Prueba técnica (30–40 min) — Programación “embedded” en Python
Contexto
Simula el control de un submódulo de cargador con:

- Entradas: `PILOT_OK`, `FAULT`, `BTN` (botón).
- Salidas: `CONTACTOR`, `LED`.
- Una **CLI por consola** (como si fuera UART).

Objetivo
Implementar en Python una **máquina de estados no bloqueante** con **debounce** de botón y una **CLI** mínima para inspección/forzado.
Estados

- `IDLE`: contactor OFF, LED parpadeo lento (1 Hz).
- `READY`: pilot OK, esperando usuario (LED fijo).
- `CHARGING`: contactor ON (si no hay fallo), LED parpadeo rápido (4 Hz).
- `FAULT`: contactor OFF, LED doble destello cada segundo.

Reglas de transición

1. Arranque → `IDLE`.
2. `IDLE` → `READY` cuando `PILOT_OK=1`.
3. `READY` → `CHARGING` al pulsar `BTN` (borde ascendente, **debounce** ≥20 ms).
4. En cualquier estado, si `FAULT=1` → `FAULT`. Desde `FAULT`, al despejarse `FAULT` y con `BTN` se vuelve a `IDLE`.
5. En `CHARGING`, si `PILOT_OK=0` → `IDLE`.
6. Sin `sleep` bloqueantes largos: usa un **tick** o `asyncio` (pasos cada ~10 ms).

CLI (líneas terminadas en `\n`)

- `GET STATE` → imprime estado actual.
- `GET IO` → imprime `PILOT_OK`, `FAULT`, `BTN`, `CONTACTOR`, `LED`.
- `SET LED <OFF|SLOW|FAST|FAULT>` → fuerza patrón LED.
- `SET IN <PILOT_OK|FAULT|BTN> <0|1>` → simula entradas.
- `HELP`

No hace falta robustez completa: parsing simple con `split()` está bien.
Requisitos técnicos

- Python 3.10+.
- Sin dependencias externas (solo stdlib).
- Arquitectura por capas:
  - `drivers/` (gpio simulado, reloj/tick, consola/uart).
  - `app/` (state_machine, led, cli).
  - `main.py` orquestando el lazo.
- **Debounce** por tiempo (≥20 ms).
- LED por patrón temporizado (no bloqueante).
- Logs/prints claros de cambios de estado y salidas.

Estructura recomendada

```
/src
  main.py
  app/state_machine.py
  app/cli.py
  app/led.py
  drivers/gpio.py
  drivers/clock.py
  drivers/uart.py
/README.md
```

Interfaces sugeridas (stubs)

```
# drivers/gpio.py
```

```
class GPIO:
    def __init__(self):
        self._pilot_ok = 0
        self._fault = 0
        self._btn_raw = 0
        self.contactor = 0
        self.led = 0
    def read_pilot_ok(self) -> int: return self._pilot_ok
    def read_fault(self) -> int: return self._fault
    def read_btn_raw(self) -> int: return self._btn_raw
    def write_contactor(self, on: int): self.contactor = 1 if on else 0
    def write_led(self, on: int): self.led = 1 if on else 0
    # setters para simular entradas:
    def set_input(self, name: str, val: int): setattr(self, f"_{name.lower()}", 1 if val else
0)

# drivers/clock.py
import time
def tick_ms() -> int:
    return int(time.monotonic() * 1000)

# app/led.py
from enum import Enum
class LedMode(Enum): OFF=0; SLOW=1; FAST=2; FAULT=3
class LedController:
    def __init__(self, gpio, clock=tick_ms): ...
    def set_mode(self, mode: LedMode): ...
    def step(self): ...  # alterna gpio.write_led según patrón

# app/state_machine.py
from enum import Enum
class State(Enum): IDLE=0; READY=1; CHARGING=2; FAULT=3
class StateMachine:
    def __init__(self, gpio, led, clock=tick_ms):
        self.state = State.IDLE
        # debounce internos
    def _debounce_btn(self) -> int: ...
    def step(self): ...
```

Bucle principal (dos opciones)

- **Opción A (simple)**: lazo con `time.sleep(0.01)` para ~10 ms entre pasos (no bloqueante para lógica; la CLI lee líneas cuando hay input).
- **Opción B (asyncio)**: tareas concurrentes `state_task()` (cada 10 ms) y `cli_task()` leyendo `stdin`.

CLI ejemplo (mínimo)

```
# app/cli.py
def process_line(line: str, sm, gpio, led):
    parts = line.strip().split()
    if parts == ["GET", "STATE"]:
        print(sm.state.name)
    elif parts == ["GET", "IO"]:
        print(f"PILOT_OK={gpio.read_pilot_ok()} FAULT={gpio.read_fault()}
BTN_RAW={gpio.read_btn_raw()} "
              f"CONTACTOR={gpio.contactor} LED={gpio.led}")
    elif parts[:2] == ["SET", "LED"]:
        ...
    elif parts[:2] == ["SET", "IN"]:
        ...
    else:
        print("Comandos: GET STATE | GET IO | SET LED <OFF|SLOW|FAST|FAULT> | SET IN
<PILOT_OK|FAULT|BTN> <0|1>")
```

Entregables

1. **Repositorio** con:

- Código fuente.

- `README.md` con:

- Cómo ejecutar (`python -m src.main` o `python src/main.py`).
- Descripción breve de la arquitectura.

- Qué quedó implementado y qué faltaría si no da tiempo.

2. **Opcional**: tests rápidos con `unittest` (p. ej., transición `IDLE→READY→CHARGING`, y debounce).

Cómo enviarlo

- Subir a un **repositorio** (GitHub/GitLab/Bitbucket) y compartir el enlace.
- Se valoran commits atómicos y mensajes claros (aunque sea una prueba corta).