Meshtastic Modules and Plugins

- Telemetry Module: Device, Environment, and Power
- Store and Forward Module
- Range Test Module
- External Notification and Canned Messages
- Serial, MQTT, and Ambient Light Modules

Telemetry Module: Device, Environment, and Power

The Telemetry module broadcasts sensor data from your node across the mesh - battery voltage, temperature, humidity, barometric pressure, and more. Other nodes receive this data and display it in the app, and it can be forwarded to external systems via MQTT. Note that telemetry (battery state, temperature, and occupancy-correlated patterns) is broadcast on the channel and, on any channel with an MQTT uplink, is republished to the broker externally. Disable telemetry types you do not need on sensitive or privacy-critical nodes.

Enabling Telemetry

In the Meshtastic app: **Radio Config → Modules → Telemetry**

Via CLI:

```
meshtastic --set telemetry.device_update_interval 1800
meshtastic --set telemetry.environment_update_interval 1800
meshtastic --set telemetry.power_update_interval 1800
```

Intervals are in seconds. The default device telemetry update interval is 1800 seconds (30 minutes), which is appropriate for most deployments. Setting an interval to 0 selects the default (1800 s) - it does *not* disable that telemetry type. To disable a telemetry type, set its `*_measurement_enabled` flag to false instead.

Device Telemetry

Device telemetry is always available - it reports internal state from the node itself, no external sensors required:

Field	Source	Notes

Battery level (%)	Hardware ADC	Accuracy varies by board; some boards always report 100% if battery not detected
Voltage (V)	Hardware ADC	Useful for deriving SoC for LiFePO4 packs
Channel utilization (%)	Radio stats	Percentage of airtime used; over 25% indicates congestion
Air utilization TX (%)	Radio stats	Percentage of time this node was transmitting
Uptime (seconds)	System clock	Resets on power cycle

Environment Telemetry

Requires an external I2C sensor. Supported sensors include (accuracy figures are from each sensor's manufacturer datasheet):

Sensor	Measures	I2C Address	Notes
BME280	Temp, Humidity, Pressure	0x76 or 0x77	Most popular; inexpensive; not suitable for air quality
BME680 / BME688 (BME68x)	Temp, Humidity, Pressure, gas resistance	0x76 or 0x77	Reports gas resistance (a VOC/air-quality proxy), not a fully-calibrated IAQ index, in Meshtastic; the gas sensor element needs a warm-up period before readings stabilize
SHT31	Temp, Humidity	0x44 or 0x45	High accuracy; ±0.3°C, ±2% RH (per Sensirion SHT3x datasheet)
MCP9808	Temperature only	0x18	±0.25°C accuracy (per Microchip MCP9808 datasheet, which is address-selectable across 0x18-0x1F); Meshtastic auto-detects it at 0x18
SHTC3	Temp, Humidity	0x70	Low power; used on some integrated boards

```
meshtastic --set telemetry.environment_measurement_enabled true
meshtastic --set telemetry.environment_screen_enabled true
```

Power Telemetry

Requires a supported I2C current/voltage sensor on the bus: INA219, INA226, INA260, or INA3221. Reports voltage, current draw, and calculated power consumption - useful for monitoring solar systems and diagnosing battery drain.

```
meshtastic --set telemetry.power_measurement_enabled true
```

Telemetry Intervals and Airtime Impact

Every telemetry broadcast consumes airtime. At Long Fast preset, a small telemetry packet takes roughly 0.3-0.5 seconds of airtime. At 30-minute intervals with one sensor type, this is negligible. But if you enable all three telemetry types at 5-minute intervals on a busy network, the combined airtime impact becomes meaningful. As a guideline:

- Fixed infrastructure nodes: 15-30 minute intervals are appropriate
- Portable/personal nodes: 30-60 minute intervals to reduce network load
- Emergency operations (where battery monitoring is critical): 5-minute intervals acceptable

Store and Forward Module

The Store and Forward module lets a designated node buffer messages for clients that were offline when a message was sent. When the offline client reconnects to the mesh, it can request the message history and receive messages it missed.

Important caveat: Store and Forward is **best-effort, not a delivery guarantee**. The server only buffers traffic it personally received over RF; clients only get the history they explicitly request while in range of the server; and the buffer is volatile (held in RAM and lost on reboot or power loss). Do not rely on it to guarantee delivery of critical messages.

How Store and Forward Works

1. A node configured as a Store and Forward **server** (typically a fixed repeater with reliable power) listens to all channel traffic and stores messages in the device's PSRAM (volatile RAM, not flash). The buffer is lost on reboot or power loss.
2. The server periodically sends a **heartbeat** advertising itself to the mesh.
3. A client that was offline requests history from the Store and Forward server (on Android, by sending the text command `SF` to the server; on the Apple/connected app this happens automatically).
4. The server replays stored messages to the requesting client.

Enabling on a Server Node (Repeater)

Configure the node that will store messages. Note that only PSRAM-equipped ESP32 boards (for example T-Beam v1.0+, T3S3) can act as a server — nRF52840 devices cannot:

```
meshtastic --set store_forward.enabled true
meshtastic --set store_forward.is_server true
meshtastic --set store_forward.records 0
meshtastic --set store_forward.history_return_max 25
```

`records` sets the maximum number of messages to store (circular buffer; oldest messages are overwritten). The default is **0**, which auto-allocates roughly two-thirds of the device's PSRAM (about 11,000 records); set a specific number to cap it. Capacity scales with the device's PSRAM, not its flash — only ESP32 boards with onboard PSRAM (e.g. T-Beam, T3S3) can run an S&F server, and nRF52840 boards cannot.

Enabling on a Client Node

A client generally does not need any special Store and Forward configuration beyond being on the same channel as the server. The `is_server` flag already defaults to false:

```
meshtastic --set store_forward.enabled true
meshtastic --set store_forward.is_server false
```

When an app connects to the server, history can be retrieved automatically. Over LoRa, an Android client requests history by sending the text command `SF` to the server node.

Limitations and Considerations

- **Only one Store and Forward server per channel is recommended.** Multiple servers on the same channel will each replay messages independently, causing duplicate deliveries.
- **Server node must be always-on.** A server that goes offline defeats the purpose. Use a mains or solar-powered repeater with reliable uptime.
- **Only text messages are stored and replayed.** Position reports and telemetry are not stored.
- **Buffer is volatile.** Messages are held in PSRAM, not flash, so the entire history is lost on reboot or power loss. There is no flash-wear concern because nothing is written to flash for this purpose.
- **Message delivery is best-effort.** If the client is out of radio range of the server, history request packets cannot reach the server. History retrieval over LoRa is also unavailable on the default public channel.

Best Deployment Practice

Deploy Store and Forward on your highest-uptime fixed node - typically the primary community repeater with mains or solar power, and one that has onboard PSRAM. This is the node most likely to have been online and collecting messages during the period a user was offline.

Range Test Module

The Range Test module automates signal strength measurement for deployment validation - letting you map exactly where in your coverage area packets arrive successfully, and at what SNR and RSSI values.

What the Range Test Module Does

Range Test operates as a sender/receiver pair:

- **Sender node** - Broadcasts a test packet at a configurable interval, incrementing a sequence number each time. The test packet carries the sender's position.
- **Receiver node** - Any node with the module enabled can receive and log each packet's SNR, RSSI, and sequence number. The canonical GPS-tagged `rangetest.csv` is written to the flash of an **ESP32-based receiver** (saving is only available on ESP32 boards).
- **Output** - A CSV file (`rangetest.csv`) saved to the ESP32 receiver's internal flash, containing all received packets with position data, signal quality, and sequence numbers. Missing sequence numbers identify packet loss. The Apple/Android apps also offer their own separate position-log exports.

Setting Up a Range Test

Configure the Sender Node

```
meshtastic --set range_test.enabled true
meshtastic --set range_test.sender 60
```

`sender` is the interval in seconds between test packets. 60 seconds works well for driving tests; 30 seconds for walking tests where you move slower.

Configure the Receiver

```
meshtastic --set range_test.enabled true
meshtastic --set range_test.save true
```

With `save true` on an ESP32-based device, received packets are logged to a file called `rangetest.csv` in the device's internal flash memory (no SD card or smartphone required). Retrieve it over WiFi by browsing to `meshtastic.local/rangetest.csv`, which downloads the file automatically. Saving is only available on ESP32 boards; nRF52 boards cannot write the CSV.

Disable Range Test after testing. Leaving `range_test.enabled` on keeps the node broadcasting extra position-bearing test packets - that adds needless airtime and location exposure. Set `range_test.enabled false` when the test is complete.

Conducting a Range Test Drive

1. Place the sender node at your repeater location or test deployment point. Ensure it has GPS lock and is transmitting.
2. Configure an ESP32-based portable node as the receiver (so it can save the CSV to flash).
3. Drive or walk through your intended coverage area.
4. After the test, retrieve the CSV file from the receiver at `meshtastic.local/rangetest.csv`. Each row contains: timestamp, GPS lat/lon, SNR, RSSI, sequence number.
5. Import the CSV into Google Maps (My Maps), QGIS, or any mapping tool to visualize coverage.

Interpreting Results

The bands below are rough rules of thumb, not a Meshtastic-published spec. The actual SNR floor for decoding shifts with the modem preset / spreading factor, so treat the RSSI and SNR columns as approximate and read them alongside the SF note that follows.

RSSI	SNR	Connection Quality
-80 to -100 dBm	>5 dB	Excellent - reliable delivery
-100 to -115 dBm	0 to 5 dB	Good - occasional packet loss
-115 to -125 dBm	-5 to 0 dB	Marginal (preset-dependent)
Below -125 dBm	near the SF floor*	Edge of range - unreliable

* Note: The "edge of range" SNR is **spreading-factor-dependent**, not a flat -10 dB. LoRa can decode packets at negative SNR values - roughly -7.5 dB at SF7 down to approximately -20 dB at SF12. On the default LongFast preset (SF11) the decode floor is about -17 dB, so a -10 dB SNR link on LongFast is still well within usable range, not the edge. This is one of LoRa's most remarkable properties. RSSI alone is not the full picture; low RSSI with high SNR can still be a reliable link. (For consistency, reconcile these thresholds with the reading-network-statistics and neighbor-info pages, which should use the same canonical bands.)

Using Range Test for Repeater Placement Decisions

Deploy a temporary repeater at a candidate site, run a range test drive across the intended coverage area, then compare the CSV output against a test from your next-best candidate site. This gives objective, data-driven evidence for repeater placement decisions rather than guessing based on map topology alone.

External Notification and Canned Messages

External Notification Module

The External Notification module triggers a visual or audio alert on the node hardware when a message is received - useful for heads-up awareness without constantly watching a screen.

Configuration

```
meshtastic --set external_notification.enabled true
meshtastic --set external_notification.output 25
meshtastic --set external_notification.output_ms 1000
meshtastic --set external_notification.active true
meshtastic --set external_notification.alert_message true
```

`output` is the GPIO pin number the buzzer or LED is wired to — it **must** be set to the pin your hardware is on, otherwise the module activates nothing (use a safe GPIO for your board; `25` is only an example). `output_ms` is how long to activate the output in milliseconds. `active` sets the logic level (true = active high output). `alert_message` triggers on any incoming text message; `alert_bell` triggers only on text messages that contain the ASCII bell character (0x07). The module monitors text messages only and never triggers on non-text packet types.

Hardware Outputs

The module drives a GPIO pin that can trigger:

- **Buzzer** - Active buzzer connected between the GPIO pin and GND (through a transistor for modules requiring more current)
- **LED** - External LED indicator, useful for daytime visibility or status display
- **Relay** - Via a relay module, can trigger any external device: alarm siren, strobe light, or notification device

Most T-Beam and Heltec boards have accessible GPIO pins; consult your board's pinout documentation for safe GPIO numbers (avoid pins used by LoRa, I2C, and UART).

Canned Messages Module

The Canned Messages module allows quick message sending from the device hardware without using the phone app - ideal for when you have a dedicated node with a small rotary encoder or button input.

Use Case

Deploy a node in a fixed location (workshop, vehicle, equipment room) with a rotary encoder. The user can navigate a pre-programmed list of messages and send them with a button press, without needing to open a phone app. Useful for sending routine status messages in situations where opening a phone is inconvenient.

Configuration

```
meshtastic --set canned_message.enabled true
meshtastic --set-canned-message "OK|On my way|Need help|ETA 10 min|Stand by"
```

Messages are separated by pipe characters and are set with the dedicated `--set-canned-message` flag (not `--set canned_message.messages`). The total message list is limited to 200 bytes, so keep individual messages short.

Input Hardware

Requires a rotary encoder connected to GPIO pins, or can use UP/DOWN/SELECT button inputs. Configure the GPIO pin assignments:

```
meshtastic --set canned_message.inputbroker_pin_a 39
meshtastic --set canned_message.inputbroker_pin_b 36
meshtastic --set canned_message.inputbroker_event_cw UP
meshtastic --set canned_message.inputbroker_event_ccw DOWN
```

Pin numbers are board-specific and must be a valid GPIO in the 1-39 range. The event values are `InputEventChar` names such as `UP`, `DOWN`, and `SELECT` (not internal firmware enum names). The RAK WisBlock Input module provides a pre-built button/encoder input board compatible with the WisBlock system.

Serial, MQTT, and Ambient Light Modules

Serial Module

The Serial module lets external hardware send and receive Meshtastic messages over a UART serial connection - enabling integration with microcontrollers, GPS units, custom sensors, and computer software.

Modes of Operation

Mode	Description	Use Case
DEFAULT	Same as SIMPLE - a raw (dumb) UART byte tunnel	Generic byte bridging
SIMPLE	Raw byte UART tunnel (no framing); requires a channel named `serial`	Bridging arbitrary bytes between two nodes
TEXTMSG	Sends/receives strings as text messages on the default text channel	Simple text message bridging
PROTO	Protobuf framing	Programmatic Arduino/ESP32 integration / full access
NMEA	Outputs NMEA sentences from node GPS	Feeding position to chartplotters
CALTOPO	CalTopo-compatible position format	SAR map integration

Configuration

```
meshtastic --set serial.enabled true
meshtastic --set serial.mode SIMPLE
meshtastic --set serial.rxd 16
meshtastic --set serial.txd 17
meshtastic --set serial.baud BAUD_9600
```

RXD/TXD pin numbers are board-specific, and `serial.baud` takes an enum value (e.g. `BAUD_9600`, `BAUD_115200`), not a raw integer. SIMPLE (and DEFAULT) is a raw UART byte tunnel that requires a channel named `serial` - it does not broadcast to the default text channel. If you want serial input broadcast as ordinary text messages on the default channel (and received text messages printed back out as `<Short Name>: <text>`), use the **TEXTMSG** mode instead.

MQTT Module

The MQTT module directly connects a Meshtastic node to an MQTT broker (internet required), enabling cloud integration without a separate gateway computer.

Warning: uplinking to the public broker `mqtt.meshtastic.org` (with the shared `meshdev` credentials) publishes your default-channel traffic and node positions to the public internet - readable by anyone and shown on third-party maps such as meshmap.net. Meshtastic also publishes to MQTT **unencrypted by default, even on a channel with a PSK**, unless you set `mqtt.encryption_enabled true`. If you do not intend that public exposure, use a private broker and enable encryption.

```
meshtastic --set mqtt.enabled true
meshtastic --set mqtt.address "mqtt.meshtastic.org"
meshtastic --set mqtt.username "meshdev"
meshtastic --set mqtt.password "large4cats"
meshtastic --set mqtt.root "msh"
meshtastic --set mqtt.uplink_enabled true
meshtastic --set mqtt.downlink_enabled false
```

`uplink_enabled` sends local mesh packets to the broker. `downlink_enabled` receives packets from the broker and re-broadcasts locally - useful for extending the mesh over the internet but can cause feedback loops if misconfigured. Start with downlink disabled until you understand the topology.

Topic Structure

Meshtastic publishes to: `msh/REGION/2/e/CHANNELNAME/USERID` for raw protobuf packets, or `msh/REGION/2/json/CHANNELNAME/USERID` when JSON output is enabled. The `2/e` (or `2/json`) segment is the protocol-version/encoding marker, and the final segment is the gateway node's USERID. There is no per-packet-type topic level - the packet type is a field inside the JSON payload, not a topic segment. (Firmware before 2.3.0 used `/c/` in place of `/e/`.)

Example: `msh/US/2/e/LongFast/!abcd1234` for protobuf packets on the LongFast channel in the US region (or `msh/US/2/json/LongFast/!abcd1234` when JSON is enabled). Subscribe with wildcards:

`msh/US/#` to receive all traffic from US nodes.

Ambient Lighting Module

The Ambient Lighting module controls an onboard I2C RGB LED controller (the NCP5623, as used on the RAK14001), letting you set the LED on/off state, its drive current, and the red/green/blue levels. It is **not** a light sensor and does not auto-adjust screen-backlight brightness. (Light-intensity sensors such as the BH1750, VEML7700, or TSL2591 are read by the Telemetry module, not by this module.)

```
meshtastic --set ambient_lighting.led_state 1
meshtastic --set ambient_lighting.current 10
```

Use `ambient_lighting.led_state` (1 to enable, 0 to disable), `ambient_lighting.current` for the LED output drive, and the `red`/`green`/`blue` fields for color. This module targets an I2C RGB LED driver (e.g. the NCP5623 on the RAK14001); it is not documented to drive addressable NeoPixel/WS2812 strips.