Control Digital
Prácticas de Automatización y Control con Tableros Didácticos

1st Elizabeth Corte, 2nd Dayana Jara
Facultad de Ingeniería
Ingeniería en Telecomunicaciones
Cuenca, Ecuador
elizabeth.corte@ucuenca.edu.ec,
dayana.jara93@ucuenca.edu.ec

***Abstract*—Este informe presenta la implementación de cuatro prácticas en un tablero didáctico, enfocadas en automatización, control digital, lógica secuencial y comunicación HMI, utilizando máquinas de estados finitos (FSM), controladores PID y programación estructurada en C++. Las prácticas incluyeron secuencias de LEDs, sistemas de semáforos y control de velocidad con retroalimentación PID, integrando interfaces HMI para visualización en tiempo real. Los resultados demostraron la eficacia de las FSM y los PID en sistemas embebidos, destacando la importancia de la programación estructurada y el manejo de interrupciones.**

I. Introducción

Los sistemas de automatización y control desempeñan un papel fundamental en la industria moderna, permitiendo la optimización de procesos, la mejora de la eficiencia y la reducción de errores humanos. En este contexto, el tablero de control (figura 1) descrito en la guía proporciona una plataforma didáctica para el aprendizaje y la experimentación con dispositivos de control lógico programable (PLC) e interfaces hombre-máquina (HMI). Este informe detalla las prácticas realizadas con el tablero, las cuales abarcan desde el manejo básico de salidas digitales hasta la implementación de sistemas avanzados de control, como el PID, para regular la velocidad de un motor DC.

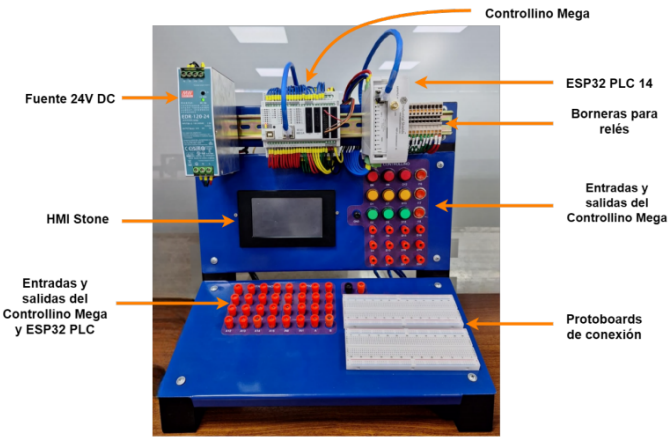

Fig. 1: Tablero de control

El tablero de control integra componentes industriales como el Controllino Mega, basado en tecnología Arduino, y el ESP32 PLC 14, que utiliza un microcontrolador ESP32. Ambos dispositivos cuentan con entradas y salidas analógicas y digitales, facilitando la implementación de sistemas de automatización. Además, la inclusión de una pantalla HMI Stone permite la interacción intuitiva con los procesos, lo que enriquece la experiencia de aprendizaje al combinar hardware y software en un entorno práctico.

Las prácticas descritas en este informe están diseñadas para desarrollar habilidades progresivas en el manejo de PLC y HMI. Se inician con ejercicios básicos, como el control de LEDs mediante salidas digitales, y avanzan hacia aplicaciones más complejas, como el diseño de interfaces gráficas y la implementación de algoritmos de control PID.

Este documento no solo sirve como registro de las actividades realizadas, sino también como una guía de referencia para futuros proyectos en el ámbito de la automatización y el control. A través de la metodología ”aprender haciendo”, se busca consolidar los conocimientos teóricos y técnicos necesarios para enfrentar desafíos reales en el campo de la ingeniería.

II. Objetivos

- Familiarizarse con los componentes del tablero de control, incluyendo su configuración, conexiones eléctricas y funcionalidades básicas.
- Implementar el control de salidas digitales y analógicas mediante programación en Arduino IDE, utilizando las librerías específicas de los PLC.
- Desarrollar interfaces gráficas en la HMI Stone para la visualización y control de procesos, aplicando el software *Stone Designer GUI*.
- Aplicar técnicas de temporización no bloqueante y máquinas de estados finitos (FSM) en el control secuencial de actuadores, como LEDs y relés.
- Adquirir y procesar datos en tiempo real, como la velocidad de un motor DC, mediante el uso de interrupciones y comunicación serial con la HMI.
- Diseñar e implementar un controlador PID para regular la velocidad del motor, ajustando sus parámetros para

garantizar estabilidad y un error mínimo en estado estacionario.
- Evaluar el desempeño del sistema de control, analizando su respuesta ante entradas escalón y perturbaciones externas.

III. Desarrollo

Práctica 1: Manejo de Salidas Digitales con Controllino Mega

Usar el Controllino Mega para controlar una matriz de 9 LEDs del tablero, dispuestos en una cuadrícula de 3x3. Los LEDs deben encenderse uno por uno siguiendo un patrón en espiral, en el siguiente orden:

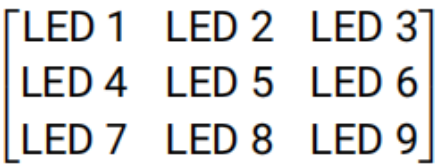

Fig. 2: Matriz de leds

- Orden de encendido: $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 8 \rightarrow 7 \rightarrow 4 \rightarrow 5$ y se repite.
- Asigna un pin digital a cada LED usando variables predefinidas de la librería `Controllino.h`.
- Cada LED debe encenderse durante 500 ms y luego apagarse, justo en ese instante se enciende el siguiente.
- Al finalizar, la secuencia se debe reiniciar.

Requisitos:

- Usar punteros.
- Usar retardos no bloqueantes.

A continuación se presenta la explicación detallada del programa implementado para controlar una secuencia de **9 LEDs** mediante un *Controllino*, simulando un patrón de encendido en forma de espiral.

Componentes del Código

- **Librería:** Se incluye `Controllino.h` para utilizar nombres de pines específicos.
- **Variables de tiempo:**
  - `t_previo`: instante del último cambio de LED.
  - `intervalo`: periodo entre cambios (500 ms).
- **Arreglos:**
  - `leds[9]`: pines de conexión física de los LEDs.
  - `secuencia[9]`: orden de encendido para el patrón en espiral.
- **Punteros:** `ptrLeds` y `ptrSecuencia` permiten recorrer los arreglos mediante aritmética de punteros.

Inicialización

En la función `setup()`, cada pin se configura como salida digital y se asegura que los LEDs estén apagados:

```
for (int i = 0; i < 9; i++) {
  pinMode(*(ptrLeds + i), OUTPUT);
  digitalWrite(*(ptrLeds + i), LOW);
}
Serial.begin(9600);
```

Funcionamiento Principal

En `loop()`, el sistema ejecuta continuamente los siguientes pasos:

1) Se obtiene el tiempo actual con `millis()`.
2) Se verifica si ha transcurrido el `intervalo` desde el último cambio.
3) Si se cumple la condición:
   - Se apagan todos los LEDs.
   - Se enciende únicamente el LED correspondiente al índice de la secuencia.
   - Se incrementa el índice de forma circular:

$$\text{indice} = (\text{indice} + 1)\%9.$$

El fragmento clave es el siguiente:

```
if (t_actual - t_previo >= intervalo) {
  t_previo = t_actual;

  for (int i = 0; i < 9; i++) {
    digitalWrite(*(ptrLeds + i), LOW);
  }

  int pinActual = *(ptrLeds + *(ptrSecuencia +
    indice));
  digitalWrite(pinActual, HIGH);

  indice = (indice + 1) % 9;
}
```

Este programa separa claramente la configuración física (`leds[]`) de la lógica de la secuencia (`secuencia[]`), permitiendo modificar la trayectoria luminosa sin alterar la conexión de hardware. El uso de punteros ilustra una técnica de acceso eficiente a arreglos en C++ para sistemas embebidos.

Práctica 2: Control básico de salidas digitales y aplicación avanzada con FSM

Parte A

Utilizar los tres botones del tablero de pruebas para controlar el patrón de encendido de los LED ubicados en forma de matriz 3x3. Tanto los botones como los LED ya se encuentran conectados directamente al Controllino Mega.

- *Botón 1: Encendido en espiral normal.*
- *Botón 2: Encendido en espiral inverso.*
- *Botón 3: Reinicia y apaga todos los LEDs.*

A continuación se presenta la explicación detallada del programa implementado para el control de una secuencia de 9 LEDs mediante una máquina de estados y el uso de tres botones físicos, permitiendo seleccionar entre modo normal, inverso o apagado.

Definición de Estados

Se define una enumeración `enum Estado` con tres estados:

- `APAGADO`: todos los LEDs están apagados.
- `SECUENCIA_NORMAL`: los LEDs se encienden de izquierda a derecha.
- `SECUENCIA_INVERSA`: los LEDs se encienden de derecha a izquierda.

El sistema inicia en el estado `APAGADO`.

Configuración de Botones

Los botones están conectados a entradas digitales del Controllino y permiten cambiar el estado de la máquina de estados:

- Botón 1: inicia la secuencia normal.
- Botón 2: inicia la secuencia inversa.
- Botón 3: apaga todos los LEDs y reinicia la secuencia.

Arreglo de LEDs

El arreglo `leds[9]` contiene los pines de salida asociados a cada LED. El orden dentro del arreglo define el patrón visual de encendido.

```
int leds[9] = {
      CONTROLLINO_D0,
      CONTROLLINO_D6,
      CONTROLLINO_D12,
      CONTROLLINO_D13,
      CONTROLLINO_D14,
      CONTROLLINO_D8,
      CONTROLLINO_D2,
      CONTROLLINO_D1,
      CONTROLLINO_D7};
```

Variables de Temporización

Se utiliza un control de tiempo **no bloqueante** con la función `millis()` para determinar cuándo cambiar de LED, evitando pausar la ejecución del programa.

- `t_previo`: almacena el instante del último cambio.
- `intervalo`: define el tiempo entre cambios (500 ms).

Inicialización (`setup()`)

- Se configura cada botón como entrada.
- Se configura cada pin del arreglo de LEDs como salida digital.
- Se apagan todos los LEDs al inicio.

Lectura de Botones y Cambio de Estado

Dentro de `loop()`, se leen continuamente los botones para cambiar el estado de la máquina de estados:

- Botón 1 → `SECUENCIA_NORMAL`
- Botón 2 → `SECUENCIA_INVERSA`
- Botón 3 → `APAGADO` y apaga todos los LEDs de inmediato.

Máquina de Estados

El control principal se implementa con una estructura `switch`, que ejecuta acciones específicas según el estado actual:

- **APAGADO:** se asegura de mantener todos los LEDs apagados.
- **SECUENCIA NORMAL:** enciende los LEDs uno por uno en orden creciente.
- **SECUENCIA INVERSA:** enciende los LEDs en orden decreciente.

Cada transición entre LEDs ocurre solo si ha transcurrido el intervalo especificado, manteniendo la ejecución no bloqueante.

La figura 3 muestra el diagrama de bloques de la máquina de estados.

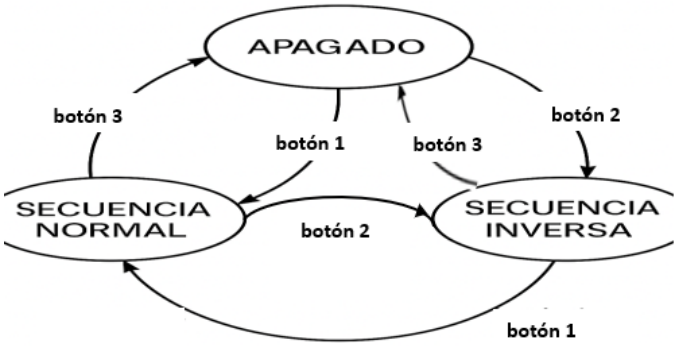

Fig. 3: Diagrama de bloques - Parte A

Funciones de Control

- `ejecutarSecuenciaNormal()`: apaga todos los LEDs, enciende el LED actual y avanza el índice de forma circular.
- `ejecutarSecuenciaInversa()`: apaga todos los LEDs, enciende el LED actual y retrocede el índice de forma circular.
- `apagarTodosLosLeds()`: apaga todos los LEDs del arreglo.

Parte B

Diseñar un sistema que controle dos semáforos (Semáforo A y Semáforo B) ubicados en una intersección perpendicular (figura 4), usando el enfoque de máquina de estados finita (FSM). El sistema debe simular el comportamiento simple de los semáforos, de manera que nunca haya luz verde simultánea para ambos sentidos, y se respeten los tiempos estándar de duración de cada luz.

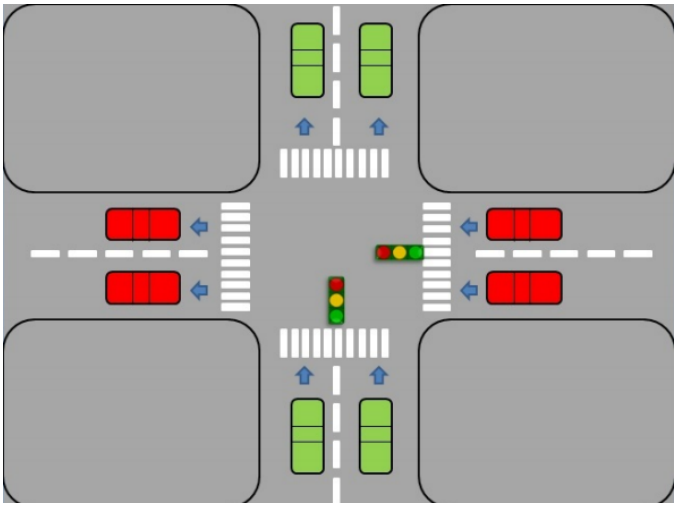

Fig. 4: Circulación vehicular simple

Requisitos:

- *Usar retardos no bloqueantes.*
- *Usar estructuras.*
- *Usar datos tipo enum.*

A continuación se describen las partes fundamentales del programa que simula el comportamiento de dos semáforos coordinados utilizando una máquina de estados en un *Controllino*.

Definición de estados

Se define un tipo `enum` denominado `EstadoSemaforo` con los siguientes estados:

- `VERDE_A`: Semáforo A en verde, B en rojo.
- `AMARILLO_A`: Semáforo A en amarillo, B en rojo.
- `VERDE_B`: Semáforo B en verde, A en rojo.
- `AMARILLO_B`: Semáforo B en amarillo, A en rojo.

Estructura para semáforo

Se declara una estructura `Semaforo` que agrupa los tres pines (verde, amarillo y rojo) de cada semáforo. Esta agrupación permite acceder y controlar fácilmente cada luz.

```
struct Semaforo {
  int verde;
  int amarillo;
  int rojo;
};
```

Variables de control

- `estado`: guarda el estado actual del sistema.
- `t_estado`: almacena la marca de tiempo del último cambio de estado.
- `duracion`: define la duración del estado activo.
- Constantes: `TIEMPO_VERDE` y `TIEMPO_AMARILLO` determinan la duración de cada color en milisegundos.

Inicialización (`setup`)

- Se configuran los pines de los semáforos A y B como salidas digitales.
- Se establece el estado inicial `VERDE_A` y se apagan todas las luces para iniciar de forma segura.

Lógica principal (`loop`)

- La función `millis()` se utiliza para controlar los tiempos de forma no bloqueante.
- En cada iteración se evalúa si el tiempo del estado activo se ha cumplido para proceder al cambio de estado.
- Se actualizan las luces en función del estado actual, asegurando la coordinación entre ambos semáforos.

Máquina de estados

El control de la lógica de los semáforos se realiza mediante una máquina de estados finitos implementada con un `switch-case`. Su funcionamiento es el siguiente:

- Cada estado tiene una duración predefinida.
- Cuando la duración expira, se realiza la transición al siguiente estado siguiendo la secuencia:

$$\texttt{VERDE\_A} \rightarrow \texttt{AMARILLO\_A} \rightarrow \texttt{VERDE\_B} \rightarrow \texttt{AMARILLO\_B} \rightarrow \texttt{VERDE\_A}.$$

- En cada transición se apagan todas las luces anteriores, se actualiza el estado y se encienden las luces correspondientes.

La figura 5 muestra el diagrama de bloques de la máquina de estados.

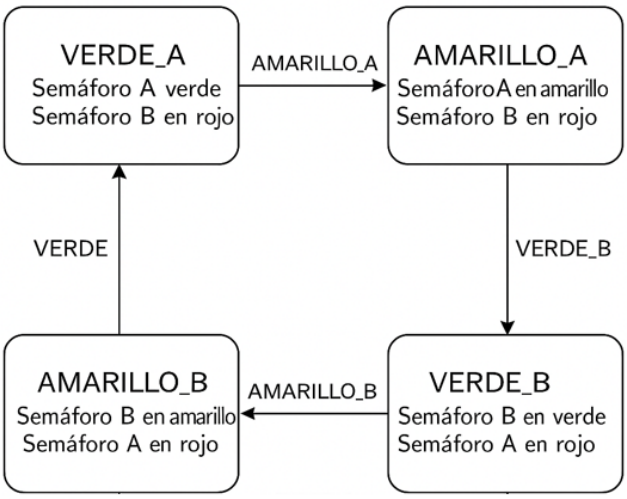

Fig. 5: Diagrama de bloques - ParteB

Función `establecerEstado`

- Cambia el estado actual del sistema.
- Apaga todas las luces de ambos semáforos.
- Actualiza la marca de tiempo `t_estado` para reiniciar el conteo de duración del nuevo estado.

Este diseño asegura un flujo de tráfico alternado entre dos vías de forma automática y sincronizada.

Práctica 3: Diseño de interfaz gráfica para el control de salidas en Controllino

Descripción de la actividad:

- *Agregar un segundo widget tipo SpinBox a la interfaz gráfica. Este segundo SpinBox permitirá controlar el duty cycle (porcentaje de ciclo de trabajo PWM) del segundo LED conectado al tablero.*
- *Configurar dos botones físicos en el tablero:*
  - *El primer botón físico controlará el encendido/apagado del primer LED.*
  - *El segundo botón físico controlará el encendido/apagado del segundo LED.*
- *Cada SpinBox deberá estar asociado de manera independiente a su respectivo LED, de forma que, al mover el SpinBox, se ajuste la intensidad del brillo del LED correspondiente, sin afectar al otro LED.*
- *La acción de los botones físicos debe ser independiente del valor del SpinBox:*
  - *El botón únicamente habilitará o deshabilitará el encendido del LED, pero el brillo será determinado por el valor actual del SpinBox asociado.*
- *El sistema debe garantizar que, si un LED está apagado por el botón físico, no se active aunque se modifique el SpinBox correspondiente (hasta que el botón vuelva a presionarse).*

Diseño de la interfaz

La interfaz se basa en el diseño proporcionado en la guía [1], con las siguientes adiciones:

1) Añadir un nuevo label

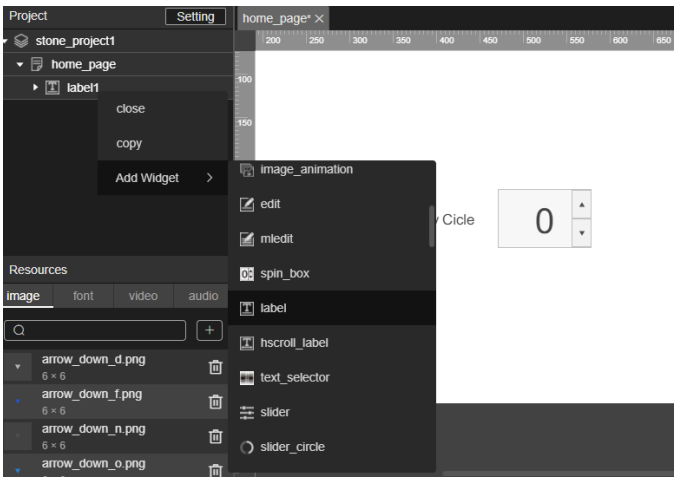

Fig. 6: Añadir label

2) Configurar label2

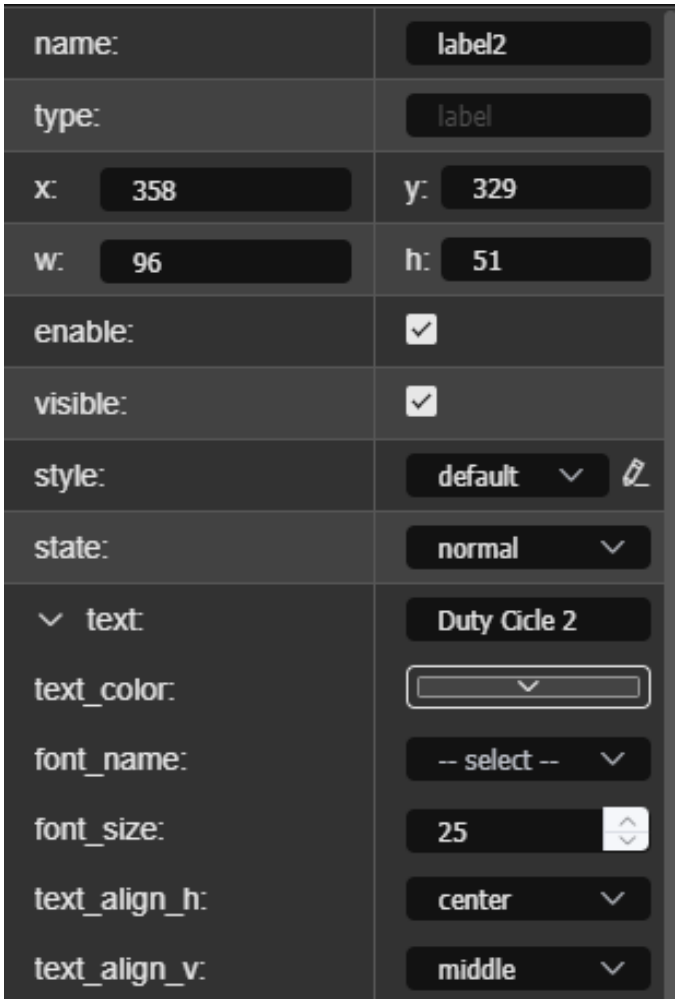

Fig. 7: Configuración del label2

3) Añadir un nuevo *spin_box*

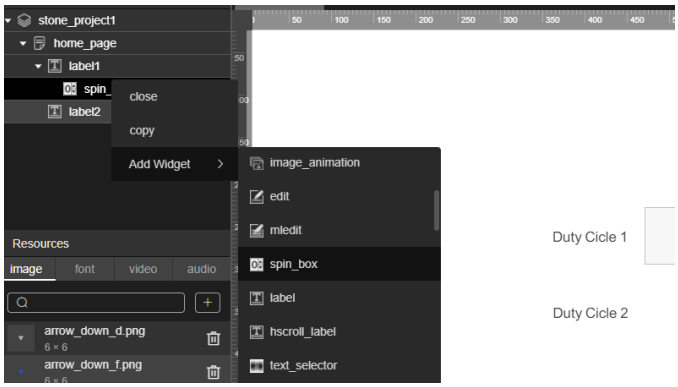

Fig. 8: Añadir *spin_box*

4) Configuración del *spin_box*2

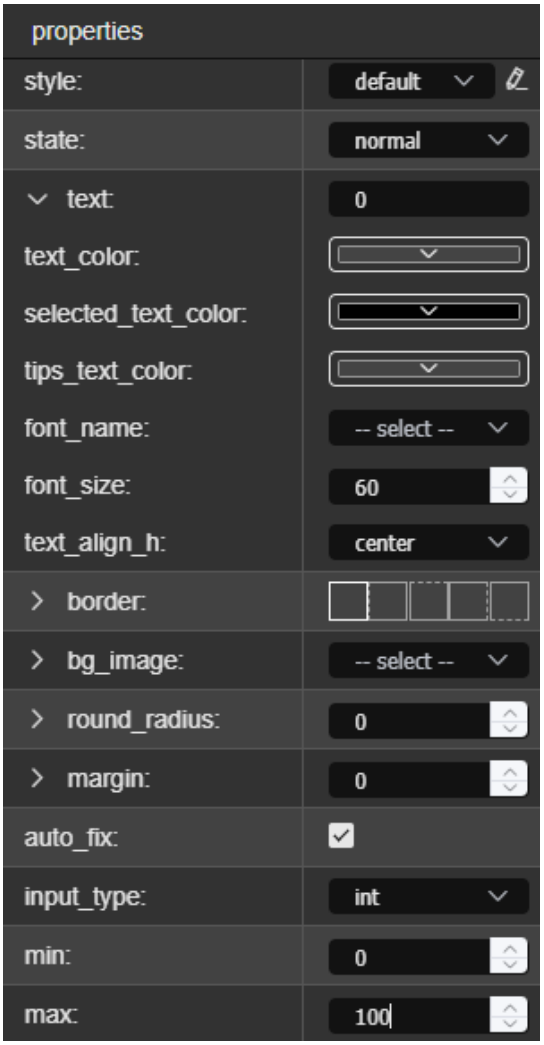

Fig. 9: Configuracion de $spin\_box2$

A continuación se describen las secciones más relevantes del código, junto con fragmentos representativos que ilustran la estructura y la lógica principal.

Inclusión de librerías

Se importan las librerías necesarias para manejar el Controllino y la comunicación con la HMI:

```
#include <Controllino.h>
#include "Stone_HMI_Define.h"
#include "Procesar_HMI.h"
```

Definición de pines y variables clave

Se declaran los pines de los LEDs y botones, así como las variables de control:

```
const int led1 = CONTROLLINO_D0;
const int led2 = CONTROLLINO_D1;

const int button1 = CONTROLLINO_I16;
const int button2 = CONTROLLINO_I17;

bool led1_enabled = false;
bool led2_enabled = false;
int pwmValue1 = 0, pwmValue2 = 0;
float dutyCyclePercent1 = 0,
dutyCyclePercent2 = 0;
```

Configuración inicial

En la función `setup`, se inicializan los pines y la HMI:

```
void setup() {
  pinMode(led1, OUTPUT);
  pinMode(led2, OUTPUT);
  pinMode(button1, INPUT);
  pinMode(button2, INPUT);

  HMI_init();
  Stone_HMI_Set_Value("spin_box",
  "spin_box1", NULL, 0);
  Stone_HMI_Set_Value("spin_box",
  "spin_box2", NULL, 0);
}
```

Lógica principal

Dentro de la función `loop`, se leen los valores de los SpinBoxes de la HMI, que representan el porcentaje de ciclo útil para cada LED, y se convierten a valores PWM de 0 a 255.

```
// Lectura de la HMI
dutyCyclePercent1 = HMI_get_value("spin_box",
    "spin_box1");
dutyCyclePercent2 = HMI_get_value("spin_box",
    "spin_box2");

// Mapeo a rango PWM
pwmValue1 = map(dutyCyclePercent1, 0, 100, 0,
    255);
pwmValue2 = map(dutyCyclePercent2, 0, 100, 0,
    255);
```

Función: permitir que el usuario ajuste en tiempo real la intensidad de cada LED mediante controles gráficos.

Lectura y control de botones físicos

Se implementa la detección de flanco de subida para cada botón, alternando el estado ON/OFF de cada LED.

```
// Detectar flanco de subida
if (boton1 pasa de LOW a HIGH)
  led1_enabled = !led1_enabled;

if (boton2 pasa de LOW a HIGH)
  led2_enabled = !led2_enabled;
```

Función: permitir el encendido o apagado manual de cada LED independientemente del valor de ciclo útil.

Aplicación de PWM y control de salida

Según el estado de cada LED, se aplica el valor PWM calculado o se apaga el LED.

```
// Control de brillo
if (led1_enabled) analogWrite(led1,
    pwmValue1);
else analogWrite(led1, 0);

if (led2_enabled) analogWrite(led2,
    pwmValue2);
else analogWrite(led2, 0);
```

Función: combinar la activación manual (botón) con el ajuste de brillo dinámico (HMI).

Este diseño permite ajustar la intensidad luminosa y encender o apagar cada LED de forma independiente, optimizando la interacción hombre-máquina.

Práctica 4: Diseñar e implementar un controlador PID que regule automáticamente la velocidad del motor DC del EPC

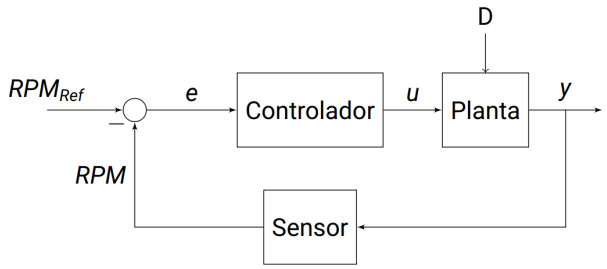

Fig. 10: Esquema de lazo de control con controlador y planta

Diseño de la interfaz

Para implementar esta práctica se realizaron cambios en los valores del slider, cuya función es establecer un valor RPM de referencia para el motor(en este caso de 0 a 5000RPM). También se agregaron labels para mostrar los parámetros utilizados en el controlador: K_p, K_i y K_d; finalmente una gráfica más adicionando un widget `line_series` que mostrará la señal de control en color verde. Dado que la señal de control está acotada entre 0 y 255 se debe cambiar los valores de `y_axis3` donde muestre los límites para la señal de control, en este caso se le asignó el color verde, como se observa en la figura 11.

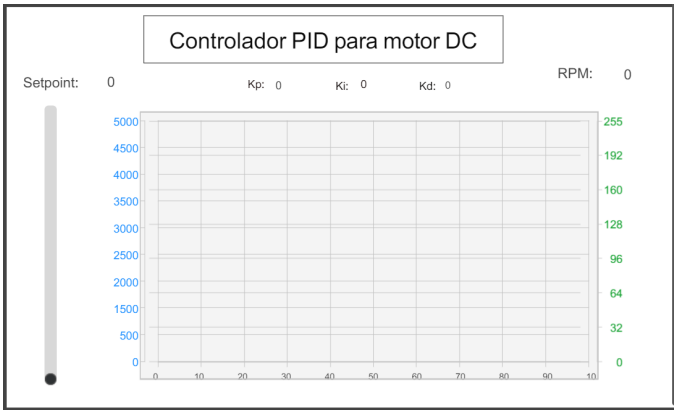

Fig. 11: Interfaz diseñada para la implementación de PID

Función de transferencia del motor y parámetros del controlador

Para determinar la función de transferencia, primero se tomaron los datos del motor mediante Labview con la respuesta a un impulso, con el archivo generado se implementó un script en Matlab donde se obtuvo la siguiente función de transferencia:

$$H = \frac{0.773}{4.124s + 1}$$

Mediante el mismo script y con la función `pidtune` se obtienen los siguientes parámetros para el controlador:

$$K_p = 1.852$$

$$K_i = 0.814$$

$$K_d = 0$$

Con estos parámetros se obtiene la siguiente respuesta:

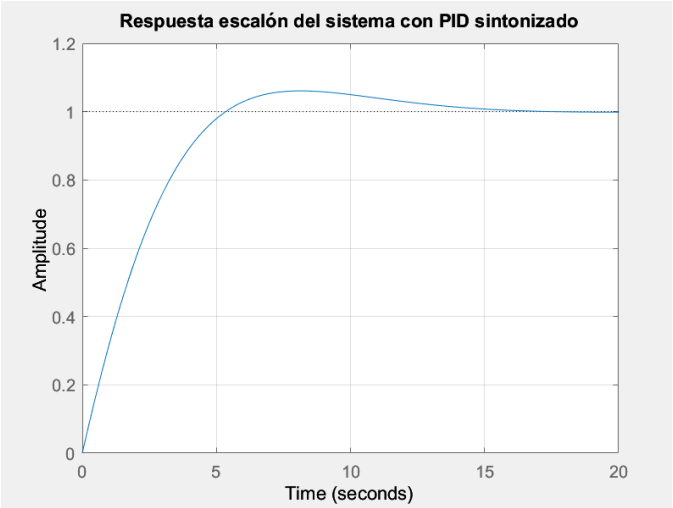

Fig. 12: Respuesta al escalón obtenida en Matlab

Código implementado

En el código se implementa un controlador PID en forma incremental discreta, que se aplica dentro del manejo de interrupciones del temporizador:

```
float delta_u =
Kp * (e[0] - e[1]) + // Proporcional
    incremental
(Kp * Ts / Ti) * e[0]; // Integral
    incremental

u = u + delta_u; // Acumular salida
u_k = constrain(u, 0, 255); // Saturar
    salida
analogWrite(pin_motor, int(u_k)); //
    Aplicar senal al motor
```

Esta implementación corresponde a la ecuación de diferencia del PID:

$$\Delta u[k] = K_p(e[k] - e[k-1]) + \frac{K_p \cdot T_s}{T_i} \cdot e[k]$$

$$u[k] = u[k-1] + \Delta u[k]$$

Donde:

- $e[k]$ es el error actual: diferencia entre la referencia y la velocidad actual en RPM.
- T_s es el periodo de muestreo.
- $T_i = \frac{K_p}{K_i}$ es el tiempo integral.
- $u[k]$ es la señal de control aplicada al motor (limitada entre 0 y 255).

El periodo de muestreo es un parámetro crítico en sistemas de control digital, ya que afecta directamente la estabilidad, la precisión y la velocidad de respuesta del sistema.

En este proyecto se seleccionó un periodo de muestreo de:

$$T_s = 0.05\,\text{s} \quad \text{(equivalente a } f_s = 20\,\text{Hz)}$$

Razones principales para esta elección:

1) **Frecuencia del sistema físico:** El motor a controlar tiene una dinámica relativamente lenta (su constante de tiempo es mayor a 100 ms), por lo que un muestreo cada 50 ms permite capturar adecuadamente los cambios sin perder información relevante.
2) **Estabilidad del controlador PID:** Si se reduce demasiado el periodo de muestreo (es decir, se aumenta f_s en exceso), el controlador PID puede volverse más sensible al ruido del encoder. Con $T_s = 0.05\,s$, se logra un buen compromiso entre rapidez y estabilidad.
3) **Carga computacional:** El microcontrolador tiene recursos limitados. Elegir $f_s = 20\,Hz$ asegura que la ISR del temporizador y otras tareas (como la comunicación con la HMI) puedan ejecutarse sin interferencias.
4) **Sincronización con la interfaz HMI:** La actualización de la interfaz gráfica se realiza cada 100 ms, por lo que tener un muestreo dos veces más rápido permite mostrar al usuario datos actualizados de manera fluida y representativa.

Se configura el `Timer1` en modo CTC (Clear Timer on Compare Match) para generar una interrupción periódica:

```
TCCR1B |= (1 << WGM12); // Modo CTC
TCCR1B |= (1 << CS12); // Prescaler de 256
OCR1A = 62500 / fs;
// Valor de comparacin para
interrupcin cada Ts
TIMSK1 |= (1 << OCIE1A);
// Habilita interrupcin
```

Cálculo del valor del comparador OCR1A:

$$\text{Frecuencia de reloj del timer} = \frac{16\ \text{MHz}}{256} = 62500\ \text{Hz}$$

Para un periodo de muestreo $T_s = 0{,}05$ s:

$$OCR1A = 62500 \times 0{,}05 = 3125$$

Por tanto, el temporizador interrumpe cada 3125 ciclos para ejecutar el control.

El encoder óptico genera pulsos que se capturan mediante una interrupción externa:

```
void contarPulso() {
  conteo_pulsos++; // Incrementa en cada
      flanco de bajada
}
```

En cada periodo de muestreo, se calcula la velocidad (RPM) usando:

$$\text{RPM} = \frac{\text{conteo\_pulsos} \times 60 \times f_s}{\text{PULSOS\_POR\_REV}}$$

- $f_s = 20$ Hz es la frecuencia de muestreo
- $PULSOS\_POR\_REV = 36$ es la resolución del encoder (pulsos por revolución)
- Se multiplica por 60 para convertir de revoluciones por segundo a revoluciones por minuto (RPM)

IV. Resultados

Práctica 1

En la figura 13 se muestra como se enciende el segundo led de la secuencia que se estableció para esta practica.

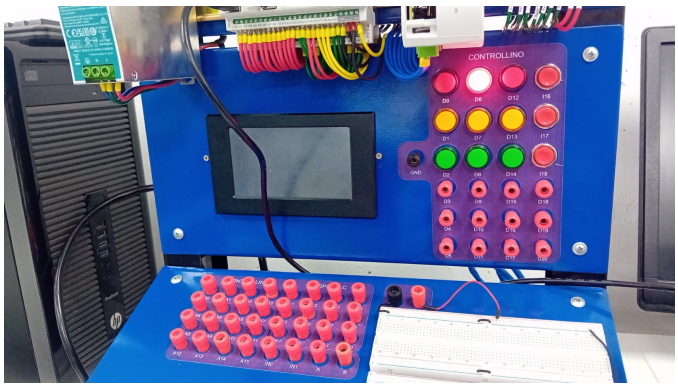

Fig. 13: Resultado

En el siguiente enlace https://github.com/elizabeth-123/Control-Digital/tree/main/Practica1 se encuentra un video demostrativo de la secuencia de leds y el código desarrollado para esta práctica.

Práctica 2

En la parte A de este ejercicio, los botones permiten al usuario seleccionar la secuencia de LED que se reproducirá. La figura 14 muestra que la secuencia cumple el objetivo de iluminar los LED uno a uno según la secuencia seleccionada.

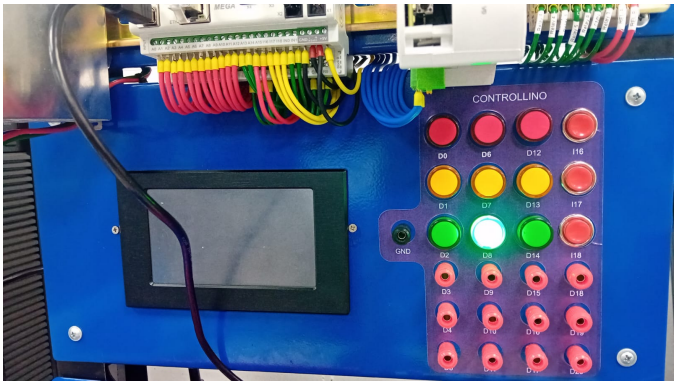

Fig. 14: Secuencia normal

En la parte B del ejercicio, se observa que el sistema controla eficazmente ambos semáforos, impidiendo luces verdes simultáneas. En la figura 15 se muestra como trabajan ambos semáforos.

Fig. 15: Funcionamiento de los dos semáforos

Igualmente para poder apreciar mejor el resultado de ambas partes de la practica se comparte el siguiente Para apreciar mejor los resultados de ambas partes de la práctica, se comparten los videos y códigos correspondientes en el siguiente enlace: https://github.com/elizabeth-123/Control-Digital/tree/main/Practica2.

Práctica 3

En la figura 16 se muestra la interfaz en el HMI que se diseño para esta practica. La interfaz tiene dos *spin_box*, los cuales permiten controlar el brillo de los leds.

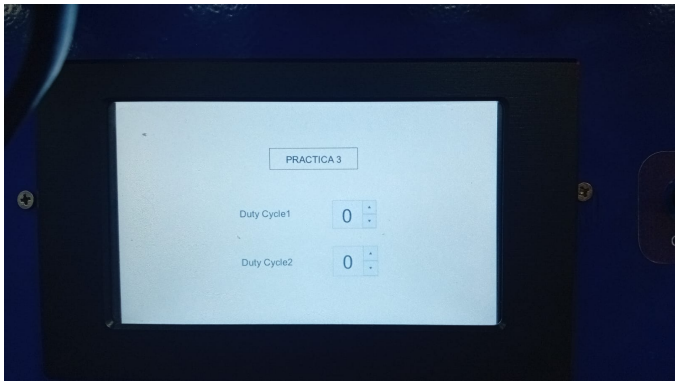

Fig. 16: Interfaz 1

En la figura 17, se observa el primer led con un nivel de brillo de 50 mientras que el segundo led tiene un nivel de brillo de 90.

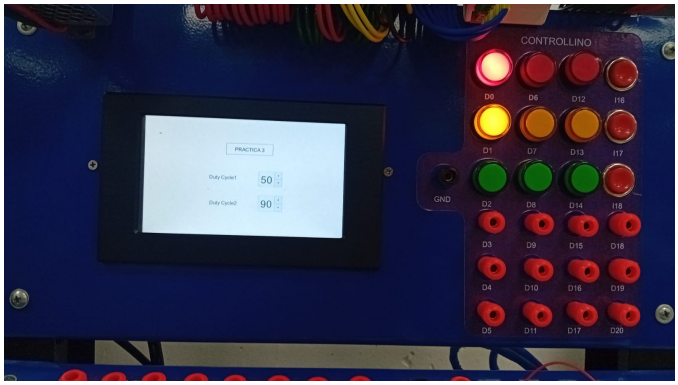

Fig. 17: Led 1 con brillo de 50% y led 2 con brillo de 90%

La figura 18, presenta el primer led con un nivel de brillo de 90 mientras que el segundo led tiene un nivel de brillo de 10.

Fig. 18: Led 1 con brillo de 90% y led 2 con brillo de 10%

El video con el resultado de la práctica y el codigo implementado están disponibles en: https://github.com/elizabeth-123/Control-Digital/tree/main/Practica3.

Práctica 4

En la figura 19, se muestra la conexión del tablero con la planta del motor.

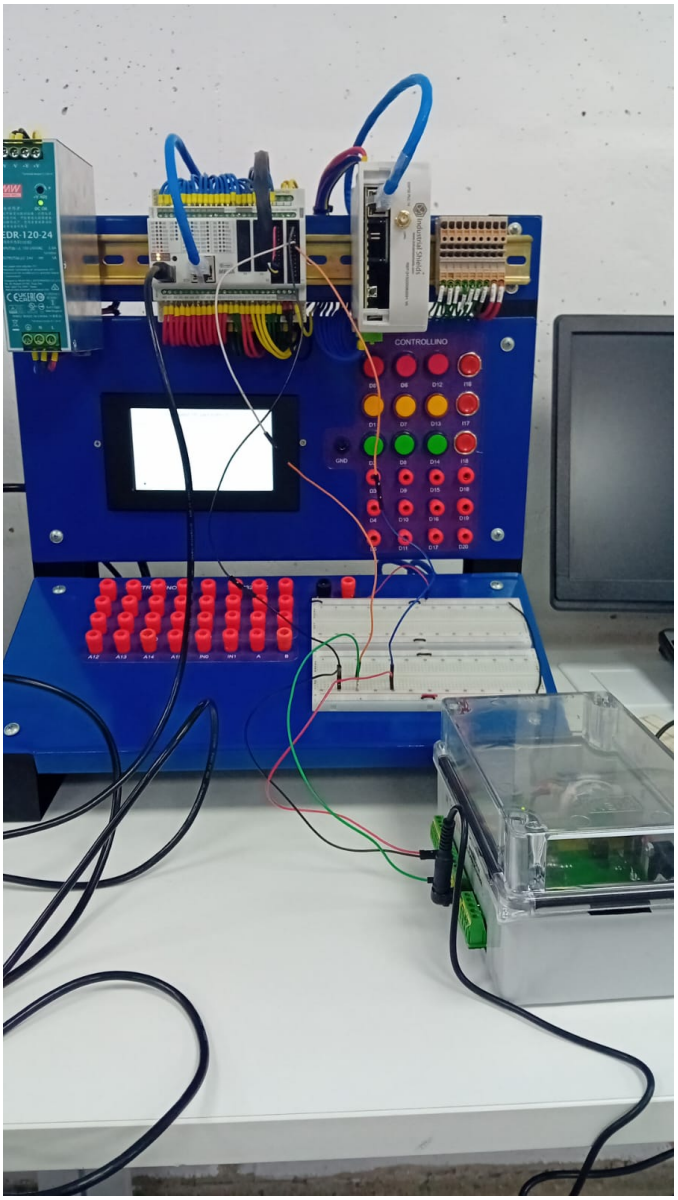

Fig. 19: Conexión del tablero con la planta

La figura 20 muestra la interfaz en el HMI que se diseño para esta practica.

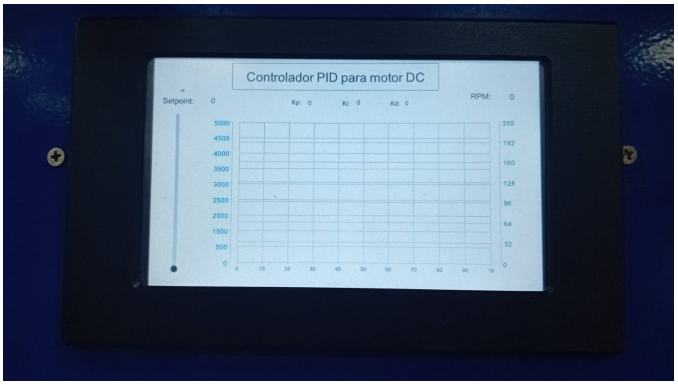

Fig. 20: Interfaz 2

La figura 21 muestra la gráfica de respuesta temporal donde se observan tres curvas principales: la línea azul representa el valor de referencia o *setpoint* de velocidad, la línea roja muestra la velocidad real del motor siguiendo dicha referencia, y la línea verde indica la señal de control. Se aprecia que, tras un aumento repentino del *setpoint*, la velocidad del motor responde con un sobreimpulso inicial, seguido de oscilaciones que se atenúan hasta estabilizarse cerca del valor deseado, mientras la señal de control presenta picos y fluctuaciones que reflejan la acción proporcional del controlador, ya que los parámetros integral y derivativo están en cero. Se observó que al variar la constante de proporcionalidad K_p, la respuesta del sistema mejora, disminuyendo las oscilaciones.

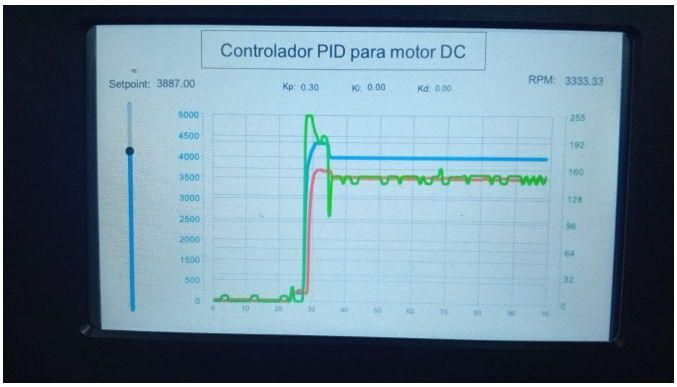

Fig. 21: Respuesta del PID

En la figura 22 se observa la gráfica de respuesta del mismo sistema PID para motor DC mostrado anteriormente, pero esta vez frente a una perturbación: la línea azul marca el *setpoint* constante, la línea roja muestra cómo la velocidad real del motor sigue estable tras alcanzar rápidamente el valor deseado, y la línea verde refleja la señal de control ajustándose continuamente; alrededor del segundo 85 se introduce una perturbación que provoca una caída momentánea en la velocidad, pero el controlador responde generando un pico en la señal de control, logrando recuperar y mantener nuevamente la velocidad cerca del *setpoint*.

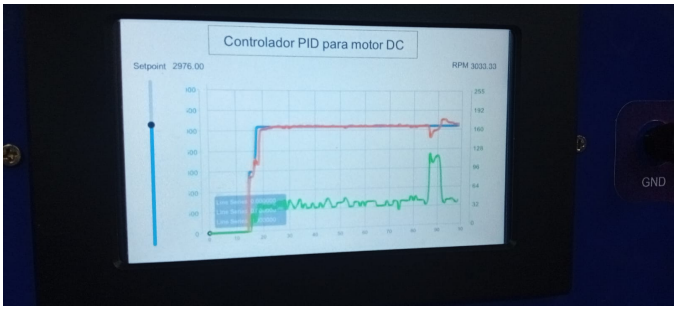

Fig. 22: Respuesta a una perturbación

Los archivos de esta práctica, junto con videos demostrativos de su funcionamiento, se encuentran en: https://github.com/elizabeth-123/Control-Digital/tree/main/Practica4.

V. Conclusiones

A lo largo del desarrollo de las cuatro prácticas implementadas en los tableros **Controllino**, se logró integrar conceptos

fundamentales de automatización, control digital, lógica secuencial y comunicación HMI en un entorno de programación embebida. Cada una de las prácticas abordó distintos desafíos, pero compartieron el objetivo común de fortalecer el dominio en la interacción entre hardware y software para sistemas de control industrial.

- En la primera práctica, se implementó una secuencia de encendido de LEDs en forma espiral utilizando punteros y temporización no bloqueante. Esto permitió afianzar el uso de estructuras de datos en C++ aplicadas al control digital.
- En la segunda práctica, se diseñó una máquina de estados para controlar una secuencia normal e inversa de LEDs, con gestión mediante botones físicos. La implementación demostró la importancia de estructurar código mediante máquinas de estado finito (FSM) para lograr un control ordenado y escalable.
- En la tercera práctica, se desarrolló una simulación de semáforos utilizando estructuras tipo `struct` y enumeraciones para representar los distintos estados del sistema. La transición entre estados se gestionó mediante temporización no bloqueante y se diagramó como una máquina de estados, mostrando cómo modelar sistemas secuenciales reales.
- Finalmente, en la cuarta práctica, se integró una interfaz HMI y un encoder para implementar un sistema de control de velocidad con retroalimentación (PID). Esta práctica representó el nivel más avanzado, combinando técnicas de muestreo temporal, manejo de interrupciones y control digital en tiempo real, permitiendo observar gráficamente el comportamiento del sistema.

En conjunto, estas prácticas permitieron consolidar conocimientos teóricos mediante aplicaciones prácticas reales. Se evidenció cómo la programación estructurada, el uso de interrupciones, el diseño de FSMs y la implementación de controladores PID pueden aplicarse efectivamente a sistemas físicos utilizando plataformas como Controllino. Además, la integración con una interfaz HMI facilitó la visualización e interacción del usuario con el sistema, mejorando la comprensión del comportamiento dinámico de las variables controladas.

References

[1] H. Maldonado, *Guía de uso de Tableros*, 1st ed., Departamento de Ingeniería Eléctrica, Electrónica y Telecomunicaciones, 2025, manual de usuario, 6 de junio de 2025.

VI. Anexos

Se creo un repositorio en Github en donde se encuentran todos los proyectos de las practicashttps://github.com/elizabeth-123/Control-Digital/tree/main.