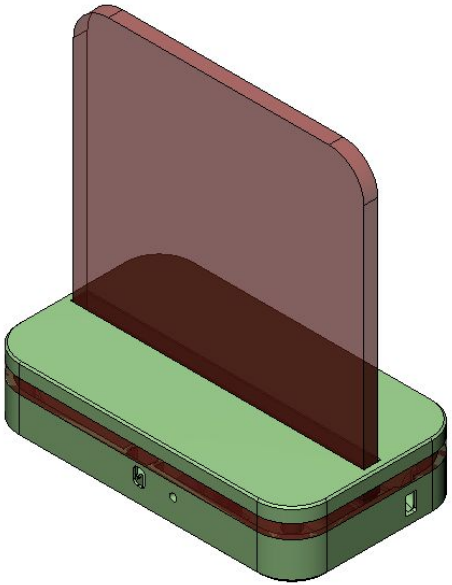
Funcionalidades

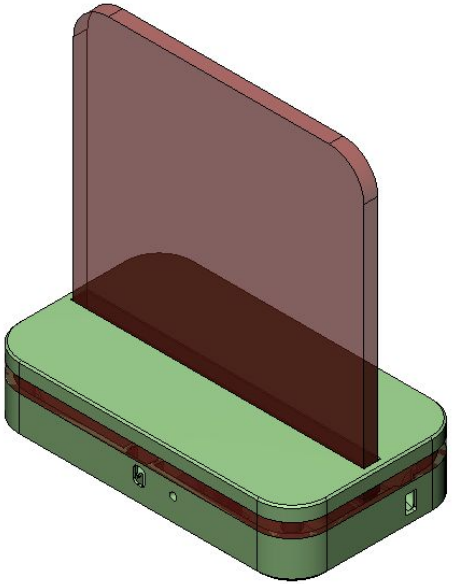
Cartel de distanciamiento



Principales:

Si se presiona el botón al inicia entra en modo calibración
Segun distancia prender leds de color , prender buzzer
Si se presiona el botón durante la ejecución cambia el estado de disparo de buzzer

Del tof
Controlar la histéresis
Filtra ruido
Determinar zona de operación



Introducción al Uso de Histéresis y Filtro Exponencial en el Sensor ToF

El código presentado implementa un sistema de distanciamiento social basado en un sensor de tiempo de vuelo (ToF) VL53L0X, que mide la distancia de un usuario y ajusta el color de LEDs para indicar niveles de peligro, precaución y seguridad. En este sistema, se han incorporado dos técnicas clave para mejorar la estabilidad y confiabilidad de las mediciones: **histéresis** y **filtro exponencial (EWMA, Exponentially Weighted Moving Average)**.

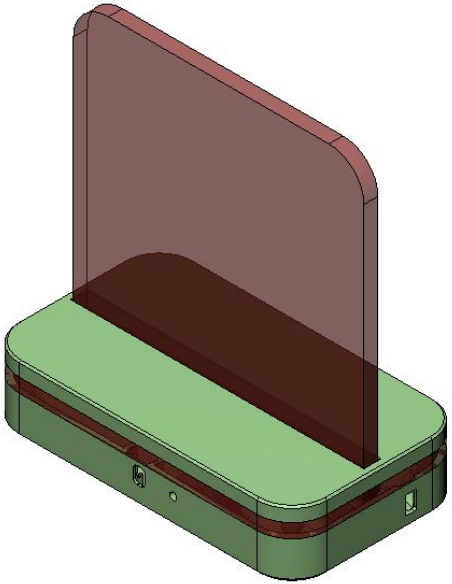
1. Uso de Histéresis

La histéresis es una técnica utilizada para evitar fluctuaciones rápidas entre estados debido a pequeñas variaciones en la medición del sensor. Sin ella, si un usuario se encuentra cerca del umbral entre dos zonas (por ejemplo, entre peligro y precaución), pequeñas variaciones en la lectura podrían causar cambios erráticos en la indicación del sistema.

En el código, la histéresis se aplica en la función `hysteresis_off()`, donde se evalúa si la medición ha descendido por debajo de un umbral ajustado para evitar cambios prematuros de estado:

Funcionalidades del tof

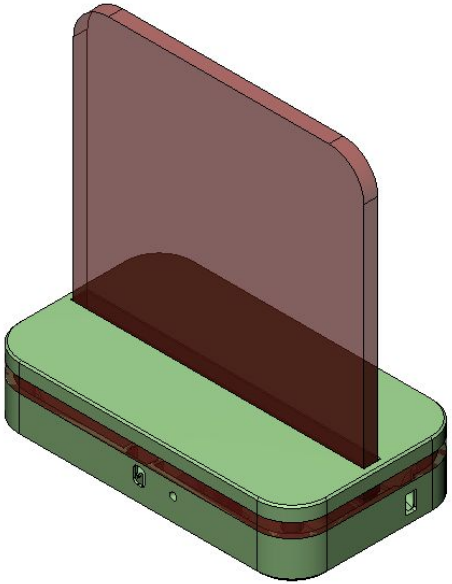
Cartel de distanciamiento



En el código, la histéresis se aplica en la función `hysteresis_off()`, donde se evalúa si la medición ha descendido por debajo de un umbral ajustado para evitar cambios prematuros de estado:

```
bool hysteresis_off( uint16_t val, uint16_t next_point )
{
    if( Config.get_hysterisis() > 0 ) {
        return ( val < (next_point - Config.get_hysterisis()) );
    }
    return false;
}
```

Este método asegura que el sistema no cambie de estado inmediatamente
cuando la distancia apenas supera un umbral, sino que se requiere una variación significativa para confirmar la transición.



2. Uso del Filtro Exponencial

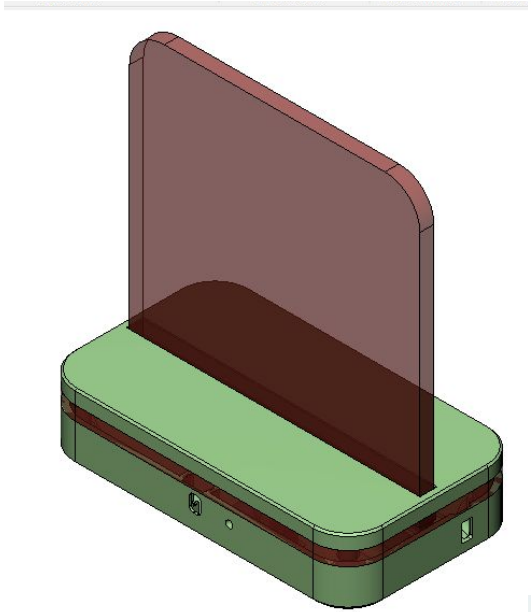
El sensor ToF puede producir lecturas con ruido, lo que podría llevar a mediciones inestables. Para mitigar este problema, se aplica un filtro exponencial de media móvil (EWMA), que suaviza las lecturas al dar más peso a valores recientes mientras mantiene información histórica.

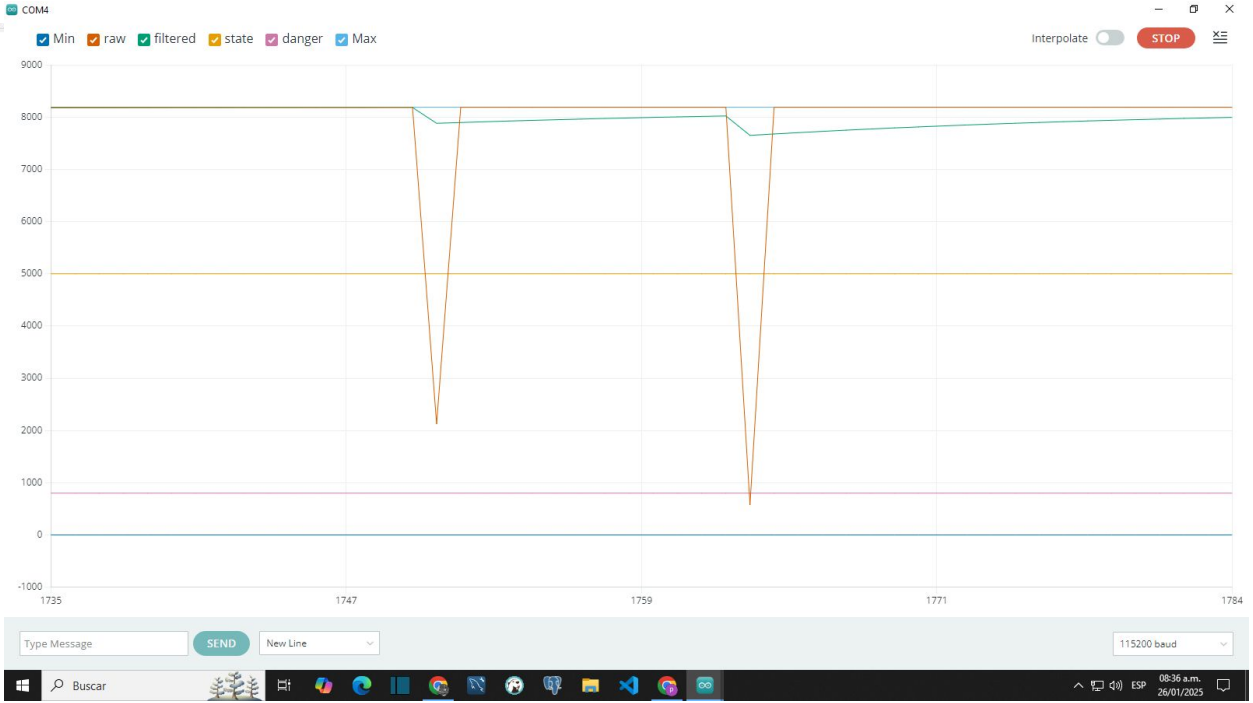
El uso conjunto de **histeresis** y **filtro exponencial** en este sistema ToF es fundamental para:

- **Reducir fluctuaciones erráticas** causadas por ruido en las mediciones.
- **Mejorar la confiabilidad de los estados de peligro, precaución y seguridad**.
- **Evitar parpadeos de LEDs y activaciones innecesarias del buzzer** cuando el usuario está cerca de los límites de las zonas definidas.

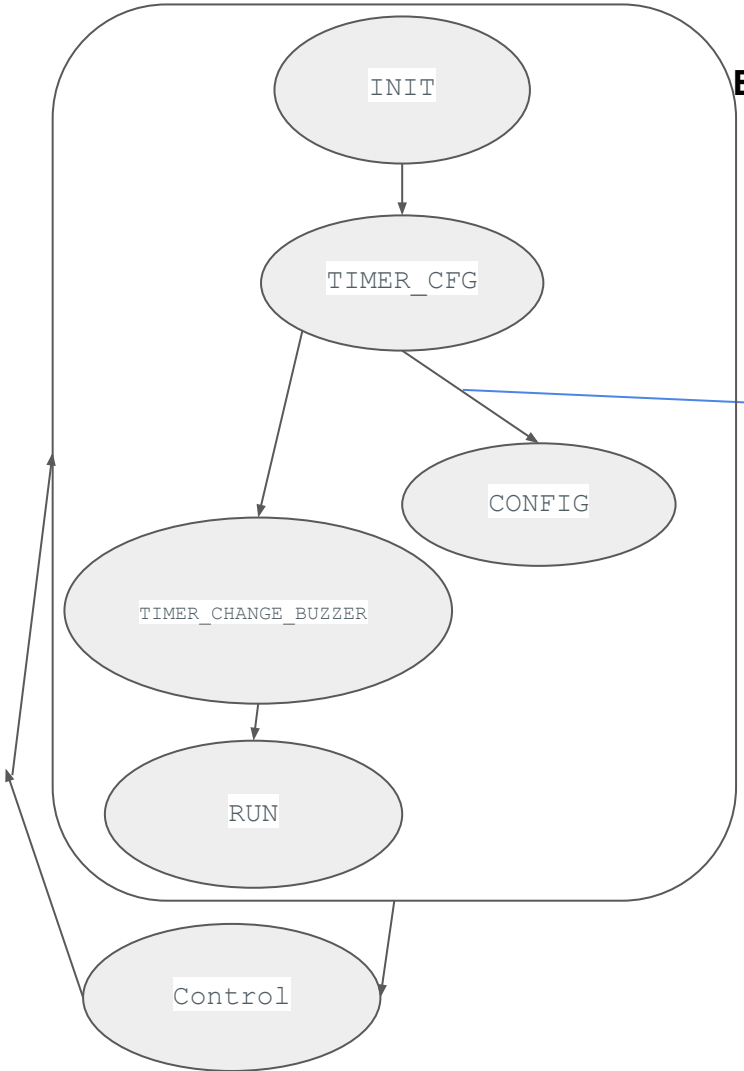
Funcionalidades del tof

Cartel de distanciamiento





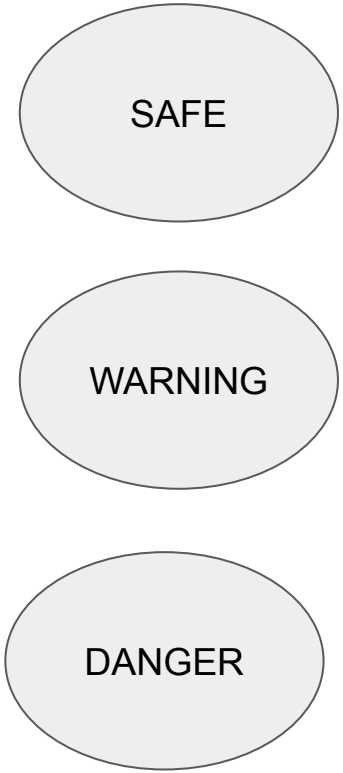
st_loop



Estados de la Máquina de Estados

1. **ST_LOOP_INIT**:
   - Se inicializa la configuración.
   - Se verifican las bandas del sistema (`check_bands()`).
   - Se ajustan los LEDs: azul si las bandas son válidas, rosado si no lo son.
2. **ST_LOOP_TIMER_CFG**:
   - Se espera a que el usuario configure la distancia de peligro.
   - Si el botón es presionado, se inicia la secuencia de configuración (`ST_LOOP_CONFIG`).
   - Si el temporizador expira (5 segundos) y las bandas son válidas, avanza al estado de control del buzzer.
3. **ST_LOOP_CONFIG**:
   - Realiza la calibración del sensor con el botón.
   - Cuando el valor del sensor está dentro del rango permitido, pasa al siguiente estado.
4. **ST_INIT_TIMER_CHANGE_BUZZER**:
   - Inicia un temporizador y avanza al estado principal (`ST_LOOP_RUN`).
5. **ST_LOOP_RUN**:
   - Monitorea el sistema.
   - Si el botón es presionado y el temporizador expira (1 segundo), invierte el estado del buzzer (`config_buzzer_on_tgl`).
   - Ejecuta la función principal de control (`control()`).

st_control



1. `ST_SAFE` (Estado Seguro):

- **Condición:**
  La distancia medida está en un rango considerado seguro.
  (Definido en los umbrales de configuración).
- **Acciones:**
  - Los LEDs muestran un color seguro (configurado por `Config.get_color_safe()`).
  - El buzzer está apagado.

2. `ST_WARNING` (Estado de Advertencia):

- **Condición:**
  La distancia medida está en un rango intermedio, que es considerado de advertencia.
  (Definido en los umbrales de configuración).
- **Acciones:**
  - Los LEDs muestran un color de advertencia (configurado por `Config.get_color_warning()`).
  - El buzzer permanece apagado.

3. `ST_DANGER` (Estado de Peligro):

- **Condición:**
  La distancia medida está dentro de un rango peligroso, que requiere alerta inmediata.
  (Definido en los umbrales de configuración).
- **Acciones:**
  - Los LEDs muestran un color de peligro (configurado por `Config.get_color_danger()`).
  - **Buzzer:**
    - Si el buzzer está habilitado (`Config.get_buzzer()`), se enciende y apaga de manera intermitente.
    - Los tiempos de encendido (`buzzer_ton`) y apagado (`buzzer_toff`) son configurables.
    - Si el buzzer no está habilitado, permanece apagado.

st_control

Gestión Adicional en los Estados

1. **Cambio de Estado Condicionado:**
   - Si el nuevo estado es más peligroso que el anterior, solo se aplica el cambio si ha expirado un temporizador (`Timer_led`), estabilizando la transición.
2. **Logs en Todos los Estados: (uso de log.cpp)**
   - Se registra información útil en los logs, incluyendo:
     - Distancia cruda y filtrada.
     - Estado actual.
     - Umbral de peligro.

Relación Entre Estados

- La transición entre estados está determinada por la distancia medida y las configuraciones de los umbrales.
- El sistema favorece mantener el estado actual si el cambio hacia un estado más peligroso ocurre muy rápidamente, evitando fluctuaciones indeseadas.

st_control

1. `hysteresis_off()`

Propósito:

Aplica una histéresis al cambio de estado para evitar fluctuaciones rápidas e indeseadas entre estados debido a pequeñas variaciones en la distancia medida.

Funcionamiento:

- **Histéresis:**
  Es una técnica que introduce un margen (o tolerancia) al cambio de estado para estabilizar las transiciones. En este caso, la histéresis es configurable a través de `Config.get_hysterisis()`.

2. `get_state()`

Propósito:

Determina el estado actual (`ST_SAFE`, `ST_WARNING`, `ST_DANGER`) comparando la distancia medida (`val`) con los umbrales configurados y aplicando histéresis para estabilizar los cambios.

Funcionamiento:

- **Entrada:**
  - `val`: Distancia medida por el sensor.
  - `last_state`: Estado anterior del sistema.

Inicialización

Modo de Operación del Sensor

Los modos de operación se controlan con macros definidas (`LONG_RANGE`, `HIGH_SPEED`, `HIGH_ACCURACY`):

1. **`LONG_RANGE`:**
   - Incrementa el alcance del sensor al reducir la sensibilidad al retorno de la señal y aumentar los pulsos del láser.
2. **`HIGH_SPEED`:**
   - Reduce el tiempo de medición para obtener resultados más rápidos (20 ms).
3. **`HIGH_ACCURACY`:**
   - Aumenta el tiempo de medición para obtener lecturas más precisas (200 ms).

5. Seguridad

- **Protección del pin `XSHUT_PIN`:**
  - El pin de "shutdown" no tolera voltajes superiores a 3.3V. Se configura cuidadosamente para evitar daños.
- **Timeouts:**
  - Si el sensor no responde en un tiempo razonable (500 ms), la operación se considera fallida.

Inicialización

tof.cpp

1. Configuración e Inicialización

Constructor: `CTof::CTof()`

- Inicializa variables internas:
  - `new_sample`: Indica si hay una nueva lectura disponible.
  - `raw`: Almacena la distancia en bruto medida por el sensor.
  - Variables específicas de los filtros (`custom_buff`, `custom_last_valid_val`, `ewma_output`).

Método `init()`

- Configura e inicializa el sensor VL53L0X:
  - Configura el pin de "shutdown" (`XSHUT_PIN`) para reiniciar el sensor correctamente.
  - Establece comunicación I2C con el sensor (`Wire.begin()`).
  - Intenta inicializar el sensor hasta 10 veces. Si falla, devuelve `false`.
  - Configura modos específicos según los preprocesadores definidos:
    - **Largo alcance:** Ajusta los parámetros para medir distancias mayores a costa de precisión.
    - **Alta velocidad:** Reduce el tiempo de medición.
    - **Alta precisión:** Aumenta el tiempo de medición para obtener datos más exactos.

tof.cpp

Flujo

Resumen del Flujo

1. **Inicialización (`init()`)**
   - Configura el sensor y establece los modos de operación.
2. **Lectura (`read()`)**
   - Obtiene una lectura no bloqueante.
   - Valida la lectura.
   - Aplica un filtro para estabilizar los datos.
3. **Obtención de Datos**
   - Usa `get_raw()` o `get_filtered()` según se necesite la lectura en bruto o suavizada.
4. **Filtrado**
   - `filter_custom()`: Elimina valores fuera de rango usando un buffer.
   - `filter_ewma()`: Suaviza las lecturas usando un filtro exponencial.

Esto permite obtener datos de distancia confiables y estables, adecuados para controlar sistemas como el de estados (`ST_SAFE`, `ST_WARNING`, `ST_DANGER`).

3. Filtrado de Lecturas

Filtro Personalizado (`filter_custom()`)

- Guarda las últimas lecturas en un buffer circular (`custom_buff`).
- Verifica si todas las lecturas del buffer están dentro de un rango permitido (diferencia máxima de 200 mm). Si alguna lectura está fuera de este rango, considera que el valor es inválido.
- Si el valor es válido, actualiza `custom_last_valid_val`, que es la salida filtrada.

Filtro EWMA (`filter_ewma(double alpha)`)

- Aplica un filtro de promedio móvil ponderado exponencial (EWMA) para suavizar las lecturas:
  - Fórmula: ewma_output=α⋅(raw−ewma_output)+ewma_output\text{ewma_output} = \alpha \cdot (\text{raw} - \text{ewma_output}) + \text{ewma_output}ewma_output=α⋅(raw−ewma_output)+ewma_output
  - `alpha`: Factor de suavizado. Un valor mayor a 0.1 da menos retardo pero más sensibilidad al ruido.

tof.cpp

Medición de tiempos de la máquina de estados

Desglose del tiempo en la fase de medición:

1. **Tiempo típico del presupuesto de rango**:
   - **33 ms**: Incluye inicialización, medición y procesamiento (*housekeeping*).
   - Este es el tiempo completo típico para una medición estándar.
2. **Tiempo de medición real**:
   - **23 ms**: Este es el tiempo que el sensor toma para emitir pulsos infrarrojos, capturar los reflejos y realizar la medición del rango.
3. **Procesamiento digital adicional**:
   - **0.8 ms**: Tiempo adicional para el procesamiento digital posterior a la medición.
4. **Tiempo mínimo para una medición de rango**:
   - **8 ms**: El sensor puede configurarse para realizar mediciones rápidas, pero esto podría reducir la precisión.
5. **Presupuesto de tiempo mínimo/máximo permitido**:
   - **20 ms** (mínimo): Esto asegura que el sensor tenga suficiente tiempo para realizar una medición precisa.
   - **5 segundos** (máximo): Permite obtener mediciones de alta precisión y alcance extendido.

Tiempo de ejecución de loop principal

loop. ino

Output Serial Monitor ×

Not connected. Select a board and a port to con

```
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 241 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 228 us
Promedio de tiempo del loop: 241 us
Promedio de tiempo del loop: 228 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 228 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 241 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 228 us
Promedio de tiempo del loop: 228 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 241 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 241 us
Promedio de tiempo del loop: 227 us
Promedio de tiempo del loop: 228 us
```

```
        tiempo_fin = micros(); // Marca el tiempo al final del loop.

        // Calcula la duración del loop en microsegundos.
        unsigned long duracion_loop = tiempo_fin - tiempo_inicio;

        // Acumula el tiempo y cuenta la muestra.
        acumulador_tiempos += duracion_loop;
        contador_muestras++;

        // Si se han tomado 100 muestras, calcula y muestra el promedio.
        if (contador_muestras == 100) {
            unsigned long promedio = acumulador_tiempos / 100;
            Serial.print("Promedio de tiempo del loop: ");
            Serial.print(promedio);
            Serial.println(" us");

            // Reinicia el acumulador y el contador.
            acumulador_tiempos = 0;
            contador_muestras = 0;
        }
    }
```

Polulo lib

Agregado a la lib de Polulo el la lectura no bloqueante


```
835     return range;
836   }
837
838   // Reads range in non-blocking, returns true when done. If the
839   // is 65535, it indicates timeout.
840   // (readRangeNoBlocking() also calls this function after start
841   // single-shot range measurement)
842   //
843   bool VL53L0X::readRangeNoBlocking(uint16_t& range)
844   {
845   bool read_end = false;
846
847     range = 0;
848
849     switch( status_no_bloking )
850     {
851       case ST_START:
852           startTimeout();
853           writeReg(0x80, 0x01);
854           writeReg(0xFF, 0x01);
855           writeReg(0x00, 0x00);
856           writeReg(0x91, stop_variable);
857           writeReg(0x00, 0x01);
858           writeReg(0xFF, 0x00);
859           writeReg(0x80, 0x00);
860
861           writeReg(SYSRANGE_START, 0x01);
862
863           status_no_bloking = ST_WAIT_START;
864       break;
865
866       case ST_WAIT_START:
867           // "Wait until start bit has been cleared"
868           if (readReg(SYSRANGE_START) & 0x01)
869           {
870               if (checkTimeoutExpired())
871               {
872                 did_timeout = true;
873                 range = 65535;
874                 read_end = true;
875                 status_no_bloking = ST_START;
876               }
877           }
878           else
879           {
880               startTimeout();
```

Polulo lib

Agregado a la lib de Polulo el la lectura no bloqueante

Resumen de la reforma:

1. Contexto:

- El propósito de la reforma es adaptar el funcionamiento del sensor para que pueda operar de manera **no bloqueante**. Esto significa que el sensor se inicia en una etapa, y luego la ejecución del código continúa sin esperar que se complete la medición. El método se ejecuta varias veces en el ciclo de `loop` hasta que se completa la medición.

2. Estructura del método:

- El método utiliza una máquina de estados con **4 estados principales**:
    1. **ST_START**: Inicia la medición de rango enviando una secuencia de registros al sensor. Este estado se encarga de comenzar la medición y cambiar el estado a `ST_WAIT_START`.
    2. **ST_WAIT_START**: Aquí se espera hasta que el bit de inicio del sensor se borre, lo que indica que el sensor ha comenzado a medir. Si el tiempo de espera se excede (timeout), el proceso se interrumpe y se devuelve un valor de error (65535).
    3. **ST_WAIT_RANGE**: Se espera que el sensor termine de calcular el rango. Si el rango está listo, se lee el valor de la medición. Si también ocurre un timeout, se devuelve un valor de error.
    4. **ST_START**: Vuelve al estado inicial para esperar la siguiente medición.

Polulo lib

Agregado a la lib de Polulo el la lectura no bloqueante

3. Puntos clave de la reforma:

- **No bloqueo**:
  - La principal ventaja de este enfoque es que no se bloquea el hilo principal del microcontrolador mientras el sensor realiza la medición. Esto es crucial en sistemas en los que se necesitan realizar otras tareas (como leer botones o controlar LEDs) mientras se esperan los resultados de la medición.
- **Timeout**:
  - Se usa un mecanismo de **timeout** para garantizar que, si el sensor no devuelve una medición en el tiempo esperado, el proceso se interrumpe para evitar que el programa quede bloqueado indefinidamente esperando.
- **Estados y transición**:
  - La máquina de estados se utiliza para dividir el proceso en pasos claros y definidos. El control de los estados asegura que el sistema pueda regresar al estado inicial en caso de error o al completar una medición correctamente.
- **Acceso a registros**:
  - La modificación implica el manejo directo de registros del sensor a través de `writeReg` y `readReg`, lo cual es un cambio en comparación con el enfoque bloqueante tradicional que espera a que el sensor termine antes de seguir con la ejecución.

Polulo lib

Agregado a la lib de Polulo el la lectura no bloqueante

4. Recurso necesario para la reforma:

- **Conocimiento de la arquitectura del sensor**:
  - Para implementar este tipo de reformas, se necesita un buen entendimiento de la **arquitectura interna del sensor VL53L0X**, especialmente sobre cómo maneja el inicio de la medición y cómo se obtiene el rango de manera eficiente.
- **Manejo de máquinas de estados**:
  - Se debe comprender cómo implementar una máquina de estados eficiente, donde el ciclo de ejecución del programa no se detenga durante las esperas. Este enfoque es esencial para realizar una reforma no bloqueante.
- **Interacción con el temporizador (timeout)**:
  - Es necesario implementar un mecanismo robusto para verificar el **timeout**, garantizando que si el sensor no responde dentro de un tiempo razonable, se pueda recuperar sin dejar que el sistema se quede esperando indefinidamente.
- **Acceso a registros del hardware**:
  - La reforma requiere un manejo preciso de los registros del sensor, lo cual implica estar familiarizado con la **documentación técnica** del VL53L0X y las **operaciones de bajo nivel** para la lectura y escritura de registros.

Polulo lib

Agregado a la lib de Polulo el la lectura no bloqueante

5. Ventajas de la reforma:

- **Eficiencia y concurrencia**: Al no bloquear el flujo principal del código, se pueden realizar otras tareas mientras se espera que el sensor complete su medición.
- **Optimización para sistemas embebidos**: Este tipo de arquitectura es muy útil para sistemas con recursos limitados, donde se necesita gestionar múltiples tareas concurrentemente sin perder tiempo esperando mediciones.

Resumen final:

La reforma transforma la lectura del sensor en un proceso no bloqueante, utilizando una máquina de estados para manejar las distintas etapas de la medición. El sistema no se queda esperando, lo que mejora la eficiencia y permite que otras tareas se realicen mientras se espera el resultado del sensor. Para implementar esta reforma es necesario comprender la arquitectura del sensor, la máquina de estados, el manejo de **timeouts** y el acceso a registros del hardware, lo que aporta flexibilidad y eficiencia al sistema.

Lectura del tof

tof.cpp

Output Serial Monitor ×

Message (Enter to send message to 'Arduino Mega or Mega 2560' on 'COM4

```
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 22 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 22 ms
Diferencia de tiempo entre muestras válidas: 20 ms
Diferencia de tiempo entre muestras válidas: 22 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 21 ms
Diferencia de tiempo entre muestras válidas: 22 ms
Diferencia de tiempo entre muestras válidas: 21 ms
```

```
    // Obtiene el tiempo actual
    tiempo_actual = millis();

    if (primer_tiempo) {
        // Si es la primera vez, guarda el tiempo actual
        tiempo_anterior = tiempo_actual;
        primer_tiempo = false;
    } else {
        // Si no es la primera vez, calcula la diferencia de tiempo
        unsigned long diferencia_tiempo = tiempo_actual - tiempo_anterior;
        tiempo_anterior = tiempo_actual; // Actualiza el tiempo anterior

        // Imprime la diferencia de tiempo
        Serial.print("Diferencia de tiempo entre muestras válidas: ");
        Serial.print(diferencia_tiempo);
        Serial.println(" ms");
    }


}
```

tof.cpp

Conclusión

Ventaja de la Arquitectura No Bloqueante en tu Proyecto

Tu medición muestra que el loop principal de tu ESP32 tiene un tiempo de ejecución promedio de **227 µs (microsegundos)**, mientras que la lectura del sensor ToF no bloqueante tarda **22 ms (milisegundos)**. Esta diferencia es clave para entender la gran ventaja de una arquitectura **no bloqueante**.

Si usaras una arquitectura **bloqueante**, la ejecución del programa se detendría **22 ms cada vez que lees el sensor ToF**, lo que es casi **100 veces más lento** que el ciclo normal de ejecución. Durante ese tiempo, no podrías realizar ninguna otra tarea, como:

- Monitorear y gestionar otros sensores o periféricos.
- Controlar el parpadeo del LED en la máquina de estados.
- Responder a eventos de la interfaz web.
- Manejar la comunicación UART y el procesamiento de datos.

En cambio, con una **arquitectura no bloqueante**, el sistema puede continuar ejecutando otras tareas mientras el sensor ToF completa su lectura en segundo plano. Así, el ESP32 sigue procesando eventos y dispositivos sin interrupciones, mejorando la **eficiencia** y **capacidad de respuesta** del sistema.

En resumen, sin una arquitectura no bloqueante, el ESP32 perdería valioso tiempo esperando la respuesta del sensor ToF en cada ciclo, lo que afectaría negativamente el rendimiento y la capacidad de multitarea del sistema

Conclusión

tof.cpp

Desventajas de la Arquitectura No Bloqueante

Si bien la arquitectura no bloqueante ofrece grandes ventajas en rendimiento y multitarea, también introduce **mayor complejidad en el diseño y desarrollo** del software. Algunas desventajas clave son:

1. **Mayor complejidad en la gestión del flujo del programa**
   - En un enfoque bloqueante, simplemente llamas a una función y esperas el resultado.
   - En un diseño no bloqueante, debes dividir la lógica en **eventos o estados** y asegurarte de que cada parte del código se ejecute en el momento adecuado sin interferir con otras tareas.
2. **Sincronización y manejo de estados**
   - Es necesario implementar **máquinas de estado** o **callbacks** para manejar correctamente el progreso de tareas que toman tiempo (como la lectura del sensor ToF).
   - Se debe asegurar que los datos sean accesibles en el momento correcto sin introducir errores o conflictos.
3. **Uso de recursos especializados**
   - No todas las bibliotecas están diseñadas para ser no bloqueantes. En este caso, fue necesario modificar la **librería del sensor ToF de Pololu** para adaptarla a este modelo. Esto implica un **conocimiento avanzado** del hardware y del código de la biblioteca.
   - Se tuvo que implementar una forma de iniciar la medición del ToF y luego recuperar los datos en otro momento sin interrumpir el flujo principal del programa.

En resumen, aunque la arquitectura no bloqueante mejora el rendimiento y la eficiencia, requiere un diseño más complejo, manejo de estados y, en algunos casos, modificar bibliotecas que originalmente fueron diseñadas para un modelo bloqueante.