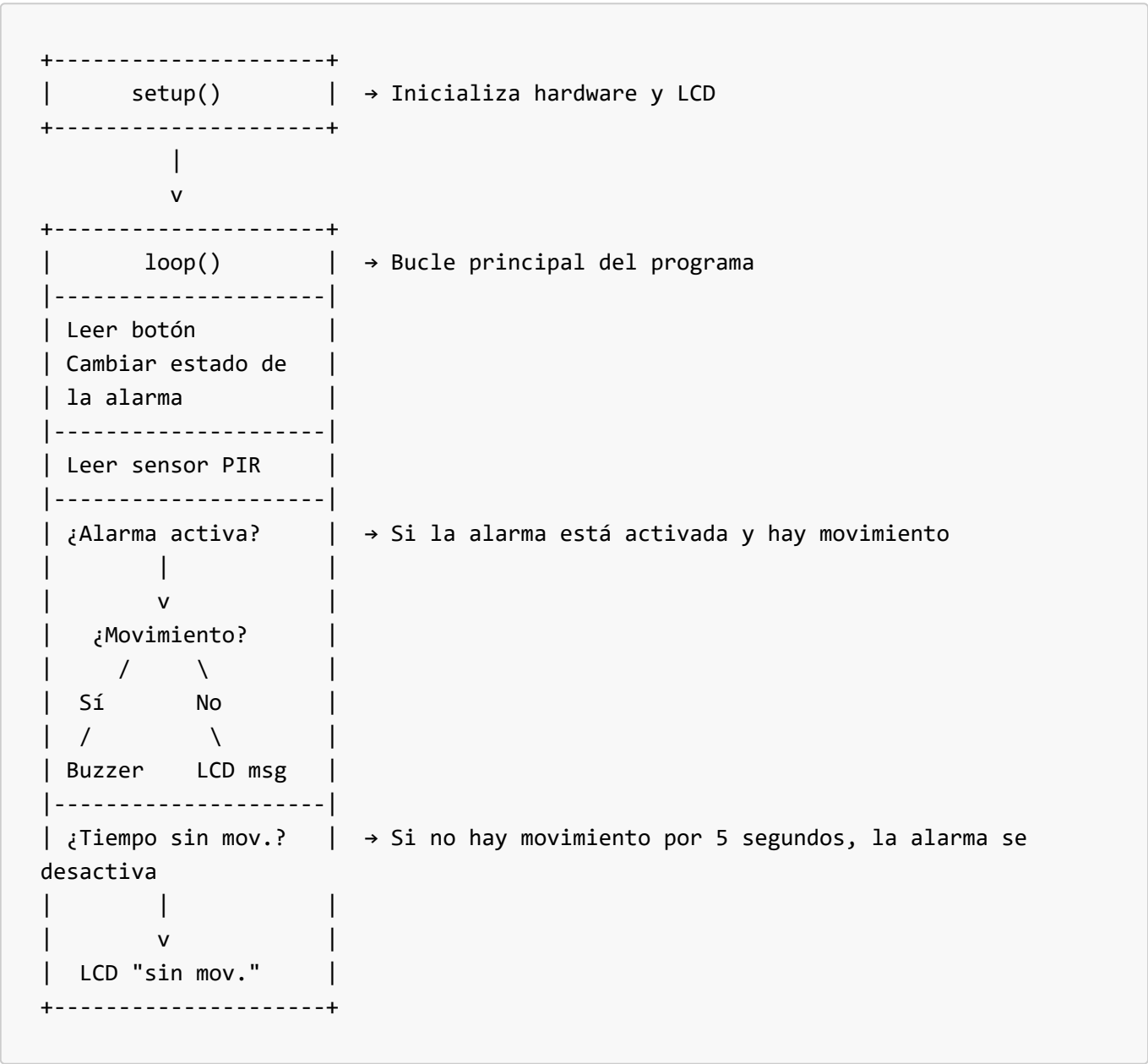


Funcionamiento General del programa

El programa cumple con los siguientes procesos:

- Detecta movimiento con un sensor PIR.
- Si la alarma está activada, y hay movimiento, activa un buzzer intermitente y muestra un mensaje de alerta en la pantalla LCD.
- Si no hay movimiento por 5 segundos, actualiza la pantalla para mostrar "sin movimiento".
- Muestra en el LCD si la alarma está activada, desactivada o si hay movimiento.
- Se puede activar o desactivar la alarma con un botón físico.

A continuación se presenta un esquema general del funcionamiento del programa:



Descripción del código del programa

Librerías utilizadas

```
#include <Arduino.h>
#include <Wire.h>
#include <LiquidCrystal_I2C.h>
```

Arduino.h

Librería estándar de Arduino para el manejo de pines digitales y analógicos, incluye funciones para leer y escribir en pines, así como para manejar temporizadores. Permite utilizar instrucciones tales como:

- *setup()*, *loop()*
- *digitalRead()*, *digitalWrite()*
- *pinMode()*
- *millis()*, *delay()*, etc.

Wire.h

Librería que permite la comunicación I2C entre el Arduino y dispositivos externos, de tal manera que se puede complementar nuestro circuito con sensores, pantallas LCD, RTC, expansores de pines, etc. Permite utilizar comandos como:

- *pinMode(pin, INPUT)* - Configura un pin especificado como entrada digital para leer señales externas.
- *pinMode(pin, OUTPUT)* - Configura un pin especificado como salida digital para enviar señales.
- *digitalWrite(pin, HIGH)* - Envía un nivel alto (voltaje) un pin configurado como salida.
- *digitalWrite(pin, LOW)* - Envía un nivel bajo (voltaje) un pin configurado como salida.
- *int estado = digitalRead(pin)* - Lee el valor digital (HIGH o LOW) desde un pin configurado como entrada.
- *int valor = analogRead(pin)* - Lee el valor analógico (de 0 a 1023) de un pin analógico.
- *analogWrite(pin, valor)* - Envía una señal PWM un pin para simular niveles analógicos (valor de 0 a 255).

LiquidCrystal_I2C.h

Una librería que permite controlar pantallas LCD usando un adaptador I2C (muy común en módulos LCD 16x2 o 20x4). Permite utilizar comandos como:

- *lcd.init()* — inicia la pantalla
- *lcd.setCursor(col, fila)* — posiciona el cursor
- *lcd.print("texto")* — escribe texto en pantalla
- *lcd.clear()* — borra el contenido
- *lcd.backlight()* — enciende la luz de fondo

Variables y objetos utilizados

Los números 2, 8, y 4 en estas líneas:

```
const int pinPIR = 2;
const int pinBuzzer = 8;
```

```
const int pinBoton = 4;
```

representan números de pines físicos del Arduino

¿Qué significan?

Pines de la Protoboard

```
const int pinPIR = 2;
```

- El sensor de movimiento PIR está conectado al pin digital 2 del Arduino.
- Este pin se configura como entrada, porque el sensor envía una señal que el Arduino debe leer.

```
const int pinBuzzer = 8;
```

- El buzzer (alarma) está conectado al pin digital 8.
- Este pin se configura como salida, porque el Arduino le envía corriente para hacer sonar el buzzer.

```
const int pinBoton = 4;
```

- El botón físico está conectado al pin digital 4.
- Este también es un pin de entrada, ya que el Arduino debe detectar si el botón se presionó.

Se usa const int para:

- Asignar un nombre legible a cada pin (más fácil que recordar "¿qué era el pin 8?").
- No modificar por accidente más adelante en el programa (es constante).

Puedes conectar los componentes a otros pines digitales si modificas los valores en el código. Por ejemplo, si conectas el buzzer al pin 7 en lugar del 8, tendrías que cambiar:

```
const int pinBuzzer = 7;
```

Siempre y cuando no elijas:

- Pines digitales (no todos los pines del Arduino tienen funciones iguales).
- El pin no estén siendo usados por otro componente.

Pantalla LCD del Arduino

```
LiquidCrystal_I2C lcd(0x27, 16, 2);
```

Crea un objeto llamado lcd que representa una pantalla LCD con interfaz I2C, este será el nombre del objeto que usarás para controlar la pantalla (como lcd.print(...), lcd.setCursor(...), etc.).

```
(0x27, 16, 2)
```

Tambien se cuenta con un objeto virtual de repuesto en caso de que no funcione el anterior:

```
LiquidCrystal_I2C lcd(0x3F, 16, 2);
```

Sientete libre de utilizar .x3F si el .x27 no funciona.

Son los parámetros de configuración:

Parámetro	Significado
0x27	Dirección I2C de la pantalla (puede ser 0x27, 0x3F, etc.).
16	Número de columnas (caracteres por línea) del LCD.
2	Número de filas del LCD.

"Voy a usar una pantalla LCD con interfaz I2C que tiene 16 columnas y 2 filas. Su dirección I2C es 0x27. Llamaré a esta pantalla lcd para controlarla desde el código."

Arreglo de estados de la alarma

```
enum EstadoAlarma {
    ALARMA_DESACTIVADA, //0
    ALARMA_ARMANDO, // 1
    ALARMA_ACTIVADA, // 2
    ALARMA_DISPARADA // 3
};
EstadoAlarma estadoActualAlarma = ALARMA_DESACTIVADA; // estado actual de la alarma
```

Estado	Descripción
ALARMA_DESACTIVADA	Sistema apagado. No reacciona ante movimiento.
ALARMA_ARMANDO	Cuenta regresiva antes de activarse (5 segundos).
ALARMA_ACTIVADA	Alarma armada, esperando movimiento.

Estado	Descripción
ALARMA_DISPARADA	Movimiento detectado. El buzzer suena intermitentemente.

Variables de control de la alarma

```
bool movimientoDetectadoActual = false;
```

Indica si la alarma ha detectado movimiento. Por defecto debe estar inicializada en falso.

```
bool movimientoDetectadoAnterior = false;
```

Indica si ha habido movimiento detectado o no y por defecto no habrá movimiento detectado. Esta variable ha de actualizarse constantemente en la función loop().

```
bool lcdNecesitaActualizar = true;
```

Se usa para evitar actualizar la pantalla LCD innecesariamente, de tal manera que en la pantalla se actualiza el mensaje cuando algo cambia (como desactivar la alarma o terminar el movimiento).

Variables para *Debounce* del botón

```
int valorBotonActual;
```

Guarda el estado actual del botón (presionado o no).

```
int valorBotonAnterior = HIGH;
```

Guarda el estado anterior del botón, de tal manera que esta variable nos permitira saber si el boton fue presionado o no al compararse con el valor de la variable valorBotonActual.

```
unsigned long tiempoUltimoCambioBoton = 0;
```

Guarda el momento en que cambió por última vez el estado del botón. La variable ha de ser usada para hacer el "debounce" (evitar falsos positivos por rebote mecánico del botón).

```
const unsigned long debounceDelay = 50;
```

Tiempo mínimo (en milisegundos) para aceptar un cambio de estado del botón como válido, en este caso está configurado en 50. *Podemos pensarlo como el tiempo mínimo que debe pasar antes de el sistema pueda detectar presión en el botón de nuevo*

Variables del buzzer (sonido intermitente)

```
unsigned long tiempoUltimaActivacionBuzzer = 0;
```

Guarda cuándo fue la última vez que se activó o desactivó el buzzer, se usa para que el buzzer suene de forma intermitente (ON-OFF-ON...). *¿Cuando se dio el último buzzer?*

```
const long duracionPulsoBuzzer = 250
```

Intervalo en milisegundos entre cada cambio de estado del buzzer (cada 250 ms), lo que produce un efecto de parpadeo sonoro cuando hay movimiento.

```
int conteoPulsosBuzzer = 0;
```

Variable de dedicada a contar cuántas veces se ha activado el buzzer;

```
const int maxPulsosBuzzer = 10;
```

"Parametro" dedicado a ha establecer el número máximo que se va a apagar y prender el buzzer del sistema de alarma; ten en cuenta que se cuenta un ciclo al dar un pitido y otro al no darlo osea que por defecto se tienen 5 veces ON + 5 veces OFF = 10 pulsos para 5 pitidos completos.

```
bool buzzerCompletadoCiclo = false;
```

Booleano que indica si se ha completado el ciclo de sonido del buzzer o no, indica si el buzzer ya terminó sus 5 pitidos

Variables del sensor de movimiento

```
unsigned long tiempoUltimoMovimientoReal = 0;
```

Guarda el tiempo en que se detectó el último movimiento, ha de ser utilizado para saber si ha pasado un tiempo determinado sin movimiento.

```
const unsigned long tiempoEsperaSinMovimiento = 500; // 0.5 segundos
```

Tiempo de espera (en milisegundos) para considerar que ya no hay movimiento. En este caso, después de 0.5 segundos el lcd mostrará el mensaje "Sin movimiento" y el buzzer se apagará.

Variables de configuración y armado

```
const unsigned long tiempoCalibracionPIR = 10000; // 10 segundos (10000 milisegundos)
```

Variable que guarda el tiempo de espera (en milisegundos) para que el sistema cargue y se calibre

```
unsigned long tiempoInicioAlarmaActiva = 0;
```

Guarda el tiempo en que la alarma pasó a estado ALARMA_ACTIVADA (después del armado)

```
const unsigned long duracionAlarmaAutomatica = 5 * 60 * 1000; // 5 minutos en milisegundos
```

Guarda el tiempo en el que la alarma estará activa en milisegundos

```
unsigned long tiempoInicioArmado = 0;
```

Guarda el tiempo en que se inició el proceso de armado

```
const unsigned long duracionRetardoArmado = 5000; // 5 segundos para el retardo de armado
```

Guarda el tiempo de retardo para el armado de la alarma

Variables para la verificación de conexión del PIR al armar

```
unsigned long tiempoInicioVerificacionPIRArmado = 0;
```

El tiempo/instante inicial en que se inició la verificación de la conexión del PIR al armar la alarma

```
const unsigned long duracionVerificacionPIRArmado = 1000; // 1 segundo
```

Tiempo a esperar para verificar la conexión del PIR al armar la alarma

```
int contadorDisparosAlarma = 0;
```

Contador para las veces que la alarma se ha disparado.

Tabla de resumen de variables y constantes

Variable	¿Para qué sirve?
<code>estadoActualAlarma</code>	Saber en qué estado está la alarma
<code>pinPIR, pinBuzzer, pinBoton</code>	Manejo físico de sensores y actuadores
<code>movimientoDetectadoActual</code>	Detección de movimiento actual
<code>movimientoDetectadoAnterior</code>	Detección del ciclo anterior
<code>lcdNecesitaActualizar</code>	Evita refrescos innecesarios del LCD
<code>valorBotonActual/Anterior</code>	Control del estado del botón
<code>tiempoUltimoCambioBoton</code>	Control del rebote del botón
<code>debounceDelay</code>	Tiempo mínimo entre lecturas válidas del botón
<code>tiempoUltimaAlternacionBuzzer</code>	Control intermitente del buzzer
<code>duracionPulsoBuzzer</code>	Frecuencia de parpadeo del buzzer
<code>conteoPulsosBuzzer</code>	Número de pitidos realizados
<code>buzzerCompletadoCiclo</code>	Indica si terminó el ciclo de 5 pitidos
<code>tiempoUltimoMovimientoReal</code>	Última detección real de movimiento
<code>tiempoEsperaSinMovimiento</code>	Tiempo sin movimiento para actualizar pantalla
<code>tiempoCalibracionPIR</code>	Tiempo inicial de calibración del PIR
<code>tiempoInicioAlarmaActiva</code>	Inicio de la fase activa
<code>duracionAlarmaAutomatica</code>	Tiempo antes de apagado automático
<code>tiempoInicioArmado</code>	Momento donde inicia el retardo de armado
<code>duracionRetardoArmado</code>	Retardo entre activación y estado activo

Bloques e instrucciones del código del programa

Bool verificarConexionPIR(){}

- Retorna true si el PIR detecta algo (un HIGH) en un corto período, false si no.
- Con una resistencia pulldown, un pin desconectado o un PIR en reposo (sin movimiento) leerá LOW.

- Solo si el PIR está conectado Y detecta movimiento, leerá HIGH. Esto ayuda a confirmar que el PIR está funcional.

Descripción : Verifica si el PIR está conectado y activo

```
bool verificarConexionPIR()
{
    unsigned long tiempoInicio = millis();
```

millis() es una función estándar de Arduino que devuelve el número de milisegundos transcurridos desde que la placa comenzó a ejecutarse (desde que se encendió o se reinició). Empezar medición del tiempo para determinar cuando apagar el sensor.

Mientras el tiempo transcurrido a partir de haberse iniciado la verificación sea menor que el tiempo de duración de la verificación, se ejecuta el siguiente bucle:

```
while((millis() - tiempoInicio) < duracionVerificacionPIRArmado)
{
    if (digitalRead(pinPIR) == HIGH)
    {
        return true; // PIR detectó algo, está conectado y funcional.
    }
    delay(10); // Pequeño delay para no sobrecargar el bucle.
}
```

Durante el bucle se ejecuta *digitalRead(pinPIR) == HIGH* como argumento de un condicional, básicamente se verifica si es que el pin donde está conectado nuestro sensor de movimiento a recibido señal de conexión, si ese es el caso se retornará un valor de "verdadero" si antes agregar un pequeño delay para evitar sobrecargar el bucle.

```
    return false; // PIR no detectó un HIGH en el tiempo de verificación
(desconectado o no funcional).
}
```

Continuando con la siguiente parte del bucle, si no se detecta "señal" en el sensor de movimiento se procede a devolver un valor booleano de "falso".

Void setup(){}

Este es bloque principal del programa, el cual se ha de ejecutar una sola vez al iniciar todo el sistema. Sus funciones principales son:

- Inicializar comunicación serial (debug por consola).
- Configurar los pines de entrada y salida.

- Inicializar y configurar la pantalla LCD.
- Realizar la secuencia de calibración del sensor PIR.
- Mostrar mensajes de estado en el LCD.
- Asegurar que el sistema comience en estado seguro (alarma desactivada y buzzer apagado).

Descripción de las funciones del bloque setup()

```
void setup() {  
  Serial.begin(9600);
```

Inicia la comunicación serial entre el computador y el Arduino para depuración

-- Configuración de pines --:

```
pinMode(pinPIR, INPUT);  
pinMode(pinBuzzer, OUTPUT);  
pinMode(pinBoton, INPUT_PULLUP);
```

- El pin del PIR (sensor de movimiento) es una entrada
- El pin del buzzer (alarma) es una salida
- Si usas resistencia pull-up INTERNA de Arduino (botón a GND), el boton del sistema es una tambien entrada, aunque algo diferente

-- Inicialización de la pantalla LCD --:

```
lcd.init();  
lcd.backlight();
```

Inicializa el LCD y enciende la luz de fondo del LCD (enciende la pantalla del lcd).

--- Secuencia de Inicio y Calibración del PIR ---:

Las intrucciones seguidas de *lcd*. manejan lo que puede mostrar en la pantalla del lcd, puedes pensar en estas como las instrucciones que se le dan a la terminal de una workspace en visual code. En este caso, para mostrar el mensaje de inicio del sistema se ha procedido de la siguiente manera

```
lcd.clear();  
lcd.setCursor(0, 0);  
lcd.print("Sistema de Alarma");  
lcd.setCursor(0, 1);  
lcd.print("Iniciando...");  
delay(2000);
```

Donde:

- `.clear`: Borra todo el contenido que puede haber estado en la pantalla LCD.
- `.setCursor(0, 1)`: Posiciona el cursor en la posición (0,0), como en la notación del objeto LCD, (0,1) se refiere al número de la fila y la columna del LCD; podemos pensar en él como una especie de '\r'.
- `.print("---")`: Muestra un mensaje en el lcd.
- `.delay(2000)`: Espera 2 segundos para mostrar el mensaje de inicio.

Ahora se puede entender:

```
lcd.clear();  
lcd.setCursor(0, 0);  
lcd.print("Calibrando PIR...");
```

-- Bucle para la cuenta regresiva en el LCD --:

```
for (int i = tiempoCalibracionPIR / 1000; i >= 0; i--) {  
  lcd.setCursor(0, 1); // Posicionar cursor en la segunda línea  
  lcd.print("Espere ");  
  if (i < 10) lcd.print(" ");  
  lcd.print(i);  
  lcd.print("s ");  
  Serial.print("Tiempo restante: ");  
  Serial.print(i);  
  Serial.println("s");  
  delay(1000); // Espera 1 segundo  
}  
Serial.println("PIR Calibrado. Sistema LISTO.");
```

En el argumento del *for* se transforma el tiempo de calibración asignado en la variable a segundos y se hace que decrezca este valor hasta que sea menor o igual a 0. En cada interacción del bucle se muestra el tiempo restante en la monitorización del sistema.

Si en algún momento el usuario modifica el tiempo de armado del sistema, el condicional se encarga de mostrar un espacio para no "desplazar" el tiempo que se muestra en pantalla ya que este estará compuesto por un dígito.

-- Asegurarse de que el buzzer esté APAGADO al final de la inicialización --:

```
digitalWrite(pinBuzzer, LOW);
```

Al pin donde está conectado el Buzzer, no se le asigna voltaje para evitar que se active

```
buzzerCompletadoCiclo = false;  
conteoPulsosBuzzer = 0;  
estadoActualAlarma = ALARMA_DESACTIVADA;
```

En dado caso de que las variables no se hayan inicializado con sus valores por defecto (configurados para que la alarma no suene), nos aseguramos de que sigan teniendo esos valores

```
lcdNecesitaActualizar = true;
```

Forzar actualización inicial del LCD

```
tiempoUltimoMovimientoReal = millis();
```

millis() Empieza a medir el tiempo desde que el sistema se ha activado.

```
movimientoDetectadoAnterior = false; // aun no debe activarse la alarma  
tiempoInicioAlarmaActiva = 0; // Asegurarse de que esté en 0 al inicio  
}
```