BleLink – Kodegennemgang og Funktionsforklaring

Denne fil gennemgår den **generiske** BleLink-implementering på **ESP32 (C++/Arduino)** og **Python (bleak)** – funktion for funktion. Målet er, at du hurtigt kan forstå lifecyclen, hvad hver metode gør, og hvordan modulerne spiller sammen.

Indhold

Designprincipper

- **Generisk transport**: BLE-laget kender ikke til specifikke JSON-felter som `type` eller `command`. Det leverer bare *enten* parsed JSON *eller* rå tekstlinjer.
- **Linje-termineret framing**: Hver meddelelse afsluttes med `\n`. Dette gør det nemt at sende streams, uanset hvor BLE-chunks bryder data.
- **Symmetri**: Begge sider kan sende JSON eller rå tekst.
- **Robusthed**: ESP32 håndterer geninitialisering ved uventede disconnects; Python har retry i `connect()`.
- **Enkel API**: Få, tydelige metoder til `send`/`receive`, `connect`/`disconnect` og status.

Dataprotokol og framing

- **Framing**: 1 linje = 1 meddelelse. Linjen afsluttes med `\n` (newline).
- **Indhold**:
  - Hvis linjen er **gyldig JSON**, leveres et *parsed objekt* til JSON-callback'en.
  - Hvis linjen **ikke** er gyldig JSON, leveres **rå tekst** til raw-callback'en.
- **Retning**:
  - ESP32 → Python: `sendJson(doc)` / `sendRaw(text)`
  - Python → ESP32: `send_json(obj)` / `send_raw(text)`

ESP32 (C++/Arduino) – BleLink.h / BleLink.cpp

Globale felter og callbacks

- `g_server`, `g_tx`: NimBLE-server og TX-characteristic (notify).
- `g_connected`: Holder forbindelsesstatus.
- `g_needReinit`: Signalerer, at BLE-stakken skal geninitialiseres (gøres i `loop()`).
- `g_rxBuf`: Akkumulerer indgående bytes indtil vi møder `\n`.
- **Server callbacks**:
  - `onServerConnected(...)`: Debouncer, sætter `g_connected=true`, stopper advertising.
  - `onServerDisconnected()`: Debouncer, starter advertising, sætter `g_needReinit=true`.
- **Char callbacks**:
  - `handleWrite(...)`: Samler line-fragmenter, parser JSON, eller emitter rå linje.

Klasseoversigt

```
class BleLink {
public:
  using JsonCb = std::function<void(const JsonDocument& doc)>;
  using RawCb  = std::function<void(const String& line)>;

  explicit BleLink(const char* deviceName = "BleLink-Device");

  void setup();
  void loop();
  void disconnect();
  bool isConnected() const;

  void sendJson(const JsonDocument& doc);
  void sendRaw(const char* cstr);

  void onReceiveJson(JsonCb cb);
  void onReceiveRaw(RawCb cb);

private:
  void _initializeBLE();
  void _sendLine(const char* cstr);
  void _emitJson(const JsonDocument& doc);
  void _emitRaw(const String& line);

  char   _name[32] = {0};
  JsonCb _jsonCb   = nullptr;
  RawCb  _rawCb    = nullptr;
};
```

Konstruktør

```
explicit BleLink(const char* deviceName = "BleLink-Device");
```

- Kopierer enhedsnavnet ind i `_name`. Dette navn bruges som **advertising name**.

`setup()`

- Kald denne i `Arduino::setup()`.
- Kalder `_initializeBLE()`:
  - Init NimBLE
  - Opretter server, service, karakteristika (NUS)
  - Starter advertising

`loop()`

- Kald denne i `Arduino::loop()`.
- Håndterer edge-case, hvor link kan være dødt uden at NimBLE kalder disconnect-callback.
- Hvis `g_needReinit` er sat (fx efter disconnect), deinit'er og init'er den BLE-stakken igen.

`disconnect()`

- Valgfri. Transportlaget er designet til at overleve uden eksplicit disconnect, men metoden findes til fremtidige udvidelser / pæn nedlukning.

`isConnected()`

- Returnerer `g_connected`.

`sendJson(const JsonDocument&)`

- Serialiserer JSON til `String`.
- Sikrer at strengen slutter med `\n` (føjer til hvis nødvendigt).
- Kalder `_sendLine(...)` som chunker og `notify()`'er.

`sendRaw(const char*)`

- Bygger en `String` fra argumentet.
- Sikrer afsluttende `\n`.
- Kalder `_sendLine(...)`.

`onReceiveJson(JsonCb)`

- Registrerer callback, der kaldes med et **ArduinoJson::JsonDocument** for hver modtaget JSON-linje.

`onReceiveRaw(RawCb)`

- Registrerer callback, der kaldes med en **String** for hver modtaget *ikke-JSON* linje.

Interne metoder

`_initializeBLE()`

- Sætter strømstyrke og MTU (`setPower`, `setMTU`).
- Opretter server/service/karakteristika:
  - Service: `NUS_SERVICE_UUID`
  - TX: `NUS_CHAR_TX_UUID` med `NIMBLE_PROPERTY::NOTIFY`
  - RX: `NUS_CHAR_RX_UUID` med `NIMBLE_PROPERTY::WRITE | WRITE_NR`
- Starter advertising og skriver log-linje.

`_sendLine(const char*)`

- Checker `g_connected` og `g_tx`.
- Sender i **chunks** á 20 bytes (sikkert for MTU=23: 3 bytes ATT header + 20 payload).
- Kald til `notify()` pr. chunk, med et lille `delay(2)` for at give BLE-stakken tid.

`_emitJson(const JsonDocument&)`

- Kalder brugers registrerede JSON-callback, hvis sat.

`_emitRaw(const String&)`

- Kalder brugers registrerede RAW-callback, hvis sat.

Server- og karakteristika-callbacks

- **ServerCallbacks**:
  - `onConnect(...)` (to varianter for kompta): kalder `onServerConnected(...)`.
  - `onDisconnect(...)` (to varianter): kalder `onServerDisconnected()`.
- **CharCallbacks**:
  - `onWrite(...)` (to varianter): kalder `handleWrite(...)`.

`handleWrite(...)`: - Læser `getValue()` (kan være fragment). - Appender til `g_rxBuf`, leder efter `\n`. - For hver komplet linje: - Forsøger `deserializeJson(doc, line)`: - **Succes**: `_emitJson(doc)` - **Fejl**: `_emitRaw(line)`

Python – ble_link.py

Felter og callbacks

- `self._client`: `BleakClient` eller `None`.
- `self._tx_char`, `self._rx_char`: referencer til karakteristika i NUS-servicen.
- `self._rxbuf`: `bytearray` hvor notifikationsdata akkumuleres indtil `\n`.
- Callbacks:
  - `self._cb_json(obj: dict)` – hvis sat, kaldes for gyldig JSON.
  - `self._cb_raw(s: str)` – hvis sat, kaldes for rå (ikke-JSON) linjer.
  - `self._cb_pair(type, payload)` – kompatibel helper: hvis JSON har `"type"`, leveres `(type, payload)`, ellers `(None, obj)`.

Konstruktør

```
def __init__(self, device_name: str):
    self.device_name = device_name
    ...
```

- Gemmer enhedsnavn (advertising name fra ESP32).

`connect(...)` (robust)

```
async def connect(self, attempts=3, delay=1.5, timeout=20.0, scan_timeout=12.0):
```

- Forsøger at forbinde op til `attempts` gange.
- Hver iteration kalder `_connect_once(...)`:
  - Scanner efter `device_name`.
  - Opretter BLE-forbindelse.
  - Finder **NUS-servicen** og **karakteristika** *inden for denne service* (undgår dublet-problemer).
  - Starter notify på TX-characteristic.
- Ved fejl: venter `delay` sekunder og prøver igen.
- Kaster `RuntimeError`, hvis alle forsøg fejler.

`disconnect()`

- Stopper notify, lukker forbindelsen pænt, rydder interne referencer og buffer.

`is_connected()`

- Returnerer sand hvis klienten findes og er forbundet.

`send_json(obj, response=True)`

- Serialiserer `obj` kompakt (uden unødige mellemrum) og tilføjer `\n`.
- Skriver til RX-characteristic med `write_gatt_char(...)`.
- `response=True` → brug *write-with-response* hvor muligt (bedre fejldiagnose).

`send_raw(text, response=True)`

- Sikrer afsluttende `\n`, skriver som ovenfor.

`send(command, payload=None, response=True)`

- **Convenience-wrapper** for at sende `{ "command": ..., "payload": {...} }`.
- Brugbar hvis du ønsker *kommando-stil* i protokollen.

`on_receive_json(cb)`

- Registrerer callback som kaldes for hver **gyldig JSON** fra ESP32.

`on_receive_raw(cb)`

- Registrerer callback som kaldes for hver **ikke-JSON** linje fra ESP32.

`on_receive(cb)`

- Kompatibilitets-callback: hvis JSON har feltet `"type"`, kaldes `cb(type, payload)`; ellers `cb(None, obj)`.

Notify-handler (intern): `_on_notify(...)`

```
def _on_notify(self, _handle: int, data: bytearray) -> None:
    self._rxbuf.extend(data)
    while True:
        try:
            idx = self._rxbuf.index(0x0A)  # '
'
        except ValueError:
            break
        line = self._rxbuf[:idx]
        del self._rxbuf[:idx+1]
        txt = line.decode("utf-8", errors="ignore").strip()
        if not txt:
            continue

        delivered = False
        try:
            obj = json.loads(txt)
            if self._cb_json:
                self._cb_json(obj); delivered = True
            if self._cb_pair:
                t = obj.get("type")
                payload = obj.get("payload", obj if t is None else {})
                self._cb_pair(t, payload); delivered = True
        except Exception:
            pass

        if not delivered and self._cb_raw:
            self._cb_raw(txt)
```

- Akkumulerer data i `self._rxbuf` indtil newline (`0x0A`) findes.
- For hver linje: prøver JSON-parse; leverer til relevante callbacks.
- Fald tilbage til rå callback, hvis ikke leveret via JSON/pair.

Livscyklus og samspil ESP32 ↔ Python

1. **ESP32**: `setup()` → `_initializeBLE()` → advertising starter.
2. **Python**: `connect()` scanner, forbinder, finder NUS, starter notify.
3. **Dataflow**:
   - Python → ESP32: `send_json()` / `send_raw()` → ESP32 `handleWrite()` → `onReceiveJson/Raw`-callbacks.
   - ESP32 → Python: `sendJson()` / `sendRaw()` → notify → `_on_notify()` → `on_receive_*` callbacks.
4. **Disconnect**:
   - Hvis linket ryger, ESP32 starter advertising igen og markerer reinit.
   - Python kan kalde `connect()` igen (har retry indbygget).

Fejlhåndtering og robusthed

- **ESP32**:
  - Debounce i connect/disconnect-callbacks for at undgå dobbelte logs/flow.
  - Reinit af BLE-stakken i `loop()` efter disconnect for "ren" state.
- **Python**:
  - `connect()` forsøger flere gange; typiske BlueZ-races dækkes ind.
  - Detekterer dublet-UUIDs ved kun at vælge karakteristika fra **samme service**.

Typiske fejl: - *failed to discover services* → Prøv igen (indbygget). Undgå aggressiv deinit på ESP32 i connect-vinduet. - *Multiple Characteristics with this UUID* → Løst ved at vælge karakteristika *via service* i stedet for global søgning.

Ydelse, MTU og chunking

- **ESP32 TX → host**: Chunk-størrelse = 20 bytes (sikkert for ATT MTU=23). Hvis du sætter højere MTU på både central og peripheral, kan du øge chunk-størrelsen – men 20 er portabelt.
- **Python TX → ESP32**: Bleak håndterer segmentering; vi sender altid en hel linje (host-side buffer).
- **Frekvens**: Undgå massive bursts af `notify()` uden pauser. Den indlagte `delay(2)` pr. chunk giver stakken luft.

Udvidelser og tips

- **Applikationsprotokol**: Læg dine felter (fx `op`, `type`, `id`, `payload`) i JSON på applikationsniveau – transportlaget er agnostisk.
- **Timeouts**: Byg evt. et request/response-lag ovenpå med `id` + timeouts på Python-siden.
- **Sikkerhed**: NUS er ukrypteret i sig selv; brug BLE-pairing/bonding hvis nødvendigt.
- **Logging**: Hold ESP32-loggen minimal i realtidssystemer. Br u g f.eks. ring-buffers og print sjældnere.
- **Test**: Start altid med kun BLE-modulet, og tilføj derefter motorstyring/IMU mv., så du let kan isolere evt. problemer.

Spørg endelig, hvis du vil have konkrete "request/response"-eksempler, versionsfelter i JSON, eller en simpel "RPC"-wrapper ovenpå dette transportlag.