# Laboratorio Control Digital
# Práctica 4: Uso de PLC Controllino

1st Mateo Eduardo Bermeo Pesántez, 2nd Santiago Andrés Guillén Malla
**Facultad de Ingeniería - Telecomunicaciones**
**Universidad de Cuenca**
mateo.bermeop@ucuenca.edu.ec [1], santiago.guillenm@ucuenca.edu.ec [2]

## 1. Introducción

El presente informe recopila el desarrollo de cuatro prácticas realizadas con el tablero de automatización que integra el Controllino Mega, el ESP32 PLC 14 y la interfaz HMI STONE. Estas actividades permitieron explorar aspectos fundamentales del control digital, la interacción con sistemas físicos y la adquisición de datos en tiempo real.

En cada práctica se aplicaron conceptos de programación estructurada en Arduino, modelado de comportamientos mediante máquinas de estados finitas (FSM), comunicación serie con HMI, y control de velocidad en motores DC mediante señales PWM y estrategias de control PID. Estas experiencias resultaron clave para afianzar el uso de controladores industriales en escenarios reales de automatización y monitoreo.

## 2. Objetivo General

- Desarrollar habilidades en la programación y configuración de PLCs industriales, utilizando el entorno Arduino IDE con el Controllino Mega y el ESP32 PLC 14 para implementar sistemas de automatización básicos y avanzados.
- Integrar interfaces gráficas HMI para el monitoreo y control de procesos, mediante la comunicación serial y el diseño de widgets interactivos, permitiendo la regulación y visualización de variables en tiempo real.
- Aplicar técnicas de control y adquisición de datos en sistemas físicos reales, como el control de velocidad de motores DC usando señales PWM, interrupciones y controladores PID programados desde cero.

## 3. Desarrollo

### 3.1. Práctica 1: Manejo de Salidas Digitales con Controllino Mega.

En esta práctica se utilizará el módulo Controllino Mega para encender una secuencia de focos LED conectados en el tablero de prácticas. El código será desarrollado utilizando la librería «`Controllino.h`», lo que permitirá familiarizarse con el uso de variables predefinidas para manipular las salidas digitales del dispositivo.

El reto de esta práctica es 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:

Orden de encendido: $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 8 \rightarrow 7 \rightarrow 4 \rightarrow 5$ y se repite.

Para ello, se debe asignar un pin digital a cada LED usando variables predefinidas de la librería «`Controllino.h`». También, cada LED debe encenderse durante $500ms$ y luego apagarse, justo en ese instante se enciende el siguiente. Al finalizar, la secuencia se debe reiniciar. El código es:

```
// Practica 1: Encender todos los LEDs del tablero secuencialmente

#include <Controllino.h> // Se incluye la libreria especifica para usar las salidas del PLC Controllino

// Variables para control de tiempo
unsigned long t_ant = 0;       // Tiempo anterior para control de temporizacion
const long tau = 500;          // Intervalo de tiempo (500 ms) entre encendidos de LEDs

int i = 0;                     // Indice que representa la posicion actual del LED a encender

// Arreglo con los pines de los LEDs que se van a encender en orden secuencial
const int leds[] = {
  CONTROLLINO_D7, CONTROLLINO_D0, CONTROLLINO_D6,
  CONTROLLINO_D12, CONTROLLINO_D13, CONTROLLINO_D14,
  CONTROLLINO_D8, CONTROLLINO_D2, CONTROLLINO_D1,
  CONTROLLINO_D7 // Repite el primero para poder apagarlo al final del ciclo
};

void setup() {
  // Configurar cada pin como salida digital
  pinMode(CONTROLLINO_D0, OUTPUT);
  pinMode(CONTROLLINO_D1, OUTPUT);
  pinMode(CONTROLLINO_D2, OUTPUT);
  pinMode(CONTROLLINO_D6, OUTPUT);
  pinMode(CONTROLLINO_D7, OUTPUT);
  pinMode(CONTROLLINO_D8, OUTPUT);
  pinMode(CONTROLLINO_D12, OUTPUT);
  pinMode(CONTROLLINO_D13, OUTPUT);
  pinMode(CONTROLLINO_D14, OUTPUT);
}

void loop() {
  // Se obtiene el tiempo actual en milisegundos
  unsigned long t_act = millis();

  // Se evalua si ha pasado el intervalo definido (tau)
  if (t_act - t_ant >= tau) {
    t_ant = t_act; // Se actualiza el tiempo anterior

    // Apagar el LED anterior
    digitalWrite(leds[i], LOW);

    // Encender el siguiente LED
    digitalWrite(leds[i+1], HIGH);

    // Avanzar al siguiente indice del arreglo
    i = i + 1;
  }

  // Si se ha llegado al final de la secuencia, reiniciar el indice
  if (i == 9){
    i = 0;
  }
}
```

**Implementación de la lógica de encendido**

La lógica de encendido secuencial de los LEDs fue implementada utilizando técnicas eficientes que permiten un control preciso sin bloquear la ejecución del programa. A continuación, se describen los elementos más relevantes utilizados en el desarrollo:

- **Uso de variables predefinidas de la librería «Controllino.h»**
  Para manejar las salidas digitales del PLC Controllino, se utilizó la librería `Controllino.h`, la cual proporciona constantes predefinidas que representan cada pin del dispositivo (por ejemplo, `CONTROLLINO_D0`, `CONTROLLINO_D1`, etc.). Estas variables facilitan el acceso a las salidas sin necesidad de conocer su numeración interna o física exacta.

- **Uso de punteros para recorrer la secuencia**
  Aunque no se utilizaron punteros explícitos, se empleó un arreglo de enteros (`leds[]`) que actúa como una lista secuencial de pines. A través de un índice entero (`i`), se recorre esta lista y se accede dinámicamente a cada LED usando `leds[i]` y `leds[i+1]`, lo cual simula el comportamiento de un puntero para navegar en memoria. Este método permite controlar de forma flexible qué LED se apaga y cuál se enciende en cada iteración.

- **Técnicas de temporización no bloqueante (uso de millis())**
  Para evitar el uso de funciones bloqueantes como `delay()`, se implementó una técnica de temporización no bloqueante utilizando la función `millis()`. Esta función proporciona el tiempo transcurrido desde el arranque del sistema en milisegundos, lo que permite comparar el tiempo actual (`t_act`) con un tiempo anterior (`t_ant`). Si ha transcurrido el intervalo deseado ($tau = 500ms$), se actualiza el tiempo y se realiza el cambio de LED. Esta técnica permite que el programa siga ejecutando otras tareas en paralelo si fuera necesario, algo fundamental en sistemas embebidos o industriales donde la eficiencia del ciclo de control es clave.

FUNCIONAMIENTO:
https://drive.google.com/drive/folders/1AbHg2ov4n7ax_nQgJ3EJ3XD61SdEtcfS?usp=sharing

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

Esta práctica tiene como objetivo familiarizar con el uso básico del PLC Controllino Mega, mediante el encendido y apagado de un foco LED por dos pulsantes en el tablero respectivamente, ademas, la practica se complementa con un reto que consiste en implementar mediante maquinas de estado finito (FMS) un semáforo funcional entre dos intersecciones, el mismo esta conformado por retardos no bloqueantes con el fin de hacer uso del reloj de la tarjeta Controllino.

**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 el código que muestra a la salida una secuencia definida de encendido de leds, haciendo uso del manejo de los botones del hardware:

```
#include <Controllino.h>

// ====== ENUMERACION DE ESTADOS =======
enum EstadoSistema { APAGADO, ESPIRAL_NORMAL, ESPIRAL_INVERSO };
EstadoSistema estadoActual = APAGADO;
```

```
6
7  // ====== STRUCT PARA LEDS ============
8  struct Led {
9    int pin;
10   bool estado;
11 };
12
13 // ====== MATRIZ DE LEDS ===============
14 Led matriz[9] = {
15   {CONTROLLINO_D0, false}, {CONTROLLINO_D6, false}, {CONTROLLINO_D12, false
       },
16   {CONTROLLINO_D13, false}, {CONTROLLINO_D14, false}, {CONTROLLINO_D8,
       false},
17   {CONTROLLINO_D2, false}, {CONTROLLINO_D1, false}, {CONTROLLINO_D7, false}
18 };
19
20 // ====== BOTONES ======================
21 const int BTN_ESPIRAL_NORMAL = CONTROLLINO_I16;
22 const int BTN_ESPIRAL_INVERSO = CONTROLLINO_I17;
23 const int BTN_RESET = CONTROLLINO_I18;
24
25 // ====== VARIABLES DE TIEMPO =========
26 unsigned long tiempoAnterior = 0;
27 const unsigned long intervalo = 250;  // milisegundos
28 int indice = 0;
29
30 void setup() {
31   // Configurar pines de LEDs como salidas
32   for (int i = 0; i < 9; i++) {
33     pinMode(matriz[i].pin, OUTPUT);
34     digitalWrite(matriz[i].pin, LOW);
35   }
36
37   // Configurar botones como entradas
38   pinMode(BTN_ESPIRAL_NORMAL, INPUT);
39   pinMode(BTN_ESPIRAL_INVERSO, INPUT);
40   pinMode(BTN_RESET, INPUT);
41 }
42
43 void loop() {
44   leerBotones();
45   actualizarPatron();
46 }
47
48 // ====== LECTURA DE BOTONES ==========
49 void leerBotones() {
50   if (digitalRead(BTN_ESPIRAL_NORMAL) == HIGH) {
51     estadoActual = ESPIRAL_NORMAL;
52     indice = 0;
53     apagarTodos();
54   }
55   else if (digitalRead(BTN_ESPIRAL_INVERSO) == HIGH) {
56     estadoActual = ESPIRAL_INVERSO;
57     indice = 9;
58     apagarTodos();
59   }
60   else if (digitalRead(BTN_RESET) == HIGH) {
61     estadoActual = APAGADO;
62     apagarTodos();
63   }
64 }
65
66 // ====== ACTUALIZACION NO BLOQUEANTE ======
```

```
67 void actualizarPatron() {
68   unsigned long tiempoActual = millis();
69
70   if (estadoActual == ESPIRAL_NORMAL && tiempoActual - tiempoAnterior >=
       intervalo) {
71     tiempoAnterior = tiempoActual;
72     if (indice < 10) {
73       apagarLed(indice-1);
74       encenderLed(indice);
75       indice++;
76       if (indice == 10) {indice = 0;}
77     }
78   }
79
80   else if (estadoActual == ESPIRAL_INVERSO && tiempoActual - tiempoAnterior
        >= intervalo) {
81     tiempoAnterior = tiempoActual;
82     if (-1 < indice) {
83       apagarLed(indice+1);
84       encenderLed(indice);
85       indice--;
86       if (indice == -1) {indice = 9;}
87     }
88   }
89 }
90
91 // ====== ENCENDER LED INDIVIDUAL ======
92 void encenderLed(int indice) {
93   matriz[indice].estado = true;
94   digitalWrite(matriz[indice].pin, HIGH);
95 }
96
97 // ====== APAGAR LED INDIVIDUAL ======
98 void apagarLed(int indice) {
99   matriz[indice].estado = false;
100  digitalWrite(matriz[indice].pin, LOW);
101 }
102
103 // ====== APAGAR TODOS LOS LEDS ========
104 void apagarTodos() {
105   for (int i = 0; i < 9; i++) {
106     matriz[i].estado = false;
107     digitalWrite(matriz[i].pin, LOW);
108   }
109 }
```

Este programa permite controlar una secuencia de encendido de 9 LEDs dispuestos en una matriz 3x3 utilizando tres botones físicos conectados al Controllino Mega. Mediante una máquina de estados finita (FSM), el sistema permite seleccionar entre dos modos de encendido: espiral normal e inverso, además, cuenta con un botón para detener o reiniciar el sistema.

Cada pulsación de botón cambia el estado del sistema y reinicia la secuencia correspondiente. La lógica se implementa con temporización no bloqueante usando `millis()`, lo que permite actualizar los LEDs sin detener el flujo del programa. Se emplea un arreglo de estructuras para representar los LEDs, y funciones auxiliares para encender, apagar y reiniciar todos los elementos de forma ordenada y eficiente.

**Parte B:**

Diseñar un sistema que controle dos semáforos (Semáforo A y Semáforo B) ubicados en una intersección perpendicular (1), usando el enfoque demáquina 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.

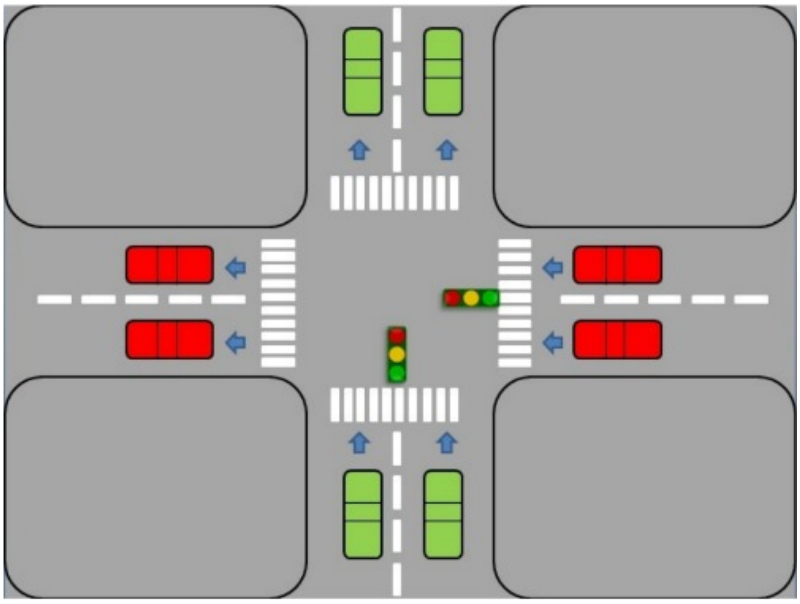

Figura 1: Circulación vehicular simple

Una vez realizada la secuencia mediante retardos no bloqueantes, estos son un preámbulo para la implementación del reto; este consiste el realizar un semáforo de una intersección usando este mismo concepto de retardos en el tiempo de transición entre cada estado del semáforo, a continuación se muestra el código implementado, posteriormente se explican los puntos mas importantes del mismo:

```
#include <Controllino.h>

// Enum para estados
enum EstadoSem { A_VERDE, A_AMARILLO, B_VERDE, B_AMARILLO };
EstadoSem estado = A_VERDE;

// estructura del semaforo
struct Semaforo {
  int rojo;
  int amarillo;
  int verde;
};

// Semaforo A: trafico vertical
Semaforo A = {CONTROLLINO_D0, CONTROLLINO_D1, CONTROLLINO_D2};

// Semaforo B: trafico horizontal
Semaforo B = {CONTROLLINO_D12, CONTROLLINO_D13, CONTROLLINO_D14};

// Temporizacion
unsigned long tiempoEstado = 0;
const unsigned long DURACION_VERDE = 5000;
const unsigned long DURACION_AMARILLO = 2000;

void setup() {
  // Configurar pines como salidas
  pinMode(A.rojo, OUTPUT);
  pinMode(A.amarillo, OUTPUT);
  pinMode(A.verde, OUTPUT);

  pinMode(B.rojo, OUTPUT);
  pinMode(B.amarillo, OUTPUT);
  pinMode(B.verde, OUTPUT);

  // Iniciar en estado A_VERDE
```

```
  tiempoEstado = millis();
  actualizarSemaforos();
}

void loop() {
  unsigned long tiempoActual = millis();

  switch (estado) {
    case A_VERDE:
      if (tiempoActual - tiempoEstado >= DURACION_VERDE) {
        estado = A_AMARILLO;
        tiempoEstado = tiempoActual;
        actualizarSemaforos();
      }
      break;

    case A_AMARILLO:
      if (tiempoActual - tiempoEstado >= DURACION_AMARILLO) {
        estado = B_VERDE;
        tiempoEstado = tiempoActual;
        actualizarSemaforos();
      }
      break;

    case B_VERDE:
      if (tiempoActual - tiempoEstado >= DURACION_VERDE) {
        estado = B_AMARILLO;
        tiempoEstado = tiempoActual;
        actualizarSemaforos();
      }
      break;

    case B_AMARILLO:
      if (tiempoActual - tiempoEstado >= DURACION_AMARILLO) {
        estado = A_VERDE;
        tiempoEstado = tiempoActual;
        actualizarSemaforos();
      }
      break;
  }
}

// Actualiza los semaforos segun el estado actual
void actualizarSemaforos() {
  switch (estado) {
    case A_VERDE:
      setSemaforo(A, LOW, LOW, HIGH);    // A verde
      setSemaforo(B, HIGH, LOW, LOW);    // B rojo
      break;

    case A_AMARILLO:
      setSemaforo(A, LOW, HIGH, LOW);    // A amarillo
      setSemaforo(B, HIGH, LOW, LOW);    // B rojo
      break;

    case B_VERDE:
      setSemaforo(A, HIGH, LOW, LOW);    // A rojo
      setSemaforo(B, LOW, LOW, HIGH);    // B verde
      break;

    case B_AMARILLO:
      setSemaforo(A, HIGH, LOW, LOW);    // A rojo
      setSemaforo(B, LOW, HIGH, LOW);    // B amarillo
```

```
99        break;
100     }
101   }
102
103   // Enciende el semaforo con los valores dados
104   void setSemaforo(Semaforo s, bool r, bool a, bool v) {
105     digitalWrite(s.rojo, r);
106     digitalWrite(s.amarillo, a);
107     digitalWrite(s.verde, v);
108   }
```

Este programa simula el funcionamiento de un sistema de semáforos para una intersección de dos vías perpendiculares, utilizando una máquina de estados finita (FSM) para alternar el paso entre el tráfico vertical (Semáforo A) y horizontal (Semáforo B). La lógica se basa en cambios de estado temporizados con `millis()` para evitar retardos bloqueantes. A continuación se enumeran aspectos importantes para el funcionamiento del programa:

1. **Definición de estados y estructuras:** En las líneas 4–5 se define un `enum EstadoSem` que contiene los cuatro estados posibles del sistema: `A_VERDE`, `A_AMARILLO`, `B_VERDE` y `B_AMARILLO`, correspondientes a los colores que puede mostrar cada semáforo en la secuencia. La estructura `Semaforo` (líneas 7–10) representa un semáforo individual con tres salidas: rojo, amarillo y verde.

2. **Inicialización de semáforos:** En las líneas 14-18, se declaran dos objetos Semaforo: A y B, que corresponden al tráfico vertical y horizontal respectivamente. A cada uno se le asignan tres pines del Controllino Mega para controlar las salidas de los LEDs.

3. **Temporización:** Las variables `tiempoEstado` y las constantes (líneas 20–23)`DURACION_VERDE` y `DURACION_AMARILLO` controlan el tiempo que permanece activo cada estado de luz, usando unidades en milisegundos ($5000ms = 5s$ para verde, $2000ms = 2s$ para amarillo).

4. **Configuración inicial:** En el `setup()` (líneas 25–38), se configuran los pines de cada luz como salidas, se guarda el tiempo de inicio del estado actual con `millis()`, y se actualiza la configuración inicial de los semáforos mediante la función `actualizarSemaforos()`.

5. **Máquina de estados finita (FSM):** La función `loop()` (líneas 40–76) contiene la FSM que transita entre los estados definidos.

   En cada iteración, se compara el tiempo actual (`millis()`) con el tiempo almacenado en `tiempoEstado`.

   Cuando el tiempo transcurrido supera la duración del estado actual (`DURACION_VERDE` o `DURACION_AMARILLO`), se avanza al siguiente estado y se actualizan los semáforos.

   Ejemplo:
   Al terminar `A_VERDE` (línea 44), se pasa a `A_AMARILLO` y se llama a `actualizarSemaforos()`.

6. **Actualización de luces según estado:** La función `actualizarSemaforos()` (líneas 79–101) enciende o apaga los LEDs de cada semáforo según el estado actual. Se utiliza la función auxiliar `setSemaforo()` para aplicar los valores deseados a cada luz.

   Ejemplo:
   En `A_VERDE` (línea 81), se enciende la luz verde de A (HIGH) y la roja de B (HIGH), dejando el resto apagado.

7. **Encendido de luces del semáforo:** Finalmente, `setSemaforo()` (líneas 103–108) recibe como parámetros un semáforo y tres valores booleanos para encender o apagar sus luces. Esta función encapsula el uso de `digitalWrite()` y mejora la legibilidad del código.

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

Esta práctica tiene como objetivo controlar el brillo de un foco LED mediante modulación por ancho de pulso (PWM), utilizando una interfaz gráfica en una pantalla HMI (Human-Machine

Interface). El valor del duty cycle será enviado desde la HMI al microcontrolador a través de comunicación serial, empleando la librería correspondiente. Esto permite ajustar dinámicamente la intensidad luminosa del LED desde la interfaz.

Para esta práctica se usará la HMI configurada con un widget Spin Box, que envía valores de 0 a 100 correspondientes al porcentaje del duty cycle, el cual ajusta el valor del PWM de una salida digital.

Para ello se busca ampliar el programa actual (proporcionado por el docente en la guía) para controlar dos LEDs de forma independiente, utilizando elementos de la interfaz gráfica (spin boxes) para regular el brillo de cada LED y botones físicos del tablero para encender y apagar cada uno de ellos.

**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).

Teniendo esto en cuenta, se procede a realizar la configuración del entorno visual del HMI con la ayuda del Software Stone Designer GUI, dando como resultado la interfaz que se aprecia en la Figura 2.

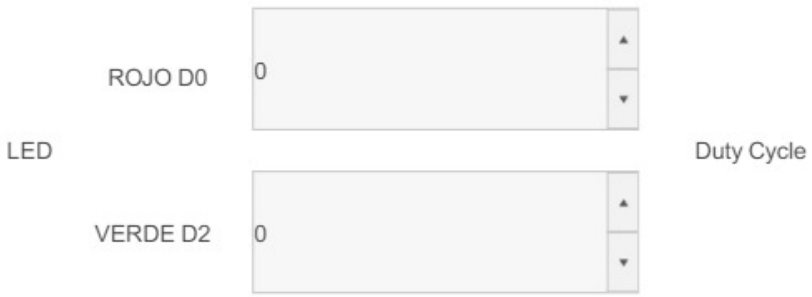


Figura 2: Interfaz visual HMI - Intensidad de LEDs

Luego, se procede a la programación en Arduino y con ello lo siguiente:

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

const int led = CONTROLLINO_D0;          // Salida digital D0
const int ledpractica = CONTROLLINO_D2; // Led D2 de la practica (reto)

const int botonRojo = CONTROLLINO_I16;  // Boton para led D0
const int botonVerde = CONTROLLINO_I18; // Boton para led D2

int       bandera          = 0;         // bandera para no repetir el
    mensaje de ingresar un valor al HMI
int       pwmValue         = 0;         // valor convertido (0-255)
int       pwmValue2        = 0;

// Estados para los botones
bool estadoLED1 = false;
bool estadoLED2 = false;

bool boton1Prev = HIGH;
bool boton2Prev = HIGH;

// Valor en porcentaje (0-100)
float dutyCyclePercent = 0;
float dutyCyclePercent2 = 0;


void setup() {
  Serial.begin(115200);                 // Comunicaci n serial con el PC
  Serial2.begin(115200);               // Comunicaci n serial con el HMI

  // Leds de salida
  pinMode(led, OUTPUT);
  pinMode(ledpractica, OUTPUT);

  // Botones
  pinMode(botonRojo, INPUT);
  pinMode(botonVerde, INPUT);

  HMI_init();                           // Inicializa el sistema de colas
    para las respuestas el HMI
  Stone_HMI_Set_Value("spin_box", "spin_box1", NULL, 0);  // Pone en 0 el
    valor del spin box en el HMI.
  Stone_HMI_Set_Value("spin_box", "spin_box2", NULL, 0);  // Pone en 0 el
    valor del spin box en el HMI (reto).
}

void loop() {
  dutyCyclePercent = HMI_get_value("spin_box", "spin_box1");  // Obtiene el
     valor del spin_box1
  dutyCyclePercent2 = HMI_get_value("spin_box", "spin_box2"); // Obtiene el
     valor del spin_box2

  // Leer boton 1 para led D0 - rojo
  bool boton1 = digitalRead(botonRojo);
  if (boton1 == LOW && boton1Prev == HIGH) {  // Transicion de HIGH a LOW
    estadoLED1 = !estadoLED1;  // Cambia el estado del LED
  }
  boton1Prev = boton1;

  // Leer boton 2 para led D2 - verde
  bool boton2 = digitalRead(botonVerde);
  if (boton2 == LOW && boton2Prev == HIGH) {
```

```
    estadoLED2 = !estadoLED2;
  }
  boton2Prev = boton2;

  // LED 1 PWM
  if (estadoLED1 && dutyCyclePercent >= 0 && dutyCyclePercent <= 100) {
    pwmValue = map(dutyCyclePercent, 0, 100, 0, 255);      // Mapea el
    valor de duty cycle en porcentaje a valores de 0 a 255
    analogWrite(led, pwmValue);
    Serial.print("Duty cycle 1 (%): ");
    Serial.print(dutyCyclePercent);
    Serial.print(" -> PWM value: ");
    Serial.println(pwmValue);
    int bandera = 0;
  }
  else if (bandera == 0){
    int bandera = 1;
    Serial.println("Ingresa un valor entre 0 y 100 para led1.");
    analogWrite(led, 0);
  }

  // LED 2 PWM
  if (estadoLED2 && dutyCyclePercent2 >= 0 && dutyCyclePercent2 <= 100) {
    pwmValue2 = map(dutyCyclePercent2, 0, 100, 0, 255);      // Mapea el
    valor de duty cycle en porcentaje a valores de 0 a 255
    analogWrite(ledpractica, pwmValue2);
    Serial.print("Duty cycle 2 (%): ");
    Serial.print(dutyCyclePercent2);
    Serial.print(" -> PWM value: ");
    Serial.println(pwmValue2);
    Serial.println("----------------------");
    int bandera = 0;
  }
  else if (bandera == 0){
    int bandera = 1;
    Serial.println("Ingresa un valor entre 0 y 100 para led2.");
    analogWrite(ledpractica, 0);
  }
}
```

De esta manera, el objetivo principal fue implementar una lógica que permitiera encender y apagar cada LED con un botón del tablero, mientras que su brillo fuera regulado por un widget tipo SpinBox desde la interfaz HMI.

**Funcionalidades implementadas**

- **Control de dos LEDs por PWM:** Se utilizaron dos salidas digitales del Controllino (`D0` y `D2`) configuradas como salidas PWM. La intensidad de cada LED es regulada mediante valores de ciclo de trabajo (`duty cycle`) en porcentaje, enviados desde la interfaz gráfica.
- **Uso de SpinBoxes en HMI Stone:** Se añadieron dos widgets tipo SpinBox en la interfaz.
  - `spin_box1` controla el brillo del LED 1 (D0).
  - `spin_box2` controla el brillo del LED 2 (D2).

  Cada uno envía un valor entre 0 y 100, que se convierte internamente a un rango de 0 a 255 mediante la función `map()`, para generar la señal PWM adecuada.
- **Botones físicos para control ON/OFF:** Se configuraron dos botones físicos conectados a entradas digitales (`I16` e `I18`) del Controllino:
  - El primer botón controla el estado de encendido del LED 1.
  - El segundo botón controla el estado del LED 2.

La lógica implementada detecta el flanco de bajada (cambio de HIGH a LOW) en cada botón, y cambia el estado del LED correspondiente (encendido/apagado). Esta lógica garantiza que el SpinBox solo controla el brillo si el botón ha habilitado previamente el LED.

- **Comunicaciones y librerías utilizadas:**
  - La librería `Controllino.h` se usó para acceder a pines digitales con nombres predefinidos.
  - Se utilizó `Serial2` a 115200 baudios para la comunicación entre el Controllino y la pantalla HMI Stone.
  - Las funciones `HMI_get_value()` y `Stone_HMI_Set_Value()` permiten la interacción con los widgets de la interfaz gráfica.
  - La función `HMI_init()` inicializa el sistema de comunicación con la pantalla.
- **Validación de entradas:** Se verificó que los valores de los SpinBoxes estén dentro del rango válido (0 a 100) antes de aplicarlos a los LEDs. Si el valor es inválido, el sistema emite un mensaje por el monitor serial solicitando corregir el dato.

## 3.4. Práctica 4: Adquisición de datos de motor y gráfica.

Esta práctica consiste en controlar la alimentación del motor DC del Entrenador de Planta de Control (EPC) mediante una señal PWM regulada con un slider, al tiempo que se recogen los datos necesarios para calcular su velocidad en RPM y se muestran tanto el duty cycle aplicado como la velocidad del motor en una gráfica del HMI Stone.

El motor del EPC requiere una señal de control de 5 V, pero las salidas digitales del Controllino Mega funcionan a 24 V, por lo que se usa el puerto X1, que opera a 5 V.

En concreto, el pin D0 de X1 emite la señal PWM al motor DC, mientras que la salida MDC del encoder del EPC genera pulsos que entran por el pin IN1 de ese mismo puerto.

Según la ficha técnica del EPC, 36 pulsos equivalen a 1 RPM; esta relación se programa en el Controllino Mega para convertir el conteo de pulsos en velocidad.

Para ello se configuran dos interrupciones:

- Una interrupción por timer de comparación que, cada segundo, recalcula las RPM del motor.
- Una interrupción externa utilizada para contar los pulsos procedentes de MDC, necesarios para calcular las RPM.

Finalmente, tanto el porcentaje de ciclo de trabajo como la velocidad resultante se representan gráficamente en el HMI.

**Descripción de la actividad:**

Diseñar e implementar un controlador PID que regule automáticamente la velocidad del motor DC del EPC a un valor de referencia definido por el usuario mediante la interfaz HMI. Para ello se debe tener en cuenta el siguiente esquema de lazo de control (Figura 3):

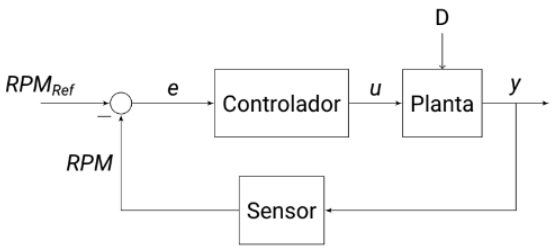


Figura 3: Esquema de lazo de control con controlador y planta

Se debe implementar un sistema de control PID que cumpla con las siguientes condiciones:

- La velocidad del motor debe alcanzar el valor de referencia en el menor tiempo posible, sin sobrepasarlo excesivamente.
- El sistema debe mostrar estabilidad (sin oscilaciones) y un error en estado estacionario inferior al 5 %.
- El sistema debe mostrar en tiempo real: valor de referencia (setpoint) en el slider, velocidad actual (RPM), gráfica de la señal de control, comparativa entre referencia y velocidad actual.

Teniendo esto en cuenta, se procede a realizar la configuración del entorno visual del HMI con la ayuda del Software Stone Designer GUI, dando como resultado la interfaz que se aprecia en la Figura 4.

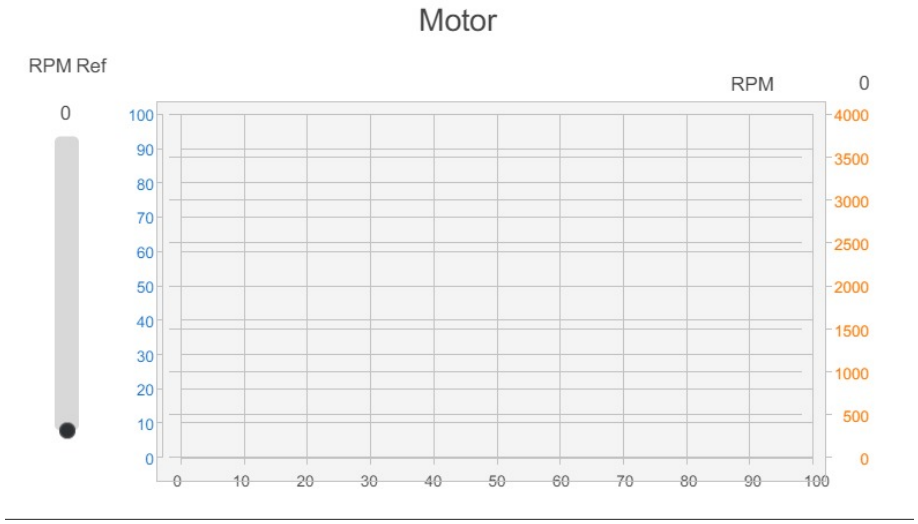


Figura 4: Interfaz visual HMI - PID Motor

Luego, se procede a la programación en Arduino y con ello lo siguiente:

```
// LIBRERIAS NECESARIAS
#include <Controllino.h>                // Libreria para acceder facilmente a los pines del PLC Controllino
#include "Stone_HMI_Define.h"           // Libreria oficial para comunicacion con HMI Stone
#include "Procesar_HMI.h"               // Funciones personalizadas para procesar valores del HMI

// DEFINICION DE PINES Y VARIABLES PARA CONTROL DE MOTOR
const int pin_motor = CONTROLLINO_D0;      // Pin de salida PWM para el motor
const int entrada = CONTROLLINO_IN1;       // Entrada digital que recibe los pulsos del sensor (encoder o similar)

// CONTADOR DE PULSOS Y VARIABLES DE CALCULO DE RPM
volatile unsigned long conteo_pulsos = 0; // Contador de pulsos generado por interrupcion
float rpm = 0;                            // Velocidad del motor en RPM
const uint16_t PULSOS_POR_REV = 36;       // Cantidad de pulsos por revolucion del sensor

// VARIABLES PARA COMUNICACION CON HMI
char label2_text[10];      // Texto para mostrar el valor del PWM en el HMI
char label4_text[10];      // Texto para mostrar la RPM actual
int rpm_ref = 0;           // Valor de referencia de RPM ingresado desde el HMI
```

```
19
20 // CONTROL DE TIEMPO
21 unsigned long t_previo = 0; // Tiempo anterior para temporizacion con
       millis()
22
23 // VARIABLES DEL CONTROLADOR PID
24 const float Ts = 0.05;     // Periodo de muestreo del PID: 50 ms
25 const float Kp = 0.5;      // Ganancia proporcional (ajustable)
26 const float Ki = 1.0;      // Ganancia integral
27 const float Kd = 0.01;     // Ganancia derivativa
28
29 float e_prev = 0;          // Error anterior (para derivada)
30 float integral = 0;        // Acumulador de la parte integral
31 float pwm_out = 0;         // Valor calculado del PWM a aplicar
32
33 // Prototipo de funcion para ISR externa
34 void contarPulso();
35
36 // ---------------------- SETUP ----------------------
37 void setup() {
38   Serial.begin(115200);       // Comunicacion serial con PC
39   Serial2.begin(115200);      // Comunicacion serial con pantalla HMI
40
41   // Configuracion de pines
42   pinMode(entrada, INPUT);            // Pin del sensor de pulsos
43   pinMode(pin_motor, OUTPUT);         // Pin PWM del motor
44
45   // Interrupcion externa para conteo de pulsos (FALLING = flanco de bajada
       )
46   attachInterrupt(digitalPinToInterrupt(entrada), contarPulso, FALLING);
47
48   // Configuracion del Timer1 para generar interrupciones cada 50ms
49   noInterrupts();                     // Desactivar interrupciones durante
       configuracion
50   TCCR1A = 0;
51   TCCR1B = 0;
52   TCCR1B |= B00000100;              // Prescaler de 256 (16MHz/256 = 62.5
       kHz)
53   TIMSK1 |= B00000010;              // Habilita interrupcion por
       comparacion con OCR1A
54   OCR1A = 3125;                     // 62.5kHz / 3125 = 20 Hz = cada 50ms
55   interrupts();                     // Activar interrupciones
56
57   // Inicializacion del HMI
58   HMI_init();
59   STONE_push_series("line_series", "line_series1", 0); // Curva referencia
60   STONE_push_series("line_series", "line_series2", 0); // Curva PWM
61   STONE_push_series("line_series", "line_series3", 0); // Curva RPM real
62   Stone_HMI_Set_Value("slider", "slider1", NULL, 0);   // Reinicia el
       slider del HMI a 0
63 }
64
65 // ---------------------- LOOP ----------------------
66 void loop() {
67   // Ejecutar cada 100 ms
68   if (millis() - t_previo >= 100) {
69     t_previo = millis();
70
71     // Leer referencia desde el HMI (valor de slider)
72     rpm_ref = HMI_get_value("slider", "slider1");
73
74     // Convertir PWM a porcentaje y mostrar en HMI
75     dtostrf((pwm_out / 255.0) * 100.0, 7, 2, label2_text); // PWM en %
```

```
    dtostrf(rpm, 7, 2, label4_text);                          // RPM real

    // Mostrar valores en etiquetas del HMI
    Stone_HMI_Set_Text("label", "label2", label2_text);
    Stone_HMI_Set_Text("label", "label4", label4_text);

    // Enviar datos para graficar en la interfaz
    STONE_push_series("line_series", "line_series1", rpm_ref);
    STONE_push_series("line_series", "line_series2", (pwm_out / 255.0) *
    100.0);
    STONE_push_series("line_series", "line_series3", rpm);
  }
}

// ---------------------- INTERRUPCION POR TIMER1 ----------------------
ISR(TIMER1_COMPA_vect) {
  TCNT1 = 0; // Reiniciar el contador

  // Calculo de la RPM a partir de los pulsos contados
  rpm = (float(conteo_pulsos) * 60) / (PULSOS_POR_REV * (1.0 / Ts));
  conteo_pulsos = 0; // Reiniciar contador

  // ----------------- PID -----------------
  float error = rpm_ref - rpm; // Error actual

  integral += error * Ts;      // Acumulacion de la integral
  float derivative = (error - e_prev) / Ts; // Derivada
  e_prev = error;              // Actualizar error anterior

  // Calcular salida del PID
  pwm_out = Kp * error + Ki * integral + Kd * derivative;

  // Limitar salida (saturacion + anti-windup)
  if (pwm_out > 255) {
    pwm_out = 255;
    integral -= error * Ts; // Anti-windup
  } else if (pwm_out < 0) {
    pwm_out = 0;
    integral -= error * Ts; // Anti-windup
  }

  // Aplicar valor PWM al motor
  analogWrite(pin_motor, (int)pwm_out);

  // DEBUG por monitor serial
  Serial.print("RPM: ");
  Serial.print(rpm);
  Serial.print(" | Ref: ");
  Serial.print(rpm_ref);
  Serial.print(" | PWM: ");
  Serial.println(pwm_out);
}

// ---------------------- INTERRUPCION EXTERNA ----------------------
void contarPulso() {
  conteo_pulsos++; // Suma 1 cada vez que se detecta un pulso del sensor
}
```

En esta práctica se implementó un sistema completo de control automático de velocidad para un motor DC, utilizando un controlador PID y una interfaz gráfica HMI Stone. El objetivo fue mantener la velocidad del motor (en RPM) cercana a un valor de referencia definido por el usuario desde la pantalla HMI, utilizando una señal PWM generada por el Controllino.

**Funcionalidades implementadas:**

- **Medición de velocidad en RPM:** Un sensor conectado al eje del motor genera una cierta cantidad de pulsos por revolución. Mediante una interrupción externa se cuenta el número de pulsos y, cada $50ms$, se calcula la velocidad actual del motor en RPM.
- **Control automático por PID:** Se implementó un controlador PID discreto que, con una frecuencia de $20Hz$ (cada $50ms$), calcula el error entre la velocidad deseada (referencia) y la velocidad actual. A partir del error se genera una señal PWM proporcional, integral y derivativa que se aplica al motor para ajustar su velocidad.
- **Interfaz gráfica con HMI Stone:** El usuario puede ajustar la velocidad deseada desde un slider en la pantalla HMI (`slider1`). Además, se muestran en tiempo real los siguientes valores:
  - La referencia de velocidad (RPM).
  - El valor de la señal PWM aplicado al motor (en porcentaje).
  - La velocidad actual del motor (RPM).
- **Visualización de datos en gráficas:** Se enviaron datos continuamente a tres gráficas:
  - Referencia (line_series1),
  - PWM aplicado (line_series2),
  - RPM medida (line_series3).

  Esto permite comparar visualmente la respuesta del sistema y evaluar el comportamiento del PID.
- **Optimización con interrupciones y temporizadores:** Se usaron interrupciones para:
  - Detectar pulsos del sensor (precisión en conteo).
  - Ejecutar el control PID periódicamente usando el `TIMER1` (sin bloquear el `loop()`).

Este sistema simula un control de velocidad cerrado típico en aplicaciones industriales, y demuestra la integración efectiva entre sensores, controladores PID, actuadores y una interfaz gráfica para el operador.

# 4. Conclusiones

- La implementación del sistema de semáforos mediante una máquina de estados finita (FSM) permitió simular de forma precisa el comportamiento de una intersección vial, garantizando una operación segura y ordenada. El uso de estructuras (`struct`) facilitó la gestión de cada semáforo como una entidad independiente, mejorando la modularidad del código. Además, el empleo de temporización no bloqueante con `millis()` permitió una transición fluida entre estados sin detener la ejecución del programa, lo cual es fundamental en sistemas embebidos donde se requiere concurrencia y eficiencia. Esta práctica reforzó el conocimiento sobre lógica secuencial, control de salidas digitales y diseño estructurado de código en aplicaciones de automatización.
- Las actividades permitieron consolidar el conocimiento en control de salidas digitales y modulación PWM utilizando el Controllino, tanto en tareas básicas como el encendido secuencial de LEDs, como en aplicaciones más avanzadas como el control de velocidad de un motor DC con retroalimentación. La incorporación de la interfaz HMI Stone brindó una plataforma

intuitiva para la interacción con el usuario, permitiendo el ajuste de parámetros y la visualización de resultados en tiempo real, lo cual es fundamental en aplicaciones industriales modernas.

- La implementación de un controlador PID discreto y su ejecución a través de interrupciones temporizadas demostró la importancia de las técnicas no bloqueantes y de tiempo real en sistemas de control. A su vez, el uso de botones físicos para control manual, combinado con entradas gráficas (SpinBoxes, sliders), permitió explorar esquemas híbridos de operación manual y automática, reforzando así competencias clave en programación embebida, electrónica de control y diseño de interfaces hombre-máquina.