Universidad de San Carlos de Guatemala.
Facultad de ingeniería.
Escuela de ingeniería en ciencias y sistemas.
Área: Ciencias de la computación
Nombre del curso: Arquitectura de computadores y ensambladores 1
Catedrático: m.sc. Luis Fernando Espino Barrios
Auxiliar: Carlos Antonio Velásquez Castellanos

Manual Técnico

No.	Nombre:	Carné
1	Henderson Migdo Baten Hernandez	201019694
2	Selim Idair Ergon Castillo	201801300
3	Jemima Solmaira Chavajay Quiejú	201801521
4	Giovanni Saul Concohá Cax	202100229
5	Johan Moises Cardona Rosales	202201405
6	Estiben Yair Lopez Leveron	202204578
7	Santiago Julián Barrera Reyes	201905884

16 de junio, 2024

Contenido

Introducción

Este manual técnico describe el diseño, desarrollo e implementación de un sistema avanzado de monitorización y control de una sucursal utilizando una Raspberry Pi. Este proyecto surge de la necesidad de una empresa de modernizar su infraestructura para mejorar la monitorización de dispositivos electrónicos y facilitar el control remoto mediante una solución práctica.

La Raspberry Pi, conocida por su versatilidad, se utiliza para integrar diversos sensores y actuadores, que permiten la interacción con el entorno físico de la sucursal. Estos componentes incluyen LEDs para iluminación, sensores de presencia, motores para automatización, y otros dispositivos de entrada y salida. Todos estos elementos están conectados y controlados a través de una interfaz web, desarrollada para ser accesible desde cualquier navegador, lo que proporciona una solución centralizada y eficiente para la gestión de la sucursal.

El sistema propuesto no solo mejora la accesibilidad y el control de los dispositivos electrónicos, sino que también ofrece una manera de recolectar y analizar datos en tiempo real, lo que puede ser invaluable para la toma de decisiones y la mejora continua de las operaciones. Este manual proporciona una guía completa para la implementación del proyecto, desde la configuración del hardware y software hasta la creación y despliegue de la interfaz web, asegurando que los usuarios puedan replicar y mantener el sistema de manera efectiva.

Arquitectura del Sistema

Raspberry Pi

La Raspberry Pi es el componente central del sistema, actuando como el cerebro que controla todos los sensores y actuadores conectados. Utiliza sus pines GPIO (General Purpose Input/Output) para comunicarse con los componentes periféricos y ejecutar el servidor web que permite la interacción del usuario con el sistema.

Sensores y Actuadores

1. **LEDs**
    - **Función:** Iluminación general y exterior.
    - **Interconexión:** Conectados a los pines GPIO de la Raspberry Pi. Requieren resistencias adecuadas para limitar la corriente y proteger los LEDs.
2. **Fotorresistencia**
    - **Función:** Detecta los niveles de luz ambiental para el control automático de la iluminación exterior.
    - **Interconexión:** Conectada a un pin GPIO a través de un convertidor ADC (Analog-to-Digital Converter), ya que la Raspberry Pi no tiene entradas analógicas.
3. **Sensores de Detección**
    - **Función:** Detectan la presencia de personas en la entrada de la sucursal.
    - **Interconexión:** Conectados directamente a los pines GPIO de la Raspberry Pi. Normalmente, utilizan módulos de detección de movimiento (PIR) o sensores infrarrojos.
4. **Motor de Portón**
    - **Función:** Controla la apertura y cierre del portón del área de carga y descarga.
    - **Interconexión:** Conectado a la Raspberry Pi a través de un controlador de motor (como un L298N o un puente H) que permite el control de la dirección y velocidad del motor.
5. **Buzzer**
    - **Función:** Emite sonidos de alerta para notificaciones.
    - **Interconexión:** Conectado directamente a un pin GPIO de la Raspberry Pi. Puede ser activado mediante una señal de salida digital.
6. **Display de 7 Segmentos**
    - **Función:** Muestra el conteo de clientes en la sucursal.
    - **Interconexión:** Conectado a los pines GPIO de la Raspberry Pi. Puede requerir un controlador adicional como un 74HC595 para manejar múltiples segmentos con menos pines.

7. **Láser Perimetral**
    - **Función:** Utilizado para la detección de intrusos en el perímetro de la sucursal.
    - **Interconexión:** Conectado a los pines GPIO. Normalmente, se utiliza junto con un receptor fotosensible que detecta la interrupción del rayo láser.
8. **Pantalla LCD**
    - **Función:** Muestra información relevante sobre el estado del sistema, como el conteo de clientes y la activación de dispositivos.
    - **Interconexión:** Conectada a la Raspberry Pi mediante la interfaz I2C, que permite la comunicación utilizando solo dos cables (SDA y SCL).

Servidor Web (Backend y Frontend)

1. **Backend (Flask)**
    - **Función:** Procesa las solicitudes de la interfaz web y controla los sensores y actuadores conectados a la Raspberry Pi.
    - **Interconexión:** Ejecutado en la Raspberry Pi. Se comunica con los scripts de Python que manejan la lógica del sistema y las operaciones de entrada/salida a través de los pines GPIO.
    - **Dependencias:** Requiere la instalación de Flask y otras bibliotecas de Python para la interacción con el hardware (por ejemplo, RPi.GPIO para el control de pines GPIO).
    -
2. **Frontend (React)**
    - **Función:** Proporciona una interfaz de usuario amigable y accesible desde cualquier navegador. Permite a los usuarios interactuar con el sistema para controlar y monitorear los dispositivos.
    - **Interconexión:** Se comunica con el backend Flask a través de solicitudes HTTP (REST API). El frontend envía comandos al backend y recibe datos en tiempo real para actualizar la interfaz.
    - **Componentes:** Incluye varias páginas y componentes de React para diferentes funcionalidades como el control de iluminación, monitoreo de clientes, control del portón, y más.

Conexiones y Comunicación

- **GPIO (General Purpose Input/Output):** Utilizado para la conexión directa de sensores y actuadores. Proporciona la capacidad de leer datos de entrada (como la detección de presencia) y enviar señales de salida (como encender un LED).
- **I2C (Inter-Integrated Circuit):** Protocolo de comunicación utilizado para conectar dispositivos como la pantalla LCD a la Raspberry Pi. Permite la

comunicación entre múltiples dispositivos usando solo dos cables (SDA y SCL).

- **UART (Universal Asynchronous Receiver-Transmitter):** Utilizado para la comunicación serial entre la Raspberry Pi y otros dispositivos.
- **HTTP (Hypertext Transfer Protocol):** Utilizado por el servidor web para la comunicación entre el frontend (React) y el backend (Flask). Las solicitudes HTTP permiten enviar comandos y recibir datos en tiempo real.

.

Especificaciones del Hardware

Listado de componentes utilizados

- Raspberry Pi 3 o superior
- Sensor de fotoresistencia
- Pantalla LCD 16x2
- Motores (tipos y cantidades)
- Sensores para detección de personas
- Láser perimetral
- Buzzer
- Display de 7 segmentos
- LEDs

Especificaciones del Software

El entorno de desarrollo para este proyecto está compuesto por varias herramientas y lenguajes que facilitan la creación y mantenimiento del sistema. La Raspberry Pi ejecuta un sistema operativo, generalmente Raspberry Pi OS, que proporciona un entorno flexible y potente para desarrollar tanto el backend como el frontend del sistema.

1. **Sistema Operativo:** Raspberry Pi OS (anteriormente Raspbian), una distribución de Linux optimizada para la Raspberry Pi.
2. **Lenguaje de Programación (Backend):** Python
3. **Framework de Desarrollo Web (Backend):** Flask
4. **Lenguaje de Programación (Frontend):** JavaScript
5. **Framework de Desarrollo Web (Frontend):** React
6. **Editor de Código:** Visual Studio Code, PyCharm, o cualquier otro editor de código que soporte Python y JavaScript.
7. **Control de versiones:** Git, utilizando plataformas como GitHub o GitLab para la gestión del código fuente.

Python para Backend

El backend del sistema está desarrollado en Python, utilizando el microframework Flask. Python es elegido por su simplicidad y robustez, permitiendo un rápido desarrollo y fácil integración con el hardware de la Raspberry Pi.

Estructura del código

app.py

Inicialización y Configuración

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
import RPi.GPIO as GPIO
import sys
import time
import threading

# Inicializa la aplicación Flask
app = Flask(__name__)
# Habilita CORS para todas las rutas que comiencen con /api/
cors = CORS(app, resources={r"/api/*": {"origins": "*"}})
```

Importaciones:

Flask, request y jsonify son módulos de Flask utilizados para crear una aplicación web y manejar solicitudes y respuestas.
CORS se utiliza para permitir solicitudes de diferentes orígenes (Cross-Origin Resource Sharing).
RPi.GPIO es la librería que permite controlar los pines GPIO de la Raspberry Pi.
sys, time, y threading son módulos estándar de Python para manejar argumentos de línea de comandos, gestionar el tiempo y manejar hilos, respectivamente.

Inicialización de Flask:
app = Flask(__name__) crea una instancia de la aplicación Flask.
cors = CORS(app, resources={r"/api/*": {"origins": "*"}}) habilita CORS para todas las rutas que comiencen con /api/.

Declaración y Configuración de Pines GPIO

```
leds = []
estado_motor = None
GPIO.setmode(GPIO.BOARD)
GPIO.setwarnings(False)

# Declaración de pines GPIO
LED1 = 11
MOTOR = 13
PIN_IN1_STEPPER = 31
PIN_IN2_STEPPER = 33
PIN_IN3_STEPPER = 35
PIN_IN4_STEPPER = 37
StepPins = [PIN_IN1_STEPPER, PIN_IN2_STEPPER, PIN_IN3_STEPPER,
PIN_IN4_STEPPER]
```

Variables Globales:
leds: Lista para almacenar el estado de los LEDs.
estado_motor: Variable para almacenar el estado del motor.

Configuración GPIO
GPIO.setmode(GPIO.BOARD): Configura el uso de la numeración física de los pines de la Raspberry Pi.
GPIO.setwarnings(False): Desactiva las advertencias de GPIO.
Definición de Pines: Se asignan números de pines GPIO específicos para los LEDs y el motor paso a paso.

Configuración del Motor Paso a Paso

```
Seq = [[1,0,0,1], [1,0,0,0], [1,1,0,0], [0,1,0,0], [0,1,1,0], [0,0,1,0],
[0,0,1,1], [0,0,0,1]]
StepCount = len(Seq)
StepDir = 1
StepCounter = 0

# Configuración del tiempo de espera
if len(sys.argv) > 1:
    WaitTime = int(sys.argv[1]) / float(1000)
else:
    WaitTime = 10 / float(1000)
```

Secuencia del Motor:
Seq: Secuencia de pasos para controlar el motor paso a paso.
StepCount: Número de pasos en la secuencia.
StepDir: Dirección de rotación (1 para horario, -1 para antihorario).
StepCounter: Contador de pasos.

Tiempo de Espera:
WaitTime: Tiempo de espera entre pasos, configurable mediante argumentos de línea de comandos.

Control de Hilos

```
running = False
pause = threading.Event()
pause.set()
iniciar_stepper = True
crear = True
```

Variables de Control de Hilos:
running: Indica si el motor está en funcionamiento.
pause: Evento para pausar o reanudar el motor.
iniciar_stepper: Controla la inicialización del motor paso a paso.
crear: Indica si la aplicación Flask debe ser creada.

Funciones para Controlar GPIO y el Motor

```
def controlar_gpio(puerto, estado):
    if puerto == 1:
        GPIO.output(LED1, estado)
    elif puerto == 2:
        GPIO.output(MOTOR, estado)
    else:
        print("No existe el puerto para activarlo.")
```

Función para Controlar GPIO:
controlar_gpio(puerto, estado): Activa o desactiva un puerto GPIO específico

```
def activar_motor_stepper():
    global StepCount, StepCounter
    while running:
```

```
        pause.wait()
        for pin in range(4):
            xpin = StepPins[pin]
            GPIO.output(xpin, Seq[StepCounter][pin])
        StepCounter += StepDir
        if StepCounter >= StepCount:
            StepCounter = 0
        if StepCounter < 0:
            StepCounter = StepCount + StepDir
        time.sleep(WaitTime)

def start_motor():
    global running
    if not running:
        running = True
        threading.Thread(target=activar_motor_stepper, daemon=True).start()
        print("Motor iniciado")

def stop_motor():
    global running
    running = False
    print("Motor detenido")

def pause_motor():
    pause.clear()
    print("Motor pausado")

def resume_motor():
    pause.set()
    print("Motor reanudado")
```

Funciones para Controlar el Motor Paso a Paso:

activar_motor_stepper(): Ejecuta la secuencia de pasos del motor mientras running sea True.
start_motor(): Inicia el motor creando un nuevo hilo para activar_motor_stepper.
stop_motor(): Detiene el motor estableciendo running en False.
pause_motor(): Pausa el motor utilizando el evento pause.
resume_motor(): Reanuda el motor utilizando el evento pause..
Rutas de la API

```
@app.route('/activarLed', methods=['POST'])
def activar_led():
    global leds
    data = request.json
    cuarto = data.get('cuarto')
    estado = data.get('estado')
    if not isinstance(cuarto, int) or not isinstance(estado, int):
```

```
        return jsonify({"error": "Los parámetros 'cuarto' y 'estado' deben ser
numéricos"}), 400
    found = False
    for led in leds:
        if led['cuarto'] == cuarto:
            led['estado'] = estado
            found = True
            break
    if not found:
        leds.append({"cuarto": cuarto, "estado": estado})
    controlar_gpio(cuarto, estado)
    return jsonify({"mensaje": "Estado del LED actualizado correctamente"}),
200

@app.route('/verEstadoLED', methods=['GET'])
def ver_estado_led():
    global leds
    cuarto = request.args.get('cuarto', type=int)
    if cuarto is None:
        return jsonify({"error": "El parámetro 'cuarto' es necesario y debe
ser numérico"}), 400
    for led in leds:
        if led['cuarto'] == cuarto:
            return jsonify({"cuarto": cuarto, "estado": led['estado']}), 200
    return jsonify({"error": "Cuarto no encontrado"}), 404

@app.route('/api/activarMotor', methods=['POST'])
def activar_motor():
    global estado_motor, iniciar_stepper
    data = request.json
    estado = data.get('estado')
    if not isinstance(estado, int):
        return jsonify({"error": "El parámetro 'estado' debe ser numérico"}),
400
    estado_motor = estado
    if estado_motor == 1:
        start_motor()
    else:
        stop_motor()
    return jsonify({"mensaje": "Estado del motor actualizado correctamente"}),
200

@app.route('/api/verEstadoMotor', methods=['GET'])
def ver_estado_motor():
    global estado_motor
    if estado_motor is None:
        return jsonify({"error": "El estado del motor no ha sido configurado
aún"}), 404
```

```
    return jsonify({"estado_motor": estado_motor}), 200
```

Rutas de la API:
/activarLed: Permite activar o desactivar un LED especificando el número de cuarto y el estado (1 para encender, 0 para apagar).
/verEstadoLED: Permite verificar el estado de un LED en un cuarto específico.
/api/activarMotor: Permite activar o desactivar el motor especificando el estado (1 para encender, 0 para apagar).
/api/verEstadoMotor: Permite verificar el estado del motor.

Configuración Inicial de GPIO

```
def setup():
    GPIO.setup(LED1, GPIO.OUT)
    GPIO.setup(MOTOR, GPIO.OUT)
    GPIO.setup(PIN_IN1_STEPPER, GPIO.OUT)
    GPIO.setup(PIN_IN2_STEPPER, GPIO.OUT)
    GPIO.setup(PIN_IN3_STEPPER, GPIO.OUT)
    GPIO.setup(PIN_IN4_STEPPER, GPIO.OUT)
    GPIO.output(LED1, 0)
    GPIO.output(MOTOR, 0)
    GPIO.output(PIN_IN1_STEPPER, 0)
    GPIO.output(PIN_IN2_STEPPER
```

`luces.py`

Configuración Inicial de GPIO

```
# Configurar la librería RPi.GPIO para usar el número de pin físico
GPIO.setmode(GPIO.BOARD)

# Definir los pines de salida (ajusta los números según tu configuración)
PIN_A = 18  # Pin físico 11 (GPIO 17)
PIN_B = 22  # Pin físico 13 (GPIO 27)
PIN_C = 23  # Pin físico 15 (GPIO 22)

# Configurar los pines como salida
GPIO.setup(PIN_A, GPIO.OUT)
GPIO.setup(PIN_B, GPIO.OUT)
GPIO.setup(PIN_C, GPIO.OUT)
```

Configuración de Modo de Pines:

GPIO.setmode(GPIO.BOARD): Configura el uso de la numeración física de los pines de la Raspberry Pi.
Definición de Pines de Salida:

PIN_A, PIN_B, PIN_C: Asigna pines específicos para controlar el demultiplexor.
Configuración de Pines como Salida:

GPIO.setup(PIN_A, GPIO.OUT): Configura el pin PIN_A como salida.
Similar para PIN_B y PIN_C.

Conversión Decimal a Binario

```
def decimal_to_binary(decimal):
    binary = format(decimal, '03b')
    return [int(bit) for bit in binary]
```

Función decimal_to_binary: Convierte un número decimal (0-7) a una lista de bits binarios (3 bits).

Control del Demultiplexor

```
def set_demultiplexer(value):
    binary_value = decimal_to_binary(value)
    GPIO.output(PIN_A, binary_value[0])
    GPIO.output(PIN_B, binary_value[1])
    GPIO.output(PIN_C, binary_value[2])
```

Función set_demultiplexer: Convierte un valor decimal a binario y establece los pines PIN_A, PIN_B y PIN_C según los bits binarios.

Bucle Principal

```
try:
    while True:
        decimal_input = input("Ingrese un número decimal (0-7): ")
        if decimal_input.isdigit():
            decimal_value = int(decimal_input)
            if 0 <= decimal_value <= 7:
                set_demultiplexer(decimal_value)
            else:
                print("Por favor, ingrese un número entre 0 y 7.")
        else:
            print("Entrada inválida. Por favor, ingrese un número decimal.")
except KeyboardInterrupt:
    print("Programa terminado.")
finally:
    GPIO.cleanup()
```

Bucle Principal:

El bucle pide al usuario un número decimal entre 0 y 7.
decimal_input = input("Ingrese un número decimal (0-7): "): Solicita una entrada del usuario.
if decimal_input.isdigit(): Verifica si la entrada es un número.
decimal_value = int(decimal_input): Convierte la entrada a un entero.
if 0 <= decimal_value <= 7: Verifica si el número está en el rango permitido.
set_demultiplexer(decimal_value): Configura los pines GPIO según el valor binario del número.
Si la entrada no es válida, se muestra un mensaje de error.

Manejo de Interrupciones:
except KeyboardInterrupt: Permite finalizar el programa limpiamente si el usuario presiona Ctrl+C.
finally: GPIO.cleanup(): Limpia la configuración de los pines GPIO cuando el programa termina.

motorStepper.py

```
def __init__(self, step_pins, seq, wait_time=10):
    self.step_pins = step_pins
    self.seq = seq
    self.step_count = len(seq)
    self.step_dir = 1
    self.step_counter = 0
    self.wait_time = wait_time / float(1000)
    self.running = False
    self.pause = threading.Event()
    self.pause.set()

    for pin in step_pins:
        GPIO.setup(pin, GPIO.OUT)
        GPIO.output(pin, False)
```

step_pins: Lista de pines GPIO que se usan para controlar el motor.
seq: Secuencia de pasos que el motor debe seguir para moverse.
step_count: Número de pasos en la secuencia.
step_dir: Dirección de los pasos (1 para adelante, -1 para atrás).
step_counter: Contador de pasos actual.
wait_time: Tiempo de espera entre pasos (convertido a segundos).
running: Bandera para controlar si el motor está funcionando.
pause: Evento de hilo para pausar y reanudar el motor.

Método run

```
def run(self):
    while self.running:
        self.pause.wait()  # Pausar el hilo si se desactiva el evento

        for pin in range(0, 4):
            xpin = self.step_pins[pin]
            GPIO.output(xpin, self.seq[self.step_counter][pin])

        self.step_counter += self.step_dir

        # Si llegamos al final de la secuencia, empezar de nuevo
        if self.step_counter >= self.step_count:
            self.step_counter = 0
        if self.step_counter < 0:
            self.step_counter = self.step_count + self.step_dir

        time.sleep(self.wait_time)
```

while self.running: Mantener el bucle mientras el motor esté en funcionamiento.
self.pause.wait(): Espera activa si el evento pause está desactivado (pausa el hilo). Itera sobre los pines de control, configurando su salida de acuerdo con la secuencia seq. Incrementa o decrementa el contador de pasos (step_counter) según la dirección (step_dir). Reinicia el contador si llega al final de la secuencia. Espera el tiempo especificado (wait_time) entre pasos.

Método start

```
def start(self):
    if not self.running:
        self.running = True
        threading.Thread(target=self.run, daemon=True).start()
        print("Motor iniciado")
```

Comprueba si el motor no está ya funcionando. Establece la bandera running a True. Inicia un nuevo hilo para ejecutar el método run. Imprime un mensaje indicando que el motor ha comenzado a funcionar.

Método stop

```
def stop(self):
    self.running = False
    print("Motor detenido")
```

Establece la bandera running a False para detener el bucle del método run. Imprime un mensaje indicando que el motor se ha detenido.

Método pause_motor

```
def pause_motor(self):
    self.pause.clear()
    print("Motor pausado")
```

Desactiva el evento pause, lo que hace que el hilo espere (pausa el motor). Imprime un mensaje indicando que el motor está en pausa.

Método resume_motor

```
def resume_motor(self):
    self.pause.set()
    print("Motor reanudado")
```

Activa el evento pause, permitiendo que el hilo continúe (reanuda el motor). Imprime un mensaje indicando que el motor se ha reanudado.

Clase ServoMotor

```
def __init__(self, pin, frequency=50):
    self.pin = pin
    self.frequency = frequency
    self.pwm = GPIO.PWM(self.pin, self.frequency)
    self.pwm.start(0)
```

pin: El pin GPIO al que está conectado el servomotor.
frequency: La frecuencia de la señal PWM (default 50 Hz).
self.pwm = GPIO.PWM(self.pin, self.frequency): Inicializa el objeto PWM en el pin especificado con la frecuencia dada.
self.pwm.start(0): Inicia la señal PWM con un ciclo de trabajo inicial de 0.

Método move

```
def move(self, angle):
    duty_cycle = angle / 18.0 + 2.5
    self.pwm.ChangeDutyCycle(duty_cycle)
    time.sleep(0.5)
    self.pwm.ChangeDutyCycle(0)  # Detener el pulso para no mantener el
servo en movimiento
```

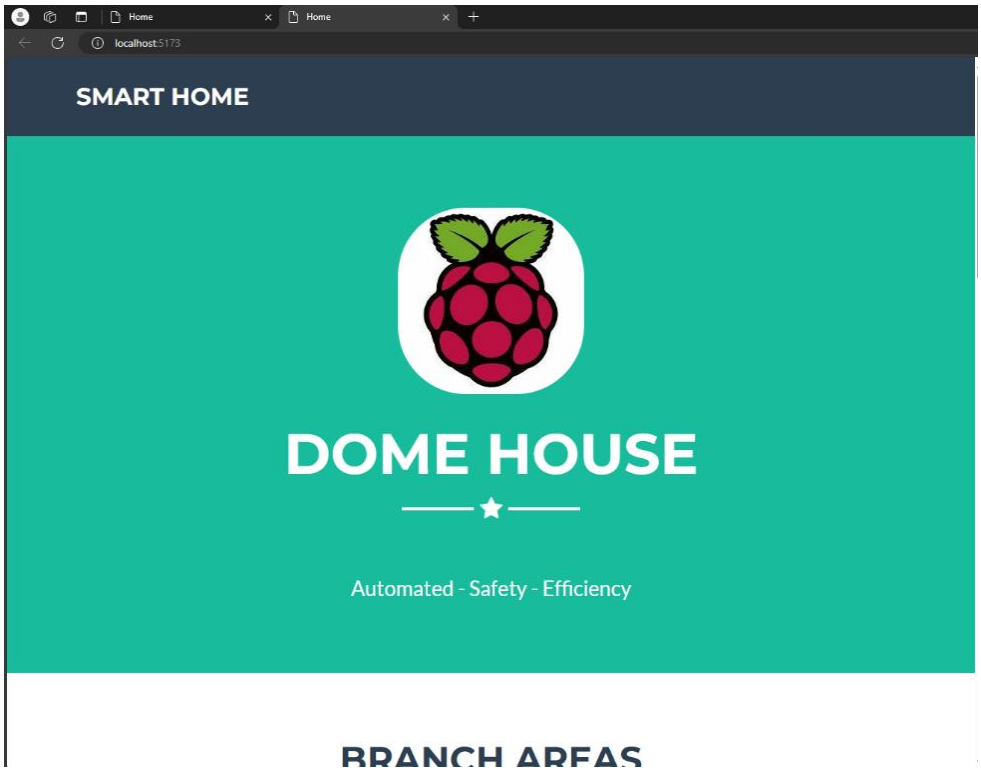
angle: El ángulo al que se quiere mover el servomotor.
duty_cycle = angle / 18.0 + 2.5: Calcula el ciclo de trabajo necesario para mover el servomotor al ángulo especificado. Esta fórmula convierte el ángulo en el ciclo de trabajo correspondiente.
self.pwm.ChangeDutyCycle(duty_cycle): Cambia el ciclo de trabajo de la señal PWM para mover el servomotor al ángulo deseado.
time.sleep(0.5): Espera 0.5 segundos para darle tiempo al servomotor para moverse.
self.pwm.ChangeDutyCycle(0): Detiene el pulso PWM para no mantener el servomotor en movimiento constante.

Método stop

```
def stop(self):
    self.pwm.ChangeDutyCycle(0)
```

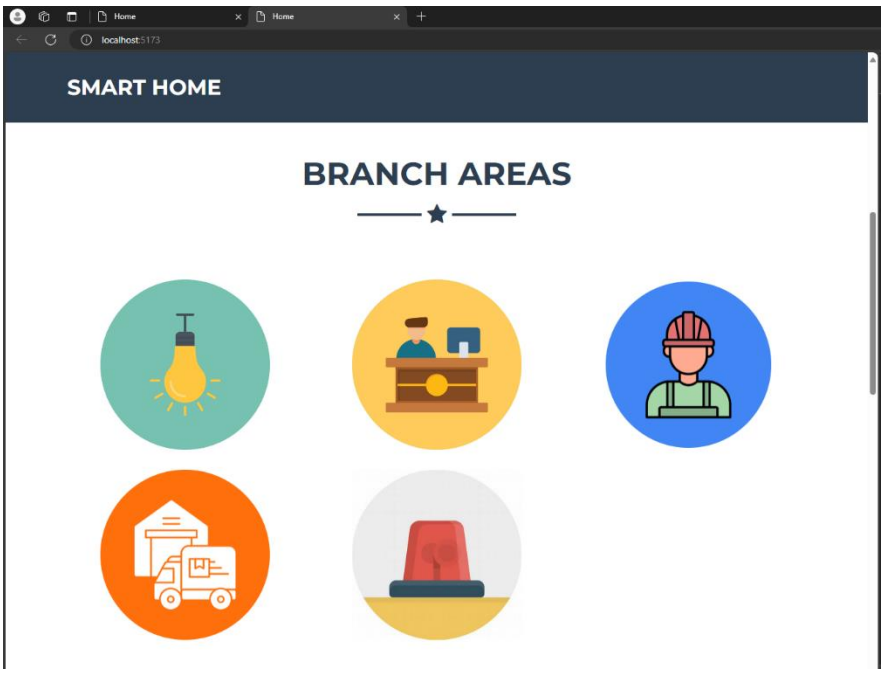
self.pwm.ChangeDutyCycle(0): Establece el ciclo de trabajo de la señal PWM a 0, deteniendo cualquier movimiento del servomotor.

Resultado del código fuente

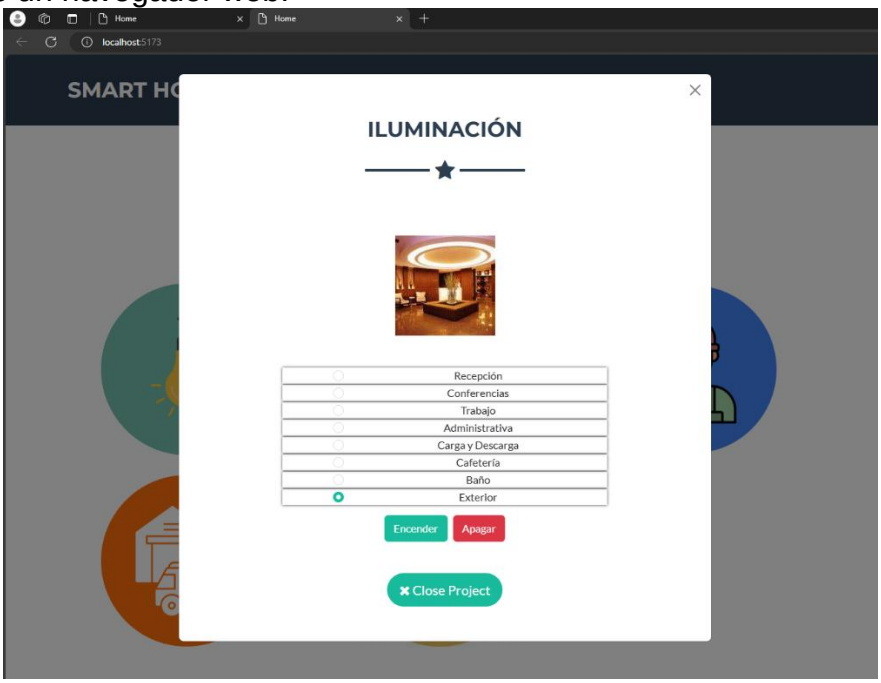


Interfaz haciendo énfasis con las respectivas opciones sobre las correspondientes áreas de distribución.

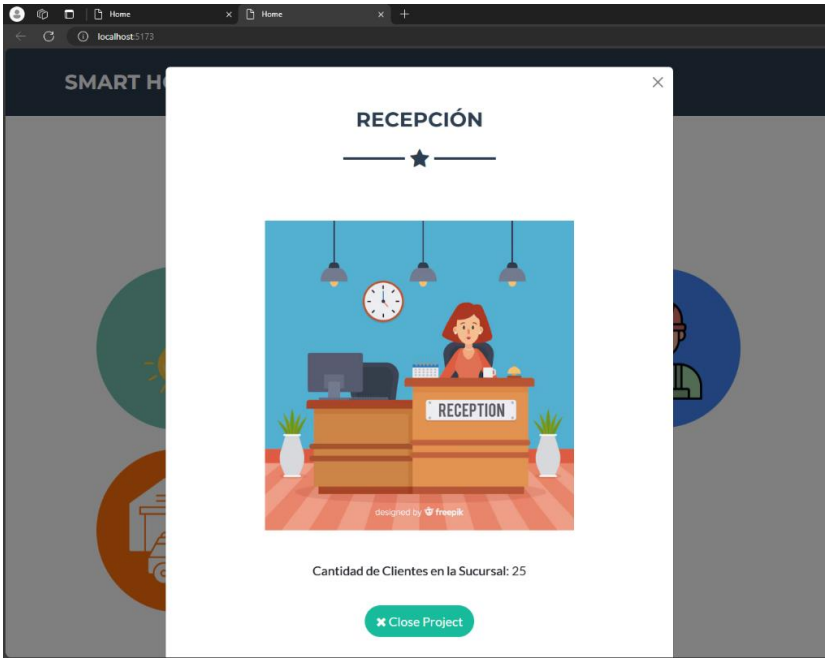
Se presentarán las diversas opciones de distribución, permitiendo así la selección adecuada para gestionar el seguimiento de la función.



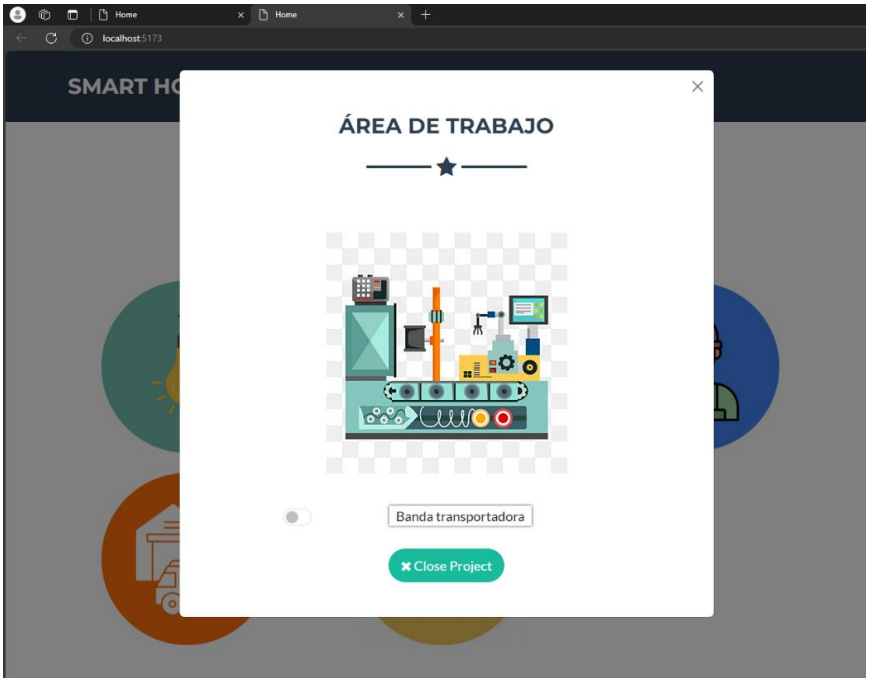
Proporcionará una interfaz visual donde se podrá observar el estado actual de todos los dispositivos conectados. Esto incluye la iluminación de diferentes áreas, el número de personas dentro de la sucursal, el estado de la banda transportadora y el portón de carga, y la activación de la alarma perimetral. Los datos se actualizarán en tiempo real, permitiendo una gestión eficiente y centralizada de los dispositivos desde un navegador web.



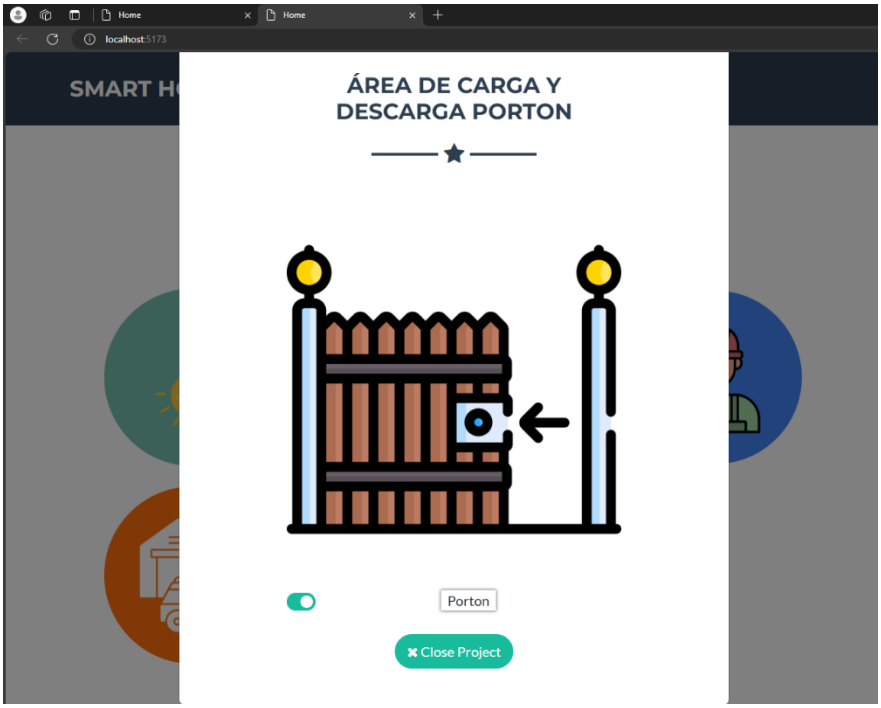
En la recepción se mostrará el estado de la iluminación (encendido o apagado) y se registrará el conteo de personas que ingresan a la sucursal, sin incluir las que salen. Estos datos se presentarán en la página web para un control preciso.



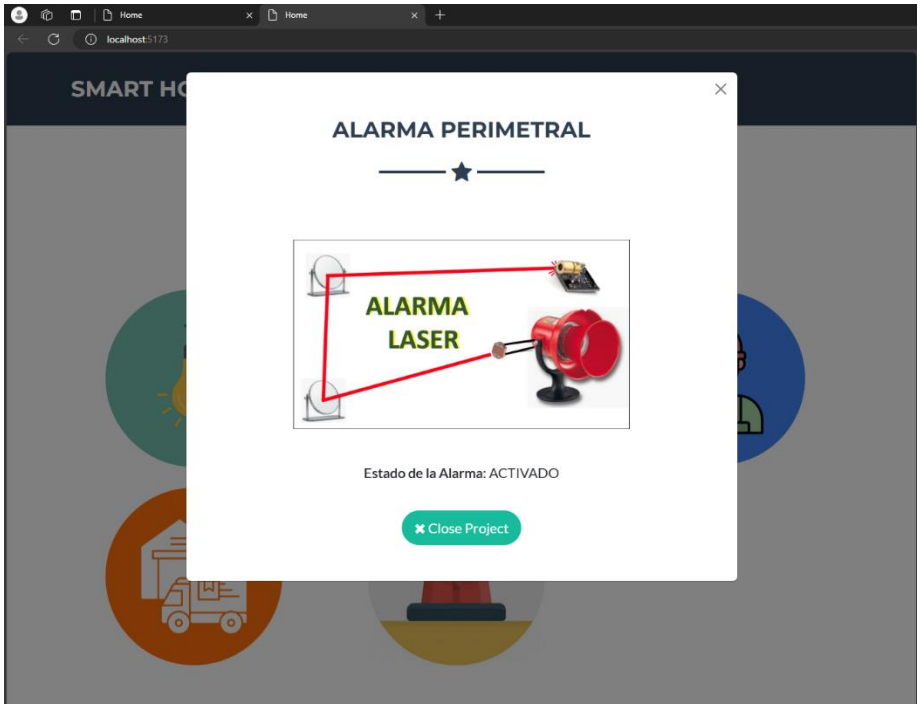
En el área de trabajo se mostrará el estado de la banda transportadora, indicando si está en movimiento o detenida. Esto se reflejará mediante dos LEDs: un LED verde para la banda en movimiento y un LED rojo para la banda detenida. Además, se podrá controlar la activación y desactivación de la banda transportadora desde la página web, permitiendo una gestión eficiente de su funcionamiento.



En el área de carga y descarga se mostrará el estado del portón (cerrado, abriendo, o abierto). Se podrá controlar la apertura y cierre del portón desde la página web, permitiendo la gestión remota de esta función.



La alarma perimetral mostrará su estado actual (apagada, detectando, activada) en la página web. Se activará automáticamente de noche mediante una fotorresistencia y emitirá un sonido de buzzer durante 10 segundos si detecta una interrupción en el láser.



Costos

Lista de materiales y costos asociados y presupuesto total del proyecto.

Componentes:			
Descripción	**Unidades**	**Precio Unidad (Q)**	**Total**
Raspberry	1	785	785
Cargador Raspberry	1	100	100
Resistencias	30	0.50	15
Display 7 seg Cat	1	5	5
Decodificador 74LS48	1	11	11
Sensor ultrasónico	1	27	27
LEDS	12	1	10
Módulo de láser KY-008	1	18	18
Foto resistencia analógicos Digitales	2	20	40
Espejos	3	5	15
Modelo de buzzer	1	15	15
Servomotor	2	31	62
Motor Stepper y Driver	1	45	45
Micro SD	1	80	80
Componente I2C	1	30	30
Pantalla LCD	1	32	32
Demux	2	6	6
Subtotal:			**1296**

Materiales para maqueta:			
Descripción	**Unidades**	**Precio Unidad (Q)**	**Total**
Cartón Chip	1	23	23
Silicon Caliente	2	2	4
Papel Cartulina	2	6	12
Subtotal:			**39**

Costo total del prototipo Q1,335.00