Trabajo Integrador:

Sistema de Control Vectorial del Empuje con PID para Estabilización de Cohetes Amateur

Microcontroladores y Electrónica de Potencia

Profesores
Iriarte, Eduardo E. (Titular)
Cruz, Martín (JTP)

Alumnos
Avalos, J. Agustín - 13291
Roby Culasso, Martina - 13510

Fecha de presentación (Rev.1): 25/07/2025

Índice

1. Introducción **3**

2. Esquema tecnológico **3**

3. Detalle de módulos **4**

3.1. Computadora Personal (PC) 4

3.2. Microcontrolador STM32F446RET6 4

3.3. Servomotor SG90 5

3.4. Sensor inercial MPU6050 6

3.5. Módulo de lectura/escritura para tarjetas SD 7

3.6. Indicadores luminosos LED 8

4. Funcionamiento general **8**

5. Configuración del microcontrolador **11**

5.1. Configuración del Clock 11

5.2. Timers 11

5.2.1. TIM2 11

5.2.2. TIM3 12

5.2.3. TIM4 13

5.2.4. TIM5 14

5.3. Conectividad 14

5.3.1. I2C1 14

5.3.2. SPI1 15

5.3.3. USART2 16

5.4. GPIO 17

5.5. Debugging 18

5.6. Middleware y Software 18

5.7. Asignación completa de pines 19

6. Programación **20**

6.1. Variables relevantes 21

6.2. Inicio del programa 24

6.2.1. Rutina de verificación de componentes 27

6.3. Intérprete de comandos 38

6.4. Máquina de estados 42

6.5. Rutinas de servicio a interrupciones (ISR) 45

6.5.1. Interrupción por recepción de mensajes vía UART 45

6.5.2. Interrupción por cumplimiento de período de Timers 46

6.5.3. Interrupción por activación de pulsador 50

6.6. Otras funciones 50

6.6.1. Funciones relacionadas al manejo de archivos en la tarjeta SD 50

6.6.2. Funciones relacionadas al MPU6050 52

6.6.3. Movimiento de servomotores 60

6.6.4. Implementación del PID 60

7. Etapas de montaje y ensayos realizados **62**

8. Resultados **66**

9. Conclusiones **68**

1. Introducción

En el presente informe se describe el desarrollo de un sistema de control PID que permite simular el funcionamiento de los mecanismos utilizados en cohetería amateur para implementar el control vectorial del empuje (TVC, por sus siglas en inglés). Esta es una técnica de estabilización que consiste en modificar la dirección del flujo de gases expulsados por el propulsor con el fin de controlar la trayectoria de la aeronave, cuando esta sufre perturbaciones o cambios de orientación no deseados durante su vuelo.

El sistema está basado en un microcontrolador STM32F446RET6, y se utiliza una estructura tipo soporte cardánico de dos ejes, accionada por servomotores, para el movimiento del cilindro donde iría ubicado el propulsor. Los motores pueden modificar la posición angular respecto de los ejes de guiñada (yaw) y cabeceo (pitch), referidos a la nave, en función de las mediciones obtenidas por un sensor inercial MPU6050, el cual integra un acelerómetro y un giroscopio. Se supone que este elemento se encontraría montado sobre el cohete. A dichas mediciones se les aplica un filtro complementario, lo que permite obtener una estimación precisa y con bajo nivel de ruido de los ángulos de interés.

La arquitectura se completa con un sistema de registro de datos en tarjeta SD, que permite almacenar los valores medidos y las salidas del controlador durante cada simulación de vuelo, si así se desea. También se incorporan LEDs para señalizar distintos estados del sistema, que en un entorno real formarían parte de un panel de control en tierra.

El programa se organiza mediante una máquina de estados que permite acceder a distintos modos de uso, gestionados a través de una interfaz UART. Se implementa además un sistema de manejo de errores que bloquea el funcionamiento normal ante fallas detectadas en la SD, el sensor o los servomotores.

La estructura física del prototipo fue diseñada a partir de modelos CAD de acceso público y componentes propios creados con el software Solid Edge. Las piezas luego se fabricaron mediante impresión 3D, y se ensamblaron adecuadamente con el resto de los componentes para obtener la maqueta final y funcional.

2. Esquema tecnológico

En la Figura 1 se presenta un diagrama de bloques en donde pueden observarse los módulos que componen al sistema, junto con los periféricos del microcontrolador que están involucrados para interactuar con ellos. También se indica mediante flechas la dirección en la que ocurre la comunicación.

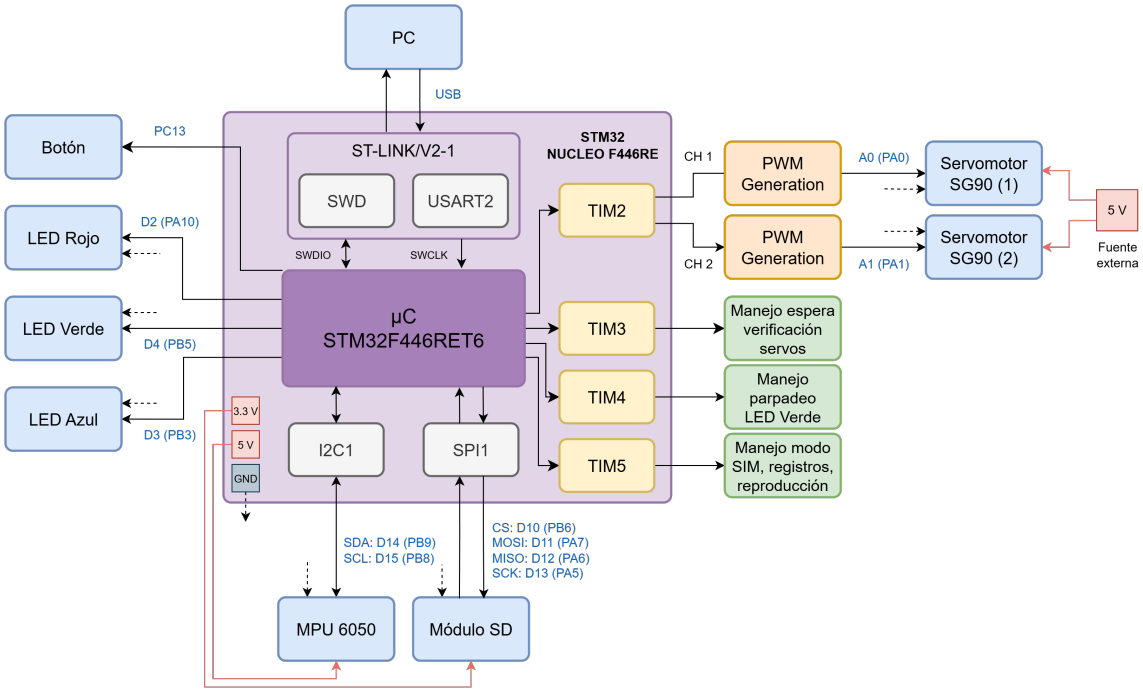


Figura 1: Esquema tecnológico.

3. Detalle de módulos

3.1. Computadora Personal (PC)

La computadora personal actúa como la interfaz principal entre el usuario y el sistema. A través de la conexión USB, es posible cargar firmware al microcontrolador y depurar el código en tiempo real. Adicionalmente, gracias a la comunicación serial UART habilitada por el ST-LINK/V2-1, el usuario puede enviar comandos y recibir información sobre el estado del sistema.

3.2. Microcontrolador STM32F446RET6

Se dispone de una placa de desarrollo Nucleo-F446RE, basada en el microcontrolador STM32F446RET6, fabricado por la empresa STMicroelectronics. La misma se muestra en la Figura 2. Se detallan algunas características de interés:

- **Arquitectura:** ARM Cortex-M4.
- **Frecuencia:** Hasta 180 MHz.
- **Memoria Flash:** 512 KB.
- **RAM:** 128 KB.

- **Número de pines Entrada/Salida (GPIO):** 50 pines que pueden ser configurables para funciones especiales como UART, I2C, SPI, PWM, etc.
- **UART:** Dispone de múltiples interfaces UART/USART (hasta 6).
- **SPI:** Posee hasta 3 interfaces SPI.
- **I2C:** Cuenta con 3 interfaces I2C independientes.
- **Timers:** 2 de 32 bits y 6 de 16 bits.
- **Depuración:** Soporte para depuración a través de ST-LINK/V2-1 integrado en la placa.

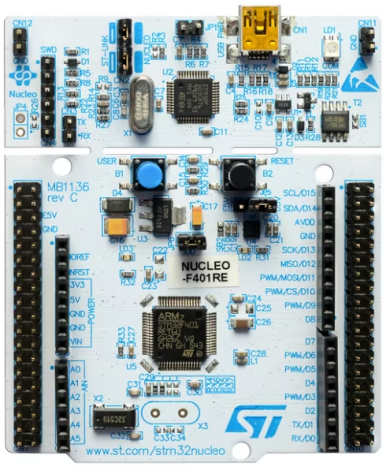


Figura 2: Nucleo-F466RE.

3.3. Servomotor SG90

El sistema emplea dos servomotores SG90, uno para cada eje del "gimbal", con el fin de orientar el tubo donde se montará el propulsor del cohete según la dirección deseada. En la Figura 3 se observa el motor mencionado.

Cada servomotor controla su posición angular en función de una señal PWM. Específicamente, el ángulo de salida del eje depende del *duty cycle* de dicha señal: diferentes anchos de pulso dentro de un período fijo corresponden a distintos ángulos de rotación, permitiendo así un control preciso de la orientación del eje.

Internamente, el servo SG90 está compuesto por un pequeño motor de corriente continua, un conjunto de engranajes reductores y un circuito de control con retroalimentación. El motor genera el movimiento, el tren de engranajes reduce la velocidad y aumenta el torque, y el sistema de control ajusta la posición del eje en función de la señal PWM recibida.

El servo incluye un potenciómetro conectado al eje de salida, que actúa como sensor de posición. Este potenciómetro permite al circuito interno comparar continuamente la posición actual del eje con la posición deseada. Si hay una diferencia entre ambas, el controlador activa

el motor en la dirección correspondiente hasta alcanzar la posición objetivo, implementando así un lazo de control cerrado.

- **Tensión de operación:** 4.8 V a 6.0 V
- **Ángulo de giro:** 0° a 180°
- **Tipo de control:** Señal PWM de 50 Hz (20 ms).
- **Rango en alto de la señal:** De 0.5 ms a 2.5 ms.
- **Torque máximo:** 1.8 kg·cm a 4.8 V
- **Conexiones:** Señal PWM, Vcc y GND.

Figura 3: Servomotor SG90.

3.4. Sensor inercial MPU6050

Para estimar la orientación que debe adoptar el sistema, se utiliza el sensor MPU6050 de la Figura 4. Este dispositivo integra un acelerómetro y un giroscopio, ambos de tres ejes, lo que permite medir tanto la aceleración lineal como la velocidad angular del cohete en el espacio.

Gracias a la combinación de estas dos mediciones, es posible obtener una estimación precisa de la inclinación y orientación del sistema.

- **Alimentación:** Funciona internamente con 3.3V pero posee un regulador de tensión que permite conectarlo a 5V.
- **Interfaz de comunicación:** I2C.
- **Resolución del acelerómetro:** 16 bits. Rangos configurables: ±2g, ±4g, ±8g y ±16g.
- **Resolución del giroscopio:** 16 bits. Rangos configurables: ±250°/s, ±500°/s, ±1000°/s y ±2000°/s.
- **Frecuencia de muestreo:** Hasta 400 kHz.

- **Filtros pasa-bajo configurables:** 260 Hz, 184 Hz, 94 Hz, 44 Hz, 21 Hz, 10 Hz y 5 Hz.
- **Salidas:** Datos de aceleración (X, Y, Z), velocidad angular (X, Y, Z), y temperatura interna.
- **Pines principales:**
  - `VCC`: Alimentación.
  - `GND`: Tierra.
  - `SCL`: Reloj de I2C.
  - `SDA`: Datos de I2C.
  - `INT`: Interrupción (opcional).
  - `ADO`: Selección de dirección I2C (opcional).

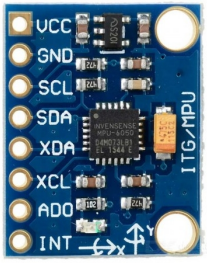


Figura 4: Sensor inercial MPU6050.

3.5. Módulo de lectura/escritura para tarjetas SD

Este módulo permite la lectura y escritura de archivos en tarjetas de memoria SD, facilitando el almacenamiento externo de datos en sistemas embebidos. En la Figura 5 se presenta dicho componente.

- **Alimentación:** 3.3 V.
- **Interfaz de comunicación:** SPI .
- **Formato de tarjeta compatible:** Tarjetas SD estándar en formato FAT16 o FAT32.
- **Pines principales:**
  - `GND`: Tierra
  - `VCC`: Alimentación.
  - `MISO`: Datos desde la tarjeta al microcontrolador.
  - `MOSI`: Datos desde el microcontrolador a la tarjeta.
  - `SCK`: Señal de reloj SPI
  - `CS`: Chip Select, para habilitar la comunicación.

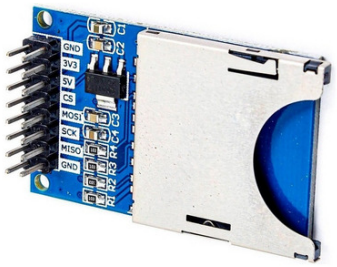


Figura 5: Módulo de lectura/escritura para tarjetas SD.

3.6. Indicadores luminosos LED

Los indicadores LED de la Figura 6 se utilizan para proporcionar señales visuales simples sobre el estado de funcionamiento del sistema.

- **Corriente típica:** Entre 10 mA y 25 mA (limitada mediante una resistencia en serie).
- **Control:** Se conectan a pines GPIO del microcontrolador y se controlan activándolos en nivel alto o bajo según la configuración del circuito.

Figura 6: LED.

4. Funcionamiento general

El usuario puede hacer uso del sistema de diferentes maneras, según el modo de operación que escoja, enviando comandos vía UART. Los tres modos de funcionamiento principales son:

- **Modo Simulación**: En este modo, el sistema imita el comportamiento que el soporte tendría en un vuelo real del cohete; es decir, los servomotores se mueven según lo que se

sensa mediante el MPU. El usuario puede elegir realizar estos movimientos a lazo abierto (predeterminado) o a lazo cerrado (activando el control PID), aunque debe determinarlo antes de entrar al modo. Además, es posible registrar la sesión en la memoria para su posterior análisis. Mientras la simulación está activa, el LED verde parpadea.

- **Modo Memoria**: Al entrar a este modo, el usuario recibe una lista de los registros que han sido guardados en la tarjeta de memoria SD, y puede elegir mostrar una vista previa de algún registro en particular o reproducirlo haciendo uso de los servomotores. Al entrar al modo Memoria, el LED azul se enciende.
- **Modo Test**: Este modo fue diseñado para la verificación y prueba del sistema. Permite posicionar manualmente los servomotores en un ángulo deseado, siempre que la consigna se encuentre dentro del rango de movimiento admitido en cada eje. A este modo le corresponde el LED verde encendido.

El modo "base" es el **Modo Espera**. Como su nombre lo indica, en este modo el sistema queda a la espera de recibir comandos del usuario. Desde aquí es posible ingresar a los otros modos de uso y realizar acciones adicionales, como calibrar el sensor, llevar a los servomotores a su posición de referencia, activar o desactivar el controlador PID para el modo Simulación, y establecer las ganancias del PID. También desde aquí el usuario puede pedir mostrar ayuda sobre los comandos disponibles y ver cómo está configurado el sistema hasta el momento (ganancias, calibración, etc.). En este caso, todos los indicadores LED se encuentran apagados.

Finalmente, ante fallas de alguno de los componentes o comportamientos inesperados, el programa entra a **Modo Error**, desde donde se le da la opción al usuario de intentar reparar el inconveniente encontrado para seguir utilizando el sistema. A este modo le corresponde el LED rojo encendido.

En la Tabla 1 se muestran los comandos que el usuario puede enviar vía UART, tanto para navegar entre modos de funcionamiento, como para realizar las acciones adicionales mencionadas.

Comando	**Descripción**	**Requisitos**	**Habilitado en:**
`:C`	Calibrar sensor.	-	Modo Espera
`:R`	Llevar servos a su posición de referencia.	-	Modo Espera
`:L[C/A]`	Activar/Desactivar PID.	-	Modo Espera
`:K[P/I/D] [Y/P] [num]`	Establecer ganancia PID para eje Yaw/Pitch.	[num] >0	Modo Espera
`:K[P/I/D] [Y/P]`	Ver una ganancia PID específica.	-	Modo Espera
`:K`	Ver todas las ganancias PID.	-	Modo Espera
`:S`	Iniciar Modo Simulación.	Sensor calibrado y servos en referencia.	Modo Espera
`:T`	Iniciar Modo Test.	Servos en posición de referencia.	Modo Espera
`:[Y/P] [num]`	Posicionar servo Yaw/-Pitch.	[num] dentro del rango de movimiento permitido.	Modo Test
`:M`	Iniciar Modo Memoria.	Servos en referencia y SD conectada.	Modo Espera
`:E`	Salir al Modo Espera.	-	Todos
`:H`	Mostrar ayuda.	-	Modo Espera

Tabla 1: Comandos disponibles en el sistema.

Lo que sucede a nivel código al utilizar cada uno de estos comandos se explica más adelante, en la Sección 6.

En la Figura 7 se resume la dinámica entre los distintos modos del sistema (máquina de estados).

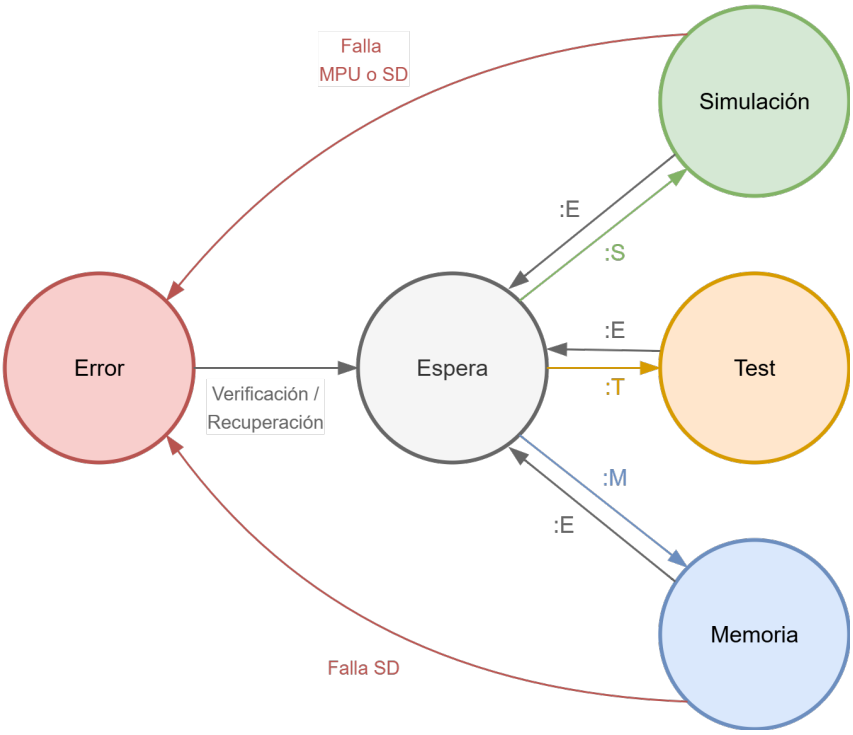


Figura 7: Máquina de estados.

Nota: El modo en el que comienza el sistema es el Modo Error, ya que se exige realizar una verificación de los módulos (tarjeta SD, sensor, servomotores) antes de poder utilizar el programa normalmente. Si la verificación es exitosa, se entra al Modo Espera.

5. Configuración del microcontrolador

La configuración de los periféricos y pines del microcontrolador se realizó mediante el software STM32CubeIDE. A continuación, se muestran algunas capturas de pantalla en donde pueden verse en detalle los ajustes realizados para el proyecto.

5.1. Configuración del Clock

El Clock principal del sistema fue configurado para funcionar a 84 MHz, utilizando el oscilador externo de alta velocidad (HSE).

5.2. Timers

5.2.1. TIM2

El `TIM2` se utiliza para generar la señal PWM que controla la posición angular de los servomotores. Se emplea un único temporizador, pero distintos canales, lo que permite generar

señales con la misma frecuencia pero diferentes ciclos de trabajo. Esto es posible debido a que los canales comparten el mismo registro `ARR` (Auto-Reload Register), que define el período, pero poseen registros `CCR` (Capture/Compare Register) independientes para ajustar el ciclo de trabajo de cada canal.

Se requiere una señal de 50 Hz, por lo que se configura el prescaler y el período del contador tal como se muestra en la Figura 8. A continuación, se detallan los cálculos que justifican dicha configuración:

$$\mathrm{F}_{\mathrm{timer}} = \frac{\mathrm{F}_{\mathrm{clk}}}{\mathrm{Prescaler} + 1} = \frac{84\,\mathrm{MHz}}{84} = 1\,\mathrm{MHz}$$

$$\text{Frecuencia PWM} = \frac{\mathrm{F}_{\mathrm{timer}}}{\mathrm{ARR} + 1} = \frac{1\,\mathrm{MHz}}{20000} = 50\,\mathrm{Hz}$$

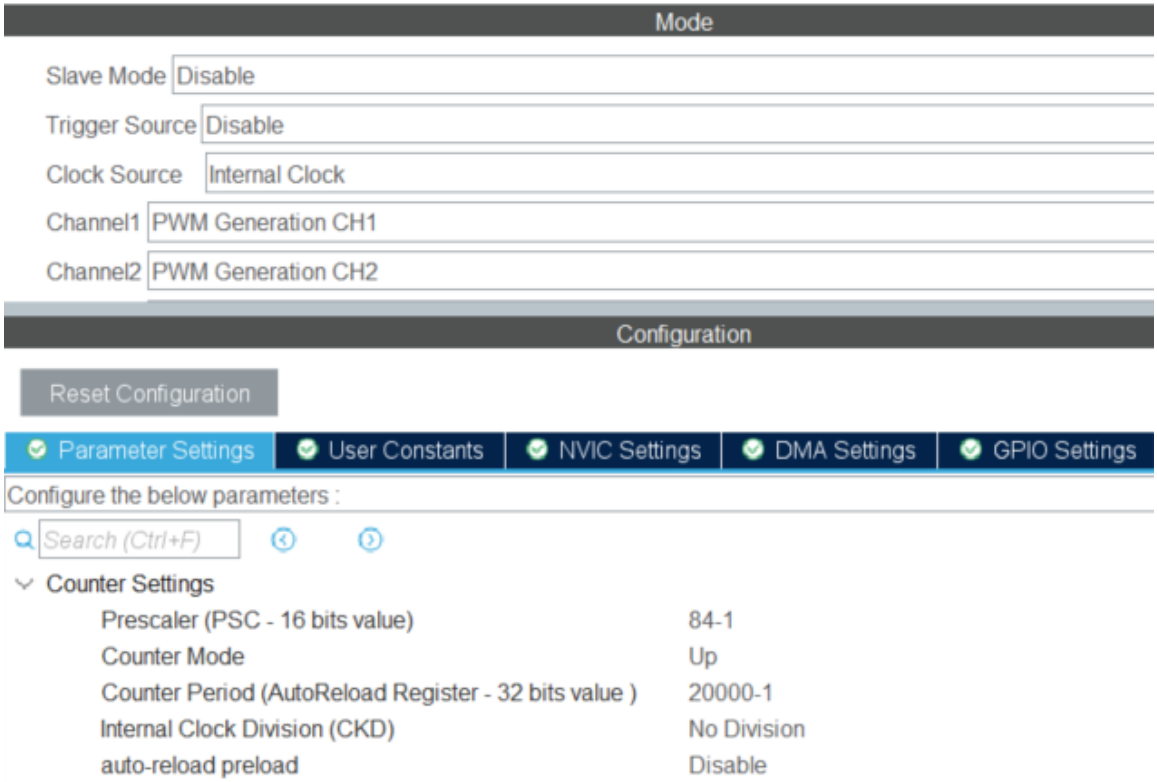


Figura 8: Configuración TIM2.

5.2.2. TIM3

La configuración del `TIM3` se muestra en la Figura 9, de este modo se genera una interrupción cada 1 ms (nótese que se han habilitado las interrupciones globales). En la Sección 6.5.2 se detalla el propósito de este temporizador y la rutina de servicio a interrupción (ISR) asociada.

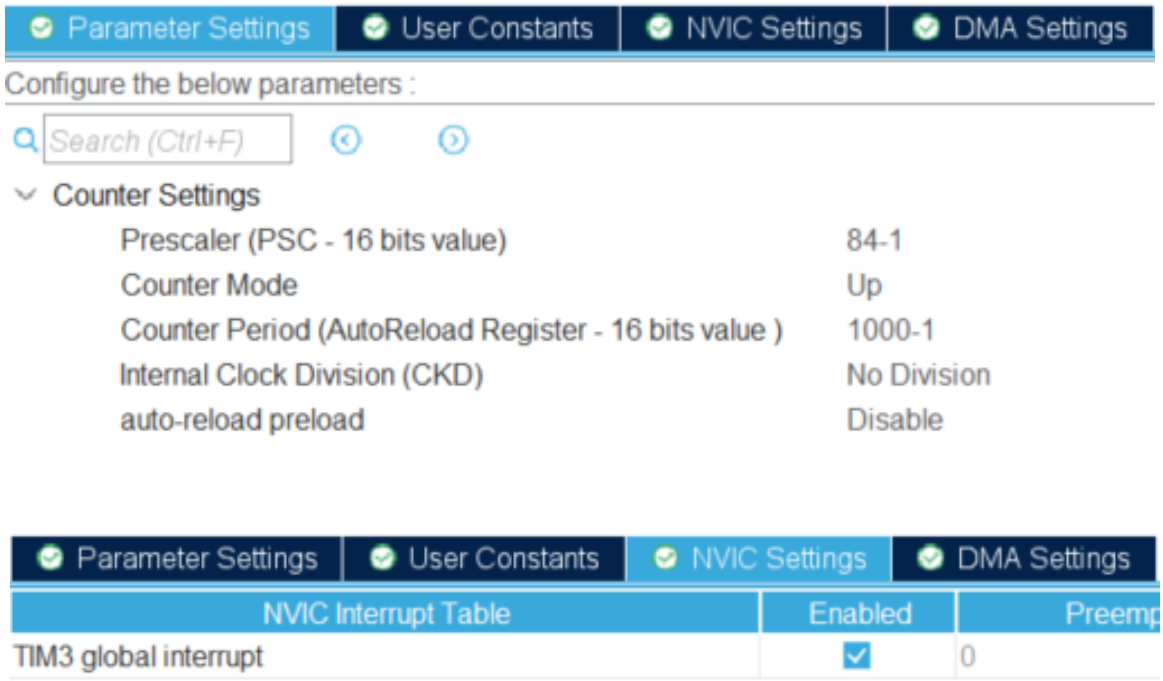


Figura 9: Configuración TIM3.

5.2.3. TIM4

En la Figura 10 se muestra la configuración de este timer, la cual produce una interrupción cada 1 s. Tanto la función que proporciona el timer como su rutina de servicio de interrupción se detallan en la Sección 6.5.2.

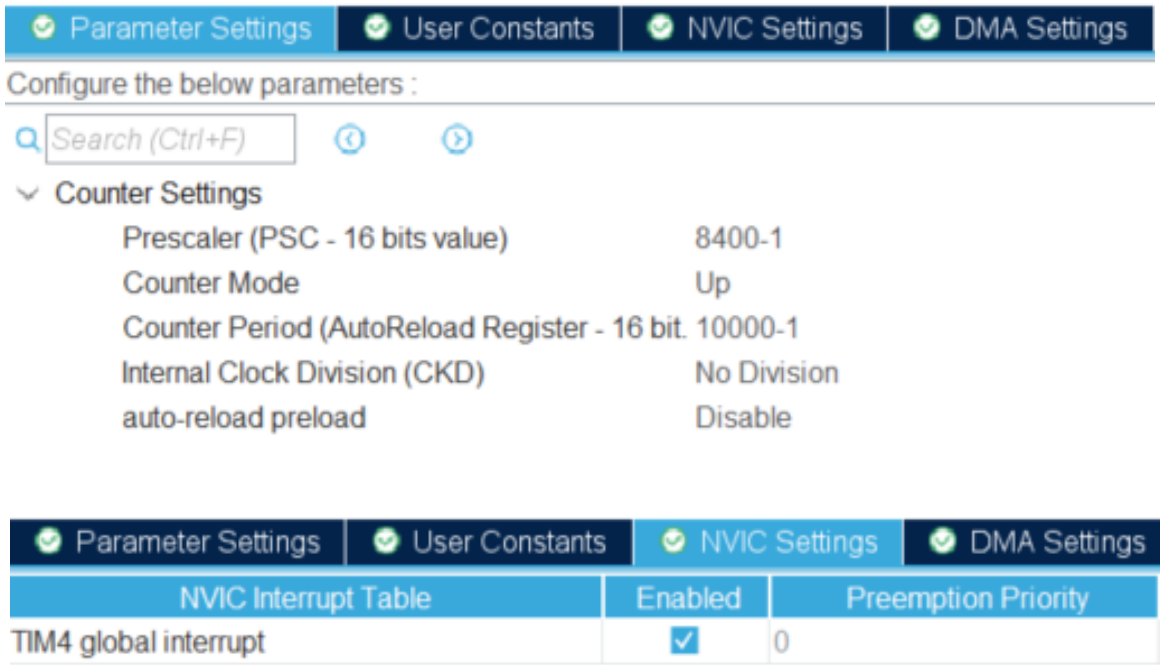


Figura 10: Configuración TIM4.

5.2.4. TIM5

Por último, la configuración del `TIM5` se muestra en la Figura 11, el cual genera una interrupción cada 50 ms. En la Sección 6.5.2 se muestra su rutina de servicio y se explica para qué es utilizado.

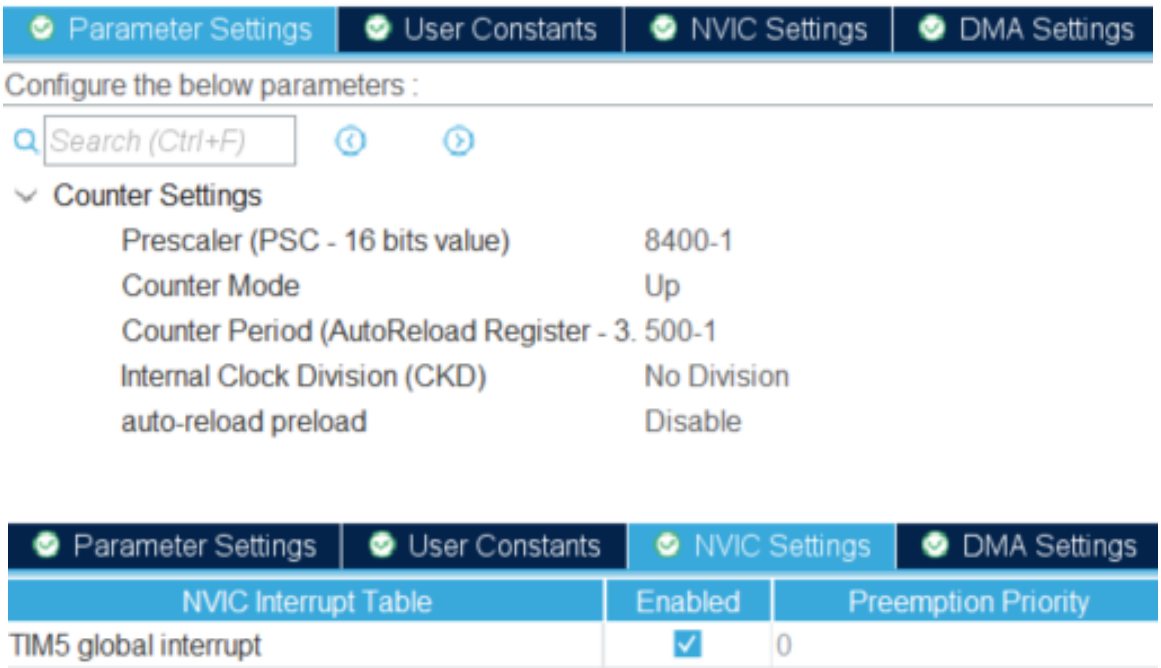
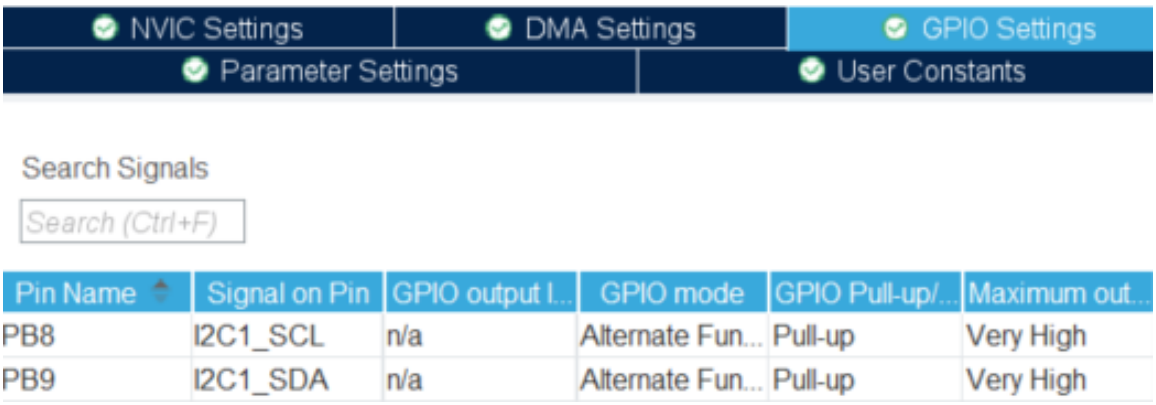


Figura 11: Configuración TIM5.

5.3. Conectividad

5.3.1. I2C1

Para la comunicación con el sensor se utiliza el puerto I2C1 con todos los parámetros por defecto, excepto la configuración de pines: se emplean PB8 para SCL y PB9 para SDA, en lugar de los pines predeterminados PB6 y PB7. La configuración se muestra en la Figura 12.



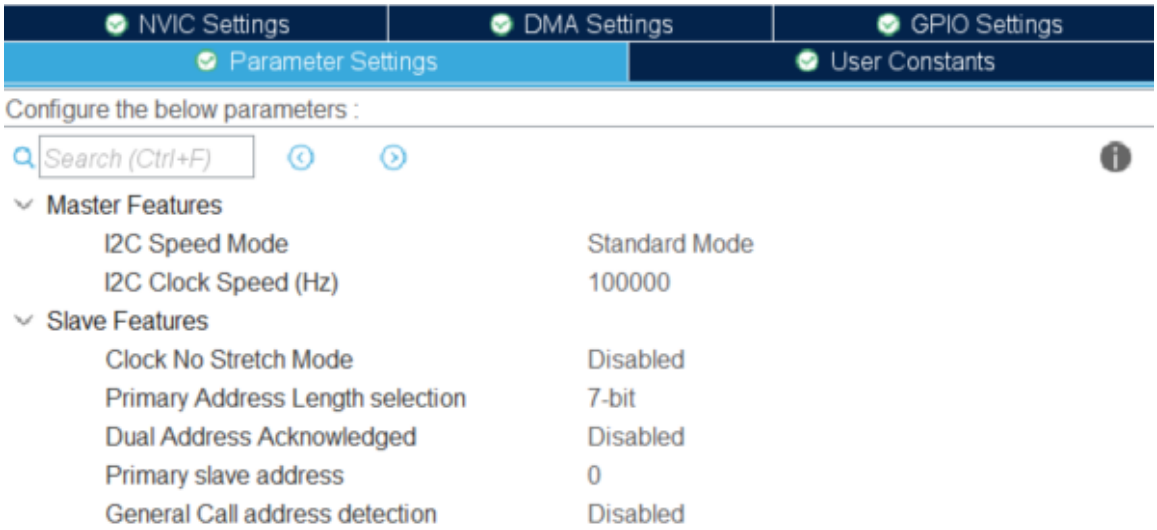


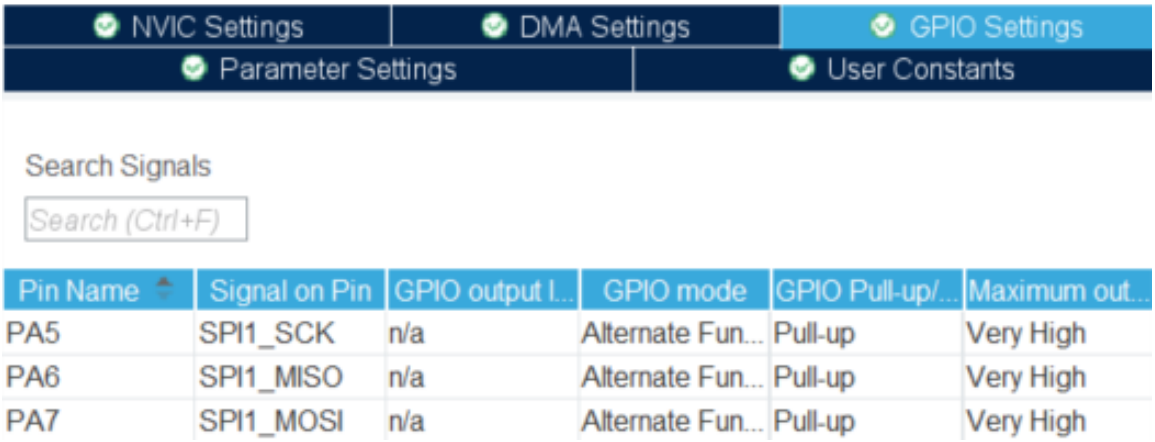
Figura 12: Configuración periférico I2C1.

5.3.2. SPI1

La comunicación con la tarjeta SD se realiza mediante el puerto SPI1, cuya configuración se muestra en la Figura 13. Notar que el modo en el que se ajusta el periférico es *Full-Duplex Master*.

Durante la fase de inicialización, las tarjetas SD requieren frecuencias de comunicación entre 100–400 kHz, por lo cual se configuró el prescaler en 256 para obtener una frecuencia de 328 kHz. El uso de frecuencias superiores durante esta etapa crítica puede causar que la tarjeta no responda a los comandos de inicialización, resultando en fallos de detección o corrupción de datos.

El pin correspondiente a la señal *Chip Select* (CS) no se configura automáticamente, por lo que se seleccionó manualmente el pin PB6 y se lo configuró como salida digital.



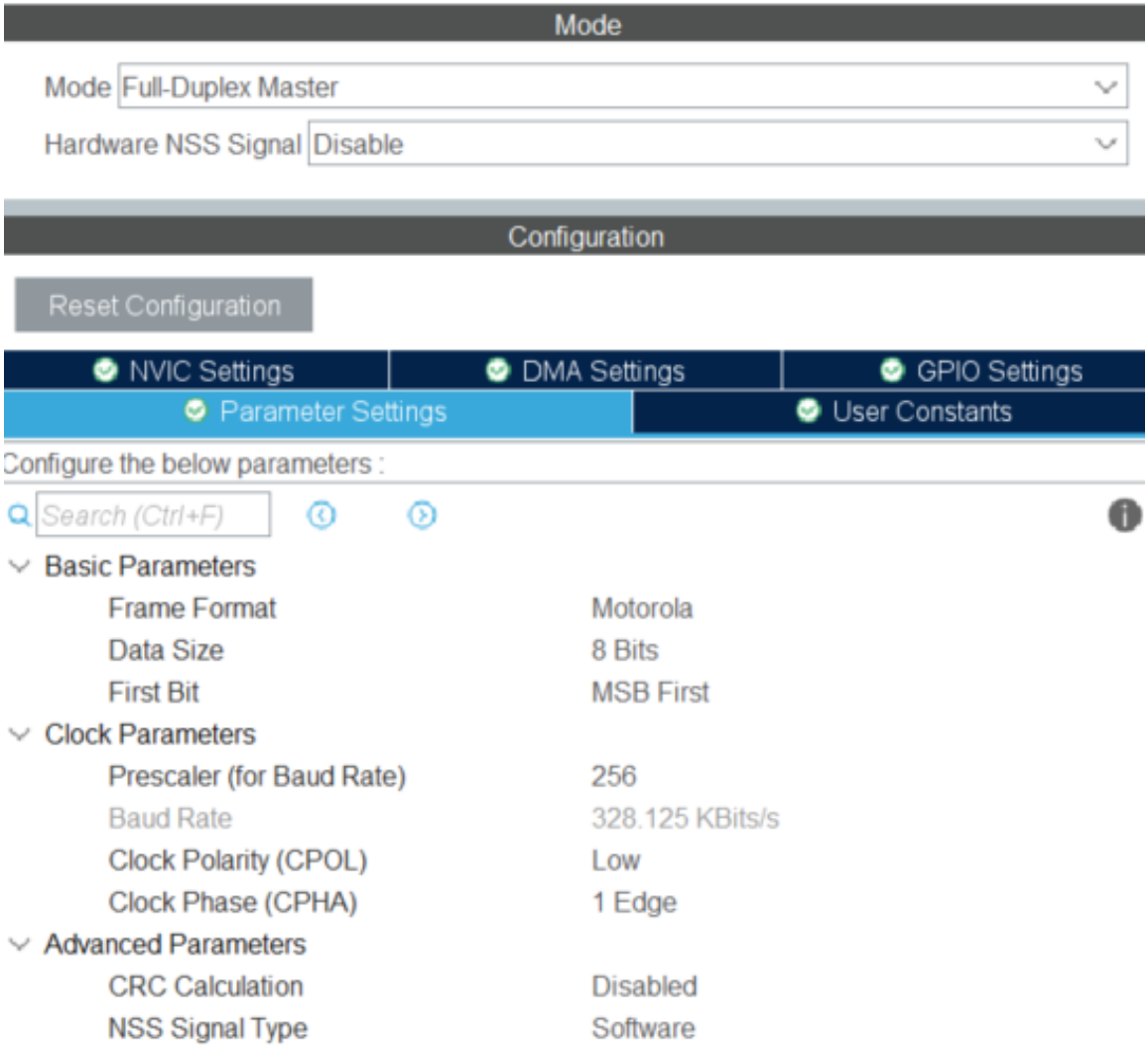


Figura 13: Configuración puerto SPI1.

5.3.3. USART2

La comunicación con el usuario para envío de comandos y recepción de información del sistema se realiza mediante USART2, configurándolo como asíncrono (UART). Su configuración se muestra en la Figura 14 y la rutina de servicio de interrupción correspondiente a la interrupción global que se habilitó se detalla en la Sección 6.5.1.

Figura 14: Configuración puerto USART2.

5.4. GPIO

Adicionalmente, se configuran los pines `PA10`, `PB3` y `PB5` como salidas digitales, los cuales están conectados a los indicadores LED rojo, azul y verde, respectivamente.

Por otro lado, el botón de usuario integrado en la placa, conectado al pin `PC13` y mostrado en la Figura 15, se inicializa en modo de interrupción externa por flanco ascendente (*External Interrupt Mode with Rising edge trigger detection*) y se habilitan las interrupciones *EXTI line[15:10] interrupts*. Esta configuración permite detectar una pulsación (cuando la señal pasa de nivel bajo a alto) y activar una rutina de servicio a interrupciones, la cual se detalla en la Sección 6.5.3.

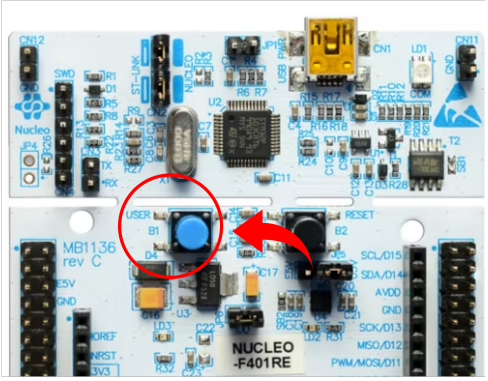


Figura 15: Botón para interrupción externa.

5.5. Debugging

Para el proceso de debugging y programación del microcontrolador se utiliza la interfaz SWD (Serial Wire Debug). Esta interfaz permite la programación, depuración y monitoreo del sistema a través del cable USB proporcionado con la placa.

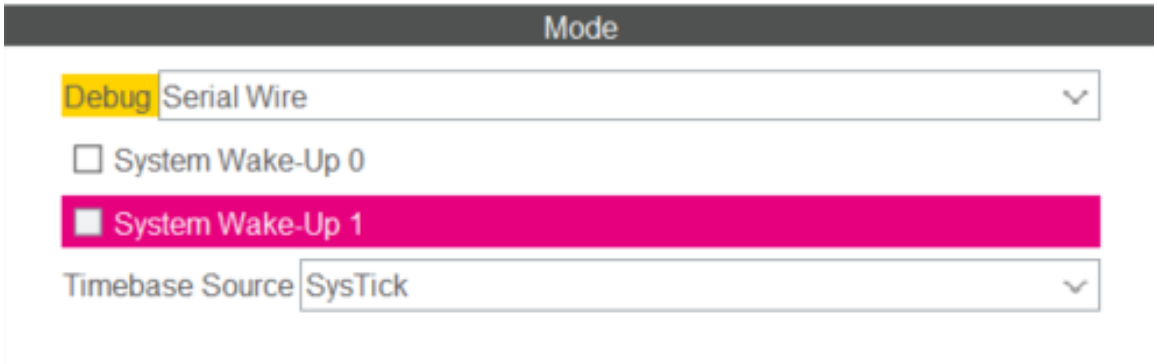


Figura 16: Configuración del debugger.

5.6. Middleware y Software

El sistema utiliza el middleware FATFS, que se incluye directamente desde la configuración. Esta librería permite interactuar con sistemas de archivos compatibles con FAT, como las tarjetas SD con formato FAT16 o FAT32, utilizando funciones de alto nivel para abrir, leer, escribir y cerrar archivos.

En este proyecto, FATFS se emplea para gestionar el registro de sesiones en una tarjeta microSD conectada mediante la interfaz SPI1. Gracias a esta integración, el microcontrolador puede almacenar datos en tiempo real durante la simulación, así como acceder posteriormente a los archivos generados para su visualización o reproducción. Las funciones principales utilizadas se describen en la Sección 6.6.1.

5.7. Asignación completa de pines

Finalmente, en las Figuras 17 y 18 puede verse la asignación completa de pines y sus especificaciones, que resumen lo descrito anteriormente.

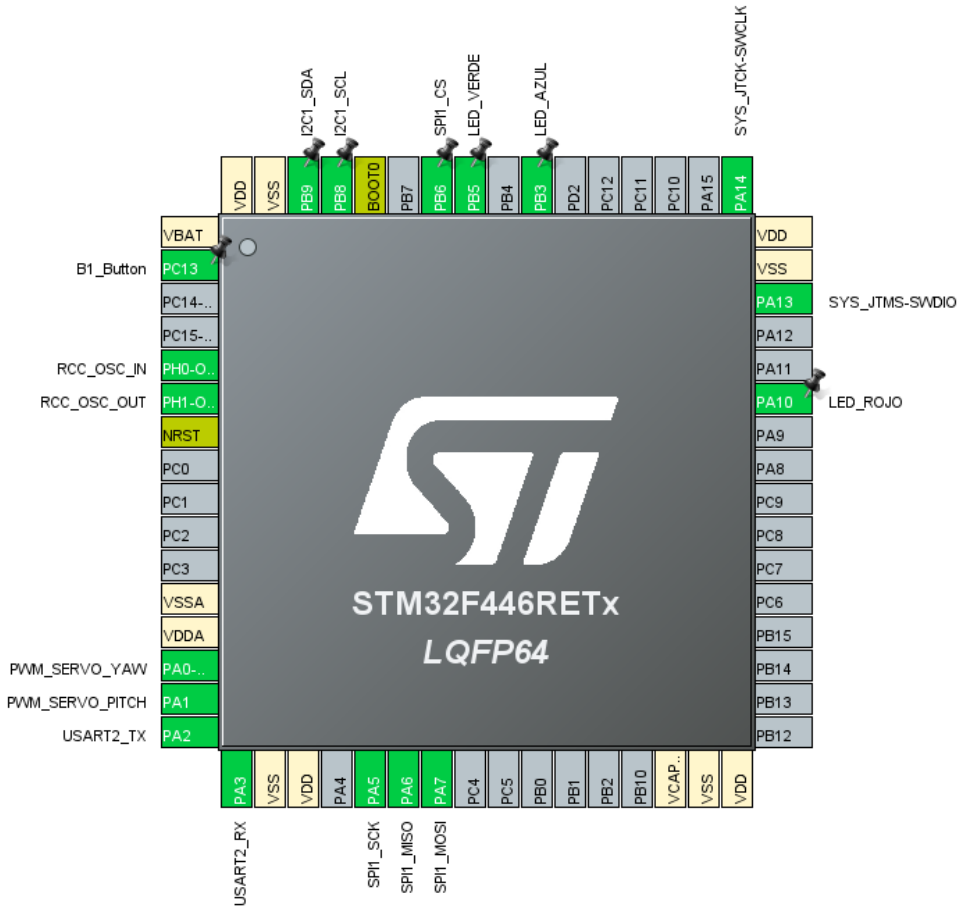


Figura 17: Asignación de pines.

IP	Pin	Signal	GPIO mode	GPIO pull/up pull down	Max Speed	User Label
I2C1	PB8	I2C1_SCL	Alternate Function Open Drain	**Pull-up** *	**Very High** *	
	PB9	I2C1_SDA	Alternate Function Open Drain	**Pull-up** *	**Very High** *	
RCC	PH0-OSC_IN	RCC_OSC_IN	n/a	n/a	n/a	
	PH1-OSC_OUT	RCC_OSC_OUT	n/a	n/a	n/a	
SPI1	PA5	SPI1_SCK	Alternate Function Push Pull	**Pull-up** *	**Very High** *	
	PA6	SPI1_MISO	Alternate Function Push Pull	**Pull-up** *	**Very High** *	
	PA7	SPI1_MOSI	Alternate Function Push Pull	**Pull-up** *	**Very High** *	
SYS	PA13	SYS_JTMS-SWDIO	n/a	n/a	n/a	
	PA14	SYS_JTCK-SWCLK	n/a	n/a	n/a	
TIM2	PA0-WKUP	TIM2_CH1	Alternate Function Push Pull	No pull-up and no pull-down	Low	PWM_SERVO_YAW
	PA1	TIM2_CH2	Alternate Function Push Pull	No pull-up and no pull-down	Low	PWM_SERVO_PITCH
USART2	PA2	USART2_TX	Alternate Function Push Pull	No pull-up and no pull-down	**Very High** *	
	PA3	USART2_RX	Alternate Function Push Pull	No pull-up and no pull-down	**Very High** *	
GPIO	PC13	GPIO_EXTI13	External Interrupt Mode with Rising edge trigger detection	No pull-up and no pull-down	n/a	B1_Button
	PA10	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	LED_ROJO
	PB3	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	LED_AZUL
	PB5	GPIO_Output	Output Push Pull	No pull-up and no pull-down	Low	LED_VERDE
	PB6	GPIO_Output	Output Push Pull	**Pull-up** *	Low	SPI1_CS

Figura 18: Detalle de pines separados por periférico.

6. Programación

En este apartado se describe la implementación del proyecto desde el punto de vista del software. El código desarrollado se basa en una arquitectura reactiva orientada a eventos, donde la lógica principal del sistema no se implementa directamente dentro del bucle infinito `while(1)`, sino que se ejecuta a través de interrupciones y máquinas de estados.

Al comenzar, se realiza la configuración e inicialización de todos los periféricos internos del microcontrolador que se utilizan (ver Sección 6.2), y se le exige al usuario iniciar una rutina de verificación de los componentes principales del sistema. La verificación consta de una parte automática para el módulo SD y el sensor, y otra que requiere de feedback visual e interacción por parte del usuario para los servomotores (ver Sección 6.2.1). Una vez superada la verificación de manera exitosa, el sistema puede comenzar a utilizarse de forma normal. En caso de falla, la verificación debe reintentarse.

A partir de este punto, el flujo del programa queda delegado a las interrupciones por UART, que ocurren cada vez que el usuario envía comandos a través de una terminal (ver Sección

6.5.1). Estos comandos pueden modificar el estado general del sistema (ver Fragmento de Código 1, Sección 6.4), el estado de entrada esperado (ver Fragmento de Código 2), o una combinación de ambos, además de ejecutar funciones internas y actualizar variables según el caso (principalmente banderas de control).

En el modo de entrada normal, el sistema invoca al intérprete de comandos (ver Sección 6.3), para organizar la lógica asociada a cada instrucción reconocida. Los demás modos de entrada permiten manejar secuencias específicas de interacción con el usuario: confirmaciones del tipo "y/n", intentos de recuperación ante fallas, selección de archivos a reproducir, ingreso de nombres para registros, entre otros. Estos modos permiten fragmentar la lógica principal y guiar al usuario paso a paso, sin recurrir a funciones bloqueantes.

También existen rutinas de interrupción asociadas a los temporizadores (`TIM3`, `TIM4` y `TIM5`) que se activan únicamente cuando es necesario mediante bloques condicionales, y permiten ejecutar tareas periódicas, como la simulación con control PID, la escritura en la tarjeta SD, la reproducción de registros, la prueba secuenciada de servos (que también implica una interrupción externa por botón, ver Sección 6.5.3) o el parpadeo de indicadores LED (ver Sección 6.5.2).

En las secciones siguientes se describen en detalle las estructuras, funciones y rutinas que componen el sistema, acompañadas de diagramas de flujo y fragmentos de código representativos.

6.1. Variables relevantes

Aquí se muestran algunas definiciones de variables que se realizaron en el código. Se limita a presentar aquellas que son relevantes para entender el funcionamiento del programa y que se mencionan luego en los diagramas de flujo.

En el Fragmento de Código 1 puede verse la enumeración creada para representar los distintos modos de funcionamiento del sistema. El uso del tipo `enum` resulta conveniente en este contexto, ya que los modos son mutuamente excluyentes y es posible asignarle un nombre descriptivo a cada uno en lugar de un valor numérico, lo que es mas intuitivo y fácil de leer.

Se tiene:

- `M_ERR` para el Modo Error.
- `M_STDBY` para el Modo Espera.
- `M_SIM` para el Modo Simulación.
- `M_TEST` para el Modo Test.
- `M_MEM` para el Modo Memoria.

```
typedef enum {
        M_ERR,
        M_STDBY,
        M_SIM,
        M_TEST,
        M_MEM
} mode_t;

mode_t modo;
```

Fragmento de Código 1: Modos de funcionamiento

La variable `modo` se modifica cuando el usuario realiza un cambio de estado y se evalúa en la estructura switch/case de la máquina de estados para realizar las acciones correspondientes a cada modo, como se explica más adelante.

El Fragmento de Código 2 muestra la enumeración que se definió para manejar los distintos tipos de entrada que se esperan vía UART mientras se interactúa con el usuario. Estos casos se incluyen en una estructura switch/case dentro de la rutina de servicio de interrupciones por recepción de mensajes vía UART.

```
typedef enum {
        INPUT_NORMAL,
        INPUT_VERIF,
        INPUT_SERVO_WAIT_B1,
        INPUT_SERVO_CONFIRM,
        INPUT_CAL,
        INPUT_CONFIRM_SD,
        INPUT_ID,
        INPUT_ELEGIR_REG,
        INPUT_VER_REPRODUCIR,
        INPUT_ERR_SD,
        INPUT_ERR_MPU
} input_state_t;

input_state_t input_state;
```

Fragmento de Código 2: Casos de entrada que se manejan por UART

Se describe brevemente la entrada que se espera en cada caso y en qué contexto se utiliza. Otros detalles de esta implementación pueden verse la Sección 6.5.1.

- `INPUT_NORMAL`: Luego de validar el comando ingresado (si empieza con ":" y si consiste de más de dos caracteres), se llama a la función `interprete()` para procesar los caracteres recibidos.
- `INPUT_VERIF`: Este caso se encuentra vacío dentro de la estructura switch/case de la interrupción, ya que se maneja en un bloque condicional externo. Lo que hace es esperar a cualquier entrada vía UART para dar inicio a la rutina de verificación inicial, y también maneja el reintento de la rutina cuando esta falla.

- `INPUT_SERVO_WAIT_B1`: Este caso también se encuentra casi vacío dentro del switch/-case, solamente lanza un mensaje de error si se intenta enviar algún comando por la terminal. Esto es porque este modo de entrada espera que el usuario presione un pulsador que hace que comience la secuencia de movimiento de los servomotores para su verificación, y se maneja con la interrupción propia del pulsador.
- `INPUT_SERVO_CONFIRM`: Aquí se espera una respuesta de tipo "y/n", con la que el usuario indica si los servomotores se han movido correctamente durante la verificación.
- `INPUT_CAL`: Este modo fue diseñado para darle tiempo al usuario de acomodar el MPU adecuadamente antes de comenzar su calibración. Espera una respuesta de tipo "y" para comenzar, o "n" para salir y hacer la calibración en otro momento.
- `INPUT_CONFIRM_SD`: A este caso se ingresa luego de entrar al Modo Simulación, en donde se le pregunta al usuario si desea registrar la sesión. Espera una respuesta de tipo "y/n".
- `INPUT_ID`: Si la respuesta del caso anterior es afirmativa, se entra a este caso para recibir el identificador del archivo a crear para el registro. Se hace una verificación de los caracteres ingresados y se revisa si un archivo con el mismo nombre ya existe antes de crearlo.
- `INPUT_ELEGIR_REG`: Este caso le da lugar al usuario para seleccionar uno de los registros disponibles en la tarjeta SD cuando entra al Modo Memoria para verlo o reproducirlo.
- `INPUT_VER_REPRODUCIR`: Este caso espera una respuesta "v/r" a la pregunta de si se desea ver o reproducir el registro seleccionado en el Modo Memoria.
- `INPUT_ERR_SD`: Aquí se maneja el intento de recuperación de la tarjeta SD cuando ocurre una falla. Entrada de tipo "y/n" según indicaciones.
- `INPUT_ERR_MPU`: Aquí se maneja el intento de recuperación del sensor cuando ocurre una falla. Entrada de tipo "y/n" según indicaciones.

Finalmente, en el Fragmento de Código 3 se muestran las banderas que fueron definidas para indicar si se han cumplido o activado ciertas configuraciones del programa. Aunque el tipo de dato utilizado para estas variables es entero de 8 bits sin signo (`uint8_t`), en el programa se utilizan como booleanos; pueden tomar los valores 0 ó 1.

```
uint8_t flagVerif = 0;
uint8_t flagCal = 0;
uint8_t flagCalCase = 0;
uint8_t flagRef = 0;
uint8_t flagMem = 0;
uint8_t flagReg = 0;
uint8_t flagRepro = 0;
uint8_t flagPID = 0;
```

Fragmento de Código 3: Flags

En la lista que sigue se explica qué señala cada una:

- `flagVerif`: indica si se ha completado o no la rutina de verificación inicial de los componentes del sistema.
- `flagCal`: indica el estado de calibración del sensor inercial.
- `flagCalCase`: flag auxiliar para mostrar el mensaje de error adecuado cuando se entra a `M_ERR` debido a una falla durante la calibración.
- `flagRef`: indica si los servomotores se encuentran en su posición de referencia o si no/no se sabe.
- `flagMem`: indica si hay registros disponibles en la SD. Es 0 cuando la tarjeta de memoria se encuentra vacía.
- `flagReg`: indica si el usuario ha activado el registro o no durante una sesión de simulación.
- `flagRepro`: indica si hay una reproducción de registro en curso.
- `flagPID`: indica si se ha activado el controlador PID para el Modo Simulación.

6.2. Inicio del programa

El programa comienza con la inicialización de todos los periféricos que fueron habilitados previamente, en la etapa de configuración del microcontrolador. Luego, se habilitan los temporizadores configurados en modo interrupción, y aquellos destinados a generar señales PWM para mover los servomotores se activan en modo de salida.

A continuación, se realiza una limpieza tanto de la terminal como del buffer de recepción UART, con el fin de descartar posibles datos residuales de ejecuciones anteriores. Una vez hecho esto, se le envía al usuario un mensaje de bienvenida con indicaciones para iniciar la rutina de verificación de los componentes. Esta rutina tiene como objetivo asegurar que, al menos al principio de la ejecución del programa, todos los módulos externos en los que se apoya el sistema funcionen correctamente (ver Sección 6.2.1). Para finalizar, se habilitan las interrupciones por UART para permitir la recepción de comandos del usuario.

Como se ha mencionado antes, el modo de uso en el que comienza el sistema es el Modo Error, y el estado de entrada inicial es `INPUT_VERIF`.

Lo explicado anteriormente se resume en el diagrama de flujo de la Figura 19. El código correspondiente a esta parte del programa se presenta en el Fragmento de Código 4.

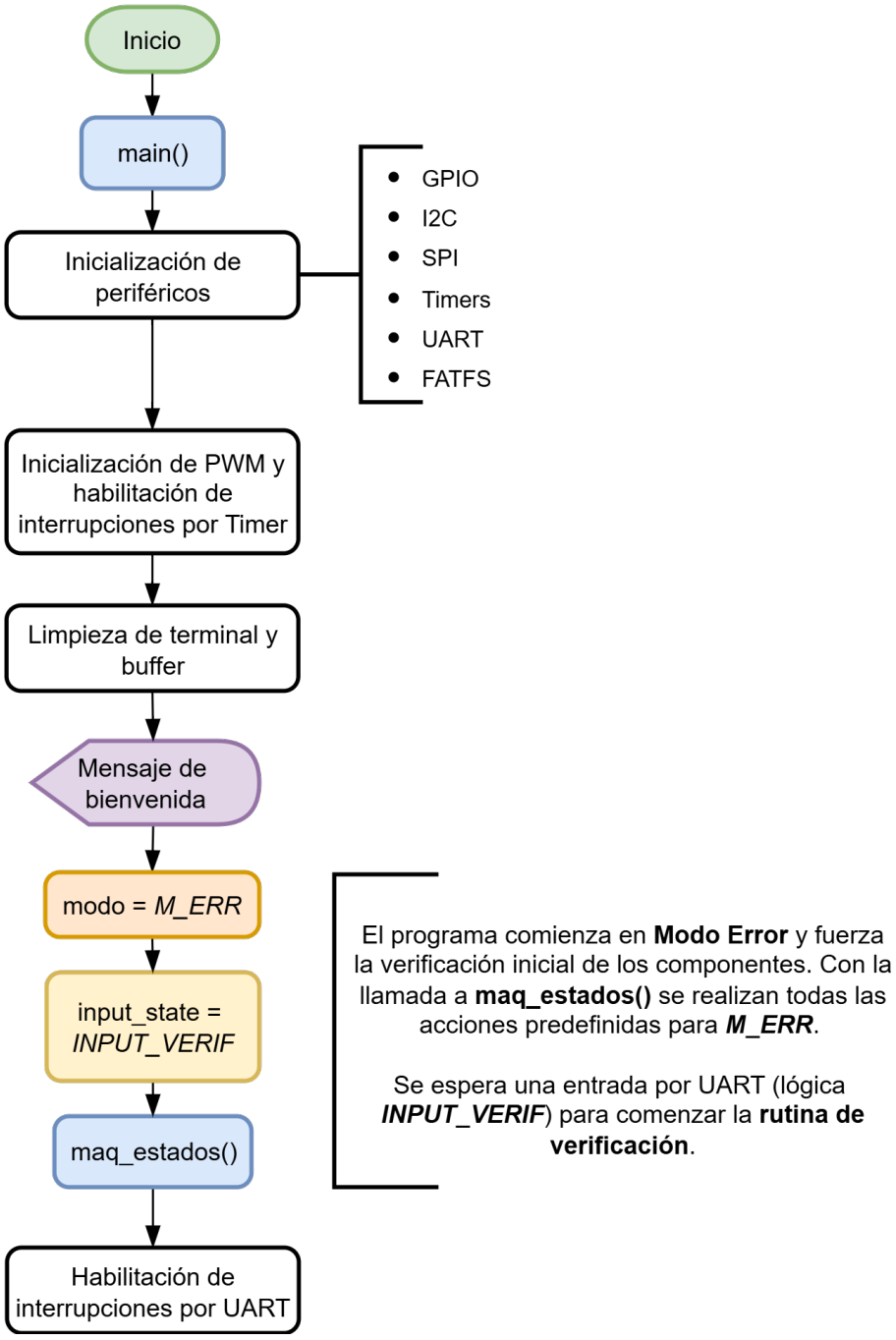


Figura 19: Diagrama de flujo: inicio del programa.

```c
int main(void)
{

  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_I2C1_Init();
  MX_SPI1_Init();
  MX_TIM2_Init();
  MX_TIM4_Init();
  MX_TIM5_Init();
  MX_USART2_UART_Init();
  MX_FATFS_Init();
  MX_TIM3_Init();
  /* USER CODE BEGIN 2 */

    // Inicializar PWM en los timers y canales correspondientes
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);  // Servo 1 (YAW)
    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_2);  // Servo 2 (PITCH)

    // Inicializar timers en modo interrupcion
    HAL_TIM_Base_Start_IT(&htim3);
    HAL_TIM_Base_Start_IT(&htim4);
    HAL_TIM_Base_Start_IT(&htim5);

    // Limpiar terminal
    HAL_UART_Transmit(&huart2, (uint8_t*) clear, strlen(clear), 10);
    // Limpiar buffer (luego encapsulado en la funcion clear_buffer())
    memset(rx_buffer, 0, sizeof(rx_buffer));
    rx_index = 0;

```

```
    // Mensajes iniciales
    myprintf("=======================\r\n");
    myprintf("BIENVENIDO\r\n");
    myprintf("=======================\r\n\r\n");

    myprintf("*ALRT: SE REQUIERE REALIZAR VERIFICACIÓN DE COMPONENTES\r\n");
    myprintf("Presione cualquier tecla para comenzar...\r\n");

    // Estados iniciales para modo e input_state
    modo = M_ERR;
    input_state = INPUT_VERIF;

    // Llamada a maquina de estados
    maq_estados();

    // Habilitar interrupciones por UART
    HAL_UART_Receive_IT(&huart2, &rx_char, 1);

  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
  /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
    /* USER CODE END 3 */
}
```

Fragmento de Código 4: Inicio del programa.

6.2.1. Rutina de verificación de componentes

El modo de entrada `INPUT_VERIF`, en el que inicia el sistema, espera a que el usuario pulse cualquier tecla para llamar a la función `rutina_verificacion()`. Esta función inicia el proceso de comprobación del correcto funcionamiento de los componentes principales: el módulo SD y la tarjeta de memoria, el sensor inercial, y los servomotores.

La verificación de los dos primeros elementos (SD y sensor) se realiza de forma automática. Para ello, se encapsulan los pasos a seguir en cada caso en las funciones `verif_sd()` y `verif_mpu()`, respectivamente, y se llama a cada una dentro del flujo principal. La lógica en esta etapa es sencilla: si alguno de los chequeos falla, se sale de `rutina_verificacion()` con `flagVerif = 0`. Esta bandera es evaluada al volver al estado `INPUT_VERIF`, y en caso de ser nula, se le ofrece al usuario la posibilidad de reintentar la verificación por completo.

Si se superan las comprobaciones automáticas, se procede a verificar los servomotores. Para ello, se invoca la función `verif_servos()`, que desencadena los pasos restantes de la rutina. Esta etapa requiere intervención directa por parte del usuario, quien debe confirmar visualmente si los servomotores se han movido como se espera. Por este motivo, la función

`rutina_verificacion()` concluye en este punto, y el resto del proceso se gestiona de manera separada.

En la Figura 20 se puede observar el diagrama de flujo para la rutina de verificación explicada hasta el momento.

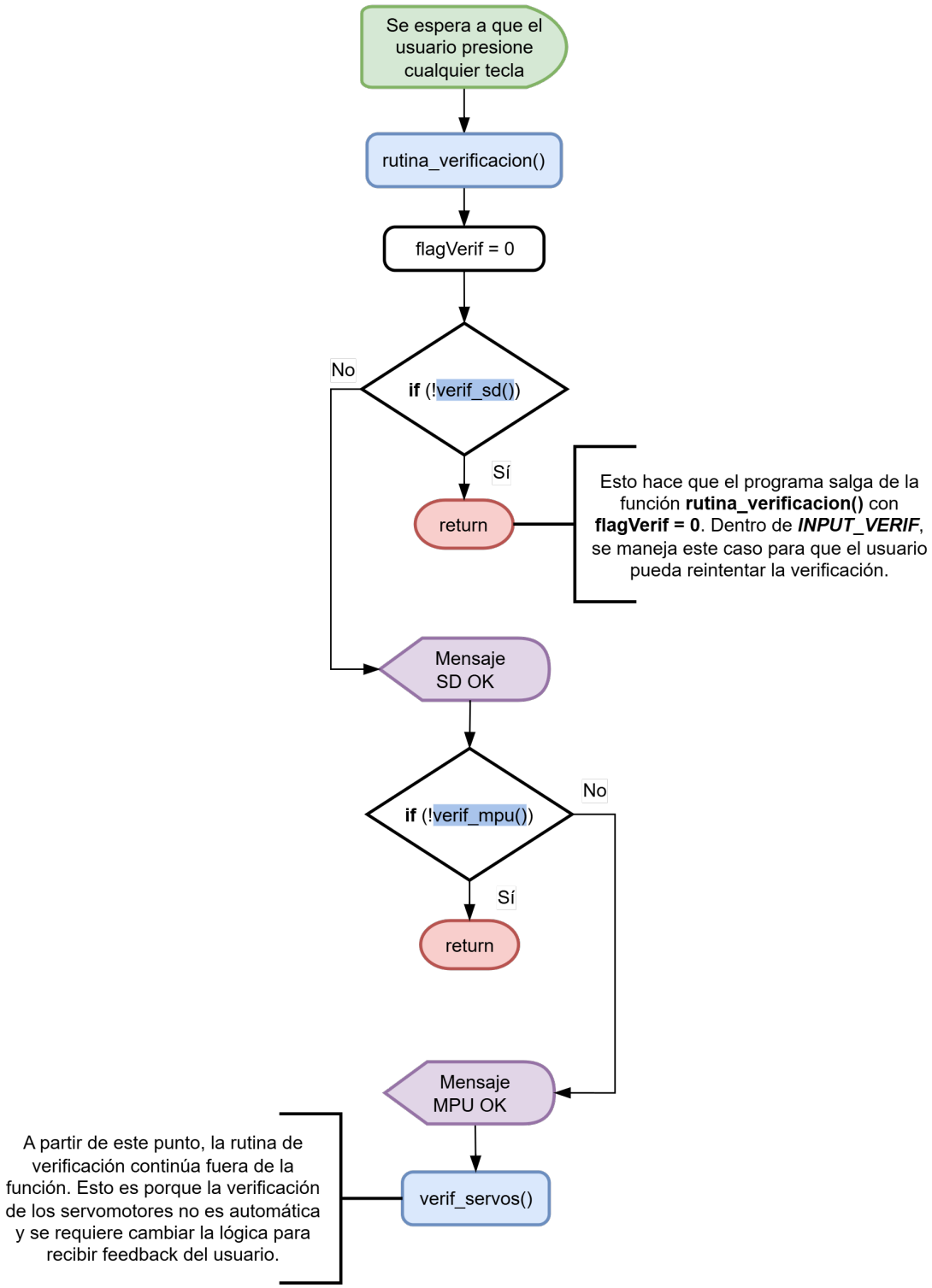


Figura 20: Diagrama de flujo: rutina de verificación.

En la verificación del módulo y de la tarjeta SD, cuya lógica en forma de diagrama puede verse en la Figura 21, se comienza por intentar "montar" la tarjeta, a través de la función `mount_sd()`, que se muestra en la Figura 22. Este proceso es necesario para que el microcontrolador pueda acceder al sistema de archivos de la memoria.

Para ello, primero se reconfigura el periférico SPI, luego se desmonta cualquier sistema de archivos que pudiera haber quedado montado de otra sesión previa, y finalmente, se intenta el montaje haciendo uso de la función `f_mount(FATFS *fs, const TCHAR* path, BYTE opt)` (de la librería FATFS, ver Sección 6.6.1). Si el proceso es exitoso, `mount_sd()` retorna un 1; en caso contrario, retorna un 0.

Cabe aclarar que la reconfiguración del periférico SPI no suele ser necesaria durante la comprobación inicial, pero sí es crucial cuando se vuelve a llamar a la función de montaje tras una desconexión del módulo o la tarjeta SD, o fallas similares. Se incluye esta acción aquí porque la función se reutiliza en estos casos.

En la función "padre", `verif_sd()`, si se recibe un 1 de `mount_sd()`, se continúa el procedimiento intentando obtener información sobre los "clusters" disponibles en la tarjeta, mediante `f_getfree(const TCHAR path, DWORD* nclst, FATFS** fs)`. Un cluster es un conjunto de sectores consecutivos del disco, típicamente de 512 Bytes, que el sistema de archivos trata como una sola unidad al guardar archivos. Si la operación resulta satisfactoria, se procede con el cálculo del espacio total y disponible que existe en la tarjeta SD en kBytes, y se le muestra al usuario mediante la terminal (la fórmula para pasar de clusters a kBytes fue obtenida del foro de ChaN [2], el creador de la librería FATFS).

Ante cualquier falla durante este proceso, se sale de la función con 0, lo que después se captura en la función de verificación principal, y posteriormente, en el caso `INPUT_VERIF`, en donde se maneja el error como ya se explicó antes.

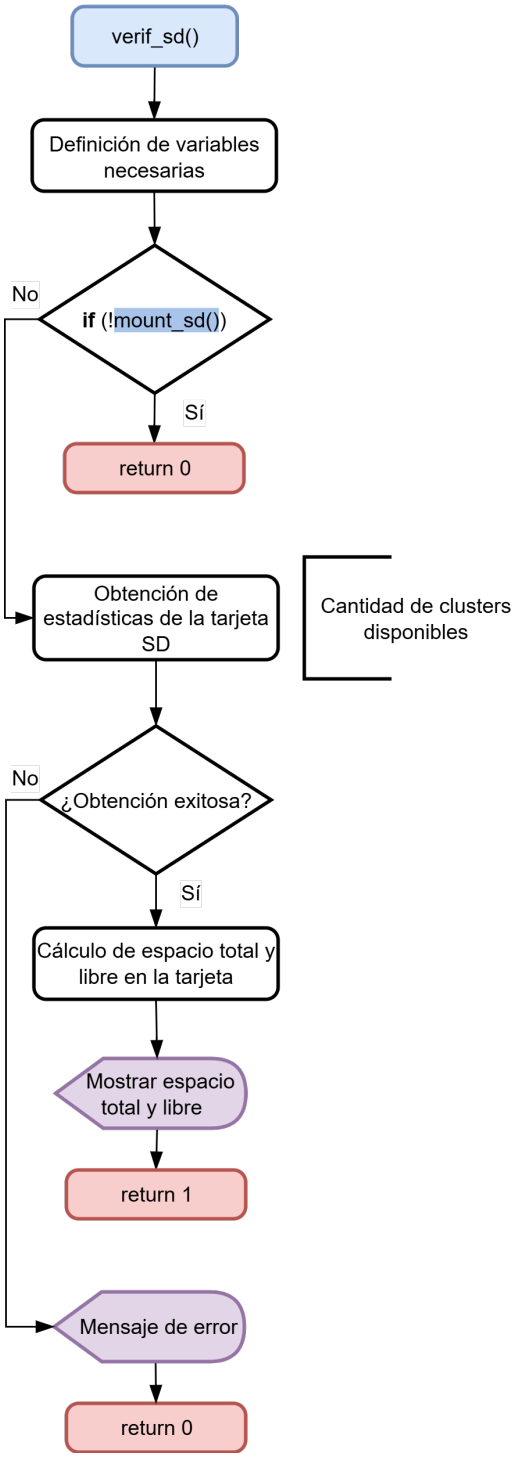


Figura 21: Diagrama de flujo: rutina de verificación del módulo y de la tarjeta SD (1).

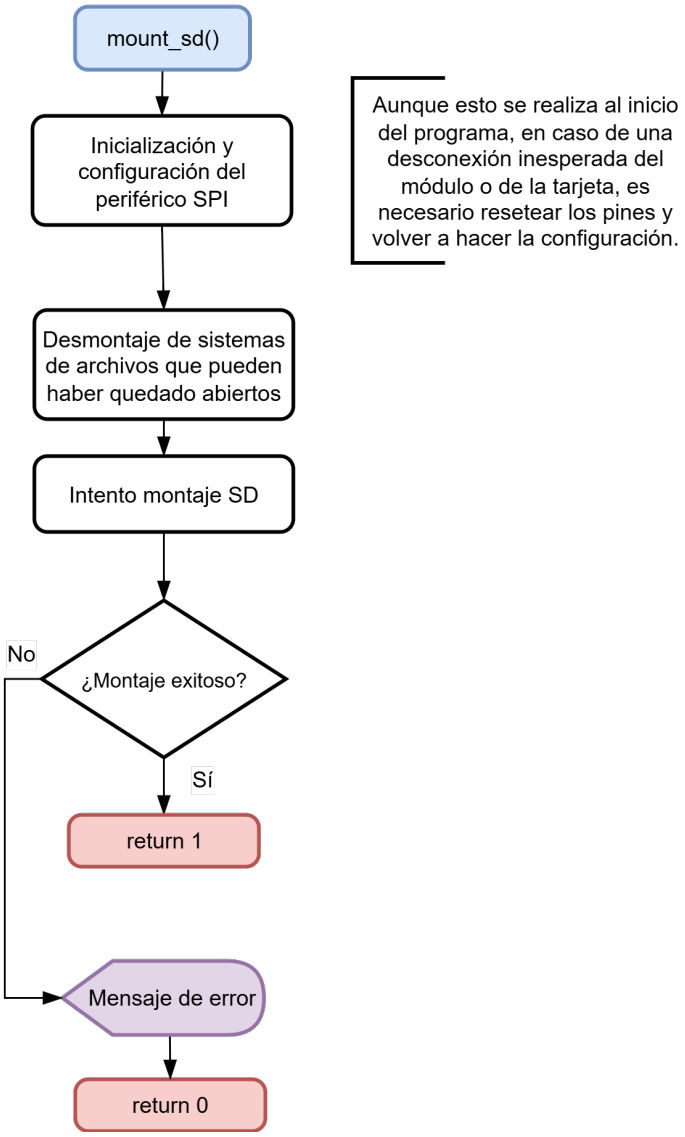


Figura 22: Diagrama de flujo: rutina de verificación del módulo y de la tarjeta SD (2).

La verificación del MPU se ejecuta una vez finalizada la verificación del módulo y la tarjeta SD. El diagrama de flujo correspondiente a `verif_mpu()` se presenta en la Figura 23.

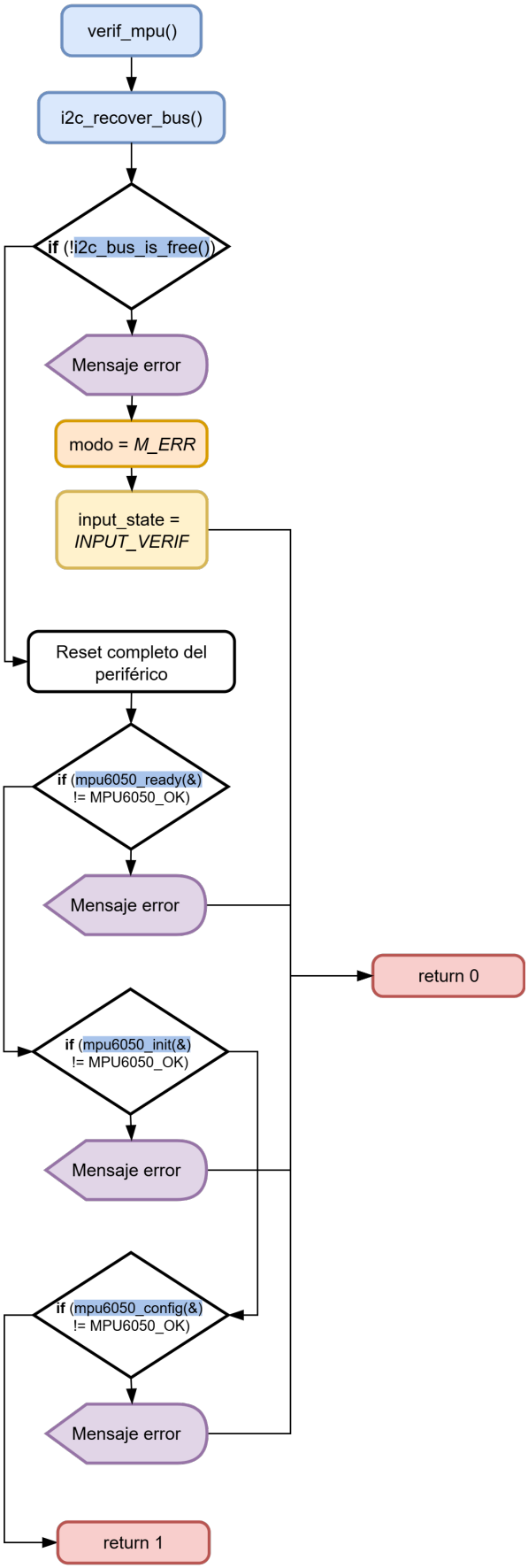


Figura 23: Diagrama de flujo: rutina de verificación del sensor.

La prueba del sensor comienza con una llamada a la función `i2c_recover_bus()`. En ella, se desactiva temporalmente el periférico I2C1 y se configuran las líneas SDA y SCL como salidas de propósito general en modo *open-drain*. Luego, se generan hasta 9 pulsos de reloj sobre la línea SCL, una técnica ampliamente utilizada para forzar al esclavo a liberar la línea SDA y cerrar correctamente una transmisión pendiente. Finalmente, se genera manualmente una condición de parada sobre el bus y se restablecen las configuraciones originales de los pines, reactivando el periférico mediante la función `MX_I2C1_Init()`.

Si bien esta acción puede parecer innecesaria al comienzo del programa, la función se reutiliza en instancias posteriores ante errores de comunicación, y resulta fundamental para recuperar el control del bus cuando el sensor se desconecta inesperadamente y el canal I2C queda en un estado inconsistente.

Luego, se evalúa el resultado entregado por la función `i2c_bus_is_free()`, que verifica el estado físico del bus I2C. Esta devuelve 0 si el bus se encuentra bloqueado, y 1 si está libre. Para ello, se redefinen momentáneamente los pines correspondientes a las líneas SCL y SDA como entradas sin pull, lo que permite leer directamente su estado. Si ambos pines se encuentran en alto, se considera que el bus está libre; si al menos uno está en bajo, se interpreta que el bus sigue bloqueado.

Una vez comprobado que el bus está libre, se reinicia completamente el periférico I2C y se procede con la verificación lógica del sensor. Las funciones utilizadas en esta parte, `mpu6050_ready()`, `mpu6050_init()` y `mpu6050_config()`, se describen en detalle en la Sección 6.6.2, pero básicamente, se encargan de:

- Confirmar que el sensor responde a través del bus I2C.
- Realizar su inicialización básica.
- Configurar registros internos, como el rango de medida, la sensibilidad y los filtros.

Justo después de esta etapa, comienza la secuencia de verificación para los servomotores, con la llamada a `verif_servos()`. El objetivo de esta función es "desbloquear"las interrupciones por activación del pulsador, modificando el valor de la variable `servo_actual` (en la ISR, se verifica que sea distinta de 0 antes de ejecutar las acciones siguientes, ver Figura 25), y cambiando el estado de la entrada esperada desde `INPUT_VERIF` a `INPUT_SERVO_WAIT_B1` (para "desbloquear" la interrupción del Timer 3 con la que se maneja el movimiento de los servos en los pasos siguientes). Así, la reanudación del proceso queda en las manos del cliente, quien recibe un mensaje por la terminal en donde se le indica que debe presionar el botón para continuar. Lo descrito hasta aquí se representa en la Figura 24.

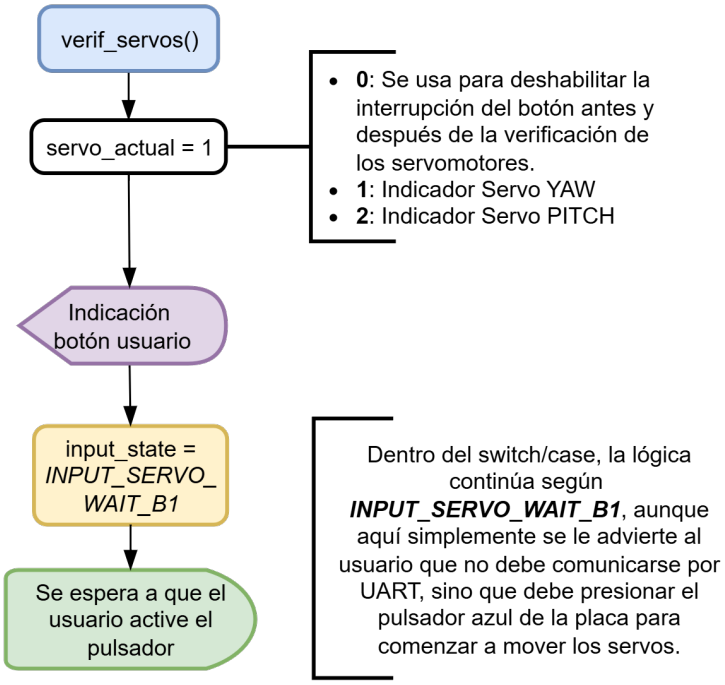


Figura 24: Diagrama de flujo: rutina de verificación de los servomotores (1).

La Figura 25 muestra lo que ocurre dentro de la interrupción disparada por el pulsador. Luego de comprobar que la variable `servo_actual` es distinta de 0 y de revisar que la interrupción proviene del pin adecuado, se le notifica al usuario que se comenzará a mover el servomotor (1 o 2, según corresponda), y se modifican las variables `servo_sequence_active`, `servo_step` y `servo_timer`. Las mismas son relevantes dentro de la función `servo_sequence_task()`, que controla el movimiento de los servos en sí.

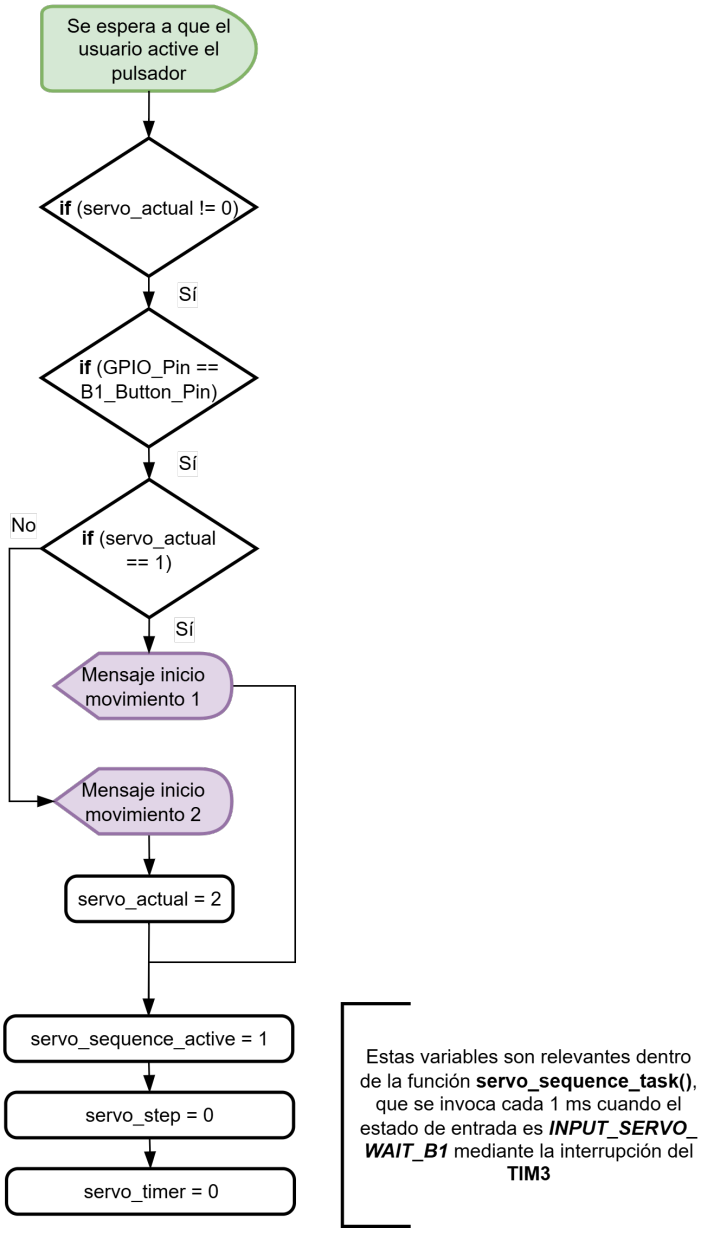


Figura 25: Diagrama de flujo: rutina de verificación de los servomotores (2).

`servo_sequence_task()` se llama dentro de la ISR del TIM3 (ver Sección 6.5.2), siempre y cuando `input_state == INPUT_SERVO_WAIT_B1`. La función se muestra en el Fragmento de Código 5.

```c
void servo_sequence_task(void) {
    if (!servo_sequence_active) return;

    if (servo_timer > 0) {
        servo_timer--;
        return;
    }

    if (servo_actual == 1) {
        // Secuencia SERVO 1 (YAW)
        switch (servo_step) {
            case 0:
                modificar_pwm(&htim2, TIM_CHANNEL_1, ang_to_pwm(REF_YAW));
                servo_timer = 500; // 0.5s
                servo_step = 1;
                break;
            case 1:
                modificar_pwm(&htim2, TIM_CHANNEL_1, ang_to_pwm(MIN_YAW));
                servo_timer = 500; // 0.5s
                servo_step = 2;
                break;
            case 2:
                modificar_pwm(&htim2, TIM_CHANNEL_1, ang_to_pwm(MAX_YAW));
                servo_sequence_active = 0;
                servo_step = 0;
                myprintf("¿Se movió correctamente el servo? (y/n): ");
                input_state = INPUT_SERVO_CONFIRM;
                break;
        }
    } else {
        // Secuencia SERVO 2 (PITCH)
        switch (servo_step) {
            case 0:
                modificar_pwm(&htim2, TIM_CHANNEL_2, ang_to_pwm(REF_PITCH));
                servo_timer = 500; // 0.5s
                servo_step = 1;
                break;
            case 1:
                modificar_pwm(&htim2, TIM_CHANNEL_2, ang_to_pwm(MIN_PITCH));
                servo_timer = 500; // 0.5s
                servo_step = 2;
                break;
            case 2:
                modificar_pwm(&htim2, TIM_CHANNEL_2, ang_to_pwm(MAX_PITCH));
                servo_sequence_active = 0;
                servo_step = 0;
                myprintf("¿Se movió correctamente el servo? (y/n): ");
                input_state = INPUT_SERVO_CONFIRM;
                break;
        }
    }
}
```

Fragmento de Código 5: Función `servo_sequence_task()`.

La primera vez que el usuario presione el pulsador, al cumplirse el período del TIM3 y eje-

cutarse, también por primera vez, la función `servo_sequence_task()`, sucederá lo siguiente:

1. Como `servo_sequence_active == 1` y `servo_timer == 0`, se saltearán los primeros dos condicionales de la función y se evaluará el valor de `servo_actual`.
2. Inicialmente, `servo_actual == 1`, por lo que se ingresará a la estructura switch/case para el servomotor 1 (eje yaw).
3. Como `servo_step == 0`, se ejecutará el primer caso del switch, que moverá el servomotor a su posición de referencia, asignará `servo_timer = 500` y avanzará el paso de la secuencia estableciendo `servo_step = 1`.
4. En el siguiente desbordamiento del TIM3, cuando se vuelva a llamar a la función, se ingresará al segundo condicional, ya que `servo_timer > 0`, y se disminuirá el valor del contador antes de retornar. Este mecanismo actuará como un retardo no bloqueante, simulando la espera necesaria (500 ms = 0,5 s) antes de ejecutar el siguiente movimiento del servo.
5. Una vez que `servo_timer == 0`, se avanzará al siguiente caso de la secuencia: se moverá el servo a su posición mínima, se volverá a asignar el retardo de 500 ms y se incrementará `servo_step`.
6. Finalmente, se repetirá el proceso para mover el servo a su posición máxima.

Al completarse los tres movimientos, se finaliza la secuencia, estableciendo `servo_sequence_active = 0` y `servo_step = 0`. Luego, se le consulta al usuario si el servo se ha movido correctamente ("y/n"), y se cambia el estado de entrada a `INPUT_SERVO_CONFIRM` para capturar este feedback. Si la respuesta es afirmativa, se envía un mensaje notificando que debe presionar nuevamente el botón para iniciar la verificación del segundo servomotor, y se establece `servo_actual = 2`, junto con `input_state = INPUT_SERVO_WAIT_B1`, replicando el flujo anterior. En cambio, si la respuesta es negativa, se retorna al estado `INPUT_VERIF`, y como se saldrá con `flagVerif = 0`, en este modo se maneja el error y se le da al usuario la opción de reintentar la verificación.

Una vez completada con éxito la verificación de ambos servomotores, se establece `flagVerif = 1` y `servo_actual = 0`, desactivando virtualmente la interrupción del pulsador, ya que no se volverá a utilizar. Luego, el sistema cambia al modo de operación `M_STDBY` y al estado de entrada `INPUT_NORMAL`, quedando listo para comenzar la interacción con el usuario mediante comandos. Adicionalmente, se imprime un mensaje en la terminal con las instrucciones básicas para continuar.

Cabe mencionar que aunque no se pide realizar otra verificación completa de componentes una vez superada la rutina inicial, se verifica individualmente la conexión y comunicación con el módulo SD/tarjeta de memoria y con el sensor inercial antes de ingresar a los modos en donde se utilizan, así como también mientras están en uso. Esto permite detectar fallas e ingresar al Modo Error en caso de ser necesario, y le da la posibilidad al usuario de reintentar la conexión con los componentes perdidos.

6.3. Intérprete de comandos

El intérprete de comandos (función `interprete()`) es llamado desde la rutina de servicio de interrupciones por recepción de mensajes vía UART, cuando `input_state == INPUT_NORMAL`. Es el encargado de decodificar la mayor parte de los mensajes que envía el usuario a través de la terminal, que deben empezar con ":" (aunque esa parte de la validación la maneja la ISR, como se explica más adelante).

Su estructura general se muestra en la Figura 26. La lógica dentro de cada uno de los casos del switch se detallan en los diagramas que siguen, por una cuestión de legibilidad. Estos casos abarcan todos los comandos detallados en la Tabla 1.

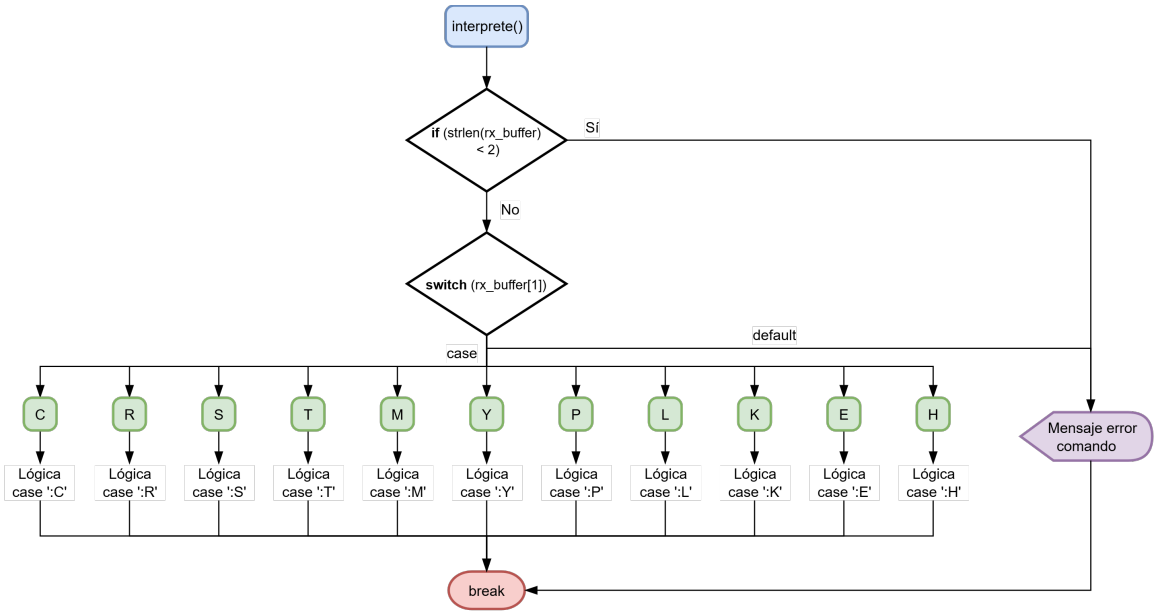


Figura 26: Diagrama de flujo: intérprete de comandos.

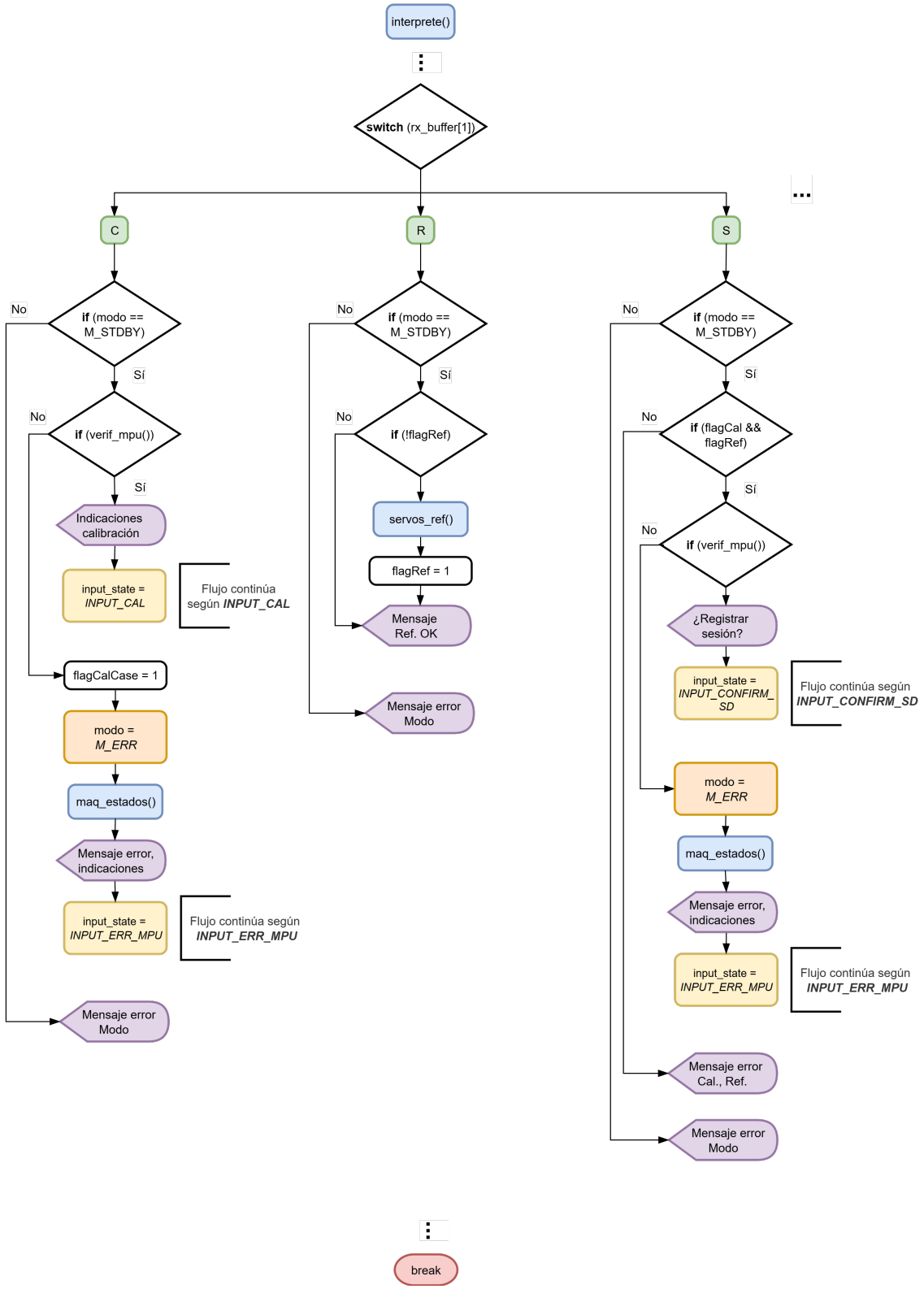


Figura 27: Diagrama de flujo: intérprete de comandos - casos C, R, S.

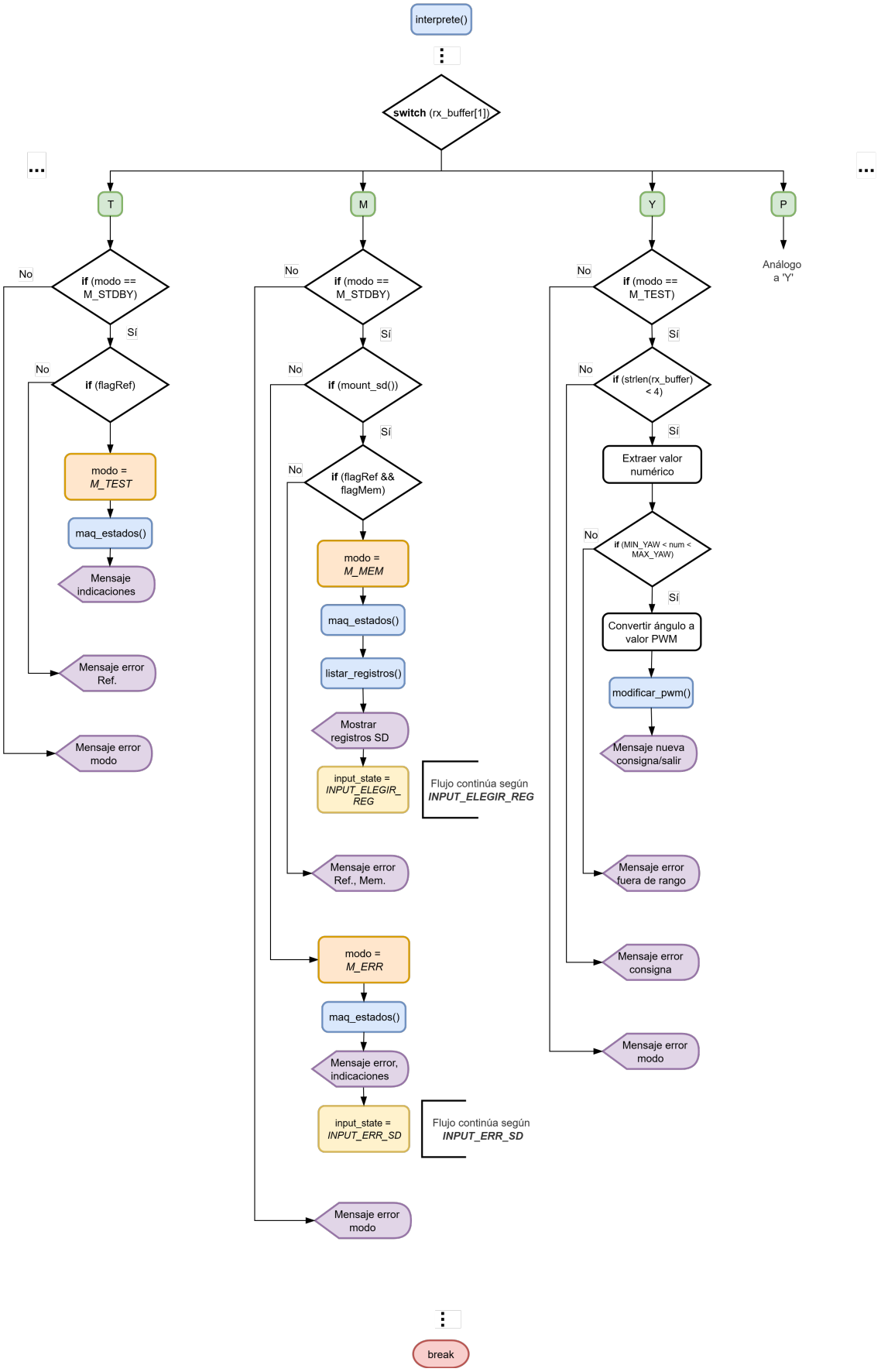


Figura 28: Diagrama de flujo: intérprete de comandos - casos T, M, Y, P.

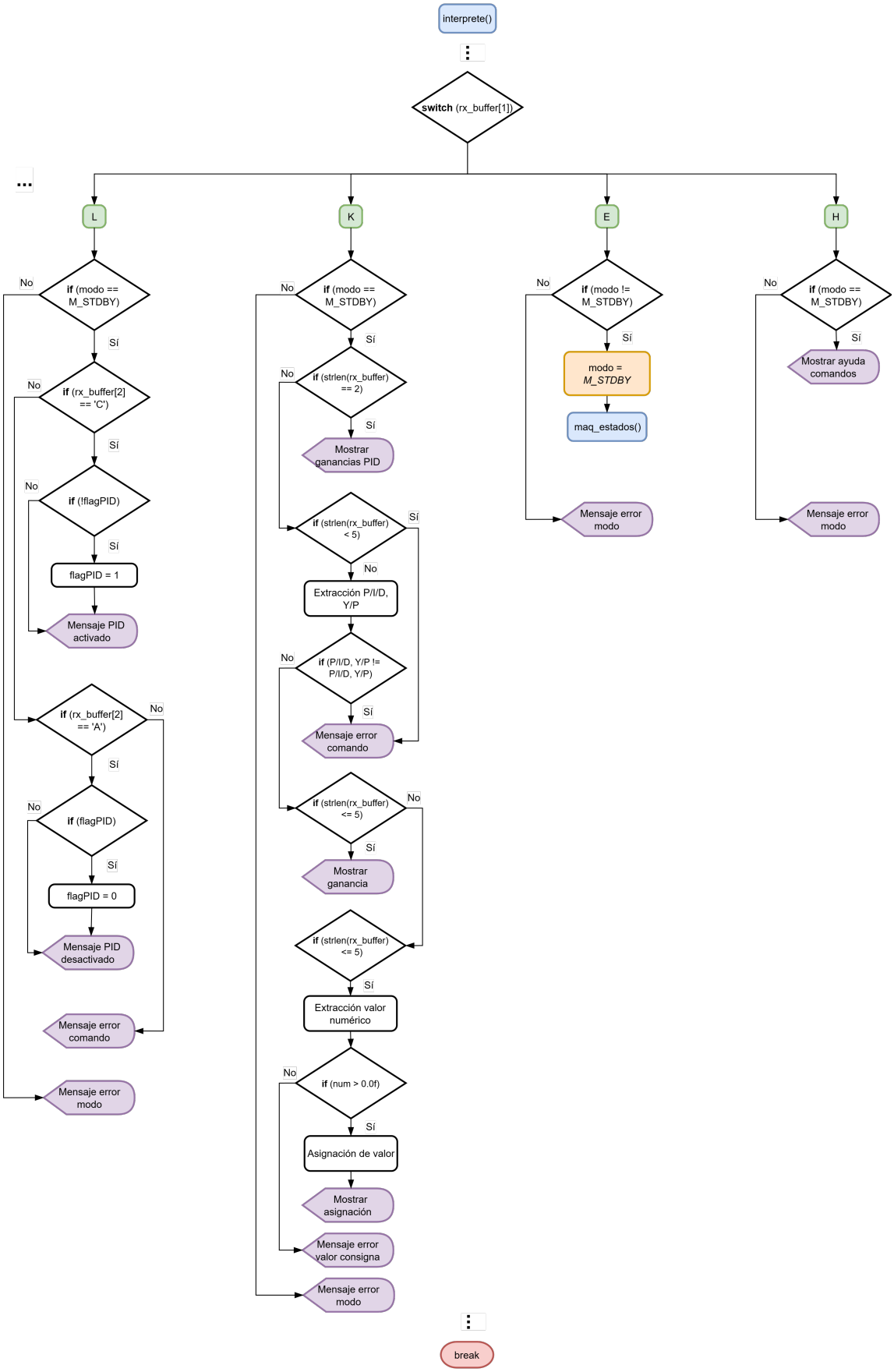


Figura 29: Diagrama de flujo: intérprete de comandos - casos L, K, E, H.

En pocas palabras, se explica qué ocurre en cada caso:

`:C` inicia la verificación del sensor inercial y, si es exitosa, cambia el estado de entrada a `INPUT_CAL` para proceder con la calibración; en caso de falla, el sistema entra en `M_ERR`. El comando `:R` se encarga de llevar los servomotores a su posición de referencia (aquella que hace que el cilindro del soporte quede en posición vertical), si aún no han sido referenciados. `:S` inicia el Modo Simulación si el sistema ha sido calibrado y referenciado, y pregunta al usuario si desea registrar la sesión; de lo contrario, muestra errores apropiados. Los comandos `:T` y `:M` permiten ingresar a los modos Test y Memoria, respectivamente, siempre que se cumplan las condiciones previas: referencia y, en el caso de Memoria, validación del sistema de archivos en tarjeta SD.

Los comandos `:Y` y `:P` permiten enviar nuevas consignas de ángulo (yaw y pitch, respectivamente), válidos únicamente en Modo Test. Se verifica que el valor esté en el rango permitido antes de convertirlo a PWM. `:L` permite activar o desactivar el controlador PID desde el Modo Espera, mientras que `:K` permite consultar o modificar las ganancias del mismo para cada eje y término (P, I, D), con validaciones integradas. El comando `:E` fuerza la salida a Modo Espera desde cualquier otro modo. Finalmente, `:H` despliega un mensaje de ayuda con los comandos disponibles, y cualquier otro comando inválido es manejado por el caso `default`, que notifica al usuario del error de sintaxis o comando no reconocido.

6.4. Máquina de estados

Cada vez que se hace un cambio de modo, se realiza una llamada a la función `maq_estados()`, cuya lógica se presenta en la Figura 30. Esto permite que se realicen todas las acciones correspondientes a cada modo de funcionamiento cuando es debido.

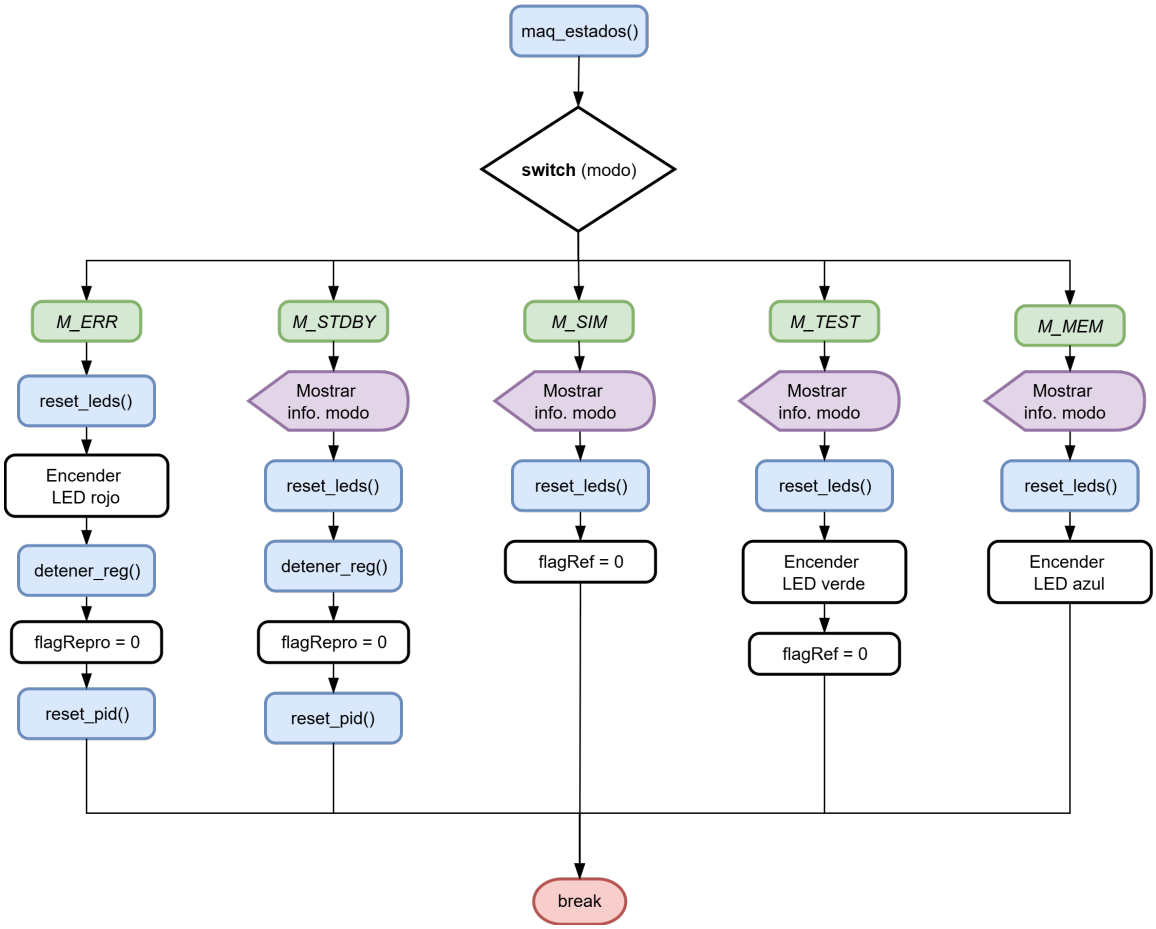


Figura 30: Diagrama de flujo: máquina de estados.

El código para esta función se muestra en el Fragmento de Código 6.

```c
void maq_estados(void) {
    switch (modo) {
        case M_ERR:
                reset_leds();
                HAL_GPIO_WritePin(LED_ROJO_GPIO_Port, LED_ROJO_Pin, GPIO_PIN_SET);
                detener_reg();
                flagRepro = 0;
                reset_pid();
            break;
        case M_STDBY:
                myprintf("----------------\r\n");
                myprintf("MODO: STANDBY\r\n");
                myprintf("----------------\r\n");
                reset_leds();
                detener_reg();
                flagRepro = 0;
                reset_pid();
            break;
        case M_SIM:
                myprintf("----------------\r\n");
                myprintf("MODO: SIMULACIÓN\r\n");
                myprintf("----------------\r\n");
                reset_leds();
                flagRef = 0;
            break;
        case M_TEST:
                myprintf("----------------\r\n");
                myprintf("MODO: TEST\r\n");
                myprintf("----------------\r\n");
                reset_leds();
                HAL_GPIO_WritePin(LED_VERDE_GPIO_Port, LED_VERDE_Pin, GPIO_PIN_SET);
                flagRef = 0;
            break;
        case M_MEM:
                myprintf("----------------\r\n");
                myprintf("MODO: MEMORIA\r\n");
                myprintf("----------------\r\n");
                reset_leds();
                HAL_GPIO_WritePin(LED_AZUL_GPIO_Port, LED_AZUL_Pin, GPIO_PIN_SET);
            break;
    }
}
```

Fragmento de Código 6: Máquina de estados.

En todos los casos, se apagan primero los LEDs mediante `reset_leds()`, y luego se vuelve a encender el LED correspondiente al modo, si aplica. Además, se imprime un mensaje por terminal informando el nuevo modo activo, con excepción del Modo Error.

En los modos Simulación y Test, se reinicia la bandera de referencia de los servomotores, ya que en estos modos puede haber movimiento y no se puede garantizar que los servos mantengan su posición de referencia al regresar al Modo Espera. Por otro lado, en los modos Error y Espera se llaman funciones específicas para detener el registro en curso, cerrar archivos de escritura en caso de quedar alguno abierto, y reiniciar los controladores PID para la próxima vez que se utilicen.

6.5. Rutinas de servicio a interrupciones (ISR)

En esta sección se describen las rutinas de servicio a interrupciones implementadas en el proyecto. Estas rutinas permiten responder de forma inmediata y eficiente a eventos asincrónicos del sistema, como la recepción de datos por UART, el desbordamiento de temporizadores o la activación de entradas externas (en el caso del proyecto, el pulsador azul integrado en la placa de desarrollo).

Las funciones de interrupción fueron generadas utilizando la biblioteca HAL (Hardware Abstraction Layer) provista por ST. Esta biblioteca facilita la implementación de callbacks específicos que son llamados automáticamente por el sistema cuando se produce un evento de interrupción.

A continuación, se detalla el comportamiento de cada una de las interrupciones utilizadas.

6.5.1. Interrupción por recepción de mensajes vía UART

Cada vez que se recibe un carácter a través de la interfaz UART, se ejecuta automáticamente la rutina de servicio `HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)`.

Manejo general del buffer

El sistema implementa un buffer de entrada carácter por carácter (`rx_buffer`), indexado mediante `rx_index`, que permite construir comandos completos hasta recibir un carácter terminador. La lógica de procesamiento se resume en los siguientes casos:

- **Terminadores de línea (`\r` y `\n`):** Al recibir un carácter de fin de línea (Enter), se completa la cadena con carácter nulo, se procesa la entrada según el estado actual del sistema, se limpia el buffer y se muestra el prompt.
- **Caracteres de control (Backspace):** Si se recibe `0x08` o `0x7F` (backspace y delete en ASCII, respectivamente) y el índice es mayor a cero, se retrocede un carácter en el buffer y se envía la secuencia `\b \b` para actualizar visualmente la terminal.
- **Caracteres alfanuméricos:** Se almacenan secuencialmente en el buffer mientras no se supere el límite máximo. Además, se transmite el carácter recibido mediante `HAL_UART_Transmit()` para realizar el *echo* en la terminal.

Estado `INPUT_NORMAL`

En este modo operativo, únicamente se aceptan comandos que comienzan con el carácter ":". Si se cumple esta condición, se llama a la función `interprete()` para analizar el comando ingresado, como se mencionó antes. En caso contrario, se devuelve un mensaje de error.

Estado especial `INPUT_VERIF`

El estado `INPUT_VERIF` se gestiona de forma separada al resto del `switch`, ya que no requiere completar una línea. Cualquier carácter recibido en este estado ejecuta de inmediato la función `rutina_verificacion()`, permitiendo reintentar el chequeo de componentes sin necesidad de una entrada completa.

Otros estados

El sistema define otros estados de entrada asociados a operaciones específicas, como la confirmación para activar el registro, la selección de archivos almacenados o la verificación de los servomotores. Todos ellos fueron descritos previamente en la Sección 6.1.

En estos casos, los comandos ingresados no deben comenzar con el carácter ":", pero se conserva la lógica general de completar la línea con `Enter` para que el sistema procese la entrada. Cada estado implementa su propia validación, que puede consistir en detectar respuestas afirmativas o negativas ("y/n"), verificar formatos específicos (como un número de dos dígitos para el ID del archivo de registro), o controlar rangos válidos para los datos ingresados.

Reactivación de la recepción

Al finalizar la ejecución, se reactiva manualmente la recepción por interrupciones mediante `HAL_UART_Receive_IT(&huart2, &rx_char, 1)`, asegurando que el sistema permanezca receptivo a nuevas entradas del usuario en todo momento.

Nota: Cabe aclarar que la comunicación UART en este proyecto se implementa de forma híbrida: mientras que la recepción es por interrupciones, la transmisión se realiza de manera bloqueante a través de `HAL_UART_Transmit()`. Por este motivo, no se emplea una rutina de interrupción para la finalización de la transmisión.

6.5.2. Interrupción por cumplimiento de período de Timers

La configuración de los temporizadores utilizados se detalló en la Sección 5. Las rutinas siguientes se ejecutan automáticamente una vez que se cumple el período establecido para cada Timer.

La estructura general del callback `HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` se muestra en el Fragmento de Código 7. El parámetro `htim` permite identificar cuál de los temporizadores generó la interrupción, mediante la comparación con `htim->Instance`, y ejecutar la lógica correspondiente para cada uno.

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM3) {
        // Implementacion para TIM3
    }

    if (htim->Instance == TIM4) {
        // Implementacion para TIM4
    }

    else if (htim->Instance == TIM5) {
        // Implementacion para TIM5
    }
}
```

Fragmento de Código 7: Callback para interrupción por desbordamiento de temporizadores.

Timer 3

El TIM3 se utiliza para manejar el tiempo de espera entre consignas de movimiento consecutivas que se le envían a los servomotores durante la verificación inicial de componentes. Esto permite evitar el uso de funciones bloqueantes, como `HAL_Delay()`. Cada vez que se entra en la interrupción, se verifica si `input_state == INPUT_SERVO_WAIT_B1`, y en caso de cumplirse dicha condición, se invoca a la función `servo_sequence_task()`. Lo anterior puede corroborarse consultando el Fragmento de Código 8.

```
if (htim->Instance == TIM3) {
    if (input_state == INPUT_SERVO_WAIT_B1) {
        servo_sequence_task();
    }
}
```

Fragmento de Código 8: Callback TIM3.

Timer 4

El TIM4 fue configurado para manejar el parpadeo del LED verde que caracteriza al Modo Simulación. Como puede verse en el Fragmento de Código 9, lo que se hace al entrar en la interrupción es revisar si se está en `modo == M_SIM`, y de ser así, se usa la función `HAL_GPIO_TogglePin(GPIOx, GPIO_Pin)` para conmutar el estado del LED.

```
if (htim->Instance == TIM4) {
    if (modo == M_SIM) {
        HAL_GPIO_TogglePin(GPIOB, LED_VERDE_Pin);
    }
}
```

Fragmento de Código 9: Callback TIM4.

Timer 5

La rutina de interrupción asociada al TIM5 es más extensa, ya que se emplea tanto para el Modo Simulación (lectura del sensor, movimiento de los servomotores con o sin PID, registro de la sesión si este fue activado), como para la reproducción de registros en el Modo Memoria.

La interrupción se ejecuta cada 50 ms. Durante una simulación, en cada ciclo se realiza la lectura de los datos del sensor inercial, se aplica un filtrado y una corrección de offset (ver Sección 6.6.2), y luego se determina la acción de control a aplicar con cada servomotor. Si el controlador PID se encuentra desactivado, los ángulos filtrados se transforman directamente al rango de operación de los servos mediante una función lineal. En cambio, si el PID está activo, se calcula la salida del controlador para cada eje, aplicando luego una saturación y una zona muerta si corresponde (ver Sección 6.6.4).

Adicionalmente, si la bandera de registro `flagReg` está activa, se escribe una nueva línea en el archivo correspondiente (creado y abierto previamente), que incluye el *timestamp*, los ángulos sensados y las señales de control calculadas para ambos ejes. Esta lógica se resume en el diagrama de flujo de la Figura 31.

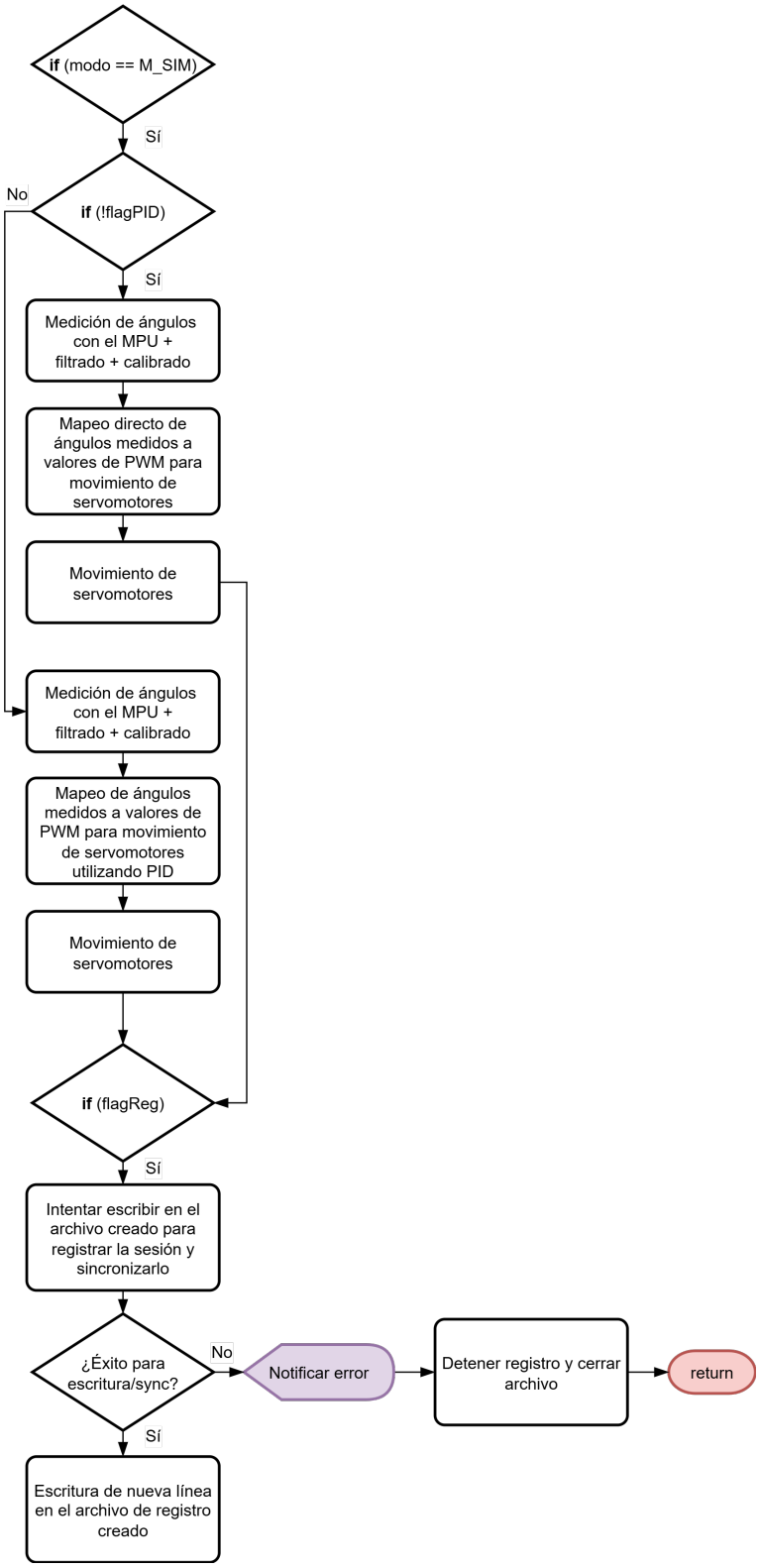


Figura 31: Diagrama de flujo: simulación mediante callback TIM5.

Por otro lado, en el Modo Memoria, esta interrupción permite la reproducción de una sesión previamente registrada. En cada ejecución, se intenta leer una nueva línea del archivo seleccionado. Si la lectura es válida, se extraen las columnas correspondientes a las señales de control, y se actualiza la posición de los servomotores con estos valores. Si se detecta un error

o el final del archivo, se notifica al usuario y se finaliza la reproducción. Este comportamiento se esquematiza en la Figura 32.

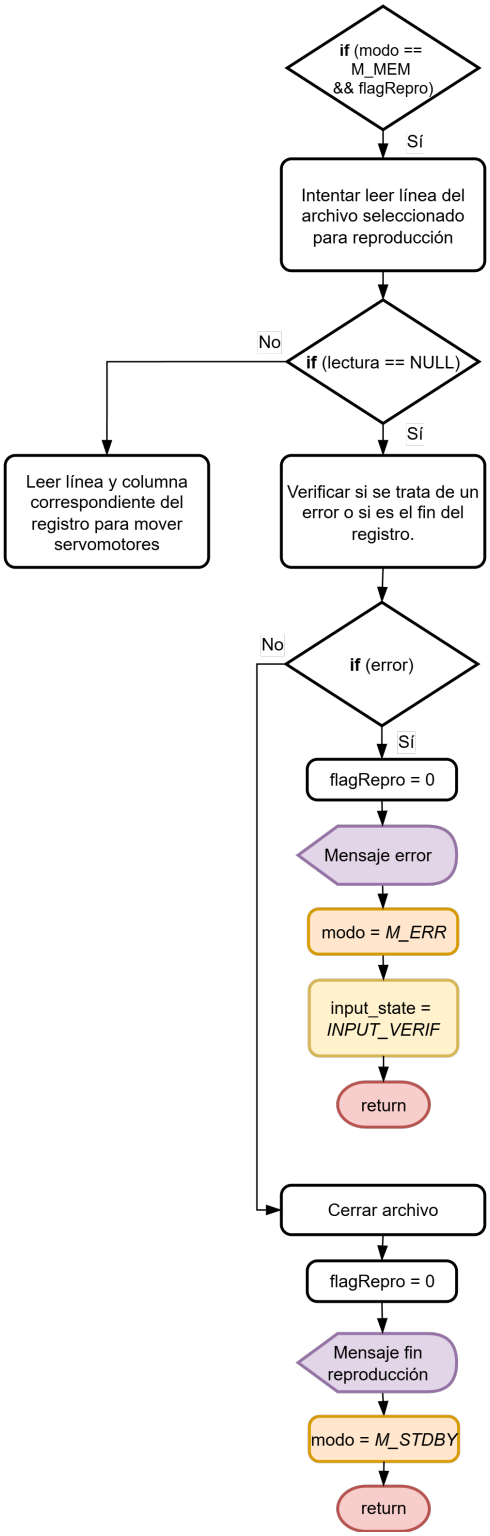


Figura 32: Diagrama de flujo: reproducción de registros mediante callback TIM5.

6.5.3. Interrupción por activación de pulsador

La interrupción por activación del pulsador se invoca durante la rutina de verificación de componentes, y se explicó su función en dicha sección. El código correspondiente puede verse en el Fragmento de Código 10.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (servo_actual != 0) {
        if (GPIO_Pin == B1_Button_Pin) {
            if (servo_actual == 1) {
                myprintf("Moviendo SERVO 1 (YAW)...\r\n");
            } else {
                myprintf("Moviendo SERVO 2 (PITCH)...\r\n");
                servo_actual = 2;
            }
        // Iniciar secuencia
        servo_sequence_active = 1;
        servo_step = 0;
        servo_timer = 0;
        }
    }
}
```

Fragmento de Código 10: Callback interrupción externa por botón.

6.6. Otras funciones

6.6.1. Funciones relacionadas al manejo de archivos en la tarjeta SD

A continuación, se describe el procedimiento general para interactuar con archivos utilizando la librería FATFS, tal como fue implementado en el sistema para registrar y reproducir sesiones.

Montaje del sistema de archivos

Antes de cualquier operación sobre archivos, es necesario montar el sistema de archivos en la tarjeta SD:

```
FATFS fs;
f_mount(&fs, "", 1);
```

- `&fs` es una estructura que contiene el contexto del sistema FAT.
- `""` se refiere al volumen lógico 0.
- `1` indica que el montaje debe realizarse de forma inmediata.

Creación o apertura de archivos

Para crear un archivo nuevo de registro en modo escritura:

```
FIL archivo;
f_open(&archivo, "reg_01.txt", FA_CREATE_NEW | FA_WRITE);
```

Para abrir un archivo ya existente en modo lectura:

```
f_open(&archivo, "reg_01.txt", FA_READ);
```

Escritura en archivos

Durante una sesión de simulación, se registran líneas de texto en el archivo abierto:

```
UINT escritos;
f_write(&archivo, linea, strlen(linea), &escritos);
```

Esto se utiliza tanto para guardar encabezados como para registrar mediciones sensadas y acciones de control, incluyendo el timestamp y los valores de los ejes Yaw y Pitch.

Lectura desde archivos

Al reproducir una sesión desde memoria, se utiliza la siguiente función para obtener línea por línea desde el archivo abierto:

```
char buffer[128];
f_gets(buffer, sizeof(buffer), &archivo);
```

Luego, el contenido del buffer se analiza para extraer las variables necesarias y accionar los servomotores en consecuencia.

Sincronización y cierre

Antes de cerrar el archivo, se puede forzar que todos los datos pendientes se escriban correctamente mediante:

```
f_sync(&archivo);
```

Finalmente, se cierra el archivo para liberar recursos:

```
f_close(&archivo);
```

Todas las funciones utilizadas pertenecen a la librería y pueden consultarse en la documentación oficial de ChaN [2]. Las mismas están diseñadas para retornar códigos de estado del tipo `FRESULT`, como `FR_OK`, `FR_EXIST` o `FR_DISK_ERR`, los cuales fueron empleados en el código para manejar errores durante la ejecución.

6.6.2. Funciones relacionadas al MPU6050

En esta sección se explica todo lo relacionado con el sensor MPU6050, incluyendo la librería desarrollada con funciones para acceder a los registros del módulo, el filtro implementado para solucionar problemas en las mediciones, y el método que se utiliza para la calibración del sensor.

Librería MPU

Para el manejo del sensor MPU6050 se implementan diferentes funciones que permiten tanto la lectura como la escritura en sus registros internos.

En el Fragmento de Código 11 se muestra la rutina de inicialización del sensor. Se establece la comunicación I2C y se verifica que el dispositivo responde correctamente mediante la lectura del registro `WHO_AM_I`. Si la respuesta coincide con la esperada, se procede a desactivar el modo de reposo (sleep mode) escribiendo en el registro `PWR_MGMT`.

```

uint8_t mpu6050_init(I2C_HandleTypeDef *hi2c) {
    HAL_StatusTypeDef ret;
    uint8_t check = 0;
    uint8_t temp_data = 0;


    ret = HAL_I2C_Mem_Read(hi2c, MPU6050_ADDR, WHO_AM_I_REG, 1, &check, 1,
    MPU6050_I2C_TIMEOUT);
    if (ret != HAL_OK) return MPU6050_ERR_READ_REG;
    if (check != 0x68) return MPU6050_ERR_WHO_AM_I;


    ret = HAL_I2C_Mem_Write(hi2c, MPU6050_ADDR, REG_PWR_MGMT, 1, &temp_data, 1,
    MPU6050_I2C_TIMEOUT);
    if (ret != HAL_OK) return MPU6050_ERR_PWR_MGMT_WRITE;

    return MPU6050_OK;
}
```

Fragmento de Código 11: Inicialización del sensor MPU6050.

El Fragmento de código 12 muestra la función `mpu6050_config` se encarga de configurar el sensor MPU6050 escribiendo en tres registros internos. En primer lugar, se configura la escala del giroscopio escribiendo en el registro `REG_CONFIG_GYRO` el valor correspondiente a `FS_GYRO_250`, que selecciona un rango de medida de $\pm 250°/s$. Luego, se configura el acelerómetro escribiendo en el registro `REG_CONFIG_ACC` el valor `FS_ACC_2G`, que establece un rango de $\pm 2g$. Finalmente, se configura el filtro digital pasa bajos escribiendo en el registro `REG_CONFIG` el valor `LPF_44`, que fija una frecuencia de corte de 44 Hz.

```c
uint8_t mpu6050_config(I2C_HandleTypeDef *hi2c) {
    HAL_StatusTypeDef ret;
    uint8_t temp_data;

    temp_data = FS_GYRO_250;
    ret = HAL_I2C_Mem_Write(hi2c, MPU6050_ADDR, REG_CONFIG_GYRO, 1, &temp_data, 1,
    MPU6050_I2C_TIMEOUT);
    if (ret != HAL_OK) return MPU6050_ERR_GYRO_CONFIG;

    temp_data = FS_ACC_2G;
    ret = HAL_I2C_Mem_Write(hi2c, MPU6050_ADDR,
    REG_CONFIG_ACC, 1, &temp_data, 1, MPU6050_I2C_TIMEOUT);
    if (ret != HAL_OK) return MPU6050_ERR_ACC_CONFIG;

    temp_data = LPF_44;
    ret = HAL_I2C_Mem_Write(hi2c, MPU6050_ADDR, REG_CONFIG, 1, &temp_data,1,
    MPU6050_I2C_TIMEOUT);
    if (ret != HAL_OK) return MPU6050_ERR_FILTER_CONFIG;

    return MPU6050_OK;
}
```

Fragmento de Código 12: Configuracion del sensor MPU6050.

La función `mpu6050_read()` del Fragmento de código 13 se encarga de obtener las mediciones completas del acelerómetro y giroscopio del sensor MPU6050. Esta función realiza una lectura secuencial de 14 bytes comenzando desde el registro `REG_DATA_START`, que corresponde al inicio de los registros de datos del sensor.

Los primeros 6 bytes leídos contienen los valores del acelerómetro en los tres ejes. Específicamente, los bytes 0 y 1 corresponden al eje X del acelerómetro, los bytes 2 y 3 al eje Y, y los bytes 4 y 5 al eje Z. A la medición de cada eje le corresponden 2 bytes, por lo que la función combina cada par de bytes mediante un desplazamiento de bits y una operación OR.

Los datos del giroscopio se encuentran en los bytes 8 al 13 de la secuencia leída. Los bytes 8 y 9 contienen la velocidad angular del eje X, los bytes 10 y 11 del eje Y, y los bytes 12 y 13 del eje Z. Notar que los bytes 6 y 7, que contienen los datos de temperatura del sensor, son leídos pero no procesados en esta función.

Los valores leídos se almacenan directamente en las variables apuntadas por los parámetros de entrada.

```c
uint8_t mpu6050_read(I2C_HandleTypeDef *hi2c, int16_t* x_acc, int16_t* y_acc,
int16_t* z_acc, int16_t* x_gyro, int16_t* y_gyro, int16_t* z_gyro) {
   HAL_StatusTypeDef ret;
   uint8_t data[14];

   ret = HAL_I2C_Mem_Read(hi2c, MPU6050_ADDR, REG_DATA_START, 1, data, 14,
   MPU6050_I2C_TIMEOUT);
   if (ret != HAL_OK) return MPU6050_ERR_READ_DATA;

   *x_acc = ((int16_t)data[0] << 8) | data[1];
   *y_acc = ((int16_t)data[2] << 8) | data[3];
   *z_acc = ((int16_t)data[4] << 8) | data[5];

   *x_gyro = ((int16_t)data[8] << 8) | data[9];
   *y_gyro = ((int16_t)data[10] << 8) | data[11];
   *z_gyro = ((int16_t)data[12] << 8) | data[13];

   return MPU6050_OK;
}
```

Fragmento de Código 13: Función de lectura de datos del sensor MPU6050.

Nota: Las direcciones de los registros utilizados están definidas en un archivo de encabezado mediante macros con nombres representativos, tales como `REG_DATA_START`. Esto permite mejorar la legibilidad y facilitar el mantenimiento del código. Además, todas las acciones realizadas en las funciones, ya sea de lectura o escritura, devuelven un código de retorno que permite verificar si la operación se completó correctamente o identificar el tipo de error que se produjo.

Filtrado de lecturas

Como se mencionó anteriormente, el sensor cuenta con un acelerómetro y un giroscopio, lo que permite estimar la inclinación del sistema mediante dos métodos distintos. Sin embargo, ambos enfoques presentan limitaciones que deben corregirse para obtener resultados confiables. Por esta razón, es necesario aplicar algún tipo de filtro con el objetivo de mejorar la precisión en la estimación de la orientación.

Medición a partir del acelerómetro: La orientación del sistema con respecto a los ejes x e y puede estimarse a partir de las siguientes fórmulas, utilizando las lecturas del acelerómetro:

$$\tan(\phi_{xyz}) = \frac{a_y}{a_z}$$

$$\tan(\theta_{xyz}) = \frac{-a_x}{\sqrt{a_y^2 + a_z^2}}$$

Este método de medición ofrece una buena estimación de la orientación estática del sistema, ya que no pierde la referencia. Sin embargo, presenta una gran sensibilidad al ruido de alta frecuencia y a vibraciones externas, debido a que responde directamente a las aceleraciones medidas en cada eje. Este comportamiento se puede observar en la Figura 33.

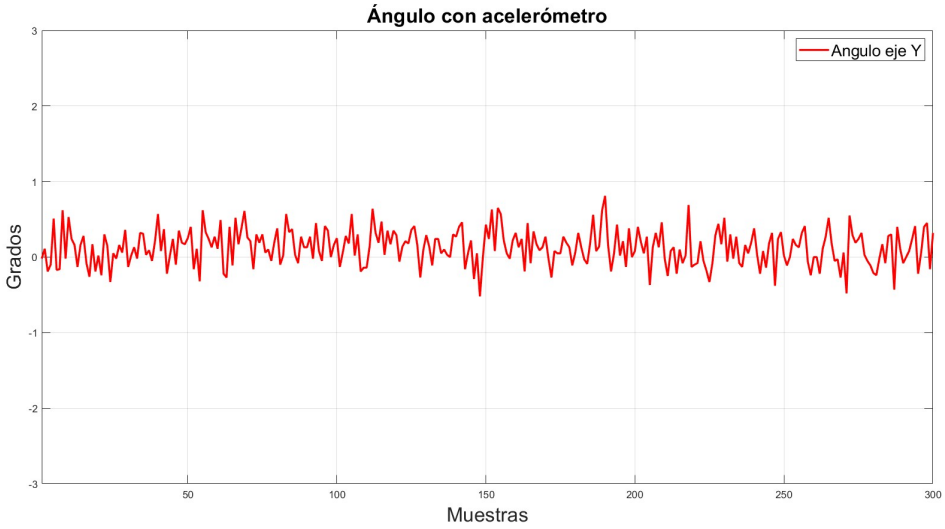


Figura 33: Ángulo estimado a partir del acelerómetro.

Medición a partir del giroscopio: Por otro lado, pueden utilizarse las lecturas de las velocidades angulares en cada eje para estimar la orientación, a partir de la integración de dichas variables:

$$\phi(t) = \phi_0 + \int_0^t g_x(\tau)\, d\tau$$

$$\theta(t) = \theta_0 + \int_0^t g_y(\tau)\, d\tau$$

Este método no presenta el problema del ruido de alta frecuencia, como ocurría en el caso del acelerómetro. Sin embargo, introduce otras limitaciones importantes. Por un lado, depende fuertemente de la condición inicial: si el sistema comienza a medir desde una posición que no corresponde a la referencia nula, tomará esa orientación inicial como cero, lo que generará un error persistente en la estimación.

Además, las mediciones del giroscopio suelen presentar un pequeño *offset*, que al integrarse en el tiempo acumula error, provocando el fenómeno conocido como *drift* o *deriva*, el cual afecta significativamente la precisión a largo plazo. Una forma común de mitigar este problema consiste en tomar n mediciones iniciales del giroscopio, calcular su promedio, y restarlo posteriormente a cada nueva medición de velocidad angular. De esta manera, se intenta desplazar la curva hacia el origen. Sin embargo, incluso aplicando esta corrección, no es posible eliminar completamente el *drift*. Puede observarse esta situación en la Figura 34.

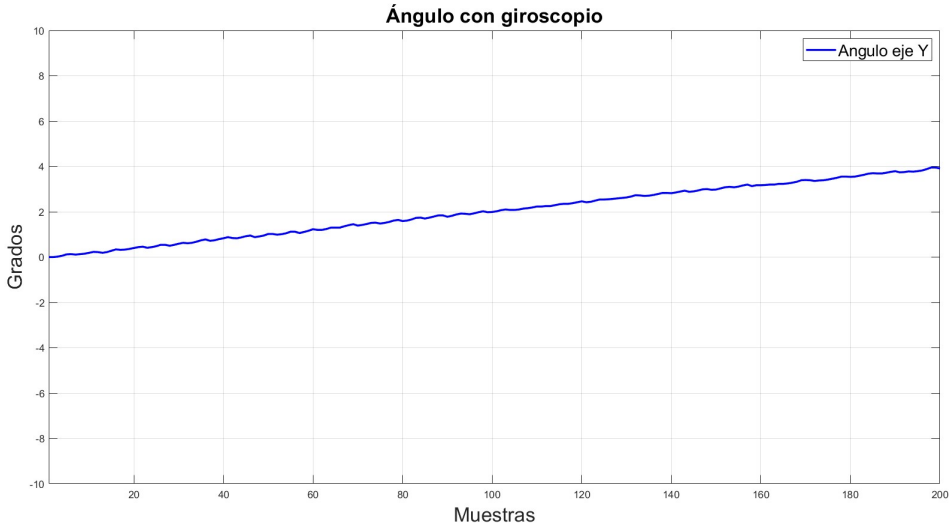


Figura 34: Ángulo estimado a partir del giroscopio.

Para abordar las limitaciones mencionadas anteriormente, se propone el uso de un **filtro complementario**, el cual consiste en realizar una ponderación entre las estimaciones obtenidas a partir del acelerómetro y del giroscopio. Esta técnica permite combinar lo mejor de ambos métodos: una medición estable y con bajo nivel de ruido, manteniendo la referencia absoluta proporcionada por el acelerómetro, y eliminando el efecto de la deriva proveniente del giroscopio. En la Figura 35 se observa el diagrama de bloques del filtro.

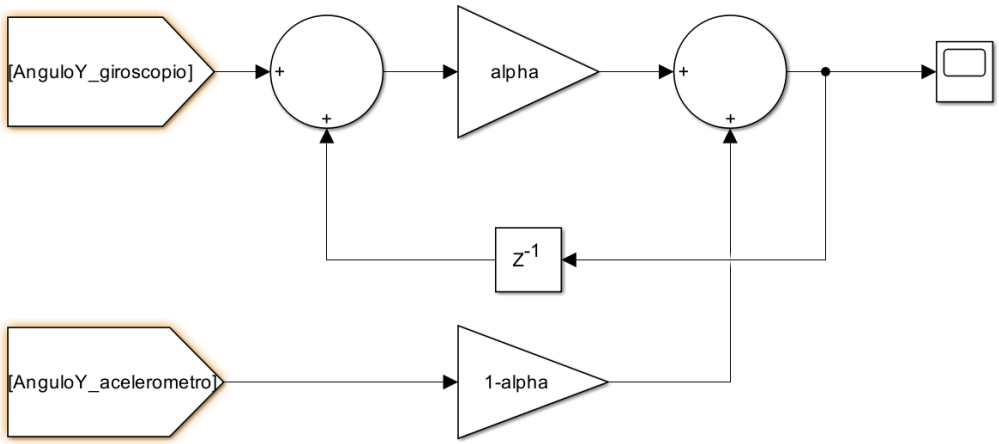


Figura 35: Diagrama de bloques de filtro complementario.

En la Figura 36 se muestra la comparación entre las tres estimaciones del ángulo respecto al eje Y: acelerómetro, giroscopio y filtro complementario. Se observa claramente que el filtro logra una estimación precisa y estable, combinando adecuadamente las ventajas de ambos sensores.

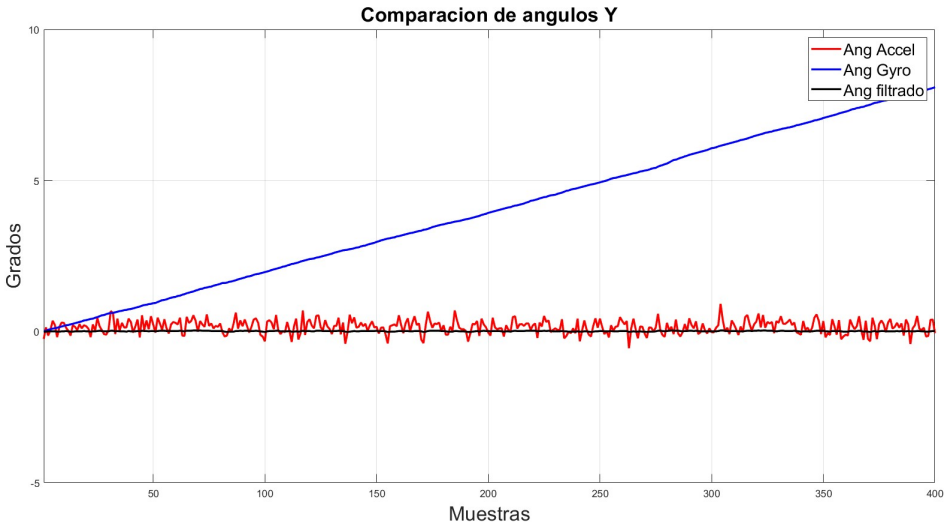


Figura 36: Comparación de estimaciones de ángulos.

La función principal que desarrolla esta tarea se encuentra en el Fragmento de código 14. Inicialmente se realiza la lectura de los datos crudos del sensor mediante la función `mpu6050_read()`. Si la lectura falla, la función maneja el error imprimiendo información de diagnóstico a través de `mpu6050_print_error()` y, en caso de estar en modo simulación, cambia el estado del sistema a modo error y configura la recepción de comandos del usuario para decidir si reintentar la operación.

Las mediciones del acelerómetro se convierten a ángulos de inclinación utilizando la función `cal_angulos_acelerometro()`, que calcula los ángulos mediante las fórmulas mencionadas anteriormente. Posteriormente se procesan los datos del giroscopio aplicando la corrección de offset. Los valores corregidos se convierten de unidades crudas a grados por segundo utilizando la función `mpu6050_gyro_raw_to_dps()` y se integran numéricamente.

Finalmente, se implementa el filtro complementario que combina linealmente los ángulos calculados por el acelerómetro y el giroscopio usando las constantes de ponderación `POND_GYRO` y `POND_ACC`.

Nota: Las constantes de ponderación se ajustaron por tanteo, en general se le daba más peso a la estimación del ángulo a partir del giroscopio, pero cuidando de no sobrepasarse porque esto hacía que el sistema respondiera con más lentitud. Los pesos de cada estimación fueron 70 % y 30 % para el giroscopio y el acelerómetro respectivamente.

```c
void lectura_filtrado_mpu(float dt) {

    // 1. Intentar leer el sensor
    uint8_t res = mpu6050_read(&hi2c1, &ax, &ay, &az, &gx, &gy, &gz);

    if (res != MPU6050_OK) {
        myprintf("\r\n");
        mpu6050_print_error(res);

        if (modo == M_SIM) {
            cambiar_modo(M_ERR);
            myprintf("*ALRT: FALLA EN LECTURA DE MPU DURANTE MODO SIM\r\n");
            myprintf("Escriba 'y' para reintentar o 'n' para salir\r\n");
            show_prompt();
            input_state = INPUT_ERR_MPU;
            HAL_UART_Receive_IT(&huart2, &rx_char, 1);
        }

        return;
    }

    // 2. Calcular ángulos del acelerómetro
    cal_angulos_acelerometro(ax, ay, az, &ang_accx, &ang_accy);

    // 3. Corregir medidas del giroscopio con los offsets
    gx_corregido = gx - (int16_t) offset_x_gyro;
    gy_corregido = gy - (int16_t) offset_y_gyro;

    gx_dps = mpu6050_gyro_raw_to_dps(gx_corregido);
    gy_dps = mpu6050_gyro_raw_to_dps(gy_corregido);

    // 4. Integración del giroscopio
    ang_gyrox = ang_filtrado_x + gx_dps * dt;
    ang_gyroy = ang_filtrado_y + gy_dps * dt;

    // 5. Filtro complementario
    ang_filtrado_x = (POND_GYRO * ang_gyrox + POND_ACC * ang_accx);
    ang_filtrado_y = (POND_GYRO * ang_gyroy + POND_ACC * ang_accy);
}
```

Fragmento de Código 14: Función de lectura y filtrado complementario del sensor.

Calibración del sensor

La función `calibrar_mpu()` establece los valores de referencia de orientación del sensor cuando el sistema se encuentra en reposo y en posición horizontal. Esta función implementa un proceso de calibración que toma 1500 muestras consecutivas del sensor para calcular los ángulos de referencia promedio.

Durante cada iteración del bucle de calibración, la función realiza la lectura de datos crudos del sensor mediante `mpu6050_read()`. En caso de detectar un error de lectura, la función maneja la situación imprimiendo información de diagnóstico con `mpu6050_print_error()`, cambia el estado del sistema a modo error. Se realiza el mismo proceso que el mencionado en el Fragmento de Código 14.

Los ángulos filtrados de cada muestra se acumulan en las variables `sum_ang_filtrado_x` y `sum_ang_filtrado_y`. Al finalizar el proceso de muestreo, se calculan los valores promedio dividiendo las sumas acumuladas entre el número total de muestras, estableciendo así los valores de referencia `ref_x` y `ref_y` que representan la orientación de reposo del sistema.

```c
uint8_t calibrar_mpu(void) {

    float sum_ang_filtrado_x = 0.0f;
    float sum_ang_filtrado_y = 0.0f;

    for (int i = 0; i < 1500; i++) {

        static float dt1 = 0.005f;

        // Verificar si lectura_filtrado_mpu detectó error
        uint8_t res = mpu6050_read(&hi2c1, &ax, &ay, &az, &gx, &gy, &gz);

        if (res != MPU6050_OK) {
            myprintf("\r\n");
            mpu6050_print_error(res);
            cambiar_modo(M_ERR);
            myprintf("ALRT: FALLA EN LECTURA DE MPU DURANTE CALIBRACIÓN\r\n");
            myprintf("Escriba 'y' para reintentar o 'n' para salir\r\n");
            input_state = INPUT_ERR_MPU;
            HAL_UART_Receive_IT(&huart2, &rx_char, 1);
            return res; // Retornar el código de error
        }

        // Continuar con el procesamiento normal
        cal_angulos_acelerometro(ax, ay, az, &ang_accx, &ang_accy);

        gx_corregido = gx - (int16_t) offset_x_gyro;
        gy_corregido = gy - (int16_t) offset_y_gyro;

        gx_dps = mpu6050_gyro_raw_to_dps(gx_corregido);
        gy_dps = mpu6050_gyro_raw_to_dps(gy_corregido);

        ang_gyrox = ang_filtrado_x + gx_dps * dt1;
        ang_gyroy = ang_filtrado_y + gy_dps * dt1;

        ang_filtrado_x = (POND_GYRO * ang_gyrox + POND_ACC * ang_accx);
        ang_filtrado_y = (POND_GYRO * ang_gyroy + POND_ACC * ang_accy);

        sum_ang_filtrado_x += ang_filtrado_x;
        sum_ang_filtrado_y += ang_filtrado_y;
    }

    ref_x = sum_ang_filtrado_x / 1500.0f; // Promedio
    ref_y = sum_ang_filtrado_y / 1500.0f;

    return MPU6050_OK; // Éxito
}
```

Fragmento de Código 15: Función de calibración del sensor MPU6050.

6.6.3. Movimiento de servomotores

Los servomotores utilizados en el sistema se controlan mediante señales PWM generadas por los timers del microcontrolador. El ciclo de trabajo de estas señales determina la posición del eje del servo, donde un pulso de 0.5 ms corresponde a un ángulo de 0°, y uno de 2.5 ms a 180°. Esto se traduce en un rango de valores entre 500 y 2500 unidades para el pulso de control.

Para convertir un ángulo deseado (en grados) a un valor de pulso compatible con el temporizador, se implementó la función `ang_to_pwm()`:

```
uint16_t ang_to_pwm(float angle) {
    return (uint16_t)(angle * (2000.0f / 180.0f) + 500.0f);
}
```

Fragmento de Código 16: Función para convertir un ángulo en grados a un valor de duty cycle correspondiente.

Esta función realiza un mapeo lineal entre el ángulo de entrada (0°-180°) y el valor correspondiente en el registro de comparación del timer.

Una vez calculado el valor, se actualiza el ciclo de trabajo del canal PWM correspondiente mediante la función `modificar_pwm()`, que emplea la macro `__HAL_TIM_SET_COMPARE` proporcionada por HAL:

```
void modificar_pwm(TIM_HandleTypeDef *htim, uint32_t channel, uint16_t pwm_pulse) {
    __HAL_TIM_SET_COMPARE(htim, channel, pwm_pulse);
}
```

Fragmento de Código 17: Función para modificar el duty cycle del canal PWM.

De este modo, el sistema puede modificar en tiempo real la posición de los servos a partir de referencias angulares, ya sea provenientes del usuario, de sensores o de una lógica de control.

6.6.4. Implementación del PID

Se implementan dos controladores PID independientes, uno para cada eje de control (yaw y pitch). Se hace uso de los mismos cuando el usuario envía el comando `:LC` (lazo cerrado, PID activado) y se comienza una simulación.

Ecuación del Controlador PID

La salida del controlador PID se calcula según la ecuación clásica:

$$u(t) = K_p \cdot e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt}$$

Donde $e(t) = r(t) - y(t)$ es el error entre la referencia (que será siempre 0° para los dos ejes, posición vertical del cohete) y el valor medido del ángulo.

Implementación Discreta

En la implementación digital, la ecuación se discretiza como:

$$u[k] = K_p \cdot e[k] + K_i \cdot I[k] + K_d \cdot \frac{e[k] - e[k-1]}{\Delta t}$$
$$I[k] = I[k-1] + e[k] \cdot \Delta t$$

donde Δt es el período de muestreo, en este caso correspondiente al período de la interrupción del TIM5, y $I[k]$ representa la acumulación del término integral.

Características Implementadas

Anti-Windup Condicional: Para prevenir la saturación del término integral, se implementa un mecanismo de anti-windup que detiene la acumulación cuando la salida alcanza los límites de saturación:

$$I[k] = \begin{cases} I[k-1] + e[k] \cdot \Delta t & \text{si } u_{min} \leq u_{temp}[k] \leq u_{max} \\ I[k-1] & \text{en otro caso} \end{cases}$$

Zona Muerta: Se implementa una zona muerta de $\pm 0{,}5°$ para eliminar oscilaciones en torno al punto de equilibrio:

$$e_{efectivo}[k] = \begin{cases} 0 & \text{si } |e[k]| < 0{,}5° \\ e[k] & \text{en otro caso} \end{cases}$$

Cuando el error está en la zona muerta, el término integral se reduce gradualmente mediante un factor de decaimiento: $I[k] = 0{,}95 \cdot I[k-1]$.

Estructura de los controladores

Cada controlador opera con los siguientes parámetros:

```c
typedef struct {
    float kp;           // Ganancia proporcional
    float ki;           // Ganancia integral
    float kd;           // Ganancia derivativa
    float prev_error;   // Error anterior
    float integral;     // Suma de errores (integral)
    float max_output;   // Límite máximo de salida
    float min_output;   // Límite mínimo de salida
} PID_Controller;
```

Fragmento de Código 18: Estructura para cada controlador PID.

Los mismos se inicializan con valores predeterminados de las ganancias, que luego el usuario puede cambiar durante la ejecución del programa.

Los límites de saturación se establecen en función del rango de movimiento de los servos: $u_{max/min} = \pm\frac{(\theta_{max}-\theta_{min})}{2}$. La división por dos del rango surge de que luego se aplica la corrección calculada desde la posición de referencia del servo, que está en la mitad del rango.

Flujo de Control

El proceso de control se maneja con la función `calculate_pid(*)`, y sigue la secuencia:

1. Recepción de la lectura del ángulo desde el sensor inercial (ya filtrado y calibrado).
2. Cálculo del error respecto a la referencia (0°).
3. Evaluación de la zona muerta.
4. Cálculo de los términos P, I, D.
5. Aplicación del anti-windup si corresponde.
6. Saturación de la salida final.
7. Conversión a señal PWM para el servo.

7. Etapas de montaje y ensayos realizados

El desarrollo del proyecto se llevó a cabo en múltiples etapas. Los primeros días de trabajo se centraron en el reconocimiento del microcontrolador a utilizar y en la familiarización con su entorno dedicado de programación, STM32CubeIDE. Gracias a la gran cantidad de tutoriales disponibles en línea y a la librería HAL provista por STM, la implementación del código para pruebas iniciales resultó sencilla.

Luego, inició el proceso de estudiar los módulos con los que se trabajarían, los protocolos de comunicación a utilizar, y la manera en la que se utilizaría todo junto para proveer las funcionalidades pensadas para la aplicación. Las primeras pruebas se centraron en el control de un solo servomotor, utilizando un potenciómetro como entrada "sensada", ya que para este punto todavía no se sabía cómo manejar el MPU. Se configuró una lectura por interrupciones del valor analógico, que luego se transformaba en una señal PWM para mover los servos, emulando el comportamiento que se tenía pensado implementar más adelante. Esta estrategia permitió entender rápidamente el funcionamiento de los actuadores.

Una vez superada esta instancia, el trabajo se dividió en dos frentes principales: por un lado, el manejo del módulo para tarjetas SD, y por otro, la lectura de datos del sensor MPU6050.

Para el sistema de registro, se utilizaron recursos de la documentación oficial de ChaN [2] y repositorios de código abierto, como el disponible en [3]. Se realizaron las configuraciones necesarias para la comunicación SPI1 y se habilitó el middleware FATFS. Las primeras pruebas consistieron en crear, escribir y leer archivos de texto desde la tarjeta SD. Se utilizó un adaptador USB–microSD para verificar desde la computadora que los archivos se generaban

correctamente. También se implementó una lógica de escritura periódica, en base a interrupciones, donde se guardaban mediciones simuladas provenientes del potenciómetro. Estas pruebas permitieron validar la estructura de los registros antes de integrarla al sistema completo.

En paralelo, se trabajó con una placa STM32 "bluepill" para realizar las pruebas con el MPU6050. A partir de videos y hojas de datos se aprendió a interactuar con los registros internos del sensor para configurar y obtener lecturas del acelerómetro y el giroscopio. Inicialmente se probaron algoritmos basados únicamente en el acelerómetro, pero se detectó un alto nivel de ruido en la señal. Se ensayaron distintas estrategias de filtrado, y finalmente, se implementó un filtro complementario que combina ambas fuentes de datos y proporciona una estimación estable de los ángulos.

Con estos módulos funcionales, se procedió a integrar todo en un único sistema, desarrollando la máquina de estados y el manejo completo de la lógica de operación. Se puso especial atención en contemplar fallos posibles, como la desconexión del sensor o errores de lectura en la SD, para evitar bloqueos del sistema y permitir al usuario reintentar la operación fallida.

Finalmente, se pasó a la etapa de montaje físico. Se imprimió en 3D (plástico PLA) el soporte cardánico utilizando el diseño encontrado en [6] (Figura 37), y se diseñó adicionalmente una base con Solid Edge que permite el movimiento libre del cilindro en el eje vertical (Figura 38). Ambos elementos juntos pueden verse en la Figura 39. Para el ensamble se utilizaron tornillos y alambres de acero que conectan los servos al soporte.

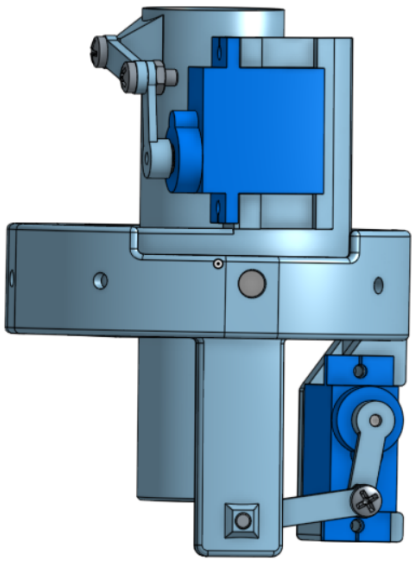


Figura 37: TVC Mount.

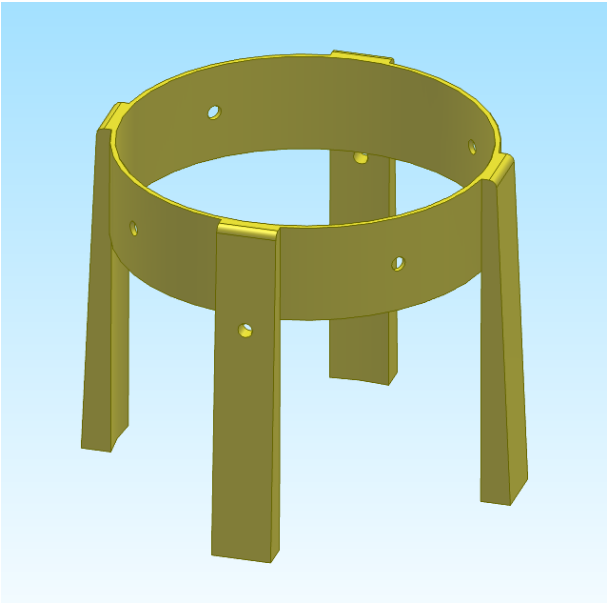


Figura 38: Soporte diseñado.

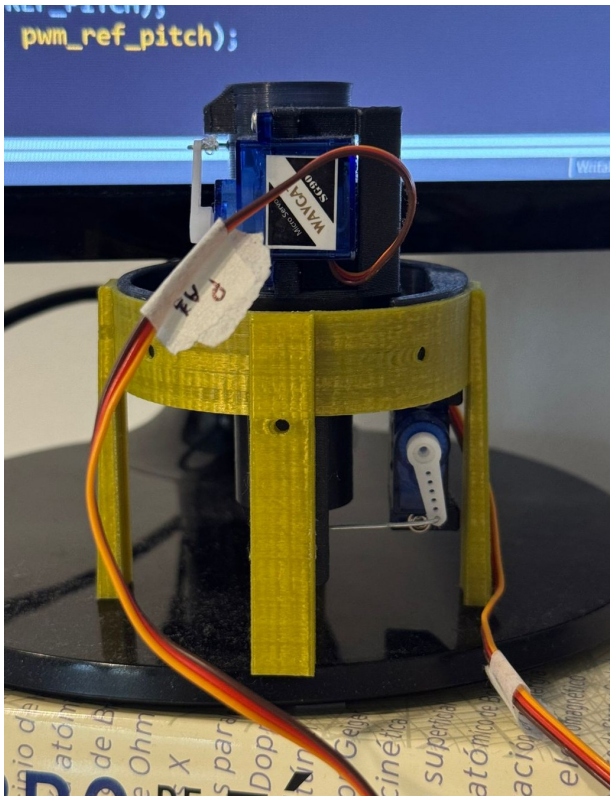


Figura 39: TVC Mount + Soporte.

Adicionalmente, se imprimió una base para el sensor inercial, obtenida de [7], la que permite mantener el MPU quieto y plano durante la etapa de calibración (Figuras 40 y 41).

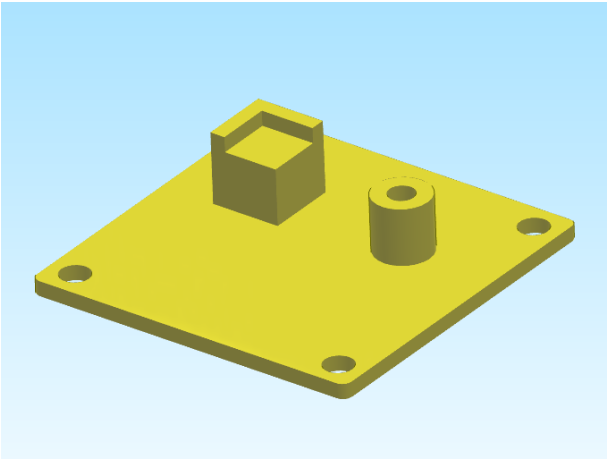


Figura 40: Base para MPU.

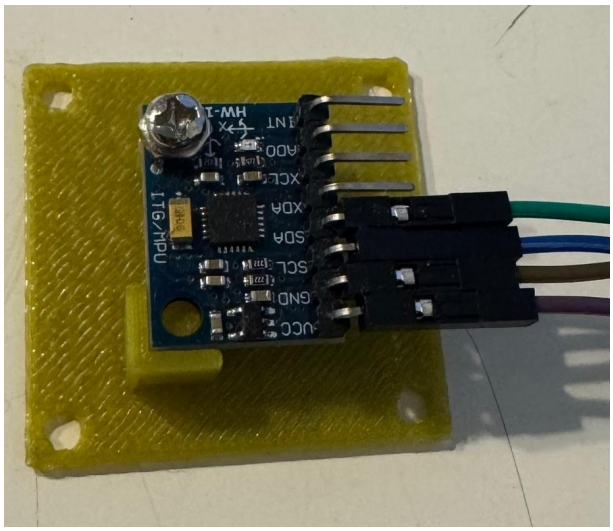


Figura 41: Base para MPU con MPU.

Una vez ensamblado el sistema, se utilizó el Modo Prueba para identificar los rangos de movimiento admisibles de los servomotores, procurando evitar colisiones del cilindro con el soporte. También aquí fue posible definir las posiciones de referencia que hacen que el cilindro se mantenga en una posición vertical.

El sistema completo puede observarse en la Figura 42. También se muestra un esquema de conexiones físico en la Figura 43.

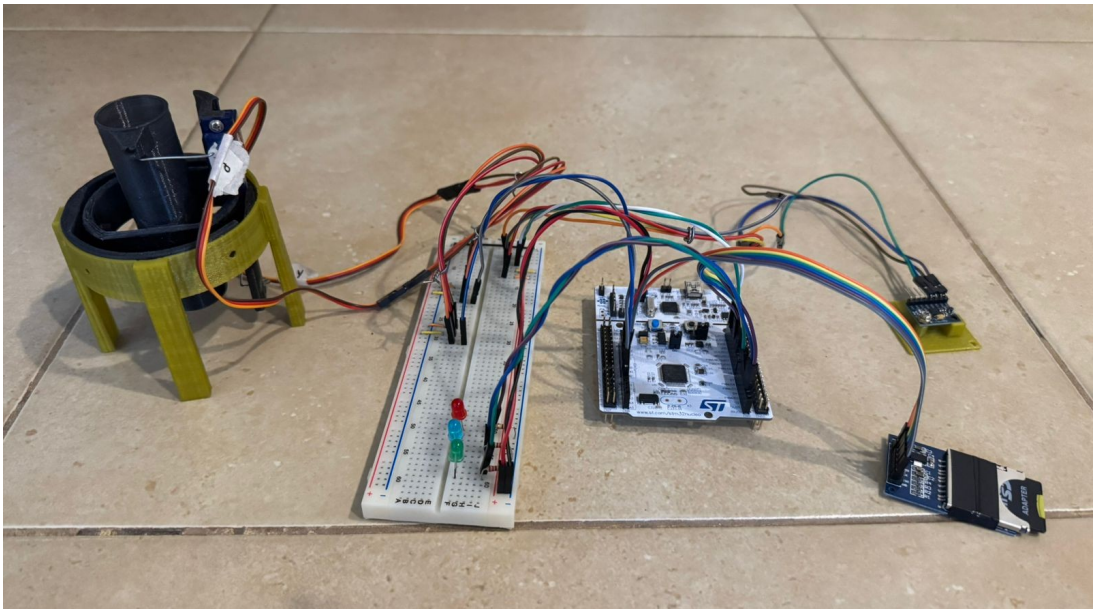


Figura 42: Maqueta del sistema completo.

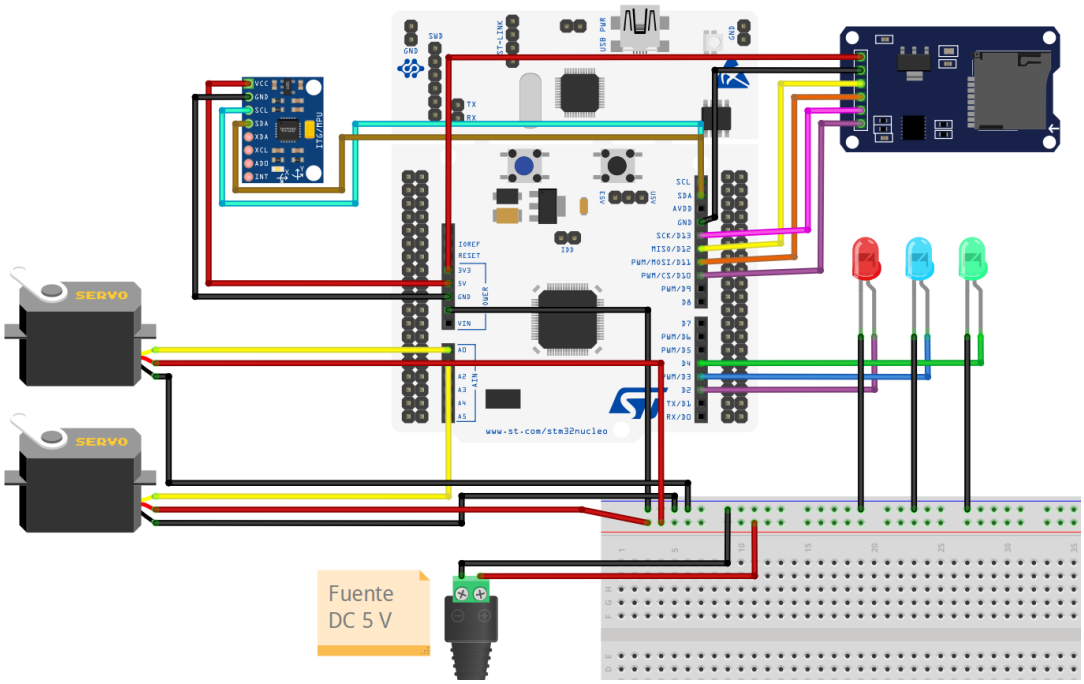


Figura 43: Esquema de conexiones físicas del sistema.

8. Resultados

A continuación, se presentan las principales especificaciones técnicas del sistema resultante.

Consumo eléctrico

El consumo total se estima en función de los siguientes componentes:

- **Servomotores SG90 (x2):** el consumo típico por unidad es de $250\,\mathrm{mA}$ en operación, con picos de hasta $500\,\mathrm{mA}$ bajo carga máxima a $5\,\mathrm{V}$.
- **MPU6050:** el sensor inercial tiene un consumo típico de $3{,}9\,\mathrm{mA}$ a $5\,\mathrm{V}$.
- **Módulo SD:** el consumo varía entre $20\,\mathrm{mA}$ y $100\,\mathrm{mA}$ según el acceso a datos.
- **STM32F446RE:** el microcontrolador presenta un consumo de alrededor de $100\,\mathrm{mA}$ en operación normal.
- **LEDs indicadores (x3):** se asume un consumo máximo de $20\,\mathrm{mA}$ por LED.

Es decir, el consumo estimado total resulta menor a $1\,\mathrm{A}$ en condiciones de carga máxima, aunque el promedio es considerablemente menor.

Autonomía

El sistema fue diseñado para operar con alimentación externa. No posee una fuente autónoma de energía ni se previó el uso de baterías en esta implementación.

Precisión

- **Servomotores SG90:** resolución teórica de $1°$.
- **MPU6050:** se configuró el acelerómetro con un rango de $2\,\mathrm{g}$ (16.384 LSB/g) y el giroscopio con un rango de $250\,°\,\mathrm{s}^{-1}$ (131 LSB/(°/s)). La resolución mínima teórica es de aproximadamente $0{,}000\,06\,\mathrm{g}$ para aceleración y $0{,}008\,°\,\mathrm{s}^{-1}$ para velocidad angular.

Velocidad de respuesta

El sistema actualiza su estado cada $50\,\mathrm{ms}$ ($20\,\mathrm{Hz}$), incluyendo:

- Lectura de sensores.
- Cálculo de control PID o transformación lineal de las mediciones, según el caso.
- Actualización de la posición angular de los servomotores.
- Registro en SD si corresponde.

Este tiempo de ciclo permite una respuesta suficientemente rápida para perturbaciones angulares suaves, coherente con las necesidades de control de actitud en cohetes amateur.

Rangos de trabajo

- **Ángulo de los servos:** el control se limita por software al rango $70°$–$120°$, aunque el dispositivo puede moverse entre $0°$–$180°$.
- **Ángulo de inclinación del cilindro del soporte:** el sistema está pensado para operar dentro de $\pm 5°$, en línea con los valores típicos en cohetería, donde las deflexiones de tobera suelen limitarse a 3–7°.
- **MPU6050:** las lecturas filtradas y calibradas del sensor se saturan en $\pm 10°$.

9. Conclusiones

Con el desarrollo de este proyecto se logró implementar un sistema funcional para simular el control del empuje vectorial, ampliamente utilizado en aplicaciones de cohetería amateur. Se integraron con éxito técnicas de programación estructurada, manejo de interrupciones, configuración de periféricos y control mediante máquinas de estado, dando lugar a una plataforma escalable hacia aplicaciones reales.

Si bien no fue posible realizar pruebas en condiciones de vuelo, el sistema cumplió satisfactoriamente con los objetivos propuestos, integrando de manera exitosa todas las funcionalidades previstas: lectura y procesamiento de datos del sensor inercial MPU6050 con filtro complementario, control de servomotores mediante señales PWM, registro de datos en tarjeta SD, implementación de un algoritmo PID, y desarrollo de una interfaz de usuario basada en comunicación serie.

Como principales mejoras para una versión de aplicación real se identifican las siguientes áreas:

Robustez mecánica: El reemplazo de componentes impresos en PLA por piezas metálicas de aluminio mejoraría significativamente la resistencia estructural del sistema ante condiciones extremas de vuelo. Adicionalmente, la utilización de servomotores con mayor torque permitiría el control de toberas de mayor tamaño y flujos de gases más intensos.

Sensor inercial: La adopción de sensores más precisos como el MPU6500 o MPU9250 proporcionaría mediciones con menor deriva y mayor resolución. La implementación de un filtro de Kalman en lugar del filtro complementario actual permitiría una fusión más sofisticada de los datos sensoriales y una mejor estimación del estado del sistema.

Redundancia del sistema: La incorporación de múltiples sensores inerciales distribuidos estratégicamente aumentaría la confiabilidad del sistema, en caso de que el sensor principal falle.

Comunicaciones inalámbricas: La inclusión de un módulo de comunicación de largo alcance, como LoRa o similar, eliminaría la dependencia de la conexión por cable USB y permitiría el monitoreo y control remoto durante el vuelo real del cohete.

Indicación remota: El traslado de los indicadores LED a una interfaz gráfica o panel de control en tierra proporcionaría mejor visibilidad del estado del sistema.

Autonomía energética: La implementación de una fuente de alimentación basada en baterías LiPo, con regulación y protección adecuadas, garantizaría la operación independiente del sistema sin limitaciones de cableado externo.

Referencias

[1] Arm Mbed. *STM32 Nucleo-F446RE – Pinout*. https://os.mbed.com/platforms/ST-Nucleo-F446RE/. Accedido en julio 2025.

[2] ChaN. *FatFs - Generic FAT Filesystem Module*. https://www.elm-chan.org/fsw/ff/. Accedido en julio de 2025. Publicado en 2024.

[3] Hayden Field. *cubeide-sd-card*. https://github.com/kiwih/cubeide-sd-card. Accedido en julio de 2025. Publicado en 2021.

[4] Kitronik Ltd. *Red 3mm Diffused LED Datasheet*. https://kitronik.co.uk/products/3501-red-3mm-diffused-led-275mcd. Accedido en julio de 2025.

[5] Khaled Magdy. *STM32 SD Card SPI FATFS Tutorial – Read and Write Files*. https://deepbluembedded.com/stm32-sd-card-spi-fatfs-tutorial-examples/. Accedido en julio de 2025. Publicado en 2022.

[6] Federico Mazzocca. *SG90 Gimbal 2 axis*. https://www.thingiverse.com/thing:4627449. Accedido en julio de 2025. Publicado en 2020.

[7] Ashraf Minhaj. *MPU6050 + Servo Pan Tilt*. https://www.thingiverse.com/thing:2911596. Accedido en julio de 2025. Publicado en 2018.

[8] STMicroelectronics. *STM32 Nucleo-64 boards (MB1136)*. Accedido en julio de 2025. Publicado en 2024.

[9] TDK Invensense. *MPU-6050 Datasheet*. https://www.alldatasheet.com/datasheet-pdf/view/1132807/TDK/MPU-6050.html. Accedido en julio de 2025. Publicado en 2016.

[10] PR TechTalk. *STM32 SD Card Interface via SPI and FatFS Tutorial (No RTOS)*. https://www.youtube.com/watch?v=PBIm8BCnbyQ&t=232s. Accedido en julio de 2025. Publicado en 2024.