main.py

|  |
| --- |
| from MPU6050 import MPU6050  from motores import Motores  from sensores import SensoresLaser  from pid\_controller import PIDController  import time  import machine  from umqtt.simple import MQTTClient  import network  import \_thread  import ujson  SSID = "EBM" # Cambia por el nombre de tu red WiFi  PASSWORD = "Rv186502" # Cambia por la contraseña de tu red WiFi  # --- Configuración MQTT ---  BROKER = "192.168.232.115" # IP del broker MQTT (Raspberry Pi)  PORT = 1883  TOPIC\_ILUMINACION = b"camara/localizacion" # Topic para la coordenada detectada  TOPIC\_OBJETIVO = b"camara/objetivo" # Topic para la coordenada objetivo  posicion\_actual = None  destino = None  coordenada\_detectada\_procesada = False  coordenada\_objetivo\_procesada = False  # Configuración  velocidad\_inicial, velocidad\_final, velocidad\_giro = 450, 340, 700  kp = 100 # Ganancia proporcional para el controlador PID  angulo\_giro = 90  # Dimensiones del entorno  celda = 70 # Tamaño de cada celda en mm (7 cm)  offset\_x = 80 # Desplazamiento en X desde el borde hasta el recuadro imaginario  offset\_y = 70 # Desplazamiento en Y desde el borde hasta el recuadro imaginario  # Inicialización de componentes  def inicializar\_componente(nombre, clase):  try:  print(f"Inicializando {nombre}...")  componente = clase()  print(f"{nombre} inicializado correctamente.")  return componente  except Exception as e:  print(f"Error al inicializar {nombre}: {e}")  machine.reset()  mpu = inicializar\_componente("MPU6050", MPU6050)  motores = inicializar\_componente("Motores", Motores)  sensores = inicializar\_componente("SensoresLaser", SensoresLaser)  mpu.calibrarGiroscopio()  # Inicialización del controlador PID  pid = PIDController(kp)  # Variables de posición  #posicion\_actual = [1, 1] # El robot comienza en la celda (1, 1)  orientacion\_actual = "NORTE" # NORTE, SUR, ESTE, OESTE  def conectar\_wifi():  wlan = network.WLAN(network.STA\_IF)  wlan.active(True)  wlan.connect(SSID, PASSWORD)  print("Conectando a la red WiFi...")  while not wlan.isconnected():  time.sleep(1)  print("Conexión WiFi establecida:", wlan.ifconfig())  def mqtt\_callback(topic, msg):  global posicion\_actual, destino  global coordenada\_detectada\_procesada, coordenada\_objetivo\_procesada  if topic == TOPIC\_ILUMINACION:  try:  coords = msg.decode("utf-8").split(",")  posicion\_actual = [int(coords[0]), int(coords[1])]  coordenada\_detectada\_procesada = True  print(f"Coordenada detectada recibida y procesada: {posicion\_actual}")  except Exception as e:  print(f"Error al procesar la coordenada detectada: {e}")  elif topic == TOPIC\_OBJETIVO:  try:  coords = msg.decode("utf-8").split(",")  destino = [int(coords[0]), int(coords[1])]  coordenada\_objetivo\_procesada = True  print(f"Coordenada objetivo recibida y procesada: {destino}")  except Exception as e:  print(f"Error al procesar la coordenada objetivo: {e}")  def conectar\_mqtt():  client = MQTTClient("ESP32", BROKER, PORT)  client.set\_callback(mqtt\_callback)  client.connect()  print("Conectado al broker MQTT")  # Suscribirse a los topics  client.subscribe(TOPIC\_ILUMINACION)  client.subscribe(TOPIC\_OBJETIVO)  print(f"Suscrito a {TOPIC\_ILUMINACION.decode()} y {TOPIC\_OBJETIVO.decode()}")  return client  def hilo\_estado\_robot():  global orientacion\_actual, sensores, client  while True:  try:  # Leer distancia frontal  distancia\_frontal, \_ = sensores.leer\_distancias()  # Determinar estado (ajusta esto a tu lógica real)  estado = "andando" if motores.en\_movimiento() else "detenido"  # Crear mensaje JSON  mensaje = ujson.dumps({  "angle": orientacion\_actual,  "status": estado,  "distance": distancia\_frontal  })  # Publicar mensaje MQTT  client.publish(b"robot/estado", mensaje)  print("[HILO] Estado publicado:", mensaje)  # Esperar 1 segundo  time.sleep(1)  except Exception as e:  print("[HILO] Error:", e)  time.sleep(2)  def girar\_90\_grados(direccion):  global yaw\_inicial  # Configuración de ángulo objetivo  target\_yaw = (yaw\_inicial - angulo\_giro) % 360 if direccion == "derecha" else (yaw\_inicial + angulo\_giro) % 360  margen\_error = 2 # Margen aceptable de error final (grados)  pre\_stop\_margin = 9 # Grados antes del objetivo para giro inverso  tiempo\_giro\_inverso = 100 # Duración del giro inverso en ms  print(f"[GIRO] Inicio: {yaw\_inicial:.1f}° | Target: {target\_yaw:.1f}°")  # 1. Giro principal  if direccion == "derecha":  motores.girar\_derecha(velocidad\_giro)  else:  motores.girar\_izquierda(velocidad\_giro)  # Giro principal con detección de proximidad al objetivo  while True:  yaw\_actual = mpu.actualizarYaw()  error = target\_yaw - yaw\_actual  # Normalización del error  if error > 180:  error -= 360  elif error < -180:  error += 360  # Iniciar giro inverso justo antes de alcanzar el objetivo  if abs(error) <= pre\_stop\_margin:  print(f"[GIRO] Dentro del margen pre-stop ({pre\_stop\_margin}°). Iniciando giro inverso.")  motores.detener\_motores()  if direccion == "derecha":  motores.girar\_izquierda(400) # Giro inverso a velocidad mínima  else:  motores.girar\_derecha(400)  time.sleep\_ms(tiempo\_giro\_inverso)  motores.detener\_motores()  break  time.sleep\_ms(10)  # Esperar estabilización tras detener motores  time.sleep\_ms(200)  # Leer el ángulo final para actualizar la referencia  yaw\_inicial = mpu.actualizarYaw()  print(f"[GIRO] Completado. Yaw final: {yaw\_inicial:.1f}°")  time.sleep\_ms(300)    def actualizar\_orientacion(direccion\_giro):  global orientacion\_actual  orientaciones = ["NORTE", "ESTE", "SUR", "OESTE"]  indice\_actual = orientaciones.index(orientacion\_actual)  if direccion\_giro == "derecha":  orientacion\_actual = orientaciones[(indice\_actual + 1) % 4]  elif direccion\_giro == "izquierda":  orientacion\_actual = orientaciones[(indice\_actual - 1) % 4]  elif direccion\_giro == "180":  orientacion\_actual = orientaciones[(indice\_actual + 2) % 4]  print(f"[ORIENTACIÓN] Actualizada a {orientacion\_actual}")  def moverse\_a\_celda(destino):  global yaw\_inicial  yaw\_inicial = mpu.actualizarYaw()  print(f"[MOVIMIENTO] Yaw inicial actualizado: {yaw\_inicial:.2f}°")  global posicion\_actual, orientacion\_actual  destino\_x, destino\_y = destino  actual\_x, actual\_y = posicion\_actual  print(f"[MOVIMIENTO] De celda {posicion\_actual} a celda {destino}")  # Movimiento en el eje Y (adelante o atrás)  if destino\_y != actual\_y:  distancia\_requerida\_y = abs(destino\_y - actual\_y) \* celda  direccion\_y = "adelante" if destino\_y > actual\_y else "atras"  print(f"[EJE Y] Movimiento {direccion\_y} {distancia\_requerida\_y} mm")  if direccion\_y == "adelante" and orientacion\_actual == "NORTE":  # Leer la distancia inicial desde el sensor frontal  distancia\_frontal, \_ = sensores.leer\_distancias()  print(f"distancia\_frontal: {distancia\_frontal :.3f}")  distancia\_inicial = distancia\_frontal  print(f"distancia\_incial: {distancia\_inicial :.3f}")  distancia\_objetivo = distancia\_inicial - distancia\_requerida\_y  print(f"distancia\_objetivo: {distancia\_objetivo :.3f}")  motores.mover\_adelante(velocidad\_inicial)  while True:  distancia\_frontal, \_ = sensores.leer\_distancias() # Actualiza la lectura frontal  yaw\_actual = mpu.actualizarYaw()  # Calcular error y ajuste proporcional  error = yaw\_inicial - yaw\_actual  ajuste = pid.calcular\_ajuste(error)  print(f"[PID] Yaw inicial: {yaw\_inicial:.2f}° | Yaw actual: {yaw\_actual:.2f}° | Error: {error:.2f}° | Ajuste: {ajuste:.2f}")  # Ajustar velocidades de las ruedas  if error > 0:  velocidad\_izquierda = max(min(velocidad\_final - ajuste, 900), 250)  velocidad\_derecha = max(min(velocidad\_final + ajuste, 900), 250)  else:  velocidad\_derecha = max(min(velocidad\_final + ajuste, 900), 250)  velocidad\_izquierda = max(min(velocidad\_final + ajuste, 900), 250)  print(f"[VELOCIDADES] Izquierda: {velocidad\_izquierda:.2f} | Derecha: {velocidad\_derecha:.2f}")  motores.mover\_ruedas(velocidad\_izquierda, velocidad\_derecha)  # Verificar si se alcanzó el objetivo  if distancia\_frontal <= distancia\_objetivo:  break  time.sleep\_ms(20)  motores.detener\_motores()  # Movimiento en el eje X (derecha o izquierda)  if destino\_x != actual\_x:  distancia\_requerida\_x = abs(destino\_x - actual\_x) \* celda  direccion\_x = "derecha" if destino\_x > actual\_x else "izquierda"  print(f"[EJE X] Movimiento {direccion\_x} {distancia\_requerida\_x} mm")  # Girar en la dirección necesaria  if direccion\_x == "derecha" and orientacion\_actual != "ESTE":  print("[MOVIMIENTO] Girando a la derecha para alinearse con el eje X.")  girar\_90\_grados("derecha")  actualizar\_orientacion("derecha")  elif direccion\_x == "izquierda" and orientacion\_actual != "OESTE":  print("[MOVIMIENTO] Girando a la izquierda para alinearse con el eje X.")  girar\_90\_grados("izquierda")  actualizar\_orientacion("izquierda")    # Leer la distancia inicial desde el sensor frontal  distancia\_frontal, \_ = sensores.leer\_distancias()  print(f"distancia\_frontal: {distancia\_frontal :.3f}")  distancia\_inicial = distancia\_frontal  print(f"distancia\_inicial: {distancia\_inicial :.3f}")  distancia\_objetivo = distancia\_inicial - distancia\_requerida\_x  print(f"distancia\_objetivo: {distancia\_objetivo :.3f}")  motores.mover\_adelante(velocidad\_inicial)    while True:  distancia\_frontal, \_ = sensores.leer\_distancias()  yaw\_actual = mpu.actualizarYaw()  # Calcular error y ajuste proporcional  error = yaw\_inicial - yaw\_actual  ajuste = pid.calcular\_ajuste(error)  print(f"[PID] Yaw inicial: {yaw\_inicial:.2f}° | Yaw actual: {yaw\_actual:.2f}° | Error: {error:.2f}° | Ajuste: {ajuste:.2f}")  # Ajustar velocidades de las ruedas  if error > 0:  velocidad\_izquierda = max(min(velocidad\_final - ajuste, 900), 250)  velocidad\_derecha = max(min(velocidad\_final + ajuste, 900), 250)  else:  velocidad\_derecha = max(min(velocidad\_final + ajuste, 900), 250)  velocidad\_izquierda = max(min(velocidad\_final + ajuste, 900), 250)  print(f"[VELOCIDADES] Izquierda: {velocidad\_izquierda:.2f} | Derecha: {velocidad\_derecha:.2f}")  motores.mover\_ruedas(velocidad\_izquierda, velocidad\_derecha)  print(f"distancia\_frontal: {distancia\_frontal :.3f}")  # Verificar si se alcanzó el objetivo  if distancia\_frontal <= distancia\_objetivo:  break  time.sleep\_ms(20)  motores.detener\_motores()  # Actualizar posición actual  posicion\_actual = [destino\_x, destino\_y]  print(f"[MOVIMIENTO] Posición actualizada a {posicion\_actual}")  # Bucle principal  try:  print("Iniciando movimiento inicial...")  yaw\_inicial = mpu.actualizarYaw()  print(f"Yaw inicial establecido en: {yaw\_inicial:.2f}°")  conectar\_wifi()  client = conectar\_mqtt()  \_thread.start\_new\_thread(hilo\_estado\_robot, ())  while True:  # Aquí se define el destino deseado  client.check\_msg() # Revisar mensajes MQTT  # Esperar hasta recibir ambas coordenadas  if not (coordenada\_detectada\_procesada and coordenada\_objetivo\_procesada):  print("Esperando ambas coordenadas...")  time.sleep(0.1)  continue    #destino = [8, 6] # Ejemplo: ir a la celda (x, y)  moverse\_a\_celda(destino)  # Reiniciar flags para esperar nuevas coordenadas  coordenada\_detectada\_procesada = False  coordenada\_objetivo\_procesada = False  # Pausa para detener el programa (puedes cambiar esta lógica para nuevas instrucciones)  time.sleep(5)    except KeyboardInterrupt:  print("\nDeteniendo motores...")  motores.detener\_motores()  except Exception as e:  print(f"Error crítico: {e}")  machine.reset() |

Motores.py

|  |
| --- |
| from machine import Pin, PWM  import time  class Motores:  def \_init\_(self):  # Primero configurar los pines como salidas digitales en LOW  self.Motor\_A1 = Pin(2, Pin.OUT, value=0)  self.Motor\_A2 = Pin(4, Pin.OUT, value=0)  self.Motor\_B1 = Pin(17, Pin.OUT, value=0)  self.Motor\_B2 = Pin(19, Pin.OUT, value=0)    # Pequeña pausa para estabilización  time.sleep\_ms(20)    # Ahora convertirlos a PWM  self.Motor\_A1 = PWM(self.Motor\_A1)  self.Motor\_A2 = PWM(self.Motor\_A2)  self.Motor\_B1 = PWM(self.Motor\_B1)  self.Motor\_B2 = PWM(self.Motor\_B2)    # Configurar frecuencia PWM  self.Motor\_A1.freq(1000)  self.Motor\_A2.freq(1000)  self.Motor\_B1.freq(1000)  self.Motor\_B2.freq(1000)    # Asegurar que todos los PWM empiecen en 0  self.detener\_motores()  time.sleep\_ms(20)  # Estado inicial  self.estado\_actual = "detenido"  def mover\_adelante(self, velocidad):  """  Mueve el robot hacia adelante a una velocidad uniforme.  :param velocidad: Velocidad para ambas ruedas (0-1023).  """  velocidad = max(0, min(1023, int(velocidad)))  self.Motor\_A1.duty(0)  self.Motor\_A2.duty(velocidad)  self.Motor\_B1.duty(velocidad)  self.Motor\_B2.duty(0)  self.estado\_actual = "andando"  def girar\_izquierda(self, velocidad):  """  Gira el robot hacia la izquierda.  :param velocidad: Velocidad del giro.  """  velocidad = max(0, min(1023, int(velocidad)))  self.Motor\_A1.duty(velocidad)  self.Motor\_A2.duty(0)  self.Motor\_B1.duty(velocidad)  self.Motor\_B2.duty(0)  self.estado\_actual = "girando"  def girar\_derecha(self, velocidad):  """  Gira el robot hacia la derecha.  :param velocidad: Velocidad del giro.  """  velocidad = max(0, min(1023, int(velocidad)))  self.Motor\_A1.duty(0)  self.Motor\_A2.duty(velocidad)  self.Motor\_B1.duty(0)  self.Motor\_B2.duty(velocidad)  self.estado\_actual = "girando"  def detener\_motores(self):  """  Detiene ambos motores.  """  self.Motor\_A1.duty(0)  self.Motor\_A2.duty(0)  self.Motor\_B1.duty(0)  self.Motor\_B2.duty(0)  self.estado\_actual = "detenido"  def mover\_ruedas(self, velocidad\_izquierda, velocidad\_derecha):  """  Ajusta las velocidades de las ruedas izquierda y derecha.  :param velocidad\_izquierda: Velocidad para la rueda izquierda.  :param velocidad\_derecha: Velocidad para la rueda derecha.  """  velocidad\_izquierda = max(0, min(1023, int(velocidad\_izquierda)))  velocidad\_derecha = max(0, min(1023, int(velocidad\_derecha)))  self.Motor\_A1.duty(0)  self.Motor\_A2.duty(velocidad\_izquierda)  self.Motor\_B1.duty(velocidad\_derecha)  self.Motor\_B2.duty(0)  self.estado\_actual = "andando"  def en\_movimiento(self):  """  Retorna True si el robot está en movimiento.  """  return self.estado\_actual != "detenido" |

MPU6050.py

|  |
| --- |
| from machine import I2C, Pin  import time  class MPU6050:  def \_init\_(self):  self.i2c = I2C(0, sda=Pin(21), scl=Pin(22), freq=400000)  self.address = 0x68  devices = self.i2c.scan()  if self.address not in devices:  raise RuntimeError(f"MPU6050 no encontrado en la dirección {hex(self.address)}")  self.i2c.writeto\_mem(self.address, 0x6B, bytes([0])) # Despertar  self.i2c.writeto\_mem(self.address, 0x1B, bytes([0x00])) # Giro ±250°/s  self.i2c.writeto\_mem(self.address, 0x1C, bytes([0x00])) # Acel ±2g  self.yaw = 0.0  self.last\_time = time.ticks\_ms()  self.offset\_z = 0.0  def LeerRegistro(self, Direccion):  try:  data = self.i2c.readfrom\_mem(self.address, Direccion, 2)  Registro = (data[0] << 8) | data[1]  if Registro > 32767:  Registro -= 65536  return Registro  except OSError as e:  print(f"Error al leer el registro {hex(Direccion)}: {e}")  return 0  def LeerGiroscopio(self):  try:  Gx = self.LeerRegistro(0x43) / 131.0  Gy = self.LeerRegistro(0x45) / 131.0  Gz = (self.LeerRegistro(0x47) / 131.0) - self.offset\_z  return (Gx, Gy, Gz)  except OSError as e:  print(f"Error al leer el giroscopio: {e}")  return (0, 0, 0)  def calibrarGiroscopio(self, muestras=300):  print("Calibrando giroscopio... No muevas el robot.")  offset\_z = 0.0  for \_ in range(muestras):  Gx, Gy, Gz = self.LeerGiroscopio()  offset\_z += Gz  time.sleep\_ms(10)  self.offset\_z = offset\_z / muestras  print(f"Calibración completada. Offset Z: {self.offset\_z:.2f}°/s")  def actualizarYaw(self):  try:  current\_time = time.ticks\_ms()  dt = (current\_time - self.last\_time) / 1000.0  self.last\_time = current\_time    Gz = self.LeerGiroscopio()[2]  # --- Añade este filtro de ruido ---  if abs(Gz) < 0.5: # Ignorar ruido pequeño  Gz = 0.0    self.yaw += Gz \* dt  # --- Normalización opcional ---      return self.yaw  except OSError as e:  print(f"Error al actualizar el yaw: {e}")  return 0.0 |

pid\_controller.py

|  |
| --- |
| class PIDController:  def \_init\_(self, kp):  """  Inicializa el controlador PID.  :param kp: Constante de ganancia proporcional.  """  self.kp = kp  def calcular\_ajuste(self, error):  """  Calcula el ajuste proporcional basado en el error.  :param error: Diferencia entre el valor objetivo y el valor actual.  :return: Ajuste calculado.  """  ajuste = self.kp \* error  return ajuste |

sensores.py

|  |
| --- |
| from machine import Pin, I2C  from vl53l0x import VL53L0X  import time  class SensoresLaser:  def \_init\_(self):  # Configuración de pines XSHUT para cada sensor  self.XSHUT\_PINS = [15, 16] # Pines XSHUT para el sensor frontal (15) y sensor izquierdo (13)  self.SENSOR\_COUNT = len(self.XSHUT\_PINS)  # Inicialización del bus I2C  self.i2c = I2C(0, sda=Pin(21), scl=Pin(22))  # Lista para almacenar los objetos de los sensores  self.sensors = []  # Configuración inicial de los sensores  for i in range(self.SENSOR\_COUNT):  address = 0x30 + i  sensor = self.setup\_sensor(self.XSHUT\_PINS[i], address)  self.sensors.append(sensor)  def setup\_sensor(self, xshut\_pin, address):  # Apagar el sensor  xshut = Pin(xshut\_pin, Pin.OUT)  xshut.value(0)  time.sleep\_ms(10)  # Encender el sensor  xshut.value(1)  time.sleep\_ms(10)  # Inicializar el sensor con la dirección predeterminada (0x29)  sensor = VL53L0X(self.i2c, address=0x29)  sensor.start()  # Cambiar la dirección I2C del sensor  sensor.change\_address(address)  return sensor  def leer\_distancias(self):  # Leer las distancias de los sensores  distancia\_frontal = self.sensors[0].read()  distancia\_izquierda = self.sensors[1].read() + 55  return distancia\_frontal, distancia\_izquierda |

Vl53l0x.py

|  |
| --- |
| from micropython import const  import ustruct  import utime  from machine import Timer  import time  \_IO\_TIMEOUT = 1000  \_SYSRANGE\_START = const(0x00)  \_EXTSUP\_HV = const(0x89)  \_MSRC\_CONFIG = const(0x60)  \_FINAL\_RATE\_RTN\_LIMIT = const(0x44)  \_SYSTEM\_SEQUENCE = const(0x01)  \_SPAD\_REF\_START = const(0x4f)  \_SPAD\_ENABLES = const(0xb0)  \_REF\_EN\_START\_SELECT = const(0xb6)  \_SPAD\_NUM\_REQUESTED = const(0x4e)  \_INTERRUPT\_GPIO = const(0x0a)  \_INTERRUPT\_CLEAR = const(0x0b)  \_GPIO\_MUX\_ACTIVE\_HIGH = const(0x84)  \_RESULT\_INTERRUPT\_STATUS = const(0x13)  \_RESULT\_RANGE\_STATUS = const(0x14)  \_OSC\_CALIBRATE = const(0xf8)  \_MEASURE\_PERIOD = const(0x04)  SYSRANGE\_START = 0x00  SYSTEM\_THRESH\_HIGH = 0x0C  SYSTEM\_THRESH\_LOW = 0x0E  SYSTEM\_SEQUENCE\_CONFIG = 0x01  SYSTEM\_RANGE\_CONFIG = 0x09  SYSTEM\_INTERMEASUREMENT\_PERIOD = 0x04  SYSTEM\_INTERRUPT\_CONFIG\_GPIO = 0x0A  GPIO\_HV\_MUX\_ACTIVE\_HIGH = 0x84  SYSTEM\_INTERRUPT\_CLEAR = 0x0B  RESULT\_INTERRUPT\_STATUS = 0x13  RESULT\_RANGE\_STATUS = 0x14  RESULT\_CORE\_AMBIENT\_WINDOW\_EVENTS\_RTN = 0xBC  RESULT\_CORE\_RANGING\_TOTAL\_EVENTS\_RTN = 0xC0  RESULT\_CORE\_AMBIENT\_WINDOW\_EVENTS\_REF = 0xD0  RESULT\_CORE\_RANGING\_TOTAL\_EVENTS\_REF = 0xD4  RESULT\_PEAK\_SIGNAL\_RATE\_REF = 0xB6  ALGO\_PART\_TO\_PART\_RANGE\_OFFSET\_MM = 0x28  I2C\_SLAVE\_DEVICE\_ADDRESS = 0x8A  MSRC\_CONFIG\_CONTROL = 0x60  PRE\_RANGE\_CONFIG\_MIN\_SNR = 0x27  PRE\_RANGE\_CONFIG\_VALID\_PHASE\_LOW = 0x56  PRE\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH = 0x57  PRE\_RANGE\_MIN\_COUNT\_RATE\_RTN\_LIMIT = 0x64  FINAL\_RANGE\_CONFIG\_MIN\_SNR = 0x67  FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_LOW = 0x47  FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH = 0x48  FINAL\_RANGE\_CONFIG\_MIN\_COUNT\_RATE\_RTN\_LIMIT = 0x44  PRE\_RANGE\_CONFIG\_SIGMA\_THRESH\_HI = 0x61  PRE\_RANGE\_CONFIG\_SIGMA\_THRESH\_LO = 0x62  PRE\_RANGE\_CONFIG\_VCSEL\_PERIOD = 0x50  PRE\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_HI = 0x51  PRE\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_LO = 0x52  SYSTEM\_HISTOGRAM\_BIN = 0x81  HISTOGRAM\_CONFIG\_INITIAL\_PHASE\_SELECT = 0x33  HISTOGRAM\_CONFIG\_READOUT\_CTRL = 0x55  FINAL\_RANGE\_CONFIG\_VCSEL\_PERIOD = 0x70  FINAL\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_HI = 0x71  FINAL\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_LO = 0x72  CROSSTALK\_COMPENSATION\_PEAK\_RATE\_MCPS = 0x20  MSRC\_CONFIG\_TIMEOUT\_MACROP = 0x46  SOFT\_RESET\_GO2\_SOFT\_RESET\_N = 0xBF  IDENTIFICATION\_MODEL\_ID = 0xC0  IDENTIFICATION\_REVISION\_ID = 0xC2  OSC\_CALIBRATE\_VAL = 0xF8  GLOBAL\_CONFIG\_VCSEL\_WIDTH = 0x32  GLOBAL\_CONFIG\_SPAD\_ENABLES\_REF\_0 = 0xB0  GLOBAL\_CONFIG\_SPAD\_ENABLES\_REF\_1 = 0xB1  GLOBAL\_CONFIG\_SPAD\_ENABLES\_REF\_2 = 0xB2  GLOBAL\_CONFIG\_SPAD\_ENABLES\_REF\_3 = 0xB3  GLOBAL\_CONFIG\_SPAD\_ENABLES\_REF\_4 = 0xB4  GLOBAL\_CONFIG\_SPAD\_ENABLES\_REF\_5 = 0xB5  GLOBAL\_CONFIG\_REF\_EN\_START\_SELECT = 0xB6  DYNAMIC\_SPAD\_NUM\_REQUESTED\_REF\_SPAD = 0x4E  DYNAMIC\_SPAD\_REF\_EN\_START\_OFFSET = 0x4F  POWER\_MANAGEMENT\_GO1\_POWER\_FORCE = 0x80  VHV\_CONFIG\_PAD\_SCL\_SDA\_\_EXTSUP\_HV = 0x89  ALGO\_PHASECAL\_LIM = 0x30  ALGO\_PHASECAL\_CONFIG\_TIMEOUT = 0x30  class TimeoutError(RuntimeError):  pass  class VL53L0X:  def \_init\_(self, i2c, address=0x29):  self.i2c = i2c  self.address = address  self.init()  self.\_started = False  self.measurement\_timing\_budget\_us = 0  self.set\_measurement\_timing\_budget(self.measurement\_timing\_budget\_us)  self.enables = {"tcc": 0,  "dss": 0,  "msrc": 0,  "pre\_range": 0,  "final\_range": 0}  self.timeouts = {"pre\_range\_vcsel\_period\_pclks": 0,  "msrc\_dss\_tcc\_mclks": 0,  "msrc\_dss\_tcc\_us": 0,  "pre\_range\_mclks": 0,  "pre\_range\_us": 0,  "final\_range\_vcsel\_period\_pclks": 0,  "final\_range\_mclks": 0,  "final\_range\_us": 0  }  self.vcsel\_period\_type = ["VcselPeriodPreRange", "VcselPeriodFinalRange"]  def \_registers(self, register, values=None, struct='B'):  if values is None:  size = ustruct.calcsize(struct)  data = self.i2c.readfrom\_mem(self.address, register, size)  values = ustruct.unpack(struct, data)  return values  data = ustruct.pack(struct, \*values)  self.i2c.writeto\_mem(self.address, register, data)  def \_register(self, register, value=None, struct='B'):  if value is None:  return self.\_registers(register, struct=struct)[0]  self.\_registers(register, (value,), struct=struct)  def \_flag(self, register=0x00, bit=0, value=None):  data = self.\_register(register)  mask = 1 << bit  if value is None:  return bool(data & mask)  elif value:  data |= mask  else:  data &= ~mask  self.\_register(register, data)  def \_config(self, \*config):  for register, value in config:  self.\_register(register, value)  def init(self, power2v8=True):  self.\_flag(\_EXTSUP\_HV, 0, power2v8)  # I2C standard mode  self.\_config(  (0x88, 0x00),  (0x80, 0x01),  (0xff, 0x01),  (0x00, 0x00),  )  self.\_stop\_variable = self.\_register(0x91)  self.\_config(  (0x00, 0x01),  (0xff, 0x00),  (0x80, 0x00),  )  # disable signal\_rate\_msrc and signal\_rate\_pre\_range limit checks  self.\_flag(\_MSRC\_CONFIG, 1, True)  self.\_flag(\_MSRC\_CONFIG, 4, True)  # rate\_limit = 0.25  self.\_register(\_FINAL\_RATE\_RTN\_LIMIT, int(0.1 \* (1 << 7)),  struct='>H')  self.\_register(\_SYSTEM\_SEQUENCE, 0xff)  spad\_count, is\_aperture = self.\_spad\_info()  spad\_map = bytearray(self.\_registers(\_SPAD\_ENABLES, struct='6B'))  # set reference spads  self.\_config(  (0xff, 0x01),  (\_SPAD\_REF\_START, 0x00),  (\_SPAD\_NUM\_REQUESTED, 0x2c),  (0xff, 0x00),  (\_REF\_EN\_START\_SELECT, 0xb4),  )  spads\_enabled = 0  for i in range(48):  if i < 12 and is\_aperture or spads\_enabled >= spad\_count:  spad\_map[i // 8] &= ~(1 << (i >> 2))  elif spad\_map[i // 8] & (1 << (i >> 2)):  spads\_enabled += 1  self.\_registers(\_SPAD\_ENABLES, spad\_map, struct='6B')  self.\_config(  (0xff, 0x01),  (0x00, 0x00),  (0xff, 0x00),  (0x09, 0x00),  (0x10, 0x00),  (0x11, 0x00),  (0x24, 0x01),  (0x25, 0xFF),  (0x75, 0x00),  (0xFF, 0x01),  (0x4E, 0x2C),  (0x48, 0x00),  (0x30, 0x20),  (0xFF, 0x00),  (0x30, 0x09),  (0x54, 0x00),  (0x31, 0x04),  (0x32, 0x03),  (0x40, 0x83),  (0x46, 0x25),  (0x60, 0x00),  (0x27, 0x00),  (0x50, 0x06),  (0x51, 0x00),  (0x52, 0x96),  (0x56, 0x08),  (0x57, 0x30),  (0x61, 0x00),  (0x62, 0x00),  (0x64, 0x00),  (0x65, 0x00),  (0x66, 0xA0),  (0xFF, 0x01),  (0x22, 0x32),  (0x47, 0x14),  (0x49, 0xFF),  (0x4A, 0x00),  (0xFF, 0x00),  (0x7A, 0x0A),  (0x7B, 0x00),  (0x78, 0x21),  (0xFF, 0x01),  (0x23, 0x34),  (0x42, 0x00),  (0x44, 0xFF),  (0x45, 0x26),  (0x46, 0x05),  (0x40, 0x40),  (0x0E, 0x06),  (0x20, 0x1A),  (0x43, 0x40),  (0xFF, 0x00),  (0x34, 0x03),  (0x35, 0x44),  (0xFF, 0x01),  (0x31, 0x04),  (0x4B, 0x09),  (0x4C, 0x05),  (0x4D, 0x04),  (0xFF, 0x00),  (0x44, 0x00),  (0x45, 0x20),  (0x47, 0x08),  (0x48, 0x28),  (0x67, 0x00),  (0x70, 0x04),  (0x71, 0x01),  (0x72, 0xFE),  (0x76, 0x00),  (0x77, 0x00),  (0xFF, 0x01),  (0x0D, 0x01),  (0xFF, 0x00),  (0x80, 0x01),  (0x01, 0xF8),  (0xFF, 0x01),  (0x8E, 0x01),  (0x00, 0x01),  (0xFF, 0x00),  (0x80, 0x00),  )  self.\_register(\_INTERRUPT\_GPIO, 0x04)  self.\_flag(\_GPIO\_MUX\_ACTIVE\_HIGH, 4, False)  self.\_register(\_INTERRUPT\_CLEAR, 0x01)  # XXX Need to implement this.  # budget = self.\_timing\_budget()  # self.\_register(\_SYSTEM\_SEQUENCE, 0xe8)  # self.\_timing\_budget(budget)  self.\_register(\_SYSTEM\_SEQUENCE, 0x01)  self.\_calibrate(0x40)  self.\_register(\_SYSTEM\_SEQUENCE, 0x02)  self.\_calibrate(0x00)  self.\_register(\_SYSTEM\_SEQUENCE, 0xe8)  def \_spad\_info(self):  self.\_config(  (0x80, 0x01),  (0xff, 0x01),  (0x00, 0x00),  (0xff, 0x06),  )  self.\_flag(0x83, 3, True)  self.\_config(  (0xff, 0x07),  (0x81, 0x01),  (0x80, 0x01),  (0x94, 0x6b),  (0x83, 0x00),  )  for timeout in range(\_IO\_TIMEOUT):  if self.\_register(0x83):  break  utime.sleep\_ms(1)  else:  raise TimeoutError()  self.\_config(  (0x83, 0x01),  )  value = self.\_register(0x92)  self.\_config(  (0x81, 0x00),  (0xff, 0x06),  )  self.\_flag(0x83, 3, False)  self.\_config(  (0xff, 0x01),  (0x00, 0x01),  (0xff, 0x00),  (0x80, 0x00),  )  count = value & 0x7f  is\_aperture = bool(value & 0b10000000)  return count, is\_aperture  def \_calibrate(self, vhv\_init\_byte):  self.\_register(\_SYSRANGE\_START, 0x01 | vhv\_init\_byte)  for timeout in range(\_IO\_TIMEOUT):  if self.\_register(\_RESULT\_INTERRUPT\_STATUS) & 0x07:  break  utime.sleep\_ms(1)  else:  raise TimeoutError()  self.\_register(\_INTERRUPT\_CLEAR, 0x01)  self.\_register(\_SYSRANGE\_START, 0x00)  def start(self, period=0):  self.\_config(  (0x80, 0x01),  (0xFF, 0x01),  (0x00, 0x00),  (0x91, self.\_stop\_variable),  (0x00, 0x01),  (0xFF, 0x00),  (0x80, 0x00),  )  if period:  oscilator = self.\_register(\_OSC\_CALIBRATE, struct='>H')  if oscilator:  period \*= oscilator  self.\_register(\_MEASURE\_PERIOD, period, struct='>H')  self.\_register(\_SYSRANGE\_START, 0x04)  else:  self.\_register(\_SYSRANGE\_START, 0x02)  self.\_started = True  def stop(self):  self.\_register(\_SYSRANGE\_START, 0x01)  self.\_config(  (0xFF, 0x01),  (0x00, 0x00),  (0x91, self.\_stop\_variable),  (0x00, 0x01),  (0xFF, 0x00),  )  self.\_started = False  def read(self):  if not self.\_started:  self.\_config(  (0x80, 0x01),  (0xFF, 0x01),  (0x00, 0x00),  (0x91, self.\_stop\_variable),  (0x00, 0x01),  (0xFF, 0x00),  (0x80, 0x00),  (\_SYSRANGE\_START, 0x01),  )  for timeout in range(\_IO\_TIMEOUT):  if not self.\_register(\_SYSRANGE\_START) & 0x01:  break  utime.sleep\_ms(1)  else:  raise TimeoutError()  for timeout in range(\_IO\_TIMEOUT):  if self.\_register(\_RESULT\_INTERRUPT\_STATUS) & 0x07:  break  utime.sleep\_ms(1)  else:  raise TimeoutError()  value = self.\_register(\_RESULT\_RANGE\_STATUS + 10, struct='>H');  self.\_register(\_INTERRUPT\_CLEAR, 0x01)  return value  def set\_signal\_rate\_limit(self, limit\_Mcps):  if limit\_Mcps < 0 or limit\_Mcps > 511.99:  return False  self.\_register(0x44, limit\_Mcps \* (1 << 7))  return True  def decode\_Vcsel\_period(self, reg\_val):  return (((reg\_val) + 1) << 1)  def encode\_Vcsel\_period(self, period\_pclks):  return (((period\_pclks) >> 1) - 1)  def set\_Vcsel\_pulse\_period(self, type, period\_pclks):  vcsel\_period\_reg = self.encode\_Vcsel\_period(period\_pclks)  self.get\_sequence\_step\_enables()  self.get\_sequence\_step\_timeouts()  if type == self.vcsel\_period\_type[0]:  if period\_pclks == 12:  self.\_register(PRE\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x18)  elif period\_pclks == 14:  self.\_register(PRE\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x30)  elif period\_pclks == 16:  self.\_register(PRE\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x40)  elif period\_pclks == 18:  self.\_register(PRE\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x50)  else:  return False  self.\_register(PRE\_RANGE\_CONFIG\_VALID\_PHASE\_LOW, 0x08)  self.\_register(PRE\_RANGE\_CONFIG\_VCSEL\_PERIOD, vcsel\_period\_reg)  new\_pre\_range\_timeout\_mclks = self.timeout\_microseconds\_to\_Mclks(self.timeouts["pre\_range\_us"],  period\_pclks)  self.\_register(PRE\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_HI, self.encode\_timeout(new\_pre\_range\_timeout\_mclks))  new\_msrc\_timeout\_mclks = self.timeout\_microseconds\_to\_Mclks(self.timeouts["msrc\_dss\_tcc\_us"],  period\_pclks)  self.\_register(MSRC\_CONFIG\_TIMEOUT\_MACROP, 255 if new\_msrc\_timeout\_mclks > 256 else (new\_msrc\_timeout\_mclks - 1))  elif type == self.vcsel\_period\_type[1]:  if period\_pclks == 8:  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x10)  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_LOW, 0x08)  self.\_register(GLOBAL\_CONFIG\_VCSEL\_WIDTH, 0x02)  self.\_(ALGO\_PHASECAL\_CONFIG\_TIMEOUT, 0x0C)  self.\_register(0xFF, 0x01)  self.\_register(ALGO\_PHASECAL\_LIM, 0x30)  self.\_register(0xFF, 0x00)  elif period\_pclks == 10:  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x28)  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_LOW, 0x08)  self.\_register(GLOBAL\_CONFIG\_VCSEL\_WIDTH, 0x03)  self.\_register(ALGO\_PHASECAL\_CONFIG\_TIMEOUT, 0x09)  self.\_register(0xFF, 0x01)  self.\_register(ALGO\_PHASECAL\_LIM, 0x20)  self.\_register(0xFF, 0x00)  elif period\_pclks == 12:  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x38)  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_LOW, 0x08)  self.\_register(GLOBAL\_CONFIG\_VCSEL\_WIDTH, 0x03)  self.\_register(ALGO\_PHASECAL\_CONFIG\_TIMEOUT, 0x08)  self.\_register(0xFF, 0x01)  self.\_register(ALGO\_PHASECAL\_LIM, 0x20)  self.\_register(0xFF, 0x00)  elif period\_pclks == 14:  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_HIGH, 0x48)  self.\_register(FINAL\_RANGE\_CONFIG\_VALID\_PHASE\_LOW, 0x08)  self.\_register(GLOBAL\_CONFIG\_VCSEL\_WIDTH, 0x03)  self.\_register(ALGO\_PHASECAL\_CONFIG\_TIMEOUT, 0x07)  self.\_register(0xFF, 0x01)  self.\_register(ALGO\_PHASECAL\_LIM, 0x20)  self.\_register(0xFF, 0x00)  else:  return False  self.\_register(FINAL\_RANGE\_CONFIG\_VCSEL\_PERIOD, vcsel\_period\_reg)  new\_final\_range\_timeout\_mclks = self.timeout\_microseconds\_to\_Mclks(self.timeouts["final\_range\_us"], period\_pclks)  if self.enables["pre\_range"]:  new\_final\_range\_timeout\_mclks += 1  self.\_register(FINAL\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_HI, self.encode\_timeout(new\_final\_range\_timeout\_mclks))  else:  return False  self.set\_measurement\_timing\_budget(self.measurement\_timing\_budget\_us)  sequence\_config = self.\_register(SYSTEM\_SEQUENCE\_CONFIG)  self.\_register(SYSTEM\_SEQUENCE\_CONFIG, 0x02)  self.perform\_single\_ref\_calibration(0x0)  self.\_register(SYSTEM\_SEQUENCE\_CONFIG, sequence\_config)  return True  def get\_sequence\_step\_enables(self):  sequence\_config = self.\_register(0x01)  self.enables["tcc"] = (sequence\_config >> 4) & 0x1  self.enables["dss"] = (sequence\_config >> 3) & 0x1  self.enables["msrc"] = (sequence\_config >> 2) & 0x1  self.enables["pre\_range"] = (sequence\_config >> 6) & 0x1  self.enables["final\_range"] = (sequence\_config >> 7) & 0x1  def get\_vcsel\_pulse\_period(self, type):  if type == self.vcsel\_period\_type[0]:  return self.decode\_Vcsel\_period(0x50)  elif type == self.vcsel\_period\_type[1]:  return self.decode\_Vcsel\_period(0x70)  else:  return 255  def get\_sequence\_step\_timeouts(self):  self.timeouts["pre\_range\_vcsel\_period\_pclks"] = self.get\_vcsel\_pulse\_period(self.vcsel\_period\_type[0])  self.timeouts["msrc\_dss\_tcc\_mclks"] = int(self.\_register(MSRC\_CONFIG\_TIMEOUT\_MACROP)) + 1  self.timeouts["msrc\_dss\_tcc\_us"] = self.timeout\_Mclks\_to\_microseconds(self.timeouts["msrc\_dss\_tcc\_mclks"],  self.timeouts[  "pre\_range\_vcsel\_period\_pclks"])  self.timeouts["pre\_range\_mclks"] = self.decode\_timeout(PRE\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_HI)  self.timeouts["pre\_range\_us"] = self.timeout\_Mclks\_to\_microseconds(self.timeouts["pre\_range\_mclks"],  self.timeouts[  "pre\_range\_vcsel\_period\_pclks"])  self.timeouts["final\_range\_vcsel\_period\_pclks"] = self.get\_vcsel\_pulse\_period(self.vcsel\_period\_type[1])  self.timeouts["final\_range\_mclks"] = self.decode\_timeout(self.\_register(FINAL\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_HI))  if self.enables["pre\_range"]:  self.timeouts["final\_range\_mclks"] -= self.timeouts["pre\_range\_mclks"]  self.timeouts["final\_range\_us"] = self.timeout\_Mclks\_to\_microseconds(self.timeouts["final\_range\_mclks"],  self.timeouts[  "final\_range\_vcsel\_period\_pclks"])  def timeout\_Mclks\_to\_microseconds(self, timeout\_period\_mclks, vcsel\_period\_pclks):  macro\_period\_ns = self.calc\_macro\_period(vcsel\_period\_pclks)  return ((timeout\_period\_mclks \* macro\_period\_ns) + (macro\_period\_ns / 2)) / 1000  def timeout\_microseconds\_to\_Mclks(self, timeout\_period\_us, vcsel\_period\_pclks):  macro\_period\_ns = self.calc\_macro\_period(vcsel\_period\_pclks)  return (((timeout\_period\_us \* 1000) + (macro\_period\_ns / 2)) / macro\_period\_ns)  def calc\_macro\_period(self, vcsel\_period\_pclks):  return (((2304 \* (vcsel\_period\_pclks) \* 1655) + 500) / 1000)  def decode\_timeout(self, reg\_val):  return ((reg\_val & 0x00FF) << ((reg\_val & 0xFF00) >> 8)) + 1  def encode\_timeout(self, timeout\_mclks):  timeout\_mclks = int(timeout\_mclks)  ls\_byte = 0  ms\_byte = 0  if timeout\_mclks > 0:  ls\_byte = timeout\_mclks - 1  while (ls\_byte & 0xFFFFFF00) > 0:  ls\_byte >>= 1  ms\_byte += 1  return (ms\_byte << 8) or (ls\_byte & 0xFF)  else:  return 0  def set\_measurement\_timing\_budget(self, budget\_us):  start\_overhead = 1320  end\_overhead = 960  msrc\_overhead = 660  tcc\_overhead = 590  dss\_overhead = 690  pre\_range\_overhead = 660  final\_range\_overhead = 550  min\_timing\_budget = 20000  if budget\_us < min\_timing\_budget:  return False  used\_budget\_us = start\_overhead + end\_overhead  self.get\_sequence\_step\_enables()  self.get\_sequence\_step\_timeouts()  if self.enables["tcc"]:  used\_budget\_us += self.timeouts["msrc\_dss\_tcc\_us"] + tcc\_overhead  if self.enables["dss"]:  used\_budget\_us += 2\* self.timeouts["msrc\_dss\_tcc\_us"] + dss\_overhead  if self.enables["msrc"]:  used\_budget\_us += self.timeouts["msrc\_dss\_tcc\_us"] + msrc\_overhead  if self.enables["pre\_range"]:  used\_budget\_us += self.timeouts["pre\_range\_us"] + pre\_range\_overhead  if self.enables["final\_range"]:  used\_budget\_us += final\_range\_overhead  if used\_budget\_us > budget\_us:  return False  final\_range\_timeout\_us = budget\_us - used\_budget\_us  final\_range\_timeout\_mclks = self.timeout\_microseconds\_to\_Mclks(final\_range\_timeout\_us, self.timeouts["final\_range\_vcsel\_period\_pclks"])  if self.enables["pre\_range"]:  final\_range\_timeout\_mclks += self.timeouts["pre\_range\_mclks"]  self.\_register(FINAL\_RANGE\_CONFIG\_TIMEOUT\_MACROP\_HI, self.encode\_timeout(final\_range\_timeout\_mclks))  self.measurement\_timing\_budget\_us = budget\_us  return True  def perform\_single\_ref\_calibration(self, vhv\_init\_byte):  chrono = Timer.Chrono()  self.\_register(SYSRANGE\_START, 0x01|vhv\_init\_byte)  chrono.start()  while self.\_register((RESULT\_INTERRUPT\_STATUS & 0x07) == 0):  time\_elapsed = chrono.read\_ms()  if time\_elapsed > \_IO\_TIMEOUT:  return False  self.\_register(SYSTEM\_INTERRUPT\_CLEAR, 0x01)  self.\_register(SYSRANGE\_START, 0x00)    def change\_address(self, new\_address):  if new\_address < 0x08 or new\_address > 0x77:  raise ValueError("La dirección I2C debe estar entre 0x08 y 0x77.")    # Escribir la nueva dirección en el registro de dirección I2C  self.\_register(0x8A, new\_address)  self.address = new\_address # Actualizar la dirección en la instancia |