Parte B: Sistema de Control de Semáforos con FSM

Descripción Detallada

Este sistema implementa el control de dos semáforos ubicados en una intersección perpendicular, utilizando el enfoque de **Máquina de Estados Finita (FSM)** para garantizar un funcionamiento seguro y eficiente del tráfico vehicular.

Análisis del Código

Definición de Hardware

cpp

```cpp
// Definición de LEDs para Semáforo A (Norte-Sur)
const int ROJO_A = CONTROLLINO_D0;    // LED Rojo Semáforo A
const int AMARILLO_A = CONTROLLINO_D1; // LED Amarillo Semáforo A
const int VERDE_A = CONTROLLINO_D2;   // LED Verde Semáforo A

// Definición de LEDs para Semáforo B (Este-Oeste)
const int ROJO_B = CONTROLLINO_D6;    // LED Rojo Semáforo B
const int AMARILLO_B = CONTROLLINO_D7; // LED Amarillo Semáforo B
const int VERDE_B = CONTROLLINO_D8;   // LED Verde Semáforo B
```

Explicación: Se utilizan las variables predefinidas del Controllino para mapear cada LED a un pin específico. Esta organización facilita el mantenimiento y la comprensión del código.

Implementación de Enum

cpp

```cpp
typedef enum {
  Averde_Brojo,     // Semáforo A en verde, B en rojo
  Aamarillo_Brojo,  // Semáforo A en amarillo, B en rojo
  Arojo_Bverde,     // Semáforo A en rojo, B en verde
  Arojo_Bamarillo   // Semáforo A en rojo, B en amarillo
} EstadoSemaforo;
```

Ventajas del uso de enum:

- **Legibilidad**: Los nombres de estados son autodescriptivos
- **Mantenibilidad**: Fácil modificación y extensión
- **Prevención de errores**: Evita el uso de números mágicos
- **Optimización**: El compilador optimiza el código automáticamente

Variables de Control FSM

```cpp
EstadoSemaforo estado_actual = Averde_Brojo;
unsigned long tiempo_anterior = 0;
unsigned long tiempo_actual = 0;

// Tiempos estándar para cada estado (en milisegundos)
const unsigned long TIEMPO_VERDE = 3000;    // 3 segundos
const unsigned long TIEMPO_AMARILLO = 2000; // 2 segundos
```

Estructura de datos: Se utilizan variables globales para mantener el estado del sistema y controlar las transiciones temporales.

Funcionamiento de la FSM

Función setup()

```cpp
void setup() {
  // Configurar LEDs como salida
  pinMode(ROJO_A, OUTPUT);
  pinMode(AMARILLO_A, OUTPUT);
  pinMode(VERDE_A, OUTPUT);
  pinMode(ROJO_B, OUTPUT);
  pinMode(AMARILLO_B, OUTPUT);
  pinMode(VERDE_B, OUTPUT);

  // Inicializar todos los LEDs apagados
  apagarTodosLEDs();

  // Inicializar tiempo y comenzar con A verde, B rojo
  tiempo_anterior = millis();
  actualizarLEDs();
}
```

Inicialización del sistema:

1. Configuración de pines como salida
2. Estado inicial seguro (todos los LEDs apagados)
3. Establecimiento del estado inicial
4. Sincronización temporal

Función loop() Principal

cpp

```cpp
void loop() {
  tiempo_actual = millis();
  ejecutarFSM();
}
```

Diseño no bloqueante: El loop principal solo actualiza el tiempo y ejecuta la FSM, manteniendo la responsividad del sistema.

Lógica de la FSM

cpp

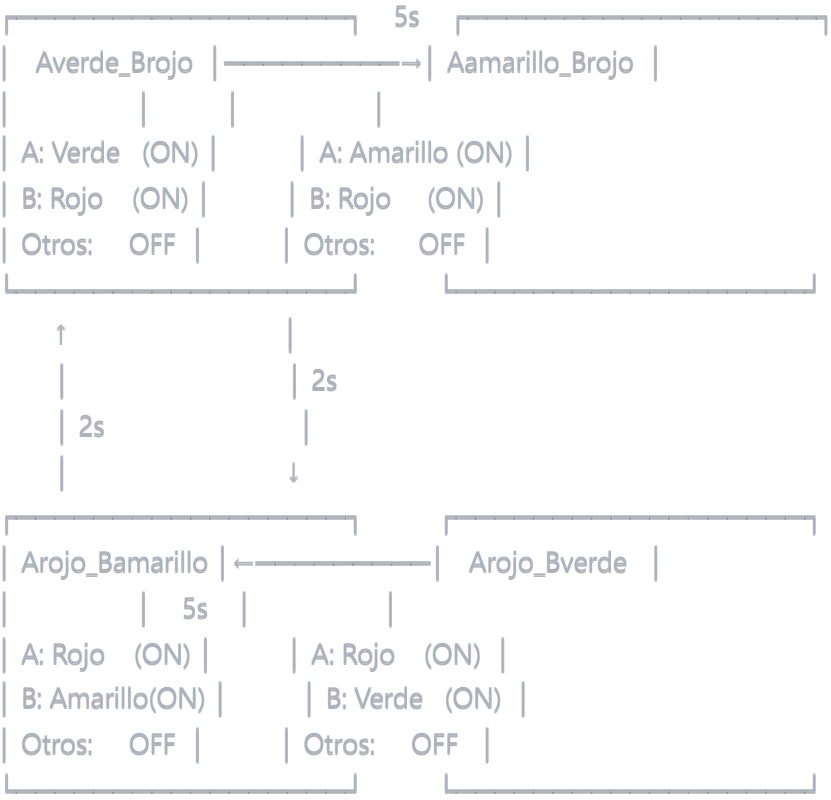
```cpp
void ejecutarFSM() {
  unsigned long tiempo_transcurrido = tiempo_actual - tiempo_anterior;

  switch (estado_actual) {
    case Averde_Brojo:
      if (tiempo_transcurrido >= TIEMPO_VERDE) {
        estado_actual = Aamarillo_Brojo;
        tiempo_anterior = tiempo_actual;
        actualizarLEDs();
      }
      break;
    // ... otros casos
  }
}
```

Características del control FSM:

- **Retardos no bloqueantes**: Uso de `millis()` para control temporal
- **Transiciones condicionadas**: Cambio de estado basado en tiempo transcurrido
- **Actualización inmediata**: Los LEDs se actualizan al momento del cambio de estado

Diagrama de Estados Detallado

Representación Visual del Sistema



Análisis de Transiciones

Transición	Tiempo	Condición	Acción
Estado 1 → Estado 2	5s	`tiempo_transcurrido >= TIEMPO_VERDE`	A: Verde→Amarillo
Estado 2 → Estado 3	2s	`tiempo_transcurrido >= TIEMPO_AMARILLO`	A: Amarillo→Rojo, B: Rojo→Verde
Estado 3 → Estado 4	5s	`tiempo_transcurrido >= TIEMPO_VERDE`	B: Verde→Amarillo
Estado 4 → Estado 1	2s	`tiempo_transcurrido >= TIEMPO_AMARILLO`	B: Amarillo→Rojo, A: Rojo→Verde

Funciones de Control

Función actualizarLEDs()

```cpp
void actualizarLEDs() {
  // Primero apagar todos los LEDs
  apagarTodosLEDs();

  // Encender LEDs según el estado actual
  switch (estado_actual) {
    case Averde_Brojo:
      digitalWrite(VERDE_A, HIGH);
      digitalWrite(ROJO_B, HIGH);
      break;
    // ... otros casos
  }
}
```

Patrón de diseño:

1. **Clear-then-Set**: Primero se apagan todos los LEDs
2. **State-based activation**: Se encienden solo los LEDs del estado actual
3. **Atomic operation**: La actualización es instantánea

Función apagarTodosLEDs()

```cpp
void apagarTodosLEDs() {
  digitalWrite(ROJO_A, LOW);
  digitalWrite(AMARILLO_A, LOW);
  digitalWrite(VERDE_A, LOW);
  digitalWrite(ROJO_B, LOW);
  digitalWrite(AMARILLO_B, LOW);
  digitalWrite(VERDE_B, LOW);
}
```

Función utilitaria: Garantiza un estado conocido antes de cada actualización.

Aspectos de Seguridad

1. Exclusión Mutua

- **Nunca verde simultáneo**: El diseño FSM garantiza que nunca ambos semáforos estén en verde
- **Estados bien definidos**: Cada estado especifica exactamente qué LEDs deben estar encendidos

2. Transiciones Seguras

- **Amarillo intermedio**: Siempre hay una fase de amarillo antes del cambio a rojo

- **Secuencia ordenada**: Las transiciones siguen un patrón lógico y predecible

3. Tiempos Estándar

- **Verde: 5 segundos**: Tiempo suficiente para el paso de vehículos
- **Amarillo: 2 segundos**: Tiempo de precaución estándar
- **Ciclo total: 14 segundos**: Tiempo razonable para el flujo de tráfico

Mejoras Implementadas

Retardos No Bloqueantes

- **Ventaja**: El sistema puede responder a otros eventos mientras controla los semáforos
- **Implementación**: Uso de `millis()` en lugar de `delay()`
- **Flexibilidad**: Permite futuras expansiones sin reestructurar el código

Código Modular

- **Separación de responsabilidades**: Cada función tiene una tarea específica
- **Reutilización**: Las funciones pueden ser llamadas desde múltiples lugares
- **Mantenimiento**: Facilita la depuración y modificación

Posibles Extensiones

1. **Sensor de tráfico**: Ajustar tiempos según densidad vehicular
2. **Modo nocturno**: Cambiar a parpadeo amarillo en horarios específicos
3. **Control manual**: Botón de emergencia para control manual
4. **Comunicación**: Interface para monitoreo remoto
5. **Peatones**: Integración de semáforos peatonales

Conclusiones

Este sistema demuestra:

- **Implementación correcta de FSM**: Estados bien definidos y transiciones claras
- **Programación no bloqueante**: Uso eficiente de recursos temporales
- **Diseño seguro**: Cumplimiento de normativas de tráfico básicas
- **Código mantenible**: Estructura clara y bien documentada

El sistema es robusto, seguro y cumple con todos los requisitos especificados en la práctica.