

Prácticas

Tableros PLC de código abierto

Juan Durán - Mateo Dután
 Universidad de Cuenca
 Facultad de Ingeniería
 Cuenca, Ecuador
 juanj.duran@ucuenca.edu.ec
 mateo.dutan@ucuenca.edu.ec

I. PRÁCTICA 1

MANEJO DE SALIDAS DIGITALES DE 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.

I-A. Objetivos

- Comprender cómo utilizar variables predefinidas para generar una secuencia de encendido de focos LED, tal como se haría en una aplicación de control secuencial básica en automatización.

I-B. Materiales

- Tablero de control con Controllino Mega integrado.
- Fuente de alimentación del tablero.
- Cable USB tipo B 2.0.
- PC con Arduino IDE instalado y configurado para Controllino.

I-C. Desarrollo

I-C1. Hardware: Para comenzar con el desarrollo de esta práctica primero realizamos las conexiones necesarias para que el tablero y controllino funcionen adecuadamente, para ello activamos el switch con la etiqueta **FUENTE AC/DC**, del mismo modo el que tiene la etiqueta **CONTROLLINO**, para después conectarlo a nuestra PC en la que verificaremos el puerto en el que está conectado.

I-C2. Firmware: Como paso inicial vamos a descargarnos un código cuyo link fue dado en la guía de la práctica, este código es el que cargamos al controllino que lo que hace básicamente es encender los leds D0, D1, D2, D6, D7, D8, D12, D13, D14 de forma secuencial.

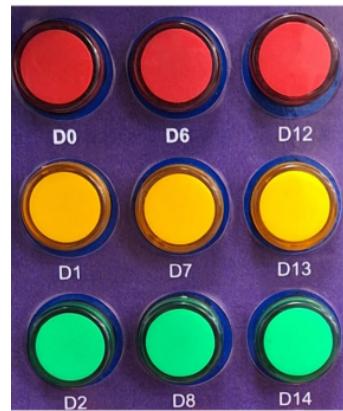


Figura 1. Leds a usar

I-C3. Reto: Nuestro trabajo en esta práctica consiste en realizar una secuencia similar a la proporcionada como ejemplo controlando la 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:**D0** → **D6** → **D12** → **D13** → **D14** → **D8** → **D2** → **D1** → **D7** y que se repita indefinidamente, es decir se apaga el led D7 y vuelve a comenzar desde el led D0 nuevamente.

Los requisitos más importantes son usar retardos no bloqueantes y punteros.

Para iniciar y entender correctamente el uso de las salidas vamos únicamente a generar la secuencia sin punteros y con retardos bloqueantes como el comando **delay()**, comenzamos por incluir la librería y declarar los pines de los leds como salida.

```
#include <Controllino.h> // Librería de controllino

void setup() {
    pinMode(CONTROLLINO_D0, OUTPUT); // Salida digital
    ↪ D0
    pinMode(CONTROLLINO_D1, OUTPUT); // Salida digital
    ↪ D1
    pinMode(CONTROLLINO_D2, OUTPUT); // Salida digital
    ↪ D2
    pinMode(CONTROLLINO_D6, OUTPUT); // Salida digital
    ↪ D6
    pinMode(CONTROLLINO_D7, OUTPUT); // Salida digital
    ↪ D7
```

```

pinMode(CONTROLLINO_D8, OUTPUT); // Salida digital
  ↪ D8
pinMode(CONTROLLINO_D12, OUTPUT); // Salida
  ↪ digital D12
pinMode(CONTROLLINO_D13, OUTPUT); // Salida
  ↪ digital D13
pinMode(CONTROLLINO_D14, OUTPUT); // Salida
  ↪ digital D14
}

```

En el loop ocurre la secuencia, para ello solo ponemos uno por uno los led en alto (HIGH) con un delay esperamos 500 ms y lo apagamos (LOW), y así para cada led, lo que resulta poco eficiente e incluso demorado.

```

void loop() {
  digitalWrite(CONTROLLINO_D0, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D0, LOW);

  digitalWrite(CONTROLLINO_D6, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D6, LOW);

  digitalWrite(CONTROLLINO_D12, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D12, LOW);

  digitalWrite(CONTROLLINO_D13, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D13, LOW);

  digitalWrite(CONTROLLINO_D14, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D14, LOW);

  digitalWrite(CONTROLLINO_D8, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D8, LOW);

  digitalWrite(CONTROLLINO_D2, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D2, LOW);

  digitalWrite(CONTROLLINO_D1, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D1, LOW);

  digitalWrite(CONTROLLINO_D7, HIGH);
  delay(500);
  digitalWrite(CONTROLLINO_D7, LOW);
}

```

Ahora usaremos **punteros**, lo que hará que el loop sea mucho más pequeño ya que no iremos led por led encendiendo y apagando si no que crearemos un vector con la posición de cada led e ir recorriendo este vector con el uso del puntero para que se enciendan en la secuencia especificada previamente. Lo primero que hacemos es declarar las salidas de los leds en el **setup()**, esta parte es la misma que el código anterior por lo que no la pondré ahora y me saltaré al **loop()** donde se evidencia la diferencia de usar punteros.

```

void loop() {
  int i = 0;

  int D0 = CONTROLLINO_D0;
  int D1 = CONTROLLINO_D1;
  int D2 = CONTROLLINO_D2;
  int D6 = CONTROLLINO_D6;
  int D7 = CONTROLLINO_D7;
  int D8 = CONTROLLINO_D8;
  int D12 = CONTROLLINO_D12;

```

```

int D13 = CONTROLLINO_D13;
int D14 = CONTROLLINO_D14;

int leds[9] = {D0, D6, D12, D13, D14, D8, D2, D1,
  ↪ D7};

// Puntero
int* ptr = leds;

for (i = 0; i < 9; i++) {
  digitalWrite(*(ptr+i), HIGH);
  delay(500);
  digitalWrite(*(ptr+i), LOW);
}

}

```

Es muy evidente la diferencia, por lo cual es importante entender esta funcionalidad, lo que se hizo es crear un puntero **ptr** que guarda todas las direcciones de los pines de salida que se observan en el vector **leds[9]**, y con un bucle recorre cada dirección para escribir un 1 lógico, encenderlo, y después de 500 ms apagarlo, cumpliendo con la secuencia pero usando un retardo bloqueante.

Ahora implementaremos el uso de **retardos no bloqueantes** que básicamente son comparaciones que permite al programa seguir realizando otras instrucciones hasta que se cumpla el tiempo de ejecutar una en específico. Esto se ve en el siguiente código donde primero definimos las variables para cada led, los tiempos que se van a comparar y el puntero que será igual a como lo usamos en el código anterior con la corrección de que no es necesario definirlo en el **loop()** si no que lo hacemos antes de los bucles como definir una constante..

```

#include <Controllino.h> // Librería de controllino

// Retardo no Bloqueante
unsigned long t_previo = 0;
unsigned long t_actual = 0;
unsigned long intervalo = 500; //ms

int i = 0;

int D0 = CONTROLLINO_D0;
int D1 = CONTROLLINO_D1;
int D2 = CONTROLLINO_D2;
int D6 = CONTROLLINO_D6;
int D7 = CONTROLLINO_D7;
int D8 = CONTROLLINO_D8;
int D12 = CONTROLLINO_D12;
int D13 = CONTROLLINO_D13;
int D14 = CONTROLLINO_D14;

int leds[9] = {D0,D6,D12,D13,D14,D8,D2,D1,D7};

// Puntero
int* ptr = leds;

```

El siguiente paso es el **setup()** que nuevamente es el mismo del primer código mostrado donde se define cada pin de los leds como salida por lo que no lo pondré y pasare al **loop()** donde a diferencia del anterior usa la función **millis()** para guardar en una variable el tiempo actual en el que llega a esa instrucción y con un condicional **if** asegurarse que haya pasado los 500 ms realizando una resta del tiempo actual y el tiempo previo, que sería el último tiempo en el que entró al **if**, este tiempo previo es actualizado cada vez que ingresa al **if**, al igual que la

variable **i** que recorre las direcciones guardadas en el puntero, como son 9 leds cuando esta variable llega a 9 debe reiniciarse a 0 para que la secuencia se repita nuevamente.

```
void loop() {
    t_actual = millis();

    digitalWrite(*(ptr+i),HIGH);

    if (t_actual - t_previo >= intervalo){
        digitalWrite(*(ptr+i),LOW);
        t_previo = t_actual;
        i += 1;
        if (i == 9){
            i = 0;
        }
    }
}
```

I-D. Resultados

Como resultado de este trabajo vimos la misma secuencia con los tres códigos, nosotros sabemos que el último es el más eficiente sin embargo de manera visual no se podría notar la diferencia, la secuencia observada es lo descrita previamente comenzando por el led D0 que se observa a continuación.



Figura 2. Encendido led D0

Pasado los 500 ms que obtuvimos comparando la resta de las variables de tiempo actual y previo o con el retardo bloqueante se apaga este led para encenderse el led D6.



Figura 3. Encendido led D6

De la misma forma el siguiente led en prenderse es el D12.



Figura 4. Encendido led D12

Y así sucesivamente logramos observar que se cumplió con la secuencia **D0 → D6 → D12 → D13 → D14 → D8 → D2 → D1 → D7** que fue la que se planteó desde un inicio.

II. PRÁCTICA 3

DISEÑO DE INTERFAZ GRÁFICA PARA EL CONTROL DE SALIDAS EN CONTROLLINO

II-A. Objetivos

- 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)

II-B. Materiales

- Tablero de control con Controllino Mega y HMI integrado.
- Fuente de alimentación del tablero.
- Cable USB tipo A a B 2.0.
- Cable USB tipo A a A.
- PC con Arduino IDE instalado y configurado para Controllino.
- Software Stone Designer GUI.

II-C. Descripción del Funcionamiento

Se tienen dos widget tipo SpinBox en la interfaz final del HMI. Estos cumplen con lo siguiente:

- **sp1**: controla el duty cycle (porcentaje de ciclo de trabajo PWM) del primer led.
- **sp2**: controla el duty cycle (porcentaje de ciclo de trabajo PWM) del segundo led.

Por otra parte, en el tablero se deben configurar dos botones físicos para cumplir con lo siguiente:

- El primer botón físico (**I_16**) controlará el encendido/apagado del primer LED.
- El segundo botón físico (**I_17**) 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).

II-D. Desarrollo

II-D1. Diseño de la interfaz en el HMI:

1. Instalar y abrir el software **STONE Designer GUI**.
2. Crear un nuevo proyecto desde el menú Project, asignando el nombre, carpeta de destino, baudios, dimensiones de la pantalla (800x480 px) y nivel de brillo correspondientes.
3. En la ventana principal *home_page*, hacer clic derecho y seleccionar Add Widget → Label para insertar una etiqueta que muestre información en pantalla.
4. Configurar el tamaño y tipo de fuente del widget **label1**, ajustando su tamaño en la ventana de propiedades para una correcta visualización.
5. Agregar un **widget tipo spin_box** desde *home_page* → Add Widget → *spin_box*, este tendrá como identificador **spin_box1**.
6. Configurar sus propiedades: fuente, límites de valores (0–100), tipo de dato **int**, y alineación del texto.
7. Repetir los puntos **3** y **6** para obtener un **label2** y **spin_box2**.
8. Organizar los widgets en pantalla de acuerdo con la distribución mostrada en la Figura 5.
9. Guardar el proyecto antes de proceder con la carga al HMI.

II-D2. Explicación del Código: El programa permite controlar el brillo de dos LEDs mediante valores de *duty cycle* (0–100 %) enviados desde una HMI (widgets *spin_box* *sp1* y *sp2*), convirtiéndolos a niveles PWM (0–255). Cada LED se habilita o deshabilita con un botón físico (modo *toggle*) usando banderas.

Asignación de pines y variables: Necesarias para el desarrollo de código.

```
const int led = CONTROLLINO_D0; // Salida digital D0
const int led2 = CONTROLLINO_D6; // Salida digital
    ↪ D6
const int boton1 = CONTROLLINO_I16; // Entrada
    ↪ Digital I16
const int boton2 = CONTROLLINO_I17; // Entrada
    ↪ Digital I17
```

PWM para controlar el brillo de dos Leds



Figura 5. Interfaz Final de HMI

```
int pwmValue = 0; // valor convertido (0-255) led 1
int pwmValue2 = 0; // valor convertido (0-255) led 2
float dutyCyclePercent = 0; // valor en porcentaje
    ↪ (0-100) led 1
float dutyCyclePercent2 = 0; // valor en porcentaje
    ↪ (0-100) led 2
int bandera1 = 0; // bandera para boton 1
int bandera2 = 0; // bandera para boton 2
int valor1 = 0; // estado del boton 1
int valor2 = 0; // estado del boton 2
```

■ Salidas PWM: Leds del tablero

- led = CONTROLLINO_D0
- led2 = CONTROLLINO_D6

■ Entradas: Botones del tablero

- boton1 = CONTROLLINO_I16
- boton2 = CONTROLLINO_I17

■ Estados:

- dutyCyclePercent y dutyCyclePercent2: toman valores de 0–100 % que definen el Duty Cycle.
- pwmValue y pwmValue2: toman valores de 0–100 % que definen los valores de encendido de leds.
- bandera1 y bandera2: toman valores de 0 o 1, que funcionan como banderas.
- valor1 y valor2: toman valores de 0 o 1, que guardan el estado de las banderas.

setup() Inicializa variables y estados antes del bucle principal.

```
Serial.begin(115200); // Comunicacion serial con
    ↪ el PC
Serial2.begin(115200); // Comunicacion serial con
    ↪ el HMI
pinMode(led, OUTPUT); // led1 como salida
pinMode(led2, OUTPUT); // led2 como salida
pinMode(boton1, INPUT); // boton 1 como entrada
pinMode(boton2, INPUT); // boton 2 como entrada
HMI_init(); // Inicializa el sistema de colas para
    ↪ las respuestas el HMI
Stone_HMI_Set_Value(spin_box, sp1, NULL, 0); //
    ↪ Pone en 0 el valor del spin box en el HMI.
Stone_HMI_Set_Value(spin_box, sp2, NULL, 0); //
    ↪ Pone en 0 el valor del spin box en el HMI.
```

■ Configura UART de PC (Serial) y HMI (Serial2) a 115200 bps.

- Define direcciones de pines (pinMode).
- HMI_init() y puesta a cero de sp1 y sp2 para un estado inicial conocido.

loop(): lectura y control: Define la lógica principal para el encendido de los leds.

```

void loop() {
dutyCyclePercent=HMI_get_value(spin_box, sp1); // 
    ↪ Obtiene el valor del spin_box1
dutyCyclePercent2=HMI_get_value(spin_box, sp2); // 
    ↪ Obtiene el valor del spin_box2

valor1 = digitalRead(boton1); //lectura digital
    ↪ del boton 1
valor2 = digitalRead(boton2); //lectura digital
    ↪ del boton 2

if (valor1 == HIGH){ //cada que aplasta el boton1
    ↪ se suma 1 a la bandera1
bandera1 += 1; //bandera 1 = 1 es para prender el
    ↪ led 1
if (bandera1 == 2){ //si aplasta de nuevo se pone
    ↪ a valor 0 la bandera1
bandera1 = 0;
} //bandera1 = 0 significa que el led 1 se apaga
}

if (valor2 == HIGH){ //cada que aplasta el boton2
    ↪ se suma 1 a la bandera2
bandera2 += 1; //bandera 2 = 1 es para prender el
    ↪ led 2
if (bandera2 == 2){ //si aplasta de nuevo se pone
    ↪ a valor 0 la bandera2
bandera2 = 0;
} //bandera2 = 0 significa que el led 2 se apaga

// ***** PARA LED 1 *****

if (dutyCyclePercent >= 0 && dutyCyclePercent
    ↪ <=100){
pwmValue = map(dutyCyclePercent, 0, 100, 0, 255);
    ↪ // Mapea el valor de duty cycle 1 en
    ↪ porcentaje a valores de 0 a 255
Serial.print(Duty cycle (%) led 1: );
Serial.print( -> PWM value: );
Serial.println(pwmValue);

if (bandera1 == 1) { // Solo aqui se prende
    analogWrite(led, pwmValue);
}else {
    analogWrite(led, 0); // Si no cumple se apaga
}
}else {
    Serial.println(Ingresa un valor entre 0 y 100.);
}

// ***** PARA LED 2 *****

if (dutyCyclePercent2 >= 0 && dutyCyclePercent2
    ↪ <=100){
pwmValue2 = map(dutyCyclePercent2, 0, 100, 0,
    ↪ 255); // Mapea el valor de duty cycle 1 en
    ↪ porcentaje a valores de 0 a 255
Serial.print(Duty cycle (%) led 2: );
Serial.print( -> PWM value: );
Serial.println(pwmValue2);

if (bandera2 == 1) { // Solo aqui se prende
    analogWrite(led2, pwmValue2);
}else {
    analogWrite(led2, 0); // Si no cumple se apaga
}
}else {
}
}

```

```

        Serial.println(Ingresa un valor entre 0 y 100.);
    }
}

```

1. **Lee HMI:** sp1 y sp2 entregan el *duty* (0–100 %).
2. **Lee botones:** digitalRead en I16 e I17.
3. **Toggle de banderas:** cada lectura en HIGH alterna 0/1 para habilitar el LED correspondiente.
4. **PWM:** si el *duty* está en rango, se mapea (0–100) → (0–255) y se aplica con analogWrite sólo si la bandera = 1; en caso contrario, se apaga (valor 0).

II-E. Resultados

En la figura 6 se puede observar que el led 1 se encuentra encendido al nivel **PWM** correspondiente al **Duty Cycle** colocado en la interfaz del tablero. Se debe mencionar que el led 1 se encuentra encendido solamente porque el usuario presionó el primer botón.

Por esta razón, el segundo led se encuentra apagado, ya que el usuario no presionó el botón 2 correspondiente al led 2.



Figura 6. Led 1 encendido

Por otra parte, en la figura 7 el segundo led se encuentra encendido ya que se sigue la misma lógica presentada anteriormente,



Figura 7. Led 2 encendido

Finalmente, se prueba un nivel diferente de **Duty Cycle** para el segundo led, verificando así su funcionamiento. Referirse a la figura 8.



Figura 8. Segundo Led con Duty Cycle menor

III. 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.

III-A. Objetivos

- Controlar señales PWM
- Implementar un controlador PID para manejar la velocidad del motor
- Obtener datos del duty cicle y velocidad del motor para graficarlos en la pantalla HMI

III-B. Materiales

- Tablero de control con Controllino Mega y HMI integrando
- Entrenador de planta de control EPC
- Fuente de alimentación del tablero
- Cable USB tipo A a B2.0
- Cable USB tipo A a A
- PCconArduino IDE y Stone Designer GUI instalado y configurado
- Software Stone Designer GUI
- 6jumpers macho ahembra largos
- 6jumpers macho amacholargos
- Destornillador plano pequeño

III-C. Desarrollo

III-C1. Hardware: Para comenzar con el desarrollo de esta práctica realizamos las conexiones físicas entre el controllino y la planta EPC, para esto usaremos el puerto **X1** que permite salidas a 5 voltios, voltaje que requiere el motor de la EPC, más específicamente usaremos en pin **D0** del puerto **X1** que genera la señal PWM. Para obtener la velocidad debemos leer los datos provenientes del encoder de la planta en el pin **IN1**, por último conectamos el pin **GND** al pin **GND** del motor DC.

Continuamos encendiendo el tablero accionando los switches con las etiquetas **FUENTE AC/DC**, **CONTROLLINO** y **HMI** para posterior conectar el puerto del controllino a nuestra PC con el uso del cable USB.

La siguiente imagen describe las conexiones que se deben realizar entre el tablero y la planta.

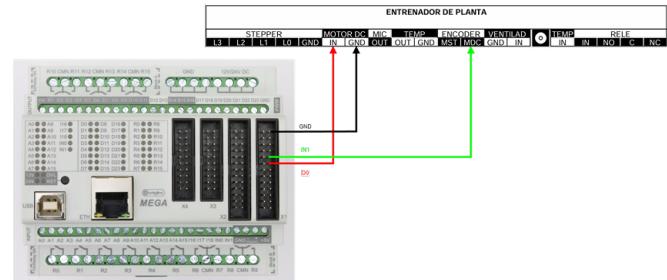


Figura 9. Diagrama de conexiones

III-C2. Interfaz en HMI: Para esta sección diseñamos una interfaz gráfica en la pantalla HMI usando el software Stone Designer GUI, este diseño debe cumplir o tener los siguientes requerimientos para cumplir con el reto:

- Un slider para setear la referencia de la velocidad
- Labels o etiquetas para indicadores y título
- Un chart view para graficar la referencia y velocidad
- 3 cuadros de texto para ingresar los valores o ganancias del controlador PID

De esta forma y con estos lineamientos la interfaz queda de la siguiente manera.

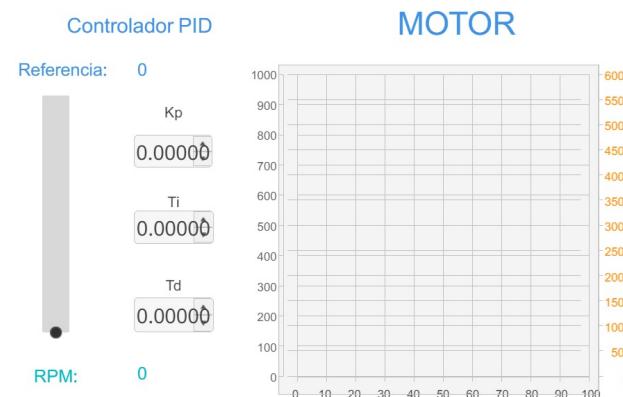


Figura 10. Interfaz a cargar en la pantalla HMI

III-C3. Firmware: En esta sección desarrollamos un controlador PID para la velocidad del motor de la planta EPC, esto lo hacemos usando la ecuación de recurrencias, no es permitido usar ninguna librería externa, a continuación describiré las secciones más importantes del código para facilitar su comprensión.

Comenzamos por definir las librerías a ocupar, ninguna corresponde a PID.

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

```
Stone_HMI_Set_Value(spin_box, spin_box2, NULL,
    ↪ 0.09167);
Stone_HMI_Set_Value(spin_box, spin_box3, NULL,
    ↪ 0.0);
```

El siguiente paso fue definir las variables que usaremos a lo largo del código.

```
volatile float u_actual = 0.0f; // PWM 0..255
volatile float du = 0.0f; // u (solo monitoreo)
volatile float Kp = 0.4567f; // Ganancia
    ↪ proporcional
volatile float Ti = 0.09167f; // Tiempo integral [s]
volatile float Td = 0.0f; // Tiempo derivativo [s]
const float T = 0.05f; // Periodo de muestreo [s]
float e[3] = {0,0,0}; // e[k], e[k-1], e[k-2]
```

Las variables **Kp**, **Ti** y **Td** tienen dichos valores iniciales que fueron los que se determinaron en prácticas previas cuando calibrámos un controlador PID en LabView con el Ing. Galo, sin embargo estos valores pueden ser actualizados en cualquier momento desde la pantalla HMI.

A continuación definimos las variables requeridas para el motor y el encoder, que permitirán medir y graficar su velocidad.

```
// ===== Motor / HMI =====
const int pin_motor = CONTROLLINO_D0; // Asegurate
    ↪ que es PWM en tu modelo
volatile int ref = 0; // Setpoint
char label_ref[10];
char label_rpm[10];

// ===== Encoder / RPM =====
const int entrada = CONTROLLINO_IN1;
volatile unsigned long conteo_pulsos = 0;
volatile float rpm = 0.0f;
const uint16_t PULSOS POR REV = 36;
const float fs = 1.0f / T;
```

En el **setup()** definimos los pines de entrada y salida correspondientes al encoder y alimentación del motor de la planta EPC.

```
Serial.begin(115200);
Serial2.begin(115200);

pinMode(entrada, INPUT);
pinMode(pin_motor, OUTPUT);
```

Además se realiza la configuración inicial de los widgets de la pantalla HMI Stone. Esta etapa garantiza que todos los controles comiencen en un estado conocido antes de iniciar el funcionamiento del sistema de control, para nuestro caso el slider como los 3 items que vamos a graficar inicializamos con 0.

```
// ===== HMI =====
Stone_HMI_Set_Value(slider, slider1, NULL, 0);
STONE_push_series(line_series, line_series1, 0);
STONE_push_series(line_series, line_series2, 0);
STONE_push_series(line_series, line_series3, 0);
```

De igual forma los valores que vemos en los spin box para las ganancias del PID son inicializados con los valores que mencioné y explique previamente.

```
// spin_box1=Kp, spin_box2=Ti [s], spin_box3=Td [s]
Stone_HMI_Set_Value(spin_box, spin_box1, NULL,
    ↪ 0.4567);
```

Lo siguiente es configurar la interrupción externa que se ejecuta cada vez que las ranuras del motor atraviesan la señal del encoder, este señal es la recibimos en el pin **IN1**, cuando se activa la interrupción llama a una función que se encarga de sumar uno a una variable cada vez que ingresa a la función.

```
// Encoder
attachInterrupt(digitalPinToInterrupt(entrada),
    ↪ contarPulso, FALLING);
```

El siguiente bloque de código configura el Timer1 del Controllino para generar una interrupción periódica cada 50 ms, la cual se utiliza como periodo de muestreo del controlador PID y del cálculo de RPM.

```
// ===== Timer1: 50 ms =====
noInterrupts();
TCCR1A = 0;
TCCR1B = 0;
TCCR1B |= B00000100; // prescaler /256 16 s/tick
TIMSK1 |= B00000010; // enable compare A
OCR1A = 3125; // 3125 * 16 s 0.05 s
interrupts();

HMI_init();
```

Primero se deshabilitan las interrupciones globales para evitar modificaciones mientras se ajustan los registros. Luego se limpian **TCCR1A** y **TCCR1B**, y se selecciona un prescaler de 256, lo que produce un incremento del contador cada 16 µs. El registro **OCR1A** se fija en 3125, cantidad de ticks necesaria para alcanzar los 50 ms. Se habilita la interrupción por comparación mediante **TIMSK1**, y finalmente se reactivan las interrupciones globales.

También hacemos un llamado a la función **HMI_init()** que se encarga de inicializar la pantalla HMI.

Eso sería toda la configuración del **setup()**, continuamos a escribir el código en el **loop()**, comenzamos por leer los datos ingresados desde la pantalla HMI cada 10 ms.

```
if (millis() - t_previol >= 10) {
    // Lee UI a variables locales primero
    float kp_ui = HMI_get_value(spin_box, spin_box1);
    float ti_ui = HMI_get_value(spin_box, spin_box2);
    float td_ui = HMI_get_value(spin_box, spin_box3);
    int ref_ui = HMI_get_value(slider, slider1);

    // Protege la actualización (sección crítica)
    noInterrupts();
    Kp = kp_ui;
    Ti = ti_ui; // si el spin box puede dar 0, el ISR
        ↪ se proteger
    Td = td_ui;
    ref = ref_ui;
    interrupts();

    t_previol = millis();
}
```

Continuación del condicional anterior tenemos otro condicional que permite ejecutar nuevamente una sección del código cada 100 ms, es decir, tenemos un retardo no bloqueante ya que mientras no se cumpla el tiempo requerido el resto

del código puede ejecutarse con total normalidad. Este otro condicional es el siguiente.

```
if (millis() - t_previo >= 100) {
    t_previo = millis();
    // Copias locales para imprimir sin jitter
    float rpm_local; int ref_local; float u_local;
    noInterrupts();
    rpm_local = rpm; ref_local = ref; u_local =
        ↪ u_actual;
    interrupts();

    dtostrf((float)ref_local, 7, 2, label_ref);
    dtostrf(rpm_local, 7, 2, label_rpm);

    Stone_HMI_Set_Text(label,label2,label_ref); //
        ↪ referencia
    Stone_HMI_Set_Text(label,label4,label_rpm); //
        ↪ rpm

    STONE_push_series(line_series, line_series1,
        ↪ ref_local);
    STONE_push_series(line_series, line_series2,
        ↪ rpm_local);
    STONE_push_series(line_series, line_series3, (int
        ↪ )u_actual);
}
```

Se encarga de enviar información actualizada a la pantalla HMI, como la referencia, las RPM medidas y la señal de control PWM. Primero copia estas variables a locales dentro de una sección protegida para evitar interferencias con la interrupción del PID. Luego convierte los valores numéricos a texto usando **dtostrf()** para mostrarlos correctamente en las etiquetas de la pantalla. Finalmente, envía los valores a las gráficas de la HMI mediante **STONE_push_series**.

Por último definimos el controlador haciendo uso de la ecuación de recurrencias que se encuentra dentro de la interrupción a la que se accede cada 50 ms.

```
// ===== ISR: muestreo y PID incremental =====
ISR(TIMER1_COMPA_vect) {
    // Si no usas CTC real, resetea contador:
    TCNT1 = 0;

    // RPM en la ventana T
    rpm = ( (float)conteo_pulsos * 60.0f * fs ) / (
        ↪ float)PULSOS POR_REV;

    // Desplazar errores
    e[2] = e[1];
    e[1] = e[0];
    e[0] = (float)ref - rpm;

    // Copias locales (evita leer compartidos varias
        ↪ veces)
    float Kp_l = Kp;
    float Ti_l = Ti;
    float Td_l = Td;

    // u = Kp[(e0e1) + (T/Ti)e0 + (Td/T)(e02e1+e2)]
    float du_loc = 0.0f;
    du_loc += Kp_l * (e[0] - e[1]); // P
    if (Ti_l > 0.0f) du_loc += Kp_l * (T / Ti_l) * e
        ↪ [0]; // I (solo si Ti>0)
    if (Td_l > 0.0f) du_loc += Kp_l * (Td_l / T) * (e
        ↪ [0] - 2.0f*e[1] + e[2]); // D

    // Acumular y limitar
    float u_test = u_actual + du_loc;
    if (u_test > 255.0f) u_test = 255.0f;
    if (u_test < 0.0f) u_test = 0.0f;
```

```
u_actual = u_test;
du = du_loc;

Serial.println(Kp);
Serial.println(Ti);
Serial.println(Td);

analogWrite(pin_motor, (int)u_actual);

// Mapeo de 05 V a 0255 (PWM)
//int pwm_out = map((int)(u_actual * 100), 0, 500,
    ↪ 0, 255);
//analogWrite(pin_motor, pwm_out);

conteo_pulsos = 0; // Reinicia conteo para la
    ↪ proxima ventana T
}
```

En primer lugar, calcula las RPM del motor a partir del número de pulsos capturados por el encoder durante la ventana de muestreo. Luego actualiza el vector de errores y aplica la ecuación del PID incremental, utilizando copias locales de K_p , T_i y T_d para evitar inconsistencias durante la ejecución. El resultado es un incremento ΔU que se suma a la señal de control previa y se limita al rango válido del PWM (0–255).

Finalmente, el valor de control actualizado se envía al motor mediante **analogWrite()**, y el contador de pulsos se reinicia para la siguiente iteración. Esta ISR garantiza que el controlador opere con un periodo exacto, independiente del resto del programa.

Es importante recalcar que la ecuación de recurrencias presenta condicionales como una forma de corregir errores ya que en versiones anteriores del código el PID no presentaba ninguna funcionalidad, por eso se añadió la condición para que tome en cuenta los factores que se ven alterados por T_i y T_d solo cuando estos dos son diferentes de 0.

III-D. Resultados

Como resultado de esta práctica veremos las gráficas obtenidas según los valores proporcionados a K_p , T_i y T_d , comenzamos por asignar los valores encontrados con el método de ganancia máxima donde

$$K_p = 0,55$$

$$T_i = 0,4167$$

$$T_d = 0$$

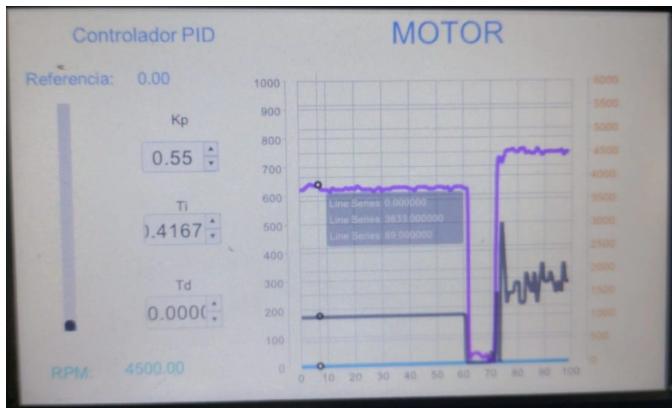


Figura 11. Ingreso de valores

Después de un tiempo observamos que estos datos no sintonizan el controlador de forma correcta ya que no es capaz de seguir la referencia (azul) a pesar de que el PWM si cambia a lo largo del tiempo, pero no produce ningún cambio significativo en la velocidad del motor como lo vemos en la siguiente imagen.

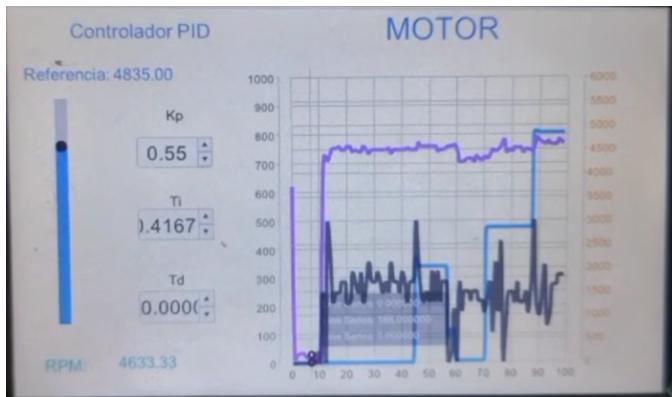


Figura 12. Gráfica resultante

Continuamos ahora probando con los datos que usamos para sintonizar el controlador PID en LabView en la práctica con el Ing. Galo, estos valores son

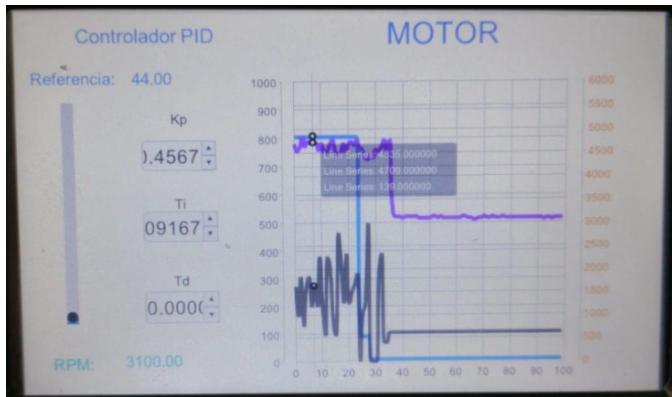


Figura 13. Ingreso de valores

$$K_p = 0,4567$$

$$T_i = 0,09167$$

$$T_d = 0$$

Con estos valores ingresados desde el HMI observamos un comportamiento aún peor que el anterior, pues estos valores de tiempo y ganancia no producen ningún cambio en el PWM lo que tampoco produce ningún cambio en la velocidad a pesar de cambiar la referencia, es decir, no está sintonizado el PID y por ende no es capaz de seguir ninguna referencia.

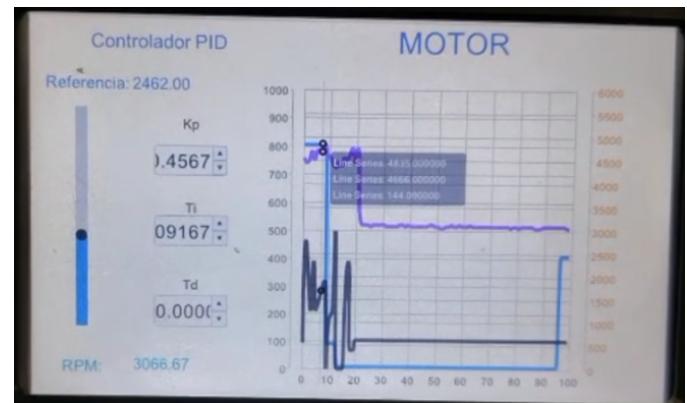


Figura 14. Gráfica resultante

Por último lo que nos queda es sintonizarlo de manera manual, es decir, asignando valores acorde a un criterio técnico hasta encontrar algunos que produzcan una salida aceptable, un seguimiento sin un sobre impulso excesivo, mostrando estabilidad (sin oscilaciones) y un error en estado estacionario inferior al 5 %.

Para nuestro caso esos valores son:

$$K_p = 0,1$$

$$T_i = 0,1$$

$$T_d = 0$$

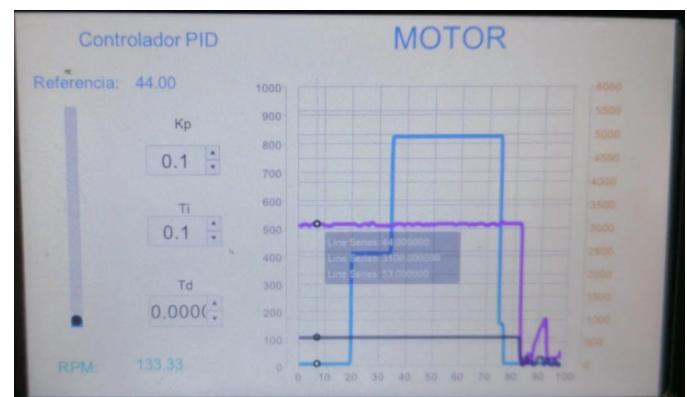


Figura 15. Ingreso de valores

La gráfica que obtenemos con esto es la siguiente, donde vemos claramente como las rpm del motor alcanzan la referencia en un tiempo muy corto gracias a que también se ajusta el PWM que lo alimenta, no vemos ningún sobre impulso excesivo por lo que llegamos a decir que estos valores de K_p , T_i y T_d sintonizan de manera correcta el controlador PID para nuestra placa en específico, cumpliendo con los requisitos planteados al inicio de esta práctica.

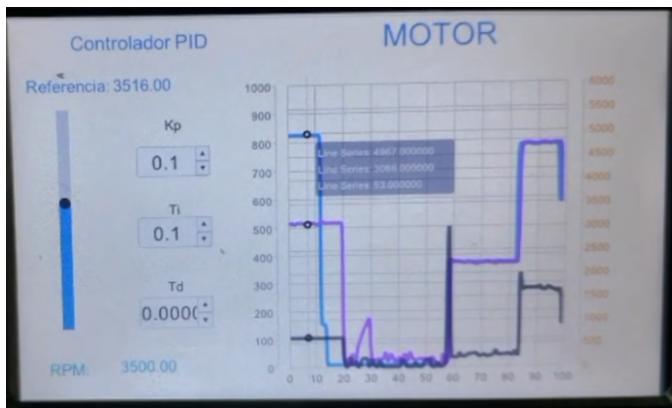


Figura 16. Gráfica resultante

IV. CONCLUSIONES

- Se logró establecer la comunicación entre el Controllino Mega y la HMI Stone, permitiendo controlar el brillo de los LEDs mediante la modulación PWM.
- El uso de widgets *spin_box* facilitó la interacción entre el usuario y el sistema, demostrando la eficacia del control gráfico en entornos de automatización.
- La integración de botones físicos permitió implementar un sistema de control combinado (manual y digital), mejorando la comprensión de los principios de entrada/salida en PLC.
- Se evidenció la importancia del procesamiento correcto de datos del HMI y del manejo de estados mediante banderas para evitar errores lógicos.
- La práctica permitió reforzar conocimientos sobre comunicación serial, control de PWM y diseño de interfaces HMI aplicadas a sistemas embebidos.

Podemos añadir a lo antes mencionado que al término de este trabajo pudimos realizar el control digital de un motor DC mediante un PID incremental ejecutado en una interrupción periódica de 50 ms, garantizando un muestreo estable y consistente. La medición de velocidad con encoder resultó precisa gracias al conteo por interrupciones y al cálculo de RPM por ventana. Además, el uso de la HMI Stone facilitó el ajuste en tiempo real de los parámetros, permitiendo observar de manera inmediata su efecto sobre la respuesta del sistema.

V. ANEXOS

V-A. Link para Repositorio de la Práctica

<https://github.com/MateoDutan/ControlDigital>