the following to your `configuration.yaml` (adjust the region, channel name, and node IDs as needed). The Meshtastic JSON topic format is `msh/REGION/2/json/CHANNELNAME/!gatewayNodeId` - the segment after `json` is the **channel name** (e.g. `LongFast`), not the packet type. The packet type (`telemetry`, `position`, etc.) is a field *inside* the JSON payload (`value_json.type`), so we subscribe with a `+` wildcard for the channel and filter on the type in each template:

```
mqtt:
  sensor:
    - name: "Node !a1b2c3d4 Battery"
      # Wildcard channel segment; replace !a1b2c3d4 with the publishing gateway node's id
      state_topic: "msh/US/2/json/+/!a1b2c3d4"
      value_template: >-
        {% if value_json.type == 'telemetry' %}
        {{ value_json.payload.battery_level }}
        {% else %}{{ states('sensor.node_a1b2c3d4_battery') }}{% endif %}
      unit_of_measurement: "%"
      device_class: battery
    - name: "Node !a1b2c3d4 Latitude"
      state_topic: "msh/US/2/json/+/!a1b2c3d4"
      value_template: >-
        {% if value_json.type == 'position' %}
        {{ value_json.payload.latitude_i | float / 1e7 }}
        {% else %}{{ states('sensor.node_a1b2c3d4_latitude') }}{% endif %}
      - name: "Node !a1b2c3d4 Longitude"
      state_topic: "msh/US/2/json/+/!a1b2c3d4"
      value_template: >-
        {% if value_json.type == 'position' %}
        {{ value_json.payload.longitude_i | float / 1e7 }}
```

```
{% else %}}{{ states('sensor.node_alb2c3d4_longitude') }}{% endif %}
```

Note: the last topic segment is the *gateway* node that published to MQTT, which is not necessarily the originating node. Verify the exact JSON payload field names (e.g. `battery_level`, `latitude_i`) against the JSON your firmware actually emits, and see this wiki's *MQTT Topic Structure and Packet Format* page for the canonical topic layout.

Reload your YAML configuration after saving.

Step 4: Automations

Alert when a node hasn't been heard in 30 minutes:

```
automation: - alias: "Mesh node offline alert"
  trigger:
    platform: state
    entity_id: sensor.node_alb2c3d4_battery
    to: unavailable
    for: "00:30:00"
  action:
    service: notify.mobile_app_your_phone
    data:
      message: "Node !alb2c3d4 has not reported in 30 minutes"
```

Notify on critical battery:

```
- alias: "Mesh node low battery"
  trigger:
    platform: numeric_state
    entity_id: sensor.node_alb2c3d4_battery
    below: 15
  action:
    service: notify.mobile_app_your_phone
    data:
      message: "Node !alb2c3d4 battery critical: {{ states('sensor.node_alb2c3d4_battery') }}"
```

Trigger lights when a family member arrives home via mesh position:

```
- alias: "Welcome home via mesh"
  trigger:
```

```
platform: zone
entity_id: device_tracker.mesh_family_member
zone: zone.home
event: enter
action:
service: light.turn_on
target:
entity_id: light.porch
```

To use zone-based presence, create a `device_tracker` that updates from the latitude/longitude sensors using a template or the MQTT device tracker integration.

Step 5: Lovelace Dashboard

For a mesh map view, install the **Map card** in Lovelace. Add a card of type `map` and reference your device tracker entities. You can also use the *auto-entities* HACS card to dynamically list all mesh node sensors. For richer visualization, some community members use Grafana with the InfluxDB integration to feed in MQTT telemetry.

Node-RED Flows for Mesh Automation

Overview

Node-RED is a visual flow-based programming tool that acts as powerful middleware between your Meshtastic MQTT feed and virtually any other service. It runs on Linux (including Raspberry Pi), inside Home Assistant, or on any Node.js-capable machine.

About the JSON MQTT topics used below: Meshtastic publishes decoded JSON packets to `msh/REGION/2/json/CHANNELNAME/USERID` - the segment after `json` is the **channel name**, not the packet type. There is no `.../json/text/` sub-topic. To catch all channels, subscribe to `msh/US/2/json/#` and filter on the packet type (a `type` field *inside* the JSON, e.g. `"text"`, `"position"`) in a switch/function node. Also note these JSON flows require the gateway to publish **decoded (plaintext)** packets to the broker - which bypasses channel encryption at the broker. If that broker is the public one, the decoded content (including any "emergency" channel) is public. Use a private, access-controlled broker for any sensitive channel.

Installing Node-RED

Standalone on Raspberry Pi / Linux:

```
bash <(curl -sL https://raw.githubusercontent.com/node-red/linux-installers/master/deb/update-nodejs-and-nodered)
sudo systemctl enable nodered
sudo systemctl start nodered
```

Access the editor at `http://your-pi-ip:1880`.

In Home Assistant: Install the *Node-RED* add-on from the add-on store. It integrates directly with your HA entities.

Core Flow Pattern 1: Message Logger

This flow captures all text messages from Meshtastic and writes them to a log file.

```
[MQTT In] → [JSON Parse] → [Switch: msg.payload.type == "text"] → [Function: format line] → [File Write]
```

MQTT In topic: msh/US/2/json/# (subscribe to all channels; filter type in the switch)

Function node:

```
msg.payload = new Date().toISOString() + " [" +  
msg.payload.from.toString(16).toUpperCase() + "]" +  
msg.payload.payload + "  
";  
return msg;  
File node: /home/pi/mesh_log.txt (append mode)
```

In the decoded text-message JSON, `from` is a top-level decimal node ID (prefix it with `!` when displaying the hex), and for a text packet the message string is at `payload.payload` (not `payload.payload.text`). Verify the exact field nesting against a live JSON sample, since it varies by packet type.

Core Flow Pattern 2: Position Tracker to Google Sheets

Filter position packets by a specific node ID and push lat/lon to a Google Sheet via the Sheets API node (`node-red-contrib-google-sheets`).

```
[MQTT In: msh/US/2/json/#] → [JSON Parse] → [Switch: msg.payload.type == "position" AND  
msg.payload.from == targetNodeId]  
→ [Function: build row] → [Google Sheets: append row]
```

Function node:

```
var pos = msg.payload.payload;
```

```
msg.payload = [  
  new Date().toISOString(),  
  pos.latitude_i / 1e7,  
  pos.longitude_i / 1e7,  
  pos.altitude  
];  
return msg;
```

Core Flow Pattern 3: Two-Way Bridge - Meshtastic ↔ Telegram

Install `node-red-contrib-telegrambot`. This flow forwards incoming mesh messages to a Telegram chat and relays Telegram replies back to the mesh channel.

Meshtastic → Telegram:

```
[MQTT In: msh/US/2/json/#] → [JSON] → [Switch: type == "text"] → [Function: build TG msg]  
→ [Telegram Sender]
```

Telegram → Meshtastic:

```
[Telegram Receiver] → [Function: build MQTT payload]  
→ [MQTT Out: msh/US/2/json/mqtt/] (downlink topic)
```

Downlink is more involved than it looks. Publishing arbitrary JSON to an output topic does *not* reliably inject a message into the mesh. For the firmware to transmit a downlinked message the gateway must have downlink enabled for that channel (`mqtt` module / channel `downlink_enabled`), and the payload must use the correct topic and envelope that the firmware accepts. If those conditions are not met, your Telegram replies will silently never reach the mesh. Confirm the current downlink topic/envelope against the Meshtastic MQTT integration docs before relying on this direction.

Emergency Channel SMS Forwarding via Twilio

Install `node-red-contrib-twilio`. Watch a dedicated emergency channel (channel index 1, for example) and forward any message to a phone number via SMS:

```
[MQTT In: msh/US/2/json/#] → [JSON] → [Switch: type == "text" AND channel == 1]
→ [Function: format SMS] → [Twilio: send SMS]
```

Function node:

```
var from = msg.payload.from.toString(16).toUpperCase();
msg.payload = "MESH EMERGENCY from !" + from + ": " +
msg.payload.payload;
return msg;
```

Set the Twilio node with your Account SID, Auth Token, and destination phone number.

Caution - this is a best-effort convenience path, not a primary emergency channel. This SMS-forwarding chain depends on the gateway hearing the packet over RF *and* on the gateway's internet/cellular connection, the MQTT broker, Node-RED, and Twilio all being up - any of which can fail silently during the exact grid-down / internet-outage scenario an emergency net plans for. Treat it as a convenience overlay on top of direct RF monitoring, never as your primary emergency communications method.

Exporting Flows

To share or back up a flow, select all nodes with **Ctrl+A**, then go to **Menu → Export → Clipboard**. The resulting JSON can be imported on any Node-RED instance. Store your flow exports in version control alongside your Meshtastic configuration.