

MeshCore Python API

The MeshCore Python library (`meshcore_py`) provides an async interface for building applications and scripts that communicate with MeshCore companion radio nodes. It is one of several programmatic access methods, alongside `meshcore.js` (NodeJS/JavaScript) and the `meshcore-cli` command-line tool.

Canonical source: github.com/meshcore-dev/meshcore_py. The API moves quickly — always check the repository README for the authoritative, up-to-date method signatures before relying on the examples below.

Requirements

- A reasonably modern Python 3 (check the repository's `pyproject.toml` for the exact minimum version)
- A MeshCore node connected via USB serial, BLE, or TCP

Installation

```
pip install meshcore
```

Source: github.com/meshcore-dev/meshcore_py

Connecting to a node

The `MeshCore` class is created with async factory methods (`create_serial`, `create_tcp`, `create_ble`), not a constructor. Commands are issued through the `mc.commands` namespace and return an `Event` whose `.payload` is a dict.

```
import asyncio

from meshcore import MeshCore, EventType


async def main():
```

```

# Connect via USB serial (most common). Port and baud are positional.
mc = await MeshCore.create_serial("/dev/ttyUSB0", 115200)

# Or via TCP (for nodes with a WiFi/TCP bridge); port is positional.
# The MeshCore TCP default is port 5000.
# mc = await MeshCore.create_tcp("192.168.1.100", 5000)

# Or via BLE by address:
# mc = await MeshCore.create_ble("AA:BB:CC:DD:EE:FF")

# Fetch device self-info (returns an Event; data is in .payload)
result = await mc.commands.send_appstart()
print(f"Connected to: {result.payload}")

await mc.disconnect()

asyncio.run(main())

```

Listing nodes and contacts

The device's stored contact list (optionally filtered by last-modification time) is returned in the event payload as a dict keyed by public key. Each contact is itself a dict accessed by string keys (e.g. `contact['adv_name']`).

```

async def main():
    mc = await MeshCore.create_serial("/dev/ttyUSB0", 115200)

    # Get the device's stored contact list
    result = await mc.commands.get_contacts()
    contacts = result.payload # dict keyed by public key

    for contact_id, contact in contacts.items():
        print(f"{contact['adv_name']} ({contact_id})")

    await mc.disconnect()

```

Note: RSSI and SNR are not stored on the contact record — they arrive on incoming message / raw-data events, not in the contact database.

Sending a message

Channel (broadcast) messages use `send_chan_msg(channel_index, msg)`. Direct messages use `send_msg(dst, msg)`, where `dst` is a contact object, a hex public-key string, or bytes — contacts are addressed by their public key, not a "node ID".

```
async def main():
    mc = await MeshCore.create_serial("/dev/ttyUSB0", 115200)

    # Send to a channel (broadcast); channel index is positional
    await mc.commands.send_chan_msg(0, "Hello mesh!")

    # Send a direct message to a contact identified by its public key
    result = await mc.commands.get_contacts()
    contacts = result.payload
    target = next(c for c in contacts.values() if c['adv_name'] == "Base Station")
    await mc.commands.send_msg(target, "Hello from Python!")

    await mc.disconnect()
```

Monitoring incoming messages

Incoming messages are handled by subscribing to an `EventType` (e.g. `CONTACT_MSG_RECV` for direct messages, `CHANNEL_MSG_RECV` for channel messages). The handler receives an `Event` whose `.payload` is a dict.

```
import asyncio
from meshcore import MeshCore, EventType

async def main():
    mc = await MeshCore.create_serial("/dev/ttyUSB0", 115200)

    async def on_message(event):
        data = event.payload
        print(f"[{data['pubkey_prefix']}] {data['text']}")

    mc.subscribe(EventType.CONTACT_MSG_RECV, on_message)
```

```
# Keep running and receiving events
print("Monitoring... press Ctrl+C to stop")
try:
    await asyncio.sleep(float('inf'))
except KeyboardInterrupt:
    pass
finally:
    await mc.disconnect()

asyncio.run(main())
```

Getting node telemetry

Self-info comes from `send_appstart()` (SELF_INFO). Core statistics — battery voltage, uptime, error counts, queue length — come from `get_stats_core()`. All return an `Event` whose `.payload` is a dict. TX power is set with `set_tx_power(val)`; it is not exposed as a read-only attribute.

```
async def main():
    mc = await MeshCore.create_serial("/dev/ttyUSB0", 115200)

    # Device self-info (name, public key, coordinates, etc.)
    self_info = (await mc.commands.send_appstart()).payload
    print(f"Self info: {self_info}")

    # Core statistics
    stats = (await mc.commands.get_stats_core()).payload
    print(f"Battery: {stats['battery_mv']} mV")
    print(f"Uptime: {stats['uptime_secs']} seconds")

    await mc.disconnect()
```

Use cases

- Network monitoring dashboards - log all messages and node activity to a database
- Gateway integrations - bridge MeshCore messages to Discord, MQTT, or other platforms
- Automated alerts - notify via SMS or email when specific keywords are detected

- Repeater health monitoring - check uptime, battery level, and contact count on a schedule
- Coverage mapping - record signal reports from automated messages during a walking survey

Error handling notes

MeshCore over serial can occasionally miss bytes or timeout. The library supports opt-in automatic reconnect (pass `auto_reconnect=True`, e.g. with `max_reconnect_attempts`, to the `create_*` factory method) — it is not enabled by default. For long-running scripts, handle errors and disconnects by subscribing to `EventType.ERROR` and `EventType.DISCONNECTED` rather than relying on a specific exception class. Check the repository README for the current error-handling surface.

Revision #3

Created 2026-05-03 03:37:05 UTC by Mesh America Admin

Updated 2026-06-09 14:18:39 UTC by Mesh America Admin