UNIVERSIDAD CATÓLICA BOLIVIANA SAN PABLO
SEDE TARIJA

DEPARTAMENTO DE CIENCIAS DE LA TECNOLOGÍA E INNOVACIÓN

CARRERA DE INGENIERÍA BIOMÉDICA

PRÁCTICA 04 - COMUNICACIÓN DIGITAL I2C Y SPI

ESTUDIANTES: JOSEPH SANTIAGO IQUIZE CONDORI

INGRID JAZMIN CRUZ SORUCO

ARACELY MELVA ZUBIETA MORALES
TUTOR: DOCENTE E. ALAN CORNEJO Q.

TARIJA-BOLIVIA

2025

Índice general

1 Introducción 1

2 Objetivos 1

2.1 Objetivo General 1

2.2 Objetivos Específicos 1

3 Materiales y equipos 1

3.1 Hardware 1

3.2 Software 2

4 Marco Teórico 2

4.1 Protocolo I2C 2

4.2 Sensor MPU6050 2

4.3 Comparación entre I2C y SPI 3

4.4 Aplicaciones Prácticas del MPU6050 3

5 Metodología 4

5.1 Diagrama de conexión 4

5.2 Implementación del código 4

5.3 Arquitectura del código y máquina de estados 4

6 Máquina de estados (FSM) del sistema 5

6.1 Descripción general de la FSM 5

6.2 Estados de la máquina 5

6.3 Representación gráfica 6

7 Resultados 7

7.1 Funcionamiento del sistema 7

7.2 Estabilidad de la comunicación 7

7.3 Captura del monitor serial 7

8 Análisis 7

8.1 Evaluación del desempeño 7

8.2 Limitaciones observadas 8

9 Conclusiones 9
10 Recomendaciones 9
11 Cuestionario 9
Anexos 11
Anexo A: Código principal (.ino) 11
Anexo B: funciones.h 12
Anexo C: funciones.cpp 13

1. Introducción

La comunicación digital entre microcontroladores y sensores es esencial para aplicaciones de medición, control y adquisición de datos en tiempo real. Entre los sensores más utilizados en sistemas embebidos se encuentra el MPU6050, un módulo que integra un acelerómetro y un giroscopio de tres ejes, permitiendo obtener información sobre movimiento, orientación y aceleración.

En el presente trabajo se implementó la comunicación digital entre un microcontrolador Arduino UNO y el sensor MPU6050, utilizando el protocolo I2C (Inter-Integrated Circuit). Este protocolo, ampliamente adoptado en dispositivos electrónicos, permite la interacción de múltiples módulos mediante dos líneas principales: SDA (datos) y SCL (reloj).

El proyecto abarca el diseño de la conexión eléctrica, la programación modular en Arduino IDE, el análisis del flujo de datos y la validación de la lectura en tiempo real mediante el monitor serial.

2. Objetivos

2.1. Objetivo General

Implementar una comunicación digital entre un microcontrolador y el sensor MPU6050 utilizando el protocolo I2C, permitiendo adquirir y visualizar datos en tiempo real.

2.2. Objetivos Específicos

- Comprender el funcionamiento del protocolo I2C y su estructura de comunicación.
- Integrar y configurar el sensor MPU6050 mediante librerías y funciones propias.
- Documentar la arquitectura modular del código fuente.
- Realizar la lectura, procesamiento y visualización de datos a través del monitor serial.

3. Materiales y equipos

3.1. Hardware

- Microcontrolador Arduino UNO.
- Sensor MPU6050.
- Protoboard y cables Dupont.
- Resistencias *pull-up* de 4.7kΩ o 10kΩ (cuando corresponde).

3.2. Software

- Entorno de desarrollo Arduino IDE.
- Librería `Wire.h` para comunicación I2C.
- Herramienta de monitoreo serial para visualizar los datos adquiridos.

4. Marco Teórico

4.1. Protocolo I2C

El protocolo I2C (Inter-Integrated Circuit) fue desarrollado por Philips como un estándar de comunicación serial síncrona que permite conectar múltiples dispositivos utilizando únicamente dos líneas compartidas: SDA (datos) y SCL (reloj). Su simplicidad y compatibilidad lo han convertido en uno de los protocolos más utilizados en sistemas embebidos, sensores digitales y módulos de adquisición de datos.

Según Carletti (2007), el bus I2C destaca por su bajo número de pines, su estructura maestro–esclavo y la facilidad con la que puede ampliarse el sistema mediante direccionamiento. El autor también enfatiza la importancia del uso de resistencias *pull-up* para garantizar niveles lógicos estables en las líneas del bus, especialmente en configuraciones con mayores longitudes de cable o presencia de ruido eléctrico.

Entre las características principales del I2C se encuentran:

- Uso de dos líneas compartidas: SDA (datos) y SCL (reloj).
- Comunicación síncrona generada por el maestro.
- Capacidad para múltiples dispositivos mediante direcciones.
- Velocidades comunes de 100 kHz (Standard Mode) y 400 kHz (Fast Mode).

4.2. Sensor MPU6050

El MPU6050 es un dispositivo MEMS que integra un acelerómetro de tres ejes y un giroscopio de tres ejes en un mismo encapsulado. De acuerdo con la documentación técnica de InvenSense, este módulo incluye un *Digital Motion Processor* (DMP), capaz de ejecutar internamente algoritmos para fusión de datos, reduciendo la carga de procesamiento en el microcontrolador principal.

Entre sus características más relevantes se encuentran:

- Rango programable del acelerómetro: $\pm$2g, $\pm$4g, $\pm$8g y $\pm$16g.
- Rango programable del giroscopio: $\pm$250, $\pm$500, $\pm$1000 y $\pm$2000 °/s.

- Conversores ADC de 16 bits para cada eje.
- Comunicación mediante I2C hasta 400 kHz.

Estas prestaciones hacen del MPU6050 una opción adecuada para aplicaciones en robótica, drones, sistemas de navegación inercial, estabilización de cámaras y análisis de movimiento humano.

4.3. Comparación entre I2C y SPI

El protocolo SPI (Serial Peripheral Interface) constituye una alternativa al I2C que se caracteriza por ofrecer mayores velocidades de transmisión, aunque a costa de utilizar un mayor número de líneas físicas (MOSI, MISO, SCK y CS).

En términos generales:

- **SPI** es más rápido y adecuado para dispositivos que requieren alto ancho de banda (memorias, pantallas, ADCs de alta velocidad).
- **I2C** es más simple, utiliza menos pines y es ampliamente soportado por sensores MEMS, como el MPU6050.

Carletti (2007) señala que, aunque SPI puede superar en velocidad a I2C, este último sigue siendo ampliamente utilizado en entornos educativos y prototipos por su flexibilidad y bajo costo de implementación.

4.4. Aplicaciones Prácticas del MPU6050

El MPU6050 es uno de los sensores inerciales más utilizados en proyectos de electrónica e ingeniería debido a su combinación de características técnicas y facilidad de integración. Entre sus aplicaciones se incluyen:

- Sistemas de estabilización para vehículos aéreos no tripulados (drones).
- Controladores de movimiento en robots móviles.
- Sistemas de navegación inercial para estimación de trayectoria.
- Análisis de movimiento humano en biomecánica y rehabilitación.
- Estabilización de cámaras y plataformas.

Bibliografía consultada (texto, no BibTeX)

Carletti, E. J. (2007). *Comunicación – Bus I2C*. Robots Argentina.

InvenSense Inc. (2013). *MPU-6000 and MPU-6050 Product Specification*.

NXP Semiconductors. (2014). *I2C-bus Specification and User Manual*.
Banzi, M. (2015). *Getting Started with Arduino*. Maker Media.

5. Metodología

El desarrollo de la práctica se organizó en etapas, desde la configuración física de las conexiones hasta la adquisición y análisis de los datos del MPU6050.

5.1. Diagrama de conexión

Las conexiones principales entre el Arduino UNO y el MPU6050 fueron:

- SDA $\rightarrow$ A4: línea de datos.
- SCL $\rightarrow$ A5: línea de reloj.
- VCC $\rightarrow$ 3.3V o 5V (según el módulo).
- GND $\rightarrow$ GND: referencia común.

Se utilizaron resistencias *pull-up* en SDA y SCL para asegurar una correcta forma de onda en el bus I2C.

5.2. Implementación del código

La lógica de control se implementó en Arduino IDE utilizando una arquitectura modular:

- Uso de la librería `Wire.h` para inicializar y manejar la comunicación I2C.
- Lectura de registros internos del MPU6050 para obtener valores crudos de aceleración y velocidad angular.
- Conversión de los datos crudos a unidades físicas (g y °/s), aplicando factores de escala.
- Envío de los datos al monitor serial para su visualización en tiempo real.

5.3. Arquitectura del código y máquina de estados

El código se estructuró mediante funciones específicas para inicialización, lectura y procesamiento de datos, apoyándose en una máquina de estados finitos (FSM) que organiza el flujo del programa.

6. Máquina de estados (FSM) del sistema

La implementación de la comunicación I2C entre el microcontrolador y el sensor MPU6050 puede describirse mediante una máquina de estados finitos (FSM), la cual organiza el flujo del programa en etapas claramente definidas. Esta estructura permite separar la inicialización, la configuración del sensor, la adquisición de datos y el envío de información al monitor serial, facilitando la comprensión y el mantenimiento del código.

6.1. Descripción general de la FSM

La FSM propuesta se compone de la siguiente secuencia de estados:

- **Estado 0 – Inicio**: encendido del sistema y preparación del entorno.
- **Estado 1 – Configuración del bus I2C**: inicialización de la comunicación mediante `Wire.begin()`.
- **Estado 2 – Configuración del MPU6050**: escritura de registros internos para establecer rangos de medición.
- **Estado 3 – Lectura de datos**: obtención periódica de los registros de aceleración y velocidad angular.
- **Estado 4 – Procesamiento**: conversión de los datos crudos a unidades físicas.
- **Estado 5 – Envío de datos**: transmisión de la información al monitor serial.
- **Estado 6 – Espera / actualización**: pequeña pausa o sincronización antes del siguiente ciclo.

6.2. Estados de la máquina

Estado 0 – Inicio: configuraciones iniciales del sistema, como la apertura del puerto serie y la preparación de variables.

Estado 1 – Configuración del bus I2C: se inicializa la interfaz I2C y se verifica que el bus esté disponible.

Estado 2 – Configuración del MPU6050: se programan los registros del sensor, definiendo rangos y modos de operación.

Estado 3 – Lectura de datos: se obtienen los valores crudos de los ejes de aceleración y giroscopio.

Estado 4 – Procesamiento: se convierten los datos crudos a g y °/s, y opcionalmente se aplican compensaciones.

Estado 5 – Envío de datos: los datos procesados son enviados al monitor serial para su observación.

Estado 6 – Espera / actualización: se introduce un retardo o sincronización que permite controlar la frecuencia de muestreo.

6.3. Representación gráfica

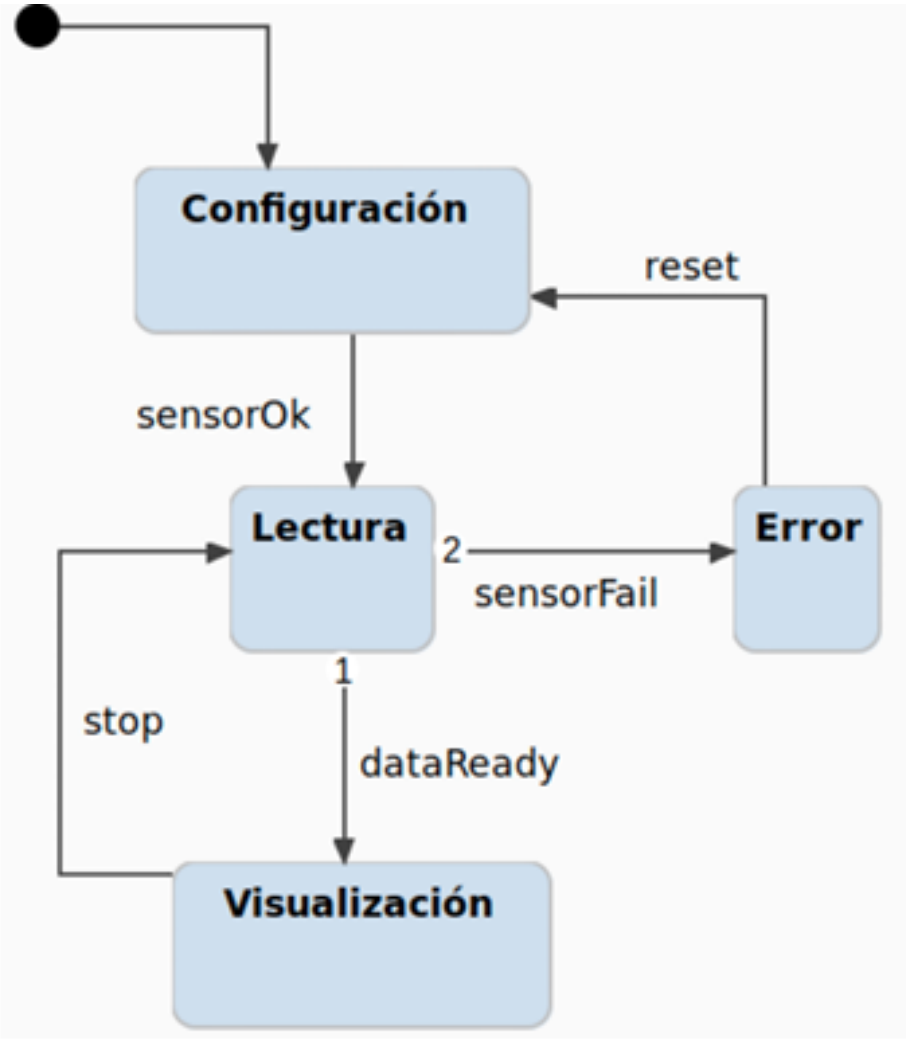

Figura 1: *Representación general de la máquina de estados para la comunicación I2C con el sensor MPU6050.*

7. Resultados

7.1. Funcionamiento del sistema

El sensor MPU6050 estableció comunicación I2C de forma correcta con el Arduino UNO. Las lecturas del acelerómetro y giroscopio respondieron de forma coherente frente a inclinaciones y movimientos del módulo.

7.2. Estabilidad de la comunicación

El uso de resistencias *pull-up* de 4.7kΩ permitió obtener señales estables en las líneas SDA y SCL, reduciendo el ruido y evitando errores de transmisión.

7.3. Captura del monitor serial

Las mediciones mostraron un formato similar al siguiente:

```
Aceleración (g): X=0.05 Y=-0.12 Z=0.98
Giroscopio (°/s): X=0.85 Y=-1.24 Z=0.10
```

Lo anterior evidenció que el sistema fue capaz de adquirir datos en tiempo real y reflejar adecuadamente los cambios de posición y movimiento del sensor.

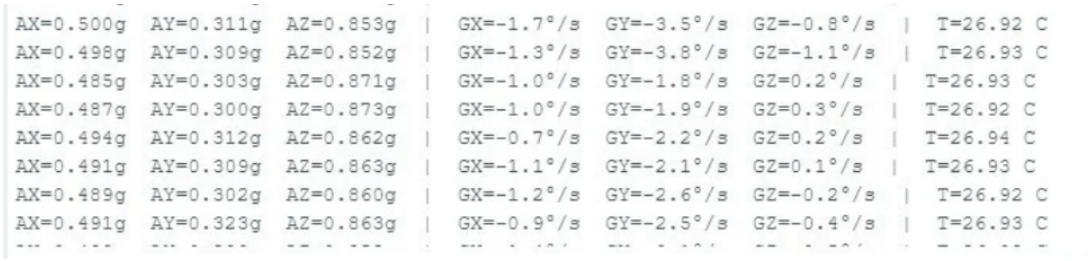

Figura 2: *Archivo* `funciones.cpp` *– Parte 1.*

8. Análisis

El desempeño general del sistema fue adecuado para los objetivos planteados en la práctica. El uso de Arduino UNO y el MPU6050 permitió comprobar en la práctica los conceptos teóricos de comunicación I2C y lectura de sensores MEMS.

8.1. Evaluación del desempeño

- La tasa de muestreo alcanzada fue suficiente para observar cambios en tiempo real.
- La modularidad del código facilitó la depuración y posibles extensiones del proyecto.
- La conversión de datos crudos a unidades físicas hizo posible una interpretación más intuitiva de las mediciones.

8.2. Limitaciones observadas

- Presencia de ruido en las lecturas, propio de sensores MEMS.
- Ligero offset en los valores iniciales, que podría compensarse mediante calibración.
- Restricción en la frecuencia máxima de muestreo debido a las limitaciones del Arduino UNO.

9. Conclusiones

- La comunicación I2C entre el Arduino UNO y el MPU6050 se implementó de manera exitosa, permitiendo la adquisición de datos de aceleración y velocidad angular en tiempo real.
- El diseño modular del código resultó beneficioso para la organización y comprensión del proyecto.
- El uso de resistencias *pull-up* fue determinante para asegurar la estabilidad de las líneas SDA y SCL.
- La práctica cumplió con los objetivos propuestos, integrando la teoría del protocolo I2C con su aplicación práctica en un sensor inercial.

10. Recomendaciones

- Implementar filtros digitales (por ejemplo, filtro complementario o filtro de Kalman) para mejorar la estabilidad de las mediciones.
- Migrar a plataformas más rápidas como ESP32 o STM32 para aumentar la frecuencia de muestreo.
- Incorporar almacenamiento de datos en tarjeta SD para registrar trayectorias y movimientos prolongados.
- Integrar herramientas de visualización gráfica en Processing o Python para un análisis más detallado de los datos.

11. Cuestionario

Responde de forma clara y fundamentada las siguientes preguntas:

1. ¿Qué diferencias existen entre los protocolos I2C y SPI en cuanto al número de líneas, velocidad de transmisión y esquema de direccionamiento?
2. ¿Qué función cumple la señal de ACK (*acknowledge*) en la comunicación I2C?
3. ¿Qué sucede si el dispositivo esclavo no envía un ACK durante una transmisión I2C?
4. ¿Cuál es la función de la línea Chip Select (CS) en el protocolo SPI?
5. ¿Qué ventajas presenta SPI frente a I2C en aplicaciones que requieren alta velocidad de transferencia?
6. ¿Qué limitaciones presenta cada protocolo cuando se conectan múltiples dispositivos en el mismo bus?
7. ¿Cómo podría verificarse la comunicación realizada utilizando un osciloscopio o un analizador lógico?

Respuestas del Cuestionario

1. ¿Qué diferencias existen entre los protocolos I2C y SPI en cuanto al número de líneas, velocidad de transmisión y esquema de direccionamiento?

El protocolo I2C utiliza únicamente dos líneas (SDA y SCL), mientras que SPI requiere cuatro líneas (MOSI, MISO, SCK y CS). I2C emplea un sistema de direccionamiento que permite conectar hasta 127 dispositivos, mientras que SPI utiliza líneas de Chip Select para habilitar cada esclavo. En cuanto a velocidad, I2C opera típicamente entre 100 kHz y 400 kHz, mientras que SPI puede alcanzar varios MHz, siendo mucho más rápido para aplicaciones que requieren alta transferencia de datos.

2. ¿Qué función cumple la señal de ACK (acknowledge) en la comunicación I2C?

El ACK es una confirmación enviada por el dispositivo receptor después de recibir un byte. Esta señal indica al maestro que el dato fue recibido correctamente y que la comunicación puede continuar. Si el receptor baja la línea SDA en el noveno pulso de reloj, se interpreta como un ACK válido.

3. ¿Qué sucede si el dispositivo esclavo no envía un ACK durante una transmisión I2C?

Si el esclavo no envía un ACK, el maestro recibe un NACK, indicando que el dato no fue aceptado. En ese caso, el maestro debe detener la comunicación mediante una condición de STOP o reintentar la transmisión. La falta de ACK puede deberse a desconexión, falla del sensor o dirección incorrecta.

4. ¿Cuál es la función de la línea Chip Select (CS) en el protocolo SPI?

La línea Chip Select (CS) permite seleccionar qué dispositivo esclavo participará en la comunicación. Cuando la línea CS del dispositivo se coloca en bajo, el módulo queda habilitado para transmitir y recibir datos; los demás esclavos permanecen inactivos.

5. ¿Qué ventajas presenta SPI frente a I2C en aplicaciones que requieren alta velocidad de transferencia?

SPI ofrece velocidades muy superiores a I2C, permite comunicación full-duplex, reduce la latencia y tiene menor complejidad en el proceso de transmisión de datos. Por estas razones, SPI es preferido en pantallas, memorias, conversores rápidos y sistemas que requieren tiempos de respuesta mínimos.

6. ¿Qué limitaciones presenta cada protocolo cuando se conectan múltiples dispositivos en el mismo bus?

En I2C, la limitación principal es la cantidad de direcciones disponibles y la mayor capacitancia del bus al añadir dispositivos. Además, todos los módulos comparten las líneas SDA y SCL. En SPI, la limitación está en el número de líneas Chip Select requeridas: cada dispositivo necesita una CS independiente, lo que aumenta el cableado y el uso de pines del microcontrolador.

7. ¿Cómo podría verificarse la comunicación realizada utilizando un osciloscopio o un analizador lógico?

Con un osciloscopio se puede observar la forma de onda de las líneas SDA/SCL o MOSI/MISO/SCK para verificar sincronización, niveles lógicos y estabilidad de la señal. Con un analizador lógico es posible decodificar directamente el protocolo I2C o SPI, mostrando bytes transmitidos, direcciones, ACK/NACK y posibles errores de comunicación. Esto permite un diagnóstico preciso del funcionamiento del bus.

Anexos

Los siguientes anexos presentan el código fuente utilizado en el proyecto, dividido en capturas del archivo principal `.ino`, del archivo de cabecera `funciones.h`, y del archivo de implementación `funciones.cpp`. Estos elementos complementan la comprensión del funcionamiento interno del sistema y permiten una mejor documentación del trabajo realizado.

Anexo A: Código principal (`.ino`)

EmbebidosFinal.ino | funciones.cpp | funciones.h

```
1  #include "funciones.h"
2  void setup() {
3    Sis_ini();     // antes: iniciarSistema / App::inicio
4  }
5  void loop()  {
6    Sis_lop();     // antes: cicloSistema / App::loop
7  }
8
```

Figura 3: *Código principal del programa (`EMBEBIDOSFINAL.ino`).*

Anexo B: Archivo `funciones.h`

```
#pragma once
#include <Arduino.h>
#define BAUDRATE          115200
#define LED_PIN           LED_BUILTIN
#define I2C_CLOCK_START   25000
#define I2C_CLOCK_RUN     100000
#define MPU_ADDR_HINT     0x68
#define PRINT_HZ          20

struct SensorData {
  float ax_g, ay_g, az_g;
  float gx_dps, gy_dps, gz_dps;
  float tempC;
};
enum AppState : uint8_t { ST_INIT, ST_READY, ST_READ, ST_PRINT, ST_ERROR };
extern AppState  g_state;
extern uint32_t  g_lastPrintMs;
void Sis_ini(uint32_t baud = BAUDRATE);
void Sis_lop();
```

Figura 4: *Contenido del archivo `funciones.h` utilizado en el proyecto.*

Anexo C: Archivo `funciones.cpp`

```cpp
#include "funciones.h"
#include <Wire.h>
#include <math.h>  // fabsf

AppState g_state       = ST_INIT;
uint32_t g_lastPrintMs = 0;
static uint8_t g_mpuAddr = (uint8_t)MPU_ADDR_HINT;  // dirección elegida (0x68/0x69)

static const uint8_t PIN_SDA_UNO = A4;
static const uint8_t PIN_SCL_UNO = A5;

static void i2cBusRecover() {
  pinMode(PIN_SDA_UNO, INPUT_PULLUP);
  pinMode(PIN_SCL_UNO, INPUT_PULLUP);
  delay(2);

  if (digitalRead(PIN_SDA_UNO) == LOW) {
    pinMode(PIN_SCL_UNO, OUTPUT);
    for (uint8_t i = 0; i < 9; ++i) {
      digitalWrite(PIN_SCL_UNO, HIGH); delayMicroseconds(8);
      digitalWrite(PIN_SCL_UNO, LOW);  delayMicroseconds(8);
    }
    pinMode(PIN_SCL_UNO, INPUT_PULLUP);
    delay(2);
  }
  pinMode(PIN_SDA_UNO, OUTPUT);
  digitalWrite(PIN_SDA_UNO, LOW);  delayMicroseconds(8);
  pinMode(PIN_SCL_UNO, INPUT_PULLUP); delayMicroseconds(8);
  pinMode(PIN_SDA_UNO, INPUT_PULLUP); delayMicroseconds(8);
  delay(2);
}
```

Figura 5: *Archivo `funciones.cpp` – Parte 1.*

```
//Driver
namespace {

class MPU6050 {
public:
  explicit MPU6050(TwoWire& w = Wire) : _wire(&w) {}

  bool begin() {
    i2cBusRecover();
    _wire->begin();
    _wire->setClock(I2C_CLOCK_START);
    delay(5);
    if (!detectAddrSilent()) return false;
    if (!wr(0x6B, 0x80)) return false;
    delay(100);
    if (!wr(0x6B, 0x01)) return false;
    delay(5);
    if (!wr(0x6A, 0x00)) return false;
    if (!wr(0x23, 0x00)) return false;
    delay(5);
    if (!wr(0x1A, 0x03)) return false;
    if (!wr(0x1B, 0x00)) return false;
    if (!wr(0x1C, 0x00)) return false;
    if (!wr(0x19, 0x07)) return false;
    wrNoStop(g_mpuAddr, 0x3A);
    _wire->requestFrom((int)g_mpuAddr, 1);
    while (_wire->available()) (void)_wire->read();
    _wire->setClock(I2C_CLOCK_RUN);
    delay(10);
    int16_t ax,ay,az,gx,gy,gz;
    readRaw(ax,ay,az,gx,gy,gz); delay(5);
    readRaw(ax,ay,az,gx,gy,gz); delay(5);
    return true;
  }
```

Figura 6: *Archivo* `funciones.cpp` *– Parte 2.*

```
  bool readRaw(int16_t& ax,int16_t& ay,int16_t& az,
               int16_t& gx,int16_t& gy,int16_t& gz) {
    uint8_t b[14];
    if (!rdBlock(0x3B, b, 14)) return false;
    ax = (int16_t)((b[0] << 8)  | b[1]);
    ay = (int16_t)((b[2] << 8)  | b[3]);
    az = (int16_t)((b[4] << 8)  | b[5]);
    gx = (int16_t)((b[8] << 8)  | b[9]);
    gy = (int16_t)((b[10] << 8) | b[11]);
    gz = (int16_t)((b[12] << 8) | b[13]);
    return true;
  }
  bool readTempC(float& tC) {
    uint8_t b[2];
    if (!rdBlock(0x41, b, 2)) return false;
    int16_t raw = (int16_t)((b[0] << 8) | b[1]);
    tC = (raw / 340.0f) + 36.53f;
    return true;
  }
private:
  bool detectAddrSilent() {
    const uint8_t cand[2] = {0x68, 0x69};
    for (uint8_t i=0;i<2;i++){
      uint8_t addr = cand[i];
      if (!wrNoStop(addr, 0x75)) continue;
      if (_wire->requestFrom((int)addr, 1) != 1) { while(_wire->available()) (void)_wire->read(); continue; }
      uint8_t who = _wire->read();
      if (who == 0x68) { g_mpuAddr = addr; return true; }
    }
    return false;
  }
  bool wr(uint8_t reg, uint8_t val) {
    _wire->beginTransmission(g_mpuAddr);
    _wire->write(reg); _wire->write(val);
    return (_wire->endTransmission() == 0);
  }
```

Figura 7: *Archivo* `funciones.cpp` *– Parte 3.*

```
102     bool wrNoStop(uint8_t addr, uint8_t reg){
103       _wire->beginTransmission(addr);
104       _wire->write(reg);
105       return (_wire->endTransmission(false) == 0);
106     }
107     int req(uint8_t addr, int n) {
108       return _wire->requestFrom((int)addr, (int)n);
109     }
110     bool rdBlock(uint8_t startReg, uint8_t* buf, uint8_t len) {
111       for (uint8_t attempt = 0; attempt < 3; ++attempt) {
112         if (!wrNoStop(g_mpuAddr, startReg)) { delay(3); continue; }
113         int n = req(g_mpuAddr, len);
114         if (n == (int)len) {
115           for (uint8_t i=0;i<len;++i) buf[i] = _wire->read();
116           return true;
117         } else {
118           while (_wire->available()) (void)_wire->read();
119           delay(3);
120         }
121       }
122       return false;
123     }
124     TwoWire* _wire;
125   };
```

Figura 8: *Archivo* `funciones.cpp` *– Parte 4.*

```cpp
MPU6050 mpu(Wire);
inline float accLSBtoG(int16_t v)    { return v / 16384.0f; }
inline float gyroLSBtoDPS(int16_t v) { return v / 131.0f;    }
bool sensorRead(SensorData& out) {
  int16_t ax,ay,az,gx,gy,gz;
  if (!mpu.readRaw(ax,ay,az,gx,gy,gz)) return false;
  bool allZero1 = (ax==0 && ay==0 && az==0 && gx==0 && gy==0 && gz==0);
  if (allZero1) {
    delay(5);
    if (!mpu.readRaw(ax,ay,az,gx,gy,gz)) return false;
  }
  out.ax_g   = accLSBtoG(ax);
  out.ay_g   = accLSBtoG(ay);
  out.az_g   = accLSBtoG(az);
  out.gx_dps = gyroLSBtoDPS(gx);
  out.gy_dps = gyroLSBtoDPS(gy);
  out.gz_dps = gyroLSBtoDPS(gz);
  float tC;
  if (!mpu.readTempC(tC)) return false;
  out.tempC = tC;
  bool allZero2 = (out.ax_g==0 && out.ay_g==0 && out.az_g==0 &&
                   out.gx_dps==0 && out.gy_dps==0 && out.gz_dps==0 &&
                   fabsf(out.tempC - 36.53f) < 0.01f);
  if (allZero2) return false;
  return true;
}
static void printLine(const SensorData& d) {
  Serial.print(F("AX=")); Serial.print(d.ax_g,3); Serial.print(F("g  "));
  Serial.print(F("AY=")); Serial.print(d.ay_g,3); Serial.print(F("g  "));
  Serial.print(F("AZ=")); Serial.print(d.az_g,3); Serial.print(F("g  |  "));
  Serial.print(F("GX=")); Serial.print(d.gx_dps,1); Serial.print(F("°/s  "));
  Serial.print(F("GY=")); Serial.print(d.gy_dps,1); Serial.print(F("°/s  "));
  Serial.print(F("GZ=")); Serial.print(d.gz_dps,1); Serial.print(F("°/s  |  "));
  Serial.print(F("T="));  Serial.print(d.tempC,2); Serial.println(F(" C"));
}
}
```

Figura 9: *Archivo* `funciones.cpp` *– Parte 5.*

```
163  void Sis_ini(uint32_t baud){
164    pinMode(LED_PIN, OUTPUT);
165    digitalWrite(LED_PIN, LOW);
166    Serial.begin(baud);
167    delay(1500);
168    i2cBusRecover();
169    if (!mpu.begin()) {
170      g_state = ST_ERROR;
171      return;
172    }
173    g_lastPrintMs = 0;
174    g_state = ST_READY;
175  }
176  void Sis_lop(){
177    static SensorData d;
178    const uint32_t periodMs = (PRINT_HZ > 0) ? (1000UL / PRINT_HZ) : 50UL;
179    const uint32_t now = millis();
180
181    switch (g_state) {
182      case ST_INIT:
183      case ST_READY:
184        if (now - g_lastPrintMs >= periodMs) g_state = ST_READ;
185        break;
186
187      case ST_READ:
188        if (sensorRead(d)) g_state = ST_PRINT;
189        else               g_state = ST_ERROR;
190        break;
191
192      case ST_PRINT:
193        printLine(d);
194        g_lastPrintMs = now;
195        digitalWrite(LED_PIN, !digitalRead(LED_PIN));
196        g_state = ST_READY;
197        break;
198
199      case ST_ERROR:
```

Figura 10: *Archivo* `funciones.cpp` *– Parte 6.*

```
199       case ST_ERROR:
200       default:
201         // Silencioso en error: reintenta
202         delay(50);
203         g_state = ST_READY;
204         break;
205     }
206   }
```

Figura 11: *Archivo* `funciones.cpp` *– Parte 7.*