Uso de tableros Control Digital

Mauro Martín Bravo Pintado
Escuela de Telecomunicaciones
Universidad De Cuenca
martin.bravo@ucuenca.edu.ec

Miguel Angel Calle Duran
Escuela de Telecomunicaciones
Universidad De Cuenca
miguel.calle@ucuenca.edu.ec

***Abstract*—This document presents a practical guide for the implementation of automation and control systems using PLC training boards that integrate Controllino Mega, ESP32 PLC 14, and a STONE HMI. The guide is structured into three main practices: the first focuses on controlling digital outputs through sequential LED activation; the second introduces finite state machines (FSM) for managing LED patterns and simulating a traffic light system; and the third involves the design of a graphical interface to control PWM signals via the HMI. Each practice is developed within the Arduino IDE environment, using predefined libraries for efficient interaction with the hardware. The document also includes step-by-step instructions, diagrams, and visual evidence of the implementations to support learning and verification of system behavior.**

***Keywords*—Control, PID, Motor.**

I. Práctica 1: Manejo de Salidas Digitales con Controllino Mega

A. Objetivo

Comprender el uso de las variables predefinidas de la librería `Controllino.h` para generar una secuencia de encendido de 9 LEDs conectados a las salidas digitales.

B. Materiales requeridos

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

C. Reto

- Controlar una matriz 3×3 de LEDs en *espiral*: $1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 9 \rightarrow 8 \rightarrow 7 \rightarrow 4 \rightarrow 5 \rightarrow$ (repetir).
- Cada LED se enciende por 500 ms y luego se apaga.
- Usar punteros para recorrer la secuencia.
- Usar retardos no bloqueantes (`millis()`).

D. Explicación de la implementación de la lógica de encendido

La lógica consiste en definir un vector que almacena las variables predefinidas que representan los LEDs del tablero, es decir, las variables CONTROLLINO_Dx de la librería Controllino.h. Se almacena la secuencia a mostrar en el vector, por lo tanto, el orden de la secuencia define los elementos del mismo. El vector se define de tamaño 10, ya que la secuencia comienza apagando el último LED y encendiendo el primero. Para ello, el vector inicia con el último LED seguido por toda la secuencia restante.

Se define un tiempo de retardo no bloqueante de 500ms y una variable para el tiempo anterior, inicializada en 0. Por último, se declara un contador para controlar la posición del vector.

En la función setup, mediante un bucle for, se configuran los pines de los LEDs como salidas, recorriendo todas las posiciones del vector.

En la función loop, se obtiene el tiempo actual con la función millis() y se evalúa si han transcurrido los 500ms. Si se cumple la condición, se apaga el LED correspondiente a la posición actual y se enciende el siguiente LED. Luego, se incrementa el contador en una unidad para continuar recorriendo el vector.

Cuando el contador deja de ser menor que 9, se reinicia a 0, se apaga el LED en la posición 0 y se enciende el siguiente. Además, se fuerza el valor del contador a 1 para corregir un desfase que ocurría debido a la lógica implementada. Con esta corrección, la secuencia vuelve a iniciar correctamente y de manera fluida.

Fuera del condicional del retardo no bloqueante, se actualiza la variable de tiempo anterior con el tiempo actual para poder aplicar el retardo cada 500ms.

E. Implementación

```
//Practica 1: Encender todos los leds del
    tablero secuencialmente
#include <Controllino.h> // Librer a de
    controllino

//Vector que contiene la secuencia deseada
int leds[10] = {CONTROLLINO_D7,CONTROLLINO_D0,
     CONTROLLINO_D6, CONTROLLINO_D12,
    CONTROLLINO_D13, CONTROLLINO_D14,
    CONTROLLINO_D8, CONTROLLINO_D2,
    CONTROLLINO_D1,CONTROLLINO_D7};

//Variable que almacena el tiempo del retardo
    no bloqueante
int tiempo_ms=500;

//Variable que almacena el tiempo anterior
unsigned long t_anterior=0;

//Contador para recorrer las posiciones del
    vector
int contador=0;

//Funcion de configuracion
void setup() {
```

```
    //Recorrer el vector con las varialbs de
        los LEDs
    for (int i = 0; i < 10; i++) {
         //Establecer las mismas como salidas
        pinMode(leds[i], OUTPUT);
    }
}

//Funcion recurrente
void loop() {
    //Almacenar el tiempo actual
    unsigned long t_actual=millis();
    //Retardo no bloqueante, se ejecuta cada
        500ms
    //Comparar diferencia del tiempo actual y
        anterior
    //Para entrar a la condicion
    if (t_actual - t_anterior >= tiempo_ms){
        //Verificar si el contador es menor a
            9
        if (contador<9){
            //Apagar el LED en la poscion
                actual
            digitalWrite(leds[contador], LOW);
            //Encender el LED en la posicion
                siguiente
            digitalWrite(leds[contador+1],
                HIGH);
            //Incrementar contador en 1
            contador++;
            //Si no cuple la condicion de
                menor a 9
        }else{
            //Reiniciar contador para poder
                repetir
            contador =0;
            //Apagar el LED en la poscion
                actual
            digitalWrite(leds[contador], LOW);
            //Encender el LED en la posicion
                siguiente
            digitalWrite(leds[contador+1],
                HIGH);
            //Forzar a que comeince la
                secuiane desde el led
                siguiente al reiniciar
            contador=1;
        }
    }
    //Actualiza el tiempo anterior con el
        actual
    t_anterior=t_actual;
  }
}
```

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

A. Objetivo

Uso básico del PLC Controllino Mega, mediante el encendido y apagado de un foco LED por dos pulsantes en el tablero respectivamente, luego en activdades reto controlar secuencias mediante tres pulsantes y al finalizar con el ultimo reto, una maquina de estados la cual controle un semaforo para dos vías.

B. Materiales requeridos

- 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.

C. Reto

1) ***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.

Para la resolución de este reto se ha implementado el siguiente código:

En las siguientes lineas de código se ha declarado la libreria necesaria, ademas de los botones que se usaran para el control de la secuencia en espiral, invertida y un botón que reinicie todo. Ademas declaramos dos vectores uno con la secuencia en espiral de izquierda a derecha y otro con la secuencia de derecha a izquierda. También se declara contadores para cada secuencia y también una bandera que nos ayudara con la transición entre cada secuencia.

```
#include <Controllino.h>

// Botones de control
const int boton_secuencia1 = CONTROLLINO_I16;
const int boton_secuencia2 = CONTROLLINO_I17;
const int boton_reinicio   = CONTROLLINO_I18;

// vectores con secuencias de LEDs para
    secuencias 1 y 2
int leds1[10] = {
  CONTROLLINO_D7, CONTROLLINO_D0,
      CONTROLLINO_D6,
  CONTROLLINO_D12, CONTROLLINO_D13,
      CONTROLLINO_D14,
  CONTROLLINO_D8, CONTROLLINO_D2,
      CONTROLLINO_D1, CONTROLLINO_D7
};

int leds2[10] = {
  CONTROLLINO_D0, CONTROLLINO_D7,
      CONTROLLINO_D1,
  CONTROLLINO_D2, CONTROLLINO_D8,
      CONTROLLINO_D14,
  CONTROLLINO_D13, CONTROLLINO_D12,
      CONTROLLINO_D6, CONTROLLINO_D0
};
const int tiempo_ms = 500;
unsigned long t_anterior = 0;
int contador = 0;
int bandera = 2;  // 1 = secuencia 1, 0 =
    secuencia 2, 2 = apagado
```

Listing 1. Declaracion de variables y librerias

Luego de esto declaramos las salidas, entradas e inicializamos los leds en estado LOW.

```
void setup() {
  // Declaracion de pines como salida y
      apagarlos
  for (int i = 0; i < 10; i++) {
    pinMode(leds1[i], OUTPUT);
    pinMode(leds2[i], OUTPUT);
    digitalWrite(leds1[i], LOW);
    digitalWrite(leds2[i], LOW);
  }

  // Configurar botones como entrada
  pinMode(boton_secuencia1, INPUT);
  pinMode(boton_secuencia2, INPUT);
  pinMode(boton_reinicio, INPUT);
}
```

Listing 2. Declaracion de variables de entrada, salida y estado de inicio

Luego revisamos el estado de cada pulsante para ver que secuencia seguir o si se debe estar apagado todos los leds, mediante el uso de retardo no bloqueante y de punteros.

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

  // Cambiar bandera segun boton presionado
  if (digitalRead(boton_secuencia1) == HIGH) {
    bandera = 1;
    contador = 0;
  }
  if (digitalRead(boton_secuencia2) == HIGH) {
    bandera = 0;
    contador = 0;
  }
  if (digitalRead(boton_reinicio) == HIGH) {
    bandera = 2;
    contador = 0;

    // Apagar todos los LEDs de ambas
        secuencias
    for (int i = 0; i < 10; i++) {
      digitalWrite(leds1[i], LOW);
      digitalWrite(leds2[i], LOW);
    }
  }

  // Temporizacion no bloqueante
  if (t_actual - t_anterior >= tiempo_ms) {
    t_anterior = t_actual;

    // Apagar todos los LEDs primero
    for (int i = 0; i < 10; i++) {
      digitalWrite(leds1[i], LOW);
      digitalWrite(leds2[i], LOW);
    }

    // Encender uno solo dependiendo de la
        secuencia activa
    if (bandera == 1) {
      digitalWrite(leds1[contador], HIGH);
    } else if (bandera == 0) {
      digitalWrite(leds2[contador], HIGH);
    }

    // Avanzar al siguiente LED
    contador = (contador + 1) % 10;
  }

  delay(10); // Antirebote b sico
}
```

Listing 3. Declaracion de variables de entrada, salida y estado de inicio

*2) **Parte B**:*

- Diseñar un FSM para dos semáforos que impida luz verde simultánea.
- Usar `enum`, estructuras y retardos no bloqueantes.

Para implementar el sistema que controle dos semáforos en una intersección perpendicular se realiza el diagrama de estados el cual permita tener en consideración los diferentes estados que se va a tener cuando esta en rojo, verde, amarillo o en algun caso fuera de estos estados que por defecto vaya al de intermitente ya sea por un corte de luz, falla en algún sistema eléctrico cercano, lo cual este estado lo llamaremos estado de inicio como se puede ver en la Figura 1. Una vez que tenemos

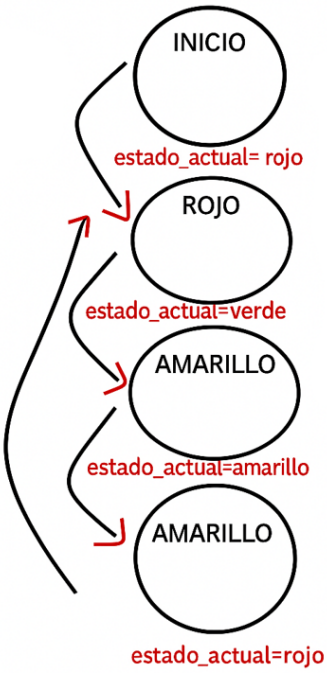

Fig. 1. Diagrama de estados del sistema de control para dos semáforos

claro los estados posibles de nuestro sistema procedemos a realizar la codificación definiendo los leds del Controllino, su respectivo color, ademas de variables de encendido y apagado,

También se define los cuatro estados descritos anteriormente en nuestro diagrama de estados, como a su vez definimos el estado actual como `INICIO`, y se inicializa en 0 las variables que permitirán controlar los tiempos para el paso del siguiente estado.

```
#define LED_ROJO CONTROLLINO_D0
#define LED_AMARILLO CONTROLLINO_D1
```

```
#define LED_VERDE CONTROLLINO_D2

#define LED_ROJO2 CONTROLLINO_D6
#define LED_AMARILLO2 CONTROLLINO_D7
#define LED_VERDE2 CONTROLLINO_D8

#define encendido 1
#define apagado 0

enum Estado {
  INICIO,
  ROJO,
  VERDE,
  AMARILLO
};

Estado estado_actual = INICIO;
unsigned long tiempo_actual = 0;
unsigned long tiempo_anterior = 0;
unsigned long duracion_estado = 0;

void setup() {
  pinMode(LED_rojo1, OUTPUT);
  pinMode(LED_amarillo1, OUTPUT);
  pinMode(LED_verde1, OUTPUT);
  pinMode(LED_rojo2, OUTPUT);
  pinMode(LED_amarillo2, OUTPUT);
  pinMode(LED_verde2, OUTPUT);

  // Iniciar con todos los LEDs apagados
  apagarTodo();
}
```

Listing 4. Declaracion de variables de entrada, salida y estado de inicio

Luego de esto, se usa la función `milis()`, la cual devuelve el número de milisegundos transcurridos desde que el programa comenzó a ejecutarse permitiendo hacer retardos no bloqueantes en base la definición de las variables declaradas anteriormente. Para luego dar paso a los diferentes casos descritos a continuación el cual nos da los diferentes estados dando un tiempo de encendido a cada led del tablero simulando al intersección perpendicular sin que tengamos leds verdes encendidos simultáneamente, lo cual para el apagado de los leds que no estén en uso usamos una función definida al final de este código.

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

  switch (estado_actual) {

    case INICIO:
      // Parpadeo amarillo1 y rojo2 por 2.5 s
      if (tiempo_actual - tiempo_anterior >=
          500) {
        static bool parpadeo = false;
        parpadeo = !parpadeo;
        digitalWrite(LED_amarillo1, parpadeo);
        digitalWrite(LED_rojo2, parpadeo);
        tiempo_anterior = tiempo_actual;
        duracion_estado += 500;
      }

      if (duracion_estado >= 2500) {
        apagarTodo();
        estado_actual = ROJO;
        tiempo_anterior = tiempo_actual;
        duracion_estado = 0;
      }
      break;

    case ROJO:
      digitalWrite(LED_rojo1, encendido);
      digitalWrite(LED_verde2, encendido);

      if (tiempo_actual - tiempo_anterior >=
          3000) {
        apagarTodo();
        estado_actual = VERDE;
        tiempo_anterior = tiempo_actual;
      }
      break;

    case VERDE:
      digitalWrite(LED_verde1, encendido);
      digitalWrite(LED_amarillo2, encendido);

      if (tiempo_actual - tiempo_anterior >=
          2000) {
        apagarTodo();
        estado_actual = AMARILLO;
        tiempo_anterior = tiempo_actual;
      }
      break;

    case AMARILLO:
      digitalWrite(LED_amarillo1, encendido);
      digitalWrite(LED_rojo2, encendido);

      if (tiempo_actual - tiempo_anterior >=
          2000) {
        apagarTodo();
        estado_actual = ROJO;
        tiempo_anterior = tiempo_actual;
      }
      break;

    default:
      estado_actual = INICIO;
      tiempo_anterior = tiempo_actual;
      break;
  }
}
//  Funcion para apagar todos los LEDs
void apagarTodo() {
  digitalWrite(LED_rojo1, LOW);
  digitalWrite(LED_amarillo1, LOW);
  digitalWrite(LED_verde1, LOW);
  digitalWrite(LED_rojo2, LOW);
  digitalWrite(LED_amarillo2, LOW);
  digitalWrite(LED_verde2, LOW);
}
```

Listing 5. Transicion de estados y funcion apagar de leds que no se usan

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

A. Objetivo

Controlar el brillo de un LED mediante PWM usando un valor de duty cycle enviado desde un HMI.

B. Materiales

- Tablero con Controllino y HMI.
- Fuente de alimentación.
- Cables USB A-A y A-B.
- PC con Arduino IDE y Stone Designer GUI.

C. Procedimiento

- Crear y cargar interfaz en HMI con un `spin box` para el duty cycle.
- Conectar y programar el Controllino para recibir datos y ajustar PWM.

D. Reto

- Agregar control de dos LEDs con dos `spin boxes`.
- Usar botones físicos para encender/apagar cada LED de forma independiente.

En el software de *Stone* se colocan dos `SpinBox`, cada uno destinado a controlar el brillo de los LED 1 y LED 2. Los valores de estos `SpinBox` están limitados en un rango de 0 a 100, lo que representa el porcentaje de brillo.

En el `Controllino`, estos valores se transforman a un rango de 0 a 255 para su correcta interpretación por el dispositivo. Los nombres de cada `SpinBox` se definen de forma que puedan ser identificados fácilmente en el código para referirse a cada uno de ellos.

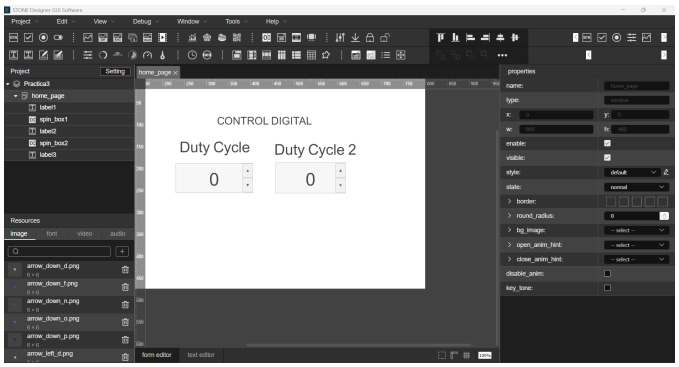

Fig. 2. Diseño del HMI.

Para el control de las señales PWM y la interfaz gráfica, es necesario usar las siguientes librerías, con el fin de implementar el Controllino y la pantalla (HMI):

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

Listing 6. Declaracion de variables de entrada, salida y estado de inicio

Se declaran variables para manejar las salidas digitales `led` y `led2` en los pines `D0` y `D1` del controlador; estos son los LEDs cuyo brillo se desea controlar. Para los botones, se definen las variables `boton_led1` y `boton_led2`. Las señales PWM se gestionan mediante las variables `pwmValue` y `pwmValue2`, mientras que los valores de *duty cycle* se almacenan en `dutyCyclePercent` y `dutyCyclePercent2`. Para controlar correctamente los botones y evitar errores al presionarlos, se crean las variables `estadoBoton1Anterior` y `estadoBoton2Anterior`, que permiten consultar el estado previo de cada botón.

```
const int led              = CONTROLLINO_D0;
    // Salida digital D0
const int led2              = CONTROLLINO_D1;
    // Salida digital D1
const int boton_led1 = CONTROLLINO_I16;    //
    Entrada I16 del controllino e/a
const int boton_led2    = CONTROLLINO_I17;    //
     Entrada I17 del controllino e/a
int       pwmValue          = 0;
    // valor convertido (0-255)
int       pwmValue2          = 0;
    // valor convertido (0-255)
float     dutyCyclePercent = 0;
    // valor en porcentaje (0-100)
float     dutyCyclePercent2 = 0;
    // valor en porcentaje (0-100)
int estado_ac=0;
int estado_ant=0;
int bandera=0;
int bandera2=0;

// Estados anteriores de los botones
int estadoBoton1Anterior = LOW;
int estadoBoton2Anterior = LOW;
```

Listing 7. Declaracion de variables de entrada, salida y estado de inicio

En la función `void setup`, se establece la comunicación serial tanto para el PC como para el HMI. Además, se configuran como entradas los botones `boton_led1` y `boton_led2`. A continuación, se inicia el HMI mediante la función `HMI_init()` y se establecen los valores de los *spin box* en cero utilizando la función `Stone_HMI_Set_Value`.

```
void setup() {
  Serial.begin(115200);    // Comunicaci n
      serial con el PC
  Serial2.begin(115200);  // Comunicaci n
      serial con el HMI
  pinMode(led, OUTPUT);    // led como salida
  pinMode(led2, OUTPUT);    // led como salida
  pinMode(boton_led1, INPUT);  //
      boton_encendido/apagado como entrada
  pinMode(boton_led2, INPUT);  //
      boton_apagado/apagado como entrada
  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.

}
```

Listing 8. Configuración de entradas y salidas

Dentro de la función `loop`, se obtiene el valor de los *spin box* mediante la función `HMI_get_value()`. Después, se lee el estado del botón y, mediante una condición, se verifica si el estado del botón es alto (es decir, si se ha presionado) y si el estado anterior fue bajo. Con esta lógica se detecta la transición del botón de bajo a alto, con el fin de evitar errores cuando se mantiene presionado el botón. Si se cumple esta condición, se suma una unidad a un contador llamado `bandera`, con el que se decide qué acción se tomará. Si el contador supera el valor de 2 se reinicia nuevamente a 1 para el correcto control de la bandera, es importante actualizar el estado del botón.

```
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_box1

    // Lectura y detecci n de flanco para
       bot n 1
  int estadoBoton1 = digitalRead(boton_led1);
  if (estadoBoton1 == HIGH &&
      estadoBoton1Anterior == LOW) {
    bandera++;
    if (bandera > 2) bandera = 1;
  }
  estadoBoton1Anterior = estadoBoton1; //
      actualizar el estado del bot n
```

Listing 9. Lectura del botón 1

De acuerdo al valor de la variable `bandera`, se puede decidir si encender o apagar el LED: cuando `bandera` es igual a 1, el LED se enciende; cuando `bandera` es igual a 2, el LED se apaga. Después de esto, la variable `bandera` debe reiniciarse a cero, ya que al presionar el botón cambia inicialmente a 1.

```
// Control del LED1 seg n bandera
 if (bandera == 1) {
   digitalWrite(led, HIGH);  // Encender LED
 } else if (bandera == 2) {
    digitalWrite(led, LOW);  // APAGAR LED
   bandera = 0;
 }
```

Listing 10. Encender y apagar el led

La misma lógica se aplica para el botón 2:

```
    // Lectura y detecci n de flanco para
       bot n 2
  int estadoBoton2 = digitalRead(boton_led2);
  if (estadoBoton2 == HIGH &&
      estadoBoton2Anterior == LOW) {
    bandera2++;
    if (bandera2 > 2) bandera2 = 1;

  }
  estadoBoton2Anterior = estadoBoton2;

  // Control del LED1 seg n bandera
  if (bandera2 == 1) {
    digitalWrite(led2, HIGH);  // Encender LED
  } else if (bandera2 == 2) {
     digitalWrite(led2, LOW);  // APAGAR LED
    bandera2 = 0;
  }
```

Listing 11. Control del segundo botón

De acuerdo al valor que toma la variable `bandera`, mediante una condición se determina qué LED funcionará. Si `bandera` toma el valor de 1, se obtiene el valor del `PWM`, se mapea del rango 0–100 al rango 0–255 y se escribe en el pin correspondiente al LED 1. De manera similar, si `bandera` toma el valor de 2, se realiza el mismo procedimiento para el LED 2.

```
  if (dutyCyclePercent >= 0 &&
      dutyCyclePercent <=100 && bandera==1){

    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␣(%):␣");
    Serial.print(dutyCyclePercent);
    Serial.print("␣->␣PWM␣value:␣");
    Serial.println(pwmValue);
  }else {
    Serial.println("Ingresa␣un␣valor␣entre␣0␣y
        ␣100␣o␣encienda␣el␣led");
  }
  if (dutyCyclePercent2 >= 0 &&
      dutyCyclePercent2 <=100 && bandera2==1){

    pwmValue2 = map(dutyCyclePercent2, 0, 100,
        0, 255);       // Mapea el valor de
        duty cycle en porcentaje a valores de
        0 a 255
    analogWrite(led2, pwmValue2);
    Serial.print("Duty␣cycle␣(%):␣");
    Serial.print(dutyCyclePercent2);
    Serial.print("␣->␣PWM␣value2:␣");
    Serial.println(pwmValue2);
  }else {
    Serial.println("Ingresa␣un␣valor␣entre␣0␣y
        ␣100␣o␣encienda␣el␣led2");
  }
}
```

Listing 12. Señales PWM

IV. Práctica 4: Adquisición de datos de motor y gráfica

A. Objetivo

Controlar un motor DC con PWM y graficar duty cycle y RPM en el HMI.

B. Materiales

- Tablero con Controllino y HMI.
- EPC y cables necesarios.
- PC con software necesario.

C. Reto

Diseñar e implementar un **controlador PID**, basados en [1]. Que regule automáticamente la velocidad del motor DC del EPC a un valor de referencia definido por el usuario mediante la interfaz HMI.

Requisitos:

- La velocidad del motor debe alcanzar el valor de referencia en el menor tiempo posible, sin sobrepasarlo excesivamente.
- El sistema debe ser estable (sin oscilaciones) y con un error en estado estacionario menor al 5%.
- El sistema debe mostrar en tiempo real:
  - El valor de referencia (setpoint) configurado mediante el slider.
  - La velocidad actual (RPM).
  - Una gráfica que compare la referencia y la velocidad actual.
  - Una gráfica de la señal de control aplicada.

Restricciones:

- El algoritmo PID debe implementarse mediante la ecuación de recurrencias, sin utilizar librerías externas de PID.

Pautas opcionales:

- Incorporar un botón de emergencia en el HMI que apague el motor.
- Permitir el ajuste manual de las constantes K_p, K_i y K_d desde la interfaz gráfica.

D. Procedimiento

- Conectar el motor y encoder al Controllino.
  Se realizan las conexiones especificadas en la práctica. Para ello, el pin GND del Controllino, del puerto X1, se conecta al GND del EPC; el pin D0 del puerto X1 del Controllino se conecta al pin IN del EPC; y el pin IN1 del puerto X1 del Controllino se conecta al MDC ENCODER del EPC, como se muestra en la figura 3.

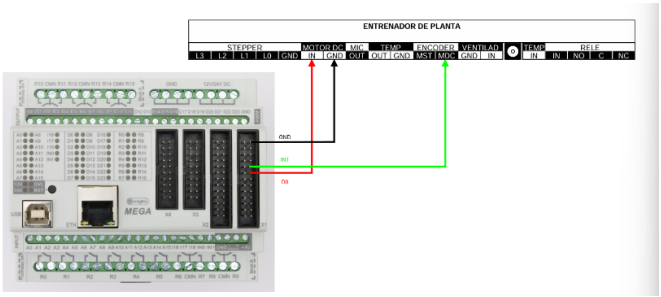

Fig. 3. Conexión del Controllino con el EPC.

- **Configurar interrupciones para conteo de pulsos y cálculo de RPM.**
  El tiempo de la interrupción que deseamos es de 50ms, de la misma manera es el tiempo de muestro seleccionado para la impementación del PID, se considera lo sigiuente: El microcontrolador ATmega2560 trabaja a una frecuencia de:

$$f_{\text{CPU}} = 16\,\text{MHz}$$

Por lo tanto, el periodo de un ciclo de reloj es:

$$T_{\text{CPU}} = \frac{1}{f_{\text{CPU}}} = \frac{1}{16000000} = 62.5\,\text{ns}$$

Con el prescaler de 256:

$$T_{\text{tick}} = T_{\text{CPU}} \times 256 = 62.5\,\text{ns} \times 256 = 16\,\mu\text{s}$$

El número de ticks necesarios para que transcurra en 50ms es:

$$\text{ticks} = \frac{0.05\,\text{s}}{16\,\mu\text{s}} = 3125$$

El **conteo de pulsos**, es el número de flancos contados por el encoder en el intervalo de tiempo considerado (50 ms), se multiplica por 36 que representa los pulsos por revolicio del motor de acuerdo al funcionamiento del EPC y se multiplica por 60 para convertir a minutos.

$$\text{RPM} = \frac{\text{conteo de pulsos} \times 60}{36}$$

- **Diseñar interfaz con slider, labels y chart en HMI.**
  Para la interfaz se utilizó un **deslizador (slider)** mediante el cual se controla la referencia de velocidad, con un valor máximo de **6000**. Las unidades representadas corresponden a **RPM**.
  Se añadieron **tres spin boxes**, que permiten ingresar valores enteros para las constantes **proporcional (Kp)**, **integral (Ki)** y **derivativa (Kd)**. Mediante el código, estos valores enteros son procesados y convertidos a números decimales dividiéndolos entre **100**.
  Los valores de **RPM** se muestran en un **label**, el cual representa las RPM leídas directamente del motor.
  Finalmente, se utiliza un **gráfico** que muestra tres variables: la **referencia**, la **variable a controlar** (RPM medida) y la **señal de control**.
  Para la señal de control se añadió un eje vertical desde la mitad de la gráfica para no sobreponer las curvas y obtener asi una mejor visuaización, la misma tiene valores de **0 a 300**, considerando que la señal tendrá un **offset** de 255; de esta manera se cubre todo el rango de la variable. Por su parte, la señal a controlar y la referencia utilizan otro eje vertical, ubicado a la derecha de la gráfica, que va de **0 a 6000 RPM**. En la figura 4 se presenta la interfaz diseñada.
- **Procedimiento para obtener la función de transferencia del motor con LabVIEW y MATLAB. Se deberá justificar también la elección del tiempo de muestreo programado en el Controllino.**

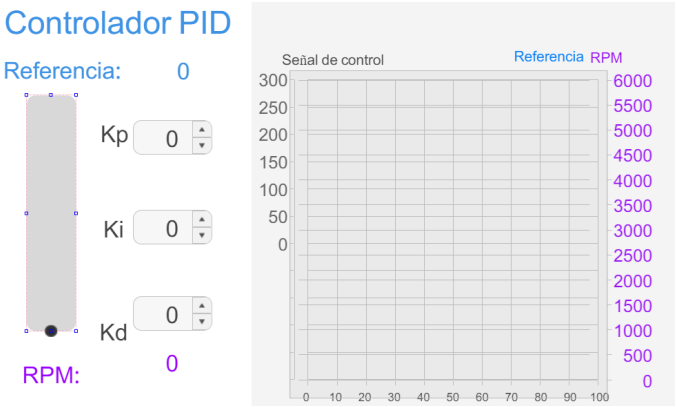

Fig. 4. Conexión del Controllino con el EPC.

Para configurar el entorno de LabView del NI DAQmx vamos a la ventana de diagrama de bloques donde buscamos en funciones la sección **Measurements**, y seleccionamos el DAC, una vez seleccionado tenemos que realizar la configuracion del DAQ Assitant.
Al seleccionar la herramienta se abre una ventana donde se selecciona la opción **Generar señales**, ya que se desea una señal de control de voltaje y se especifica que sea de tipo analógica. En la siguiente ventana se configura dicha señal para con valores máximos y mínimos de 5v y 0v respectivamente, la unidad que es voltios y el modo de generación en **1 Sample (On Demand)**, como se observa en la figura 5

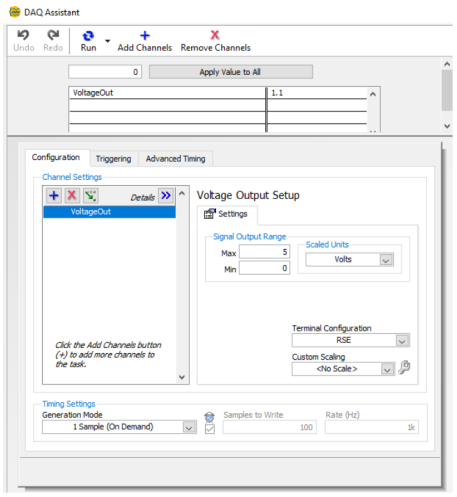

Fig. 5. Configuración de la señal analógica.

Para contar los pulsos que lee el encoder del motor, se añade un DAQ Assistant el cual se configura en la venta seleccionando adquisición de una señal y se selecciona la opción conteo de ingreso (counter input), a continuación se marca la opción conteo de pulsos (Edge count) y seguimos en la siguiente ventana, figura 6

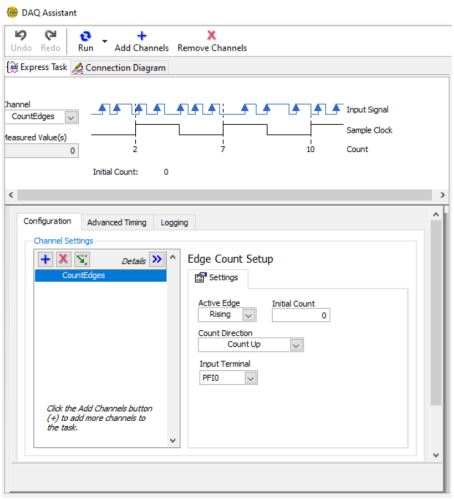

Fig. 6. Configuración de la lectura de los flancos del encoder.

Configuramos el canal en conteo de pulsos en flancos de subida con un conteo inicial de cero, la dirección de conteo de la misma manera se configura hacia arriba y el terminal de ingreso como PFI0 que está seleccionado por defecto.
Una vez configurado el entorno de LabView, se busca manipular la velocidad del motor a través de un control, además de medir y graficar la velocidad en el programa. Aregamos un registro de desplazamiento, un restador y realizamos la conexión para que haga la resta del valor actual con el valor anterior del contador de pulsos, donde tendremos cuantos pulsos se contaron. Este valor es conectado a un divisor y dividido para un valor constante de 36 pulsos por revolución, el cual nos dará el valor de revoluciones y consecuentemente este valor vuelve a ser dividido para 50 (tiempo de muestreo), el cual este valor representa la espera en cada iteración, y después de esto agregamos un multiplicador el cual es por un factor de 60 el cual nos dará un valor de revoluciones por minuto.

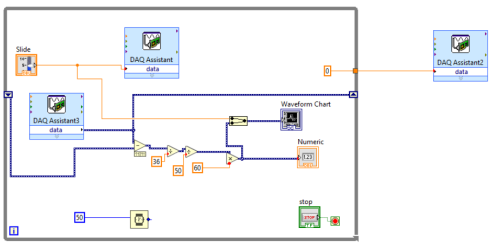

Fig. 7. Diagrama de bloques de control del motor y obtención de datos

Ahora es importante agregar un botón de stop el cual nos permita parar en cualquier momento el while. Y para poder observar de manera gráfica agregamos en la interfaz HMI, un Waveform Chart, conectado a un Merge signals en donde ingresaremos el valor de las revoluciones por

minuto y la velocidad ingresada mediante el slide que controla el voltaje.
Una vez con los datos generados, se genera un archivo de texto el cual contiene el tiempo, el valor de la entrada y el valor de la respuesta del sistema (salida), para el tiempo, se convierte cada iteración multiplicando por 0.05 que representa el tiempo de cada iteración para convertir a segundos y para guardar los valores generados se saca del bucle como un array, el valor de entrada se toma directo de la entrada y de la misma manera se saca como un array, el último valor que es la respuesta del sistema, se toma de la conversión a RPM y se saca como array. Con la herramienta **Build Array** con 3 entradas se conectan las 3 salidas generadas en el punto anterior para generar el archivo de texto con 3 columnas correspondientes al voltaje de ingreso, respuesta del sistema y el tiempo transcurrido, finalmente para almacenar el archivo se usa la herramienta **write delimited** y **filed path** para escribir el archivo en una ruta específica del ordenador, además con un valor booleano en True se transpone los datos para la correcta representación como una tabla.
El archivo generado, es procesado en MATLAB para graficar los mismos, figura 8 y calcular la función de transferencia, figura 9 por el método gráfico.
Considerando la entrada escalón de 4.988V aplicada aproximadamente en t = 0.35 s, la salida que llega a una respuesta estacionaria alrededor de 5.4 V, la constante k es:

$$k = \frac{5.4 - 0}{4.988 - 0} = 1.083$$

El punto de 63.2 % que define el valor de $\theta + \tau$ en 0.55s, el tiempo muerto θ de 0.2s, por lo tanto τ de 0.35, se obtiene la siguiente función de transferencia:

$$G(s) = \frac{k}{\tau s + 1} = \frac{1.083}{0.35s + 1}$$

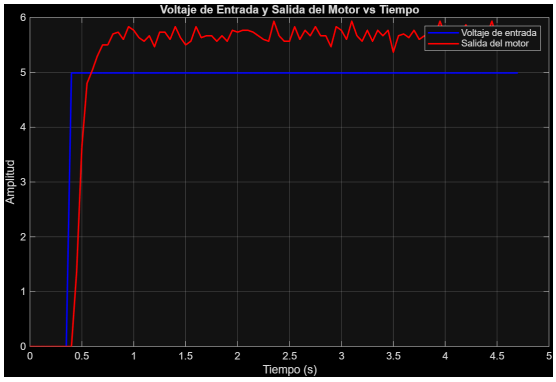

Fig. 8. Respuesta del motor al escalón de ingreso.

Con la función de transferencia definida, con el comando **pidtune(G,'PID')**, se determinan las constantes proporcional, integral y derivativa del sistema con ayuda de MATLAB, las cuales son:

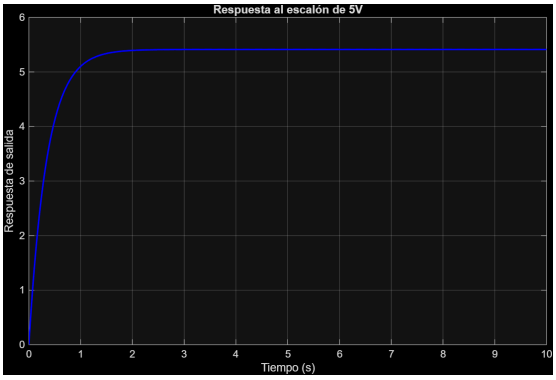

Fig. 9. Respuesta al escalón de la función de transferencia del sistema.

$$Kp = 1.32$$

$$Ki = 6.85$$

$$Kd = 0$$

- **Implementar un controlador PID para regular la velocidad del motor.**
  Para la implementación del PID, se usa la ecuación de recurrencias
  $\mathrm{u}_k = u_{k-1} + K_p \left(e_k - e_{k-1} + \frac{T}{K_i} e_k + \frac{T_d}{T} \left(e_k - 2e_{k-1} + e_{k-2} \right) \right)$
  La misma es codificada dentro de la interrupción, con el fin de aplicar correctamente la señal de control y seguir la referencia ante cualquier cambio. Las variables del error anterior y del error anterior al anterior se actualizan dentro de la misma interrupción, al igual que el error actual, calculado como la diferencia entre la referencia y las RPM medidas.
  La adquisición de las constantes del PID se realiza desde una interrupción no bloqueante, la cual se ejecuta cada 10 ms, con el objetivo de tener listos los datos provenientes del HMI para aplicarlos en la ecuación de recurrencias.

E. Resultados

Los resultados obtenidos con la implementación del controlador en la planta, utilizando la interfaz previamente explicada y las constantes proporcionadas por MATLAB, muestran un seguimiento de la referencia. Sin embargo, dicho seguimiento es lento y presenta algunas oscilaciones. En la figura 10 se observa este comportamiento.

Además, en la misma figura, se puede obsevar que aplicando un ajuste fino, la referencia es alcanzada sin problemas de una manera muy extacta, el ajuste es el siguiente:

$$Kp = 0.01$$

$$Ki = 0.19$$

$$Kd = 0$$

Ya que las oscilaciones eran muy marcadas reducimos el valor de la constante proporcional para suavizar ese efecto,

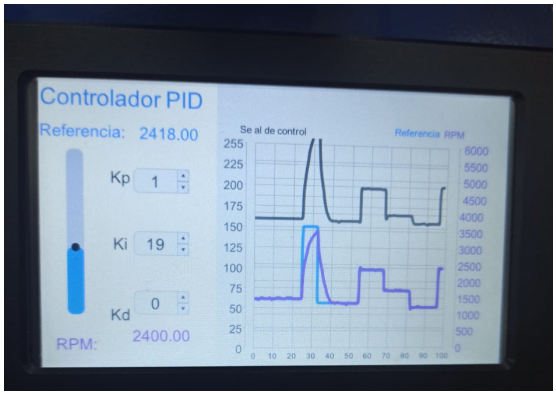

Fig. 10. Sintonización del controlador PID

mientras que, en la ganancia integral al disminuir su valor, la respuesta mostró un mejor seguimiento, menos error a lo largo del tiempo.

El código generado y comentado es el siguiente:

```
#include <Controllino.h>
#include "Stone_HMI_Define.h"          //
    Librer a ofical de HMI Stone
#include "Procesar_HMI.h"              //
    Librer a implementada para procesar
    respuestas del HMI

// Definicion de variables para ecuacion de
    recurrencias PID
float u_actual=0;                      //
    Almacena la se al de control actual
float u_anterior=0;
float Ki=19;                 // Almacena la
    se al de control anterior
float Kp = 10, Td = 0;    // Pesos de la
    ecuacion de recurrencias
float T=0.05;                          //
    Tiempo de muestreo para ecuacion de
    reccurencias (50ms)
float e[3];                            //
    Vector que almacena los errores (dimension
     3)

// VARIABLES PARA PWM DEL MOTOR
const int pin_motor        = CONTROLLINO_D0;
                  // Pin de salida PWM al motor
int    ref = 0;

    // Valor le do del slider del HMI
char label2_text[10];
                                       //
    Char para mostrar el Duty Cycle en el
    label2 del HMI

// VARIABLES PARA CONTEO DE PULSOS y RPM
const int entrada                      =
    CONTROLLINO_IN1;   // Pin de entrada de
    pulsos
volatile unsigned long conteo_pulsos = 0;
                    // Contador de pulsos
char label4_text[10];
                                       //
    Char para mostrar las RPM en el label4 del
     HMI
float rpm                              = 0;
                    // RPM calculadas
const uint16_t PULSOS_POR_REV          = 36;
                   // N mero de pulsos por
    revoluci n (Datos del EPC)
const float fs                  = 1/T;
                         // Frecuencia de
    muestreo

// VARIABLES PARA CONTROLAR EL TIEMPO DE ENVIO
     DE DATOS AL HMI
unsigned long t_previo=0;
unsigned long t_previo1=0;

// FUNCIONES ADICIONALES
void contarPulso();

void setup() {

  //Comunicaciones seriales
  Serial.begin(115200);   // Comunicaci n
      serial con el PC
  Serial2.begin(115200);  // Comunicaci n
      serial con el HMI

  //Establece el pin de entrada del EPC y
      salida del motor
  pinMode(entrada, INPUT);
  pinMode(pin_motor, OUTPUT);

  // Inicializar valores de error a 0
  e[0] = 0;  e[1] = 0;  e[2] = 0;

  //CONFIGURACION HMI
  //Inicializa el deslizador de referencia en
      0
  Stone_HMI_Set_Value("slider", "slider1",
      NULL, 0);

  // Inicializa los valores de las graficas en
       0
  STONE_push_series("line_series", "
      line_series1", 0); //Env a un valor del
       eje X a graficar en el line_series1 que
       se pondr  al final
  STONE_push_series("line_series", "
      line_series2", 0); //Env a un valor del
       eje X a graficar en el line_series2 que
       se pondr  al final
  STONE_push_series("line_series", "
      line_series3", 0); //Env a un valor del
       eje X a graficar en el line_series3 que
       se pondr  al final




  //Inicializa los valores de spin_box Kp, Ki,
       Kd: Ki diferente de cero para evitar
      errores por division con 0
  Stone_HMI_Set_Value("spin_box", "spin_box1",
       NULL, 0.01);  // Kd en 0 el valor del
      spin box1 en el HMI.
  Stone_HMI_Set_Value("spin_box", "spin_box2",
```

```
      NULL, 0.01);  // Ki en 0 el valor del
      spin box2 en el HMI.
  Stone_HMI_Set_Value("spin_box", "spin_box3",
      NULL, 0.01); // Ti en 0 el valor del
      spin box3 en el HMI.

  //Configuracion de la interrupcion, pin
      externo, funcion a ejecutar, flancos de
      bajada
  attachInterrupt(digitalPinToInterrupt(
      entrada), contarPulso, FALLING);
  noInterrupts();           //Comienza apagada
      la interrupcion
  TCCR1A = 0b00000000;      // Todo apagado,
      modo normal registro A. Counter1 del
      ATMEGA2560
  TCCR1B = 0b00000000;      // Todo apagado,
      modo normal registro B. Coun ter1 del
      ATMEGA2560
  TCCR1B |= B00000100;      // Configuraci n
      de preescaler a 256 (BIT CS12)
  TIMSK1 |= B00000010;      // Habilitar
      interrupci n por comparaci n usando el
       registro TIMSK1 (modo CTC)
  OCR1A = 3125;             // Establecer valor
       TOP para generar interrupci n: Tiempo
      interrupcion/16 ns=0.05/16ns=3125
  interrupts();             // Enciende la
      interrupcion
  HMI_init();               // Inicializaci n
      del sistema de colas para procesar las
      respuestas del HMI
}

void loop() {


  //Retardo no bloqueante de 10ms
  if(millis()-t_previo1>=10){

    ref= HMI_get_value("slider", "slider1");
            // Solicita el valor del widget
        slider1 (Referencia)
    Kp= HMI_get_value("spin_box", "spin_box1")
        /100;  // Solicita el valor del widget
         spin_box1 (Kp)
    Td= HMI_get_value("spin_box", "spin_box3")
        /100;  // Solicita el valor del widget
         spin_box3 (Kd)
    Ki= HMI_get_value("spin_box", "spin_box2")
        /100;  // Solicita el valor del widget
         spin_box2 (Ki)
    t_previo1=millis();
                                  // Acualizar
        tiempo previo con tiempo actual
  }

  //Retardo no bloqueante de 100ms
  if(millis()-t_previo>=100){

    t_previo=millis();

        // Almacena en tiempo previo el tiempo
         actual
    dtostrf(ref, 7, 2, label2_text);
                                          //
        Convertir float a char
    dtostrf(rpm, 7, 2, label4_text);
                                          //
        Convertir float a char
//    Kp= HMI_get_value("spin_box", "
    spin_box1")/100;  // Solicita el valor
    del widget spin_box1 (Kp)
//  Ti= HMI_get_value("spin_box", "spin_box2
    ")/100;  // Solicita el valor del widget
     spin_box2 (Ki)
//  Td= HMI_get_value("spin_box", "spin_box3
    ")/100;  // Solicita el valor del
    widget spin_box3 (Kd)
    Stone_HMI_Set_Text("label","label2",
        label2_text);                 //
        Env a el texto de referencia al
        label2 (Referencia)
    Stone_HMI_Set_Text("label","label4",
        label4_text);                 //
        Env a el texto de rpm al label4 (RMP)

    //Env a un valor del eje X a graficar en
        el line_series1
    STONE_push_series("line_series", "
        line_series1", ref);            //
        Graficar referencia
    //Env a un valor del eje X a graficar en
        el line_series2
    STONE_push_series("line_series", "
        line_series2", rpm);            //
        Graficar rpms
    //Env a un valor del eje X a graficar en
        el line_series3
    STONE_push_series("line_series", "
        line_series3", 150+(int)u_actual);
                // Graficar se al de control
  }
}

// Interrupci n por TIMER1 para muestrear las
    RPM debido a que la librer a del HMI se
    demora mucho
ISR(TIMER1_COMPA_vect){

  TCNT1=0;                // Resetea el timer (
      Timer 1 a cero)
  // Calcular RPM (pulsos por segundo / pulsos
      por revoluci n) * 60
  rpm = (float(conteo_pulsos)*60)*fs / (
      PULSOS_POR_REV);
  e[2] = e[1];            //Almacenar el error
      anterior del error anterior
  e[1] = e[0];            //Almacenar el error
      anterior
  e[0] = ref - rpm;       // Calculo del error
      actual

  if(Ki>0){ //Evitar division para 0,  Ti=(Kp/
      Ki)
    // Ecuacion de recurrencias del
        controlador PID
    u_anterior = Kp * (e[0] - e[1] + T / (Kp/
        Ki) * e[0] + Td / T * (e[0] - 2 * e[1]
         + e[2]));



  }
```

```
  // Sumar u anterior con u actual
  u_actual +=u_anterior;
  // Limitar la se al de 0 a 255 para PWM
  if (u_actual > 255){ //Si es mayor a 255,
      establece 255
    u_actual = 255;
  }
  if (u_actual < 0){ //Si es menor a 0,
      establece 0
    u_actual = 0;
  }
  //Escribir la senal de control PWM al pin
      del motor
  analogWrite(pin_motor, (int)u_actual);

  // Mostrar datos en el monitor serial
  Serial.print("\nRPM:␣");
  Serial.println(rpm);
  Serial.print("Pulsos:␣");
  Serial.println(conteo_pulsos);
  Serial.print("Referencia:␣");
  Serial.println(ref);
  Serial.print("Se al␣de␣control␣U:␣");
  Serial.println(u_actual);


  conteo_pulsos=0;  // Resetea los pulsos
}

// Interrupci n por Hardware para contar los
    pulsos del motor
void contarPulso() {
  conteo_pulsos++;  // Incrementar contador al
      detectar pulso
}
```

Listing 13. Implementación del controlador y uso del HMI con el EPC

V. Enlaces

El enlace al repositorio de las prácticas es:
https://github.com/martinjhvbnm/Control-Digital

VI. Conclusiones

El desarrollo de las prácticas permitió comprender el funcionamiento de un PLC y cómo podemos programarlo para realizar diferentes tareas, como encender LEDs, manejar botones, interactuar con plantas externas como el ECP e integrar el PLC con la interfaz HMI, con el fin de mejorar la experiencia de uso del sistema.

La implementación de diferentes técnicas de control, como el modelado de sistemas y el uso de controladores PID, facilitó la aplicación práctica de los conocimientos adquiridos en el curso, lo que contribuyó a una mejor comprensión de los mismos.

El uso de herramientas como MATLAB y LabVIEW permitió realizar procesos de forma más sencilla y gráfica, como la caracterización de la planta utilizando el módulo DAQ para obtener la función de transferencia, así como su análisis y la identificación de sistemas de manera visual, además con la función PIDTUNE, podemos obtener las constantes del controlador PID de una manera sencilla.

Por último, el uso del controlador Controllino, nos permitió aprender cómo, con la ayuda del temporizador (TIMER), es posible generar interrupciones muy precisas y aplicarlas en procesos de control, como se hizo con el controlador PID. En conjunto, todas estas técnicas contribuyeron a comprender el funcionamiento de los sistemas reales y a aplicar la teoría a través de la práctica.

References

[1] K. Ogata *y otros*, *Modern control engineering*. Prentice Hall India, 2009.