Trabajo de Fin de Grado
Ingeniería en Electrónica, Robótica y Mecatrónica
Mención en Instrumentación Electrónica y Control

Diseño y desarrollo de un driver de control para la estación SPS30 y su integración en FreeRTOS

Autor: Pedro Barba Lozano

Tutores: Ramón González Carvajal, Eduardo Hidalgo Fort

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo de Fin de Grado
Ingeniería en Electrónica, Robótica y Mecatrónica Mención en
Instrumentación Electrónica y Control

Diseño y desarrollo de un driver de control para la estación SPS30 y su integración en FreeRTOS

Autor:

Pedro Barba Lozano

Tutores:

Ramón González Carvajal, Eduardo Hidalgo Fort

Catedrático de Universidad, Investigador PostDoctoral

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Trabajo de Fin de Grado: Diseño y desarrollo de un driver de control para la estación SPS30 y su integración en FreeRTOS

Autor: Pedro Barba Lozano
Tutores: Ramón González Carvajal, Eduardo Hidalgo Fort

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

.

Índice

Índice de figuras

Índice de códigos

Índice de cuadros

1. Introducción

1.1. Motivación

De acuerdo con la EPA (Agencia de Protección Ambiental de Estados Unidos), el tamaño de las partículas está directamente vinculado a su potencial para generar problemas de salud, [1]. Los dos grupos en los que se suele hacer distinción son:

- Partículas con diámetro menor a 10 µm (PM10), que pueden alcanzar los pulmones en la respiración normal.
- Partículas con diámetro menor a 2.5 µm (PM2.5), que además pueden llegar a penetrar en la sangre.

Estas partículas son sensiblemente más pequeñas que el cabello humano, de ahí su facilidad para adentrarse en nuestro sistema respiratorio. En la Figura 1 se muestra una comparación de los diámetros de dichas partículas con el cabello.

Algunos de los efectos que puede ocasionar la larga exposición a partículas de este tipo en el aire son:

- Muerte prematura en personas con enfermedades pulmonares o del corazón.
- Ataques al corazón.
- Alteración del ritmo cardíaco.
- Agravación del asma.
- Empeoramiento de la capacidad pulmonar.
- Incremento de síntomas respiratorios, como la irritación de las vías respiratorias, tos o dificultad para respirar.

Por otra parte, las partículas de diámetro igual o menor a 2.5 µm son la principal causa de reducción de la visibilidad, generando neblina. Además, dada su facilidad para ser arrastradas por el viento largas distancias, contribuyen a los siguientes fenómenos:

- Hacer ácidos los lagos y ríos.
- Alterar el balance de los nutrientes en embales de agua.
- Disminuir los nutrientes en el sustrato.
- Dañar los bosques y cultivos: deterioro del follaje, reducción de la tasa de crecimiento o muerte prematura de la planta.
- Facilitar la generación de lluvia ácida.

Los causantes suelen ser fuentes localizadas, como áreas en construcción, caminos sin pavimentar, chimeneas o fuegos. También hay que considerar los químicos volátiles generados en distintas reacciones, que podemos encontrar, por ejemplo, en centrales de energía, la industria y los automóviles.

Según el rango de diámetro en el que busquemos, podemos encontrar distintos tipos de elementos, [3]:

- Entre 0.1 y 1 µm: sulfatos, nitratos, amoníaco, carbón, plomo y otros residuos orgánicos.
- Entre 2.5 y 10 µm: tierra, arena, sal marina, bioaerosoles [4] (virus, bacterias, esporas protozoos, polen, ácaros del polvo, etcétera).

Es importante mantener un control de estos parámetros para asegurar que no se perjudica el ambiente en el que se trabaja, además de asegurar la salud de las personas que estén trabajando en contacto directo con el foco emisor o que habiten en sus cercanías.

En este trabajo vamos a elaborar una librería para controlar un sensor capaz de medir el tamaño y número de partículas suspendidas en el aire, con el objetivo de hacer un seguimiento de la calidad de éste. La idea es integrar dicho sensor en un entorno completo orientado a realizar medidas medioambientales. También se implementará dicho controlador en un sistema operativo en tiempo real, dada la necesidad de coordinar las distintas tareas que formarán el proyecto global.

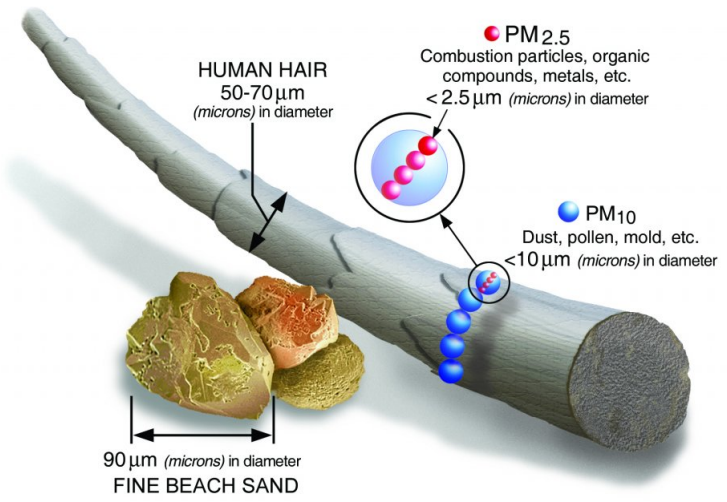

Figura 1: Comparación de las partículas PM10 y PM2.5 con un cabello humano donde se puede apreciar la facilidad de éstas para atravesar las fosas nasales y llegar a los pulmones. Imagen tomada de [2].

1.2. Elección de un sensor

Se ha realizado una búsqueda de distintos sensores en proveedores habituales como *RS Components* o *Mouser Electronics*. Vamos a realizar una breve comparación entre varios de ellos:

Nombre	Tera Sensor	Monnnit MNS2-9-W2-AQ-P25A	HonneyWell HPMA215S0	Sensirion SPS30	Panasonic SN-GCJA5
Consumo en hibernación	20 mA	0.2 µA	20 mA	50 µA	No disponible
Consumo estándar	80 mA	2.5 mA	80 mA	65 mA	Menor a 100 mA
Medidas	PM1, PM2.5, PM10, Temperatura, Humedad relativa	PM1, PM2.5, PM10	PM1.0, PM2.5, PM4.0, PM10	PM0.5, PM1.0, PM2.5, PM4.0, PM10	PM1, PM2.5, PM10
Tiempo de refresco	1s, 10s, 60s	10s	6s	1s	1s
Precio aproximado	100€	121€	30€	45€	21€

Cuadro 1: Comparación entre modelos de sensores de medida del las partículas en el aire.

Teniendo en cuenta el contenido de la Figura 1, el sensor elegido es el SPS30, debido a que por un precio intermedio, ofrece un consumo bajo, lo cual nos interesa especialmente como se desarrollará en la Subsección 1.3.2, un abanico amplio medidas distintas y un tiempo de refresco de las medidas bajo. Aunque el sensor de Monnit tiene unas mayores prestaciones, su precio es el triple y en esta aplicación hemos buscado que el sistema sea económico.

1.3. Objetivos

1.3.1. Diseño y control

En este trabajo nos comunicaremos con el SPS30. Como se ha descrito en la Subsección 1.1, se trata de un sensor que mide el tamaño de las partículas suspendidas en el aire con el objetivo de realizar una monitorización de la calidad de éste. El principio de medida de este tipo de sensor se basa en la dispersión del láser al atravesar las partículas y en el análisis estadístico de esta desviación. El conjunto medible de

nuestro sensor se muestra en el Cuadro 2.

La parte fundamental del proyecto será la creación de una librería de comandos para el sensor, cubriendo todas las opciones que éste ofrece para su posterior implementación en un entorno real con especificaciones concretas.

Medida	Unidades
Concentración en Masa PM1.0	[µg/m^3]
Concentración en Masa PM2.5	[µg/m^3]
Concentración en Masa PM4.0	[µg/m^3]
Concentración en Masa PM10	[µg/m^3]
Concentración Numérica PM0.5	[#/cm^3]
Concentración Numérica PM1.0	[#/cm^3]
Concentración Numérica PM2.5	[#/cm^3]
Concentración Numérica PM4.0	[#/cm^3]
Concentración Numérica PM10	[#/cm^3]
Tamaño Típico de Partícula	[µm] o [nm]

Cuadro 2: Medidas realizadas por el sensor SPS30.

1.3.2. Diseño optimizado para el bajo consumo

Uno de los criterios de elección del sensor en la Subsección 1.2 ha sido el bajo consumo que ofrece, especialmente al permanecer en modo de hibernación, debido a que nuestro sistema no va a ir conectado a la red eléctrica, sino que dependerá de una batería para permitir su portabilidad y la capacidad de instalarlo en cualquier lugar. Centrándonos en el propio sensor y en su librería, posee tres estados distintos, cada uno con un consumo propio. El detalle de la máquina de estados que conforma el sensor se desarrolla en su hoja de datos, [6], y son los siguientes:

- Reposo, consumo de entre 45 y 65 mA.
- Medida, consumo de unos 330 µA.
- Hibernación, consumo menor a 50 µA.

En la implementación que se hará en un futuro se empleará una batería con una capacidad de 13 A h. Suponiendo un caso ideal donde solo se usa la batería para alimentar el sensor, podríamos dejarlo en modo de hibernación durante 30 años. Si se realizara una medida cada minuto, considerando que se tarda un segundo en realizar una medida, podríamos usarlo durante 478 días.

Evidentemente, no es factible usar una batería únicamente para alimentar un sensor, sin embargo, nos sirve para hacernos una idea del bajo consumo del mismo. En el caso real, esta batería se compartiría con otros sensores o con el microcontrolador, además de que se añadiría, por ejemplo, una pequeña placa solar para que el sistema se autoabastezca.

1.3.3. Integración en posibles proyectos complejos

Este no es un trabajo aislado, sino que busca formar parte de un proyecto mayor donde se integran un grupo de sensores orientados a las mediciones meteorológicas. Esto implica que las librerías que se han creado serán posteriormente utilizadas y sincronizadas con otras tareas. Es por esto último que necesitamos realizar una implementación en tiempo real, para así, con un mismo dispositivo controlar varios sensores sin que haya incompatibilidades entre ellos, como se verá en la Subsección 1.3.4.

Formar parte de un proyecto mayor implica también que partimos de cierto trabajo previo. Como se desarrollará en las Subsecciones 2.2.1 y 2.2.3, contamos con ciertas librerías que implementan la interfaz de comunicación necesaria.

Aunque el objetivo de este proyecto no ha sido el análisis de las medidas y su monitorización con vistas al control, la utilidad de esta librería será cimentar ese trabajo futuro a la hora de implementar una estación meteorológica sin importar las especificaciones necesarias.

1.3.4. Operativa en tiempo real

Un Sistema Operativo en Tiempo Real (en inglés, Real Time Operative System, RTOS) es aquel que está diseñado para aplicaciones que se ejecutan en tiempo real. Esto significa que el correcto funcionamiento del sistema depende, además de su resultado lógico, del tiempo en el que se realiza sus operaciones, refiriéndonos a tiempo real como una medida temporal en la misma escala que la del ambiente con el que se quiere trabajar.

El sistema operativo FreeRTOS se adecúa a estas aplicaciones en tiempo real para microcontroladores o sistemas embebidos, las cuales suelen incluir especificaciones ligeras y exigentes en tiempo real:

- Las especificaciones ligeras establecen un tiempo límite en el que ejecutar una tarea, pero que no es crítico para que el sistema funcione.
- Las especificaciones exigentes necesitan ser cumplimentadas antes de que se sobrepase el límite de tiempo, de lo contrario, el sistema ha fallado en su tarea y se deben tomar medidas alternativas. Un ejemplo puede ser el tiempo de apertura del airbag de un coche, si se detecta un choque y no se ha abierto en un intervalo de tiempo determinado de manera automática, se activarán otros mecanismos alternativos para dispararlo.

En el caso de FreeRTOS, es un planificador en tiempo real sobre el cual podemos construir nuestras aplicaciones en forma de tareas independientes. Estas tareas se ejecutarán en paralelo incluso si nuestro procesador solo tiene un núcleo, ya que el planificador decide en qué momento se ejecuta cada una. Cuenta con múltiples mecanismos para asignar una prioridad a cada tarea y sincronizarlas entre ellas, permitiendo abordar una gran cantidad de posibles escenarios a tratar. En la Subsección 2.2.4 se hablará de la integración dentro de este entorno de los distintos elementos que componen el driver del sensor.

Como se ha mencionado en la Subsección 1.3.3, en el proyecto global se busca integrar numerosos sensores, además de módulos de comunicación y otros periféricos. Viendo las ventajas que ofrece un sistema como FreeRTOS a la hora de manejar paralelamente varias aplicaciones de manera simultánea, resulta conveniente integrar nuestro trabajo en este sistema operativo. Si este fuera un trabajo aislado, no sería aconsejable implementar un sistema operativo en tiempo real, ya que necesita cierta capacidad de procesamiento que no aporta nada si el proyecto no es considerable.

1.4. Metodología

Las distintas fases que se han seguido para realizar este proyecto son las siguientes:

- Análisis de la bibliografía sobre FreeRTOS para comprender sus fundamentos y funciones básicas.
- Análisis de la hoja de datos del SPS30 para realizar el conexionado y programar la librería.
- Programación de los comandos propios de la librería.
- Testeo del funcionamiento de los comandos.
- Integración en el entorno de FreeRTOS.
- Testeo del funcionamiento de las tareas.

2. Arquitectura de referencia

2.1. Diagrama de bloques Hardware del proyecto

Los elementos Hardware usados para este proyecto son el sensor SPS30, una placa Nucleo-L152RE, un convertidor de lógica TTL a UART y un PC. Vamos a explicar el papel que desempeña cada uno, así como el conexionado entre ellos. En la Figura 2 se ve todo el sistema montado.

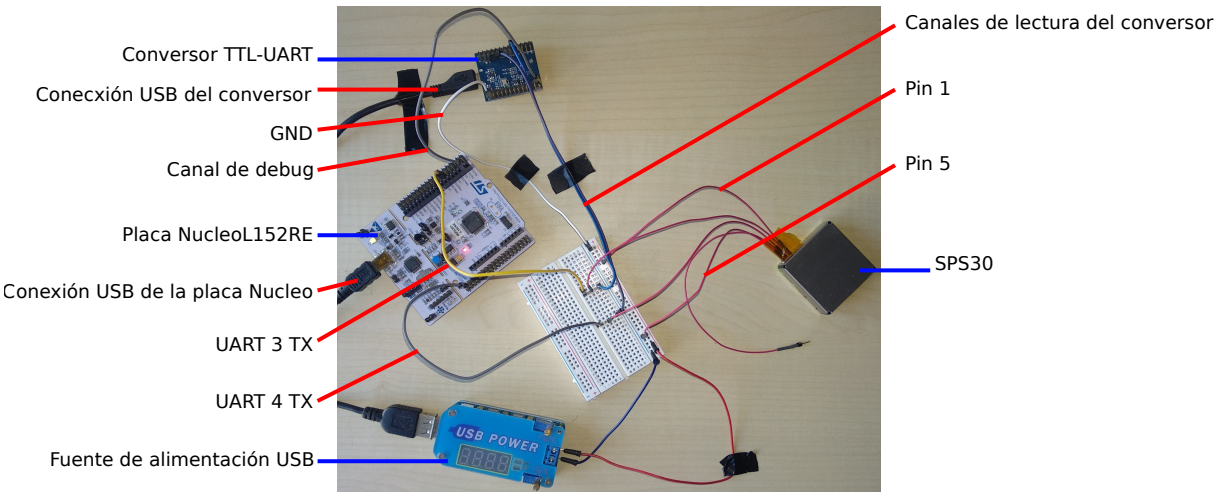

Figura 2: Montaje completo de los elementos Hardware.

2.1.1. Sensor, SPS30

El SPS30 puede utilizar dos interfaces, UART y SPI. Siguiendo la recomendación de la hoja de datos, elegimos usar la UART ya que nuestros cables de conexión miden menos de 20 cm, ofreciendo una mejor resistencia a las interferencias electromagnéticas.

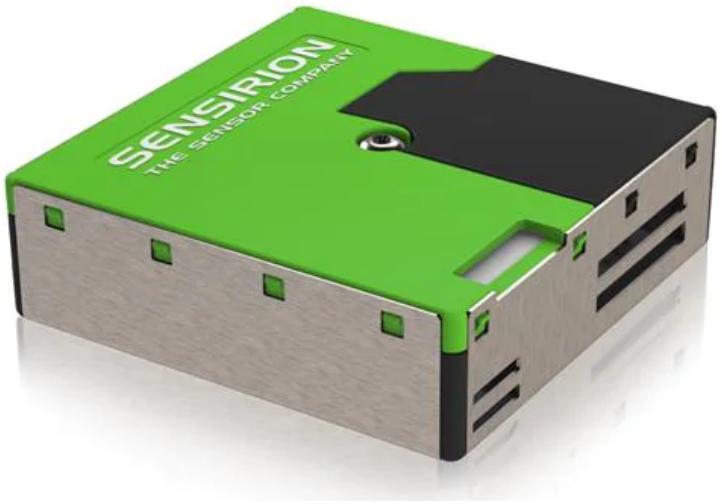

Figura 3: Imagen del sensor SPS30.

Es importante saber que la carcasa metálica del sensor está unida a la conexión de tierra, de modo que nos debemos asegurar de mantener esta carcasa aislada, ya que la circulación de corriente en la conexión entre la carcasa y la tierra del sensor puede provocar daños en éste.

En el Cuadro 3 se muestra el pinout del SPS30 en la configuración UART.

Protocolo SHDLC En un nivel superior a la interfaz de la UART, se encuentra el protocolo SHDLC (Control del Enlace de Datos de Alto Nivel de Sensirion, en inglés Sensirion High-Level Data Link Control). Es un protocolo de comunicación en serie basado en una arquitectura de master/slave, donde el SPS30 actúa siempre como slave.

Cada comunicación se transfiere en una unidad lógica llamada frame, la cual siempre comenzará desde el master en un frame MOSI (Salida del Master - Entrada al Slave, en inglés Master Output - Slave Input) y será respondido por el SPS30 en un frame MISO (Entrada al Master - Salida del Slave, en inglés Master

Pin	Función	Comentarios
1	Alimentación (VDD)	5 V±10 %
2 3	Recepción de datos (RX) Transmisión de datos (TX)	Compatible con 5 V-TTL y 3.3 V-LVTTL
4	Selección de interfaz	Dejar flotante para seleccionar la UART
5	Tierra (GND)	Conectada con la carcasa metálica

Cuadro 3: Pinout del sensor SPS30.

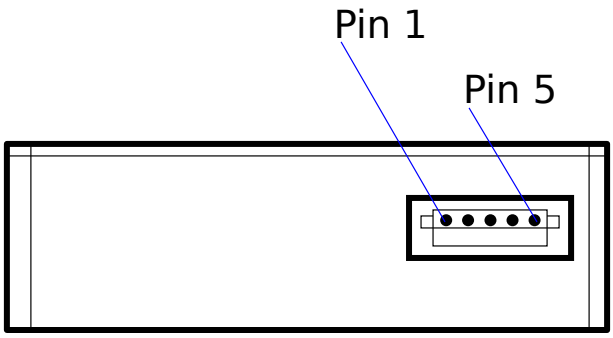

Figura 4: Diagrama con el pinout del SPS30.

Input - Slave Output).

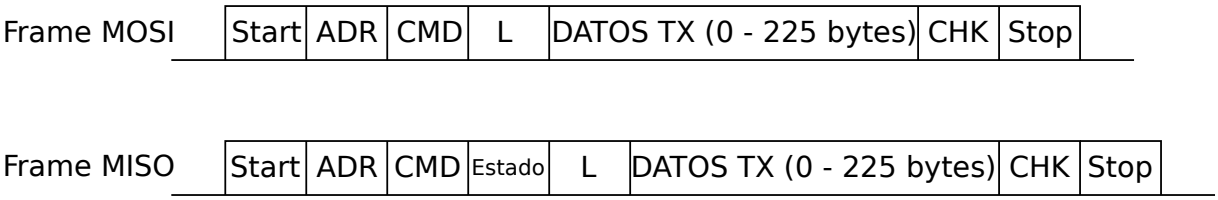

Figura 5: Esquema de los diagramas MISO y MOSI en el protocolo SHDLC.

A continuación, se enumeran los distintos bytes que conforman este protocolo:

- Bytes de Comienzo (Start) y Fin (Stop) del frame: toman en valor hexadecimal 0x7E, que queda reservado para estas señales
- Direción (ADR): la del dispositivo slave, que siempre será 0.
- Comando que se desea enviar (CMD).
- Longitud de los datos que se envían (L).
- Estado del SPS30 (State).
- Datos enviados por el master (TX Data) o por el SPS30 (Rx Data). Pueden ocupar hasta 225 bytes.
- Suma de comprobación, en inglés Checksum (CHK).

Byte-stuffing Un detalle importante sobre la comunicación que realiza el sensor es que hay ciertos valores que no pueden ser enviados entre los bits de Start y Stop, uno de ellos debe ser el propio valor reservado a éstos. Para poder interpretar dichos valores en caso de que necesiten ser enviados, se sustituirán por otros dos bytes. En el Cuadro 20 se muestran los bytes prohibidos y los bytes con los que se sustituyen.

2.1.2. Placa NUCLEO-L152RE

La placa NUCLEO-L152R de STM constituye nuestra herramienta principal para controlar el sensor. Es la que incorpora el microprocesador que vamos a programar, encargado de realizar todas las operaciones pertinentes. Como su nombre indica, el microcontrolador que incorpora esta placa es un STM32152RE. A continuación, se listan las características destacadas en la hoja de datos de dicho microcontrolador.

- Consumo ultra-bajo: alimentación entre 1.65 V y 3.6 V, numerosos modos de bajo consumo con corrientes mínimas del orden de nA, 8 µs de tiempo de reanudación.
- Core Arm Cortex-M3 de 32-bit: desde 32 kHz hasta 32 MHz, 1.25 DMIPS/MHz (Millones de Instrucciones Por Segundo de tipo Dhrystone), unidad de protección de memoria.
- Hasta 34 canales con sensor capacitivo.
- Unidad de cálculo del CRC (Verificación de redundancia cíclica, en inglés Cyclic redundancy check).
- Browout reset, Power-on Reset, Power-down reset, detector de tensión programable.
- Seis fuentes de reloj.
- Hasta 116 puertos de entrada-salida, que se pueden mapear en 16 vectores de interrupción.
- 512 kbytes de memoria Flash, 80 kbytes de RAM, 16 kbytes de EEPROM.
- Driver de LCD .
- 2 amplificadores operacionales, ADC y DAC de 12 bits, 2 comparadores.
- Interfaces de comunicación: 1 USB, 5 USART, 8 SPI, 2 I2C.

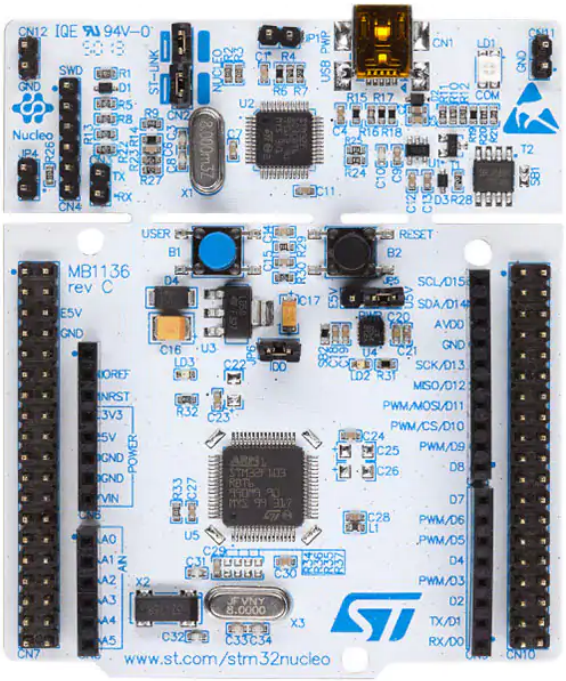

Figura 6: Imagen de la placa Nucleo-L152RE.

Dentro de la gran cantidad de funcionalidades que ofrece este microcontrolador, alrededor de la cual vamos a programar será la UART. La interfaz de Transmisión-Recepción Asíncrona Universal (en inglés Universal Asynchronous Receiver-Transmitter, UART) es la elegida para conectar nuestro sensor con la placa Nucleo. Como parte de un proyecto mayor, contamos con una librería de comandos ya dada la cual se encarga de realizar las configuraciones necesarias para trabajar con la UART. En la Sección 3.1.2

explicaremos que ha sido necesario hacer una serie de modificaciones a dicha librería.

Se ha decidido utilizar dos UART para realizar la comunicación con el sensor, una para enviar comandos y otra para recibir las respuestas. De esta manera podemos solucionar por separado los problemas que puedan surgir durante las transmisiones. En la Figura 7 se muestra el pinout de la UART dentro de la placa Nucleo. Este pinout viene ya definido por la librería dada.

Se muestran tanto el puerto RX como el TX de ambas UART por si en una aplicación futura se decide hacer uso exclusivo de una de ellas. En nuestro caso, tomaremos el puerto PB10, para enviar comandos al sensor y el PC8 para recibirlos.

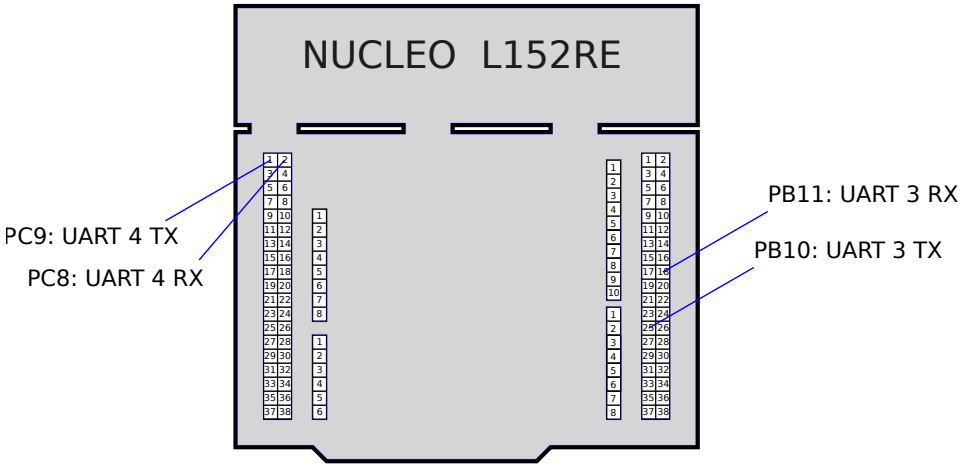

Figura 7: Diagrama con el pinout de la placa Nucleo.

2.1.3. Módulo de conversión TTL - UART, FT4232H-56Q

El módulo FT4232H-56Q nos permite leer y enviar datos a través de una UART y mostrar dicho contenido en el PC, utilizando un software como Docklight en este caso. Este dispositivo no forma parte del producto final, sino que es una herramienta orientada a la depuración.

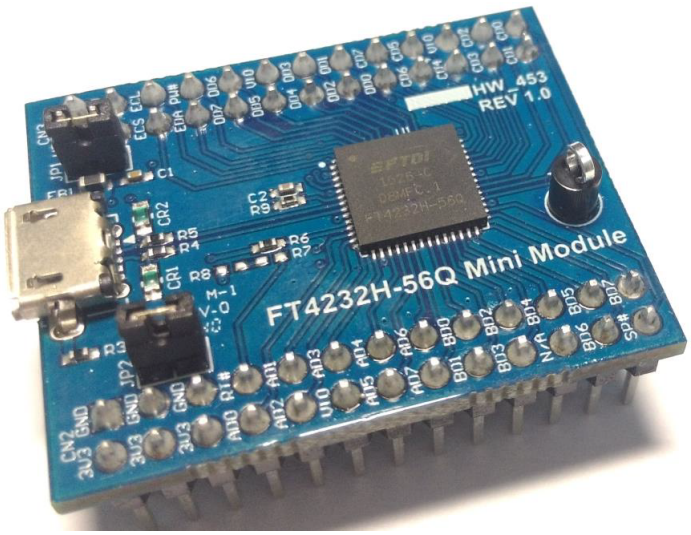

Figura 8: Imagen del módulo de conversión TTL-UART, FT4232H-56Q.

Vamos a utilizar tres puertos distintos de este módulo para monitorizar las conexiones especificadas en la Subsección 2.1.4. La conexión se muestra en la Figura 9.

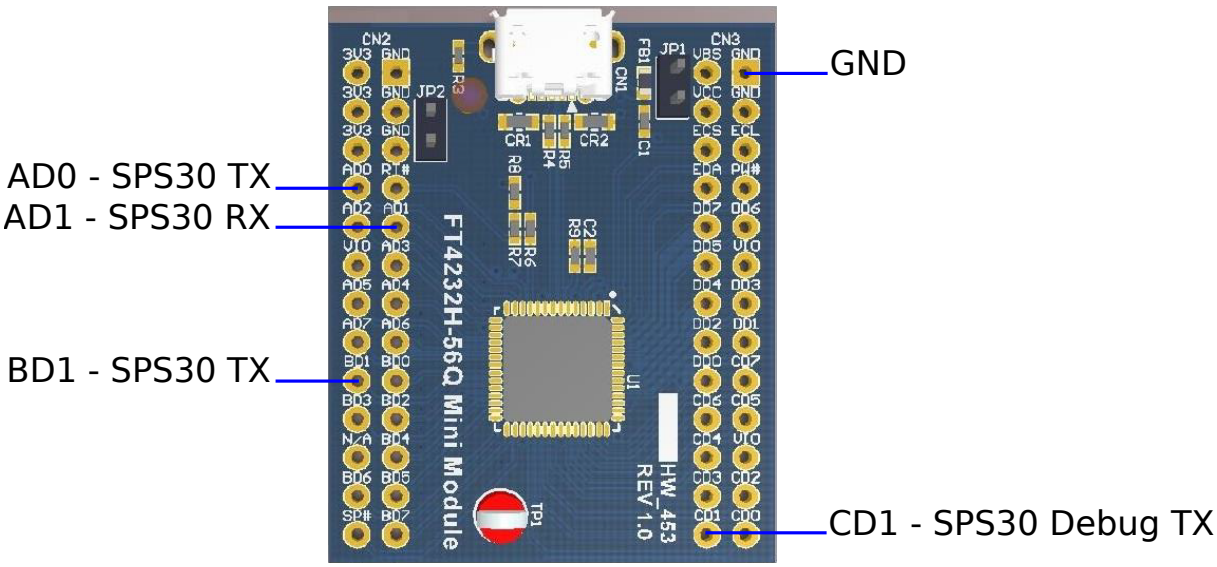

Figura 9: Conexionado del módulo conversor TTL-UART.

2.1.4. PC

El PC realiza una monitorización utilizando el software Docklight. A través del módulo conversor de TTL-UART muestra por pantalla aquella comunicación que se esté realizando. En nuestro caso son siempre comunicaciones usando la interfaz de la UART. Contamos con tres lecturas.

- La conexión entre el pin TX (UART 3) de la placa Nucleo y el pin RX del sensor, por la cual se envían comandos a este último.
- La conexión entre el pin RX (UART 4) de la placa núcleo y el pin TX del sensor, por la cual se reciben las respuestas del sensor a los comandos enviados por la placa.
- El puerto TX (UART 2) de la placa núcleo, el cual está reservado para enviar mensajes para la depuración.

2.2. Diagrama de bloques Firmware del proyecto

En este apartado vamos a mostrar las diferentes funciones de las que se han hecho uso, tanto las que vienen dadas como las creadas. Concretamente, mostraremos el molde de cada función y un diagrama de flujo que muestre su funcionamiento. En la Sección 3.1 desarrollaremos dichos diagramas de flujo.

2.2.1. Driver del Debug

Como parte del proyecto global, partimos de una librería elaborada por el Grupo de Ingeniería Electrónica (GIE), que proporciona los comandos para manejar un puerto de la UART dedicado a la depuración. Aunque la librería no se ha desarrollado en este trabajo, se muestra en el Código 1 la función utilizada para enviar mensajes por el puerto de debug.

```
uint8_t Debug_SendMessage(uint64_t timestamp, uint8_t verbose,char * message, ...);
```

Código 1: Prototipo de la función *Debug_SendMessage*.

Valor de retorno	Expliación
uint8_t	Retorna el código de estado, en caso de ser correcta, retorna DEBUG_MESSAGE_OK
Parámetros de entrada	**Expliación**
timestamp	Marca temporal de cuando se mandó el mensaje, en el caso de valor 0, no se enviará dicho dato
verbose	Nivel de verbose, en el caso de mandar datos por el puerto de debug, VERBOSE_INFO, lo cual hace que se envíe también la etiqueta [INFO]
message	Cadena de caracteres en ASCII que se desea enviar
...	si en la cadena *message* especificamos moldes de datos, por ejemplo %c para un valor tipo char, dicho valor se leerá desde el parámetro de la elipsis

Cuadro 4: Valor de retorno y parámetros de entrada de la función *Debug_SendMessage*.

2.2.2. Driver del SPS30

En este apartado se muestra la lista de funciones implementadas dentro del driver para el sensor SPS30.

secuencia de envío-recepción En todas las funciones que implican la comunicación con la UART se sigue un patrón similar, al que llamaremos en la Subsección 3.1.1 *secuencia de envío-recepción*. Será un bloque predefinido en los diagramas de bloques con el nombre de *comando(envío, tamaño_lectura, tamaño_envío)*.

En la Figura 10 se muestra el contenido de dicho bloque.

empezar_medida El sensor comienza en estado de reposo, con esta función enviamos el comando que hace que comience a realizar medidas. Se muestra el prototipo en el Código 2, sus parámetros en el Cuadro 5 y el diagrama de flujo en la Figura 11.

```
SPS30_command_response empezar_medida(SPS30_measurement_format formato);
```

Código 2: Prototipo de la función *empezar_medida*.

Valor de retorno	Explicación
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	**Explicación**
formato	El sensor puede devolver los datos en formato de coma flotante siguiendo la norma IEEE754 o en un valor entero de 16 bits. El tipo definido SPS30_measurement_format puede tomar los valores *IEEE754* o *Unsigned16* para sendos tipos de medida.

Cuadro 5: Tipo de valor de retorno y parámetro de entrada de la función *empezar_medida*.

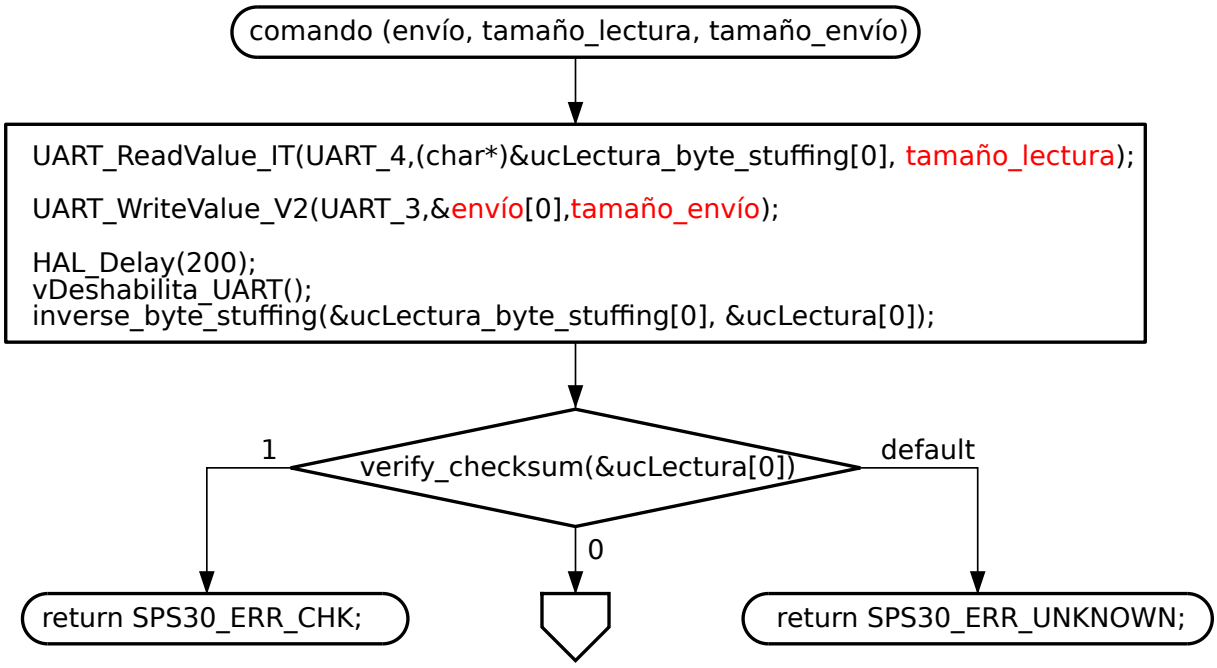

Figura 10: Diagrama de flujo de la secuencia de envío-recepción.

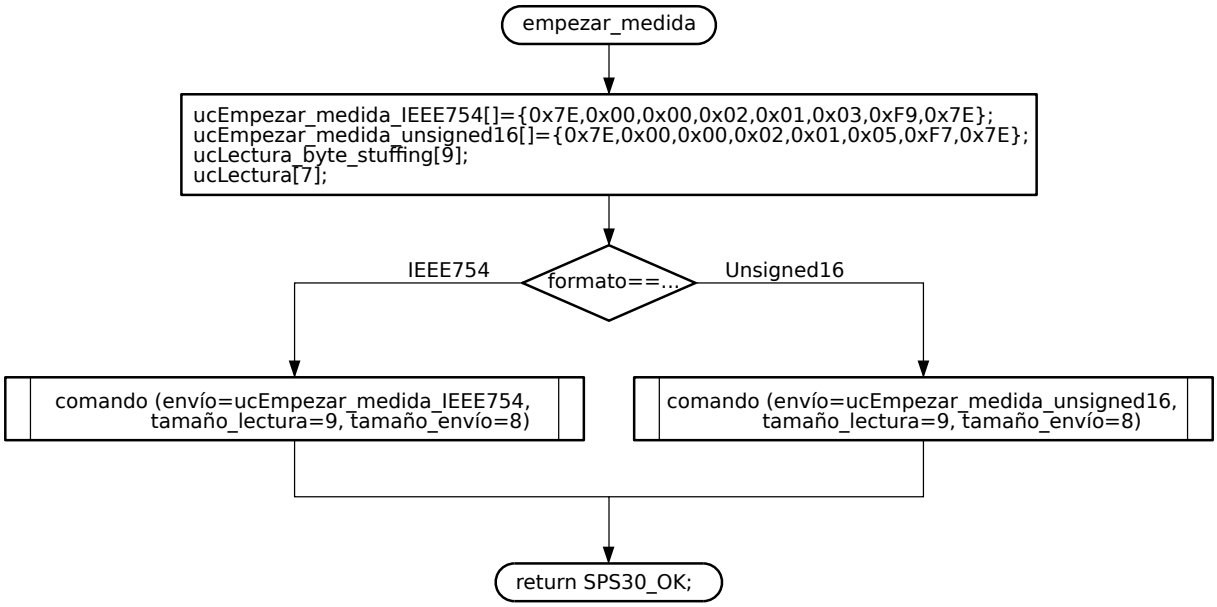

Figura 11: Diagrama de flujo de la función *empezar_medida*.

parar_medida Si necesitamos enviar otro comando al sensor, por ejemplo, para establecer un nuevo tiempo de activación del ventilador, necesitamos que el sensor vuelva al estado de reposo. Para esto utilizamos el comando *parar_medida*. Se muestra el prototipo en el Código 3, sus parámetros en el Cuadro 6 y el diagrama de flujo en la Figura 12.

```
SPS30_command_response parar_medida();
```

Código 3: Prototipo de la función *parar_medida*.

Tipo de valor de retorno	**Explicación**
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	

Cuadro 6: Tipo de valor de retorno de la función *parar_medida*.

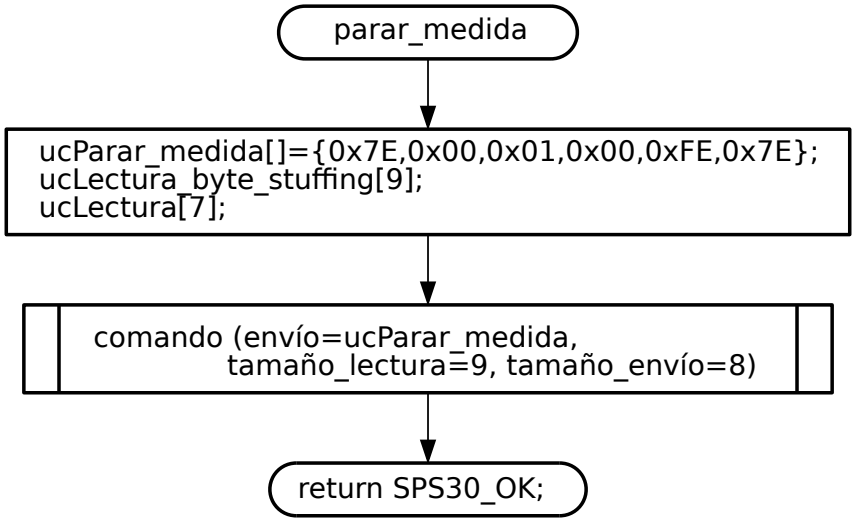

Figura 12: Diagrama de flujo de la función*parar_medida*.

lee_medida Mientras el sensor permanece en el estado de medida, tenemos la posibilidad de solicitar la última realizada. Para ello usamos la función *lee_medida*. Se muestra el prototipo en el Código 4, sus parámetros en el Cuadro 7 y el diagrama de flujo en la Figura 13.

```
SPS30_command_response lee_medida(SPS30_measurement_format formato, xDatos_IEEE754 *medida_IEEE754,
    xDatos_u16 *medida_u16);
```

Código 4: Prototipo de la función *lee_medida*.

sleep En caso de que no se necesite utilizar activamente el sensor, lo más eficiente es cambiar su estado de reposo a estado de hibernación, reduciendo sensiblemente su consumo. Para ello usamos la función *sleep*. Se muestra el prototipo en el Código 5, sus parámetros en el Cuadro 8 y el diagrama de flujo en la Figura 14.

```
SPS30_command_response sleep();
```

Código 5: Prototipo de la función *sleep*.

Tipo de valor de retorno	**Explicación**
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	
Parámetro de entrada	**Explicación**
formato	El sensor puede devolver los datos en formato de coma flotante siguiendo la norma IEEE754 o en un valor entero de 16 bits. El tipo definido *SPS30_measurement_format* puede tomar los valores *IEEE754* o *Unsigned16* para sendos tipos de medida.
Valor devuelto por referencia	**Explicación**
*medida_IEEE754	Puntero a la estructura medida_IEEE754 preparada para almacenar los datos devueltos en formato IEEE754, se usará solo si el formato elegido es el compatible.
*medida_u16	Puntero a la estructura medida_u16 preparada para almacenar los datos devueltos en formato Unsigned de 16 bits, se usará solo si el formato elegido es el compatible.

Cuadro 7: Tipo de valor de retorno, parámetro de entrada y valores devueltos por la función *lee_medida*.

Tipo de valor de retorno	**Explicación**
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	

Cuadro 8: Tipo de valor de retorno de la función *sleep*.

wake_up Solo es posible salir del estado de hibernación mediante el uso de la función *wake_up*. Se muestra el prototipo en el Código 6, sus parámetros en el Cuadro 9 y el diagrama de flujo en la Figura 15.

```
SPS30_command_response wake_up();
```

Código 6: Prototipo de la función *wake_up*.

Tipo de valor de retorno	**Explicación**
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	

Cuadro 9: Tipo de valor de retorno de la función *wake_up*.

limpieza El ventilador con el que cuenta el sensor se activa periódicamente según el tiempo establecido, ya sea por defecto o de manera manual. Si queremos activar el proceso de limpieza de manera puntual podemos usar la función *limpieza*. Se muestra el prototipo en el Código 7, sus parámetros en el Cuadro 10 y el diagrama de flujo en la Figura 16.

```
SPS30_command_response limpieza();
```

Código 7: Prototipo de la función *limpieza*.

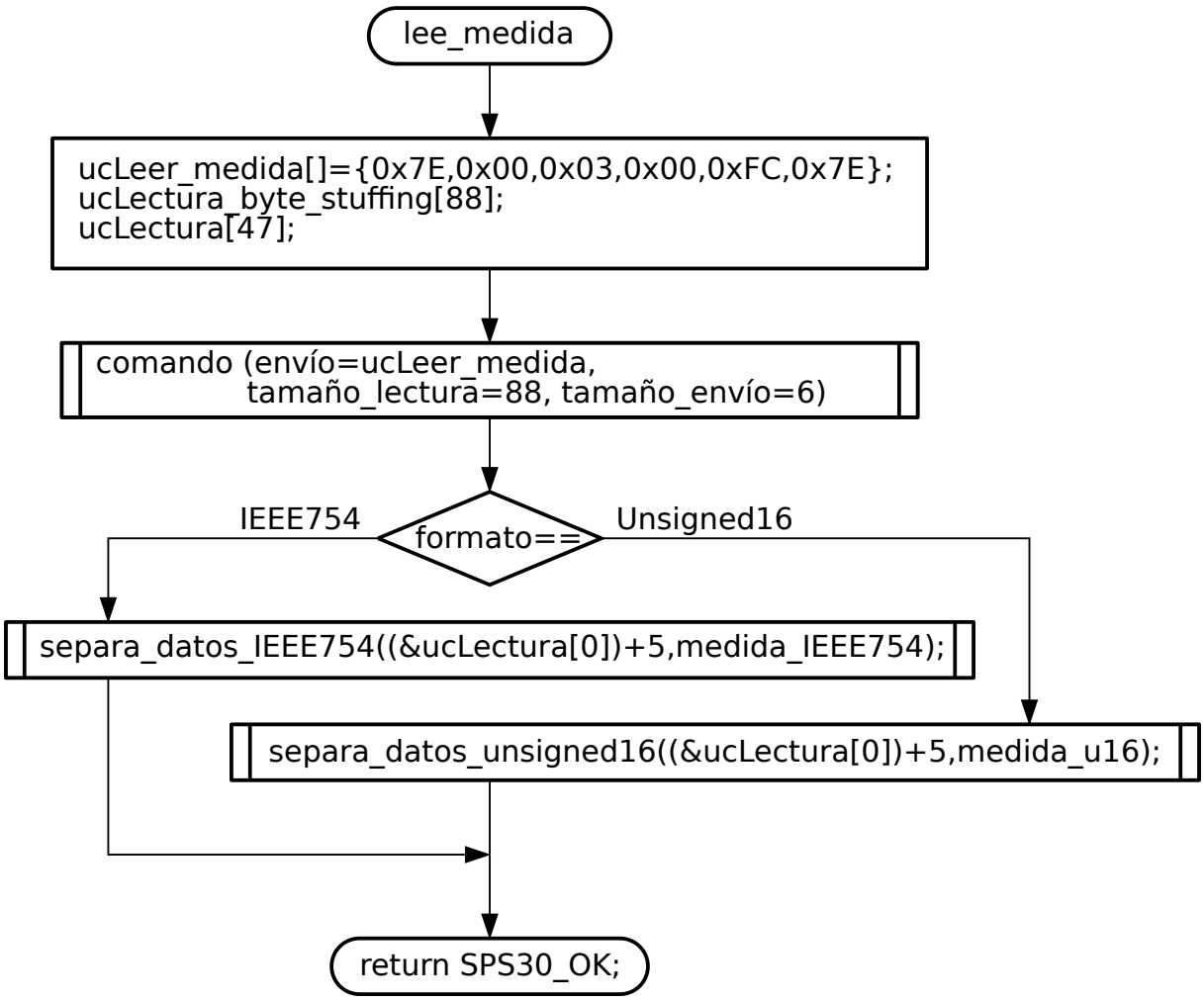

Figura 13: Diagrama de flujo de la función*lee_medida*.

set_cleaning_interval El sensor activará su ventilador para realizar un limpieza de manera periódica. Este tiempo está configurado por defecto a una semana, se puede modificar usando la función *set_cleaning_interval*. Se muestra el prototipo en el Código 8, sus parámetros en el Cuadro 11 y el diagrama de flujo en la Figura 17.

```
SPS30_command_response set_cleaning_interval(uint32_t ulIntervalo);
```

Código 8: Prototipo de la función *set_cleaning_interval*.

read_cleaning_interval En caso de que queramos conocer el tiempo establecido entre limpiezas usaremos la función *read_cleaning_interval*. Se muestra el prototipo en el Código 9, sus parámetros en el Cuadro 12 y el diagrama de flujo en la Figura 18.

```
SPS30_command_response read_cleaning_interval(float * out_cleaning_interval);
```

Código 9: Prototipo de la función *read_cleaning_interval*.

sensor_aire_reset La función *sensor_aire_reset* implementa un reinicio suave del sensor, el cual vuelve al estado de reposo y actúa de la misma manera que si se hubiera apagado y vuelto a encender desde la alimentación. Se muestra el prototipo en el Código 10, sus parámetros en el Cuadro 13 y el diagrama de flujo en la Figura 19.

```
SPS30_command_response sensor_aire_reset();
```

Código 10: Prototipo de la función *sensor_aire_reset*.

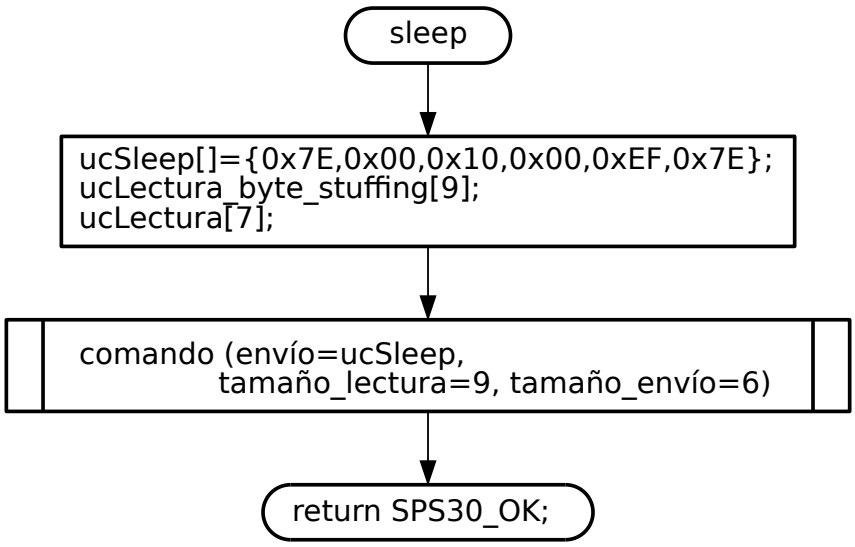

Figura 14: Diagrama de flujo de la función*sleep*.

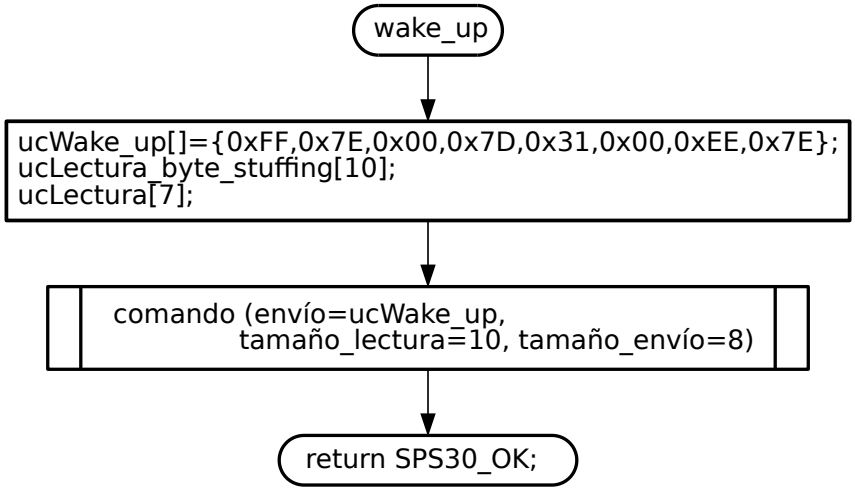

Figura 15: Diagrama de flujo de la función*wake_up*.

get_serial_number Para leer el número de serie de nuestra unidad usamos la función *get_serial_number*. Se muestra el prototipo en el Código 11, sus parámetros en el Cuadro 14 y el diagrama de flujo en la Figura 20.

```
SPS30_command_response get_serial_number(char * serial_number);
```

Código 11: Prototipo de la función *get_serial_number*.

read_version La función *read_version* permite conocer las distintas versiones del sensor:

- Versión de firmware, mayor y menor.
- Versión de hardware.
- Versión del protocolo SHDLC, mayor y menor.

Se muestra el prototipo en el Código 12, sus parámetros en el Cuadro 15 y el diagrama de flujo en la Figura 21.

```
SPS30_command_response read_version(version *version_leida);
```

Código 12: Prototipo de la función *read_version*.

Tipo de valor de retorno	Explicación
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	

Cuadro 10: Tipo de valor de retorno de la función *limpieza*.

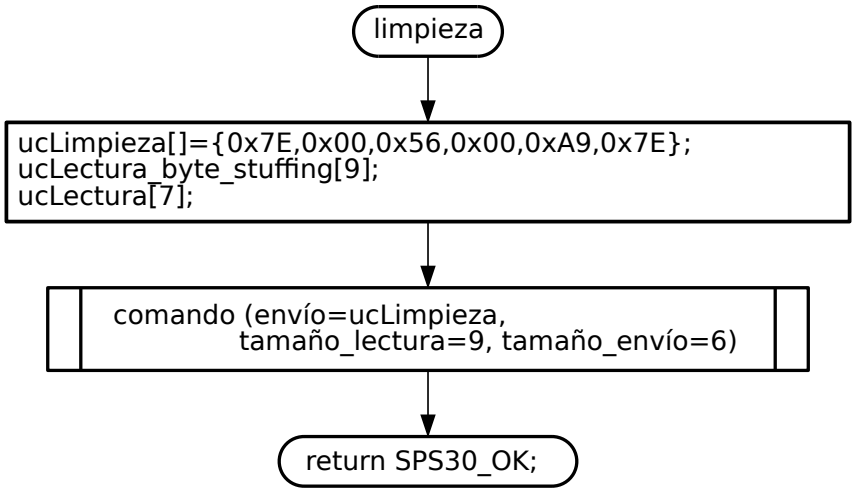

Figura 16: Diagrama de flujo de la función*limpieza*.

read_status_register La función *read_status_register* permite leer el registro de estado del sensor, que determina tres posibles problemas según los bits que estén activados:

- Bit 21: velocidad del ventilador fuera de rango.
- Bit 5: fallo en el láser.
- Bit 4: fallo en el ventilador, pudiendo estar atascado o roto.

Se muestra el prototipo en el Código 13, sus parámetros en el Cuadro 16 y el diagrama de flujo en la Figura 22.

```
SPS30_command_response read_status_register(char formato,char * status_code);
```

Código 13: Prototipo de la función *read_status_register*.

longint_to_byte_string En la Función *set_cleaning_interval* necesitamos convertir el tiempo especificado de una variable de 32 bits a una cadena de cuatro datos de 8 bits. La función *longint_to_byte_string* realiza esta conversión. Se muestra el prototipo en el Código 14, sus parámetros en el Cuadro 17 y el diagrama de flujo en la Figura 23.

```
void longint_to_byte_string(unsigned long int entrada, unsigned char * puntero_vector_salida);
```

Código 14: Prototipo de la función *longint_to_byte_string*.

verify_checksum El sensor puede responder a los distintos comandos devolviendo los datos solicitados o devolviendo una cadena de confirmación con un byte de estado. En cualquiera de los casos es conveniente comprobar que el checksum sea válido y no se ha producido un fallo en la comunicación. La función *verify_checksum* recibe un frame y analiza si el byte de checksum incluido se corresponde con el resto del frame. Se muestra el prototipo en el Código 15, sus parámetros en el Cuadro 18 y el diagrama de flujo en la Figura 24.

```
unsigned char verify_checksum(uint8_t * pcPuntero_datos_checksum);
```

Código 15: Prototipo de la función *verify_checksum*.

Tipo de valor de retorno	Explicación
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	
Parámetro de entrada	**Explicación**
ulIntervalo	Tiempo en segundos que queremos que dure el intervalo entre limpiezas. Dado que este valor es de tipo Unsigned int de 32 bits, el valor máximo posible será de más de 100 años.

Cuadro 11: Tipo de valor de retorno y parámetro de entradade la función *set_cleaning_interval*.

Tipo de valor de retorno	Explicación
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	
Valor devuelto por referencia	**Explicación**
*out_cleaning_interval	Tiempo en segundos que dura el intervalo entre limpiezas.

Cuadro 12: Tipo de valor de retorno y valor devuelto por referencia de la función *read_cleaning_interval*.

calculate_checksum En la Función *set_cleaning_interval* tenemos como entrada el parámetro *ulIntervalo*, cuyo valor no conocemos previamente, de manera que el comando a enviar se genera dentro de la función. Esto implica que necesitamos calcular un checksum de acuerdo con los datos generados. Esta tarea la realiza la función *calculate_checksum*. Se muestra el prototipo en el Código 16, sus parámetros en el Cuadro 19 y el diagrama de flujo en la Figura 25.

```
unsigned char calculate_checksum(unsigned char * puntero_datos_checksum);
```

Código 16: Prototipo de la función *calculate_checksum*.

byte_stuffing En la Función *set_cleaning_interval* tenemos como entrada el parámetro *ulIntervalo*, cuyo valor no conocemos previamente, de manera que el comando a enviar se genera dentro de la función. Esto implica que se pueden generar bytes que necesitan ser reemplazados para su correcta interpretación. La función *byte_stuffing* realiza este cambio. Se muestra el prototipo en el Código 17, sus parámetros en el Cuadro 21 y el diagrama de flujo en la Figura 26.

```
int byte_stuffing (uint8_t ucVector_original[],uint8_t ucSize, uint8_t *pucCadena_bit_stuffing);
```

Código 17: Prototipo de la función *byte_stuffing*.

inverse_byte_stuffing Cada vez que el sensor nos responde a un comando o nos entrega datos, existe la posibilidad de que los bytes que llegan hayan sido sustituidos por byte-stuffing. La función *inverse_byte_stuffing* invierte este cambio, según el Cuadro 20. Se muestra el prototipo en el Código 18, sus parámetros en el Cuadro 22 y el diagrama de flujo en la Figura 27.

```
int inverse_byte_stuffing(uint8_t ucCadena_original[], uint8_t ucCadena_inverse_byte_stuffing[]);
```

Código 18: Prototipo de la función *inverse_byte_stuffing*.

separa_datos_IEEE754 En la Función *lee_medida* recibimos por parte del sensor un frame de tamaño variable en el que vienen incluidas las medidas realizadas en forma de cadena de datos de tamaño 1 byte. Necesitamos procesar esa cadena y separarla en las distintas medidas que contiene. Para separar

Valor de retorno	Explicación
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	

Cuadro 13: Tipo de valor de retorno de la función *sensor_aire_reset*.

Tipo de valor de retorno	Explicación
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	
Valor devuelto por referencia	**Explicación**
*serial_number	Puntero hacia el número de serie de la unidad que se esté utilizando

Cuadro 14: Tipo de valor de retorno, parámetro de entrada y valores devueltos por la función *lee_medida*.

las medidas realizadas con formato IEEE754 usamos la función *separa_datos_IEEE754*. Se muestra el prototipo en el Código 19, sus parámetros en el Cuadro 23 y el diagrama de flujo en la Figura 28.

```
void separa_datos_IEEE754(uint8_t ucLectura[],xDatos_IEEE754 *medida);
```

Código 19: Prototipo de la función *separa_datos_IEEE754*.

separa_datos_unsigned16 En la Función *lee_medida* recibimos por parte del sensor un frame de tamaño variable en el que vienen incluidas las medidas realizadas en forma de cadena de datos de tamaño 1 byte. Necesitamos procesar esa cadena y separarla en las distintas medidas que contiene. Para separar las medidas realizadas con formato Unsigned de 16 bits usamos la función *separa_datos_unsigned16*. Se muestra el prototipo en el Código 20, sus parámetros en el Cuadro 24 y el diagrama de flujo en la Figura 29.

```
void separa_datos_unsigned16(uint8_t *pucPuntero_lectura,xDatos_u16 *medida)
```

Código 20: Prototipo de la función *separa_datos_unsigned16*.

Tipo de valor de retorno	**Explicación**
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	
Valor devuelto por referencia	**Explicación**
∗firmware_major_version	Puntero hacia la versión mayor del Firmware
∗firmware_minor_version	Puntero hacia la versión menor del Firmware
∗hardware_version	Puntero hacia la versión mayor del Hardware
∗SHDLC_protocol_major_version	Puntero hacia la versión mayor del Protocolo SHDLC
∗SHDLC_protocol_minor_version	Puntero hacia la versión menor del Protocolo SHDLC

Cuadro 15: Tipo de valor de retorno, parámetro de entrada y valores devueltos por la función *read_version*.

Tipo de valor de retorno	**Explicación**
SPS30_command_response	El tipo definido puede devolver los valores *SPS30_OK*, *SPS30_ERR_CHK*, *SPS30_ERR_UNKNOWN*
Parámetro de entrada	
Parámetro de entrada	Explicación
formato	Podemos elegir entre resetear el registro de estado tras leerlo dando el valor de 1 al formato, o no borrarlo dando un 1.
Valor devuelto por referencia	**Explicación**
∗status_code	Vector hacia el vector de registro de estado. Cada celda representa respectivamente los bits 21, 5 y 4 del registro de estado.

Cuadro 16: Tipo de valor de retorno, parámetro de entrada y valores devueltos por la función *read_version*.

Parámetro de entrada	**Explicación**
entrada	Número en formato Unsigned de 32 bits el cual queremos separar.
Valor devuelto por referencia	**Explicación**
∗puntero_vector_salida	Puntero que señala a un vector de cuatro celdas de tipo Unsigned de 8 bits donde separaremos el parámetro de entrada.

Cuadro 17: Parámetro de entrada y valores devueltos por la función *longint_to_byte_string*.

Tipo de dato de retorno	**Explicación**
unsigned char	0 en caso de validar la operación o 1 en caso de haber discrepancia.
Parámetro de entrada	**Explicación**
∗pcPuntero_datos_checksum	Puntero que señala al vector que contiene el frame que se desea analizar

Cuadro 18: Tipo de dato de retorno y parámetro de entrada de la función *verify_checksum*.

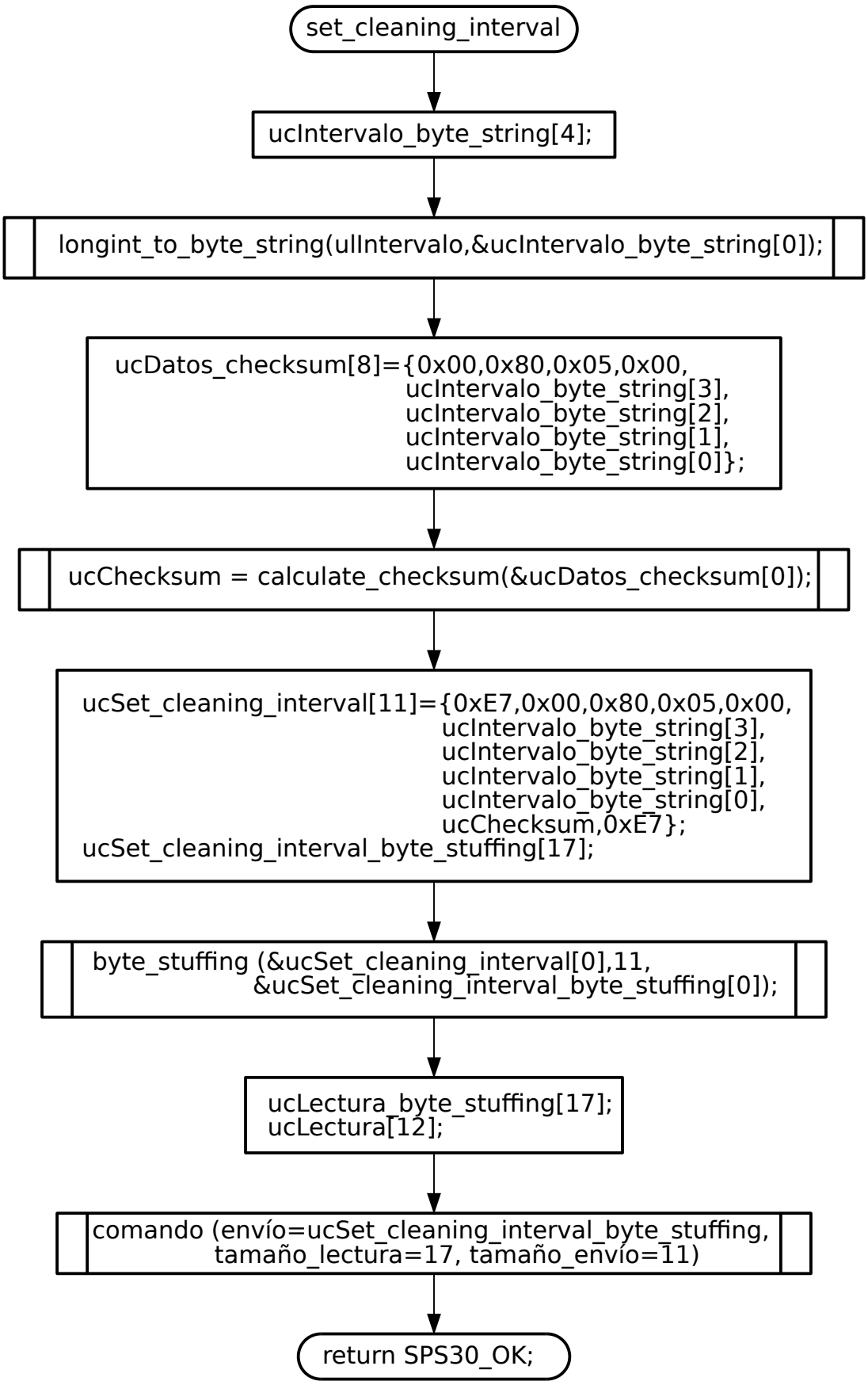

Figura 17: Diagrama de flujo de la función*set_cleaning_interval*.

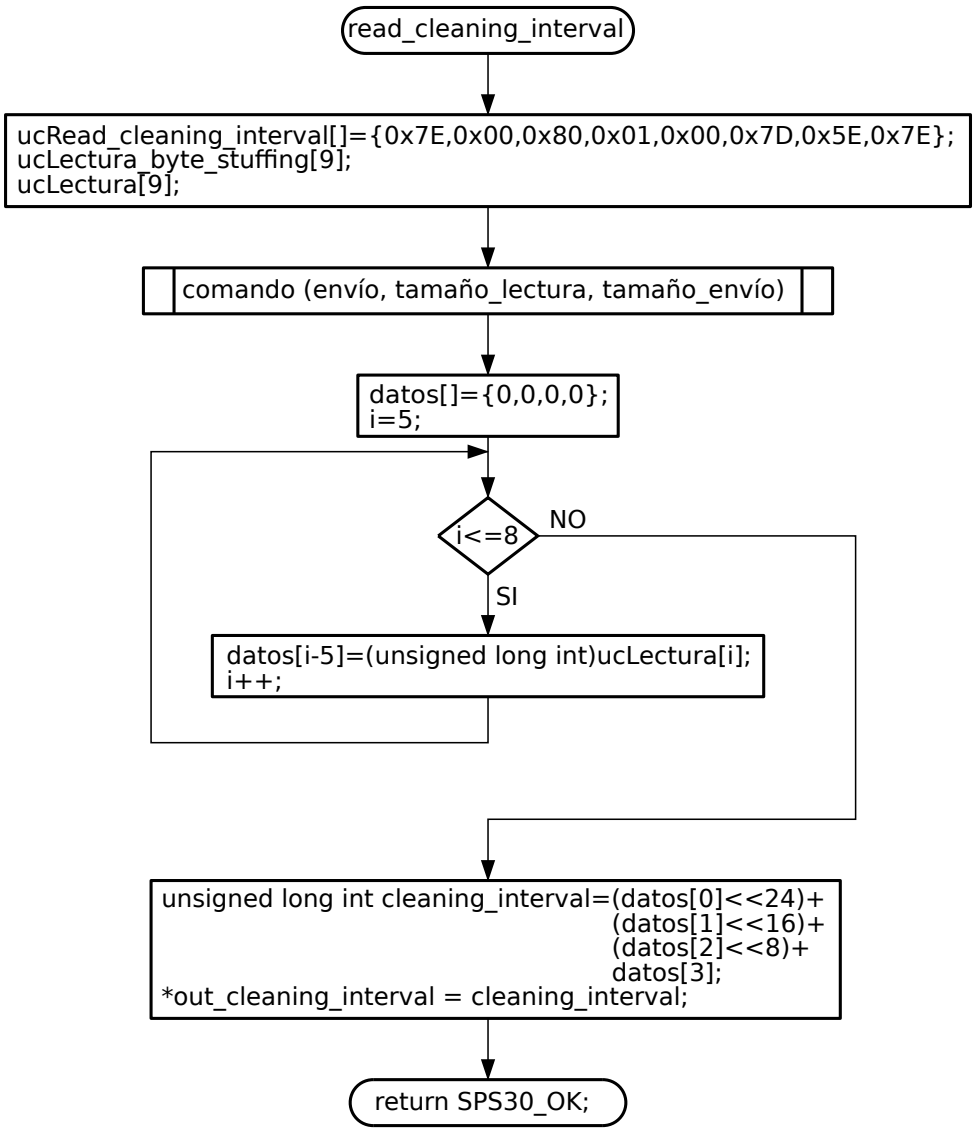

Figura 18: Diagrama de flujo de la función*read_cleaning_interval*.

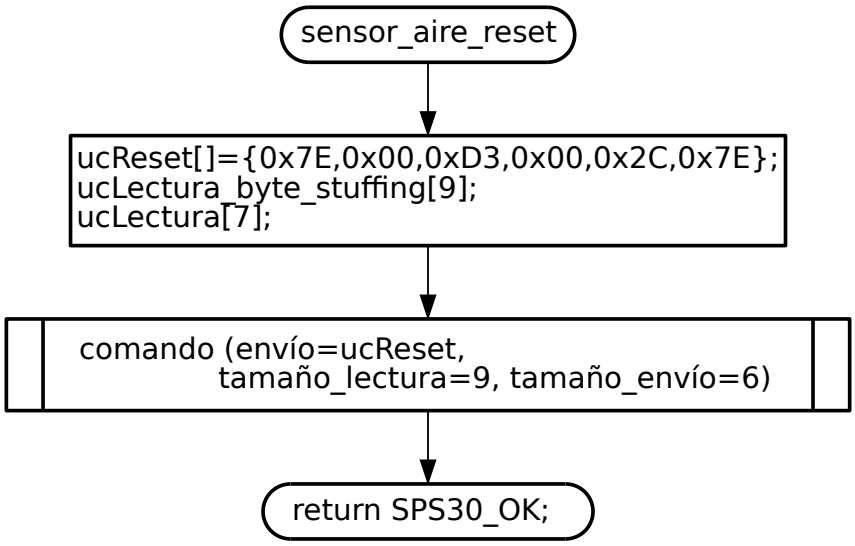

Figura 19: Diagrama de flujo de la función*sensor_aire_reset*.

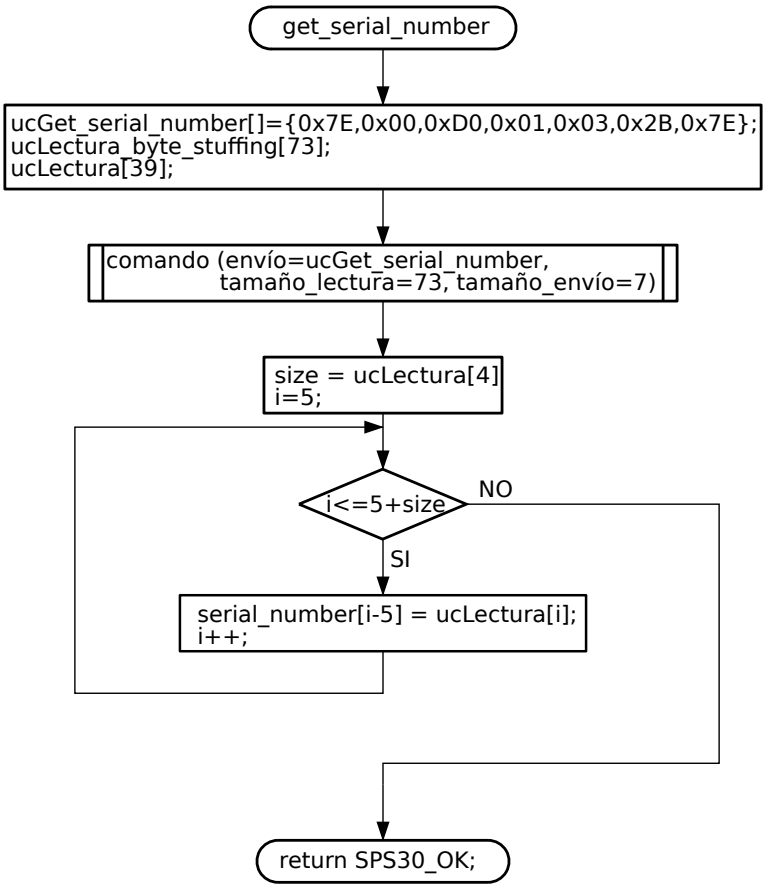

Figura 20: Diagrama de flujo de la función*get_serial_number*.

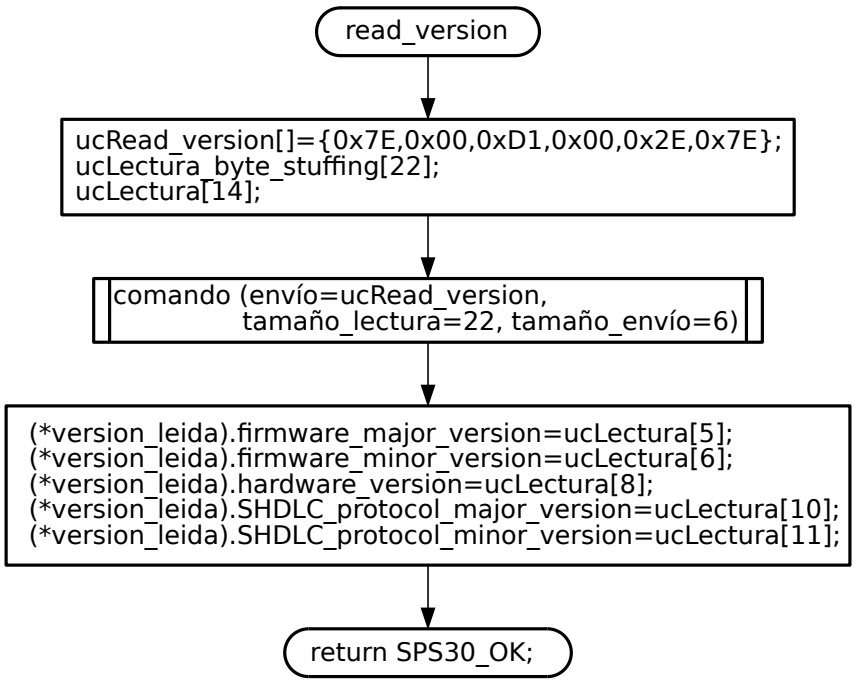

Figura 21: Diagrama de flujo de la función*read_version*.

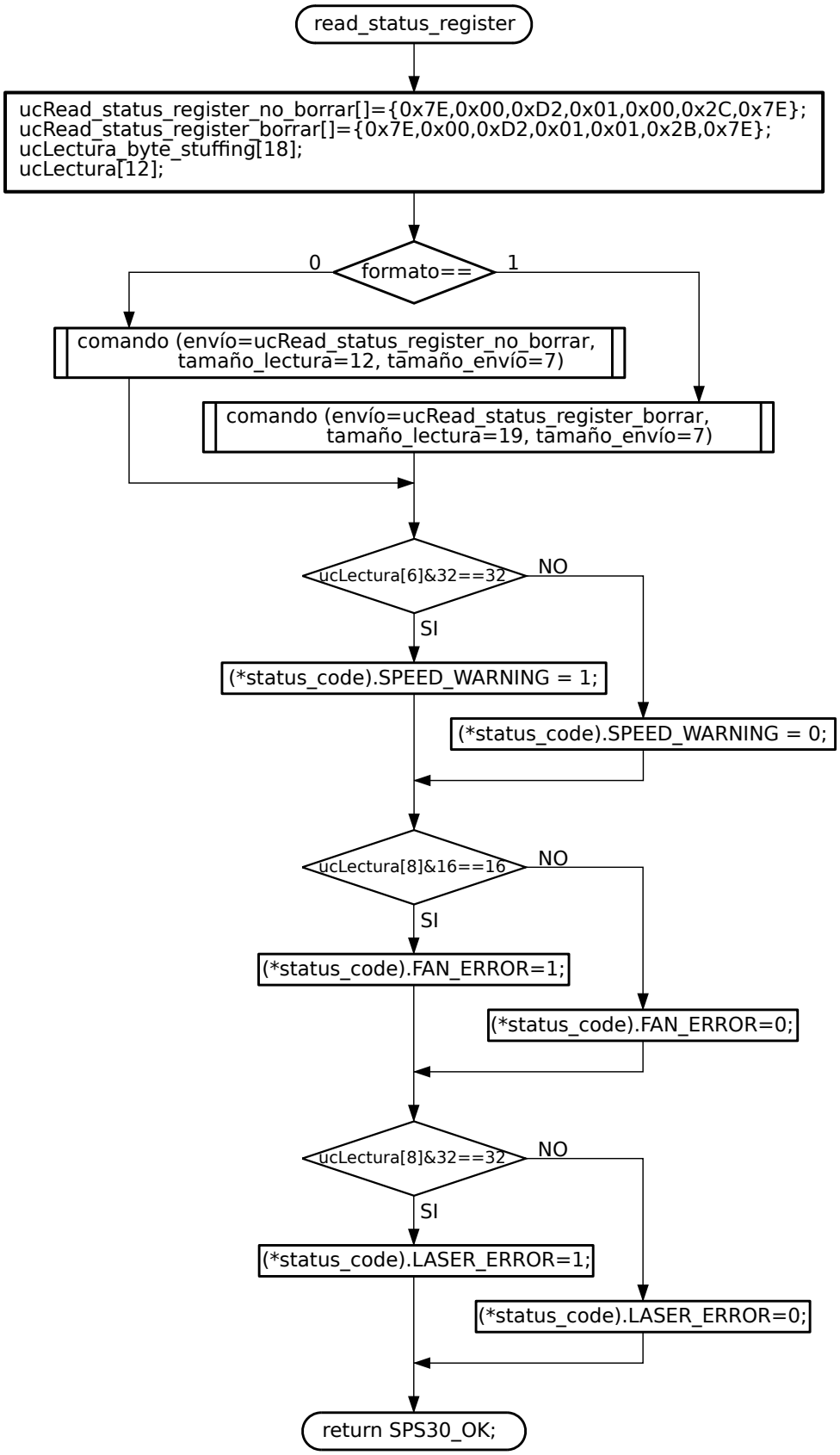

Figura 22: Diagrama de flujo de la función*read_status_register*.

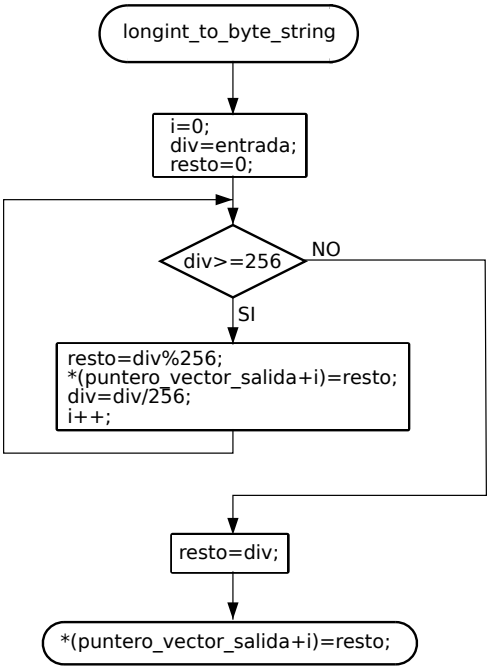

Figura 23: Diagrama de flujo de la función*longint_to_bytestring*.

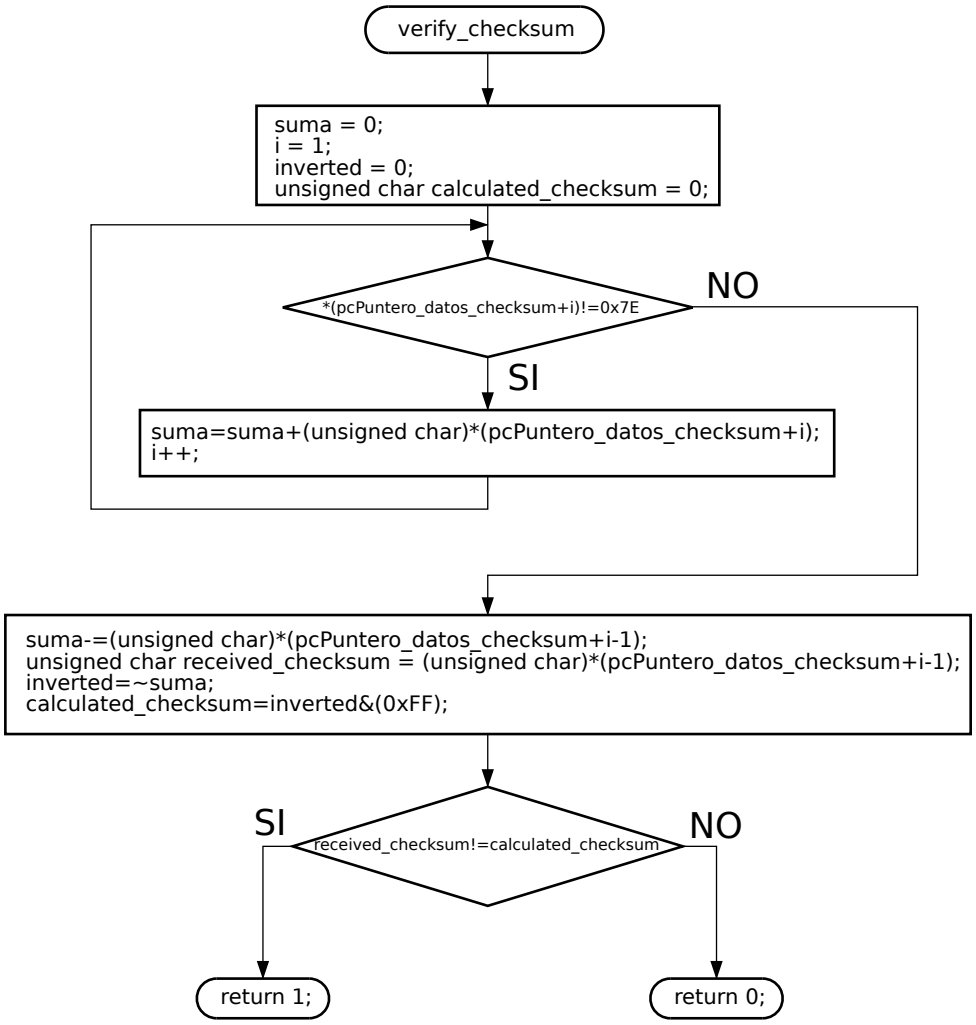

Figura 24: Diagrama de flujo de la función*verify_checksum*.

Tipo de dato de retorno	Explicación
unsigned char	Se devolverá el valor del checksum, que ocupa un byte, de manera que lo procesamos como un char (tamaño de un byte) sin signo
Parámetro de entrada	**Explicación**
∗puntero_datos_checksum	Puntero que señala al vector que contiene los datos necesarios para calcular el checksum

Cuadro 19: Tipo de dato de retorno y parámetro de entrada de la función *calculate_checksum*.

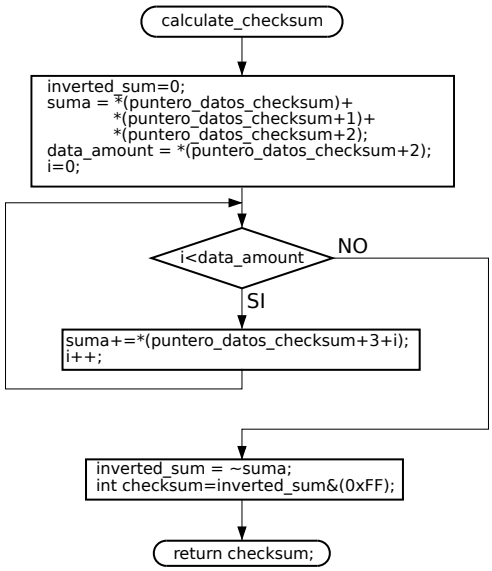

Figura 25: Diagrama de flujo de la función*calculate_checksum*.

Byte original	Grupo de bytes enviados
0x7E	0x7D, 0x5E
0x7D	0x7D, 0x5D
0x11	0x7D, 0x31
0x13	0x7D, 0x33

Cuadro 20: Valores que necesitan ser reemplazados por byte-stuffing.

Tipo de dato de retorno	Explicación
int	Se devolverá el tamaño de la nueva cadena de envío generada
Parámetros de entrada	**Explicación**
ucVector_original[]	Vector que contiene la cadena original que se quiere transmitir
ucSize	Tamaño de la cadena original
Valor devuelto por referencia	**Explicación**
∗pucCadena_bit_stuffing	Vector donde se guardará la cadena resultante de aplicar byte-stuffing

Cuadro 21: Tipo de dato de retorno y parámetro de entrada de la función *calculate_checksum*.

Figura 26: Diagrama de flujo de la función*byte_stuffing*.

Tipo de dato de retorno	**Explicación**
int	Se devolverá el tamaño de la nueva cadena
Parámetro de entrada	**Explicación**
ucCadena_original[]	Vector que contiene la cadena original a la que se quiere deshaer el byte-stuffing
Valor devuelto por referencia	**Explicación**
ucCadena_inverse_byte_stuffing[]	Vector donde se guardará la cadena resultante de deshacer el byte-stuffing

Cuadro 22: Tipo de dato de retorno, parámetro de entrada y valor devuelto por la función *inverse_byte_stuffing*.

Parámetro de entrada	**Explicación**
ucLectura	Vector que contiene el frame de datos del sensor depués de deshacer el byte-stuffing
Valor devuelto por referencia	**Explicación**
∗medida	Puntero hacia la estructura que contiene en cada elemento las distintas medidas que el sensor otorga en tipo float

Cuadro 23: Parámetro de entrada y valor devuelto por la función *separa_datos_IEEE754*.

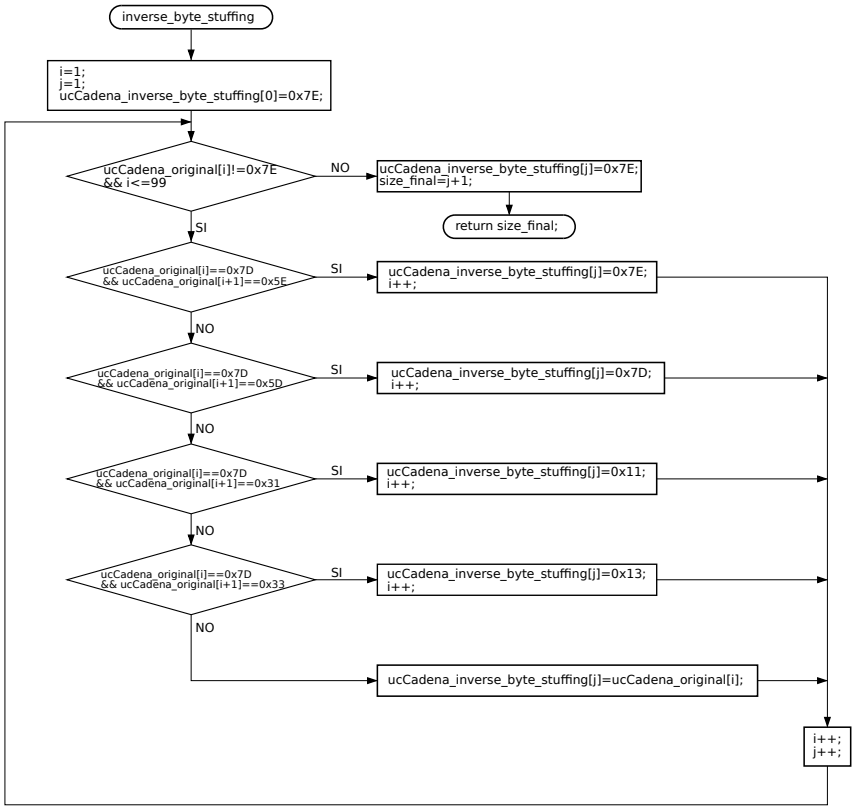

Figura 27: Diagrama de flujo de la función*inverse_byte_stuffing*.

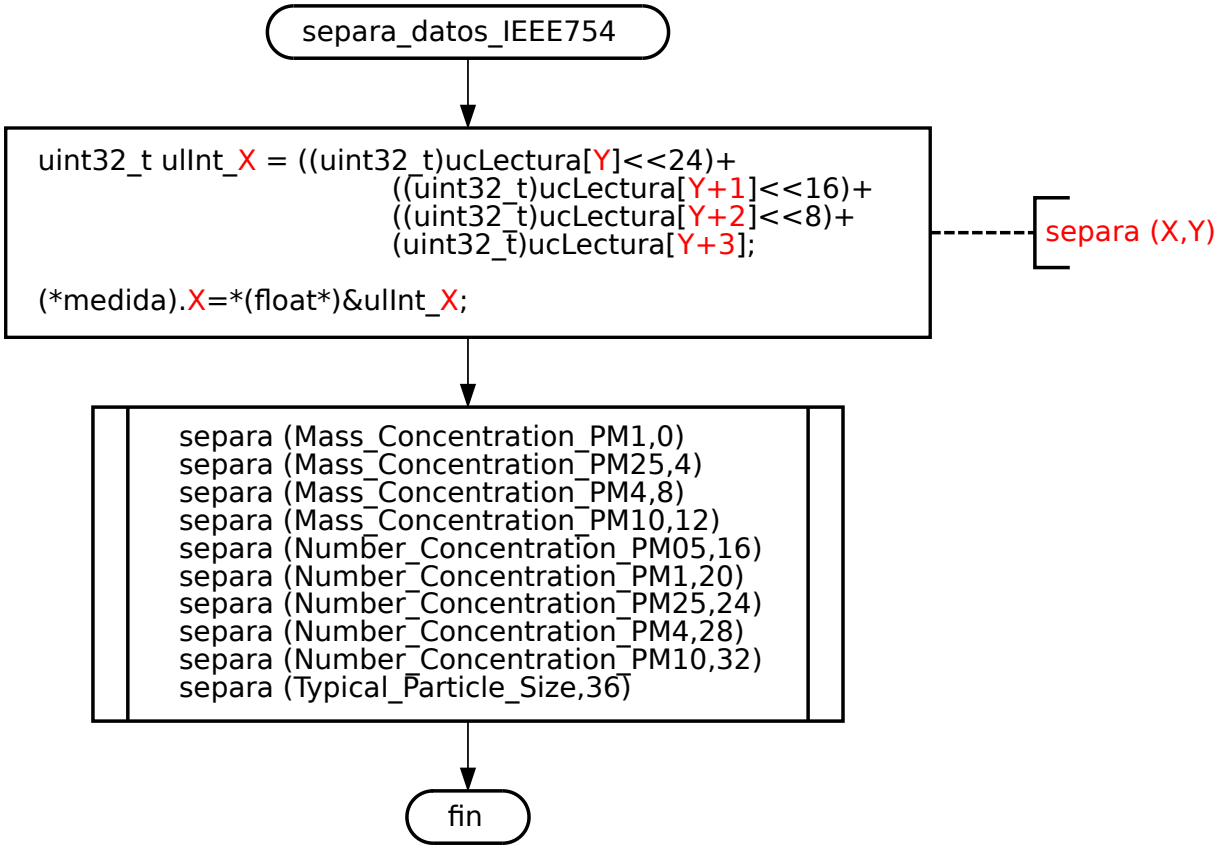

Figura 28: Diagrama de flujo de la función*separa_datos_IEEE754*.

Parámetro de entrada	Explicación
∗pucPuntero_lectura	Puntero al vector que contiene el frame de datos del sensor depués de deshacer el byte-stuffing
Valor devuelto por referencia	**Explicación**
∗medida	Puntero hacia la estructura que contiene en cada elemento las distintas medidas que el sensor otorga en tipo Unsigned de 16 bits

Cuadro 24: Parámetro de entrada y valor devuelto por la función *separa_datos_unsigned16*.

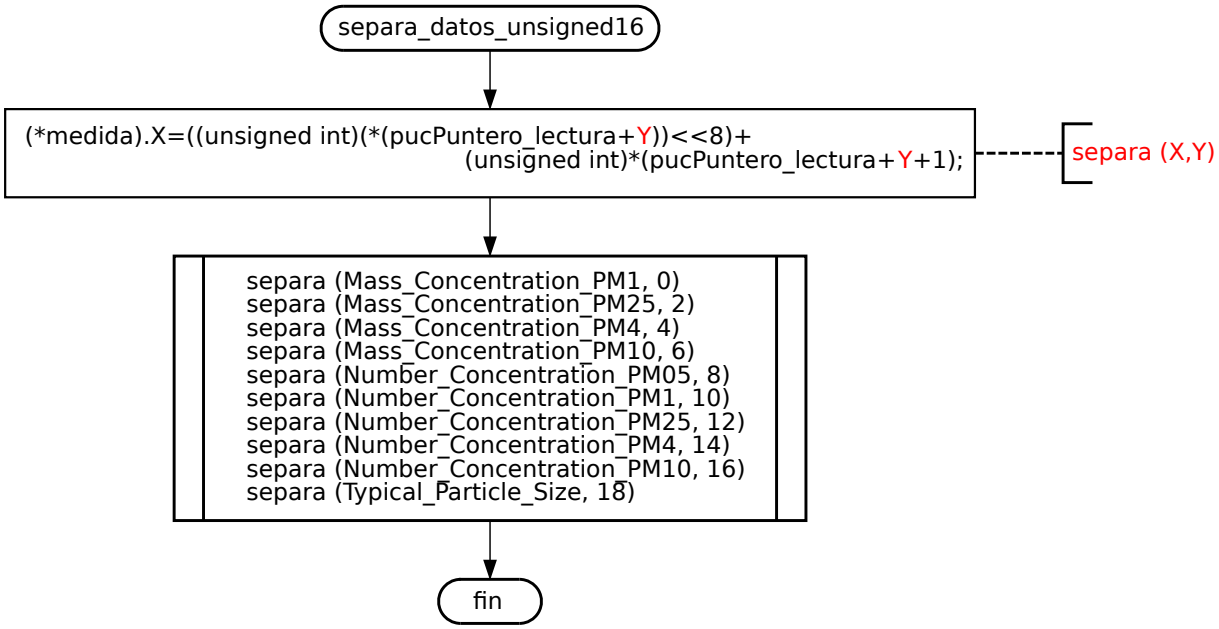

Figura 29: Diagrama de flujo de la función*separa_datos_unsigned16*.

2.2.3. Driver de la UART

Anteriormente hemos hablado de que partimos de una librería elaborada por el GIE. Entre otras funciones, proporciona los comandos para manejar la UART. Para esta aplicación se ha extendido dicha librería con comandos propios. No se mostrarán diagramas de flujo de las funciones que no has sido creadas desde cero ya que no son el objetivo de este trabajo.

Función de partida, UART_WriteValue La función asignada para recibir datos es *UART_WriteValue*. En la Subsección 3.1.2 vamos a explicar que esta función no se adecúa al formato que necesitamos y por tanto realizaremos modificaciones.

```
uint8_t UART_WriteValue(const uint8_t UART_NUMBER,char * str,...);
```

Código 21: Prototipo de la función *UART_WriteValue*.

Valor de retorno	**Explicación**
uint8_t	La función devuelve el resultado de la operación, que puede ser UART_OK, UART_OUT_OF_RANGE, UART_INVALID_PRINF_ARG o UART_NOT_INIT
Parámetros de entrada	**Explicación**
UART_NUMBER	Número de la UART que se quiere usar para escribir, de la 1 a la 5
*str	Cadena de caracteres que se quiere enviar
...	La elipsis permite introducir un número indeterminado de parámetros haciendo que ésta sea una función *variadic*

Cuadro 25: Tipo de valor de retorno y parámetros de entrada de la función *UART_WriteValue*.

Función modificada, UART_WriteValue_V2 Se muestra el prototipo en el Código 22 y sus parámetros en el Cuadro 26.

```
uint8_t UART_WriteValue_V2(const uint8_t UART_NUMBER,uint8_t * cstr,uint8_t csize);
```

Código 22: Prototipo de la función *UART_WriteValue_V2*.

Valor de retorno	**Explicación**
uint8_t	La función devuelve el resultado de la operación, que puede ser UART_OK, UART_OUT_OF_RANGE, UART_INVALID_PRINF_ARG o UART_NOT_INIT
Parámetros de entrada	**Explicación**
UART_NUMBER	Número de la UART que se quiere usar para escribir, de la 1 a la 5
*cstr	Cadena de caracteres que se quiere enviar
csize	Tamaño de la cadena que se quiere enviar

Cuadro 26: Tipo de valor de retorno y parámetros de entrada de la función *UART_WriteValue_V2*.

Función de partida, UART_ReadValue_IT La función asignada para recibir datos por interrupción es *UART_ReadValue_IT*.

```
uint8_t UART_ReadValue_IT(const uint8_t UART_NUMBER, char *str, int size);
```

Código 23: Prototipo de la función *UART_ReadValue_IT*.

Valor de retorno	Explicación
uint8_t	La función devuelve el resultado de la operación, que puede ser UART_OK, UART_OUT_OF_RANGE, UART_PARAM_EXC, HAL_TIMEOUT, UART_DATA_LOST, UART_NOT_INIT o UART_RX_OVER
Parámetros de entrada	**Explicación**
UART_NUMBER	Número de la UART que se quiere usar para escribir, de la 1 a la 5
*str	Cadena de caracteres que se quiere enviar
size	Tamaño de la cadena que se espera leer

Cuadro 27: Tipo de valor de retorno y parámetros de entrada de la función *UART_ReadValue_IT*.

vDeshabilita_UART Hemos dicho que se ha implementado una lectura de la UART por interrupción, mediante la función *UART_ReadValue_IT*. Para que esta lectura por interrupción se detenga es necesario recibir el número de caracteres indicado como parámetro. Pero como hemos explicado, debido al byte-stuffing este valor no puede ser conocido de antemano. Si volvemos a ejecutar este comando sin haber finalizado una lectura, los datos no se sobrescriben en el vector destino, sino que se añaden a nuevas celdas hasta saturar dicho vector.

Para evitar este comportamiento cerramos manualmente la transmisión por interrupción de la UART después de cada lectura mediante la función *vDeshabilita_UART*.

```
void vDeshabilita_UART();
```

Código 24: Prototipo de la función *vDeshabilita_UART*.

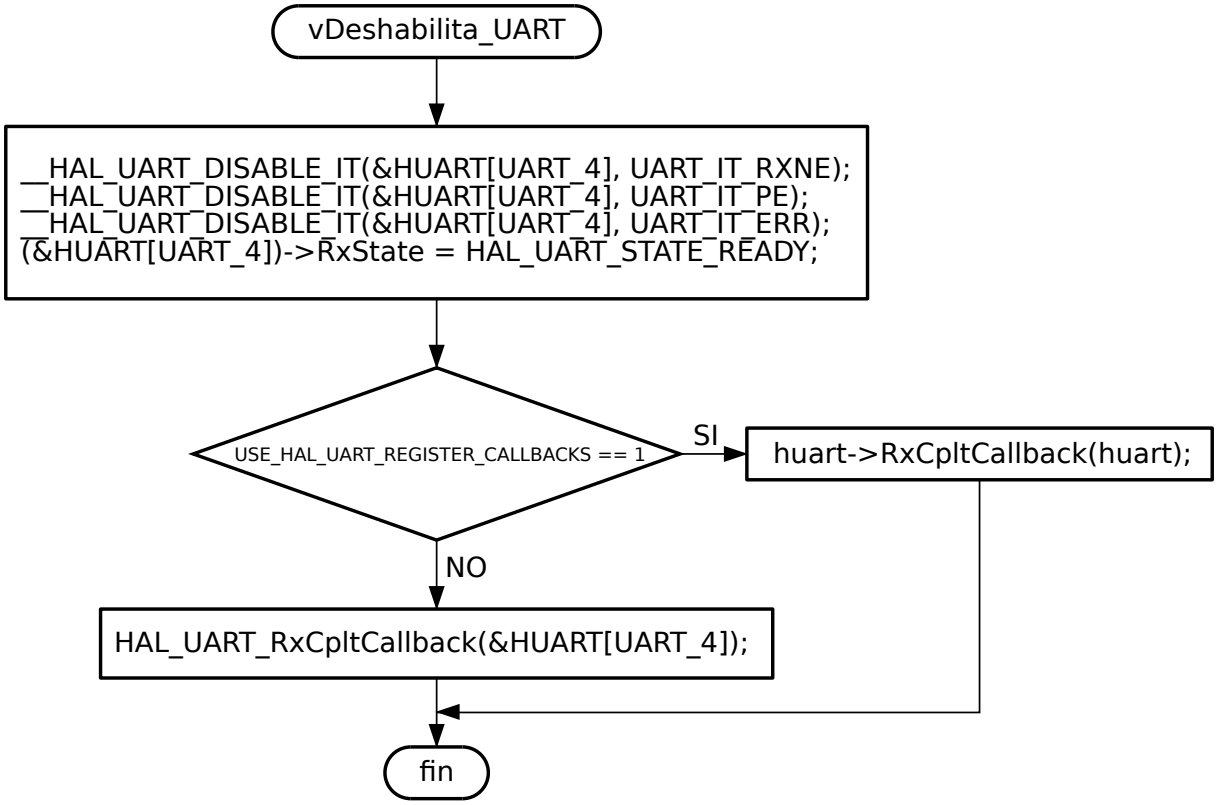

Figura 30: Diagrama de flujo de la función*vDeshabilita_UART*.

2.2.4. Integración en FreeRTOS

Una vez que tenemos preparados los drivers y se ha corroborado su correcto funcionamiento, es hora de integrar las funciones creadas en el entorno de FreeRTOS. Para ellos crearemos distintas tareas que llamarán a funciones concretas de nuestra librería, con el objetivo de, posteriormente, llamar y asignar una determinada prioridad a esas tareas para que se vayan ejecutando.

Drivers anteriores Recapitulando, en este punto tenemos tres drivers para utilizar, cada uno con un propósito:

- Driver del SPS30, que contiene todos los protocolos para transmitir y recibir datos con el sensor y se construye sobre el driver de la UART.
- Driver de la UART, para realizar las comunicaciones usando dicha interfaz.
- Driver de debug, que se puede suprimir pero que utilizaremos en los test para verificar el correcto funcionamiento de las librerías.

Como pueden existir distintas tareas de este driver intentando usar el sensor, es importante que éstas estén coordinadas para no saturar la UART. Esto es debido a que como el RTOS realiza una división temporal para ejecutar distintas tareas en paralelo, ocurre que mientras una tarea se comunica con el sensor, pasado dicho tiempo se interrumpe y otra tarea realiza la misma acción, corrompiendo los datos enviados o invalidando las lecturas realizadas.

Para evitar esto, antes de ejecutar código las distintas tareas intentarán tomar un semáforo binario, del cual solo hay uno. Si no se puede tomar el semáforo porque otra tarea está haciendo uso del él, la tarea que ha fallado la toma entra en estado de bloqueo. Para implementar esta funcionalidad empleamos las funciones *xSemaphoreTake*.

```
1 BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

Código 25: Prototipo de la función *xSemaphoreTake*.

Valor de retorno	**Expliación**
BaseType_t	Si se ha podido tomar el semáforo dentro del tiempo de espera establecido, se retorna pdPASS. En caso contrario, pdFALSE
Parámetros	**Explicación**
xSemaphore	Nombre del semáforo que se quiere tomar
xTicksToWait	Cantidad máxima de tiempo que la tarea permanece en estado bloqueado esperando a que se libere el semáforo.

Cuadro 28: Valor de retorno y parámetro de entrada de la función *xSemaphoreTake*.

Cuando el recurso, en este caso la UART, ha terminado de ser usado, es necesario devolver el semáforo. Para esto, empleamos la función *xSemaphoreGive*.

```
1 BaseType_t xSemaphoreGiveFrom(SemaphoreHandle_t xSemaphore);
```

Código 26: Prototipo de la función *xSemaphoreGive*.

Valor de retorno	**Expliación**
BaseType_t	Si se ha podido devolver el semáforo se retorna pdPASS. En caso contrario, pdFALSE
Parámetro	**Explicación**
xSemaphore	Nombre del semáforo que se quiere devolver

Cuadro 29: Valor de retorno y parámetro de entrada de la función *xSemaphoreGive*.

Una vez se ha devuelto el semáforo, la tarea cede la ejecución a alguna otra tarea mediante la macro *taskYIELD*. En la Figura 31 se explica por qué es conveniente realizar esta acción.

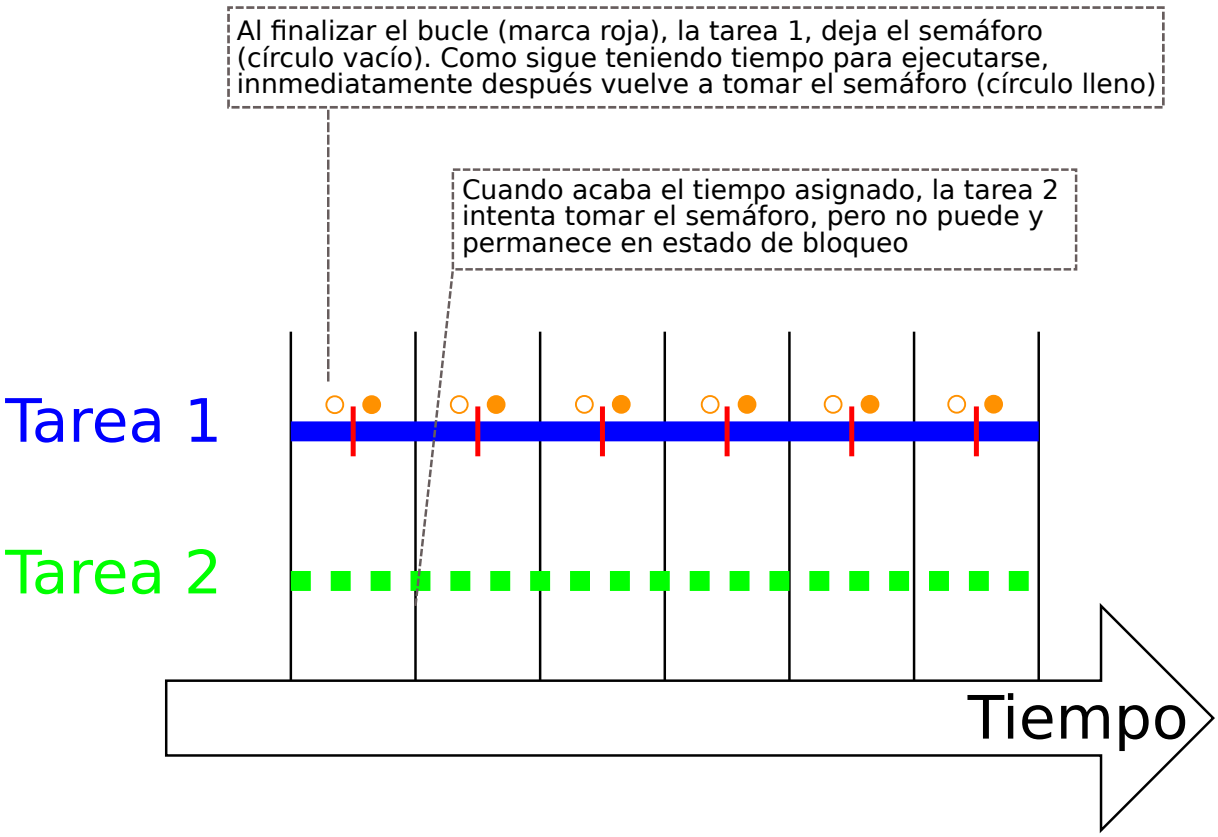

Figura 31: Esquema temporal que muestra el comportamiento de dos tareas en comparten la UART mediante un semáforo sin *taskYIELD*.

Colas de Comunicación FreeRTOS es un entorno sofisticado y todos sus procesos requieren de un protocolo concreto. En el caso de la comunicación entre distintas tareas no basta con, por ejemplo, utilizar variables globales a la hora de intercambiar información. Para este cometido se emplean colas de comunicación.
Una cola puede contener un número finito de cierto tipo de datos, que queda determinado a la hora de crear dicha cola. Éstas se comportan como un búfer FIFO (Primero en Entrar, Primero en Salir,en inlés First In First Out), de manera que el primer dato que se escribe en dicha cola, será el primero en ser enviado, y los datos posteriores quedan acumulados en la cola esperando.

En nuestra implementación tenemos una sola tarea que recoge la información del resto. Como cada lectura que podemos realizar con el sensor tiene un tamaño distinto, implementaremos distintas colas para enviar dichos datos.

- Con *xCola_SPS30_IEEE* enviaremos las medidas realizadas en la tarea *vTaskLecturaIEEE754*.
- Con *xCola_SPS30_U16* enviaremos las medidas realizadas en la tarea *vTaskLecturaUnsigned16*.
- Con *xCola_cleaning_interval* enviamos el tiempo de intervalo entre limpiezas leído del sensor con la tarea *vTaskFunciongenerica*.

Función xQueueCreate El uso de las colas es simple, primero debemos crear una cola utilizando la función *xQueueCreate*.

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

Código 27: Prototipo de la función *xQueueCreate*.

Funcón xQueueSendToBack Cuando queremos guardar un dato en la cola, recurrimos a la función *xQueueSendToBack*.

Valor de retorno	Expliación
QueueHandle_t	Retornará NULL en caso de no poder crear la cola solicitiada. En caso contrario, la cola ha sido creada con éxito.
Parámetros	**Explicación**
uxQueueLength	Numero máximo de datos que puede haber en la cola a la vez
uxItemSize	Tamaño de cada dato que se guarda en la cola

Cuadro 30: Valor de retorno y parámetros de entrada de la función *xQueueCreate*.

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait
    );
```

Código 28: Prototipo de la función *xQueueSendToBack*.

Valor de retorno	Expliación
BaseType_t	Retorna pdPASS si se han podido guardar los datos o errQUEUE_FULL si la cola estaba llena
Parámetros	**Explicación**
xQueue	Nombre de la cola donde queremos escribir
∗pvItemToQueue	Puntero hacia el dato que queremos guardar
xTicksToWait	Tiempo máximo que la tarea permanece en estado de bloqueo a la espera de que la cola quede con espacio disponible en caso de que estuviera llena

Cuadro 31: Valor de retorno y parámetros de entrada de la función *xQueueCrexQueueSendToBackate*.

Función xQueueReceive Nos permite el primer dato almacenado en una cola.

```
BaseType_t xQueueReceive( QueueHandle_t xQueue, void * const pvBuffer, TickType_t xTicksToWait );
```

Código 29: Prototipo de la función *xQueueReceive*.

Valor de retorno	Expliación
BaseType_t	Retorna pdPASS si se han podido guardar los datos o errQUEUE_FULL si la cola estaba llena
Parámetros	**Explicación**
xQueue	Nombre de la cola donde queremos leer
∗pvBuffer	Puntero hacia la posición de memoria donde queremos volcar la lectura
xTicksToWait	Tiempo máximo que la tarea permanece en estado de bloqueo a la espera de que la cola quede con algún dato disponible en caso de que estuviera vacía

Cuadro 32: Valor de retorno y parámetros de entrada de la función *xQueueReceive*.

Envío de datos medidos por puerto serie Ya que nuestro trabajo se limita a tomar medidas, vamos a simular el envío y representación de datos a un servidor por una representación en pantalla. Haciendo uso del comando *Debug_SendMessage* enviaremos los números leídos en formato ASCII. Se puede ver el resultado en la Figura 41 de la Subsección 4.2.

3. Diseño y desarrollo

3.1. Detalle de los bloques definidos en el diagrama FW

En la Sección 2 hemos expuesto las distintas funciones que hemos empleado para crear esta librería y, en el caso de las creadas desde cero, un diagrama de flujo. Ahora vamos a desarrollar los pasos ilustrados en los diagramas, así como la modificación a la función *UART_WriteValue* y el contenido de las tareas de FreeRTOS.

3.1.1. SPS30

secuencia de envío-recepción Cuando nos referimos a esta secuencia, lo primero que se hace es activar la lectura por interrupción de la UART, que se guarda en *ucLectura_byte_stuffing*. De no activar una lectura por interrupción, este comando se vería interrumpido constantemente por el sistema FreeRTOS y se corromperían los datos.

Mientras se espera la llegada de datos, se manda el comando correspondiente, luego se espera un tiempo prudencial de 200 ms para que no se siga ejecutando código antes de obtener la respuesta del sensor.

Tras realizar la lectura, se debe deshabilitar manualmente la UART tal y como se comentó en la función *vDeshabilita_UART* en la Subsección 2.2.3.

De la respuesta leída se verifica el checksum. Si se valida se continúa ejecutando el código correspondiente tras la secuencia de envío recepción, si no se valida se retorna *SPS30_ERR_CHK* y si se diera un caso imprevisto se retorna *SPS30_ERR_UNKNOWN*.

empezar_medida La función comienza declarando de las variables empleadas:

- *ucEmpezar_medida_IEEE754* y *ucEmpezar_medida_unsigned16* son las cadenas de caracteres en hexadecimal correspondientes a los comandos que hacen que el sensor pase de estado de reposo a estado de medida, respectivamente en formato IEEE754 o Unsigned de 16 bits.
- *ucLectura_byte_stuffing* es la cadena donde se van a volcar los datos leídos directamente del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 9.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Su tamaño debe ser 7.

Dependiendo si el parámetro *formato* toma el valor de *IEEE754* o *Unsigned16*, la función realizará una secuencia de envío-recepción utilizando respectivamente la cadena *ucEmpezar_medida_IEEE754* o *ucEmpezar_medida_Unsigned16*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, la función devolverá *SPS30_OK*.

parar_medida La función comienza declarando las variables empleadas:

- *ucParar_medida* es la cadena de caracteres en hexadecimal correspondiente con el comando que hace que el sensor pase de estado de medida a estado de reposo.
- *ucLectura_byte_stuffing* es la cadena donde se van a volcar los datos leídos directamente del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 9.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener tamaño 7.

La función realizará una secuencia de envío-recepción utilizando el comando *ucParar_medida*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, la función devolverá *SPS30_OK*.

leer_medida La función comienza declarando las variables empleadas:

- *ucLeer_medida* es la cadena de caracteres en hexadecimal correspondiente con el comando que hace que el sensor devuelva la última medida que ha realizado.
- *ucLectura_byte_stuffing* es la cadena donde se van a volcar los datos leídos directamente del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 88.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. El tamaño puede ser como máximo 47, cuando la lectura tenga como formato el IEEE754.

Se realiza una secuencia de envío-recepción con el comando *ucLeer_medida*. Si dicha secuencia obtiene una validación del checksum recibido en la respuesta se comprueba el parámetro *formato*.

En caso de que el formato sea *IEEE754* o *Unsigned16*, se hace una llamada respectivamente a la función *separa_datos_IEEE754* o *separa_datos_unsigned16* otorgando la *ucLecura* para que se vuelquen los resultados en la estructura *medida_IEEE754* o *medida_u16*. Por último se retorna *SPS30_OK*.

sleep La función comienza declarando las variables empleadas:

- *ucSleep* es la cadena de caracteres en hexadecimal correspondiente con el comando que hace que el sensor pase a estado de reposo.
- *ucLectura_byte_stuffing* es la cadena donde se va a volcar la respuesta del sensor al comando. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 9.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 7.

La función realizará una secuencia de envío-recepción utilizando el comando *ucSleep*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, la función devolverá *SPS30_OK*.

wake_up La función comienza declarando las variables empleadas:

- *ucWake_up*es la cadena de caracteres en hexadecimal correspondiente con el comando que hacer que el sensor salga del estado de reposo.
- *ucLectura_byte_stuffing* es la cadena donde se va a volcar la respuesta del sensor al comando. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 10.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 7.

La función realizará una secuencia de envío-recepción utilizando el comando *ucWake_up*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, la función devolverá *SPS30_OK*.

limpieza La función comienza declarando las variables empleadas:

- *ucLimpieza*es la cadena de caracteres en hexadecimal que debemos enviar para que el sensor active el ventilador de limpieza.
- *ucLectura_byte_stuffing* es la cadena donde se va a volcar la respuesta del sensor al comando. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 9.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 7.

La función realizará una secuencia de envío-recepción utilizando el comando *ucLimpieza*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, la función devolverá *SPS30_OK*.

set_cleaning_interval La función comienza declarando la variable *ucIntervalo_byte_string*, que es el tiempo deseado separado en una cadena de cuatro celdas de tipo Unsigned de 8 bits para que puedan ser enviados por la UART y procesador por el sensor. Es el resultado de dividir la entrada de 32 bits en cuatro celdas.

Para rellenar el el vector *ucIntervalo_byte_string*, se hace una llamada a la función *longint_to_byte_string*, pasando como parámetro de entrada *ulIntervalo*, el tiempo que hemos especificado.

Con el vector calculado, se declara *ucDatos_checksum*, que contiene todos los bytes que se necesitan para calcular el checksum. Debido a que el valor introducido no está previamente determinado, es necesario cada vez que se ejecute la función.

Se calcula el checksum pasando como parámetro el vecetor *ucDatos_checksum* y el resultado se guarda en la variable *ucChecksum*.

Con todos los bytes necesarios calculados, se construye la cadena ucSet_cleaning_interval, correspondiente al comando antes de procesarlos con byte-stuffing.

Se hace una llamada a la función *byte_stuffing* para procesar el comando anterior y guardarlo en *ucSet_cleaning_inteval_byte_stuffing*, el cual estará listo para ser enviado.

Se declaran las variables restantes:

- *ucLectura_byte_stuffing* es la cadena donde se va a volcar la respuesta del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 17.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 12.

Se realiza una secuencia de envío-recepción utilizando el comando *ucSet_cleaning_inteval_byte_stuffing*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, la función devolverá *SPS30_OK*.

read_cleaning_interval La función comienza declarando las variables empleadas:

- *ucRead_cleaning_interval* es la cadena de caracteres en hexadecimal correspondiente con el comando para el sensor devuelva el valor del intervalo entre limpiezas.
- *ucLectura_byte_stuffing* es la cadena donde se va a volcar la respuesta del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 16.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 11.
- *datos* es un vector de cuatro celdas de tipo Unsigned 8 bits donde se guardará el intervalo de espera separado en estas cuatro celdas.

Se realiza una secuencia de envío-recepción utilizando el comando *ucRead_cleaning_interval*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, se guardan en las 4 celdas de *datos* los 4 bytes correspondientes al intervalo de limpieza (bytes del 6 al 9, en código C del 5 al 8) desde *ucLectura*.

Con los 4 bytes se reconstruye el valor temporal en formato Unsigned de 32 bits aplicando desplazamientos a cada byte y sumando el resultado, que es guardado en la dirección a la que apunta el puntero de entrada $*out\_cleaning\_interval$, finalmente retorna *SPS30_OK*.

sensor_aire_reset La función comienza declarando las variables empleadas:

- *ucReset* es la cadena de caracteres en hexadecimal correspondiente con el comando que provoca el reinicio del sensor.
- *ucLectura_byte_stuffing* es la cadena donde se van a volcar los datos devueltos por el sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 9.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 7.

Se realiza una secuencia de envío-recepción utilizando el comando *ucLectura_byte_stuffing*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, la función devolverá *SPS30_OK*.

get_serial_number La función comienza declarando las variables empleadas:

- *ucGet_serial_number* es la cadena de caracteres en hexadecimal correspondiente con el comando que hace que el sensor devuelva su número de serie.
- *ucLectura_byte_stuffing* es la cadena donde se van a volcar los datos leídos directamente del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 73.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño máximo de 39.

Se realiza una secuencia de envío-recepción utilizando el comando *ucGet_serial_number*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, se toma el número de datos en *ucLectura[4]* y se itera esa cantidad volcando los datos en el vector *serial_number*, que guarda el número de serie. Por último, retorna *SPS30_OK*.

read_version La función comienza declarando las variables empleadas:

- *ucRead_version*es la cadena de caracteres en hexadecimal correspondiente con el comando que hace que el sensor devuelva su versión.
- *ucLectura_byte_stuffing* es la cadena donde se van a volcar los datos leídos directamente del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 22.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 14.

Se realiza una secuencia de envío-recepción utilizando el comando *ucRead_version*.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, guarda las celdas de *ucLectura* que se corresponden con cada versión.

- Versión mayor del firmware: celda 5.
- Versión menor del firmware: celda 6.
- Versión del hardware: celda 8.
- Versión mayor del protocolo SHDLC: celda 10.
- Versión mayor del SHDLC: celda 11.

Por último, retorna *SPS30_OK*.

read_status_register La función comienza declarando las variables empleadas:

- *ucRead_status_register_borrar ucRead_status_register_no_borrar* y son las cadenas de caracteres correspondientes con los comandos que hacen que el sensor devuelva devuelva el registro de estado, borrando o no su contenido, respectivamente
- *ucLectura_byte_stuffing* es la cadena donde se van a volcar los datos leídos directamente del sensor. De acuerdo con los bytes que pueden ser reemplazados por byte-stuffing, el tamaño máximo que esta cadena puede tomar es 18.
- *ucLectura* es la cadena donde se van a volcar los datos decodificados. Debe tener un tamaño de 12.

Si el valor de *formato* es 0 o 1, se realiza una secuencia de envío-recepción utilizando el comando *cRead_status_register_no_borrar* o *cRead_status_register_borrar*, respectivamente.

Si al finalizar la secuencia de envío-recepción el checksum recibido es validado, guarda las celdas de *ucLectura* que se corresponden con cada versión.
se procesará la cadena *ucLectura*, de la siguiente manera:

- En la posición 7 del frame recibido (6 en código C) se encuentran los bits del 16 al 23 del registro de estado. Realizar la operación AND con una máscara de valor 32 (en binario, 00100000), nos da el valor del bit 21. Dependiendo de su valor se activa el elemento *SPEED_WARNING* en la estructura *status_code*.
- En la posición 9 del frame recibido (8 en código C) se encuentran los bits del 1 al 15 del registro de estado. Realizar la operación AND con una máscara de valor 32 (en binario, 00100000), nos da el valor del bit 5. Realizar la operación AND con una máscara de valor 16 (en binario, 00010000), nos da el valor del bit 4. Dependiendo del valor de cada uno se activan los elementos *FAN_ERROR* y *LASER_ERROR* respectivamente en la estructura *status_code*.

Por último, la función devolverá *SPS30_OK*.

Funciones auxiliares A lo largo del código expuesto hasta este punto, hemos empleado funciones que realizan tareas concretas pero cuyo funcionamiento no hemos desarrollado. Ahora realizaremos un repaso de los algoritmos implementados en cada una de ellas.

longint_to_byte_string Un número N codificado en binario, es decir,

$$N_2 = \{b_m, b_{m-1}, ..., b_1, b_0\}$$

se puede convertir a decimal de la siguiente manera

$$N_{10} = \sum_{i=0}^{i=m} b_i \cdot 2^i$$

si lo dividimos por una potencia de 2

$$\frac{N_{10}}{2^p} = \frac{= \sum_{i=0}^{i=m} b_i \cdot 2^i}{2^p} = \sum_{i=0}^{i=m} b_i \cdot 2^{i-p}$$

la operación es equivalente a desplazar todos los bits a la derecha un número de veces igual al exponente de dicha potencia (p). En este caso, tomamos el valor *div* y lo dividimos entre 256, es decir, 2^8. De esta manera, el conjunto $\{b_7, ..., b_0\}$, que representa el byte bajo del número N, pasa a tener un exponente negativo, es decir, no es entero y forma el resto de la división. De esta manera hemos aislado un byte del número N como resto de una división.
Dicho resto es el valor que guardaremos en una de las celdas a las que apunta el puntero *puntero_vector_salida*. Posteriormente nos quedamos solo con el cociente de la división anterior para repetir el procedimiento hasta obtener un número menor a 255 (que quepa en un byte).
Ejemplo: separar el número 0xA1B2C3D4 en cuatro bytes.

$$0xA1B2C3D4/256 = 0xA1B2C3, 0xA1B2C3D4\,\%256 = \mathbf{0xD4}$$

$$0xA1B2C3/256 = 0xA1B2, 0xA1B2C3\,\%256 = \mathbf{0xC3}$$

$$0xA1B2/256 = 0xA1, 0xA1B2\,\%256 = \mathbf{0xB2}$$

$$\mathbf{0xA1} < 255$$

verify_checksum Los bytes del frame que contribuyen a calcular el checksum son todos excepto los bytes de start y stop, es decir, todos menos el primero y el último. Hay que tener en mente no utilizar el propio byte de checksum para realizar la comprobación.

Con un bucle while suman todos los bytes del frame desde el byte número 2 hasta el penúltimo (que es cuando se lee un 0x7E). A la suma anterior debemos restar el último valor, ya que se trata del propio checksum recibido, que además se almacena en *reveiced_checksum*.

La suma final es invertida y se le aplica una máscara para tomar solo el byte más bajo, es decir, se realiza la operación AND con el valor 0xFF (en binario, 11111111). Este valor es el checksum calculado y se guarda en *calculated_checksum*.

Por último se comparan los valores *receiced_checksum* y *calculated checksum*. Si coinciden, la función retorna un 0, en caso contrario, retorna un 1.

calculate_checksum El procedimiento de esta función es similar al de *verify_checksum*.

Se suman los bytes correspondientes a *ADR*, *CMD* y *L*. Se toma el valor de la longitud de los datos, *L*, y con un bucle se suman el resto de datos contenidos en esa longitud. Se invierte la suma y se aplica la máscara 0xFF (en binario, 11111111) para quedarnos con el byte más bajo, que será el checksum. Por último, la función retorna dicho valor.

byte_stuffing Tenemos dos variables para recorrer los vectores:

- i para el vector original.
- j para el vector creado.

Se realiza un bucle entre los bits de start y stop, es decir, se recorren las celdas del vector original desde la 2 a la penúltima (en código C, desde i=1 hasta i=ucSize-1).

Se comprueba si la celda *ucVector_original[i]* contiene uno de los cuatro bytes sujetos a cambio. Si fuera el caso se aumenta el contador llamado *aumento* y se guardan en las dos celdas siguientes donde apunta el puntero *pucCadena_bit_stuffing* los bytes descritos en el Cuadro 20.

El tamaño final de $*pucCadena\_bit\_stuffing$ será el tamaño del original *ucSize* más el valor del contador *aumento*.

Se guarda el byte 0x7E al principio y al final de $*pucCadena\_bit\_stuffing$ y se retorna *size_final*.
En la Figura 32 se pone un ejemplo de los pasos descritos.

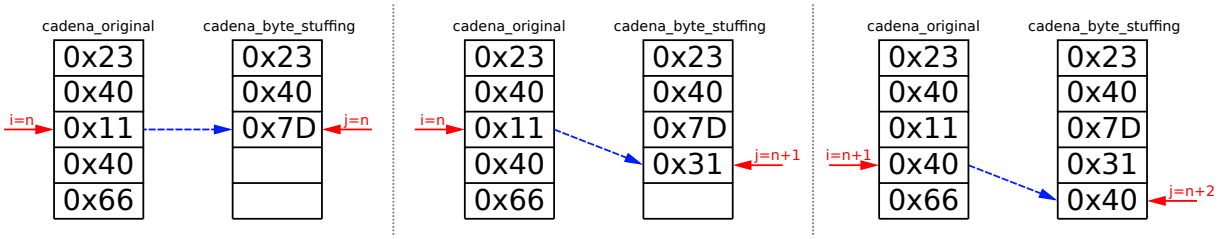

Figura 32: Ejemplo de la operación de la función *byte_stuffing*.

inverse_byte_stuffing Tenemos dos variables para recorrer los vectores:

- i para el vector recibido.
- j para el vector devuelto.

Se comienza estableciendo el bit de start 0x7E en la posición 0 del vector devuelto. Para deshacer el byte-stuffing se recorre el vector recibido hasta volver a encontrar el byte 0x7E, que solo aparece para marcar el stop. Se añade la condición de que el índice i sea menor a 99 para evitar un atasco en el bucle en caso de que los datos estén corruptos y no se encuentre un 0x7E.

Se leen dos posiciones consecutivas del vector *ucCadena_original*, si sendas posiciones coinciden con los valores descritos en el Cuadro 20, se escribe el byte correspondiente en *ucCadena_inverse_byte_stuffing[j]* y se avanza el índice i para no volver a leer el segundo valor correspondiente al byte-stuffing. Finalmente, se guarda el byte de stop y se devuelve el tamaño del vector *ucCadena_inverse_byte_stuffing*, que coincidirá con *j+1*.

En la Figura 32 se muestra un ejemplo de los pasos descritos.

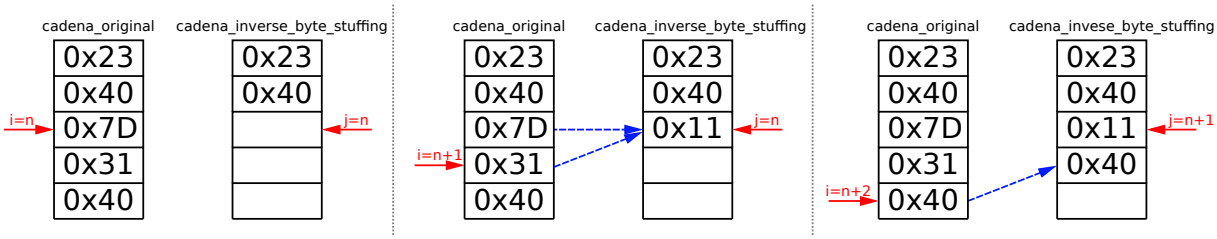

Figura 33: Ejemplo de la operación de la función *byte_stuffing*.

separa_datos_IEEE754 El frame contiene diez datos diferentes, los cuales son:

- Concentración de masa PM1.0 [$\mu g/m^3$].
- Concentración de masa PM2.5 [$\mu g/m^3$].
- Concentración de masa PM4.0 [$\mu g/m^3$].
- Concentración de masa PM10 [$\mu g/m^3$].
- Concentración numérica PM0.5 [$\#/cm^3$].
- Concentración numérica PM1.0 [$\#/cm^3$].
- Concentración numérica PM5.5 [$\#/cm^3$].
- Concentración numérica PM0.4 [$\#/cm^3$].
- Concentración numérica PM10 [$\#/cm^3$].
- Tamaño típico de partícula [μm].

Dentro del frame, como deben formar un valor en formato IEEE754, correspondiente con un float, están agrupados en grupos de 4 bytes, según el orden descrito.

La función realiza la misma operación para las diez medidas diferentes, lo que en el diagrama de flujo se ha llamado *separa(X,Y)*. Vamos a ejemplificarlo con la medida de Concentración de masa PM1.0 en la Línea 3.

El formato, además de float, es big-endian, esto significa que los bytes de mayor valor son los que se transmiten primero. Como la Concentración de masa PM1.0 es la primera medida otorgada, los bytes que conforman la medida van de la celda 0 a la 3 del vector *ucLectura*. Para mover el dato de la celda en formato Unsigned int de 8 bits a una posición alta de una variable de 32 bits, realizamos un desplazamiento de 24 bits, es decir, 3 bytes hacia la izquierda. Es necesario realizar primero una conversión a una variable de tipo Unsigned de 32 bits, de otra manera tendríamos un desbordamiento al desplazar varios bytes una variable de solamente un byte. De la misma manera, el dato en la celda 1 lo desplazamos 16 bits (2 bytes) a la izquierda, el dato en la celda 2 lo desplazamos 8 bits (1 byte) a la izquierda, y el dato en la celda 3 no lo desplazamos porque conforma el byte más bajo de la medida. Estos cuatro valores los

sumamos, agrupando en una sola variable de 32 bits (*ulInt_MC_PM1*) las cuatro celdas de 8 bits.

Queda un detalle, el valor de los bytes que hemos agrupado está en formato Unsigned 32 bits porque hemos requerido sumar las celdas tomando su valor como Unsigned. Sin embargo, este grupo de 4 bytes debe ser interpretado de ahora en adelante como un float. No queremos modificar el contenido de la variable, solamente la manera en la que es interpretada.

Para esto debemos tomar el puntero de la dirección de memoria de la variable *ulInt_MC_PM1* con el operador & y realizar una conversión a puntero de flotante con *(float*)*. Ahora que tenemos este nuevo puntero a float, accedemos a su contenido interpretándolo como flotante con el operador *, y lo guardamos en la dirección que indica el puntero al elemento correspondiente en la estructura medida, en este caso, *(*medida).Mass_Concentration_PM1*.

separa_datos_unsigned16 El frame contiene diez datos diferentes, son los mismos que en el Cuadro 2, a diferencia de las unidades de:

- Tamaño típico de partícula [*nm*]

Dentro del frame, como deben formar un valor en formato Unsigned de 16 bits, están agrupados en grupos de 2 bytes, según el orden descrito en la Subsección 2.2.2.

La función realiza la misma operación para las diez medidas diferentes. Vamos a ejemplificarlo con la medida de Concentración de masa PM1.0 en la Línea 3.

El formato, además de Unsigned de 16 bits, es big-endian, esto significa que los bytes de mayor valor son los que se transmiten primero. Como la Concentración de masa PM1.0 es la primera medida otorgada, los bytes que conforman la medida van de la celda 0 a la 1 del vector *ucLectura*. Para mover el dato de la celda en formato Unsigned int de 8 bits a una posición alta de una variable de 16 bits, realizamos un desplazamiento de 8 bits, es decir, un byte hacia la izquierda. Es necesario realizar primero una conversión a una variable de tipo Unsigned de 16 bits, de otra manera tendríamos un desbordamiento al desplazar varios bytes una variable de solamente un byte. El dato en la celda 1 no lo desplazamos porque conforma el byte más bajo de la medida. Estos dos valores los sumamos, agrupando en una sola variable de 16 bits los dos valores de 8 bits, que vamos a guardar en el elemento correspondiente de la estructura **medida*.

vDeshabilita_UART Esta secuencia de código ha sido tomada de la librería del GIE, donde se activa la secuencia una vez se han leído todos los datos esperados, pero como hemos comentando, en nuestro caso es necesario activar dicha secuencia en cualquier momento. La secuencia de comandos genera lo siguiente:

- Desactivación de la interrupción que marca que el registro de datos de la UART no está vacío
- Desactivación de la interrupción que marca que error de paridad en la UART
- Desactivación de la interrupción que marca que error en la interrupción a causa de un frame corrupto o ruido.
- Restablecimiento del estado de la UART a *ready*, lo que permite volver a llamar a la función *UART_ReadValue_IT*.
- Llamada a los callbacks de la recepción de datos

3.1.2. UART

El problema de UART_WriteValue En esta función, se itera en la cadena de entrada **str* hasta encontrar un terminador de cadena \0. En cada iteración se añade el carácter encontrado al búfer de transmisión. Si el carácter leído resulta ser %, se busca que el siguiente carácter sea 's', 'd', 'c' o 'x' para aplicar un formato concreto al dato que se especifique en la elipsis, leído con el comando *va_arg(arg,int)*. Por ejemplo, si el caracter leído en **str* es '%d', se tomará el valor a transmitir de la elipsis y se escribirá el valor ASCII del int especificado.

El problema reside en que lo único que se está haciendo, en el caso de especificar un número hexadecimal (*p='x'), es añadir al búfer los caracteres '0' y 'x', además de convertir el número de entrada en su carácter ASCII correspondiente. Es decir, mientras que en nuestra aplicación queremos preservar el valor numérico hexadecimal de entrada, aquí se envían los caracteres correspondientes. En las Figuras 34 y 35 se puede ver el resultado de hacer una escritura mediante el Código 30.

```
1 //indicar que se mandan 8 valores en hexadecimal
2 char stringwrite[]=" %x %x %x %x %x %x %x %x ";
3 //cadena escrita en decimal: 0x7E,0x0,0x0,0x2,0x1,0x3,0xF9,0x7E
4 UART_WriteValue(UART_3,&stringwrite[0],126,0,0,2,1,3,249,126);
```

Código 30: Envío con *UART_WriteValue*.

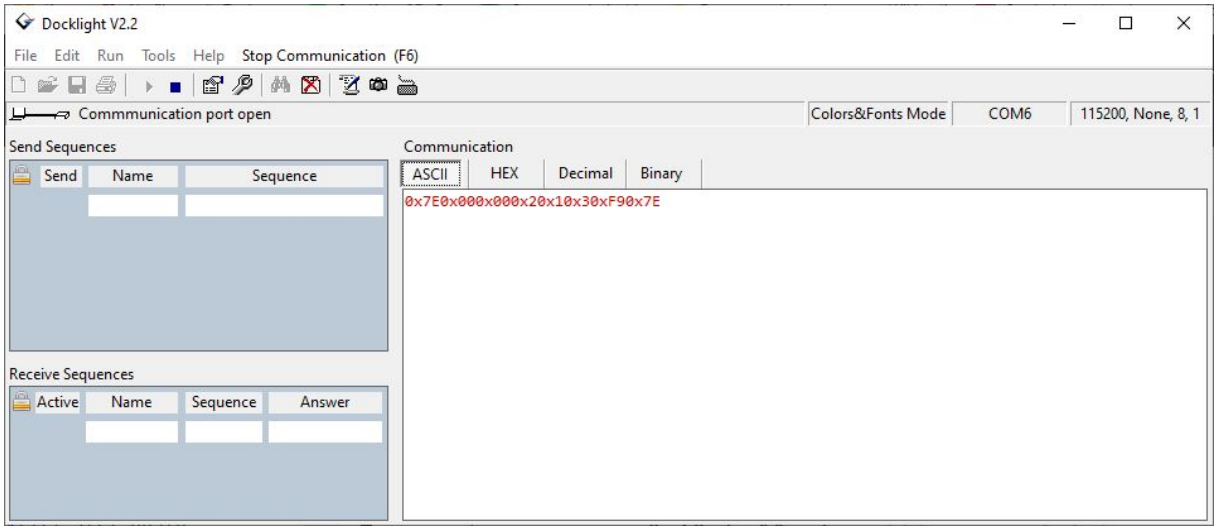

Figura 34: Resultado en ASCII obtenido con el Código 30.

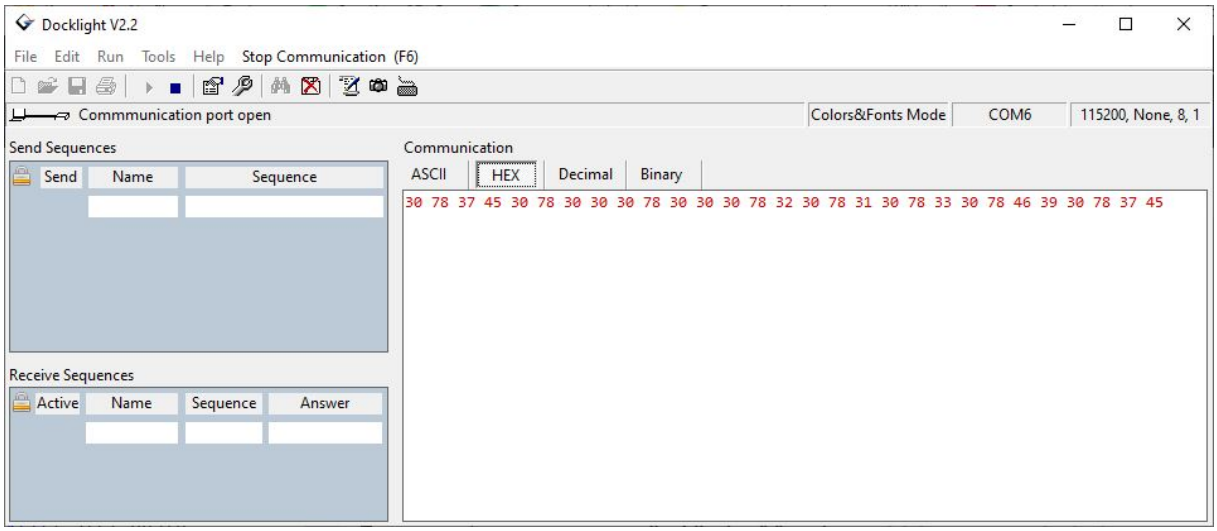

Figura 35: Resultado en hexadecimal obtenido con el Código 30.

Modificación de WriteValue_V2 Dado que nos interesa exclusivamente leer caracteres en hexadecimal, lo que hacemos directamente es concatenar dichos valores contenidos en **csrt* al búfer sin realizar ninguna conversión a ASCII.

3.1.3. FreeRTOS

Las tareas en FreeRTOS se ejecutan de forma paralela. Aunque la creación de estas dentro del código viene limitada por las condiciones en las que se desea que trabaje el sensor y no forman parte de este trabajo se proponen algunas que pueden ser de utilidad.

No se van a mostrar prototipos o parámetros de entrada/salida ya que no se tratan de funciones al uso. Si se desean usar solamente hay que iniciarlas externamente haciendo referencia a su nombre.

vTaskLecturaIEEE754 Esta tarea solicita que el sensor comience a leer en formato IEEE754, pide una medida y solicita la parada del sensor. Está pensada para ser ejecutada en paralelo con otros comandos, principalmente el de medida en formato Unsigned de 16 bits, ya que estos dos son los únicos que se prestan a usarse de manera cíclica.
La tarea intenta el semáforo de la UART, si lo consigue, procede a ejecutar *empezar_medida(IEEE754)*, al acabar deja 500 ms para asegurar que el puerto de debug ha terminado de enviar el resultado y reinicia las UART para asegurar que no se quedan colgadas. Repite con *lee_medida* y *parar_medida*.

Los datos leídos los envía por la cola *xCola_SPS30_IEEE*, devuelve el semáforo y cede la ejecución a otra función.

vTaskLecturaUnsigned16 Esta tarea es idéntica a la anterior. Con la diferencia del parámetro *formato* de los comandos *empezar_medida* y *lee_medida*. Para enviar los datos emplea la cola *xCola_SPS30_U16*.

vTaskFunciongenerica Esta tarea implementa las funciones *set_cleaning_interval* y *read_cleaning_interval*. La susamos para mostrar cómo se podría implementar una función que no se busca ejecutar periódicamente en el entorno de FreeRTOS.

En este caso, como las dos funciones son independientes, se ejecutan en tiempos distintos tomando cada una el semáforo y dejándolo, en lugar de ejecutarlas de una sola vez.

Para especificar a la tarea el tiempo te intervalo convertimos el valor deseado a un puntero vacío con *(void*)*. Dentro de la tarea deshacemos el cambio con *(uint32_t)pvParameters*. Por otra parte, el interalo leído del sensor lo guardamos en la cola *xCola_cleaning_interval*.

Como solo nos interesa ejecutarla una vez, borramos la tarea con *vTaskDelete*.

vTaskRecogeDatos Esta tarea intenta leer los datos de las tres colas mencionadas anteriormente para mostrar su resultado por el puerto de debug. En el caso de *xCola_cleaning_interval* solo la utilizará una vez, ya que la tarea *vTaskFunciongenerica* se borra automáticamente al completar un bucle.

En la Figura 36 se muestra un diagrama con las cuatro tareas y la forma en la que están relacionadas. Vemos que tres de ellas hacen uso de la UART, por lo que necesitan compartir el semáforo *xMutex*, y envían datos a sus respectivas colas, que son recogidas por *vTaskRecogeDatos*.

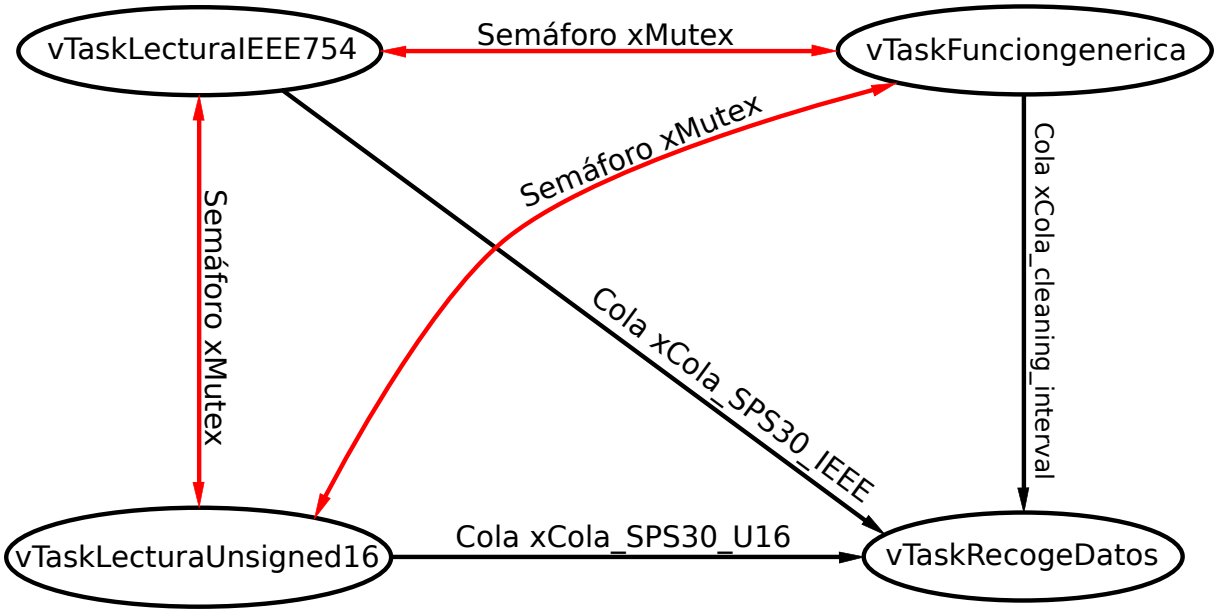

Figura 36: Diagrama con las relaciones entre las tareas de FreeRTOS

4. Pruebas de validación

4.1. Drivers

Una vez tenemos nuestro driver listo, es hora de hacer un test para corroborar que todas las funciones implementadas toman datos del sensor correctamente.

En este test se hace una llamada a cada una de las funciones implementadas correspondientes con un comando del sensor. Cunando se recibe una respuesta del sensor a cada comando y se verifica que el checksum de la respuesta leída es correcto, el test del comando se da por validado y se imprime por el puerto de debug un mensaje del tipo *COMANDO_OK*. Los resultados se muestran en las Figuras 37, 38, 39 y 40. La secuencia de comandos enviados es:

1. Empezar a medir en formato IEEE754
2. Leer en formato IEEE754
3. Parar de medir
4. Empezar a medir en formato Unsigned de 16 bits
5. Leer en formato Unsigned de 16 bits
6. Parar de medir
7. Entrar en modo hibernación
8. Salir del modo hibernación
9. Activar la limpieza con ventilador
10. Establecer el tiempo de intervalo entre limpiezas
11. Leer el tiempo de intervalo entre limpiezas
12. Leer el número de serie
13. Leer la versión del sensor
14. Leer el registro de estado
15. Reiniciar el sensor

Watch 1

Expression	Value	Location	Type
resultado1	<struct>	0x20000160	xDatos_ll
Mass_Concentration_...	4.03145122	0x20000160	float
Mass_Concentration_...	8.48295784	0x20000164	float
Mass_Concentration_...	1.1906158444E+1	0x20000168	float
Mass_Concentration_...	1.3709343911E+1	0x2000016C	float
Number_Concentratio...	1.66096553E+1	0x20000170	float
Number_Concentratio...	2.7118881222E+1	0x20000174	float
Number_Concentratio...	3.16716613E+1	0x20000178	float
Number_Concentratio...	3.2518119811E+1	0x2000017C	float
Number_Concentratio...	3.2727455133E+1	0x20000180	float
Typical_Particle_Size	9.9965262411E-1	0x20000184	float
resultado2	<struct>	0x20000188	xDatos_u
Mass_Concentration_...	3	0x20000188	uint32_t
Mass_Concentration_...	3	0x2000018C	uint32_t
Mass_Concentration_...	3	0x20000190	uint32_t
Mass_Concentration_...	3	0x20000194	uint32_t
Number_Concentratio...	19	0x20000198	uint32_t
Number_Concentratio...	23	0x2000019C	uint32_t
Number_Concentratio...	23	0x200001A0	uint32_t
Number_Concentratio...	23	0x200001A4	uint32_t
Number_Concentratio...	23	0x200001A8	uint32_t
Typical_Particle_Size	430	0x200001AC	uint32_t
serial_number	<array>"A3EDCA5A9FDA1BD4"	0x200000CC	char[32]
version_leida	<struct>	0x200001B0	version
firmware_major_version	'.' (0x02)	0x200001B0	uint8_t
firmware_minor_version	'.' (0x02)	0x200001B1	uint8_t
hardware_version	'\a' (0x07)	0x200001B2	uint8_t
SHDLC_protocol_maj...	'.' (0x02)	0x200001B3	uint8_t
SHDLC_protocol_min...	'\0' (0x00)	0x200001B4	uint8_t
out_cleaning_interval	123456	0x200001B8	unsigned
status_code	<struct>	0x2000015C	SPS30_S
FAN_ERROR	'\0' (0x00)	0x2000015C	uint8_t
LASER_ERROR	'\0' (0x00)	0x2000015D	uint8_t
SPEED_WARNING	'\0' (0x00)	0x2000015E	uint8_t

Figura 37: Valores leídos del sensor con el test del SPS30.

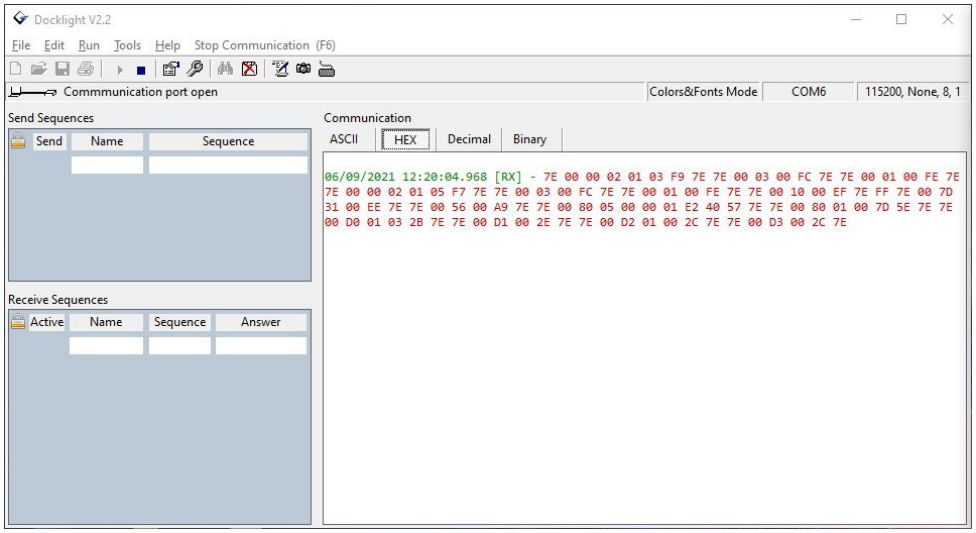

Figura 38: Valores enviados por el puerto TX de la placa Nucleo.

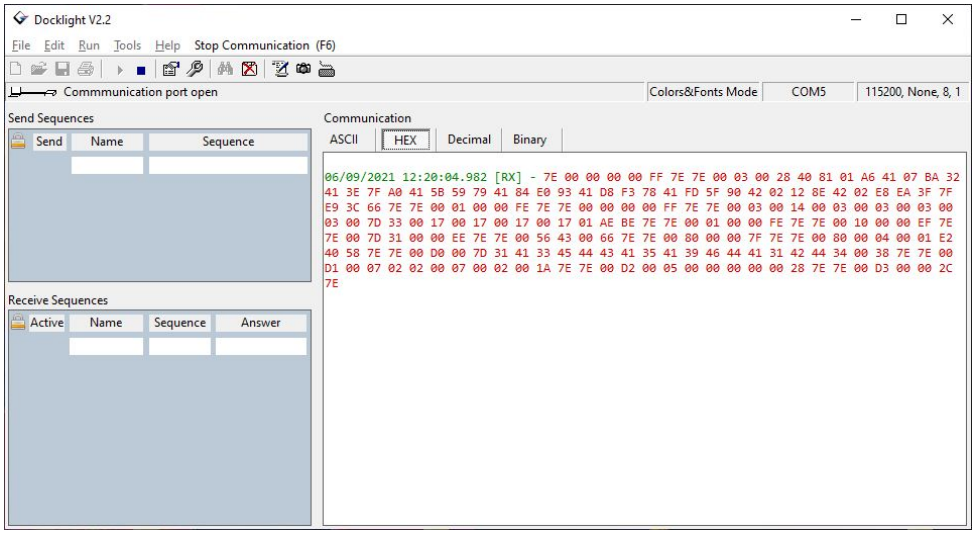

Figura 39: Valores recibidos por el puerto RX de la placa Nucleo.

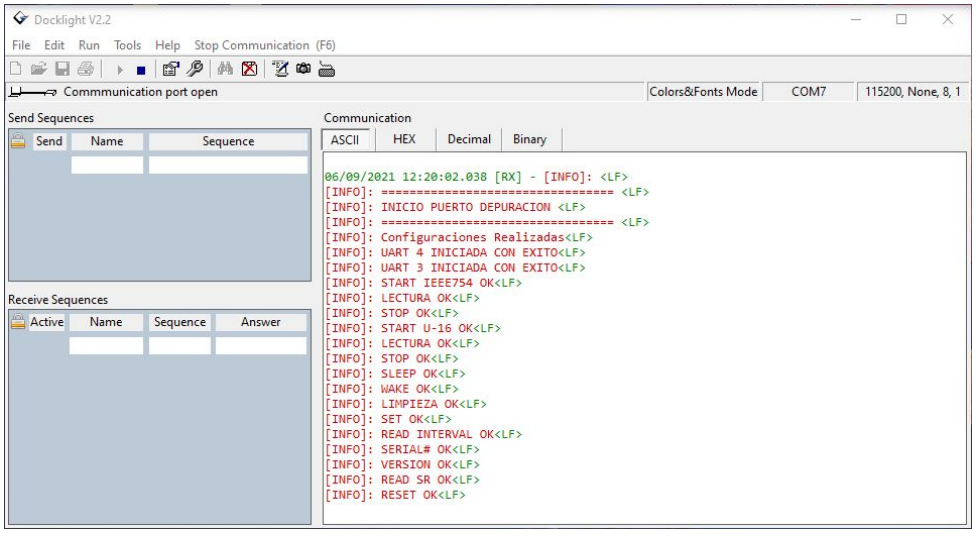

Figura 40: Valores leídos desde el puerto TX de debug de la placa Núcleo.

En la Figura 37 vemos cómo la secuencia de comandos se ha ejecutado correctamente. Para asegurarnos de que la interpretación de los datos recibidos es correcta, vamos a analizar los mensajes monitorizados que toman datos del sensor.

Envío	**Recepción**
7E 00 03 00 FC 7E después de 7E 00 00 02 01 03 F9 7E(lectura en formato IEEE754)	7E 00 03 00 28 40 81 01 A6 41 07 BA 32 41 3E 7F A0 41 5B 59 79 41 84 E0 93 41 D8 F3 78 41 FD 5F 90 42 02 12 8E 42 02 E8 EA 3F 7F E9 3C 66 7E
Datos separados en 4 bytes	**Equivalente en float**
40 81 01 A6 (Concentración de masa PM1)	4.031451
41 07 BA 32 (Concentración de masa PM2.5)	8.482958
41 3E 7F A0 (Concentración de masa PM4)	11.9061584
41 5B 59 79 (Concentración de masa PM10)	13.7093439
41 84 E0 93 (Concentración numérica PM0.5)	16.6096554
41 D8 F3 78 (Concentración numérica PM1)	27.1188812
41 FD 5F 90 (Concentración numérica PM2.5)	31.6716614
42 02 12 8E (Concentración numérica PM4)	32.51812
42 02 E8 EA (Concentración numérica PM10)	32.7274551
3F 7F E9 3C (Tamaño típico de partícula)	0.9996526

Cuadro 33: Interpretación de los datos leídos en la lectura con formato IEEE754.

4.2. FreeRTOS

El test que hemos realizado para comprobar el funcionamiento de FreeRTOS es el siguiente:

- Se crean las tareas *vTaskLecturaIEEE754*, *vTaskLecturaUnsigned16*, *vTaskRecogeDatos* y *vTaskFunciongenerica*.
- La última de las mencionadas se ejecutará una vez y se borrará.
- El resto permanecerán funcionando en bucle.

La tarea *vTaskFunciongenerica* supone una muestra del proceso de uso de las funciones que no forman parte de la medida dentro de FreeRTOS. No están pensadas para ser ejecutadas en bucle, sino para realizar una configuración inicial. En un caso real, se habilitaría la creación de estas tareas dada una orden externa por medio, por ejemplo, de una interrupción. Para hacer esta tarea más ilustrativa, se ha elegido la función *set_cleaning_interval* con 3456 segundos, ya que requiere de un parámetro externo, para así trabajar con el paso de parámetros específico de FreeRTOS.

Las tareas *vTaskLecturaIEEE754* y *vTaskLecturaUnsigned16* se ejecutan en paralelo para mostrar la capacidad del sistema de alternar entre distintas tareas, dado que en circunstancias normales se elegiría tan solo un formato de medida.

En el Código 31 se muestra la secuencia a seguir para iniciar este test
.

```
1 //creacion del semaforo para utilizar la UART
2 xMutex = xSemaphoreCreateMutex();
3 //creacion de la cola para transmitir los datos leidos
4 xCola_SPS30_IEEE = xQueueCreate(1 , sizeof(xDatos_IEEE754));
5 //creacion de la cola para transmitir los datos leidos
6 xCola_SPS30_U16 = xQueueCreate(1 , sizeof(xDatos_u16));
7 //creacion de la cola para transmitir los datos leidos
```

Envío	**Recepción**
7E 00 03 00 FC 7E después de 7E 00 00 02 01 05 F9 7E(lectura en formato Unsigned 16)	7E 00 03 00 14 00 03 00 03 00 03 00 03 00 7D 33 00 17 00 17 00 17 00 17 01 AE BE 7E
Datos separados en 2 bytes	**Equivalente en Unsgined 16**
00 03 (Concentración de masa PM1)	3
00 03 (Concentración de masa PM2.5)	3
00 03 (Concentración de masa PM4)	3
00 03 (Concentración de masa PM10)	3
00 13 (Concentración numérica PM0.5)	19
00 17 (Concentración numérica PM1)	23
00 17 (Concentración numérica PM2.5)	23
00 17 (Concentración numérica PM4)	23
00 17 (Concentración numérica PM10)	23
01 AE (Tamaño típico de partícula)	430

Cuadro 34: Interpretación de los datos leídos en la lectura con formato Unsigned de 16 bits.

Envío	**Recepción**
7E 00 80 01 00 7D 5E 7E (lectura del intervalo entre limpiezas)	7E 00 80 00 04 00 01 E2 40 58 7E
Valor en hexadecimal	**Valor decimal**
00 01 E2 40	123456

Cuadro 35: Interpretación del intervalo temporal entre limpiezas.

```
8  xCola_cleaning_interval=xQueueCreate(1 , sizeof(uint32_t));
9  //creacion de la tarea de lectura en formato IEEE754
10 xTaskCreate( vTaskLecturaIEEE754,"LEE_IEEE754",1000,NULL,6,NULL );
11 //creacion de la tarea de lectura en formato Unsigned-16
12 xTaskCreate( vTaskLecturaUnsigned16,"LEE_U16",1000,NULL,6,NULL );
13 //creacion de la tarea que toma los datos leidos
14 xTaskCreate( vTaskRecogeDatos,"RECOGE",1000,NULL,6,NULL );
15 //creacion de la tarea que toma los datos leidos
16 xTaskCreate( vTaskFunciongenerica,"FUNCION",1000,(void*)3456,6,NULL );
17 //inicio del planificador
18 vTaskStartScheduler();
```

Código 31: Secuencia para comenzar el test de validación de FreeRTOS.

En la Figura 41 podemos ver el resultado satisfactorio de esta operación, donde se van intercalando cada medida y, tras un cierto tiempo, se ejecuta la tarea *vTaskFunciongenerica* y muestra su resultado.

Envío	**Recepción**
7E 00 D0 01 03 2B 7E (lectura del número de serie)	7E 00 D0 00 7D 31 41 33 45 44 43 41 35 41 39 46 44 41 31 42 44 34 00 38 7E
Caracteres en hexadecimal	**Equivalente ASCII**
41 33 45 44 43 41 35 41 39 46 44 41 31 42 44 34 00	A3EDCA5A9FDA1BD4

Cuadro 36: Interpretación del número de serie.

Envío	**Recepción**
7E 00 D1 00 2E 7E (lectura de la versión)	7E 00 D1 00 07 02 02 00 07 00 02 00 1A 7E
Datos separados	**Valor decimal**
02 (Firmware major version)	2
02 (Firmware minor version)	2
07 (Hardware version)	7
02 (SHDLC major version)	2
00 (SHDLC minor version)	0

Cuadro 37: Interpretación de la versión.

Envío	**Recepción**
7E 00 D2 01 00 2C 7E (lectura de la versión)	7E 00 D2 00 05 00 00 00 00 00 28 7E
Registro de estado	**Bits revisados**
00 00 00 00 00	Bit 4 = 0, Bit 5= 0, Bit 21 = 0

Cuadro 38: Interpretación del registro de estado.

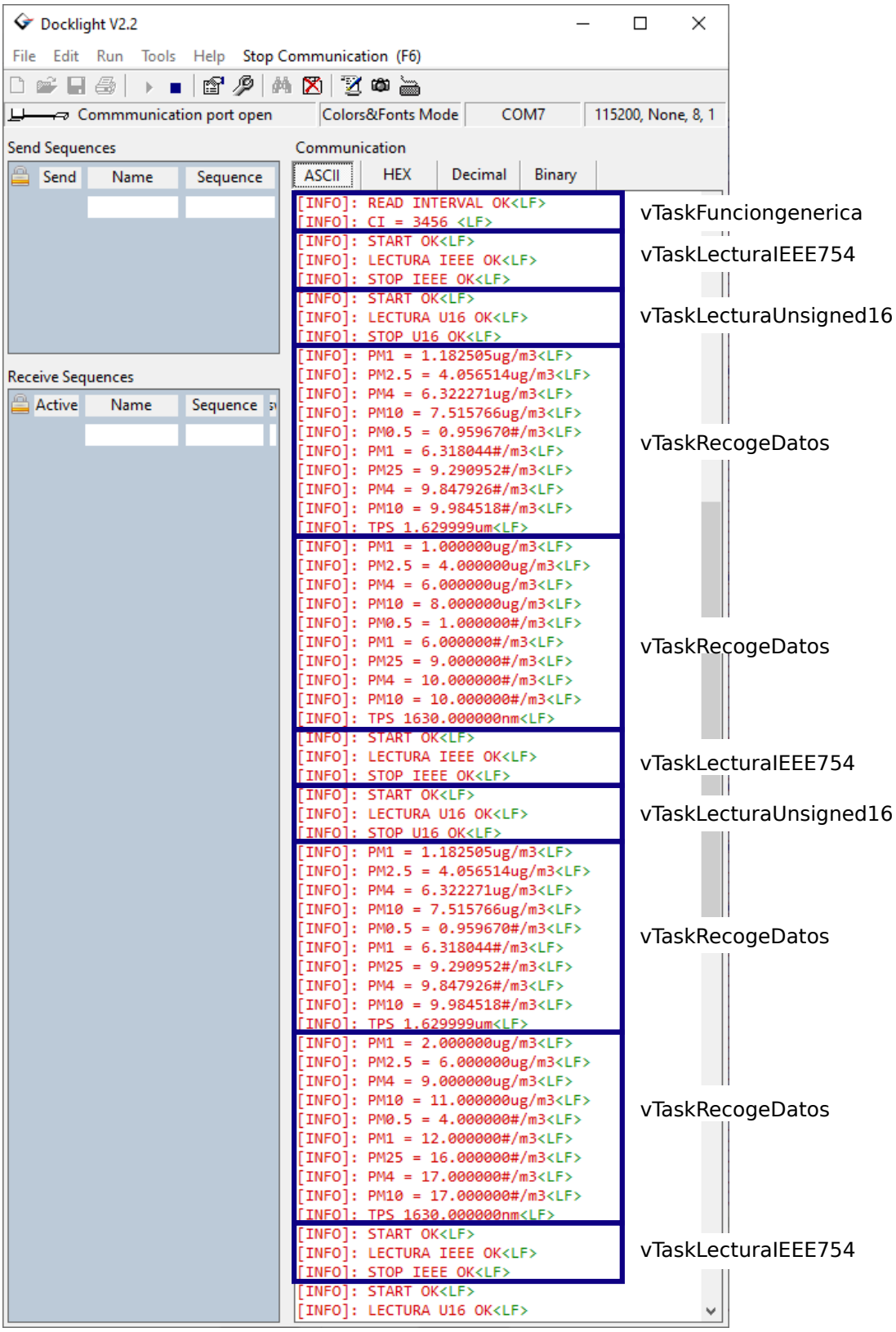

Figura 41: Resultado del test de validación de FreeRTOS.

5. Conclusiones y líneas futuras

Comenzando por las aportaciones que ha supuesto este trabajo, me ha servido para tener una toma de contacto con la metodología de trabajo que implica un proyecto de este tipo, donde las tareas que se van realizando en distintos trabajos están ligadas entre sí. Esto queda reflejado en que he contado con librerías ya hechas, con el objetivo de agilizar el trabajo a cualquier integrante que necesite usar un módulo. De la misma manera, la librería que he elaborado no busca implementar una funcionalidad concreta con unos pasos predefinidos, sino que será utilizada de una manera u otra según las especificaciones concretas que se tengan en su implementación real. A nivel técnico, en este trabajo he tenido la oportunidad de desarrollar mis conocimientos en el lenguaje C, especialmente a la hora de analizar algoritmos para entender su funcionamiento y realizar modificaciones o correcciones, además de entender cómo se estructuran proyectos de mayor dimensión.

Volviendo a la implementación futura, esta propia memoria servirá como manual de uso en caso de necesitar modificar o agregar utilidades a las funciones implementadas. Por otra parte, las tareas en FreeRTOS que se han creado son circunstanciales a este trabajo y su verificación, ya que la jerarquía entre tareas y las funciones que ejecuta cada una también dependerá de la aplicación real en la que se trabaje.

A modo de ejemplo de la modularidad del proyecto global y su futuro uso, en otro trabajo se ha desarrollado una librería para módulos de comunicación inalámbrica enfocados al IoT (Internet de las Cosas, en inglés Internet of Things). Dicha librería también estará implementada en FreeRTOS y para tomar datos usará consecuentemente una cola de comunicación. De la misma manera que hemos empleado una cola para recoger datos con una función específica, en la implementación real dicha función será la encargada de enviar datos a un servidor remoto, facilitando la monitorización y el análisis de datos.

Respecto a los resultados obtenidos, tal y como queda reflejado en la Sección 4, hemos conseguido implementar con éxito la librería para nuestro sensor, confirmando que, no solo se realizan medidas, sino que éstas se analizan correctamente dentro de nuestro código.

En resumen a lo conseguido con esta trabajo, el sensor SPS30 ofrece utilidad dentro de muchos sectores distintos, lo que permitirá poder trabajar con la librería creada en mediciones de la calidad del aire para electrodomésticos, niveles de polución, partículas tóxicas en minas y excavaciones en general, fábricas, ubicaciones para demoler, industrias petroquímicas, deshechos de la agricultura y la quema de material, entre otros.

Referencias

[1] United States Enviromental Protection Agency, "Health and Environmental Effects of Particulate Matter (PM)", https://www.epa.gov/, 26 May., 2021. [Online]. Disponible en: https://www.epa.gov/pm-pollution/health-and-environmental-effects-particulate-matter-pm. [Acceso el 1 Sep. 2021].

[2] United States Enviromental Protection Agency, "Health and Environmental Effects of Particulate Matter (PM)", https://www.epa.gov/, 26 may., 2021. [Online]. Disponible en: https://www.epa.gov/sites/default/files/styles/large/public/2016-09/pm2.5_scale_graphic-color_2.jpg?VersionId=h3EHSFq62Q1Gy93A8B.
3x3x9Ewm8M1x.&itok=kGST2raa. [Acceso el 1 Sep. 2021].

[3] United States Enviromental Protection Agency, "Overview of Particle Air Pollution (PM2.5 and PM10)", https://www.epa.gov/, 17 abr., 2012. [Online]. Disponible en: https://www.epa.gov/sites/default/files/2014-05/documents/huff-particle.pdf. [Acceso el 1 sep. 2021].

[4] Aire limpio, "¿Qué son los bioaerosoles?", https://www.airelimpio.com/, 19 dic., 2018 . [Online]. Disponible en https://www.airelimpio.com/faq-items/que-son-los-bioaerosoles/. [Acceso el 1 sep. 2021].

[5] Kernighan, B. y Ritchie, D., 2011. *The C programming language.* Englewood Cliffs, N.J.: Prentice-Hall.

[6] Sensirion, “ Particulate Matter Sensor for Air Quality Monitoring and Control” SPS30 Datasheet , Mar. 2020.

6. ANEXO. Códigos de las funciones

En esta sección se muestra el código correspondiente a las estructuras y tipos definidos, así como los prototipos del conjunto de tareas y funciones de este trabajo.

```
#ifndef GIE_AIRE_DRIVER
#define GIE_AIRE_DRIVER
#pragma once

#include "GIE_DEBUG_DRIVER.h"
#include "stm32l1xx_hal.h"//freertos
#include "cmsis_os.h"//freertos
#include "initialize.h"//freertos
#include "ProjectConfig.h"//freertos
#define wait_time 15000

typedef enum
{
  SPS30_OK       = 0x00U,//operacion realizada con exito
  SPS30_ERR_CHK    = 0x01U,//error en el checksum, operacion no valida
  SPS30_ERR_UNKNOWN     = 0x02U,//error desconocido por estado interno de la UART
  SPS30_TIMEOUT =0x03U,//no se han recibido datos en el tiempo establecido [no se usa]
  SPS30_BUSY = 0x04U,//no se han recibido datos en el tiempo establecido [no se usa]
  SPS30_ERROR = 0x05U,//no se han recibido datos en el tiempo establecido [no se usa]

} SPS30_command_response;

typedef enum
{
  IEEE754       = 0x00U,
  Unsigned16    = 0x01U,

} SPS30_measurement_format;

struct xDatos_IEEE754    //estructura de datos para agrupar las lecturas en formato IEEE754
{
  float Mass_Concentration_PM1;
  float Mass_Concentration_PM25;
  float Mass_Concentration_PM4;
  float Mass_Concentration_PM10;

  float Number_Concentration_PM05;
  float Number_Concentration_PM1;
  float Number_Concentration_PM25;
  float Number_Concentration_PM4;
  float Number_Concentration_PM10;

  float Typical_Particle_Size;
};typedef struct xDatos_IEEE754 xDatos_IEEE754;

struct xDatos_u16    //estructura de datos para agrupar las lecturas en formato unsigned 16-bits
{
  uint32_t Mass_Concentration_PM1;
  uint32_t Mass_Concentration_PM25;
  uint32_t Mass_Concentration_PM4;
  uint32_t Mass_Concentration_PM10;

  uint32_t Number_Concentration_PM05;
  uint32_t Number_Concentration_PM1;
  uint32_t Number_Concentration_PM25;
  uint32_t Number_Concentration_PM4;
  uint32_t Number_Concentration_PM10;

  uint32_t Typical_Particle_Size;
};typedef struct xDatos_u16 xDatos_u16;


struct version    //estructura de datos para agrupar las versiones del sensor
{
  uint8_t firmware_major_version;
  uint8_t firmware_minor_version;
  uint8_t hardware_version;
  uint8_t SHDLC_protocol_major_version;
  uint8_t SHDLC_protocol_minor_version;
};typedef struct version version;


struct SPS30_SR    //estructura de datos con los bits de error del registro de estado
{
  uint8_t FAN_ERROR;
  uint8_t LASER_ERROR;
  uint8_t SPEED_WARNING;
};typedef struct SPS30_SR SPS30_SR;
```

```
82
83  /*COMANDOS PARA EL SENSOR*/
84  //comienza una nueva medida
85  SPS30_command_response empezar_medida(SPS30_measurement_format formato);
86  //manda al sensor activar el ventilador para limpiarse
87  SPS30_command_response limpieza();
88  //para de medir
89  SPS30_command_response parar_medida();
90  //lee la medida
91  SPS30_command_response lee_medida(SPS30_measurement_format formato, xDatos_IEEE754 *medida_IEEE754,
        xDatos_u16 *medida_u16);
92  //pone al sensor en modo hibernacion
93  SPS30_command_response sleep();
94  //saca al sensor del modo hibernacion
95  SPS30_command_response wake_up();
96  //establece el tiempo en el que automaticamente se limpia
97  SPS30_command_response set_cleaning_interval(uint32_t ulIntervalo);
98  //lee el tiempo establecido para que se limpie automaticamente
99  SPS30_command_response read_cleaning_interval(unsigned long int * out_cleaning_interval);
100 //pide el numero de serie del sensor
101 SPS30_command_response get_serial_number(char * serial_number);
102 //lee la version del firmware, hardware y protocolo SHDLC (el protocolo que usa el sensor)
103 SPS30_command_response read_version(version *version_leida);
104 //lee el registro de estado del sensor, con la opcion de borrar las flags que se han activado
105 SPS30_command_response read_status_register(char formato,SPS30_SR * status_code);
106 //reinicia el sensor
107 SPS30_command_response sensor_aire_reset();
108
109
110 /*FUNCIONES AUXILIARES*/
111 //divide una entrada de tipo long int (32 bits) en un vector de dimension 4 de char (8 bits) para
112 //mandar cada uno por la UART
113 void longint_to_byte_string(unsigned long int entrada, unsigned char * puntero_vector_salida);
114 //comprueba el checksum de los datos recibidos
115 unsigned char verify_checksum(uint8_t * pcPuntero_datos_checksum);
116 //calcula el checksum de los datos que se quieren mandar
117 unsigned char calculate_checksum(unsigned char * puntero_datos_checksum);
118 //el sensor necesita que si ciertos caracteres en la transmision, estos se sustituyan por otros
119 int byte_stuffing (uint8_t ucVector_original[],uint8_t ucSize, uint8_t *pucCadena_bit_stuffing);
120 //recupera los datos recibidos deshaciendo el byte stuffing
121 int inverse_byte_stuffing(uint8_t cadena_original[], uint8_t cadena_inverse_byte_stuffing[]);
122 //toma los 4 bytes de datos leidos y los convierte a float
123 void separa_datos_IEEE754(uint8_t lectura[],xDatos_IEEE754 *medida);
124 //toma los 2 bytes de datos leidos y los convierte a Unsigned 16
125 void separa_datos_unsigned16(uint8_t *pucPuntero_lectura,xDatos_u16 *medida);
126 //desactiva manualmente la lectura por interrupcion de la UART
127 void vDeshabilita_UART();
128
129
130 /*TAREAS DE FREE-RTOS*/
131 void vTaskLecturaIEEE754( void *pvParameters );//tarea que lee en formato IEEE754
132 void vTaskLecturaUnsigned16( void *pvParameters );//tarea que lee en formato uNSIGNED 16
133 void vTaskFunciongenerica( void *pvParameters );//tarea que ejecuta dos comandos y se borra
134 void vTaskRecogeDatos( void *pvParameters );//tarea que lee datos de las colas
135
136
137 #endif
```

Código 32: Contenido del fichero de estructuras, tipos y funciones.