

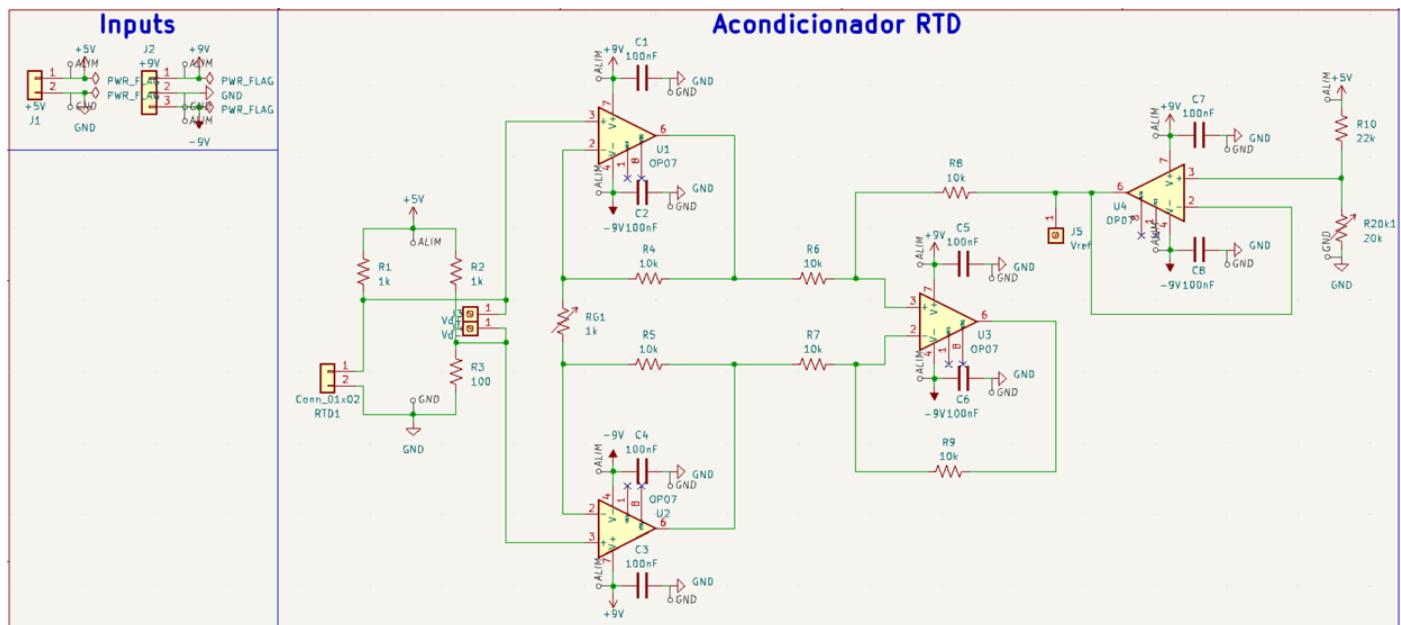
APUNTES PROYECTO B2 – ISE

Este documento tiene como objetivo la recopilación de apuntes del proyecto realizado en el B2, InverTech. Recogerá explicaciones tanto de la parte analógica como de la parte digital.

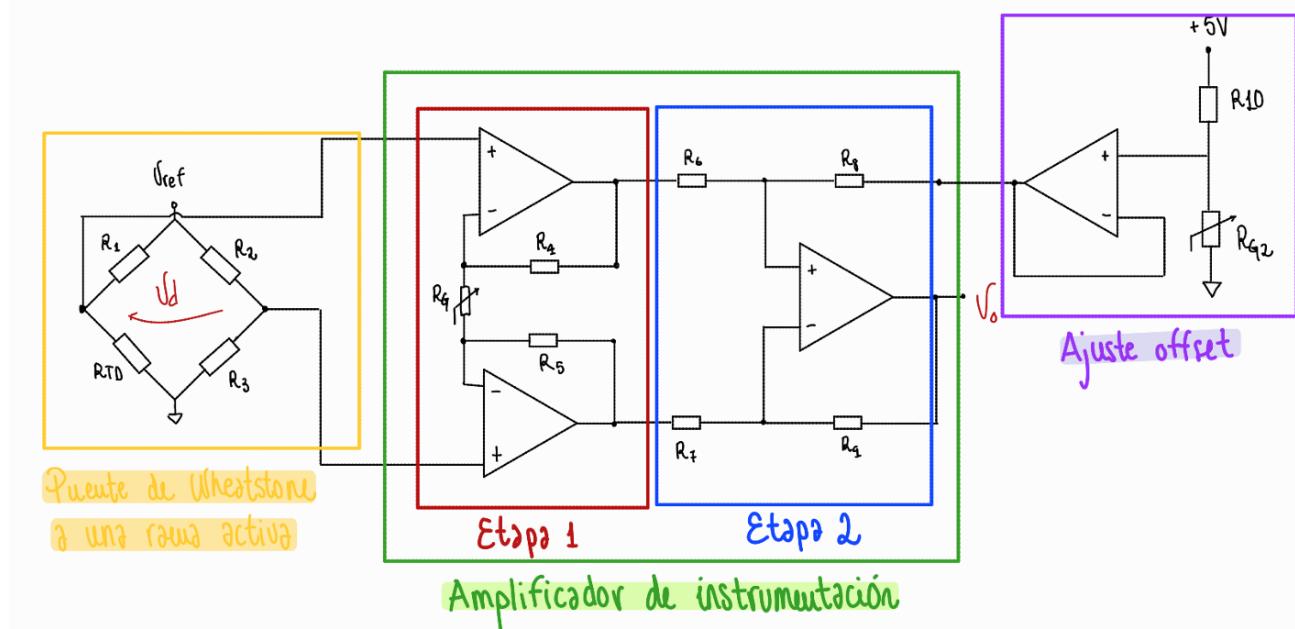
1. Parte analógica

1.1. Acondicionador RTD

Este circuito se caracteriza por estar compuesto por un puente de Wheatstone con una rama activa en el cual se introduce la RTD por el conector. Tiene una tensión de referencia de +5 V y el AI está ajustado para tener a la salida una variación de 0 a 3,3 V para poder introducirlo al ADC de la tarjeta. Se emplea una RTD de tipo Pt-100.



Los valores resistivos del puente de Wheatstone están elegidos para que este se encuentre balanceado cuando el valor de la Pt-100 son 0 °C que equivalen a 100 Ω. La equivalencia de las resistencias del puente de Wheatstone es de $10 \cdot R$ con respecto a las resistencias de abajo del puente. Esto ocurre ya que, si no fuese así, la corriente por las ramas aumentaría, provocando que la RTD se vea recorrida por más corriente provocando que se caliente.



$$U_d = U_1 - U_2 = U_{REF} \cdot \left(\frac{RTD}{10k + RTD} - \frac{R}{10k + R} \right) \rightarrow \begin{cases} U_d|_{T=-5^\circ C} = -7,96 \text{ mV} \\ U_d|_{T=45^\circ C} = 70,48 \text{ mV} \end{cases}$$

- Amplificador de Instrumentación:

- Etapa 1:

$$Ad_1 = 1 + \frac{2R_3}{R_G}$$

$$Ac_1 = 1$$

- Etapa 2:

$$Ad_2 = R_9 / R_7$$

$$Ac_2 = 0$$

Por lo tanto, el objetivo es tener la mayor ganancia posible en la etapa 1 para lograr mayor CMRR.

$$CMR_{AI} = \frac{Ad}{Ac} = \frac{\frac{Ad_1 \cdot Ad_2}{Ad_1 \cdot Ad_2}}{\frac{CMR_1 \cdot CMR_2}{CMR_1 \cdot CMR_2}} = CMR_1 \cdot CMR_2$$

- Cálculo de ganancias del AI:

$$U_o = A_{AI} \cdot U_d \Rightarrow A_{AI} = \left(1 + \frac{2 \cdot R_3}{R_G} \right) \Rightarrow U_o = \left(1 + \frac{2 \cdot R_3}{R_G} \right) \cdot U_d$$

$$\rightarrow \begin{cases} U_o = 0 \text{ V} \longrightarrow U_d = -7,96 \text{ mV} \\ U_o = 3,3 \text{ V} \longrightarrow U_d = 70,48 \text{ mV} \end{cases} \rightarrow Ad_1 = \frac{3,3 - 0}{70,48 \text{ mV} + 7,96 \text{ mV}} = 42 \text{ V/V}$$

$$A_{AI} = 1 + \frac{2 \cdot R_3}{R_G} \Rightarrow 42 \text{ V/V} = 1 + \frac{2 \cdot 10k}{R_G} \rightarrow R_G = \frac{20k}{41} = 487,8 \Omega$$

- Cálculo del CMRR del AI: $R_4 = R_5 = R_6 = R_7 = R_8 = R_9 = 10k\Omega @ 1\%$

De estos sabemos que $CMR_{AI} = CMR_1 \cdot CMR_2$.

$$CMR_1 = \frac{Ad_1}{AC_1} = 42 ; CMR_2 = \frac{Ad_2}{AC_2} = \frac{R_9/R_7}{\frac{1}{CMR_R} + \frac{1}{CMRAO}}$$

iDatasheet!

Common-Mode Rejection Ratio			CMRR
103	123	...	dB ...
103	123	...	dB ...
103	123	...	dB ...
103	123	...	dB ...

Caso peor

* $CMR_{AO} = 106 \text{ dB} \rightarrow \text{Caso peor para saber el mínimo rendimiento que tendriamos}$

$$CMR_{AO} = 10^{\frac{103}{20}} = 141253$$

$$* CMR_R = \frac{1 + R_9/R_7}{4 \cdot \varepsilon} = \frac{1 + 1}{4 \cdot 0.01} = 50$$

$$\textcircled{2} \quad CMR_2 = \frac{1}{\frac{1}{50} + \frac{1}{141253}} \times CMR_R = 50 ; \text{ Por ende: } CMR_{AI} = 50 \cdot 42 = 2100$$

$$CMRR_{AI} = 20 \cdot \log CMR_{AI} = 20 \cdot \log (2100) = 66,44 \text{ dB}$$

- Ajuste de offset: este amplificador actúa como sumador inversor para ajustar la tensión a la salida del AI en caso de que se salga del margen de salida

$$V_{OFF} = 5 \cdot \frac{R_{G2}}{R_{10} + R_{G2}} \quad \text{si} \quad \begin{cases} R_{G2} = 0 \Omega \rightarrow V_{OFF} = 0 \text{ V} \\ R_{G2} = 20k\Omega \rightarrow V_{OFF} = 2,38 \text{ V} \end{cases} \rightarrow \begin{array}{l} \text{Margen suficiente} \\ \text{para ajustar la tensión} \\ \text{de salida} \end{array}$$

Ecuación a la salida del AI

Ahora la ecuación que tenemos es: $V_o = A_{AI} \cdot V_d + V_{OFF} \rightarrow V_o = 0 \text{ V} \rightarrow V_d = -7,96 \text{ mV}$

$$0 = -42 \cdot 7,96 \text{ mV} + V_{OFF} \rightarrow V_{OFF} = 0,3342 \text{ V}$$

- Cálculo de R_{G2} :

$$0,3342 = 5 \cdot \frac{R_{G2}}{22k + R_{G2}} \rightarrow 0,3342 \cdot (22k + R_{G2}) = 5 \cdot R_{G2}$$

$$7352,4 + 0,3342 \cdot R_{G2} = 5 \cdot R_{G2} \rightarrow \underline{R_{G2} = 1575,8 \Omega}$$

1.1.1. Gestión de la tensión en el código:

La tensión a la salida del AI va al pin PC3 (Canal 13) de la STM32 configurado como ADC. En el código se hace una media con las últimas 10 medidas que llegan por el **ADC1 de la STM32**, realizando un promedio que se asemeje al real ya que se toman valores cada 250 milisegundos lo que provoca que varíe mucho. En el módulo de los ADCs no se realiza la conversión a la tensión si no a hexadecimal (16 bits). Este valor en hexadecimal se envía por una cola al ‘slave.c’ y posteriormente se agrupa en una cola general con el resto de medidas y se envía al ‘com.c’. Aquí se hace lo siguiente:

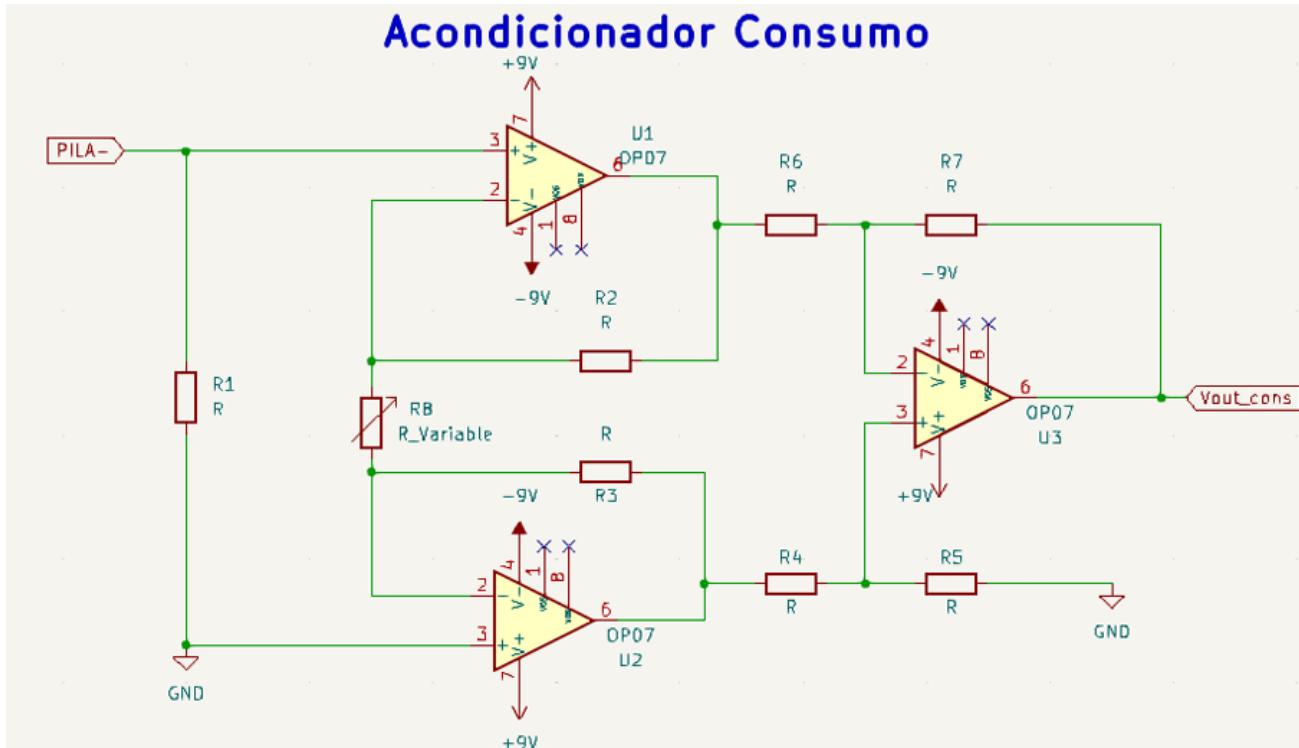
```
float rtdv_aux = ((float)v_rtd*3.3f)/4096.0f;
float rtd_V = (rtdv_aux*15.1745)-4.48f;
rtd_V = rtd_V * 100;
uint16_t rtd_scaled = (uint16_t)rtd_V;
DATATx[1] = (uint8_t)(rtd_scaled >> 8) & 0xFF; // Byte más significativo (MSB)
DATATx[2] = (uint8_t)(rtd_scaled & 0xFF); // Byte menos significativo (LSB)
```

Previo a enviarlo por la USART se realiza la conversión de hexadecimal a tensión empleando la fórmula del ADC que lo permite:

$$\text{Voltaje} = \frac{\text{Valor}_{\text{hexadecimal}} * 3.3}{4096}$$

Una vez hecho esto obtenemos la tensión y lo pasamos a temperatura usando la fórmula obtenida tras la caracterización de este acondicionador: **y = 0,0659x + 0,2231** que esta función relaciona la tensión con la temperatura ya que la x es temperatura y la ‘y’ es tensión. Dado a que nosotros queremos justo lo contrario, queremos que a raíz de la tensión obtener temperatura, por lo que es necesario hacer el inverso: **y = 15,174x - 4,48**. Una vez realizada la conversión a temperatura, se multiplica por 100 eliminando los posibles decimales. Después se castea a uint16_t para poder almacenar el valor en dos bytes. Después en DATATx[1] almacena el byte más significativo dividiéndolo entre 2^8 y después DATATx[2] almacena el byte menos significativo haciendo una AND con 0xFF.

1.2. Acondicionador consumo



Este circuito nos permitirá conocer el consumo con el que contará toda la tarjeta slave, incluyendo todo tipo de sensores y pcbs. Esto será posible ya que la resistencia R_{shunt} está colocada entre el negativo de la pila y la masa de la STM32, la cual es la masa común de todo el circuito de la slave. Además, esta configuración ayudará a reducir el CMRR del acondicionador lo más cercano a cero posible.

Para determinar el consumo vamos a estimar la corriente máxima de consumo que habrá en la tarjeta slave incluyendo todos los subsistemas. Con ese valor de corriente junto con la R_{shunt} (que no fue medida a 4 hilos) se obtiene la tensión máxima en bornas de la entrada del AI:

$$V_{d(máx)} = I_{máx} \cdot R_{shunt} = 500 \text{ mA} \cdot 0,1 = 50 \text{ mV}$$

La corriente mínima nunca será cero porque, aunque en el modo de bajo consumo deshabilitemos puertos, los operacionales del AI siguen alimentados, haciendo que consuman. Por lo tanto, la tensión máxima que queremos a la salida del AI es el fondo de escala del pin ADC de la tarjeta, que es de 3,2 V debido a un margen de seguridad. Dado a que tenemos la tensión máxima a la entrada y a la salida, podemos determinar la ganancia:

$$A_{AI} = \frac{3,2 \text{ V}}{50 \text{ mV}} = 64 \text{ V/V}$$

Para determinar el valor de la R_b del operacional usamos la fórmula de la ganancia del AI, que es la ganancia de la etapa 1 del AI:

$$A_{AI} = A_{d1} = 1 + \frac{2 \cdot R_3}{R_b} = 1 + \frac{2 \cdot 100k}{R_b} = 64 \frac{\text{V}}{\text{V}} \rightarrow R_b = 3174,6 \Omega$$

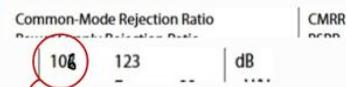
- Cálculo del CMRR del acondicionador de consumo:

- Cálculo del CMRR del AI: $R_2 = R_3 = R_4 = R_5 = R_6 = R_7 = 100\text{k}\Omega @ 1\%$

De estos sabemos que $\text{CMR}_{AI} = \text{CMR}_1 \cdot \text{CMR}_2$.

$$\text{CMR}_1 = \frac{A_{d1}}{A_{c1}} = 64 ; \quad \text{CMR}_2 = \frac{A_{d2}}{A_{c2}} = \frac{R_7/R_6}{\frac{1}{\text{CMR}_R} + \frac{1}{\text{CMR}_{AO}}}$$

i Datasheet!



Caso peor

* $\text{CMR}_{AO} = 106 \text{ dB} \rightarrow \text{Caso peor para saber el mínimo rendimiento que tendríamos}$

$$\text{CMR}_{AO} = 10^{\frac{106}{20}} = 199526$$

$$* \text{CMR}_R = \frac{1 + R_7/R_6}{4 \cdot \varepsilon} = \frac{1 + 1}{4 \cdot 0.01} = 50$$

$$\textcircled{*} \quad \text{CMR}_2 = \frac{1}{\frac{1}{50} + \frac{1}{199526}} \approx \text{CMR}_R = 50 ; \text{ Por ende: } \text{CMR}_{AI} = 50 \cdot 64 = 3200$$

$$\text{CMRR}_{AI} = 20 \cdot \log \text{CMR}_{AI} = 20 \cdot \log (3200) = 70,1 \text{ dB}$$

1.2.1. Gestión de la tensión en el código

La tensión a la salida del AI va al pin PC0 (Canal 10) de la STM32 configurado como ADC. En el código se hace una media con las últimas 10 medidas que llegan por el **ADC1 de la STM32**, realizando un promedio que se asemeje al real ya que se toman valores cada 250 milisegundos lo que provoca que varíe mucho. En el módulo de los ADCs no se realiza la conversión a la tensión si no a hexadecimal (16 bits). Este valor en hexadecimal se envía por una cola al ‘slave.c’ y posteriormente se agrupa en una cola general con el resto de medidas y se envía al ‘com.c’. Aquí se hace lo siguiente:

```
// Consumo
float consumption_V = (((float)v_crnt*3.3f)/4096));
consumption_V = consumption_V * 100;
uint16_t consumption_scaled = (uint16_t)consumption_V;
DATATx[3] = (uint8_t)(consumption_scaled >> 8) & 0xFF; // Byte más significativo (MSB)
DATATx[4] = (uint8_t)(consumption_scaled & 0xFF); // Byte menos significativo (LSB)
```

Previo a enviarlo por la USART se realiza la conversión de hexadecimal a tensión empleando la fórmula del ADC que lo permite:

$$\text{Voltaje} = \frac{\text{Valor}_{\text{hexadecimal}} * 3.3}{4096}$$

Una vez realizada la conversión a tensión, se multiplica por 100 eliminando los posibles decimales. Después se castea a uint16_t para poder almacenar el valor en dos bytes. Después en DATATx[3] almacena el byte más significativo dividiéndolo entre 2⁸ y después DATATx[4] almacena el byte menos significativo haciendo una AND con 0xFF.

1.3. Alimentación

Para alimentar de forma autónoma a la tarjeta slave se ha empleado un regulador 7809, ya que se pretende alimentar a la STM32 con +9V. Además, se hace uso del LMC7660 que está conectado de forma paralela a la salida del regulador ya que convierte los +9 V en -9 V. El regulador 7809 alimenta también a los amplificadores operacionales OP07 y el LMC también, ya que estos permiten tensiones de alimentación del siguiente rango:

		MIN	MAX	UNIT
V _{CC+} ⁽²⁾	Supply voltage	0	22	
V _{CC-} ⁽²⁾		-22	0	V

Además, los OP07 no son rail-to-rail (no es capaz de sacar a su salida los valores a los que se alimenta), pero la diferencia entre la salida que queremos sacar para el acondicionador de RTD de 3,3 V y el acondicionador de consumo de 3,2 V no afecta a esto ya que la diferencia es de algo menos de 6 V.

- Cálculo de la tensión de Drop-Out del regulador:

Cuando las pilas de las baterías están cargadas al máximo, alcanzan una tensión nominal de 4.1 V, lo que hace que al haber tres en serie, la tensión máxima de las baterías es de 12.3 V, y que superemos la tensión de Drop-Out del regulador que según el datasheet es de 2 V:

Dropout Voltage	V _{Drop}	I _O = 1A, T _J = +25 °C	-	2	-	V
-----------------	-------------------	--	---	---	---	---

$$V_{in(\min)} = V_{out} + V_{dropout}$$

Por lo que, la tensión mínima a la entrada para garantizar la tensión de Drop-Out es de:

$$V_{in(\min)} = 8,9 V + 2 V = 10,9 V$$

2. Parte digital

2.1. Sensor luminosidad (BH1750)

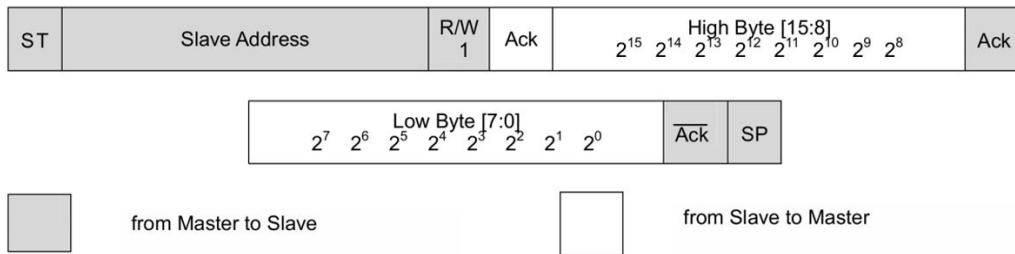
1º) La dirección de este sensor es la **0x23 (BH1750_ADDR)** que es esta ya que al pin ADDR del sensor no hay conectada tensión o al menos no le llega 0,3·Vcc.

2º) Se enciende el sensor enviándole el byte **0x01 (BH1750_POWER_ON)**

3º) Se configura la medida de la luminosidad de manera periódica y la resolución de la luminosidad a 1 lx enviándole el byte **0x10 (CONTINUOSLY_H_RESOLUTION)**

4º) En el while del hilo se llama a **Brightness_Reading()** leyendo cada segundo la luminosidad. Lo leído se almacena en un array de dos posiciones de uint8_t. El sensor devuelve el valor de la luminosidad en **Big Endian** lo que quiere decir que el sensor devuelve los MSB primero por lo que hay que desplazarlos y almacenarlos en la posición [0] para después concatenar los otros en la [1].

4) Read Format



ex)

High Byte = "1000_0011"
 Low Byte = "1001_0000"
 $(2^{15} + 2^9 + 2^8 + 2^7 + 2^4) / 1.2 \approx 28067 \text{ [lx]}$

2.2. Sensor de proximidad (HC-SR04)

1º) Se inicializan los pines del sensor y el TIM4. Cada segundo, llama a la función **getMeasure()** y lo envía por la cola.

2º) Dentro de la función, lo que se hace primero es reiniciar y arrancar el contador del TIM4, configurado para contar en microsegundos (**PRESCALER: 84 y APB1 está a 84 MHz**)

3º) Se llama a **Init_Sensor()** que genera un pulso a nivel alto de 10 µs (PB2) para iniciar la captura. Después va a esperar a un flanco de subida (primer while aunque ponga un RESET) ya que cuando haya un flanco de subida sale del while, iguala el counter a 'start' y entra en el segundo while. Cuando salga, iguala a 'end' para iniciar el cálculo. **NO SE USA TIM4 EN MODO INPUT CAPTURE. CON EL PB6 LEO EL ESTADO DEL PIN Y CON EL TIM4 GENERO LA CUENTA DE LA DURACIÓN DEL PULSO.**

4º) Calcula la duración del pulso HIGH. Si el contador se desborda lo vuelve a ajustar.

5º) Convierte el tiempo medido en distancia: $d = (\text{Tiempo} \cdot V_{\text{sonido}})/2 = (\text{Tiempo} * 0,0343)/2 = \text{Tiempo}/58$

$$\text{Distance} = \frac{\text{time}}{58} = \frac{\mu\text{s}}{\mu\text{s}/\text{cm}} = \text{cm}$$

6º) Calcula la distancia desde lo alto del depósito hasta el agua: **waterLevel = ALTURA_DEPO_CM - distance**

7º) Devuelve el nivel de agua en porcentaje: **else return (waterLevel / ALTURA_DEPO_CM) * 100.0f;**

2.3. Sensor CA y humedad (BME680)

IMPORTANTE: este sensor no está alimentado con VCC normal, sino que está alimentado mediante un GPIO. **A su vez, este sensor necesita ser configurado cada vez que se le solicite una medida.** Este sensor se puede configurar para usar el protocolo SPI o I2C dependiendo del estado del pin CSB. Se ha implementado una secuencia de inicialización software para forzar el modo I2C en el arranque del sistema. Se usa el **PG2 (Pág. 33 memoria)**. La función importante de este módulo es **Conf_bme680 ()**; que configura al sensor para comenzar toda la medición. El proceso de configuración es el siguiente:

1º) Inicialmente se hace un reset del sensor en el registro **reset (RESET_REG, 0xE0)** el valor de **0xB6** con la finalidad de hacer un reset para asegurar una inicialización correcta. Despues se esperan 5 ms para asegurar que el reset se ejecuta correctamente.

2º) Posteriormente se seleccionan los niveles de muestreo para los parámetros de humedad y temperatura, **ya que el gas se configura aparte**. Para ello se escribe en el registro **ctrl_hum (CTRL_HUM, 0x72)** el valor **OVERx16 = 101**, configurando el **sobremuestro en x16**.

3º) Tras esto, se configura el coeficiente del filtro IIR. Para ello escribe en el registro **config (CONFIG, 0x75)** el valor de **0x1C = 0001 1100**, ya que solo se escriben los **bits [4:2] = 127**.

4º) La temperatura se configura en el registro **ctrl_meas (CTRL_MEAS, 0x74)** el valor **ctrl_meas = 0x40** configurando a la temperatura con un índice de **sobremuestro en x2**. La presión no se configura.

5º) Posteriormente llama a la función que configura el gas **config_GAS ()**; para configurar el gas. En dicha función se escribe sobre el registro **gas_wait_x (GAS_WAIT, 0x64)** el valor de **0x59** lo que significa que [7:6] es un 4 y del [5:0] es un 25, por lo tanto, es un tiempo de configuración del calentamiento de 100 ms (25x4). Posteriormente se configura el registro **res_wait_x (RES_WAIT_0, 0x5A)** pasando el valor calculado ‘**res_heat**’ de la función **calc_res_heat ()** para que este registro lo convierta en un código. Por último, se enviará el registro **ctrl_gas_1 (CTRL_GAS_1m 0x71)** el valor de **RUN_GAS 0x10 (0001 0000)** lo que indica que comienza la conversión de gas (bit[4] = ‘1’) y el índice es igual a 0.

6º) Finalmente, vuelve a configurar el registro **ctrl_meas (CTRL_MEAS 0x74)** con el valor de **ctrl_meas = 0x81**, seleccionando el modo **FORZADO** ya que **mode[1:0] = 01**.

2.4. Bajo consumo

IMPORTANTE: el modo de bajo consumo en este proyecto se gestiona mediante **hardware**, conectando el **PB1** mutuamente entre ambas tarjetas núcleo. El master es el que gestiona en todo momento como se va a encontrar el slave.

1º) Primeramente, el pin **PB1** se configurará en modo interrupciones cada vez que se detecte un flanco de subida, con su correspondiente gestión en el archivo de interrupciones, además de darle prioridad suficiente a dicha interrupción.

2º) Cuando queremos entrar en el modo de bajo consumo, se llama a la función **Enter_SleepMode ()** que será en la que se gestionará todo este modo, que será el **SLEEP MODE** del bajo consumo. Dentro de esa función, primero se habilitan todos los relojes que no queramos dejar activos en este modo, en este caso son todos menos el B, para posteriormente configurarse en **modo analógico** a fin de reducir el consumo. Despues se deshabilitarán todos los relojes, excepto el B. Por último, antes de entrar en este modo, se deshabilitan todos los relojes, se llama a la función que inicializa el PB1 y se deshabilita el reloj del sistema, para terminar, introduciendo al micro en el modo de bajo consumo. La única forma de **salir de este modo es mediante una interrupción (WFI)**, en este caso del PB1. Cuando se detecte su interrupción configurada en flanco de subida, el micro se despertará, volviendo a activar el tick del sistema y todos los puertos GPIO junto al USB.

3º Gestión del bajo consumo desde el MÁSTER:

Como se ha mencionado antes, el MÁSTER gestionará el estado del PB1, para introducir en bajo consumo cuando se desee. Para ello, en ‘**Principal_Master.c**’ hay dos funciones que gestionan este estado: **despertar_Slave()** y **dormir_Slave()**. La primera genera un nivel alto constante al PB1 mientras que la segunda genera un nivel bajo constante.

4º Gestión del bajo consumo desde el SLAVE:

En el slave, la gestión del bajo consumo depende tanto del comando recibido por la UART enviado por el Master como del estado del pin PB1. En lo que al comando respecta, esta será en un primer momento escuchado por el hilo de recepción del Slave en la función '**UART_ListenCommand()**', donde se guardará la trama recibida en el array '**commandrx**'. Cuando en **commandrx[1]** tengamos el byte 0xAA, el slave pasará al estado LOW_POWER y tras esto se pasará a leer el estado del pin PB1 mediante la función '**readGPIO_Input(currentState)**' que dentro de esta función lo inducirá al modo de bajo consumo. Por el contrario, cuando se desee despertar al slave, desde el Master recibiremos un flanco de subida, saliendo del modo Sleep y habilitando la UART. Una vez hecho esto, el hilo de recepción volverá a estar escuchando y cuando nos llegue **commandrx[1] = 0xBB** significará que el master nos ha despertado. Después de esto, el hilo de transmisión del Slave enviará una trama de confirmación al master indicando en el penúltimo byte **DATATx[12]** se iguale al comando recibido 0xBB. Paralelamente a esto, este comando provocará que el slave pase a estar en el estado de WAKE_UP donde se verificará el nivel alto del pin PB1 y se encenderá el led rojo PB14 indicando que está despierto.

2.5. Led RGB

1º) Se inicializan los pines **PD11 (Verde), PD12 (Rojo) y PD13 (Azul)** que irán a la tarjetaMBED de aplicaciones. Dichos pines se inicializan a SET (cátodo común).

2º) En la función **ledsON()** se le pasa por parámetro el porcentaje del nivel de agua, y dependiendo de dicho valor, siempre que sea entre 0 y 100% se encenderá un color u otro.

Rango (%)	Color mostrado	LED activo
0.0 – 20.0	Rojo	PD13
20.1 – 70.0	Amarillo	PD13 + PD12 (Rojo + Azul)
70.1 – 99.9	Verde	PD12

2.6. Zumbador

1º) Inicialización del pin PA0 y el TIM2 en modo Output-Compare configurado a una frecuencia de 1 kHz (PRESCALER: 4200 y PERIOD: 20). **Un zumbador funciona con una onda cuadrada.**

2º) Después el hilo está constantemente esperando a un flag que llega del RFID. Dependiendo del flag recibido: **SPK_ON** (flag correcto), pitido de 500 ms; o **SPK_OFF** (flag incorrecto), pitido de 1500 ms.

2.7. LCD

1º) Se inicializa el LCD con sus respectivas funciones

2º) Por defecto, se muestra en la pantalla el valor estimado del bajo consumo con el que cuenta el circuito. Está esperando a que llegue el flag **ENTER_INV** enviado desde el RFID, indicando que se ha accedido al invernadero de manera presencial.

3º) Una vez recibido el flag se muestra una pantalla de bienvenida durante cinco segundos. Después, se muestra en bucle tres subpantallas distintas con intervalos de 2.5 s entre cada una, mostrando los parámetros del invernadero. Entre pantalla y pantalla se hace una espera activa esperando a que llegue el flag del RFID **EXIT_INV** indicando la salida del invernadero, asignando la variable **exitLoop** a 'true' y saliendo del bucle de las tres pantallas.

4º) Por último, se muestra una última pantalla de despedida durante cinco segundos y se vuelve al inicio del bucle while(1) mostrando la pantalla del consumo estimada en bajo consumo.

2.8. RFID

Previo a poner lo importante del módulo, vamos a hacer consideraciones previas. El RFID utiliza el bus SPI configurado a 20 MHz (SCK), utilizando además transferencias de datos **NO bloqueantes** ya que utiliza una Callback y en esta envía un flag cuando se completa una transferencia. Además, el **(Chip Select) CS NO** se configura en el RTE_Device.h, sino se configura mediante GPIO en el código, y lo importante es que está activo a **NIVEL BAJO**. Siempre que hay una lectura o escritura, primero se hace un **Write_Pin** a **RESET** y antes de salir de la función un **SET**.

1º) Obviando todo el código que se ejecuta en el while del hilo, la función más importante donde se escribe en varios registros es la función **TM_MFRC522_Init()** en la que primero se inicia el bus SPI y se hace un reset del sensor entero por posibles configuraciones internas que tenga. Lo primero de todo es la **configuración de los timers internos del sensor**.

2º) El primer registro en el que se escribe es **TModeReg (MFRC522_REG_TIM_MODE 0x2A)** en el que se configura el timer pasándole un **0x8D (10001101)**. Después se configura el registro **TPrescalerReg MFRC522_REG_TIM_PRESCALER (0x2B)** en el que se configura el prescaler con **0x3E (00111110)**. Después se configura el registro **TReloadReg MFRC522_REG_TIM_RELOAD_L (0x2D)** con un **30 (0x1E)** y el registro **MFRC522_REG_TIM_RELOAD_H (0x2C)** con un **0**. El valor de RELOAD está dividido en estos dos registros, pero se llaman igual. (**Mirar a partir de la página 37 del Datasheet**)

3º) Después se configura la ganancia de la antena de la tarjeta RFID mediante el registro **RFCfgReg (MFRC522_REG_GAIN_CFG 0x26)** escribiendo un **0x70 (01110000)**, lo que indica que se configura a 48 dB.

4º) Posteriormente se configura la transmisión a modulación ASK al 100% mediante el registro **TxAutoReg (MFRC522_REG_TX_AUTO 0x15)** escribiendo en este un **0x40 (0100 0000)** configurando la transmisión como se ha indicado.

5º) Por último, se configura el registro **ModeReg (MFRC522_REG_MODE 0x11)** el cual define los ajustes generales para la transmisión y recepción. Se escribe el valor **0x3D (00111101)** en el cual se configura lo siguiente:

- Bit 7: MSBFIRST = 0 (CRC check LSB first)
- Bit 5: TXWaitRF = 1 (Transmisor only starts if exist RF)
- Bit 3: PolMfin = 1 (MFIN polarity high active)
- Bits [1:0]: CRCPreset = 01 (Initial value CRC = 6363h) -> Is the standard for RFID communication

6º) Por último se activa la antena: **TM_MFRC522_AntennaOn();**

7º) Dentro del bucle while se espera al flag **(READID)** del timer de 1 segundo para que esté comprobando cada segundo si se ha pasado una tarjeta. La segunda función importante es **TM_MFRC522_Check()** que devuelve un booleano si se ha leído una tarjeta correcta. Si se da el caso, se pasa a comparar si el UID es uno de los que están en nuestro sistema y posteriormente se hace toda la lógica implementada. De primeras **insideGreenhouse = false** ya que se supone que de primeras estás fuera del invernadero. Al pasar la tarjeta siempre que **NO ESTEMOS EN LA WEB**, **insideGreenhouse = true**. La próxima vez que volvamos a pasar una tarjeta **VÁLIDA**, entrará en el else, significando que se va a salir del invernadero.

2.9. Memoria EEPROM AT24C256

DATOS A TENER EN CUENTA DE LA MEMORIA: Se encuentra organizada en **512 páginas de 64 bytes** cada una. Requerirá de una alimentación que se encuentre entre **2,7V y 5,5V**. Se empleará el protocolo **I2C** como medio de comunicación, con una frecuencia de **400 kHz** (modo fast). Se puede realizar la operación de escritura como de lectura byte a byte o por páginas (**64 bytes**). La dirección base de la memoria será la **[1010A₂ A₁ A₀ R/W]**, donde los últimos bits dependerán de los pines hardware. Por lo tanto, si estos pines se encuentran conectados a masa, la dirección de la memoria será la **0x50** y las direcciones a la hora de realizar la operación de escritura será la **0xA0** y la de lectura será la **0xA1**.

1º) Secuencia para realizar la operación de escritura en la memoria de un byte [máx. 64 bytes x págs]:

La función que usamos para escribir en la EEPROM es '**writememory()**' que la forma de enviar datos a la memoria es usando el **MasterTransmit**, cuya secuencia de pasos es:

1. START
2. Enviar dirección del esclavo [0x50] + "0" [escritura]
3. Enviar dirección alta [MSB]
4. Enviar dirección baja [LSB]
5. Enviar dato a escribir
6. STOP

Dentro de **I2Cdrv->MasterTransmit (ADDRESS_MEMORY, WriteData, longwrite, false)**; se envía la dirección del esclavo **ADDRES_MEMORY 0x50**; **WriteData**, es la dirección interna [2 bytes] + los datos que quiera escribir; **longWrite**, longitud del array a transmitir por I2C; y por último el campo booleano que indica si la transferencia ha terminado.

Esta función es utilizada en una función de nivel superior para guardar los bytes en memoria, denominada **guardar_medida(uint16_t base_addr, uint8_t* data, uint8_t data_len, uint8_t* index)**. En **index** se indica la cantidad de bytes a guardar en memoria. La función para calcular la posición dentro de una página es **addr = base_addr + (*index * data_len)**. Base_addr siempre es la constante del .h, ya después el índice y la posición en la página va variando a medida que escribimos.

```

void guardar_medida(uint16_t base_addr, uint8_t* data, uint8_t data_len, uint8_t* index) {
    // Calculamos cuantas entradas caben por página segun el tamaño de los datos
    uint8_t max_entries = (uint8_t)(data_len / 2); // tamaño de página/tamaño del dato = 64/2 = 32 (bytes)
    // Si el indice supera el límite, lo reseteamos
    if (*index >= max_entries) {
        *index = 0;
    }
    // Dirección en memoria para esta medida
    uint16_t addr = base_addr + (*index * data_len);
    // Preparar el buffer: 2 bytes de dirección + data_len bytes de dato
    write_buffer[0] = (addr >> 8) & 0xFF; // Dirección MSB
    write_buffer[1] = addr & 0xFF; // Dirección LSB
    for (uint8_t i = 0; i < data_len; i++) {
        write_buffer[2 + i] = data[i]; // Copiamos el dato al buffer
    }
    writememory(write_buffer, 2 + data_len); // Dirección + dato
    (*index)++; // Aumentamos el índice
}

```

Este código sirve para guardar medidas de dos bytes

en la última posición lo haremos poniendo 10 para todos los restantes.

Como siempre va a ser 2

Este lo utilizo para que si llega al final de la página, empieza a sobrescribir los datos mas antiguos.

base_addr es el comienzo de página que ha dejado anteriormente

PAG 0 → Temperatura [2 bytes] PAG 1 → Humedad [2 bytes] PAG 2 → calidad del aire [2 bytes] PAG 3 → luminosidad [2 bytes]

PAG 4 → hora y fecha, no se han cogido las medidas

PAG 5 → X datos que se nos pide escribir

Sabiendo por ejemplo que el comienzo de mi página para temperatura es 0x0000 entonces base_addr = 0x0000.

Y dependiendo de los datos que vamos escribiendo [index] vamos rellenando la página.

Temperatura, humedad y nivel de agua se guardan 2 bytes. CA es 1 byte. Luminosidad y hora son 3 bytes. **LA MEMORIA ESTÁ LIMITADA A GUARDAR ÚNICAMENTE 10 MEDIDAS DE CADA PARÁMETRO EN CADA PÁGINA.**

¿Cuál es el propósito del array **write_buffer[]** y por qué se escriben al menos 2 bytes al inicio?

El **write_buffer[]** almacenará los datos que se van a escribir en la EEPROM. Los primeros 2 bytes corresponden a la dirección interna de la memoria, seguido del dato real. El resto corresponde a dirección y luego los datos a guardar.

Estos bytes son 2: **write_buffer[0]** y **write_buffer[1]** son bytes que nos indican la dirección interna de donde voy a escribir los datos y luego el resto de posiciones de **write_buffer** son para los datos que se escriben en memoria, pero siempre a partir del índice de la escritura interior para no sobreescribir ningún dato.

2º Secuencia para realizar la operación de lectura de la memoria:

La función que usamos para leer de la EEPROM es ‘readmemory()’. En esta función hay un primer **MasterTransmit** donde la indicas a la memoria de que registro vas a leer (pasos 1-4) y un **MasterReceive** que es lo que lees de dicho registro (pasos 5-8):

1. START
2. Dirección del esclavo + “0”
3. Enviar dirección alta [MSB]
4. Enviar dirección baja [LSB]
5. REPEAT START
6. Dirección del esclavo + “1”
7. Recibes byte
8. STOP

Dentro de **I2Cdrv->MasterTransmit (ADDRESS_MEMORIA, Registerdirectory, 2, true);** se envía la dirección del esclavo **ADDRES_MEMORIA 0x50;** **RegisterDirectory**, que es un puntero que apunta a la dirección interna que voy a querer leer y por último el booleano a true para indicar que no hay más transferencias.

Después, la línea **I2Cdrv->MasterReceive (ADDRESS_MEMORIA, data, longlecture , false);** leo de donde había apuntado antes en la variable ‘**data**’ mientras que ‘**longlecture**’ me da los bytes que voy a querer leer.

Esta función es empleada por una función de nivel superior para leer medidas de la memoria, denominada **leer_medida(uint16_t base_addr, uint8_t* destino, uint8_t data_len, uint8_t* read_index)** donde **base_addr** es constante, **read_index** es el índice de lectura de cada página y **data_len** es el tamaño de lo que queremos leer que es exactamente igual que guardar.

A la función de **guardar_medida** se le llama en el archivo ‘uart.c’. A la función **leer_medida** se le llama en el archivo ‘HTTP_Server_CGI.c’.


```

77     printf("INVERT TECH se ha configurado correctamente\n\r");
78     dormir_Slave();
79     →→→→estadoMaster = STANBY; } Se le vuelve a dormir y va al STANBY
80     →→break;
81
82     →→→→case PRESENCIAL: → Se accede cuando se detecta una tarjeta correcta
83     →→→→if(insideGreenhouse) { flag = 0x20 e insideGreenhouse = true y web = false
84     →→→→printf("\n\n MODO PRESENCIAL \n\n\r");
85     →→→→comando[1] = MEAS; } Solicitamos medidas al hilo TX y lo envía al Slave
86     →→→→osThreadFlagsSet(TID_UartTx, 0xFF);
87     →→→→ledsON(meas_global.quantity);
88     →→→→auxiliarmedida = osThreadFlagsWait(MEAS, osFlagsWaitAny, osWaitForever);
89     →→→→osDelay(1000);
90     →→→→} else{ → Espera al hilo Rx de que han llegado las medidas
91     →→→→if(web){ → Si se pasa de nuevo el RFID (InsideGreenhouse = false)
92     →→→→estadoMaster = WEB; → Si se pulsa la web cambiamos de estado
93     →→→→} else{ → Si no, cambiamos al STANBY
94     →→→→    RTC_Alarm_Config();
95     →→→→    dormir_Slave();
96     →→→→    estadoMaster = STANBY;
97     →→→→}
98     →→→→}
99     →→→→
100    →→break;
101
102    →→→→case WEB: → Se accede cuando llega un flag desde el HTTP_Server-CGI.c tras pulsar el botón
103
104    →→ if(!web){ → y web = true
105    →→ if(insideGreenhouse == true){ → Si web = false, se comprueba si se está en modo
106    →→     estadoMaster = PRESENCIAL; → PRESENCIAL. Si no, cambia al estado STANBY.
107    →→ } else{
108    →→     RTC_Alarm_Config();
109    →→     dormir_Slave();
110    →→     estadoMaster = STANBY;
111    →→ }
112
113    →→→→} else{ → Esté en la web
114    →→→→printf("\n\n MODO WEB \n\n\r");
115    →→→→comando[1] = INIT_WEB; 0x01
116    →→→→ledsOFF();
117    →→→→osThreadFlagsSet(TID_UartTx, 0xFF);
118    →→→→osThreadFlagsWait(INIT_WEB, osFlagsWaitAny, osWaitForever);
119    →→→→osDelay(1000);
120
121    →→break;
122
123    →→}
124
125
126 void despertar_Slave(void){
127     →→ HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
128     →→ osDelay(5000); //5s → Tiempo suficiente para que el slave se despierte
129     →→ comando[1] = WAKE_UP; } Se envía al hilo TX para pedir al slave confirmación de que ha despertado
130     →→ osThreadFlagsSet(TID_UartTx, 0xFF);
131     →→ auxiliar_despertar = osThreadFlagsWait(0x80, osFlagsWaitAll, osWaitForever); //Cuando recibe 0xBB
132     →→ (Slave despierto)
133
134 static uint32_t dormir_Slave(void){
135     →→ uint32_t aux;
136     →→ comando[1] = SLEEP;
137     →→ osThreadFlagsSet(TID_UartTx, 0xFF);
138     →→ aux = osThreadFlagsWait(0x08, osFlagsWaitAll, osWaitForever); //Cuando recibe 0xAA (Slave va a dormirse)
139     →→ HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
140     →→ return aux;
141
142 static void init_despertar_Slave(void){
143     →→ // INICIALIZACIÓN DEL GPIO PARA DESPERTAR EL SLAVE
144     →→ __HAL_RCC_GPIOB_CLK_ENABLE();
145     →→
146     →→ GPIO_InitTypeDef despertar = {
147     →→     .Pin = GPIO_PIN_1,
148     →→     .Mode = GPIO_MODE_OUTPUT_PP,
149     →→     //→→.Pull = GPIO_PULLDOWN,
150     →→     .Speed = GPIO_SPEED_FREQ_VERY_HIGH
151     →→ };

```

ESTE ES EL BUCLE DE PRESENCIAL

Si se pulsa la web cambiamos de estado

Si no, cambiamos al STANBY

Si se pasa de nuevo el RFID (InsideGreenhouse = false)

Si se pulsa la web cambiamos de estado

Si no, cambiamos al STANBY

Se accede cuando se detecta una tarjeta correcta

Solicitamos medidas al hilo TX y lo envía al Slave

Espera al hilo Rx de que han llegado las medidas

Si se pulsa la web cambiamos de estado

Si no, cambiamos al STANBY

Se accede cuando llega un flag desde el HTTP_Server-CGI.c tras pulsar el botón

y web = true

Si web = false, se comprueba si se está en modo PRESENCIAL. Si no, cambia al estado STANBY.

Se lo enviamos al hilo TX para solicitar al Slave las medidas

Espera a que el hilo RX le envíe la confirmación de que ha llegado las medidas

NIVEL ALTO = DESPIERTO

Tiempo suficiente para que el slave se despierte

Se envía al hilo TX para pedir al slave confirmación de que ha despertado

Espera a recibir la confirmación del slave (hilo RX) de que ha despertado

Se envía al hilo TX para comunicar al slave que va a dormirse

Espera a recibir confirmación del slave de que está listo para dormirse

NIVEL BAJO = DORMIDO

```
152     →  
153     →HAL_GPIO_Init(GPIOB, &despertar);  
154     →HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);  
155  
156 }  
157
```

```

1 #include "uart.h"
2 #include "Principal_Master.h"
3 #include "AT24C256.h"
4
5 ///////////////UART///////////////////////////////
6 extern ARM_DRIVER_USART Driver_USART3;
7 static ARM_DRIVER_USART *UARTdrv = &Driver_USART3;
8
9 //////////////////Auxiliares///////////////////////
10 static bool uart_initialized = false;
11 static bool first_command_send = false;
12
13 //////////////////Variables de comunicacion UART///////////////////
14 static uint8_t rx_buffer[14];
15 uint8_t data_rx[12];
16
17 //////////////////EXTERN's///////////////////
18 extern Estados_Principal_t estadoMaster;
19 extern osTimerId_t tim_id1;
20
21 extern RTC_HandleTypeDef RtcHandle;
22 extern RTC_TimeTypeDef rtcTimeConfig;
23 uint8_t aShowTime[10] = {0};
24 uint8_t aShowDate[10] = {0};
25 uint8_t hh, mm, ss = 0;
26
27 void USART_Callback(uint32_t event) {
28
29     if (event & ARM_USART_EVENT_SEND_COMPLETE) {
30         →→→→→osThreadFlagsSet(TID_UartTx, ARM_USART_EVENT_SEND_COMPLETE);
31     }
32     if (event & ARM_USART_EVENT_RECEIVE_COMPLETE) {
33         →→→→→osThreadFlagsSet(TID_UartRx, ARM_USART_EVENT_RECEIVE_COMPLETE);
34     }
35 }
36
37 void UART_Init(void) {
38     // 1. Inicialización
39     int32_t status;
40     →→status = UARTdrv->Initialize(USART_Callback);
41     if (status != ARM_DRIVER_OK) {
42         return;
43     }
44     // 2. Encender el periférico
45     status = UARTdrv->PowerControl(ARM_POWER_FULL);
46     if (status != ARM_DRIVER_OK) {
47         return;
48     }
49     // 3. Configurar parámetros UART
50     status = UARTdrv->Control(
51         ARM_USART_MODE_SYNCHRONOUS | // Modo UART (asíncrono)
52         ARM_USART_DATA_BITS_8 | // 8 bits de datos
53         ARM_USART_PARITY_NONE | // Sin paridad
54         ARM_USART_STOP_BITS_1 | // 1 bit de parada
55         ARM_USART_FLOW_CONTROL_NONE, // Sin control de flujo
56         9600 // Baudrate
57     );
58     →→→→→ 9600 baudios (bps)
59     →→if (status == ARM_DRIVER_OK) {
60         printf("[UART] Inicializado correctamente a 9600 baudios\n");
61         uart_initialized = true;
62     }
63     →→// 4. Habilitar TX y RX
64     status = UARTdrv->Control(ARM_USART_CONTROL_TX, 1);
65     if (status != ARM_DRIVER_OK) {
66         return;
67     }
68
69     status = UARTdrv->Control(ARM_USART_CONTROL_RX, 1);
70     if (status != ARM_DRIVER_OK) {
71         return;
72     }
73
74 }
75
76 int32_t UART_SendCommand(uint8_t* command) {
77     if (!uart_initialized) {

```

COM MÁSTER

→ Significa que hay bit de START

Significa que por byte

transmitido (8 bits)

hay un bit de start

y un bit de stop.

↓

10 bits totales

(Por cada byte transmitido)

→ Hay bit de stop

↓

10 bits totales

(Por cada byte transmitido)

```

78     return ARM_DRIVER_ERROR;
79 }
80 int32_t result = UARTdrv->Send(command, 3); → Siempre se envían tres bytes al slave
81 if(result != ARM_DRIVER_OK) {
82     return result;
83 }
84 → osThreadFlagsWait(ARM_USART_EVENT_SEND_COMPLETE, osFlagsWaitAny, osWaitForever);
85     return ARM_DRIVER_OK;
86 }
87
88
89 int32_t UART_ReceiveData(uint8_t* data) {
90     if(!uart_initialized) {
91         return ARM_DRIVER_ERROR;
92     }
93     → return UARTdrv->Receive(data, 14); → Siempre se reciben 14 bytes del slave
94 }
95
96
97 /*-----*
98     Thread 'UART': UART handler for transmission
99     */
100 void UartTx (void *arg) { → Recibe los flags para enviar los tramas al slave
101     while (1) {
102         // 1. Esperar señal para iniciar comunicación/enviar comando
103         → osThreadFlagsWait(0xFF, osFlagsWaitAll, osWaitForever);
104         comando[0] = COMIENZO_TRAMA;
105         comando[2] = FINAL_TRAMA;
106         // 2. Enviar comando
107         → uint32_t timeout = 100;
108         while(timeout-- && UARTdrv->GetStatus().tx_busy) osDelay(1);
109         if (UART_SendCommand(comando) == ARM_DRIVER_OK) {
110
111     }
112 }
113 }
114
115 /*-----*
116     Thread 'UART': UART handler for reception
117     */
118 void UartRx (void *arg) {
119     while (1) {
120         // 1. Iniciar recepción
121         if (UART_ReceiveData(rx_buffer) != ARM_DRIVER_OK) {
122             //printf("[ERROR UART RECEIVE] Rx:Error al preparar recepción\n");
123         }
124         // 2. Esperar datos (con callback USART)
125         osThreadFlagsWait(ARM_USART_EVENT_RECEIVE_COMPLETE, osFlagsWaitAny, osWaitForever);
126         HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_7);
127         // Procesar los datos recibidos
128         if(rx_buffer[0] == COMIENZO_TRAMA) {
129             if(rx_buffer[13] == FINAL_TRAMA) {
130                 // Copiar los datos importantes que son 12 bytes (quitando el byte de comienzo de trama y
131                 final de trama)
132                 memcpy(data_rx, &rx_buffer[1], 12); → Copia únicamente los 12 bytes
133                 //Procesar trama recibida desde el slave
134                 procesar_trama(); → Alguna a la función
135             } else{
136                 // Trama no válida, resetear buffer
137                 memset(rx_buffer, 0, sizeof(rx_buffer));
138             }
139             if(comando[1]==0x01){ → Le pide al master que lo vea
140                 osThreadFlagsSet(TID_PRINC_MASTER, comando[1]);
141             } else if(comando[1]==0xAA){
142                 osThreadFlagsSet(TID_PRINC_MASTER, 0x08);
143             } else if(comando[1]==0xBB){
144                 osThreadFlagsSet(TID_PRINC_MASTER, 0x80);
145             }
146         }
147         else if(rx_buffer[0] == ERROR_TRAMA) {
148             if(comando[1]==0x01){
149                 osThreadFlagsSet(TID_PRINC_MASTER, comando[1]);
150             } else if(comando[1]==0xAA){
151                 osThreadFlagsSet(TID_PRINC_MASTER, 0x08);
152             } else if(comando[1]==0xBB){
153                 osThreadFlagsSet(TID_PRINC_MASTER, 0x80);
154             }
155         }
156     }
157 }
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
900

```

154 }
155 }else{ → Por si en otro byte cualquier que no se comunique
156 if(comando[1]==0x01){
157 osThreadFlagsSet(TID_PRINC_MASTER, comando[1]);
158 }else if(comando[1]==0xAA){
159 osThreadFlagsSet(TID_PRINC_MASTER, 0x08);
160 }else if(comando[1]==0xBB){
161 osThreadFlagsSet(TID_PRINC_MASTER, 0x80);
162 }
163 }
164 }
165 }
166 void procesar_trama(void){
167 → // 1. Procesar comandos →
168 if(comando[1]==0x01){
169 //////// Temperatura
170 uint16_t rtd_scaled = ((uint16_t)data_rx[0] << 8) | data_rx[1]; → Ya viene en temperatura (°C)
171 meas_global.temperatura = ((float)rtd_scaled) / 100.0f; // ~~meas_global.temperatura~~
172 printf("[Comando 0x01] Temperatura: 0x%02X 0x%02X °C(Hexadecimal) || Temperatura: %0.1f °C \n",
173 data_rx[0],data_rx[1],meas_global.temperatura);
174 //////// Consumo
175 uint16_t consumption_scaled = ((uint16_t)data_rx[2] << 8) | data_rx[3];
176 meas_global.consumo = (((float)consumption_scaled) / 100.0f)/0.0064f; // Te da el consumo en mA
177 original con decimales
178 printf("[Comando 0x01] Consumo: 0x%02X 0x%02X V(Hexadecimal) || Consumo: %0.2f mA\n",
179 data_rx[2],data_rx[3],meas_global.consumo);
180 //////// Luminosidad
181 uint32_t luz_100x = (data_rx[4] << 16) | (data_rx[5] << 8) | data_rx[6];
182 meas_global.lum = (float)luz_100x / 100.0f;
183 printf("[Comando 0x01] Luminosidad: 0x%02X 0x%02X 0x%02X lx(Hexadecimal) || Luminosidad: %.2f
184 lx\n", data_rx[4],data_rx[5],data_rx[6],meas_global.lum);
185 //////// Calidad del aire
186 meas_global.air_quality = (int)data_rx[7];
187 printf("[Comando 0x01] Calidad de aire: 0x%02X (Hexadecimal) || Calidad de aire: %d
188 IAQ\n",data_rx[7],meas_global.air_quality);
189 //////// Humedad
190 uint16_t hum_scaled = ((uint16_t)data_rx[8] << 8) | data_rx[9];
191 meas_global.humidity = (float)hum_scaled/ 100.0f;
192 printf("[Comando 0x01] Humedad: 0x%02X 0x%02X || Humedad: %.2f %%\n",
193 data_rx[8],data_rx[9],meas_global.humidity);
194 //////// Nivel de agua
195 uint16_t nivel_agua_recibido = ((uint16_t)data_rx[10] << 8) | (uint16_t)data_rx[11];
196 meas_global.quantity = (float)nivel_agua_recibido / 100.0; // Convertir a porcentaje
197 printf("[Comando 0x01] Nivel de agua: %02X %02X %% || Nivel de agua: %0.2f %%\n\n",
198 data_rx[10],data_rx[11],meas_global.quantity);
199 // [EEPROM] Guardar en la memoria solo en el modo STANDBY
200 if(estadoMaster == STANDBY){
201 guardar_temperatura(rtd_scaled);
202 guardar_humedad(hum_scaled);
203 guardar_luminosidad(luz_100x);
204 guardar_calidad_aire(meas_global.air_quality);
205 guardar_nivel_agua(nivel_agua_recibido);
206 RTC_Hora_Fecha(aShowTime, aShowDate);
207 hh = stimestructure.Hours;
208 mm = stimestructure.Minutes;
209 ss = stimestructure.Seconds;
210 guardar_hora(hh, mm, ss);
211 }
212 //////// Flag para hilo principal_master avisando de que ya recibió los datos y han sido
213 procesados →
214 osThreadFlagsSet(TID_PRINC_MASTER, comando[1]);
215 }else if(comando[1]==0xAA){
216 if(data_rx[11]==0xAA){
217 osThreadFlagsSet(TID_PRINC_MASTER, 0x08);
218 printf("[Comando %02X]ACK de confirmacion dormir al SLAVE\n", comando[1]);
219 }
220 }
221 } → Si el comando enviado en un 0xAA y se ha recibido el mismo, significa que el slave
222 está listo para dormirse y le comunica al master.

Aquí es donde se guardan los datos que no llegan del slave, únicamente cuando nos encontramos en el modo de Standby

Más vez procesados todos los datos que nos llegan del slave, se le envía al master un flag para saber.

```
222 }else if(comando[1]==0xBB) {  
223     if(data_rx[11]==0xBB){  
224         osThreadFlagsSet(TID_PRINC_MASTER, 0x80);  
225         printf("[Comando %02X]ACK de confirmacion despertar al SLAVE\n\n", comando[1]);  
226     } }  
227 } }  
228 } }  
229 } }  
230 } }  
231 } }  
232 } }  
233 } }  
234 }
```

De la misma forma, si el comando enviado es un 0xBB y se recibe el mismo, querrá decir que el slave se ha despertado correctamente, y le comunica al master.

```

1 #include "slave.h"
2
3 /*-----*
4 * Thread 1 'Thread_Name': Sample thread
5 *-----*/
6
7 /* Threads */
8 osThreadId_t tid_slave; // thread id
9
10 /* Colas de medidas */
11 MSGQUEUE_ADCs_t rx_ADCs;
12 MSGQUEUE_BME680_t rx_BME680;
13 MSGQUEUE_BH1750_t rx_BH1750;
14 MSGQUEUE_HCSR04_t rx_HCSR04;
15
16 /* Queue Objects*/
17
18 MSGQUEUE_OBJ_MEAS tx_medidas = {
19     .Voltaje_RTD = 0.0,
20     .humidity = 0.0,
21     .luminosity = 0.0,
22     .air_quality = 0,
23     .aquaLevel = 0.0,
24     .Voltaje_CONSUMPTION = 0
25 };
26
27 osMessageQueueId_t mid_Slave_Com_MsgQueue;
28 extern osMessageQueueId_t mid_Slave_Com_MsgQueue;
29
30 /* Functions prototypes */
31 int Init_Slave_Com_Queue(void);
32 void ThSlave(void *argument); // thread function
33 void InitializeSensors(void);
34 void processCommand(uint8_t cmd);
35 void RefreshMeas(void);
36
37 /* FUNCIONES AUXILIARES*/
38 void PrintSensorValues(void);
39
40 /* Variables */
41 float waterLevel = 0.0;
42 float luminosity = 0.0;
43 uint32_t RTD_Voltaje = 0;
44 uint32_t CONSUMPTION_Voltaje = 0;
45 extern uint8_t commandrx[3];
46
47 /* Structure */
48 typedef enum {
49     INITIALIZE,
50     WAKE_UP,
51     LOW_POWER,
52     ALL_MEAS
53 } StatesPrincipal_t;
54
55 StatesPrincipal_t currentState = LOW_POWER; //Por defecto
56
57 void readGPIO_Input(StatesPrincipal_t estado);
58
59 /* Externs */
60 extern void Init_I2C(void);
61 extern void Init_I2C_MASTER(void);
62 extern void RTD_pin_F429ZI_config();
63 extern void CONSUMPTION_pin_F429ZI_config();
64 extern void GPIO_HCSR04(void);
65
66
67 int Init_Slave(void) {
68
69     tid_slave = osThreadNew(ThSlave, NULL, NULL);
70     if (tid_slave == NULL) {
71         return(-1);
72     }
73
74     Init_Slave_Com_Queue();
75
76     return(0);
77 }

```



Compartido por slave y com

```

78
79 int Init_Slave_Com_Queue (void) {
80
81     mid_Slave_Com_MsgQueue = osMessageQueueNew(SLAVE_COM_MSG_QUEUE, sizeof(MSGQUEUE_OBJ_MEAS), NULL);
82     if(mid_Slave_Com_MsgQueue == NULL) {
83         return -1;
84     }
85
86     return (0);
87 }
88
89 void ThSlave (void *argument) {
90
91     InitializeSensors(); → Iniciar de primera los sensores
92     Init_PB1_WAKEUP(); → Iniciar PB1
93     while (1) {
94         // Insert thread code here...
95         /* Crear una función que se llame wait for command y dependiendo del comando se vaya a un estado
96         u otro */
97         //→→PrintSensorValues();
98         //→→osDelay(1000);
99         processCommand(commandrx[1]);
100
101     switch (currentState) {
102
103         //Here we will initialize the COM Thread
104         case INITIALIZE: //NO SE USA PARA NADA } NO SE USA
105             InitializeSensors();
106             break;
107
108         case WAKE_UP:
109             readGPIO_Input(currentState);
110             currentState = INITIALIZE;
111             break;
112
113         case LOW_POWER:
114             readGPIO_Input(currentState);
115             break;
116
117         case ALL_MEAS:
118             osMessageQueueGet(mid_MsgQueueADCs, &rx_ADCs, NULL, 0u);
119             osMessageQueueGet(mid_MsgQueueBME680, &rx_BME680, NULL, 0u);
120             osMessageQueueGet(mid_MsgQueueBH1750, &rx_BH1750, NULL, 0u);
121             osMessageQueueGet(mid_MsgQueueHCSR04, &rx_HCSR04, NULL, 0u);
122
123             tx_medidas.Voltaje_RTD = rx_ADCs.RTD_Vol;
124             tx_medidas.Voltaje_CONSUMPTION = rx_ADCs.CONSUMPTION_Vol;
125             tx_medidas.humidity = rx_BME680.humidity;
126             tx_medidas.air_quality = rx_BME680.air_quality;
127             tx_medidas.luminosity = rx_BH1750.lum;
128             tx_medidas.aquaLevel = rx_HCSR04.quantity;
129
130             osMessageQueuePut(mid_Slave_Com_MsgQueue, &tx_medidas, NULL, 100); Put que va al
131             break; com.c
132
133     }
134     //osThreadYield(); // suspend thread
135 }
136 }
137
138
139 void InitializeSensors (void) {
140     Init_HCSR04();
141     Init_ThBH1750();
142     Init_ThADCs();
143     Init_Thbme680();
144     Init_LEDs();
145 }
146
147 void PrintSensorValues(void) {
148     // printf("----- Medidas de los sensores ----- \n");
149     // printf("Voltaje RTD: %u mV\n", tx_medidas.Voltaje_RTD);
150     // printf("Humedad: %.2f %%\n", tx_medidas.humidity);
151     // printf("Luminosidad: %.2f lux\n", tx_medidas.luminosity);
152     // printf("Calidad del aire (indice): %d\n", tx_medidas.air_quality);
153     // printf("Nivel de agua: %.2f cm\n", tx_medidas.aquaLevel);
154 }
```

Se hacen gets de las colas de cada sensor para juntarlos en una única cola y enviarla al com.c

```

154 // printf("Voltaje de consumo: %u mV\n", tx_medidas.Voltaje_CONSUMPTION);
155 // printf("-----\n");
156 }
157
158
159 void processCommand (uint8_t cmd){ → Establece el estado del slave en función del
160
161 switch(cmd){
162 case 0x01: //All meas
163 currentState = ALL_MEAS;
164 break;
165
166 case 0xAA: //Sleep
167 currentState = LOW_POWER;
168 break;
169
170 case 0xBB: //Wake-Up
171 currentState = WAKE_UP;
172 break;
173 }
174 }
175
176 void readGPIO_Input(StatesPrincipal_t estado){ → Comprueba el estado del pin PB1 en
177
178 if(estado == WAKE_UP){ 0xBB
179 if(HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_1) == GPIO_PIN_SET){ //Pin del pulsador
180 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_SET);
181 →→ osDelay(10); } }
182 }
183
184
185 if(estado == LOW_POWER){ 0xAA
186 if(HAL_GPIO_ReadPin(GPIOB, GPIO_PIN_1) == GPIO_PIN_RESET){ → Aquí el slave se queda bloqueado cuando entra
187 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, GPIO_PIN_RESET);
188 Deinit_I2C_BME680();
189 DeInit_I2C();
190 NVIC_DisableIRQ(USART3_IRQn);
191 Enter_SleepMode(); → en modo de bajo consumo. Dentro de esta función
192 NVIC_EnableIRQ(USART3_IRQn);
193 GPIO_HCSR04();
194 CONSUMPTION_pin_F429ZI_config();
195 RTD_pin_F429ZI_config();
196 Init_LEDs();
197 Init_I2C();
198 Init_I2C_MASTER();
199 }
200 osDelay(10);
201 }
202 }
203
204 void Init_LEDs(void){
205 →
206 →GPIO_InitTypeDef GPIO_InitStruct;
207 →
208 →_HAL_RCC_GPIOB_CLK_ENABLE(); // Habilitar el reloj asociado al puerto de los LEDs. En este caso
el reloj del puerto B
209 →
210 →// Configuración de los LEDs VERDE|AZUL|ROJO = PIN 0|7|14
211 →GPIO_InitStruct.Pin = GPIO_PIN_0 | GPIO_PIN_7 | GPIO_PIN_14;
212 →GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
213 →GPIO_InitStruct.Pull = GPIO_PULLUP;
214 →GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
215 →
216 →HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
217 →
218 };
219
220

```

```

1 #include "com.h"
2
3 /* Colas de mensajes */
4 extern osMessageQueueId_t mid_Slave_Com_MsgQueue;
5 MSGQUEUE_OBJ_MEAS rx_medidas;
6
7 /* Functions prototypes */
8 void Th_Slave_Com (void *argument); // thread function
9 void Th_Com_Slave (void *argument); // thread function
10
11 /* Variables de medición */
12 uint16_t v_rtd; // Voltaje RTD
13 double hum; // Humedad
14 float luz; // Luminosidad
15 int aq; // Calidad del aire
16 float aqLvl; // Nivel de agua
17 uint16_t v_crnt; // Voltaje de consumo
18
19
20 static bool uart_initialized = false;
21 uint8_t commandrx[3]; // Comando que se recibe desde la Placa A(Master)
22 extern uint8_t commandrx[3];
23
24 osThreadId_t TID_UartRx;
25 osThreadId_t TID_UartTx;
26
27 uint8_t DATATx[14]; // Datos que enviamos al master/Placa A
28 bool trama_incorrecta = false;
29
30 int Init_ThCom (void) {
31
32     /* Hilo encargado de enviar info al PC */
33     TID_UartTx = osThreadNew(UartTx, NULL, NULL); // Hilo de transmisión
34     if (TID_UartTx == NULL) {
35         return (-1);
36     }
37
38     /* Hilo encargado de recibir info del PC */
39     TID_UartRx = osThreadNew(UartRx, NULL, NULL); // Hilo de recepción
40     if (TID_UartRx == NULL) {
41         return (-1);
42     }
43
44     UART_Init(); // Inicializa el periférico UART
45
46     return (0);
47 }
48
49
50
51 void USART_Callback(uint32_t event) {
52     if (event & ARM_USART_EVENT_RECEIVE_COMPLETE) {
53         osThreadFlagsSet(TID_UartRx, ARM_USART_EVENT_RECEIVE_COMPLETE);
54         // printf("[UART CALLBACK] Comando recibido\n");
55     }
56
57     if (event & ARM_USART_EVENT_SEND_COMPLETE) {
58         osThreadFlagsSet(TID_UartTx, ARM_USART_EVENT_SEND_COMPLETE);
59         // printf("[UART CALLBACK] Respuesta enviada al master\n\n");
60     }
61 }
62
63
64 void UART_Init(void) {
65     // 1. Inicialización
66     int32_t status;
67
68     status = UARTdrv->Initialize(USART_Callback);
69     if (status != ARM_DRIVER_OK) {
70         // printf("[UART] Error en Initialize: %d\n", status);
71         return;
72     }
73     // 2. Encender el periférico
74     status = UARTdrv->PowerControl(ARM_POWER_FULL);
75     if (status != ARM_DRIVER_OK) {
76         // printf("[UART] Error en PowerControl: %d\n", status);
77         return;
78     }
79 }
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
637
638
639
639
640
641
642
643
644
645
645
646
647
647
648
649
649
650
651
652
653
653
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
```

```

78         }
79     →→→
80
81     // 3. Configurar parámetros UART
82     status = UARTdrv->Control(
83         ARM_USART_MODE ASYNCHRONOUS | // Modo UART (asíncrono)
84         ARM_USART_DATA_BITS_8 | // 8 bits de datos
85         ARM_USART_PARITY_NONE | // Sin paridad
86         ARM_USART_STOP_BITS_1 | // 1 bit de parada
87         ARM_USART_FLOW_CONTROL_NONE, // Sin control de flujo
88         9600 // Baudrate
89     );
90     →→→ 9600 baudios (bps) →→→ Hay bit de stop
91     →→→ if(status == ARM_DRIVER_OK) {
92     //         printf("[UART] Inicializado correctamente a 9600 baudios\n");
93     uart_initialized = true;
94     } else {
95     //         printf("[UART] Error en Control: %d\n", status);
96     }
97     →→→
98     →→→ // 4. Habilitar TX y RX
99     status = UARTdrv->Control(ARM_USART_CONTROL_TX, 1);
100    if (status != ARM_DRIVER_OK) {
101    //         printf("[UART] Error al habilitar TX: %d\n", status);
102    return;
103    }
104    →→→ // Habilitar interrupción por recepción usando el callback (no IRQ directa)
105    status = UARTdrv->Control(ARM_USART_CONTROL_RX, 1);
106    if (status != ARM_DRIVER_OK) {
107    //         printf("[UART] Error al habilitar RX: %d\n", status);
108    return;
109    }
110 }
111
112 void UART_Deinit(void) {
113
114     int32_t status;
115     →status = UARTdrv->Uninitialize();
116 }
117
118 int32_t UART_ListenCommand(uint8_t* command_buffer) {
119     if (!uart_initialized) {
120     //         printf("[UART] Error: No inicializado antes de recibir comando\n");
121     return ARM_DRIVER_ERROR;
122     }
123     →printf("[UART] Esperando comando...\n");
124     →→→
125     int32_t result = UARTdrv->Receive(command_buffer, 3); →→→ El slave siempre recibe tres bytes del master
126     if (result != ARM_DRIVER_OK) {
127     //         printf("[UART] Error al iniciar recepcion del comando: %d\n", result);
128     return result;
129     }
130
131     →osThreadFlagsWait(ARM_USART_EVENT_RECEIVE_COMPLETE, osFlagsWaitAny, osWaitForever);
132
133
134     return ARM_DRIVER_OK;
135 }
136
137 int32_t UART_SendData(uint8_t* datatx) {
138     if (!uart_initialized) {
139     //         printf("[UART] Error: No inicializado antes de enviar temperatura\n");
140     return ARM_DRIVER_ERROR;
141     }
142
143     →int32_t result = UARTdrv->Send(datatx, 14); →→→ El slave siempre envia 14 bytes al master
144     if (result != ARM_DRIVER_OK) {
145     //         printf("[UART] Error al enviar temperatura: %d\n", result);
146     return result;
147     }
148
149     →osThreadFlagsWait(ARM_USART_EVENT_SEND_COMPLETE, osFlagsWaitAny, osWaitForever);
150
151
152     return ARM_DRIVER_OK;
153 }
154

```

Significa que hay bit de START

Significa que por byte transmitido (8 bits) hay un bit de start y un bit de stop.

10 bits totales
(por cada byte transmitido)

El slave siempre recibe tres bytes del master

El slave siempre envia 14 bytes al master

```

155 /*
156 */
157 ----- Thread 'UARTRx': UARTRx handler for reception -----
158 */
159 */
160 void UartRx (void *arg) {
161     while (1) {
162         // 1. Esperar comando: EL MASTER SOLO ME ENVÍA TRES BYTES
163         if (UART_ListenCommand(commandrx) == ARM_DRIVER_OK) {
164             if (commandrx[0] == COMIENZO_TRAAMA) {
165                 if (commandrx[2] == FINAL_TRAAMA) {
166                     // 2. Notificar a Tx SOLO cuando el comando esté listo
167                     osThreadFlagsSet(TID_UartTx, 0x01);
168                     // 3. Esperar confirmación de que Tx ha procesado el comando
169                     osThreadFlagsWait(0x02, osFlagsWaitAny, osWaitForever);
170                 }
171             } else {
172                 trama_incorrecta = true;
173                 osThreadFlagsSet(TID_UartTx, 0x01);
174                 osThreadFlagsWait(0x02, osFlagsWaitAny, osWaitForever);
175             }
176         } else {
177             printf("Error al recibir comando\n");
178         }
179     }
180 }
181 */
182 ----- Thread 'UARTTx': UARTtx handler for transmission -----
183 */
184 */
185 void UartTx (void *arg) {
186     while (1) {
187         // Espera nuevos datos de medición
188         osMessageQueueGet(mid_Slave_Com_MsgQueue, &rx_medidas, NULL, 0U);
189
190         // Almacena las mediciones
191         v_rtd = rx_medidas.Voltaje_RTD;
192         hum = rx_medidas.humidity;
193         luz = rx_medidas.luminosity;
194         aq = rx_medidas.air_quality;
195         aqLvl = rx_medidas.aquaLevel;
196         v_crnt = rx_medidas.Voltaje_CONSUMPTION;
197
198         // 1. Esperar notificación de comando recibido
199         osThreadFlagsWait(0x01, osFlagsWaitAny, osWaitForever);
200         // 2. Borrado de buffer de transmisión a enviar al MASTER
201         memset(DATATx, 0x00, sizeof(DATATx));
202
203         // Poner byte de inicio de trama (por ejemplo 0x7E)
204         if (!trama_incorrecta) // De primeras es false (Pero puede darse el caso)
205             DATATx[0] = COMIENZO_TRAAMA;
206         else
207             DATATx[0] = ERROR_TRAAMA;
208
209         // 3. Procesar comando (ya está almacenado en commandrx)
210         switch (commandrx[1]) {
211             case 0x01: // Envío de todos los datos
212                 // Temperatura
213                 float rtdv_aux = ((float)v_rtd*3.3f)/4096.0f;
214                 float rtd_V = (rtdv_aux*15.1745)-4.48f;
215                 rtd_V = rtd_V * 100;
216                 uint16_t rtd_scaled = (uint16_t)rtd_V;
217                 DATATx[1] = (uint8_t)(rtd_scaled >> 8) & 0xFF; // Byte más significativo (MSB)
218                 DATATx[2] = (uint8_t)(rtd_scaled & 0xFF); // Byte menos significativo (LSB)
219
220                 // Consumo
221                 float consumption_V = (((float)v_crnt*3.3f)/4096));
222                 consumption_V = consumption_V * 100;
223
224         }
225     }
226 }
227

```

COMIENZO_TRAAMA: 0x7E
FINAL_TRAAMA: 0x7F

En el hilo de recepción estoy escuchando constantemente esperando a que me llegue una trama con datos.

Si el primer byte coincide con el del inicio de trama, envío el del final de trama, lo indicando al hilo TX que prepare la respuesta para enviar al MASTER. Una vez hecho esto, el hilo RX se queda esperando a que el TX le diga que ha enviado

Para que no se quede bloqueado el hilo en caso de que no lleguen medidas

Hacemos el get del slave.c que nos envía todas las medidas juntas

Aquí ya empieza a formar la trama

Dependiendo del comando que llegue, formará una trama u otra.

```

224     uint16_t consumption_scaled = (uint16_t)consumption_V;
225     DATATx[3] = (uint8_t)(consumption_scaled >> 8) & 0xFF; // Byte más significativo (MSB)
226     DATATx[4] = (uint8_t)(consumption_scaled & 0xFF); // Byte menos significativo (LSB)
227
228     // Lux
229     uint32_t luz_100x = (uint32_t)(luz * 100.0f);
230     DATATx[5] = (uint8_t)((luz_100x >> 16) & 0xFF); // Byte más significativo (MSB)
231     DATATx[6] = (uint8_t)((luz_100x >> 8) & 0xFF); // Byte menos significativo (LSB)
232     DATATx[7] = (uint8_t)(luz_100x & 0xFF); // Byte menos significativo (LSB)
233
234     // Calidad del Aire
235     DATATx[8] = (uint8_t)aq; // Byte más significativo (MSB)
236
237     // Humedad
238     uint16_t humscaled = (uint16_t)(hum * 100); // Byte más significativo (MSB)
239     DATATx[9] = (uint8_t)(humscaled >> 8); // Byte menos significativo (LSB)
240     DATATx[10] = (uint8_t)(humscaled & 0xFF); // Byte menos significativo (LSB)
241
242     // Nivel de agua
243     uint16_t level_scaled = (uint16_t)(aqLvl * 100); // Byte más significativo (MSB)
244     DATATx[11] = (uint8_t)(level_scaled >> 8); // Byte menos significativo (LSB)
245     DATATx[12] = (uint8_t)(level_scaled & 0xFF); // Byte menos significativo (LSB)
246
247     break;
248 }
249
250 case 0xBB: { //COMANDO SLEEP
251     DATATx[12] = 0xBB;
252     break;
253 }
254
255 case 0xAA: { //WAKE UP
256     DATATx[12] = 0xAA;
257     break;
258 }
259
260 default:{
261     trama_incorrecta = false;
262     printf("[UART TX] Comando desconocido: 0x%02X\n", commandrx);
263     break;
264 }
265
266 // 4. Enviar datos
267
268 // Byte de fin de trama (por ejemplo 0x7F)
269 DATATx[13] = FINAL_TRAAMA;
270
271 if (UART_SendData(DATATx) != ARM_DRIVER_OK) {
272     printf("[UART TX] Error al enviar respuesta\n");
273 }
274
275 HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_7); //Indica envío de información por la UART (Julián)
276
277 // 5. Confirmar a Rx que hemos terminado
278 osThreadFlagsSet(TID_UartRx, 0x02);
279 }
280

```

do que se hace en cada parámetro es multiplicarlo por 100 para no tener decimales y después desplazarnos para poder almacenar el valor en dos bytes de datos

- Si **commandrx[1] = 0x01**. La trama que forma para enviar al máster es: [0x7E][D1]...[D12][0x7F], que son 14 bytes

- Si **commandrx[1] = 0xAA**. La trama que forma para enviar al máster es: [0x7E][0x00]...[0x00][0xAA][0x7F], que son 14 bytes para confirmar que se va a dormir al Máster.

- Si **commandrx[1] = 0xBB**. La trama que forma para enviar al máster es: [0x7E][0x00]...[0x00][0xBB][0x7F], que son 14 bytes para confirmar que se ha despertado al Máster.

APUNTES EXAMEN PROYECTO ISE

1. ¿Cuántos bits por segundo se trasmite en tu comunicación?

Basándonos en nuestra comunicación, que se realiza mediante USART, tanto en el modo PRESENCIAL como en el modo WEB, el master solicita medidas al slave cada segundo. Según nuestra configuración de USART tenemos: modo síncrono, bits de datos(8), no paridad, bits de parada(1), no control de flujo y 9600 baudios. Esto quiere decir que por byte (8 bits) se transmiten: un bit de inicio (start), 8 bits de datos, 0 de paridad y 1 bit de parada. En total 10 bits por byte transmitido.

El Master siempre va a enviar 3 bytes:

0x7E	CMD	0x7F
------	-----	------

 $\rightarrow \text{CMD} = 0x01 = 0xAA = 0xBB$

El Slave siempre va a enviar 14 bytes:

0x7E	D1	D2	D3	D4	D5	D6	D7	D8	D9	D10	D11	D12	0x7F
0x7E	0x00	0xAA	0x7F										
0x7E	0x00	0xBB											

NOTA IMPORTANTE: dentro de cada uno de estos bytes hay que sumar dos bits más, el de start y stop, por lo que vamos a suponer a partir de ahora que **1 byte = 10 bits**

En PulseView, sea cual sea la trama, se observa que el master envía 3 bytes y el slave 14 bytes. Es decir 17 bytes totales en un periodo de 17,68 ms, solicitan medidas cada segundo.

Cálculo total de bits transmitidos en 17,68 ms:

1. Hemos considerado: 1 byte = 10 bits \longrightarrow 17 bytes \cdot 10 bits = 170 bits
2. Tiempo total de transmisión: 17,68 ms = 0,01768 s
3. Tasa de bits(instantánea) = 170 bits / 0,01768 s = 9615,4 bps

Por lo tanto, la velocidad media por segundo es de 170 bits por segundo a una velocidad instantánea de aproximadamente 9600 baudios.

2. Autonomía de nuestro sistema

- Alimentamos con tres baterías en serie → cargadas al máximo → $3 \cdot 4,1 = 12,3 \text{ V máx}$
- Capacidad: 3500 mAh
- Regulador: 7809 linear (salida 9 V, dropout $\approx 2 \text{ V}$)

NOTA: tensión mínima que garantiza el dropout → $U_{in(\min)} = 8,9 \text{ V} + 2 \text{ V} = 10,9 \text{ V}$

1. Modo standby: consumo de 159 mA

- La energía total disponible será de: $E = 12,3 \text{ V} \cdot 3,5 \text{ Ah} = 43,05 \text{ Wh}$
- Potencia consumida: $P = 12,3 \text{ V} \cdot 0,159 \text{ A} = 1,956 \text{ W}$
- Duración estimada: $t = 43,05 \text{ Wh} / 1,956 \text{ W} \approx 22 \text{ horas}$

Considerando que al menos el 7809 necesita al menos 10,9 V para funcionar, solo tendremos un uso de batería de $10,9 \text{ V} / 12,3 \text{ V} \approx 89\%$ de su vida útil

$$\text{Autonomía restante: } 22 \text{ h} \cdot 0,89 = 19,6 \text{ horas}$$

2. Modo normal (PRESENCIAL/WEB) → consumo de 190,6 mA

- La energía total disponible será de: $E = 12,3 \text{ V} \cdot 3,5 \text{ Ah} = 43,05 \text{ Wh}$
- Potencia consumida: $P = 12,3 \text{ V} \cdot 0,1906 = 2,345 \text{ W}$
- Duración estimada: $t = 43,05 \text{ Wh} / 2,345 \text{ W} \approx 18,36 \text{ horas}$
- Autonomía restante: $18,36 \cdot 0,89 = 16,3 \text{ horas}$

3. Modo standby (1h dormido + 1 Medición rápida):

Suponiendo que cada hora, durante 6s va a estar activo (consumiendo 190,6 mA) y el resto del tiempo está dormido (159 mA):

$$\text{- Consumo promedio por hora: } I_{\text{medio}} = \frac{(6s \cdot 190,6 \text{ mA}) + (3594 \cdot 159 \text{ mA})}{3600s} = 159,05 \text{ mA}$$

Es prácticamente el modo dormido, por lo que sigue siendo de 19,6 horas

3. Preguntas sobre la EEPROM

- ¿Cuál es el propósito del array write-buffer[] y por qué se escriben 2 bytes de más al principio?

Este almacena los datos que se van a escribir en la EEPROM. Los primeros 2 bytes indican la posición interna en la que voy a escribir, y el resto son datos reales a guardar.

- ¿Qué función cumplen los índices de lectura y escritura?

Cada índice controla la posición de memoria para almacenar la primera medida del tipo correspondiente.

Cuando llega a las 10 se reinicia a 0, pero el booleano "full" ya se activa indicando que se ha llegado al tope de medidas a almacenar. A partir de este punto vamos a estar sobreescritiendo los datos en cada posición.

- ¿Cómo se calcula la dirección de memoria para escribir una nueva medida?

Se usa la fórmula: $addr = base_addr + index \cdot data.length$. Donde base-addr es una posición fija y multiplicar por data.length asegura que las escrituras NO se solapan en una misma página. El index varía cada vez que escribimos.

- ¿Qué ventaja ofrece almacenar las medidas por tipo de página separadas dentro de la EEPROM?

Facilita la organización, evita solapamientos y hace más eficiente la lectura secuencial de un tipo concreto

- ¿Cuál es la diferencia entre writememory() y guardarMedida()?

writememory → es la instrucción que realiza la escritura con el sensor I2C

guardar-medida → función de alto nivel que prepara la dirección de escritura en memoria, gestiona los datos a guardar, actualiza el índice y llama a writememory

- ¿Por qué se utiliza el osDelay(5) después de cada escritura en la EEPROM?

Es para darle un pequeño tiempo a la EEPROM para completar la escritura interna en su memoria no volátil. Esto garantiza que no se inicie otra operación antes de terminar.

- ¿Qué sucede si se reinicia la placa sin haber guardado los índices? ¿Se pierde?

Se pierde el índice de escritura y lectura actual. Se empieza desde el índice cero y se sobreescrivirán las medidas anteriormente guardadas.

- ¿Cómo podríamos modificar el sistema para guardar más de 10 medidas?

Haría que aumentar max-entries, el valor que queremos teniendo en cuenta que el máximo son 64 bytes y cada página es de 64 bytes.

- ¿Qué pasaría si read_index no se reiniciara al llegar a 10 y sigue aumentando?

Se accederían a posiciones inválidas fuera del rango de la página (posiciones donde no ha sido escrito y hay datos que no me interesan).

- ¿Qué modificación harías si la EEPROM fuese solo de 4 kB en lugar de 32 kB?

Reducir la cantidad de medidas por tipo, compactar el formato o combinar medidas en una misma página.

→ con 32 kB → podríamos tener unas 500 páginas de 64 bytes

→ con 4 kB → podríamos tener aprox 62 páginas de 64 bytes

- ¿Qué ventaja tiene usar DMA o IRQ para manejar las transferencias I2C en lugar de polling con bucle while?

Menor uso de CPU y eficiencia, ya que no se bloquea el hilo esperando. El sistema puede seguir ejecutando las mismas tareas mientras espera el evento.

4. Preguntas sobre la UART

- ¿Qué garantiza que la USART no terminado de enviar o recibir antes de seguir?

Usando osThreadFlagsWait(...) esperando a los flags SEND / RECEIVE que llegan de la callback. (NO BLOQUEANTE)

- ¿Cuánto tarda en enviarse una trama USART completa del master al slave y viceversa?

Baudrate = 9600 baudios

Realmente: 1 byte = 10 bits (start + 8 bytes + stop)

Máster: envía 3 bytes al slave // Slave: envía 14 bytes al máster

Tiempo de transmisión:

$$\text{Máster} \rightarrow \text{Slave}: T = \frac{30 \text{ bits}}{9600 \text{ bits/s}} = 3,125 \text{ ms}$$

$$\text{Slave} \rightarrow \text{Máster}: T = \frac{140 \text{ bits}}{9600 \text{ bits/s}} = 14,58 \text{ ms}$$

- ¿Qué ocurre si aumentamos el baudrate en el Máster y en el Slave?

1. Habrá más rapidez en el envío de datos, enviándose y recibiéndose más rápido
2. Puede aumentar el consumo al trabajar más rápido la UART.

- ¿Qué ocurre si el master y el slave tienen un bufer distinto?

Master y slave no usan la misma velocidad, por lo que se recibirán bytes corruptos o mal sincronizados.

- ¿Por qué se usan bytes de comienzo y final de trama?

Se usan para establecer un protocolo. De esta forma habrá sincronización, evitando errores de trama.

- ¿Qué función tiene memoria en el master?

Es la estructura global que contiene todos los medios ambientales [temperatura, humedad, luminosidad, consumo, calidad aire y agua].

- ¿Porque la temperatura y la humedad se codifican en 2 bytes en vez de 4 bytes con Plant?

Para reducir el tamaño de la trama y facilitar el parsing. Codificar Plant requiere 4 bytes y más operaciones. En 2 bytes en formato entero se preserva precisión sin ampliación.

- ¿Porque se usan baudios? ¿es arbitrario o tiene razones técnicas?

No es arbitrario, los baudios son fiables a largos distancias o cables ruidosos y además compatibles con casi todos los dispositivos UART por defecto.

- ¿Porque no se utiliza control de flujo [ARM - USART - Flan - CANTAN - NONE] en esta comunicación?

Porque la comunicación es simple, controlada por software. No se requiere gestión de buffers por hardware ni hay riesgo de sobre cargo de datos.

- Preguntas fisiológicas:

- ¿Qué pasaría si nos cambiamos la tensión de referencia en el puente de Wheatstone?
- ¿Qué pasaría si en vez de usar una Pt-100 usámos una Pt-1000?
- ¿Qué ocurriría si una de las resistencias del puente de Wheatstone cambiase su valor?

$$V_d = V_1 - V_2 = V_{REF} \cdot \left(\frac{RTD}{10R + RTD} - \frac{R}{10R + R} \right) \rightarrow \begin{cases} V_d|_{T=-5^\circ C} = 2,02 V \\ V_d|_{T=45^\circ C} = 2,24475 V \end{cases}$$

Ahora la ganancia del AI:

$$V_o = A_{AI} \cdot V_d = \left(1 + \frac{2 \cdot R_3}{R_G} \right) \cdot V_d \quad \begin{cases} V_o = 0 V \rightarrow V_d = 2,02 V \\ V_o = 3,3 V \rightarrow V_d = 2,24475 V \end{cases}$$

$$A_{d1} = \frac{\Delta V_o}{\Delta V_d} = \frac{3,3}{2,24475 - 2,02} = 14,68 \text{ V/V} \rightarrow R_Q = \frac{2 \cdot 10k}{13,68} = 1461,98 \Omega > R_Q = 1k$$

Con esta ganancia, en el mejor de los casos para $V_d(T = -5^\circ C) = 2,02 V \cdot 14,68 = 29,65 V$.

Esto es imposible ya que los OP07 tienen alimentación bipolar de $\pm 9 V$ y se saturaría.

Además, el valor del potenciómetro R_Q es mayor al empleado, por lo que tendría que cambiarse.