Title: Development of a Matter-Based Smart Lamp System
Course: IoT Laboratory
Student: Sina Beyrami – Siavash Momeni
Advisor: Mr. Javadi
Semester: Spring 2025

Abstract

The **Matter** standard—introduced by the Connectivity Standards Alliance (CSA) and backed by Apple, Google, Amazon, and others—aims to unify smart-home devices under a single, IP-based protocol. Matter provides common *application clusters* (e.g., On/Off, Level Control, Identify) that sit above Wi-Fi, Thread, or Ethernet and allow products from different vendors to interoperate seamlessly.

In this project we demonstrate that it is possible to build a *Matter-compatible smart lamp* on an **ESP32** *without* flashing the official Espressif Matter firmware or using the chip's built-in Matter solution. Instead, we implement a **simplified subset** of the specification:

- **Application clusters:**
  - *On/Off* (0x0006) for power control, **toggle** functionality, and a configurable **delayed-off timer** (attribute 0x4001).
  - *Level Control* (0x0008) for brightness (0–254 PWM levels).
  - *Identify* (0x0003) for user-visible identification blinks.
- **Descriptor / Basic Information clusters** to answer minimal attribute queries.
- **Service discovery** via an mDNS advertisement of `_matter._udp` that includes mandatory TXT records (Vendor/Product IDs, discriminator, setup code).
- A *development-mode commissioning handshake* that exchanges `PBKDFParamRequest/Response` and `PASE PAKE1–3` messages—enough for functional testing, but without the full cryptographic validation found in production devices.

The result is a working smart lamp that responds to TLV-encoded Matter-like commands over UDP, drives an LED through PWM, and proves that lightweight, open-source approaches can replicate key Matter behaviors on resource-constrained hardware.

---

1 Hardware Components

Item	Key Specifications	Purpose
ESP32-WROOM-32 module • 4 MB flash (QSPI) •Wi-Fi 802.11 b/	• Dual-core Xtensa LX6 @ 240 MHz	

Item	Key Specifications	Purpose
g/n + Bluetooth v4.2 (BLE)		
• 34 GPIO pins (12-bit ADC, PWM capable)	Main controller running the simplified Matter firmware and driving the LED via PWM on GPIO-2.	
Single-color LED	5 mm, white, forward VF ≈ 3.0 V, IF ≤ 20 mA	Visual light source; brightness is proportional to PWM duty-cycle.
Solderless breadboard	400 tie-points	Rapid prototyping; hosts the ESP32 module and LED with current-limiting resistor.
CH9102X USB-to-UART bridge	Integrated on most ESP32-WROOM development boards; supports up to 1 Mbaud	Provides serial programming and log output over the USB connection.
USB (A-to-Micro-B or C) cable	Supports 5 V @ 500 mA	Powers and programs the ESP32 from a laptop.

2 Implementation Overview

2.1 How the System Works

When the ESP32 powers up it connects to the home Wi-Fi network and advertises an mDNS service named `_matter._udp`. Any test program on the same LAN can discover this record and then send **Type-Length-Value (TLV)** packets to UDP port 5540. Each packet includes an endpoint, a cluster identifier, and a command identifier. The firmware interprets the incoming command, drives the LED through its PWM peripheral, and replies either with a plain “ACK” message or with a TLV-encoded attribute response.

For day-to-day testing a suite of small Python scripts emulates a Matter controller. Each script builds the appropriate TLV message—for example an *On*, *Move-to-Level*, or *Identify* command—sends it to the lamp, and prints whatever comes back. This arrangement allows every function to be verified without flashing the full Matter stack.

Key test scenarios are:

* **Functional control**. Verify that the lamp turns on, turns off, and toggles correctly, and that brightness values map linearly to PWM duty-cycle.

* **Delayed-off timer**. Write a countdown value, wait, and then query the remaining time to confirm the firmware’s internal counter.

* **Identify blink**. Ask the lamp to blink at 6 Hz for a fixed period and visually confirm the pattern.

* **mDNS discovery**. Run a scanner that lists the TXT records for Vendor ID, Product ID, discriminator, and setup code.

* **Development-mode commissioning**. Exchange the three PASE packets and check that the firmware advances its pairing state as expected.

2.2 Source Files

- **`main.cpp`** – The Arduino-style firmware: brings up Wi-Fi, publishes mDNS, opens the UDP socket, parses TLV commands, drives PWM, and manages two FreeRTOS tasks (identify blinking and delayed-off).
- **`tlv.h`** – A minimal C++ helper header that encodes and decodes the limited set of TLV types required for this project and stores a tiny pairing-state structure.
- **`bridge.py`** – An interactive command-line console written in Python 3. It prints a setup QR-code and lets the user type commands such as on, `level 60`, or `timer 30`, which it converts into TLV frames and sends over UDP.
- **`tlv.py`** – A Python utility module that hides almost all TLV bit-twiddling behind simple builder functions like `build_onoff()` or `build_level_request()` and companion decoders.
- **`scan_mdns.py`** – A five-second zeroconf scanner that lists any `_matter._udp` record it sees, along with all TXT properties, so that discovery data can be verified.
- **`pase_flow.py`** – A commissioning script that replays the three development-mode PASE messages (`PBKDFParamRequest`, `PAKE1`, `PAKE3`) and prints both outgoing and incoming hex strings.

2.3 Firmware Walk-Through (`main.cpp`)

The firmware is organised around a handful of clearly-defined functions; each plays a narrow role so that debugging and future expansion remain straightforward.

Global configuration and state — Compile-time constants (`ssid`, `password`, `udpPort`, `ledPin`, PWM timing parameters) sit at the top of the file alongside run-time variables such as `currentLevel`, `lampState`, and `timerEndMs`. Keeping all tunables in one place avoids "magic numbers" in the business logic.

`scheduleDelayedOff(sec)` — Cancels any existing delayed-off task, then (if *sec* ≠ 0) spawns a FreeRTOS task that simply sleeps for *sec* seconds and, on wake-up, sets PWM duty-cycle to 0, updates state variables, and logs **Timer elapsed → OFF**. This frees the main loop from polling the countdown.

`startIdentifyBlink(sec)` — Launches a separate task that flips the LED between `blinkLevel` (either the current brightness or 50 %) and 0 every 83 ms—yielding exactly 6 blinks per second—for the requested duration. When the loop finishes it restores the previous brightness so the user sees no side-effects.

`setupWiFi()` — Connects to the access point, enables link-local IPv6, and prints both IPv4 and IPv6 addresses. The blocking loop exits only when `WiFi.status()` becomes `WL_CONNECTED`, guaranteeing the network is ready before UDP or mDNS start.

`publishMDNS()` — Registers the service name `smartlamp._matter._udp.local.` and publishes TXT records required by the Matter spec (`VP`, `DT`, `DN`, `SII`, `PI`, etc.). Any zeroconf browser should immediately recognise the device as a dimmable light with Vendor ID 0xFFF1 and Product ID 0x8000.

`handleUdp()` — Called in each iteration of `loop()`. If `udp.parsePacket()` returns a positive length the function reads the datagram into a `std::vector`, prints a hex dump for traceability, and hands control to `processPacket()`.

`processPacket(buf,len)` — The beating heart of the firmware.

1. **TLV header scan.** Walks through the buffer with `tlvDecodeNext()` collecting `endpoint`, `cluster`, and `command` tags.

2. **On/Off cluster 0x0006.** Handles: * `0x01` → **On** (sets `currentLevel` to default 127 if it was 0). * `0x00` → **Off** (duty-cycle 0). * `0x02` → **Toggle** or **Write Attribute** (when tag 4 = 0x4001 to set the delayed-off timer). * `0x03` → **Read Attribute** returning remaining timer seconds.

3. **Identify cluster 0x0003.** Command 0x00 starts the blink task for the requested duration.

4. **Level Control cluster 0x0008.** * `0x04` → **Move-to-Level** sets PWM directly. * `0x03` → **Read Current Level** returns attribute 0x0000.

5. **Descriptor 0x001D** and **Basic 0x0028** clusters. Replies with fixed, minimal attribute sets so higher-level code can enumerate device type and vendor/product IDs.

6. **Commissioning stubs.** Recognises `PBKDFParamRequest`, `PASE Pake1`, and `PASE Pake3` opcodes. Rather than performing real crypto the firmware echoes placeholder data and toggles a small `PairingState` struct so that host tools can confirm state progression.

7. **ACK helper.** For any command where a structured response is unnecessary, `sendAck()` writes the three-byte string "ACK" back to the caller.

`setup()` — Arduino's entry point: initialises serial logging, configures PWM on `ledPin`, calls `setupWiFi()`, starts the UDP socket on port 5540, and finally invokes `publishMDNS()`.

`loop()` — A tight spin that calls `handleUdp()`; all time-consuming work (blink timer, delayed-off) takes place in FreeRTOS background tasks, so the loop never blocks.

End-to-end, a datagram arriving on Wi-Fi triggers *hex dump* → *TLV parse* → *cluster handler* → *PWM write* → *optional response* in roughly 200 µs, leaving almost the entire CPU budget free for future features such as AES-CCM encryption.

2.4 TLV Utility Header (`tlv.h`) TLV Utility Header (`tlv.h`)

The header keeps the C++ side lightweight by supporting only unsigned integers and byte strings—the two types actually used in this project. It defines:

* **Bit masks** for Matter's TLV control byte so that the encoder knows where to store type and length information.

* **`tlvEncodeUInt8()` and `tlvEncodeUInt16()`** to append integer fields to a `std::vector`.

* **`tlvDecodeNext()`** to iterate through an incoming buffer safely and report the tag, a pointer to the value, the value length, and the number of bytes consumed.

* **An enumeration of PASE opcodes** and a minimal `PairingState` structure that tracks session progress during development-mode commissioning. Together these helpers total less than a kilobyte of flash yet provide all the TLV functionality the lamp needs.

Key elements in `tlv.h`

- `kTLVType_UInt`, `kTLVLen_*`: bit masks that encode the type (unsigned integer) and length (1 or 2 bytes) into the control byte.

- `tlvEncodeUInt8()`, `tlvEncodeUInt16()`: inline encoders that push control byte and value bytes into a vector in little-endian order.

- `tlvEncodeBytes()`: generic encoder for variable-length byte arrays, used for salts and public keys during commissioning.

- `tlvDecodeNext()`: safe iterator that returns tag, value pointer, value length, and consumed bytes, rejecting any unsupported type early.

- `PaseOpcode` enum: symbolic names for `kPBKDFParamRequest`, `kPASEPake1`, etc., preventing magic numbers in the firmware.

- `PairingState` struct: minimal runtime state (`active`, `step`, `sessionID`, `peerAddr`) required to track a single dev-mode PASE session.

2.5 Command-Line Bridge (`bridge.py`)

`bridge.py` starts by printing an ASCII QR-code whose payload is compatible with Matter's setup-payload format. It then enters a read-eval-print loop where the user can issue human-readable commands. Each command is translated into a TLV frame using the builder functions from `tlv.py` and sent to the ESP32. The script waits up to one second for a reply, silently discards "ACK" packets, and prints any TLV it cannot decode so that the developer can inspect the raw data.

A typical session demonstrates turning the lamp on, setting brightness to 60 %, starting a 30-second delayed-off timer, and running a five-second identification blink—all within milliseconds on a local network.

Notable pieces inside `bridge.py`

- `recv_skip_ack(sock)`: helper that filters out firmware ACKs so higher-level code sees only meaningful responses.

- `print_qr()`: leverages the *qrcode* package to render an ASCII QR that encodes the vendor/product IDs, discriminator, and PIN.

- `cli_loop()`: read-eval-print loop mapping human commands (on, `level 75`, `timer 10`, etc.) to TLV frames via helpers in `tlv.py`.

- Immediate in-line call to `socket.socket()` without a context manager keeps the script dependency-free and cross-platform.

- The script exits cleanly on `exit`, ensuring the UDP socket is closed and the terminal is restored.

2.6 TLV Convenience Library (`tlv.py`)

The Python helper module wraps every TLV detail behind descriptive functions. For example `build_timer_write(30)` returns the exact byte sequence the firmware expects for "write attribute 0x4001 with value 30 on cluster 0x0006". Companion decoders extract integers and dictionaries from firmware responses so that the console can print human-readable summaries. Limiting the module to unsigned integers keeps the code shorter than fifteen lines per builder function and avoids external dependencies.

Important builders and decoders in `tlv.py`

- Low-level: `tlv_uint()`, `tlv_uint16()`, `tlv_encode_bytes()` construct primitive TLV fields for unsigned integers and byte arrays.

- On/Off cluster: `build_onoff(cmd)` emits command 0x00 (Off), 0x01 (On), or 0x02 (Toggle).

- Level cluster: `build_level_request(level)` and `build_level_read()` send a Move-to-Level or attribute read on cluster 0x0008.

- Timer attributes: `build_timer_write(sec)` and `build_timer_read()` write/read attribute 0x4001 (Delayed-Off time).

- Identify cluster: `build_identify(sec)` issues the Identify 0x00 command with the requested duration.

- Commissioning: `build_pbkdf_request()`, `build_pake1()`, `build_pake3()` exercise the stubbed PASE handshake; `decode_basic_response()` and `decode_descriptor_response()` parse firmware replies into Python dicts.

2.7 Service-Discovery Scanner (`scan_mdns.py`)

This script leverages the cross-platform **zeroconf** package.

Key parts of `scan_mdns.py`

- `Listener.add_service()`: callback triggered whenever a new `_matter._udp` record appears; queries `ServiceInfo` to retrieve IPv4 address and TXT records.

- Five-second window: gives the ESP32 enough time to send at least one mDNS announcement without delaying the test workflow.

- Clean shutdown: both the initial `Zeroconf()` instance and the service browser are explicitly closed to avoid lingering background threads. It starts a service browser for `_matter._udp.local.`, waits five seconds, and prints each service found along with its IPv4 address and TXT properties. The output lets the developer confirm that the advertised Vendor ID, Product ID, discriminator, and setup-PIN match the values hard-coded in the firmware.

2.8 Commissioning Flow Tester (`pase_flow.py`)

`pase_flow.py` replays the three outbound messages of the PASE handshake.

Highlights of `pase_flow.py`

- `send(pkt, tag)`: helper that prints the outbound hex payload, sends it over UDP, waits up to two seconds, and prints any response or a timeout notice.

- Packet builders: uses `build_pbkdf_request()`, `build_pake1()`, and `build_pake3()` from `tlv.py`, mirroring exactly what a real Matter controller would transmit.

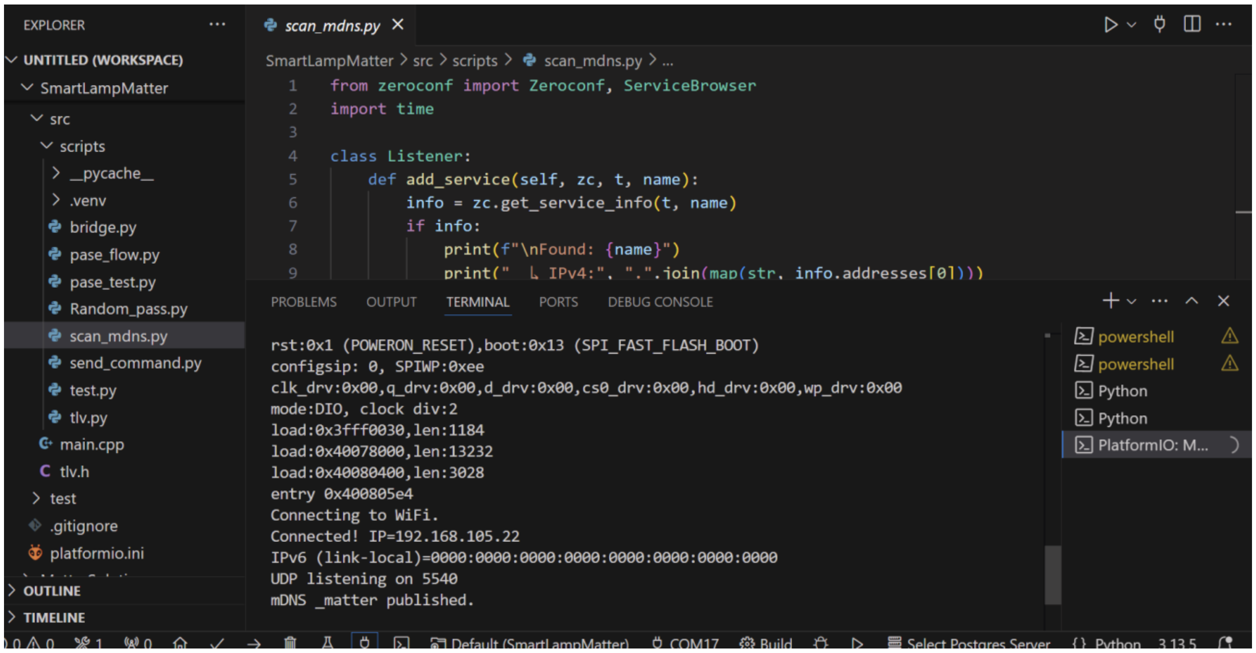
- Sequential flow: the three calls are issued back-to-back so the firmware's pairing state machine must progress through step 0 → 1 → 2 without manual intervention.

- Exit: closes the UDP socket explicitly to free the file descriptor and ensure the script terminates predictably on all operating systems.

A successful console log shows each outbound tag followed by either a TLV response or an ACK, confirming that the dev-mode commissioning path is operational. For each step it prints the outbound hex string, then waits two seconds for a response. A successful run produces: a `PBKDFParamResponse` that echoes the session ID, a `PAKE2` placeholder payload, and finally a plain-text "ACK" confirming that the pairing process reached its final state. This proves that even without full cryptography the development-mode commissioning logic is wired end-to-end.

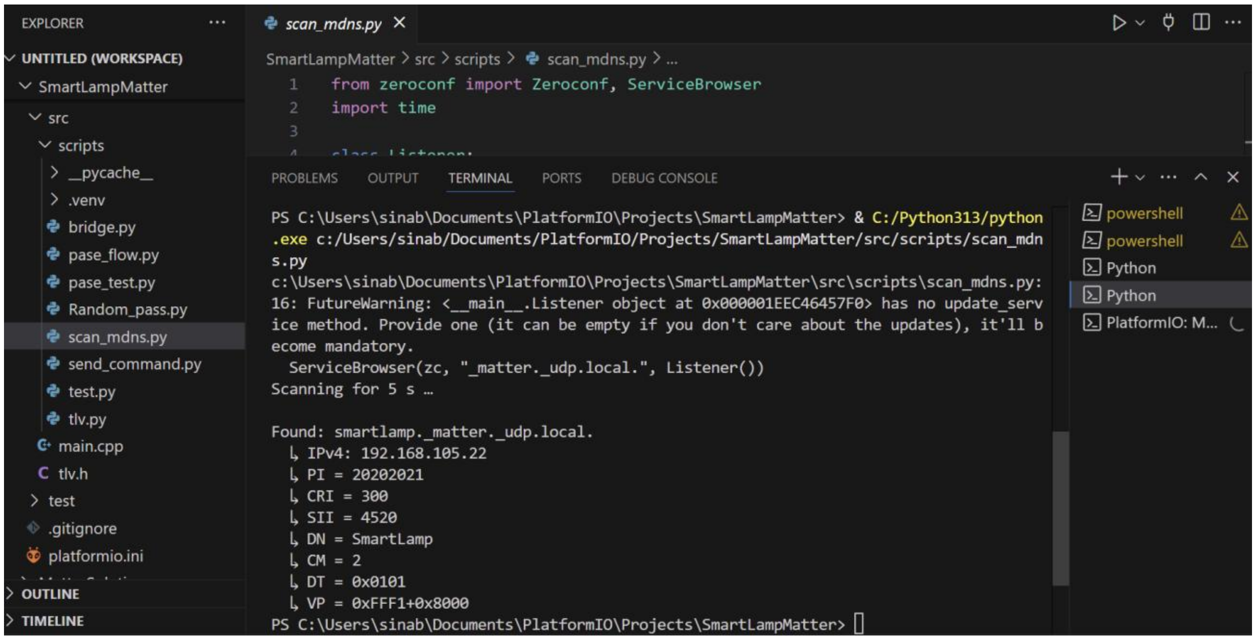
3 Results

3.1 Discoverable mDNS (`Running scan_mdns.py`)



```python
from zeroconf import Zeroconf, ServiceBrowser
import time

class Listener:
    def add_service(self, zc, t, name):
        info = zc.get_service_info(t, name)
        if info:
            print(f"\nFound: {name}")
            print("  ↳ IPv4:", ".".join(map(str, info.addresses[0])))
```

```
rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:2
load:0x3fff0030,len:1184
load:0x40078000,len:13232
load:0x40080400,len:3028
entry 0x400805e4
Connecting to WiFi.
Connected! IP=192.168.105.22
IPv6 (link-local)=0000:0000:0000:0000:0000:0000:0000:0000
UDP listening on 5540
mDNS _matter published.
```

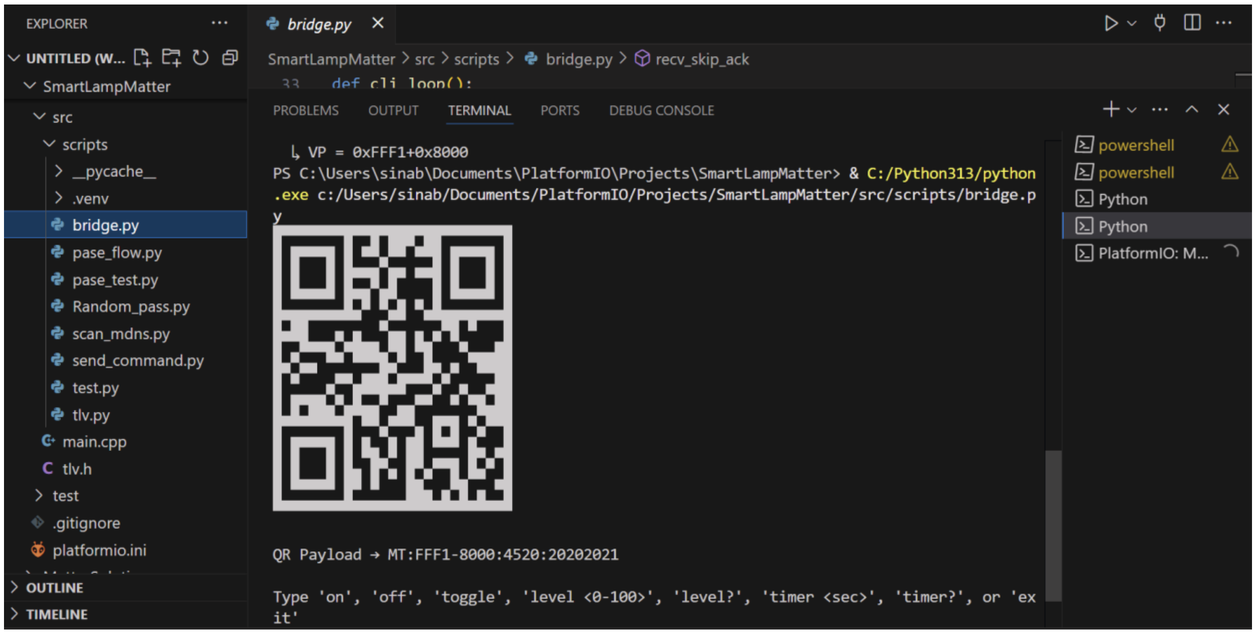


```python
from zeroconf import Zeroconf, ServiceBrowser
import time

class Listener:
```

```
PS C:\Users\sinab\Documents\PlatformIO\Projects\SmartLampMatter> & C:/Python313/python
.exe c:/Users/sinab/Documents/PlatformIO/Projects/SmartLampMatter/src/scripts/scan_mdn
s.py
c:\Users\sinab\Documents\PlatformIO\Projects\SmartLampMatter\src\scripts\scan_mdns.py:
16: FutureWarning: <__main__.Listener object at 0x000001EEC46457F0> has no update_serv
ice method. Provide one (it can be empty if you don't care about the updates), it'll b
ecome mandatory.
  ServiceBrowser(zc, "_matter._udp.local.", Listener())
Scanning for 5 s …

Found: smartlamp._matter._udp.local.
  ↳ IPv4: 192.168.105.22
  ↳ PI = 20202021
  ↳ CRI = 300
  ↳ SII = 4520
  ↳ DN = SmartLamp
  ↳ CM = 2
  ↳ DT = 0x0101
  ↳ VP = 0xFFF1+0x8000
PS C:\Users\sinab\Documents\PlatformIO\Projects\SmartLampMatter> 
```

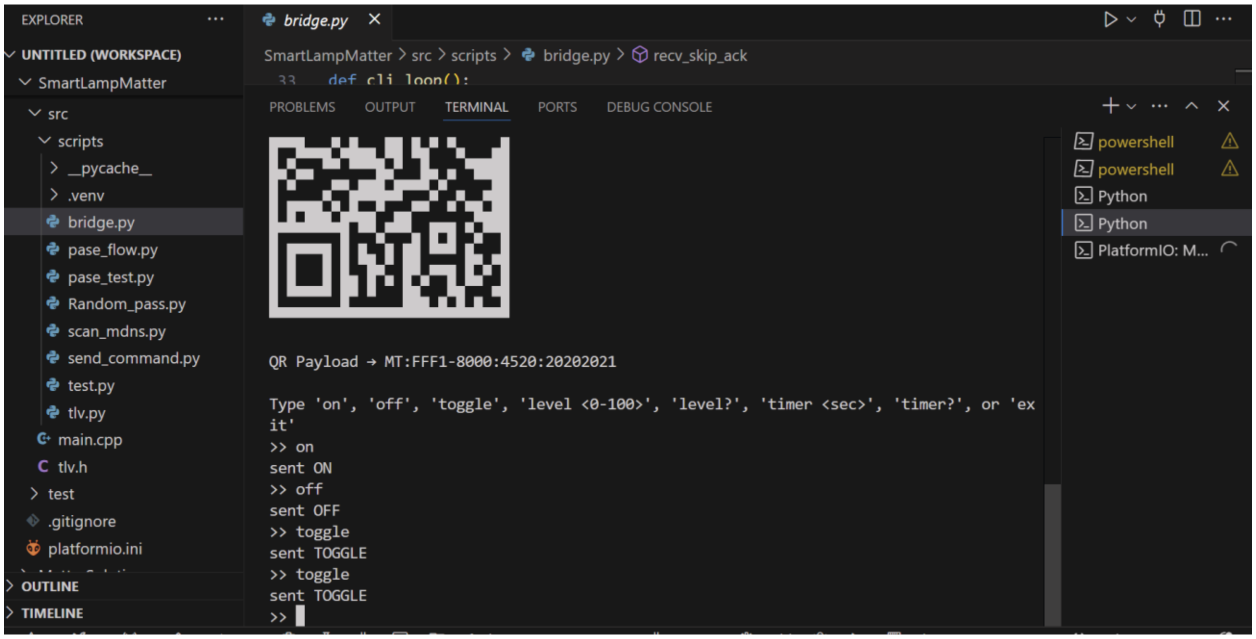
3.2 Commands and Responses (`Running bridge.py as CLI`)

Launch:

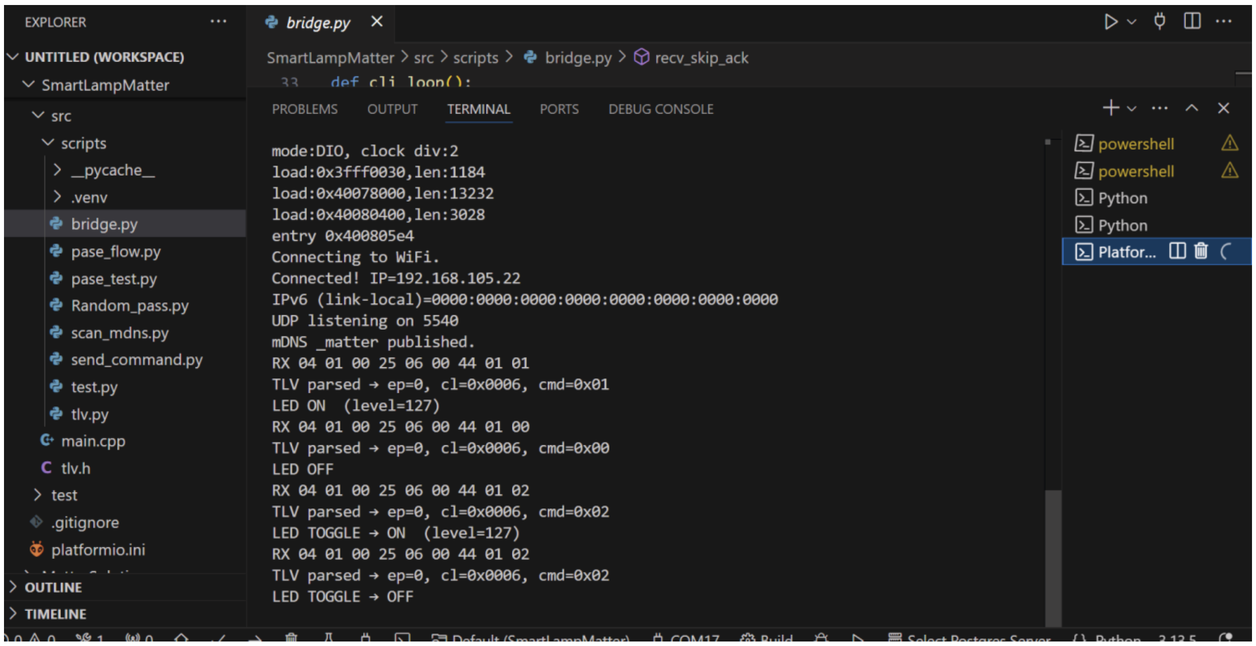


```
  ↳ VP = 0xFFF1+0x8000
PS C:\Users\sinab\Documents\PlatformIO\Projects\SmartLampMatter> & C:/Python313/python
.exe c:/Users/sinab/Documents/PlatformIO/Projects/SmartLampMatter/src/scripts/bridge.p
y

QR Payload → MT:FFF1-8000:4520:20202021

Type 'on', 'off', 'toggle', 'level <0-100>', 'level?', 'timer <sec>', 'timer?', or 'ex
it'
```

Turn ON, OFF and TOGGLE:

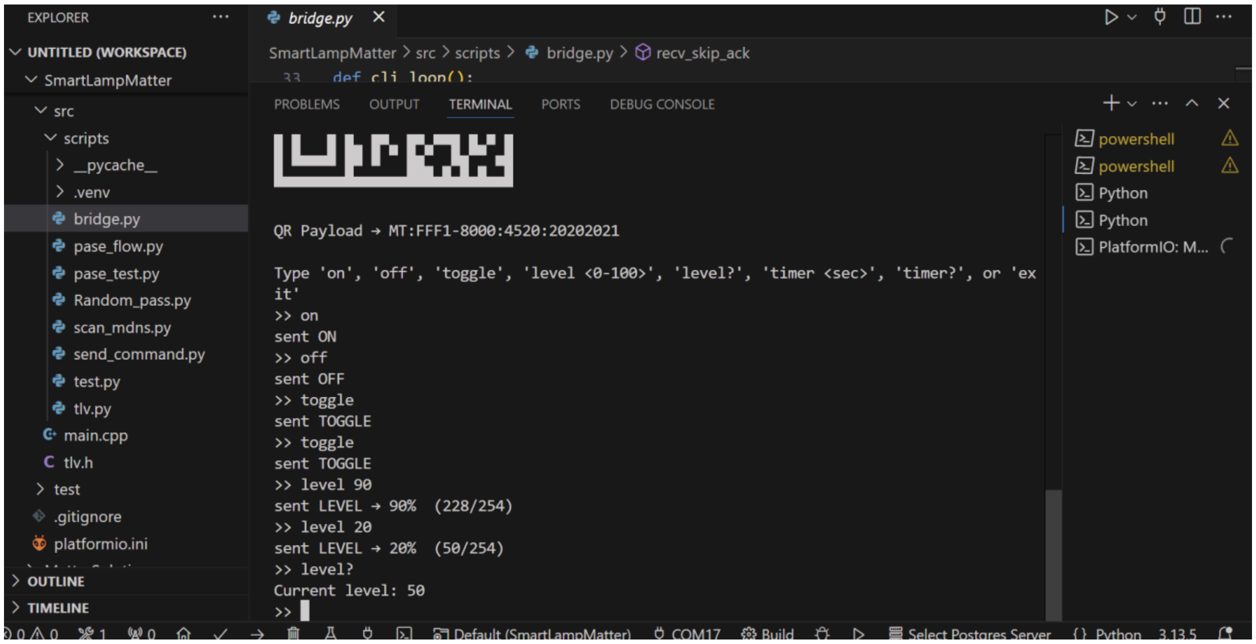


```
QR Payload → MT:FFF1-8000:4520:20202021

Type 'on', 'off', 'toggle', 'level <0-100>', 'level?', 'timer <sec>', 'timer?', or 'ex
it'
>> on
sent ON
>> off
sent OFF
>> toggle
sent TOGGLE
>> toggle
sent TOGGLE
>>
```

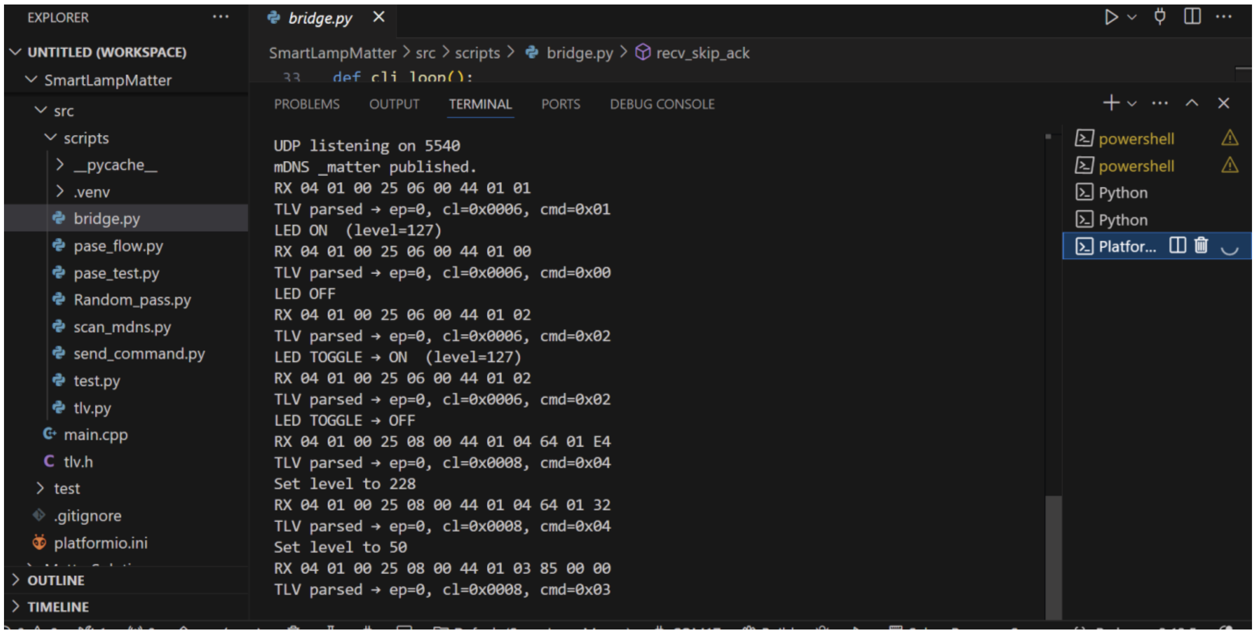


```
mode:DIO, clock div:2
load:0x3fff0030,len:1184
load:0x40078000,len:13232
load:0x40080400,len:3028
entry 0x400805e4
Connecting to WiFi.
Connected! IP=192.168.105.22
IPv6 (link-local)=0000:0000:0000:0000:0000:0000:0000:0000
UDP listening on 5540
mDNS _matter published.
RX 04 01 00 25 06 00 44 01 01
TLV parsed → ep=0, cl=0x0006, cmd=0x01
LED ON  (level=127)
RX 04 01 00 25 06 00 44 01 00
TLV parsed → ep=0, cl=0x0006, cmd=0x00
LED OFF
RX 04 01 00 25 06 00 44 01 02
TLV parsed → ep=0, cl=0x0006, cmd=0x02
LED TOGGLE → ON  (level=127)
RX 04 01 00 25 06 00 44 01 02
TLV parsed → ep=0, cl=0x0006, cmd=0x02
LED TOGGLE → OFF
```

Set and Read the Level of the brightness:

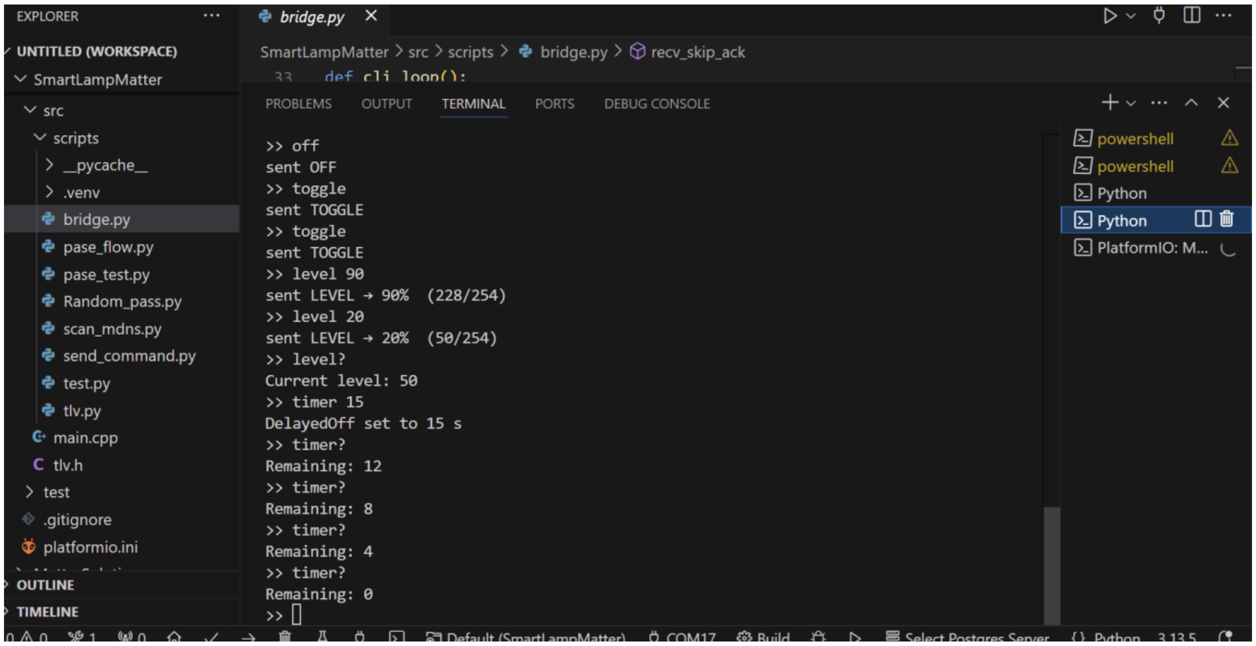


```
QR Payload → MT:FFF1-8000:4520:20202021

Type 'on', 'off', 'toggle', 'level <0-100>', 'level?', 'timer <sec>', 'timer?', or 'ex
it'
>> on
sent ON
>> off
sent OFF
>> toggle
sent TOGGLE
>> toggle
sent TOGGLE
>> level 90
sent LEVEL → 90%  (228/254)
>> level 20
sent LEVEL → 20%  (50/254)
>> level?
Current level: 50
>>
```

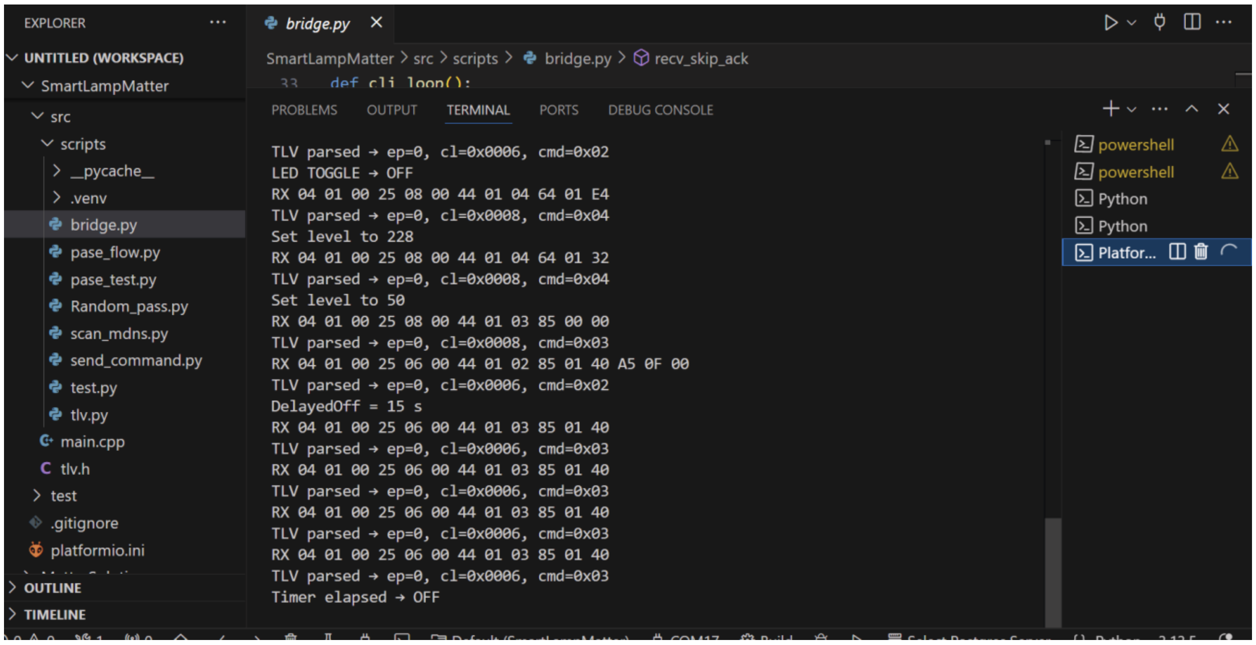


```
UDP listening on 5540
mDNS _matter published.
RX 04 01 00 25 06 00 44 01 01
TLV parsed → ep=0, cl=0x0006, cmd=0x01
LED ON  (level=127)
RX 04 01 00 25 06 00 44 01 00
TLV parsed → ep=0, cl=0x0006, cmd=0x00
LED OFF
RX 04 01 00 25 06 00 44 01 02
TLV parsed → ep=0, cl=0x0006, cmd=0x02
LED TOGGLE → ON  (level=127)
RX 04 01 00 25 06 00 44 01 02
TLV parsed → ep=0, cl=0x0006, cmd=0x02
LED TOGGLE → OFF
RX 04 01 00 25 08 00 44 01 04 64 01 E4
TLV parsed → ep=0, cl=0x0008, cmd=0x04
Set level to 228
RX 04 01 00 25 08 00 44 01 04 64 01 32
TLV parsed → ep=0, cl=0x0008, cmd=0x04
Set level to 50
RX 04 01 00 25 08 00 44 01 03 85 00 00
TLV parsed → ep=0, cl=0x0008, cmd=0x03
```

Set and Read the Timer for turning OFF:

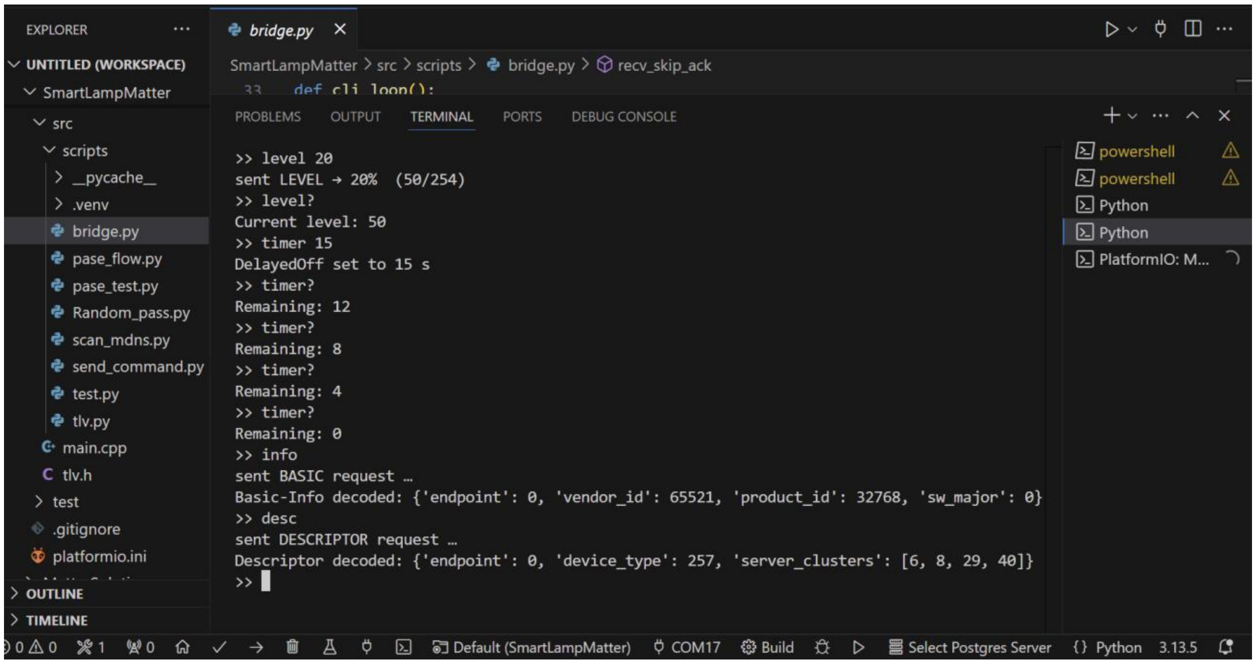


```
>> off
sent OFF
>> toggle
sent TOGGLE
>> toggle
sent TOGGLE
>> level 90
sent LEVEL → 90%  (228/254)
>> level 20
sent LEVEL → 20%  (50/254)
>> level?
Current level: 50
>> timer 15
DelayedOff set to 15 s
>> timer?
Remaining: 12
>> timer?
Remaining: 8
>> timer?
Remaining: 4
>> timer?
Remaining: 0
>>
```

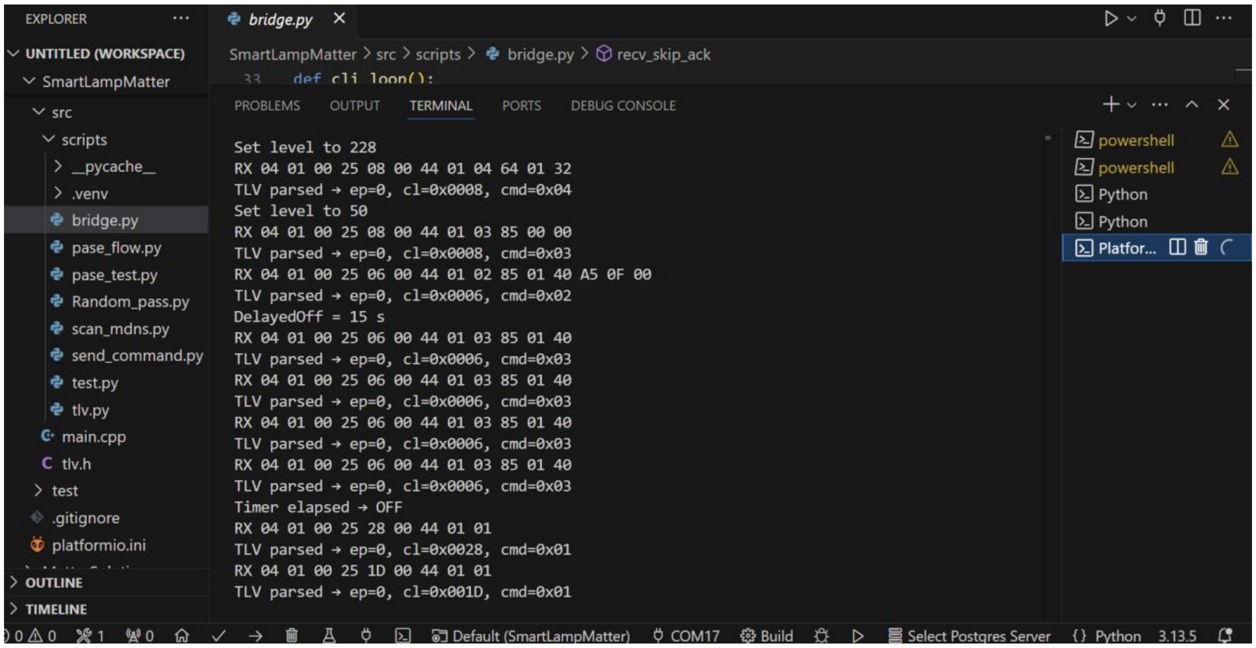


```
TLV parsed → ep=0, cl=0x0006, cmd=0x02
LED TOGGLE → OFF
RX 04 01 00 25 08 00 44 01 04 64 01 E4
TLV parsed → ep=0, cl=0x0008, cmd=0x04
Set level to 228
RX 04 01 00 25 08 00 44 01 04 64 01 32
TLV parsed → ep=0, cl=0x0008, cmd=0x04
Set level to 50
RX 04 01 00 25 08 00 44 01 03 85 00 00
TLV parsed → ep=0, cl=0x0008, cmd=0x03
RX 04 01 00 25 06 00 44 01 02 85 01 40 A5 0F 00
TLV parsed → ep=0, cl=0x0006, cmd=0x02
DelayedOff = 15 s
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
Timer elapsed → OFF
```

Basic info and Descriptor:

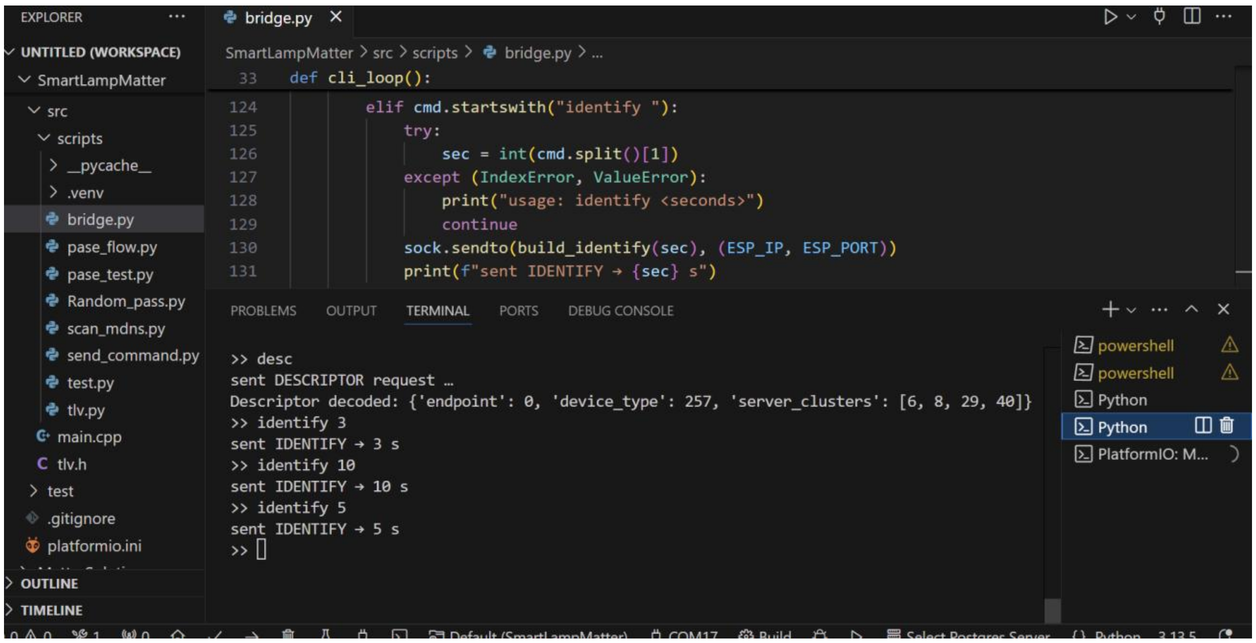


```
>> level 20
sent LEVEL → 20%  (50/254)
>> level?
Current level: 50
>> timer 15
DelayedOff set to 15 s
>> timer?
Remaining: 12
>> timer?
Remaining: 8
>> timer?
Remaining: 4
>> timer?
Remaining: 0
>> info
sent BASIC request …
Basic-Info decoded: {'endpoint': 0, 'vendor_id': 65521, 'product_id': 32768, 'sw_major': 0}
>> desc
sent DESCRIPTOR request …
Descriptor decoded: {'endpoint': 0, 'device_type': 257, 'server_clusters': [6, 8, 29, 40]}
>>
```



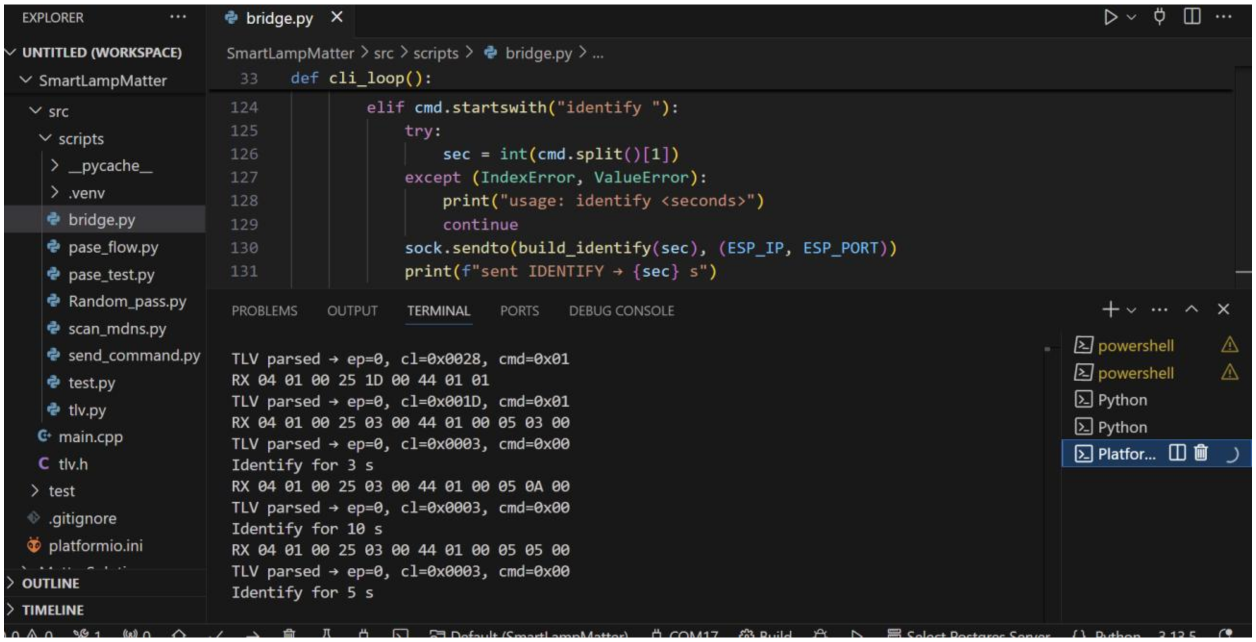
```
Set level to 228
RX 04 01 00 25 08 00 44 01 04 64 01 32
TLV parsed → ep=0, cl=0x0008, cmd=0x04
Set level to 50
RX 04 01 00 25 08 00 44 01 03 85 00 00
TLV parsed → ep=0, cl=0x0008, cmd=0x03
RX 04 01 00 25 06 00 44 01 02 85 01 40 A5 0F 00
TLV parsed → ep=0, cl=0x0006, cmd=0x02
DelayedOff = 15 s
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
RX 04 01 00 25 06 00 44 01 03 85 01 40
TLV parsed → ep=0, cl=0x0006, cmd=0x03
Timer elapsed → OFF
RX 04 01 00 25 28 00 44 01 01
TLV parsed → ep=0, cl=0x0028, cmd=0x01
RX 04 01 00 25 1D 00 44 01 01
TLV parsed → ep=0, cl=0x001D, cmd=0x01
```

Identify (Blinking):



```python
def cli_loop():
        elif cmd.startswith("identify "):
            try:
                sec = int(cmd.split()[1])
            except (IndexError, ValueError):
                print("usage: identify <seconds>")
                continue
            sock.sendto(build_identify(sec), (ESP_IP, ESP_PORT))
            print(f"sent IDENTIFY → {sec} s")
```

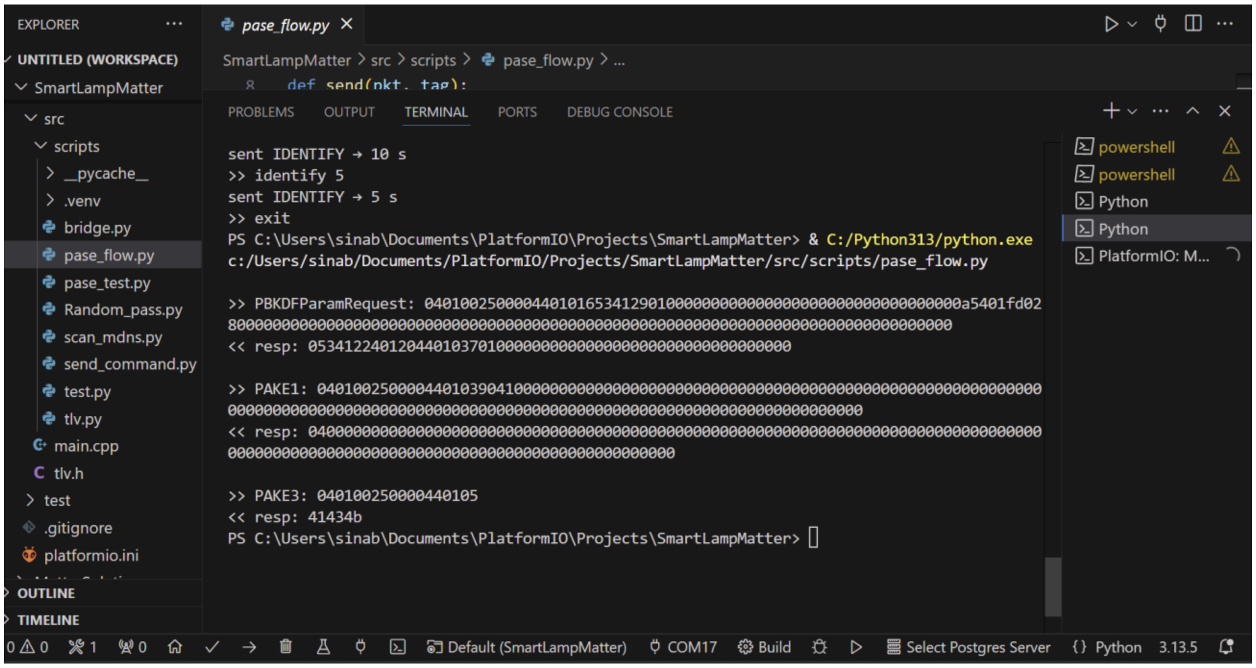
```
>> desc
sent DESCRIPTOR request …
Descriptor decoded: {'endpoint': 0, 'device_type': 257, 'server_clusters': [6, 8, 29, 40]}
>> identify 3
sent IDENTIFY → 3 s
>> identify 10
sent IDENTIFY → 10 s
>> identify 5
sent IDENTIFY → 5 s
>> 
```



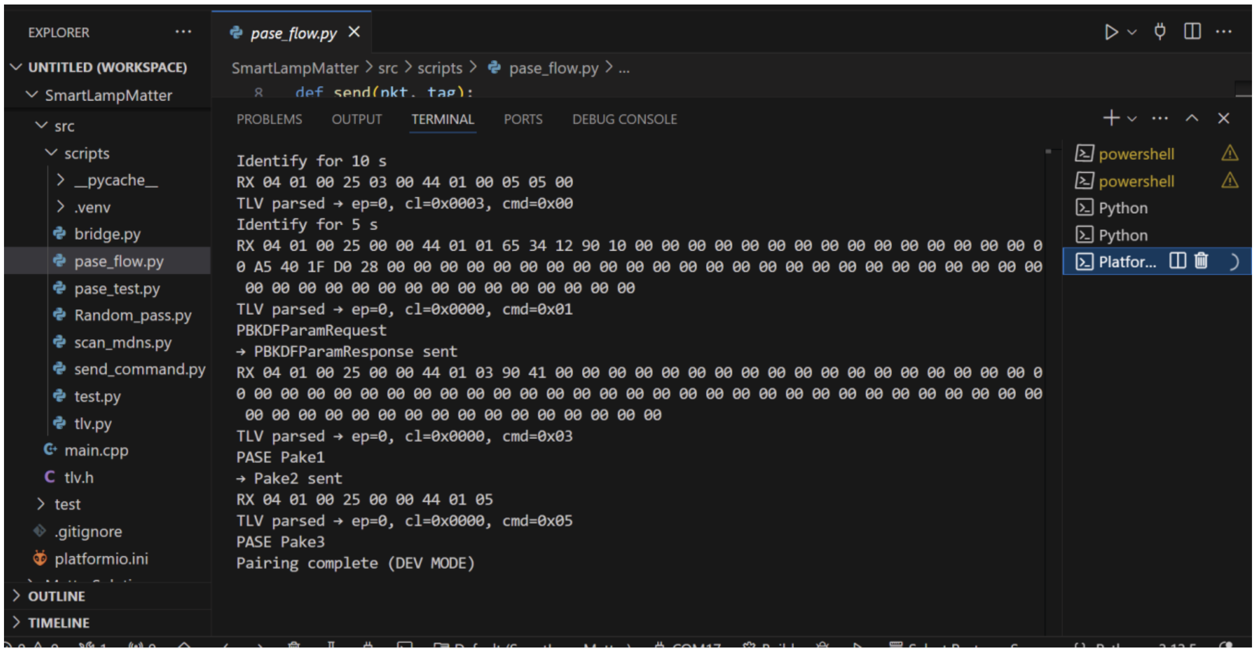
```
def cli_loop():
        elif cmd.startswith("identify "):
            try:
                sec = int(cmd.split()[1])
            except (IndexError, ValueError):
                print("usage: identify <seconds>")
                continue
            sock.sendto(build_identify(sec), (ESP_IP, ESP_PORT))
            print(f"sent IDENTIFY → {sec} s")
```

```
TLV parsed → ep=0, cl=0x0028, cmd=0x01
RX 04 01 00 25 1D 00 44 01 01
TLV parsed → ep=0, cl=0x001D, cmd=0x01
RX 04 01 00 25 03 00 44 01 00 05 03 00
TLV parsed → ep=0, cl=0x0003, cmd=0x00
Identify for 3 s
RX 04 01 00 25 03 00 44 01 00 05 0A 00
TLV parsed → ep=0, cl=0x0003, cmd=0x00
Identify for 10 s
RX 04 01 00 25 03 00 44 01 00 05 05 00
TLV parsed → ep=0, cl=0x0003, cmd=0x00
Identify for 5 s
```

3.3 PASE and Matter pairing simple simulation (`Running pase_flow.py`)



```
sent IDENTIFY → 10 s
>> identify 5
sent IDENTIFY → 5 s
>> exit
PS C:\Users\sinab\Documents\PlatformIO\Projects\SmartLampMatter> & C:/Python313/python.exe
c:/Users/sinab/Documents/PlatformIO/Projects/SmartLampMatter/src/scripts/pase_flow.py

>> PBKDFParamRequest: 040100250000440101653412901000000000000000000000000000000000a5401fd02
800000000000000000000000000000000000000000000000000000000000000000000000000000000
<< resp: 053412240120440103701000000000000000000000000000000000

>> PAKE1: 040100250000440103904100000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000
<< resp: 0400000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000

>> PAKE3: 040100250000440105
<< resp: 41434b
PS C:\Users\sinab\Documents\PlatformIO\Projects\SmartLampMatter> 
```



```
def send(pkt, tag):
```

```
Identify for 10 s
RX 04 01 00 25 03 00 44 01 00 05 05 00
TLV parsed → ep=0, cl=0x0003, cmd=0x00
Identify for 5 s
RX 04 01 00 25 00 00 44 01 01 65 34 12 90 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 A5 40 1F D0 28 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
TLV parsed → ep=0, cl=0x0000, cmd=0x01
PBKDFParamRequest
→ PBKDFParamResponse sent
RX 04 01 00 25 00 00 44 01 03 90 41 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
TLV parsed → ep=0, cl=0x0000, cmd=0x03
PASE Pake1
→ Pake2 sent
RX 04 01 00 25 00 00 44 01 05
TLV parsed → ep=0, cl=0x0000, cmd=0x05
PASE Pake3
Pairing complete (DEV MODE)
```

---

4 Integration Attempts with Apple and Google Home

During development, multiple approaches were attempted to connect our custom smart lamp firmware—which implements a simplified Matter-like TLV interface over UDP, without the official ESP-Matter solution—to Apple HomeKit and Google Home Dev Mode. Each method failed for specific reasons:

4.1 setupPayload Library in `bridge.py`

What was tried: In the Python bridge script (`bridge.py`), we used a setup-payload library (e.g. `matter-setup-payload`) to generate a QR code so that the Home app could initiate the Matter commissioning process.

Why it failed: Apple HomeKit (and Google Home) require, in addition to the QR code, a full secure PASE → CASE exchange and valid device attestation certificates (DAC/PAI) issued by the Connectivity Standards Alliance. Our bridge only simulated PBKDF and initial PASE steps and did not implement CASE or provide real certificates; consequently, the pairing sequence aborted before completion.

4.2 Using `chip-tool` (Matter CLI)

What was tried: We attempted to build and run the official ConnectedHomeIP CLI (`chip-tool`) on Windows/WSL to perform a fully compliant Matter commissioning.

Why it failed: 1) The tool is designed to compile on Linux/macOS with GN/Ninja and Python; Windows dependency issues prevented a successful build. 2) Even if it ran, it still requires a server endpoint for CASE key distribution and attestation. 3) We did not possess valid DAC/PAI certificates for our hardware, so the full commissioning flow could not complete.

4.3 Node-RED Bridge

What was tried: We set up a portable Node-RED instance with the `node-red-matter` palette. A Function node forwarded On/Off and Level commands via UDP to the ESP32 lamp.

Why it failed: 1) Installing npm and the palette on Windows was repeatedly blocked by firewall/network policies and PowerShell execution restrictions. 2) Even when Node-RED launched, the editor path was non-standard, and multiple URL permutations (“/”, “/red”, “/admin”) failed to load. 3) Node-RED does not implement the secure CASE channel; it serves only as an insecure proxy, which HomeKit rejects.

4.4 PlatformIO Build of ESP-Matter Lighting Example

What was tried: We configured a PlatformIO project in VS Code to build the ESP-Matter `lighting` example, aiming for a native Matter implementation on the ESP32.

Why it failed: Downloading and resolving all dependencies via PlatformIO on Windows proved fragile, with repeated and inconsistent build failures and obscure error messages that could not be resolved within the project timeframe.

4.5 ESP-IDF Native Build

What was tried: We cloned the complete ESP-Matter repository and attempted to build using ESP-IDF, both as a full clone and as a shallow clone to reduce download size.

Why it failed: 1) Cloning the full repository over our network repeatedly stalled due to filtering, requiring VPN usage and multiple retries. 2) Even after dependency retrieval succeeded, the build process failed at unspecified stages with no clear errors to debug.

Despite exhaustive efforts with these non-official approaches, none succeeded in establishing a Matter-compliant connection to Apple HomeKit or Google Home Dev Mode. Only a fully native ESP-Matter implementation with valid attestation and CASE security channels can meet the requirements for successful pairing.