Universitat Oberta de Catalunya

uoc.edu

Diseño y desarrollo de un datalogger de bajo consumo para sensores oceanográficos y meteorológicos

Autor: José Raúl Santana Jiménez
Máster Universitario en Ingeniería de Telecomunicación
TFM-Electrónica

Consultor: Xavier Saura Mas
Profesor responsable de la asignatura: Carlos Monzo Sánchez

01/2020

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Diseño y desarrollo de un datalogger de bajo consumo para sensores oceanográficos y meteorológicos*
Nombre del autor:	*José Raúl Santana Jiménez*
Nombre del consultor/a:	*Xavier Saura Mas*
Nombre del PRA:	*Carlos Monzo Sánchez*
Fecha de entrega (mm/aaaa):	01/2020
Titulación::	*Máster en Ingeniería de Telecomunicación*
Área del Trabajo Final:	*TFM-Electrónica*
Idioma del trabajo:	*Español*
Palabras clave	*Sistema embebido, datalogger, oceanografía.*

Resumen del Trabajo

Las actuales estaciones y plataformas que están destinadas a la obtención a largo plazo de medidas precisas, consistentes y continuas de parámetros meteorológicos y fisicoquímicos marinos, para uso tanto científico como de monitorización medioambiental, están equipadas con sistemas de adquisición y control muy dispares y, en ocasiones, incompatibles entre sí. En muchos casos se trata de sistemas muy genéricos y no optimizados para su uso en este tipo de plataformas, en otros, de sistemas desarrollados específicamente para una estación en concreto de muy difícil adaptación a otras plataformas.

La motivación de este trabajo viene dada por que se ha detectado la necesidad de disponer de un datalogger de este tipo de sensores, de bajo consumo, de pequeño tamaño y fácilmente configurable por el usuario, de manera que pueda ser instalado y configurado en el mayor número de estaciones y plataformas posible, permitiendo avanzar en la estandarización y facilitando la comparabilidad de las observaciones tomadas en las diferentes plataformas.

En este trabajo se efectuó el diseño y desarrollo de un datalogger de datos de sensores oceanográficos y meteorológicos con flexibilidad y adaptabilidad suficiente para poder ser fácilmente instalado en diferentes plataformas y estaciones oceanográficas y meteorológicas.

Para la ejecución del trabajo se ha empleado la tarjeta CY8CKIT-062-WIFI-BT, como soporte hardware sobre el que se ha diseñado e implementado el datalogger. Se ha desarrollado un programa cliente que permite al usuario la configuración y uso del datalogger. Finalmente se han efectuado una prueba de funcionamiento con un sensor oceanográfico adquiriendo datos reales.

Abstract

The currently deployed stations and platforms around the earth, that are intended to obtain long-term, consistent and continuous measurements of meteorological and marine physicochemical parameters, for both scientific and environmental monitoring use, are equipped with very different acquisition and control systems and, in occasions, incompatible with each other. In many cases there are very generic and not optimized systems for it's use on this kind of platforms, in other cases, there are systems developed specifically for a particular station that are unflexible and very difficult to adapt to other platforms.

The motivation of this work is given by the fact that, the need for a flexible, low power, small size and very easily configurable by the user datalogger of this kind of sensors has been detected, so that it can be installed and configured in the maximun amount and diversity of stations and platforms possible, and thus, enabling the standardization and making smoother the comparability of observations taken on different platforms.

In this work, the design and development of a datalogger for oceanographic and meteorological sensors data, with high flexibility and capacity to be easily installed on different oceanographic and meteorological stations and platforms, was carried out.

The CY8CKIT-062-WIFI-BT board has been used as the hardware support over which the datalogger has been designed and implemented. A client program, that allows the user to configure and use the datalogger has been developed. Finally, a functional test was done using an oceanographic sensor acquiring real data through the datalogger.

Dedicado a Santiago, Antonia, Ana y Valeria

Índice

Lista de figuras

Índice de tablas

1. Introducción

1.1 Contexto y justificación del Trabajo

La medida continuada en el tiempo de parámetros meteorológicos y fisicoquímicos de agua marina permite crear y alimentar bases de datos que almacenan series temporales de larga duración. Estas series temporales son de gran valor para diversas áreas científicas oceanográficas y ambientales, por lo que su disponibilidad y la calidad de datos que albergan es crucial para el desarrollo del conocimiento de aspectos como variaciones climáticas a pequeña y gran escala espacial, dinámica marina y atmosférica o monitorización de calidad de agua, entre otros.

En la actualidad se encuentra desplegada por todo el planeta una gran cantidad de estaciones fijas y plataformas móviles autónomas, destinadas a la adquisición de parámetros fisicoquímicos marinos y meteorológicos. Estas estaciones suelen integrar y operar diferentes tipos de sensores con capacidad para la adquisición de datos válidos tanto para estudios científicos como para monitorización de calidad de agua, de los que existe un mercado de marcas y modelos que proporciona instrumentos de elevada calidad.

Las funciones de control de los sensores y el funcionamiento general de las estaciones suelen estar gestionadas por sistemas electrónicos diseñados específicamente para cubrir las necesidades propias de cada estación. Esta situación deriva en el uso de muy diversas tecnologías de adquisición de datos, lo que dificulta la interoperabilidad y, en ocasiones, la comparativa entre datos adquiridos por las diferentes estaciones.

Gran parte de estas estaciones y plataformas de observación se encuentran localizadas en lugares remotos y con espacio físico limitado, de modo que tanto el factor de forma como la energía necesaria para alimentar los dispositivos electrónicos suelen ser los principales factores limitantes a la hora de acometer el diseño del sistema de integración y control.

En este Trabajo Fin de Máster (TFM) se plantea el diseño y desarrollo de un *datalogger* en un sistema electrónico embebido, de bajo consumo energético y fácilmente configurable por el usuario para la adquisición y procesado de datos oceanográficos y meteorológicos. El *datalogger* será capaz de adquirir datos de sensores de parámetros como temperatura, conductividad eléctrica, oxígeno disuelto, clorofila y turbidez de agua marina, velocidad y dirección del viento, temperatura y presión atmosférica y humedad relativa. El dispositivo desarrollado permitirá la configuración por parte del usuario de variables relacionadas con la adquisición de las medidas, tales como frecuencia de medida, tipo de procesado aplicado a los datos adquiridos, envío de datos en tiempo real, entre otras. De manera que se trata de un dispositivo fácilmente adaptable a las necesidades específicas de observación de diferentes estaciones.

La configuración del *datalogger* se realizará por medio de una aplicación cliente que se ejecuta en un computador conectado al *datalogger* por medio de un enlace *WiFi*.

El *datalogger* contará con dos formas de adquisición de datos compatibles entre si: autocontenido y en tiempo real.

En la adquisición autocontenida, el sistema adquiere datos de los sensores, los procesa según su configuración y los almacena en su memoria interna. Una vez finalizado el periodo de adquisición, el usuario, haciendo uso del cliente, puede detener la adquisición de datos y descargar el contenido de la memoria por medio de un enlace *WiFi*.

En la adquisición en tiempo real, el *datalogger* envía los datos adquiridos y procesados de forma periódica al cliente. En el cliente los datos son mostrados al usuario y guardados en un directorio local.

En la versión inicial del *datalogger* se emplea la interfaz *WiFi* como único medio para establecer la adquisición de datos en tiempo real. De esta manera el uso del *datalogger* trabajando en este modo se restringe a zonas portuarias o costeras con disponibilidad de un punto de acceso *wifi* para la recepción de datos.

1.2 Objetivos del Trabajo

Los objetivos principales del trabajo se citan a continuación:

- Desarrollar un sistema *datalogger* de adquisición de datos con capacidad para operar con sensores oceanográficos y meteorológicos de calidad científica.

- Diseñar el sistema de adquisición sencillo de configurar y usar por parte del usuario, de baja potencia y factor de forma. El sistema incluirá un programa cliente que permita al usuario la configuración y uso del *datalogger* por medio de un enlace inalámbrico.

- Diseñar el sistema de adquisición para su fácil integración en diferentes plataformas y estaciones de medida.

1.3 Enfoque y método seguido

El autor de este TFM cuenta con experiencia profesional en la configuración y despliegue de diferentes tipos de estaciones y plataformas de adquisición de datos meteorológicos y oceanográficos. A lo largo de este ejercicio profesional se ha detectado la disparidad de dispositivos *dataloggers* empeados, así como la escasez de *dataloggers* específicos para datos meteorológicos y oceanográficos, de configuración flexible y sencilla para el usuario. Además se ha podido identificar los principales parámetros de configuración de las adquisiciones de datos que demandan

las comunidades que trabajan con este tipo de datos. Los principales parámetros de configuración identificados son el intervalo de adquisición, el número de lecturas por adquisición y el procesado en cada adquisición que suelen ser el cálculo de medias o mediana del conjunto de datos adquiridos.

En la actualidad no se ha identificado ningún dispositivo *datalogger* específico para datos meteorológicos y oceanográficos que permita de manera simple y flexible al usuario configurar adquisiciones en diferentes estaciones, plataformas y sensores. Por otro lado, sí que existen dispositivos programables que permiten al usuario crear sus propios sistemas de adquisición a medida, sin embargo no se trata de una tarea trivial, ya que es preciso contar con conocimientos básicos de programación y superar cierta "curva de aprendizaje" propia del sistema a programar.

Es por esto por lo que la estrategia empleada en este TFM pasa por el diseño y la implementación desde cero de un dispositivo *datalogger* flexible y sencillo de usar que ocupe el espacio entre los *datalogger* comerciales actuales y las necesidades de adquisición de datos de las comunidades científicas oceanográficas y meteorológicas y de monitorización medioambiental. La metodología de desarrollo del TFM está basada en las tarjetas *hardware* seleccionadas para la implementación del *datalogger* y el uso de sus herramientas de desarrollo asociadas.

1.4 Planificación del Trabajo

La figura 1 contiene las tareas a realizar durante el TFM y su planificación temporal en forma de diagrama de Gantt. Los diferentes tipos de tareas han sido codificados por colores. Las tareas en color amarillo se corresponden con trabajos de redacción de documentación asociada a las PECs de la asignatura. Los hitos de entrega de PECs aparecen marcados en color rojo. Las tareas de color verde están relacionadas con trabajos de estudio de documentación de herramientas de desarrollo. Las tareas naranja son de desarrollo y pruebas del sistema. Por último, las tareas de color azul son tareas de selección y adquisición de herramientas *hardware* y *software*.

Los principales recursos necesarios para la ejecución del TFM y su breve descripción se citan a continuación:

- Tarjeta electrónica con dispositivo microcontrolador (MCU) con capacidad para dar soporte a aplicación del *datalogger*. La tarjeta debe disponer, además de suficientes líneas de entrada/salida para las comunicaciones con los sensores, capacidad de almacenamiento de

los datos recibidos y contar con un módem y antena para la creación de un punto de acceso a una red *WiFi*.

- Computador (PC) con adaptador de red inalámbrica integrado y con las herramientas de desarrollo propias de la tarjeta electrónica seleccionada instaladas.

- Osciloscopio digital y multímetro analógico, necesarios para la implementación y depuración de los puertos serie asíncronos del *datalogger* y chequeo de niveles de tensión en la tarjeta.

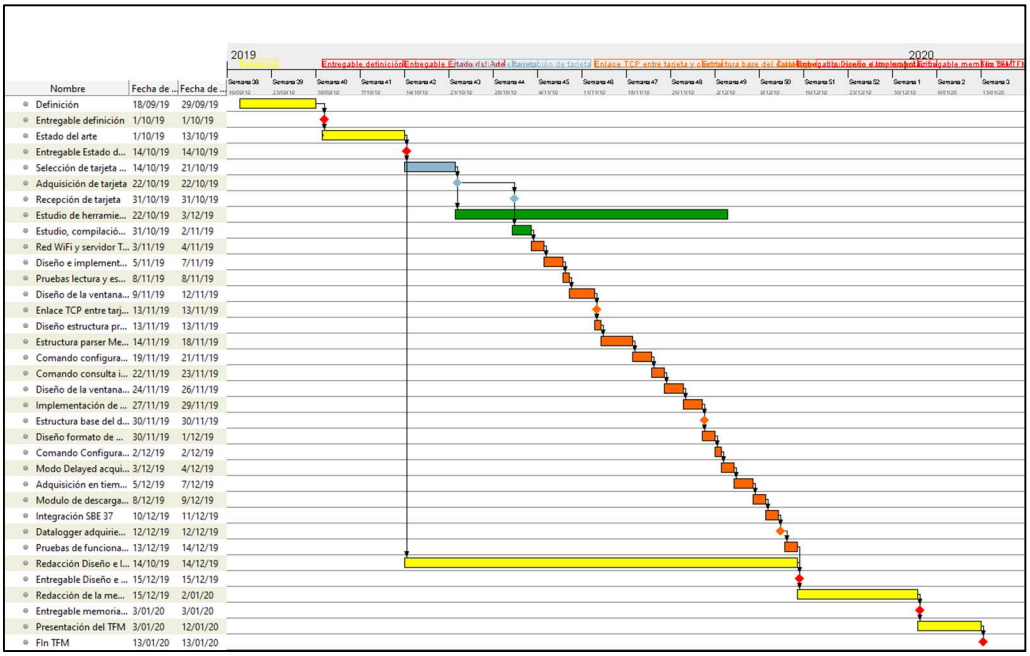


Figura 1. Diagrama de Gantt del TFM

1.5 Breve sumario de productos obtenidos

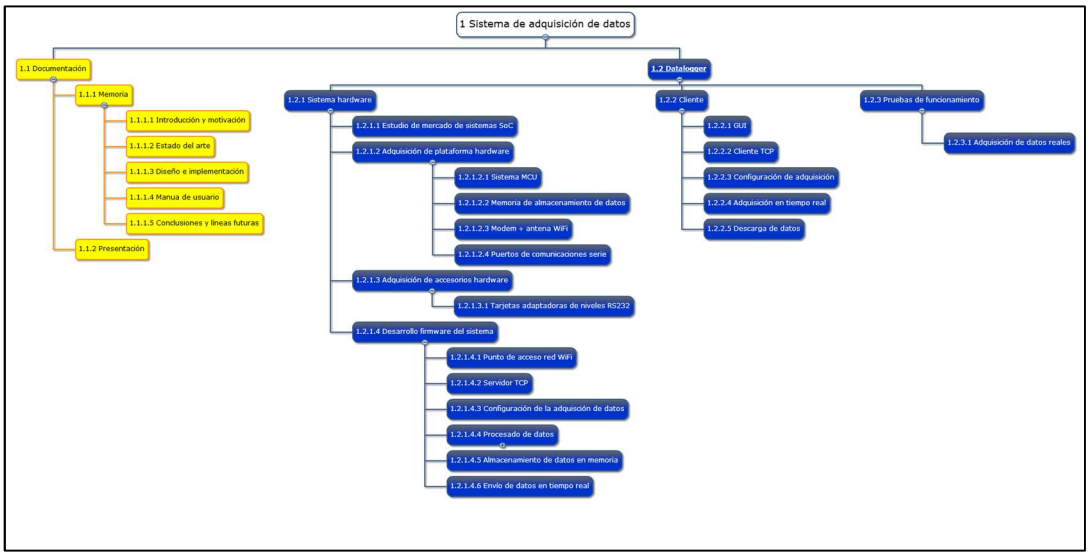


Figura 2. Diagrama WBS

La figura 2 se citan, en forma de diagrama WBS (*Work Breackdown Structure*), el conjunto de productos obtenidos en este TFM.

1.6 Breve descripción del contenido de la memoria

Esta memoria está estructurada en cuatro capítulos y dos anexos cuyo contenido se resume a continuación.

- **Capítulo 1 - Introducción**. Este capítulo establece una aproximación al contexto en el que se plantea la ejecución del TFM. Se establecen las líneas maestras que seguirá el desarrollo del trabajo y los principales objetivos a alcanzar. El capítulo incluye una planificación temporal y el listado de productos a obtener.

- **Capítulo 2- Estado del arte**. Este capítulo se estructura en tres apartados destinados a proporcionar al lector una visión del ámbito de aplicación y el encaje del TFM desde tres puntos de vista distintos e intrínsecamente relacionados.

  En primer lugar se hace un breve recorrido por las principales iniciativas, a diferentes niveles espaciales y administrativos, destinadas a la coordinación de plataformas y estaciones de observación oceánicas y meteorológicas. Se pone de manifiesto que son muy discretos los avances en la homogeneización de los sistemas electrónicos de adquisición de datos en las principales redes de observatorios.

  El segundo ámbito estudiado es el de los *dataloggers* disponibles en el mercado para aplicaciones oceanográficas y meteorológicas. Se hace un breve recorrido por los principales fabricantes y modelos de *dataloggers* comerciales que suelen ser empleados en las redes de observatorios citadas en el apartado anterior.

  Por último se presentan al lector, de forma somera, las tecnologías empleadas para la medición con sensores de los principales parámetros fisicoquímicos y biológicos monitorizados por los observatorios oceánicos y atmosféricos actuales. El apartado se completa con tablas comparativas de las características de sensores oceanográficos y meteorológicos de los principales fabricantes a nivel global.

- **Capítulo 3 – Diseño e implementación**. En este capítulo se detalla el diseño realizado del disositivo *datalogger* objeto de este TFM. Incluye la descripción del sistema *hardware* empleado, las herramientas de desarrollo y la estructura del *firmware* del *datalogger* y del *software* del cliente. El capítulo incluye una descripción detallada de la implementación del *firmware* y del *software* diseñados. Concluye con la descripción y los resultados obtenidos de la prueba de funcionamiento del dispositivo.

- Capítulo 4 – Conclusiones y líneas futuras. Capítulo compuesto por reflexiones críticas del alcance final del TFM y de sus usos potenciales. El capítulo incluye un listado de línas futuras de trabajo para completar y añadir nuevas funcionalidades y mayor versatilidad al sistema.

- Anexo A. Se incluye en este anexto todo el código generado para la implementación del *firmware* del datalogger y del *software* del cliente.

- Anexo B. Incluye los esquemáticos de los bloques hardware que componen el *datalogger*.

La memoria se completa con los listados de referencias bibliográficas y el glosario de acrónimos usados a lo largo del documento.

2. Estado del arte

2.1 Sistemas actuales de observación meteo-oceanográficos y principales iniciativas de coordinación e integración

En los mares y océanos de todo el planeta se encuentran desplegadas centenares de plataformas de observación equipadas con sensores para la adquisición de parámetros atmosféricos y oceanográficos. Estas plataformas son diseñadas, fabricadas y operadas por multitud de instituciones y organismos independientes entre sí. Los datos adquiridos por muchas de estas plataformas son destinados al estudio de fenómenos ambientales, climáticos y oceánicos o de control de contaminantes y calidad de agua, entre otras aplicaciones.

En este escenario, es necesaria la adopción de iniciativas enfocadas al impulso de la coordinación y estandarización de estos observatorios a nivel global. De esta manera se facilita el intercambio y la comparación entre datos de los observatorios. Esta estandarización debe abarcar toda la cadena de adquisición y tratamiento de los datos, incluyendo los sensores y los sistemas electrónicos de adquisición de datos.

En este sentido, el *Global Ocean Observation System (GOOS)*, de la *Intergovernmental Oceanographic Commission (IOC)* de la UNESCO ha definido una serie de *Essential Ocean Variables (EOVs)* a las que acompañan con recomendaciones acerca de cómo efectuar las mediciones y el tratamiento de los datos que pueden ser adoptadas por los observatorios oceánicos.

En la línea del avance en la cooperación para el intercambio de datos y la estandarización de los observatorios oceánicos se están ejecutando diversas iniciativas a nivel regional y global con diferentes alcances.

2.1.1 Iniciativas de coordinación e integración a nivel europeo

En el ámbito europeo, la principal iniciativa para centralizar las observaciones ambientales a nivel global es el programa Copérnico (1). Copérnico se estructura en tres componentes: el componente espacial, el componente de servicios y el componente de medidas *in situ.*

Dentro del componente *in situ,* el *Copernicus Marine Environment Monitoring System (CMEMS)* ofrece acceso a datos procedentes de una extensa red de más de un centenar de boyas equipadas con instrumentación meteorológica y oceanográfica (2).

Por otro lado, el proyecto *Fixed-point Open Ocean Observatories* (FixO3), financiado por la Comisión Europea (3) entre los años 2013 y 2017, supuso un importante esfuerzo por parte de la Unión Europea para la integración y armonización tecnológica y procedimental en una amplia red de observatorios oceánicos europeos. Entre sus objetivos se incluían

facilitar el acceso a los observatorios por parte de toda la comunidad científica interesada y la generación de documentación de buenas prácticas para toda la cadena de adquisición y procesado de datos. Un total de 21 observatorios desarrollados y operados por diferentes instituciones de países de la Unión Europea participaron en *FixO3*.

Por otro lado, actualmente, en el seno de la Unión Europea, se encuentra en operación el consorcio *European Multidisciplinary Seafloor and water column Observatory (EMSO)* (4), con estructura formal de *European Research Infraestructure Consortium (ERIC)*. El consorcio está formado por once infraestructuras diseñadas y operadas por instituciones de ocho países, de las cuales tres son observatorios con boyas superficiales y el resto son estructuras desplegadas en el lecho marino.

Entre los objetivos de EMSO se encuentra el desarrollo de tecnología relacionada con sensores, comunicaciones y sistemas *offshore.* En este sentido, en el seno del consorcio, se encuentra en desarrollo el proyecto EMSODev (5), cuyo principal objetivo es el diseño y desarrollo de un sistema denominado *Emsodev Generic Instrument Module (EGIM).*

El dispositivo EGIM está siendo desarrollado con el objetivo de llegar a ser el sistema de adquisición y control estándar y comun en los observatorios participantes en el consorcio EMSO. Es por lo tanto el primer esfuerzo en el ámbito de las instituciones europeas de estandarización a nivel de sistemas electrónicos de adquisición empleados.

2.1.2 Iniciativas de coordinación e integración en cuencas oceánicas

2.1.2.1 Cuenca atlántica

Por su parte, y también con financiación de la Unión Europea, dentro del programa de financiación H2020, el proyecto AtlantOS (6) trabaja en la implementación del GOOS en el ámbito del océano Atlántico. En AtlantOS están integrados países de ambos márgenes del océano Atlántico y su principal objetivo se centra en el establecimiento de un sistema de coordinación de las actividades de observación realizadas en el océano Atlántico, inicialmente denominado *Integrated Atlantic Ocean Observing System* (IAOOS).

En la latitud tropical del océano Atlántico se encuentra desplegada la red *Prediction and Research Moored Array in the Tropical Atlantic* (PIRATA) (7). Constituye una red de observatorios flotantes existente desde la década de los años 90 del siglo pasado y cuyo objetivo es la adquisición de datos destinados al estudio de las interacciones océano-atmósfera en la región. En la actualidad participan en el mantenimiento de la red instituciones de Estados Unidos, Francia y Brasil.

2.1.2.2 Cuenca del Océano Índico

En la cuenca del Océano Índico se identifica la iniciativa *Research Moored Array for African-Asian-Australian Monsoon Analysis and Prediction* (RAMA) (8). Es una iniciativa en la que participan instituciones de Estados Unidos, Indonesia, China y la India. En su estado actual de desarrollo, la red RAMA cuenta con 39 observatorios flotantes ya desplegados del total de 46 previstos. Su misión fundamental es la del suministro de datos para el estudio del papel que juega el Índico en la formación de los monzones.

2.1.2.3 Cuenca del Océano Pacífico

En la cuenca del Océano Pacífico se encuentra desplegada la red *Tropical Atmosphere Ocean* (TAO/TRITON) (9). Se trata de una red de estaciones de observación cuyo despliegue se inició a mediados de la década de los 80 del siglo pasado. La red está destinada a aportar datos para facilitar la obtención de conocimiento acerca del fenómeno de *El Niño Southern Oscillation* (ENSO). Actualmente se encuentra participada por dos instituciones de Estados Unidos y de Japón y está compuesta por 67 observatorios fijos. Las boyas de la red TAO comparten el mismo diseño de datalogger para la adquisición de sus datos. Se trata de un dispositivo denominado *Advanced Modular Payload System* (AMPS) (10), diseñado específicamente para la red TAO.

Además, dentro de la cuenca del Océano Pacífico se encuentra en ejecución el *proyecto Tropical Pacific Observing System* (TPOS) (11). Este proyecto, auspiciado por el GOOS, cuenta con la participación de 65 instituciones pertenecientes a países de los continentes americano, asiático, oceánico y europeo. Su principal objetivo se focaliza en detectar necesidades no cubiertas de observación y planteamiento de soluciones para aumentar la efectividad observacional de la región tropical del Pacífico aplicada al estudio del fenómeno ENSO. Entre las acciones propuestas se plantea el fondeo de una extensa red de boyas en la región con capacidad de observar parámetros oceánicos y atmosféricos.

2.1.3. Iniciativas de coordinación e integración a nivel global

A nivel global, la iniciativa OceanSITES (12) es parte integrante del GOOS y constituye desde 1999 una red mundial de observatorios oceánicos con capacidad para medir multitud de parámetros atmosféricos y oceánicos desde la superficie hasta el fondo marino. La red OceanSITES está participada por numerosas organizaciones de todo el globo y los observatorios integrados en ella intercambian datos e información. Los observatorios de la red se encuentran desplegados en todas las cuencas oceánicas del planeta, incluidas las dos regiones polares, además de en los principales mares del globo.

Otra destacada iniciativa a nivel global es el proyecto *Deep Ocean Observing Strategy* (DOOS) (13). Se trata de un proyecto enmarcado en el GOOS y participado por miembros de organismos pertenecientes a países de América, Europa y Asia. Entre sus principales objetivos se citan el diseño y la evaluación de sistemas de observación del océano profundo y actuar como órgano centralizador para el intercambio de información y datos entre los actores interesados en el estudio y observación del océano profundo a nivel global. Actualmente DOOS está manteniendo y ampliando un inventario de los observatorios relacionados con la adquisición de datos oceánicos profundos con el fin de identificar su estado actual y como base para articular una estrategia de observatorios de océano profundo global y sostenible.

2.2 Sistemas de adquisición de datos oceanográficos y meteorológicos

En el apartado anterior se citan las principales iniciativas que, a nivel internacional, tratan de aunar y coordinar esfuerzos de múltiples organismos de diferentes países en el despliegue y operación de grandes redes de sistemas de adquisición de datos meteo-oceanográficos.

Cada organismo encargado del desarrollo de estos observatorios emplea diseños y sistemas electrónicos propios para la adquisición de datos y control. Este hecho hace que muchas de estas redes de observatorios internacionales estén compuestas por estaciones de medida muy dispares entre ellas en lo que a sistemas electrónicos de adquisición de datos se refieren.

En este sentido, el esfuerzo realizado en el consorcio europeo EMSO, dirigido la futura adopción del dispositivo EGIM como sistema de adquisición y control estándar de los observatorios del consorcio, es la aproximación más ambiciosa destinada a la unificación tecnológica en toda una red de observatorios a nivel internacional.

En los siguientes párrafos se citan y comentan las características de algunos de los principales dispositivos de adquisición y control empleados por diferentes observatorios meteo-oceanográficos actuales.

2.2.1 *Seawatch* y *Seawatch Wavescan*

Seawatch (14) es un dispositivo comercial completo, compuesto por una boya, un sistema de alimentación autónomo, un *datalogger* y un sistema de comunicaciones satélite. El *datalogger* de estos dispositivos, denominado *Oceanor WaveSense*, es un diseño propiedad del fabricante y configurable por el propio usuario para cada aplicación. Entre sus principales características electrónicas y mecánicas se citan las siguientes:

- 16 entradas analógicas diferenciales con ADC de 16 bits.

- 18 puertos RS232-C, 1 RS485 y 2 RS422.
- 26 líneas digitales de entrada/salida.
- 11 líneas de control de alimentación de sensores.
- 512 Mb de almacenamiento de datos.
- Alimentación entre 7 y 15 Vdc.
- Consumo mínimo de 110 mA adquiriendo y 75 mA en modo suspendido.
- Peso 2,5 kg

Dispositivos de este tipo son empleados en el sistema POSEIDON (15), que es una red de boyas de monitorización meteoceanográfica destinada a la monitorización del Mar Egeo y de Creta e integrados en la red OceanSITES y del consorcio EMSO ERIC.

2.2.2 Gama de *dataloggers* de *Campbell Scientific*

Uno de los principales fabricantes de *dataloggers* para sistemas de adquisición de datos ambientales es *Campbell Scientific*. Entre sus productos se encuentran los datalogger *CR1000X*, *CR800* y *CR300* (16).

Se trata de dispositivos de adquisición de datos de diferentes tamaños y capacidades en cuanto a líneas de entrada/salida y protocolos de comunicaciones disponibles.

Son sistemas de bajo consumo y cuentan con un lenguaje de programación y herramientas de diseño propio que permite al usuario crear programas de adquisición adaptados a cada caso de uso. Sin embargo es necesaria la adquisición de conocimientos de las herramientas de desarrollo de *Campbell* por parte del usuario para la creación de programas propios de adquisición.

Campbell Scientific cuenta además con una gama de periféricos con capacidad para dotar a los *dataloggers* de ampliaciones del número de puertos de entrada/salida y comunicaciones *WiFi*, GPRS o satélite.

El *CR1000X* es el *datalogger* de mayores prestaciones dentro de la gama de *Campbell*. Como caso de uso de este dispositivo se puede citar su empleo para la adquisición de datos en la boya superficial de la Estación Europea de Series Temporales Oceánicas de Canarias (ESTOC) (17), se trata de un observatorio oceánico integrado en las redes oceanSITES y EMSO. Entre sus principales características electrónicas y mecánicas se citan las siguientes:

- Procesador Renesas RX63N, 32 bits, 100 MHz.
- 128 MB de capacidad de almacenamiento de datos, ampliable hasta 16 GB con un módulo de expansión *micro SD* de memoria flash.

- Hasta 7 líneas de comunicaciones configurables como puertos SDI-12, RS232-C, RS485 y SDM (protocolo propiedad de *Campbell Scientific*).
- Disponibilidad de módulos de expansión de puertos RS232-C, RS485.
- Alimentación entre 10 y 18 Vdc
- Consumo mínimo en modo activo con un *scan rate* de 20 Hz: 55 mA. Consumo en modo suspendido < 1 mA
- Peso: 0,86 kg.

2.2.3 *NexSens*

Nexsens technology (18) cuenta con un catálogo de *dataloggers* específicamente diseñados para su instalación en localizaciones remotas y ambientes corrosivos.

Dentro de su gama, la serie *X2* es un conjunto de *dataloggers* ambientales de muy bajo consumo que cuentan con tres puertos digitales de entrada/salida compatibles con los protocolos de comunicaciones más empleados por sensores ambientales (SDI-12, RS232-C y RS485).

Los *X2* tienen capacidad para almacenar los datos adquiridos en una memoria *flash* interna de 64MB y en una tarjeta *micro SD flash* de hasta 32GB. Como opciones para las comunicaciones del dispositivo se encuentra el establecimiento de comunicaciones *WiFi*, satélite, radio enlace a 900MHz o por medio de la red 4G *LTE*.

2.2.4 *Observator Instruments OMC-043-III*

Por su parte, el *datalogger OMC-043-III* (19) de *Observator Instruments* es un dispositivo de bajo consumo con capacidad de comunicaciones GPRS y satélite.

Cuenta con un puerto serie RS232-C, un puerto RS422/485, además de dos entradas analógicas en corriente (4-20 mA) y otras dos en tensión (0 – 10 Vdc). Cada una de estas entradas es digitalizada en palabras de 12 bits.

El *datalogger* cuenta además con un módem GSM/GPRS y un posicionador GPS internos. El dispositivo es capaz de adquirir datos de un listado predefinido de sensores fabricados por *Observator Instruments* y por terceros fabricantes. La configuración de las medidas y sensores a adquirir en cada una de las entradas se realiza por medio de una utilidad *software* de programación específica.

2.2.5 *Skye Instruments DataHog2*

DataHog2 datalogger de *Skye* (20) es un dispositivo de adquisición de datos con intervalo de medidas configurable en valores predefinidos por el fabricante. Dispone de hasta 16 entradas analógicas y digitales. Las

entrada analógicas pueden ser en tensión, tanto *single ended* como diferenciales, o en corriente. Las entradas digitales son contadores de pulsos digitales de 5V de amplitud.

2.2.6 *Datalogger Kunak K111*

Kunak K111 (21) es un *datalogger* de bajo consumo diseñado para ser instalado en ambientes hostiles y remotos. Cuenta con entradas analógicas tanto en tensión como en corriente con ADC de 16 bits, y con entradas digitales como contadores de pulsos y medidores de frecuencia y *Modbus*. Dispone de opciones de comunicación GPRS, *WiFi* y LoRa 169/433/868.

2.2.7 *ONSET HOBO RX3000*

HOBO RX3000 de *ONSET* (22) es un *datalogger* diseñado específicamente para adquirir datos de un conjunto de sensores propios del fabricante o de terceros, entre los que se encuentran un amplio conjunto de sensores meteorológicos. Cuenta con 10 conectores para sensores. El intervalo de medida es configurable por el usuario. Las opciones de comunicaciones disponibles son por *Ethernet*, *WiFi* o por GPRS

2.3 Sensores meteorológicos y oceanográficos

Entre los sensores oceanográficos capaces de obtener datos de variables o subvariables oceanográficas esenciales, los más empleados en todos los observatorios marinos del planeta son los sensores de temperatura y conductividad del agua marina. Adicionalmente se emplean también, de forma común, sensores de oxígeno disuelto, turbidez y clorofila, entre otros menos comunes como pH o pCO. Se repasan a continuación las principales tecnologías de sensores disponibles en el mercado para la adquisición de los datos más usuales en los observatorios marinos globales.

2.3.1 Temperatura

Los sensores de temperatura de agua marina se basan en el empleo de resistencias variables con la temperatura. Los transductores empleados son, generalmente, resistencias de platino. El platino es físicamente estable en el ambiente marino, tienen un coeficiente de temperatura positivo y la variación de su resistividad tiene un tiempo de respuesta bajo frente a cambios en la temperatura ambiental. Son sensores muy estables que requieren muy poco mantenimiento.

Como circuito acondicionador se suele emplear el puente de *Wien* en el que una de sus ramas está ocupada por la resistencia variable de platino. El valor óhmico de la resistencia de platino en función de la temperatura se aproxima por la siguiente expresión:

$$R_t = R_0(1 + at + bt^2)$$

Donde R_0 es el valor de la resistencia del transductor a 0ºC, y a y b son coeficientes propios del transductor.

Los sensores de temperatura para aplicaciones científicas en oceanografía física deben ser capaces de efectuar mediciones con exactitud en el rango de unos pocos miligrados centígrados. Para alcanzar este nivel, el diseño del sensor debe extremar el control sobre aspectos como el tiempo de respuesta, el efecto de la presión a grandes profundidades o el autocalentamiento por la disipación de potencia eléctrica en el transductor. Además, gran parte del esfuerzo en la obtención de estos niveles de exactitud se destina a las tareas de calibración de los sensores, para lo que es necesario disponer de instalaciones con baños extremadamente aislados de temperatura controlada por patrones generalmente ajustados a la ITS-90.

En la tabla 1 se citan algunos de los principales sensores de temperatura empleados en los observatorios actuales.

Marca	Modelo	Rango ºC	Exactitud ºC	Resolución ºC	Interfaz
Ocean Tools	C-Temp	0 a 50	0,05	-	RS232
SeaBird	SBE 38	-5 a 35	±0,001	0.00025	RS232
RBR	RBRcoda^3 T	-5 a 35	±0,002	<0.00005	RS232
Aaderaa	4060	-4 a 36	±0.03 a (0.054°F) ±0.01 a (0.018°F)	0.001	RS232

Tabla 1. Sensores de temperatura de agua marina

2.3.2 Conductividad

El método más común en la medida de la conductividad del agua marina, para aplicaciones oceanográficas, es el uso de sensores basados en la medida de la resistividad eléctrica en un volumen de agua situado entre dos o más electrodos por medio de la técnica analítica de la conductimetría.

Estos electrodos están compuestos de materiales conductores y resistentes al ambiente marino y a las reacciones de reducción/oxidación que se pueden producir en su superficie.

Ante una excitación eléctrica externa, la carga eléctrica en el agua marina es transportada por los electrolitos presentes en ella. La carga transportada por unidad de tiempo depende, además de la señal de excitación aplicada a los electrodos, de la movilidad de los iones presentes, de la temperatura a la que se encuentran y su concentración, por lo que las señales eléctricas de excitación aplicadas a los electrodos deben ser diseñadas teniendo en cuenta estas características del transporte de cargas para limitar al máximo los posibles efectos que pueden afectar a la determinación de la conductividad por este método.

Los principales efectos a minimizar son la polarización por la concentración de iones en torno a los electrodos y la electrólisis.

Generalmente las señales de excitación aplicadas a los electrodos son alternas con diferentes formas de onda y con frecuencias de entre unos cientos de hercios hasta varios kilohercios.

En la tabla 2 se citan los sensores de conductividad más empleados en los observatorios desplegados a nivel global.

Marca	Modelo	Rango mS/cm	Exactitud mS/cm	Resolución mS/cm	Interfaz
Aanderaa	4319	0 a 75	0,05	0,002	RS232
SeaBird	SBE4plus	0 a 70	± 0.003	± 0.003	Señal de Frecuencia
Sea & Sun Technology	7 Pole Cell	0 a 60	-	-	RS232
AML	C-XChange	0 a 90	±0,01	±0,001	RS232

Tabla 2. Sensores de conductividad

2.3.3 Sistemas CT

En muchas ocasiones se efectúan las medidas de los valores de temperatura y conductividad integradas en un único instrumento. Ambos valores son necesarios para el cálculo de la salinidad del agua marina.

Empleando un único instrumento para los dos parámetros se facilita la sincronización de las medidas de ambos parámetros y aumenta la fiabilidad del cálculo de la salinidad. En la tabla 3 se citan algunos de los principales instrumentos comerciales de medida de conductivida y temperatura de agua marina.

Marca	Modelo	Rango	Exactitud	Resolución	Interfaz
RBR	RBRduo	0 a 85mS/cm	±0.003mS/cm	0.001mS/cm	USB-C o RS-232/485
		-5°C a 35°C	±0.002°C	<0.00005°C	
Idronaut	OS304	0 a 70 mS/cm	0.003 mS/cm	0.0002 mS/cm	RS232
		-5 a+35 °C	0.002 °C	0.0001 °C	
SeaBird	SBE37-SM	0 a 70 mS/cm	0.003 mS/cm	0.0001 mS/cm	RS-232 o RS-485
		-5 a 45 °C	± 0.002 °C	0.0001 °C	
Valeport	miniCT	0 a 80 mS/cm	±0.01mS/cm	0.001mS/cm	RS232 o RS485
		-5 a +35°C	±0.01°C	0.001°C	

Tabla 3. Sistemas de conductividad y temperatura

2.3.4 Oxígeno disuelto

En la medida de oxígeno disuelto en agua marina generalmente se emplean sensores basados en dos tecnologías diferentes:

Sensores ópticos. Se basan en el fenómeno de desactivación fluorescente o *quenching* de un luminóforo embebido en una membrana permeable al oxígeno. El oxígeno es capaz de absorber la fluorescencia emitida por el luminóforo, de manera que la intensidad de la fluorescencia, medida por técnicas de espectroscopia de luminiscencia, es inversamente proporcional a la concentración de oxígeno.

Debido a sus características de estabilidad en las medidas a largo plazo, los sensores ópticos son empleados comúnmente en aplicaciones de despliegues de larga duración temporal.

Sensores polarográficos. Estos sensores emplean una membrana basada en la membrana polarográfica de *Clarck*. En ellos la membrana de *Clarck* permite la difusión al interior del sensor donde se produce una reacción electroquímica en la que el oxígeno ambiente es reducido y se genera una corriente eléctrica. La corriente eléctrica generada en la reacción es proporcional a la concentración de oxígeno disponible. El tiempo de respuesta en este tipo de sensores suele ser mucho menor que en el caso de los sensores ópticos, por lo que son empleados en aplicaciones de perfiladores de columnas de agua. Por otro lado, las características fisicoquímicas de la membrana polarográfica presentan una elevada deriva con el tiempo, lo que hace que necesiten procesos de calibración y mantenimiento con mayor frecuencia que los sensores ópticos, de manera que los hacen menos apropiados para aplicaciones de monitorización durante largos periodos de tiempo.

En la tabla 4 se citan algunos de los principales sensores comerciales de oxígeno disuleto en agua marina.

Marca	Modelo	Rango	Exactitud	Resolución	Interfaz
Kongsberg	CONTROS HydroFlash O2	0 - 300 mbar pO_2	±1 %	< 0.1 %	RS-232C
Aanderaa	Optode 4835	0 – 1000 µM	<8 µM o 5%	<0.1 µM	RS-232
SeaBird	SBE64	120% de saturación superficial	± 3 µmol/kg	0.2 µmol/kg	RS-232
JFE Advanced	Rinko	0 to 200%	±2%	0.04%	RS-232C

Tabla 4. Sensores de oxígeno disuelto

2.3.5 Clorofila

La clorofila, como molécula de pigmento, tiene capacidad de absorción de determinadas longitudes de onda. Tras la absorción de energía

permanecen en estado excitado, para volver a su estado fundamental puede emitir radiación lumínica por medio de un proceso de fluorescencia.

Los sensores empleados para la determinación de la concentración de fitoplancton en agua de mar se basan, por un lado, en la emisión de luz próxima al ultravioleta que excita a las moléculas de clorofila fitoplanctónica, y por otro lado, en la medida de la potencia de emisión de la longitud de onda de fluorescencia emitida por la clorofila tras la excitación. Se trata por tanto de una medida cuantitativa de la concentración de clorofila por fotoluminiscencia. La concentración de clorofila modula la potencia de luz recibida en el sensor tras la excitación. A bajas concentraciones de clorofila, la modulación de la potencia lumínica de la fluorescencia presenta un comportamiento que se puede aproximar como lineal con la concentración de clorofila.

Las longitudes de onda de excitación y de emisión de fluorescencia habitualmente empleadas en los sensores de clorofila se centran en torno a los 470 y los 695 nm respectivamente.

Marca	Modelo	Rango	Exactitud	Sensitividad	Interfaz
SeaPoint Sensors	Chlorophyll Fluorometer	0 a 150 µg/l	ND	0.033V/µg/l	0 – 5 V_{DC}
WetLabs	ECO FL	0–125 µg/L	ND	0.02 µg/L	RS232
Turner Designs	Cyclops 7F-c	0-500 µg/L	ND		0 – 5 V_{DC}

Tabla 5. Sensores de clorofila

2.3.6 Turbidez

Los sensores de turbidez generalmente emplean la técnica de nefelometría para la determinación del grado de turbidez en el agua marina. Se basan en la medida de la luz dispersada por las partículas sólidas en suspensión en el agua.

La longitud de onda de la radiación empleada se debe seleccionar para minimizar efectos de absorción en el medio. Generalmente emiten una excitación lumínica en la región infrarroja y miden la luz dispersada por las partículas en suspensión que se encuentran en el entorno del sensor.

Marca	Modelo	Rango	Exactitud	Sensitividad	Interfaz
SeaPoint Sensors	Turbidity meter	0 a 25 FTU	ND	200mV/FTU	0 – 5 V_{DC}
WetLabs	ECO NTU	0 a 125 NTU	ND	0.02 NTU	RS232
Turner Designs	Cyclops 7F-T	0-1,500 NTU	ND	0.05 NTU	0 – 5 V_{DC}

Tabla 6. Sensores de clorofila

2.3.7 Sensores meteorológicos

En observatorios marinos flotantes, el espacio físico disponible para la instalación de instrumentación suele ser uno de los principales factores limitantes. Debido a esto, las estaciones meteorológicas integradas son, generalmente, la opción elegida para la adquisición de valores de parámetros meteorológicos en este tipo de plataformas.

Las estaciones meteorológicas integradas suelen contar con capacidad para la medida de la velocidad y dirección del viento, temperatura y humedad relativa del aire y presión atmosférica.

En estas estaciones, la medida de la velocidad y dirección del viento se realiza por medio de sensores acústicos activos. Estos sensores cuentan con tres transductores acústicos enfrentados entre sí dentro de un plano horizontal y localizados cada uno a la misma distancia de los otros dos, formando ángulos de 120º entre ellos.

	Marca/Modelo		
	Vaisala WXT520	Airmar PB200	Gill InstrumentsMetPak
Velocidad del viento			
Rango	0 a 60 m/s	0 a 40 m/s	0-60m/s
Exactitud	≤ ±5% o ±0,3 m/s	≥ 0,5m/s	±2%@12m/s
Resolución	0,1m/s	0,5 m/s	0.01m/s
Dirección del viento			
Rango	360°	360°	360°
Exactitud	±3%	5° RMS	±3° @12m/s
Resolución	0,1°	0,1°	1°
Temperatura del aire			
Rango	-52°C a 60°C	-25 °C a 55°C	-50°C to +100°C
Exactitud	±0,3°C (a 20 °C)	±1°C	±0.1°C
Resolución	0,1°C	0,1°C	0.1°C
Presión atmosférica			
Rango	600 hPa a 1100 hPa	850 hPa a 1150 hPa	600-1100hPa
Exactitud	≤ ±1 hPa	±2 hPa	±0.5hPa
Resolución	0,1 hPa	0,1 hPa	0.1hPa
Humedad relativa			
Rango	0% hasta 100% RH	Sensor no disponible	0-100%
Exactitud	≤ ±5% RH		±0.8% @ 23°C
Resolución	0,1% RH		0.1% RH

Tabla 7. Estaciones meteorológicas integradas

Las medidas de velocidad y dirección del viento se efectúan haciendo uso de señales acústicas ultrasónicas. Cada transductor emite y recibe señales. Las medidas se obtienen tras el procesado de los tiempos que tardan las señales acústicas en viajar entre cada uno de los transductores.

En lo que respecta a la determinación de la temperatura y humedad relativa del aire y debido a la relación entre estos dos parámetros, los sensores de ambos parámetros suelen estar montados en un único instrumento. En el caso de las estaciones integradas se presentan integrados con el resto de sensores.

La medida de la humedad relativa se suele basar en el empleo de sensores capacitivos con un dieléctrico con capacidad higroscópica, de manera que la capacidad varía en función del contenido de moléculas de agua presente en el dieléctrico. La medida de la temperatura del aire, por su parte, se basa en resistencias variables con la temperatura.

Respecto a la medida de la presión atmosférica existen diversas tecnologías de transductores. Las principales tecnologías empleadas se basan en transductores que aprovechan el efecto piezoeléctrico o el uso de una membrana a modo de diafragma que transmite las variaciones de presión a un capacitor variable con la presión.

La tabla 7 muestra las principales características de los sensores de algunas de las estaciones integradas más comunes en los observatorios a nivel global.

3. Diseño e implementación

3.1 Diseño del sistema

El diseño del sistema datalogger para cumplir los objetivos de este TFM se muestra en la Figura 3 en forma de diagrama de bloques a alto nivel de abstracción.

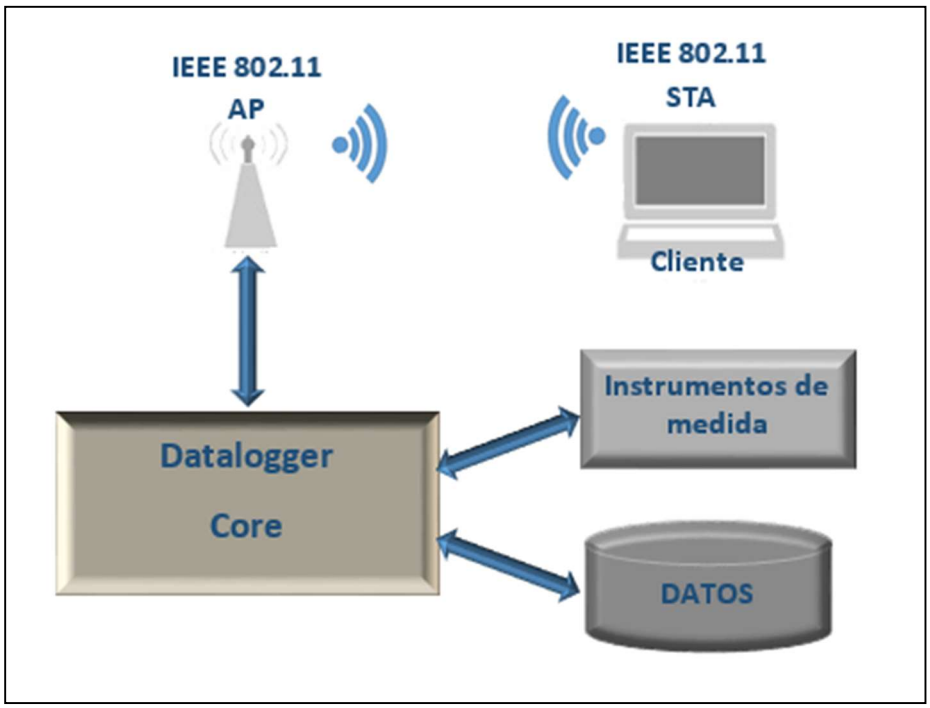


Figura 3. Diagrama de bloques del sistema diseñado

El sistema constituye un dispositivo *datalogger* para adquirir datos de instrumentos de medida de parámetros oceanográficos y meteorológicos. El *datalogger* diseñado tiene capacidad para adquirir datos de hasta ocho instrumentos de medida de manera concurrente en el tiempo. Los instrumentos de medida se conectan al *datalogger* por medio de puertos serie asíncronos según el protocolo RS-232C.

El *datalogger* tiene capacidad para almacenar los datos adquiridos en una memoria interna del propio dispositivo.

El diseño del sistema incluye la capacidad de establecer comunicaciones inalámbricas con una aplicación cliente del *datalogger*. La aplicación cliente incluye la interfaz gráfica que permite al usuario interactuar con el *datalogger*.

Las acciones que puede efectuar el usuario a través del cliente son la configuración de la adquisición de datos del *datalogger*, la visualización y almacenamiento en un directorio local de los datos adquiridos en tiempo

real, la descarga de los datos almacenados en la memoria del *datalogger*, y el borrado de la memoria del *datalogger*.

Las acciones de configuración que el usuario puede efectuar a través del cliente se citan a continuación:

- Seleccionar, de entre los disponibles, los instrumentos de los que se desea tomar datos.

- El establecimiento de un intervalo de tiempo de adquisición para cada instrumento seleccionado.

- El establecimiento del número de lecturas de datos efectuadas en cada adquisición. El *datalogger* puede efectuar la cantidad de medidas que el usuario indique, en cada periodo de adquisición.

- Procesado de datos en cada adquisición. Con el conjunto de datos leídos en cada adquisición, el usuario puede configurar al *datalogger* para que efectúe procesamiento de cálculo de media o mediana y almacene el resultado como producto de la adquisición y/o lo envíe al cliente como dato en tiempo real.

- Establecimiento de fecha y hora del *datalogger*. El *datalogger* dispone de un reloj de tiempo real (RTC-*Real Time Clock*). El usuario puede establecer la fecha y hora del RTC, cuyo valor será tomado como referencia horaria que acompaña a los datos adquiridos.

- Inicio de adquisición de datos. El usuario puede iniciar la adquisición de datos de forma instantánea, o puede programar al *datalogger* para que inicie la adquisición en una fecha y hora determinada.

3.1.1 Diseño del sistema *hardware*

La plataforma *hardware* sobre la que se desarrolla el *datalogger* es la CY8CKIT-062-WiFi-BT (23) de *Cypress Semiconductor* (Figura 4). Sus características más destacadas en relación al diseño del *datogger* se citan a continuación.

- Es un sistema basado en el PSoC 62 (*Programable System on Chip*) (24) desarrollado por *Cypress Semiconductor*, el PSoC 62 es un sistema de bajo consumo y alta capacidad de procesado diseñado para aplicaciones IoT.

- El PSoC 62 cuenta con dos núcleos ARM-Cortex-M de 32 bits: un procesador principal ARM-Cortex M4 (25), y un procesador ARM-Cortex M0+ (26) como CPU secundaria, para tareas de menores consumo y necesidad de cómputo.

- 1MB de memoria *flash* interna para programa de aplicación.

- Memoria S25FL512S (27). 512 Mb de memoria externa QSPI *flash* para el almacenado de datos.

- Módulo de comunicaciones inalámbricas 1DX LBEE5KL1DX (28) de *Murata*. Para la banda de 2,4 GHz según el estándar IEEE802.11 b/g/n 65 Mbps.

- Antena integrada.

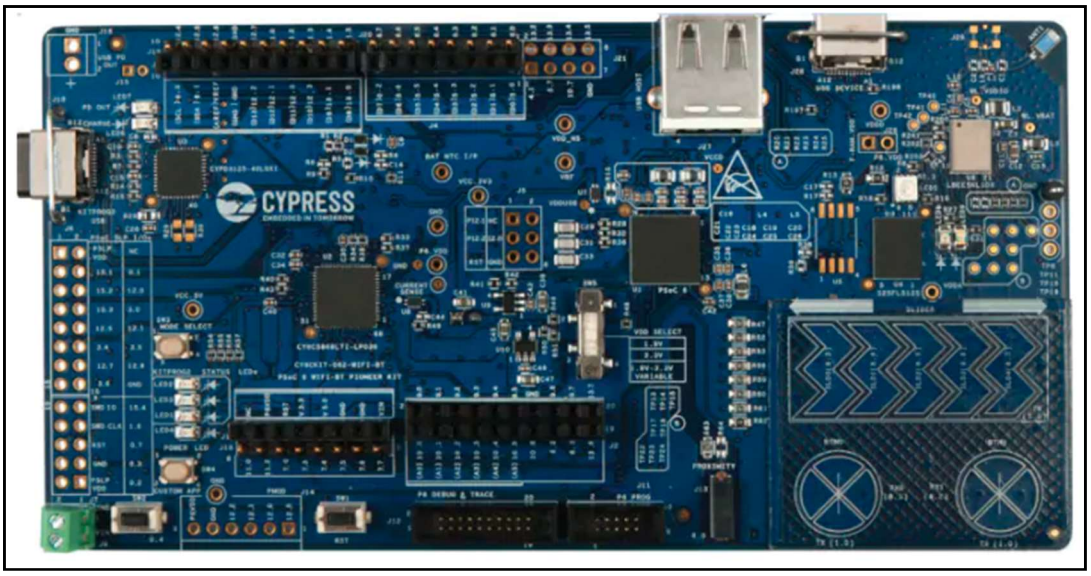


Figura 4. Tarjeta CY8CKIT-062-WiFi-BT

3.1.1.1 Microprocesadores

El PSoC 62 tiene integrado un procesador *ARM Cortex M4* y un procesador *ARM CORTEX M0+.* Ambos procesadores están basados en una arquitectura *Hardvard* y ejecutan el set de instrucciones *Thumb* de ARM.

El procesador M4 tiene integrada una unidad de punto flotante que cumple el estándar IEEE754, con capacidad de suma, resta, multiplicación, división y raíz cuadrada, que dota de mucha mayor capacidad de cálculo al microprocesador.

Para la gestión de la priorización de las interrupciones y la optimización de la latencia durante la gestión de las interrupciones, el sistema tiene implementado un vector anidado de control de interrupciones NVIC. En este sistema de gestión de interrupciones cada fuente de interrupción habilitada cuenta con un nivel de prioridad definido. Existen 256 niveles de prioridad programables para cada una de las fuentes de interrupción. Ambos procesadores cuentan con un módulo WIC (*Wakeup Interurpt Controller*) que permite al procesador salir de un estado de latencia ante la llegada de una interrupción determinada.

Tanto el M4 como el M0+ pueden integrar un bloque de protección de memoria MPU, el cual establece diferentes permisos de acceso a bloques de memoria determinados.

Ambas CPUs comparten una única unidad de memoria *flash*. Esta unidad de memoria destina 1024 kB para el almacenamiento de *firmware* de aplicaciones de usuario. Dentro del propio dispositivo de memoria se destinan otros 64 kB para el almacenamiento de datos e instrucciones relativos a comandos de arranque del PSoC, configuración del dispositivo e información propia de *Cypress.*

3.1.1.2 Líneas de entrada/salida

En el PSoC 62 las líneas GPIO (*General Purpose Input Output*) que están conectadas a los pines del circuito integrado son programables y pueden dar salida o entrada a diferentes señales o bloques del sistema. A nivel *hardware*, la selección de la función de cada una de las GPIO se efectúa por medio de un conjunto de multiplexores de alta velocidad denominado HSIOM (*High-Speed I/O Matrix*). Antes de la conexión con el pin externo, cada GPIO está conectada a una célula de entrada/salida **¡Error! No se encuentra el origen de la referencia.**).

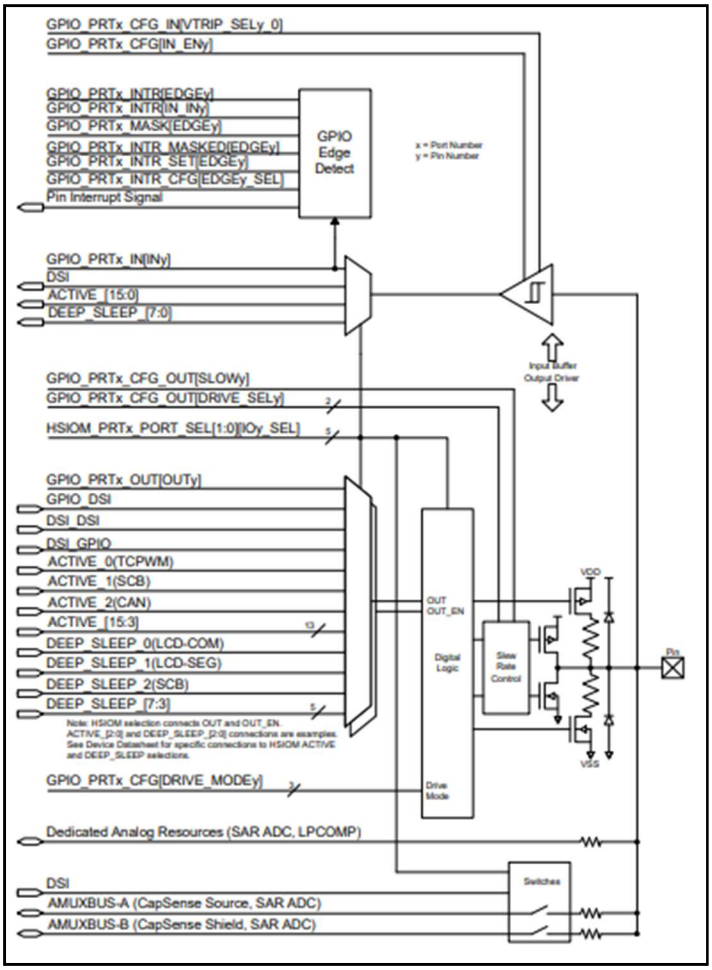


Figura 5. Etapa de entrada/salida de las GPIO (Cypress Semiconductor)

Las células de entrada/salida son configurables y constituyen la etapa de entrada o salida de cada uno de los pines GPIO. Contiene un *buffer* de alta impedancia de entrada para almacenar el estado de la señal aplicada en el caso de estar configurado como entrada. Como etapa de entrada cuenta, además, con un bloque de detección de flancos que está asociado a la capacidad de cada GPIO de generar una interrupción que dispara la ejecución de una rutina de servicio (ISR) programable por el usuario.

Como etapa de entrada/salida, la célula tiene implementado un sistema electrónico configurable en ocho modos de entrada/salida, este sistema configurable está directamente conectado a los pines del circuito integrado. Los estados disponibles de configuración de la etapa de salida son los siguientes:

- Alta impedancia.
- Resistencia *pull-up.*
- Resistencia *pull_down.*
- Drenador abierto a nivel bajo.
- Drenador abierto a niel alto.
- Resistencias de *pull-up* y *pull_down.*
- Modo *strong*. Modo estándar de salida de señales digitales.

El modo *strong* de salida digital cuenta con un bloque de control de *slew rate*. El *slew rate* puede estar configurado en modo lento o en modo rápido. Por defecto la salida digital está configurada en modo rápido, el modo lento está indicado en aplicaciones en las que las EMI o el efecto *crosstalk* sean factores más importantes que el alcance de frecuencias elevadas en las señales de salida.

3.1.1.3 Sistema de reloj

El PSoC 62 dispone de las siguientes fuentes de reloj tanto internas como externas que, en función de la configuración establecida en cada aplicación, son empleadas para la sincronización de todos los bloques funcionales que lo constituyen:

- Oscilador principal interno de 8 MHz (IMO).

- Oscilador interno de baja velocidad (ILO), con una frecuencia nominal de 32.768 Hz. Se trata de un oscilador de bajo consumo y baja precisión.

- Oscilador interno de alta precisión y baja velocidad (PILO). Al igual que el oscilador ILO, el PILO tiene una frecuencia nominal de 32.768 Hz pero de mayor consumo que el ILO.

- *Watch Crystal Oscillator* (WCO) es una fuente de reloj de alta precisión con frecuencia nominal de 32.768 Hz generada por un

cristal de cuarzo externo al PSoC 62. Constituye la fuente primaria que alimenta al RTC del sistema PSoC.

La Figura 6 muestra el diagrama de bloques del sistema de fuentes de reloj del PSoC 62.

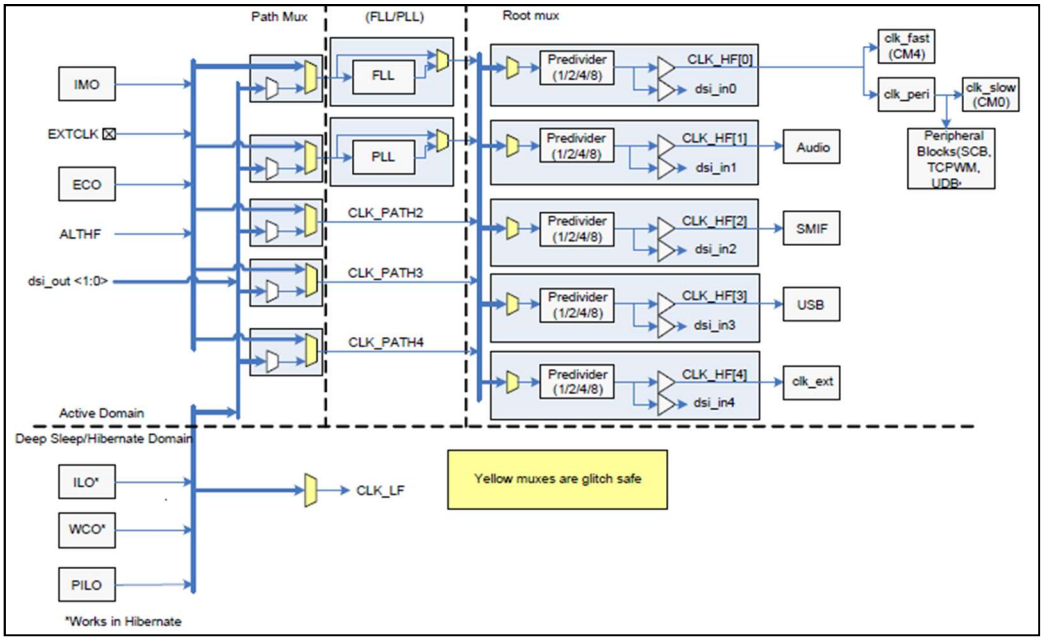


Figura 6. Diagrama de bloques del sistema de reloj (Cypress Semiconductor)

El sistema de reloj del PSoC 64 tiene implementados un módulo oscilador de bucle enganchado en fase (PLL) y un módulo oscilador de bucle enganchado en frecuencia (FLL), ambos de frecuencias de salida configurables y que son empleados para generar un amplio rango de frecuencias con las que alimentar tanto a los procesadores como a los dispositivos periféricos del PSoC 62.

El módulo FLL es capaz de fijar la frecuencia objetivo más rápidamente que el PLL además de tener menor consumo energético, como contrapartida el PLL es capaz de alcanzar mayor frecuencia de salida.

La señal *CLK_LF* es la suministrada, entre otros, al módulo RTC. La señal *clk_fast* es la señal de reloj del procesador M4, mientras que la señal *clk_peri* es la fuente de la que se obtienen las señales de reloj de todos los bloques periféricos y la señal *clk_slow* que se corresponde con la señal de reloj del procesador M0+.

En la aplicación implementada en este TFM, la fuente de las señales de reloj de los procesadores y los periféricos es el oscilador IMO, empleando el FLL como módulo intermedio para alcanzar la frecuencia final. El sistema está configurado para alimentar a los dos procesadores y a los bloques de periféricos con una señal de reloj de 100 MHz generada por el módulo FLL.

Esta configuración de reloj viene dada por el SDK WICED, la herramienta de desarrollo usada en la implementación del *datalogger* y cuyas principales características se indican en el apartado de implementación del *firmware* de este documento.

3.1.1.4 Interfaz serie de memoria externa

El PSoC 62 tiene integrado un módulo *Serial Memory Interface* (SMIF) con capacidad para controlar, como *master,* hasta cuatro módulos de memoria serie externos configurados como *slave*.

Las comunicaciones entre el SMIF y las memorias externas se efectúan por medio de una interfaz serie síncrona SPI que puede ser configurada para trabajar con protocolo *simple, dual, quad* y *octal.*

Para la transmisión y recepción de comandos y datos, el módulo SMIF tiene implementados tres *buffers* FIFO cuya función principal es la de proporcionar la capacidad de efectuar las transferencias de datos y comandos entre bloques con diferentes dominios de reloj de forma asíncrona.

Uno de los FIFO, con capacidad para almacenar ocho entradas de un byte cada una, es empleado para la transmisión de datos a la memora. El segundo *buffer* FIFO, con capacidad para almacenar cuatro entradas de veinte bits cada una, es empleado en la transmisión de comandos. El tercero es el *buffer* de recepción de datos desde la memoria, con capacidad para guardar ocho entradas de un byte cada una.

En la aplicación diseñada e implementada en este TFM se emplea una memoria *flash* externa para el almacenamiento de los datos adquiridos por el datalogger. La SMIF se configura con el SPI en modo *quad*, esto es, con cuatro líneas serie de datos, una línea de reloj de sincronismo y una línea de *chip select,* activa a nivel bajo (figura 7).

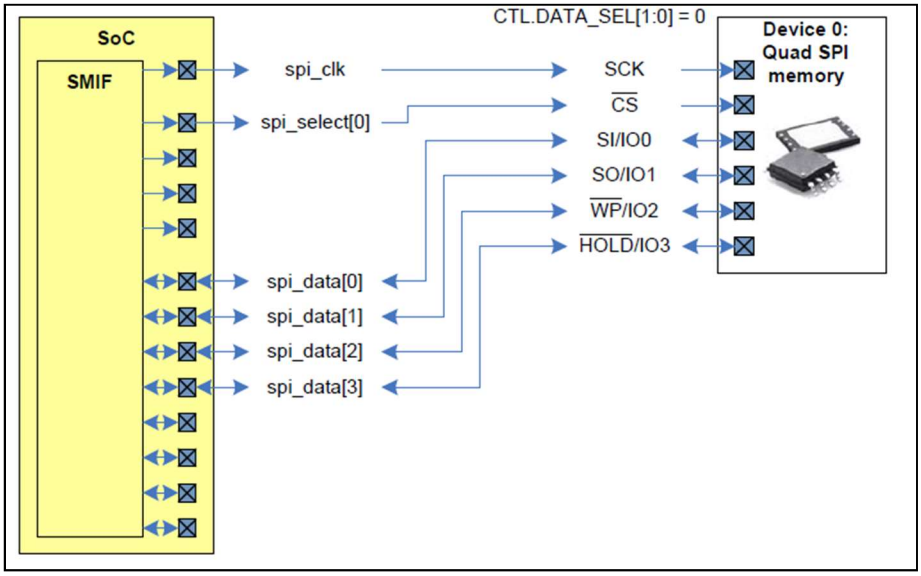


Figura 7. Interfaz QSPI de la memoria flash externa (Cypress Semiconductor)

La interfaz *quad* SPI tiene la capacidad de transferir un byte en dos ciclos del reloj de sincronismo, transfiriéndose en el primer ciclo de reloj cada uno de los cuatro bits más significativos del byte por las líneas IO3, IO2, IO1 e IO0 respectivamente, y en el segundo ciclo de reloj se transfieren los cuatro bits menos significativos por las mismas líneas IO (**¡Error! No se encuentra el origen de la referencia.**).

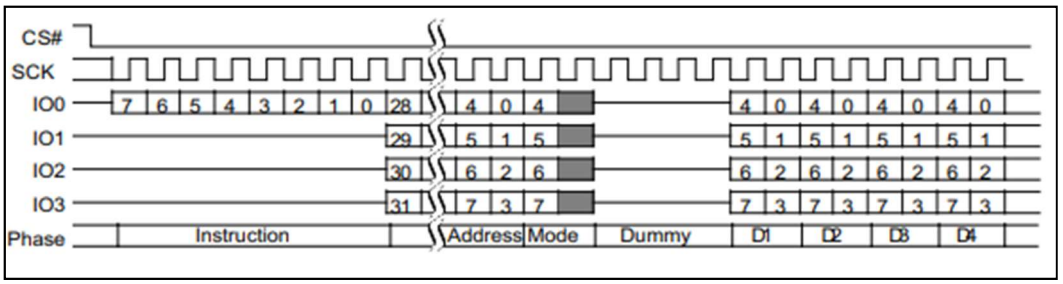


Figura 8. Ejemplo de acceso a memoria Quad SPI (Cypress Semiconductor)

En el datalogger implementado en este TFM la señal de reloj del SMIF es la resultante de la salida del módulo FLL (Figura 6) con el divisor programado en 2, de modo que la frecuencia del reloj del *quad* SPI es de 50 MHz.

3.1.1.5 Memoria de datos externa

El módulo de memoria empleado para el almacenamiento de los datos adquiridos por el datalogger es el S25FL512S de *Cypress Semiconductor*.

Se trata de un sistema de memoria flash con 512 bits de capacidad de almacenamiento implementada células tipo NOR fabricadas siguiendo la tecnología *MirrorBit*. La tecnología *MirrorBit* permite el almacenamiento de dos bits en cada célula de memoria frente a la capacidad de almacenamiento de un bit en las células tradicionales NOR de puerta flotante, aumentando de esta manera la densidad de datos en el circuito integrado y manteniendo la mayor velocidad de acceso a los datos de la tecnología de memorias de células NOR en relación a las memorias de células NAND.

3.1.1.6 Interfaz WiFi

El módulo encargado del control de la red WiFi del sistema *datalogger* es el LBEE5KL1DX de Murata. Se trata de un sistema creado específicamente para su empleo en aplicaciones embebidas de bajo consumo. Está diseñado para minimizar el consumo energético y se basa en el CYW4343W de *Cypress Semiconductor* (29).

El CYW4343W implementa en un solo sistema un transreceptor de 2.4 GHz con soporte para los estándares WLAN IEEE 802.11 b/g/n, y soporte para *Bluetooth* v4.1 (BR/EDR/BLE). Incluye un procesador ARM Cortex M3 y 512 kB de memoria SRAM y 640 kB de memoria ROM destinados a dar soporte a las operaciones del *driver* WLAN.

El módulo CYW4343W tiene integrados los filtros, etapas de ganancia y moduladores/demoduladores necesarios para la implementación completa del IEEE 802.11 b/g/n.

La implementación MAC del CYW4343W incluye una máquina de estados programable encargada del control a bajo nivel del *hardware* del sistema para la ejecución del IEEE 802.11 b/g/n. Un módulo específico para la encriptación de la información, provisto de la ejecución de los algoritmos WEP, WPA TKIP, y WPA2 AES-CCMP para el cifrado de las comunicaciones, se encarga de interactuar con los módulos de envío y recepción para encriptar, desencriptar validar y codificar el MIC (*Message Integrity Check*).

A nivel MAC, los módulos de transmisión y recepción interactúan con el DMA integrado para la gestión del acceso a los *bufferes* de memoria de transmisión y recepción. Estos módulos integran, además filtros programables de tramas de datos recibidos, programables por criterios como la dirección del receptor, el tipo de trama o el BSSID.

La interfaz de conexión entre el PSoC 62 y el LBEE5KL1DX para el sistema WLAN sigue especifiación SDIO 3.0 (30). El PSoC 62 no dispone de un bloque periférico específico para funcionar como un bus SDIO, sin embargo está implementado en el SDK de WICED a modo de librería de *firmware*. La librería de WICED hace uso de seis líneas genéricas GPIO para la implementación del bus. La capa física del bus emplea cuatro líneas para el trasiego de datos (P2.0, P2.1, P2,2 y P2.3 del PSoC 62), una línea de comandos (P2.4) y la línea de reloj (P2.3) (Ver anexo B).

La frecuencia del reloj del bus SDIO implementado es un cuarto de la frecuencia de la señal *clk_peri* (figura 6), en la configuración *hardware* del datalogger *clk_peri* es una señal de 100 MHz, por lo que la señal de reloj del SDIO es de 25 MHz.

3.1.1.7 Puerto serie RS232-C

Como se puede observar en el apartado destinado a sensores oceanográficos y meteorológicos del capítulo 2 de esta memoria, los principales fabricantes de emplean protocolo RS232-C como interfaz de comunicación con su instrumentación.

El estándar RS232-C especifica una interfaz de comunicaciones serie asíncronas entre dos dispositivos empleando como mínimo tres líneas para el trasiego de datos: dos líneas para la transmisión y recepción de datos entre los dos terminales (Tx y Rx) y una tercera línea de tensión de referencia o *GND*.

En RS232-C ambos terminales deben tener la misma configuración para ejecutar el transporte de datos. La configuración incluye el *baud rate* que se corresponde con la frecuencia de envío de los bits, el número de bits

de datos de cada trama, el número de bits de *stop*, si se incluye o no un bit de paridad y en caso afirmativo el tipo de paridad y si hay o no control de flujo y de qué tipo.

El estándar RS232-C establece además los niveles de tensión de los niveles lógicos, siendo un '1' lógico un valor de tensión de entre -3 y -25 V, y un '0' lógico entre entre 3 y 25 V. Todos estos valores referidos a la línea de referencia o *GND* de la interfaz RS232-C.

La figura 9 muestra la composición de una trama de bits con niveles de tensión LVTTL. La trama está configurada con un *baud rate* de 9600, con un bit de paridad, ocho bits de datos y un bit de stop. En estas tramas el bit de *start* siempre es un '1' lógico y los bits de stop son '0' lógicos.

El PSoC 62 trabaja con niveles de tensión CMOS/LVTTL. Para trabajar con el estándar RS232-C es necesario añadir un sistema *hardware* adaptador de los niveles de tensión CMOS/LVTTL a los definidos en RS232-C. Los dispositivos adaptadores seleccionados para el datalogger son los *RS-232C Click* de *Mikroe* (31). Estos son sistemas basados en el circuito integrado MAX3232 (32) con capacidad para ser alimentados a 3,3 Vdc y que hace uso de una bomba de carga (33) para alcanzar niveles de tensión dentro del rango RS232-C a partir de los 3,3 Vdc de alimentación.

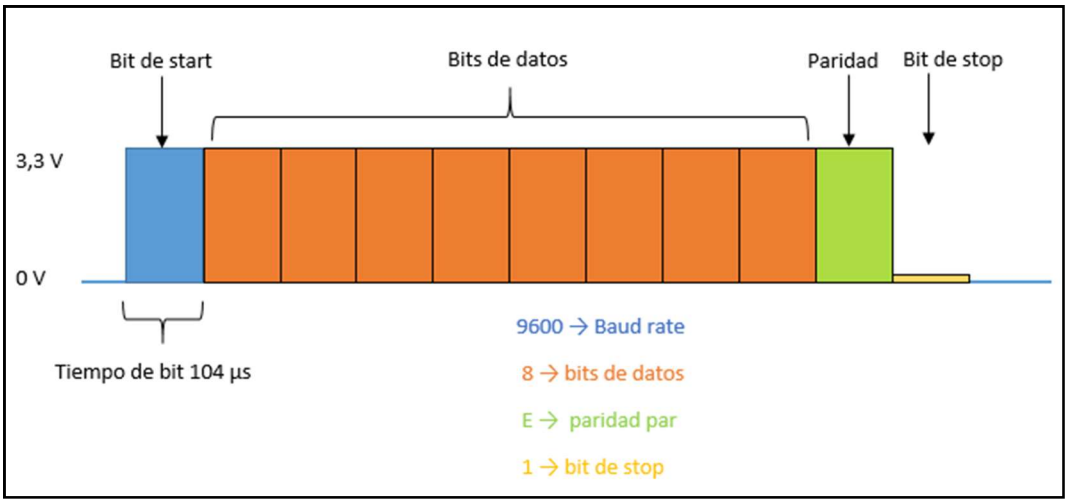


Figura 9. Trama del puerto serie asíncrono 96008E1

El SDK de WICED tiene implementados siete UARTS para el PSoC 62 que pueden ser configuradas para trabajar siguiendo la especificación RS232-C, sin embargo no todos están disponibles en la tarjeta CY8CKIT-062-WiFi-BT, ya que algunas líneas GPIO están ocupadas en otros módulos de la tarjeta o no están accesibles en el *pinout* de la tarjeta.

Con el objetivo de disponer de los puertos de comunicaciones establecidos en el diseño del *datalogger* se ha diseñado e implementado en el *firmware* del *datalogger* un módulo para hacer trabajar a dos líneas

GPIO del PSoC 62 como líneas de transmisión y recepción de datos siguiendo la norma RS232-C (ver apartado 3.2.1.6).

Finalmente y a modo de resumen de los módulos citados hasta aquí, la figura 10 muestra un diagrama con los módulos *hardware* que componen el *datalogger* y las relaciones entre ellos.

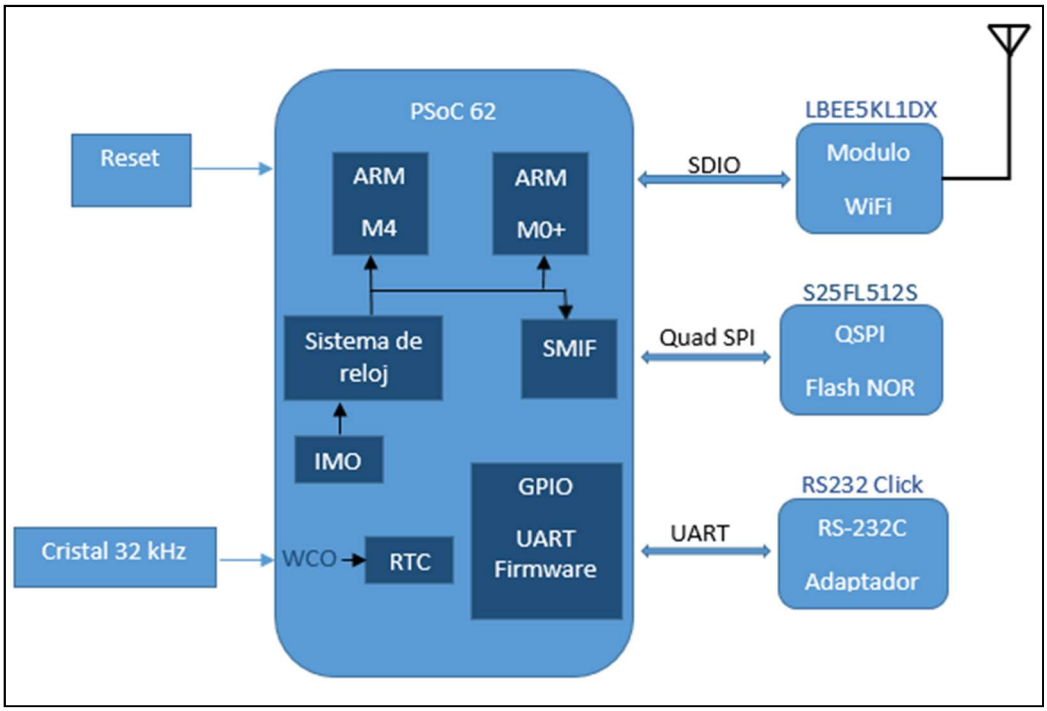


Figura 10. Relaciones entre los módlos del datalogger

3.1.1.8 Sistema de alimentación

La tarjeta CY8CKIT-062-WiFi-BT proporciona cuatro opciones diferentes para ser alimentada a diferentes niveles de tensión:

- Alimentación a través del conector USB tipo C de 5, 9 o 12 Vdc hasta 3 A.

- Alimentación a través del conector compatible con Arduino UNO R3 o del conector externo de tensión variable J9 (Ver anexo B) de 5 a 12 Vdc hasta 2 A.

- Alimentación por medio de una batería recargable de Li-Po a 3,7 Vdc (Ver anexo B).

- Alimentación a través del conector de programador/depurador externo a 5 Vdc.

El rango de voltaje de alimentación entre 5 y 12 Vdc dota al sistema de mayor capacidad de integración en plataformas con diferentes tensiones de trabajo. Cuando la tarjeta está siendo alimentada con una tensión de entre 5 y 12 Vdc tiene la opción de cargar una batería Li-Po externa, de la

que puede alimentarse cuando la alimentación principal no este disponible. El dispositivo cargador de batería es un BQ24266RGER (34). La batería recomendada es de Li-Po de 3,7 V y 850 mAH.

El sistema de alimentación de la tarjeta está compuesto por un conversor *book-bost* al que llegan las tensiones de alimentación externa de entre 5 y 12 Vdc y suministra una tensión estable de salida de 5 Vdc. Un regulador de tensión con salida de 3,3Vdc y hasta 600 mA genera la alimentación para los módulos PSoC 62, memoria *flash* externa y *WiFi*, además de proporcionar la alimentación del conector J1 al que se conectan los módulos *RS232 Click* (Ver anexo B).

3.1.2 Diseño del sistema *firmware*

3.1.2.1 Herramienta de desarrollo

En el desarrollo del *firmware* del *datalogger* se ha empleado la herramienta WICED Studio (*Wireless Internet Connectivity for Embedded Devices*) (35) de *Cypress Semiconductor*. WICED Studio es un entorno basado en el IDE *Eclipse* que incluye un SDK diseñado dar soporte a la creación de aplicaciones embebidas con conectividad *WiFi* y *Bluetooth*.

La API de WICED Studio proporciona un amplio rango de funciones que son agrupadas en componentes. Entre los componentes de WICED, empleados en el desarrollo del *datalogger*, se citan los siguientes:

- *Platform functions*: incluye funciones para el control de las GPIO. Estas funciones son usadas en las líneas de comunicaciones con los sensores e instrumentos externos.

- *Management*: funciones para la inicialización del sistema, de la interfaz de red, entre otras.

- *WiFi* (IEEE 802,11): funciones para la inicialización de un punto de acceso a la red inalámbrica.

- *IP Communication*: gestión del servidor TCP y las comunicaciones por *sockets*.

- RTOS (*Real Time Operating System*): funciones para la gestión de tareas ejecutadas en paralelo o *threads*.

3.1.2.2 Sistema operativo

Entre las necesidades del dispositivo *datalogger* diseñado se encuentran el control de la temporización precisa de la adquisición de datos de los sensores y la capacidad de que estas adquisiciones se puedan ejecutar de manera concurrente en el dominio temporal. La ejecución de las tareas de adquisición debe ser prioritaria en relación con otras tareas menos críticas en el desempeño general del sistema, y proporcionar seguridad

en los posibles accesos concurrentes a recursos compartidos, como la escritura de datos en memoria.

Estas características en un sistema embebido las aportan los sistemas operativos en tiempo real.

Un RTOS en un sistema embebido proporciona herramientas para dotarlo con la capacidad de responder ante eventos externos que pueden ser asíncronos, con tiempos de respuesta críticos y necesidades de acceso concurrente a recursos compartidos. Un RTOS mantiene una lista de tareas que ejecuta en modo paralelo asignando tiempo de ejecución de microprocesador en función de las prioridades de cada una de las tareas, estas tareas genéricamente son denominadas *threads*.

Entre los RTOS existen dos grandes grupos de sistemas en función del control realizado sobre los tiempos asignados para la ejecución de las diferentes tareas concurrentes o *threads*: los sistemas de multitarea apropiativa y los sistemas de multitarea colaborativa.

En un RTOS de multitarea apropiativa, cuando un *thread* de un nivel de prioridad superior al que se está ejecutando en ese momento necesita ser ejecutada, se suspende el de menor prioridad para pasar a ejecutar el de mayor prioridad.

Por su parte en un RTOS de multitarea colaborativa todos los *threads* que se encuentran en estado de ejecución comparten tiempo de microprocesador sin que ninguno tenga prioridad sobre los otros. En este caso el control del *thread* que se ejecuta en cada instante reside en los propios *threads*. cada *thread* debe ser codificado teniendo en cuenta que cada cierto tiempo debe ceder el control del microprocesador al resto de *threads*.

El núcleo del RTOS empleado en el desarrollo del *datalogger* es el sistema operativo *ThreadX* (36), del que WICED proporciona licencia para su empleo en los desarrollos que usan su SDK.

En WICED, el RTOS es un sistema híbrido entre los dos grandes grupos antes citados. Para *threads* del mismo nivel de prioridad se comporta como un sistema colaborativo, pero entre *threads* de diferente nivel de prioridad es un sistema apropiativo.

La API del componente RTOS de WICED proporciona herramientas para garantizar la seguridad en la coordinación entre *threads* y el acceso concurrente a recursos compartidos.

Estos recursos son los semáforos que permiten arrancar la coordinación en la ejecución de un *thread* desde otro *thread*, los retardos (*rtos_delay*) en los que los *threads* no necesitan tiempo de microprocesador y este es asignado a otros *threads* que están en espera, o los *mutex*, empleados para evitar conflictos en el acceso a recursos compartidos entre los

threads, de manera que cuando un *thread* va a acceder a un recurso compartido bloquea el *mutex* asociado a este recurso y lo desbloquea una vez finalizado el acceso al recurso. Si el *mutex* esta siendo bloqueado por otro *thread* la ejecución del último *thread* en acceder al recurso pasa a modo suspendido hasta que el *mutex* queda desbloqueado.

3.1.2.3 Control de procesos y modos de funcionamiento

El *datalogger* diseñado cuenta con dos modos de funcinamiento: administración y adquisición.

Cuando el *datalogger* está en modo administración, permite al usuario seleccionar los sensores o instrumentos de los que desea obtener datos así como su intervalo de medida, el número de medidas que desea tomar en cada adquisición y si desea efectuar algún procesado de los datos medidos en cada adquisición. Los procesados disponibles son la obtención de la media y la mediana del conjunto de valores

Además en modo administración es posible sincronizar la hora del *datalogger*, descargar los datos almacenados en memoria, borrar la memoria y comenzar en modo inmediato una nueva adquisición o programar la fecha y hora de inicio de una nueva adquisición.

En modo adquisición, el *datalogger* toma datos según la configuración introducida por el usuario cuando estaba en modo administración. En este modo la única interacción que el *datalogger* recibe por parte del usuario es la finalización de la adquisición y el paso al modo administración.

Los diferentes procesos que forman parte de estos dos modos de funcionamiento se estructuran en *threads* con diferentes niveles de prioridad que se ejecutan en el seno del RTOS proporcionado por el SDK WICED. En el diseño del *firmware* del *datalogger*, los *threads* se sitúan en cuatro niveles de ejecución. Los *threads* de los niveles superiores inicializan y controlan el estado de los *threads* de nivel inferior. Los *threads* agrupados en cada uno de los niveles tienen el mismo nivel de prioridad entre ellos, y los *threads* situados en los niveles inferiores tienen prioridad sobre los situados en los niveles superiores.

3.1.2.4 Diseño de la memoria de datos

La memoria externa S25FL512S está conectada con el PSoC 62 por medio de un puerto SPI, para el envío de comandos y datos. Cuenta con una capacidad de 64 MB para el almacenamiento de datos y, en ocasiones, de programa o recursos propios de la aplicación. Los proyectos basados en el SDK de WICED guardan parte de los recursos necesarios para su ejecución en esta memoria, permitiendo el libre acceso al resto de la memoria para el almacenamiento de datos por parte de las aplicaciones en ejecución.

En la aplicación del *datalogger* se reservan los primeros 16 MB de memoria para almacenamiento de recursos propios del SDK WICED. Los 48 MB restantes son empleados como memoria de datos del *datalogger*.

Toda la información almacenada en la memoria de datos está codificada en forma de caracteres ASCII, en donde cada línea de datos está delimitada en ambos extremos por el carácter '\n'.

La figura 11 muestra el esquema diseñado de la distribución del espacio de memoria disponible como memoria de datos del *datalogger*.

Los últimos cinco bytes de la memoria están reservados por el diseño del formato propio del *datalogger* y son empleados para el control de las operaciones de lectura y escritura. Se ha diseñado e implementado un módulo de *firmware* específico para el control de las operaciones de inicialización de la memoria, lectura y escritura de datos.

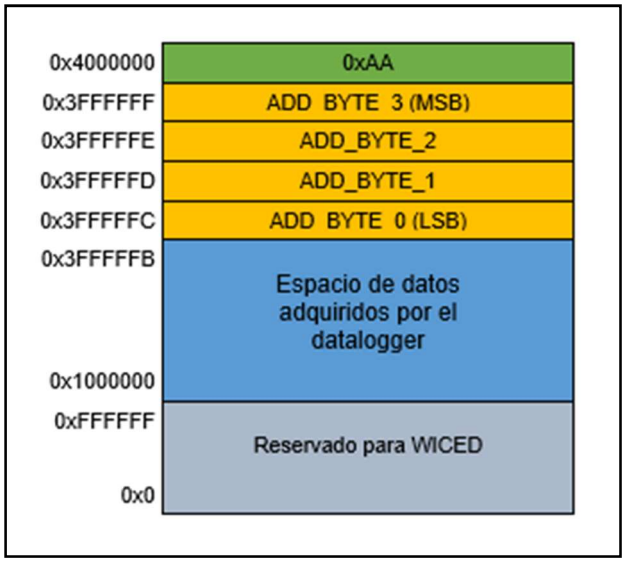


Figura 11. Mapeo de la memoria de datos del datalogger

De los cinco bytes reservados, el situado en la última posición de la memoria es empleado para chequear si la memoria está correctamente formateada para su funcionamiento en la aplicación del *datalogger*. Este byte de control de formato, en una memoria que ha sido formateada por el *firmware* del *datalogger*, siempre debe almacenar el valor 0xAA. Los cuatro bytes anteriores al byte de control de formateo contienen la dirección de memoria en la que se escribirá el próximo byte de datos adquirido por el *datalogger*, siendo el MSB de la dirección almacenada el inmediatamente anterior al byte de control de formateo. La dirección base del sector de la memoria destinada a datos es 0x1000000.

3.1.3 Diseño del cliente

El cliente del sistema *datalogger* incluye una GUI con la que el usuario interactúa para la configuración y recepción de datos del *datalogger*.

El diseño de la GUI está compuesto por la propia interfaz gráfica, un módulo de comunicaciones TCP para el establecimiento del enlace con el *datalogger*, y los correspondientes módulos de gestión de las comunicaciones con el *datalogger*.

El diseño del cliente está basado en el uso de varios módulos de *python*, los más importantes son el módulo *ttk* para la creación de la interfaz gráfica, el módulo *socket* para las comunicaciones TCP y el módulo *datetime* empleado en la gestión de horas y fechas.

3.1.4 Protocolo de comunicaciones entre el *datalogger* y el cliente

La comunicación entre el *datalogger* y la interfaz gráfica de usuario se establece en un entorno de red con una arquitectura modelo cliente/servidor, donde el *firmware* del *datalogger* actúa como servidor y el *software* de la interfaz gráfica de usuario actúa como cliente.

Los siguientes apartados contienen la descripción de los componentes del enlace. Esta descripción se basa en el modelo de capas TCP/IP.

3.1.4.1 Interfaz de red. Capa de acceso al medio

El soporte físico empleado en la comunicación entre el *datalogger* y la aplicación cliente es un enlace inalámbrico según el estándar IEEE 802.11g.

El módulo *WiFi/Bluetooth* LBEE5KL1DX, con capacidad para trabajar con 802.11 b/g/n a 65 Mbps con esquemas de modulación DSSS, CCK y OFDM es el encargado de la gestión de la capa de acceso al medio en el *datalogger*. Es configurado por el *firmware* del *datalogger* para generar un punto de acceso (AP) de una red *WiFi*, con los siguientes parámetros:

SSID: MetOcean Datalogger
Password: abcd1234
Seguridad: WPA2-PSK

El dispositivo en el que se ejecuta la aplicación cliente se debe conectar a la red *MetOcean Datalogger* para el establecimiento del enlace a nivel de capa de acceso al medio.

3.1.4.2 Capa de red

Emplea el protocolo de red IPv4. El punto de acceso creado por el *datalogger* tiene asignada la dirección IP fija de clase C 192.168.10.1. Por su parte, el *datalogger* cuenta con un servidor DHCP interno que asigna la dirección IP a la interfaz de red del equipo cliente que se conecte al punto de acceso de la red *MetOcean Datalogger*.

3.1.4.3 Capa de transporte

A nivel de capa de transporte se emplea el protocolo TCP. De esta manera se garantiza que los datos enviados por un nodo de la red, llegan al destino en el orden en que fueron enviados, con corrección de errores y control de flujo.

En el *firmware* del *datalogger* se ha configurado un servidor TCP que se encarga de la implementación del protocolo sobre la red *WiFi* creada.

El servidor TCP inicializa un *socket* TCP vinculado al puerto 7777 del *datalogger*, y define las funciones a ejecutar cuando el cliente se conecta, se desconecta y envía datos al puerto.

3.1.4.4 Capa de aplicación

A nivel de aplicación se ha diseñado específicamente para el *datalogger* un protocolo propio denominado *MetOcean protocol* cuya función es el establecimiento de una regla de codificación de los comandos y la información que fluyen entre el cliente y el *datalogger*. Su diseño se detalla a continuación.

MetOcean se sitúa en la capa de aplicación, dentro del modelo OSI. Está basado en palabras compuestas por cadenas de caracteres de tamaño variable. Cada palabra está formada por un comando principal y un comando secundario, separados por un carácter guión ('-'), y entre paréntesis y separados por punto y coma la información de los argumentos que acompañan a los comandos. Cada argumento está compuesto por un par "opción-valor" que indican el tipo de argumento y el valor del argumento. La figura 12 muestra la estructura de una palabra del protocolo de comunicaciones *MetOcean*.

Figura 12. Estructura de un comando del protocolo MetOcean

Donde:

- C_P: subcadena de caracteres numéricos que codifican el comando principal.
- C_S: subcadena de caracteres numéricos que codifican el comando secundario.
- T_{ARGN}: subcadena de caracteres numéricos que codifican el tipo de argumento de la tupla "N".
- Arg_N: subcadena de caracteres alfanuméricos que contienen el argumento de tipo T_{ARGN}.

Comando Primario	Comando Secundario	Argumentos	Rango de valores de los argumentos	
SET	INSTRUMENT	INSTRUMENT_ID	0,1,2,3,4,5,6,7. El número que identifica cada uno de los ocho instrumentos que pueden ser habilitados para la adquisición de datos.	
		TYPE_INSTRUMENT	SBE 37 SMP	Selecciona el tipo de instrumento conectado a la entrada ID
		ENABLED	0 → deshabilita la adquisición del instrumento. 1 → habilita la adquisición del instrumento	
		SAMPLE_INTERVAL	Establece el intervalo en segundos que debe transcurrir entre adquisiciones de datos del instrumento.	
		MEASUREMENTS_PER_SAMPLE	Establece el número de adquisiciones en cada periodo de adquisición.	
		PROCESSING	NONE MEDIAN AVERAGE	Establece el tipo de procesado a aplicar a cada conjunto de adquisiciones tomado en cada periodo de adquisición
	MODE	TYPE_MODE	CONFIGURING, ACQUIRING	
		SECONDS_UTC_ACQUIRE_t	Establece el número de segundos restantes hasta que el datalogger comience a adquirir datos en una adquisición programada en una fecha y hora dados.	
	DATE_TIME	YEAR_t	Establece el año actual en el RTC del datalogger.	
		MONTH_t	Establece el mes actual el en RTC del datalogger.	
		DAY_t	Establece el día actual en el RTC del datalogger.	
		HOUR_t	Establece la hora actual en el RTC del datalogger.	
		MINUTE_t	Establece los minutos actuales en el RTC del datalogger.	
		SECONDS_t	Establece los segundos actuales en el RTC del datalogger.	
GET	INSTRUMENT	INSTRUMENT_ID	0,1,2,3,4,5,6,7. Devuelve el número que identifica cada uno de los ocho instrumentos que pueden ser habilitados para la adquisición de datos.	
		TYPE_INSTRUMENT	SBE 37 SMP OPTODE 3835	Devuelve el tipo de instrumento conectado a la entrada ID
		ENABLED	0 → Indica que el instrumento está deshabilitado. 1 → Indica que el instrumento está habilitado	
		SAMPLE_INTERVAL	Devuelve el intervalo en segundos programados que transcurrirá entre adquisiciones de datos del instrumento una vez el datalogger entre en modo adquisición.	
		MEASUREMENTS_PER_SAMPLE	Devuelve el número de adquisiciones que se tomarán en cada periodo de adquisición cuando el datalogger esté en modo adquisición.	
		PROCESSING	NONE MEDIAN AVERAGE	Devuelve el tipo de procesado que se realiza sobre cada conjunto de adquisiciones tomado en cada periodo de adquisición.
	MODE	TYPE_MODE	CONFIGURING ACQUIRING	Devuelve el modo actual de funcionamiento del datalogger.
	DATE_TIME	SECONDS_UTC_t	Devuelve la fecha y hora actual del datalogger en segundos UTC	
	REALTIME_DATA	REAL_TIME_DATA_Arg	Devuelve una cadena de caracteres con los últimos valores adquiridos de los instrumentos habilitados cuando el datalogger está en modo adquisición en tiempo real.	
	MEMORY_DOWNLOAD	MEMORY_READ_ADDRESS	Contiene la dirección de comienzo del bloque de memoria a descargar.	
		MEMORY_READ_BUFFER_LENGTH	Ttamaño del bloque de memoria a descargar en bytes.	
	MEMORY_STATE	MEMORY_STATE_Arg	Devuelve el número de bytes de datos que contiene la memoria	

Tabla 8. Referencia de comados del protocolo MetOcean

La tabla 8 contiene los comandos primarios y secundarios y los argumentos definidos para cada uno de ellos.

Los comandos del tipo primario SET modifican algún parámetro del datalogger desde el cliente, mientras que los comandos del tipo primario GET son enviados por el cliente para obtener información desde el *datalogger*.

La figura 13 muestra a alto nivel los componentes del enlace de comunicaciones entre el *datalogger* y el cliente.

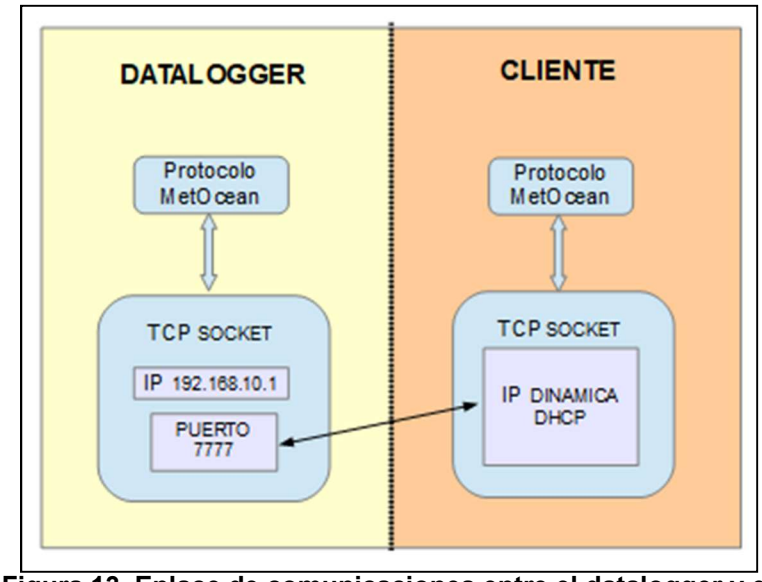


Figura 13. Enlace de comunicaciones entre el datalogger y el cliente

Las figuras 14 a 18 muestran los flujos de comandos del protocolo *MetOcean* entre el *datalogger* y el cliente con indicaciones de las acciones ejecutadas en el cliente y *datalogger* ante cada comando.

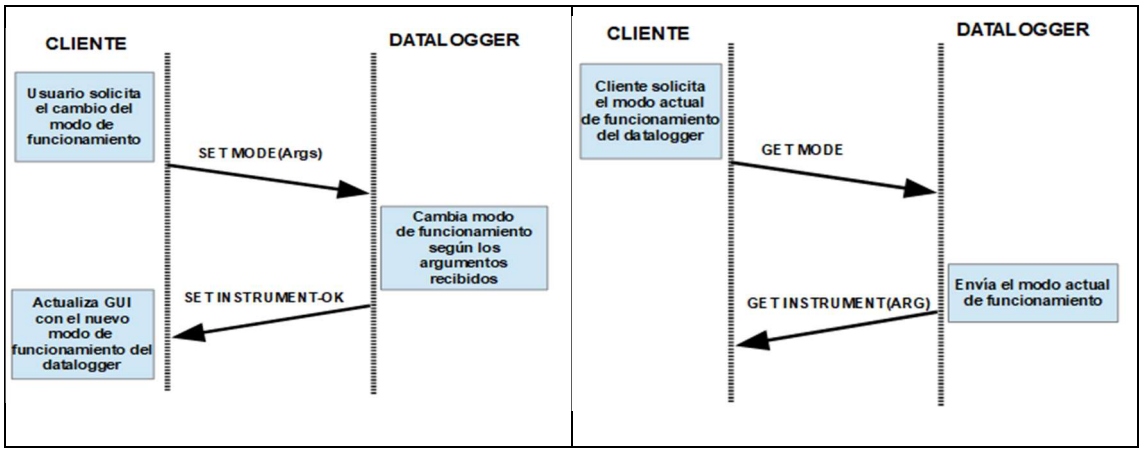


Figura 14. Comandos SET y GET MODE

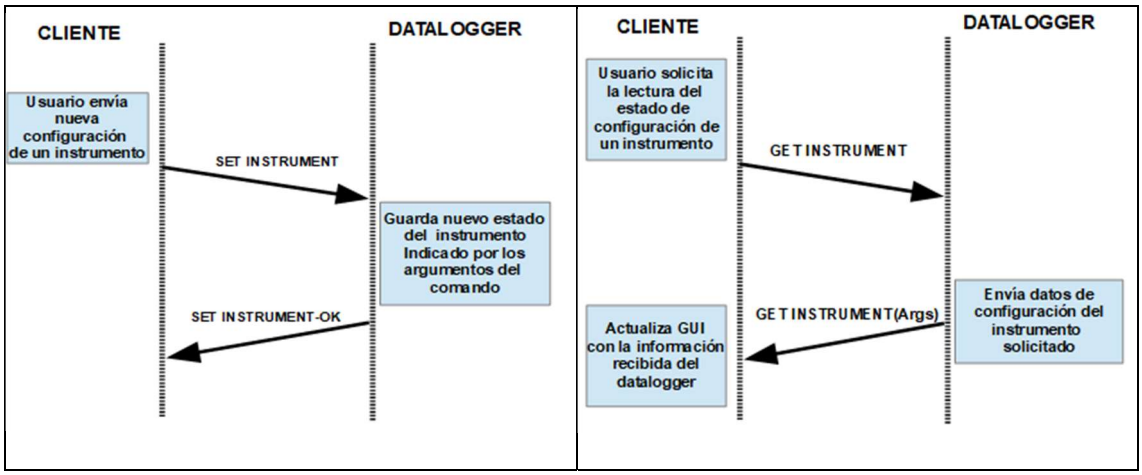


Figura 15. Comandos SET y GET INSTRUMENT

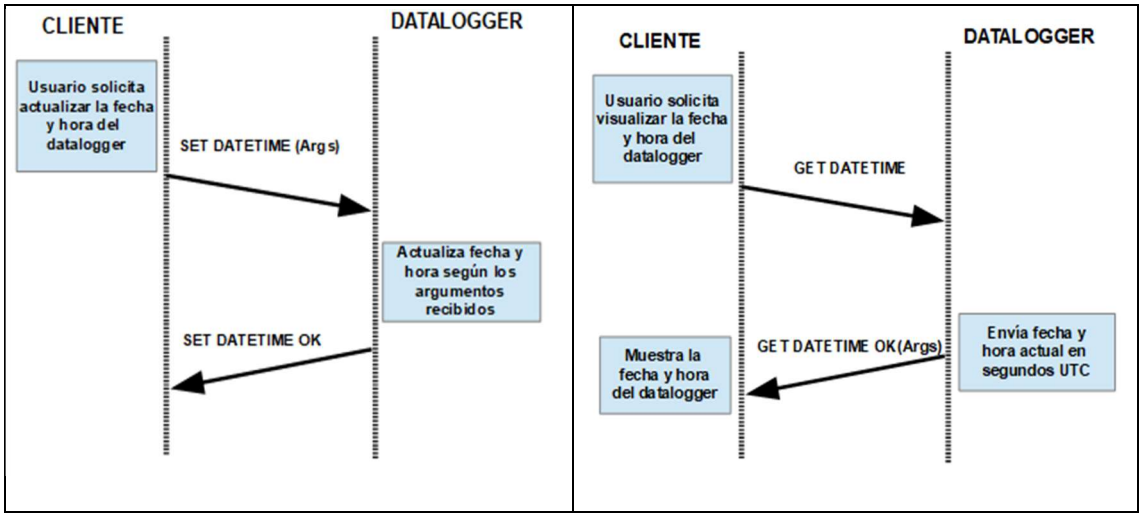


Figura 16. Comandos SET y GET DATETIME

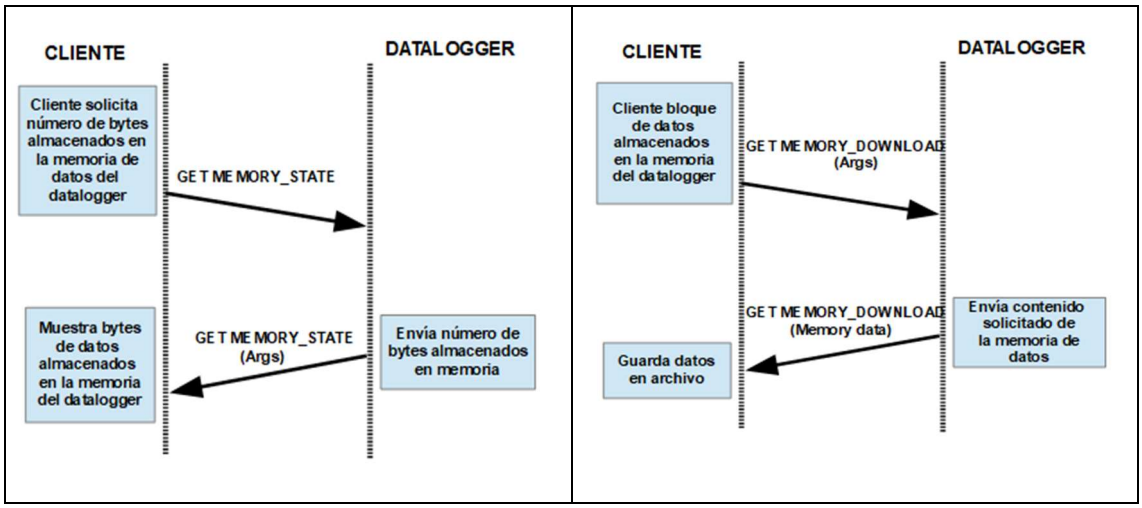


Figura 17. Comandos GET MEMORY STATE y DOWNLOAD

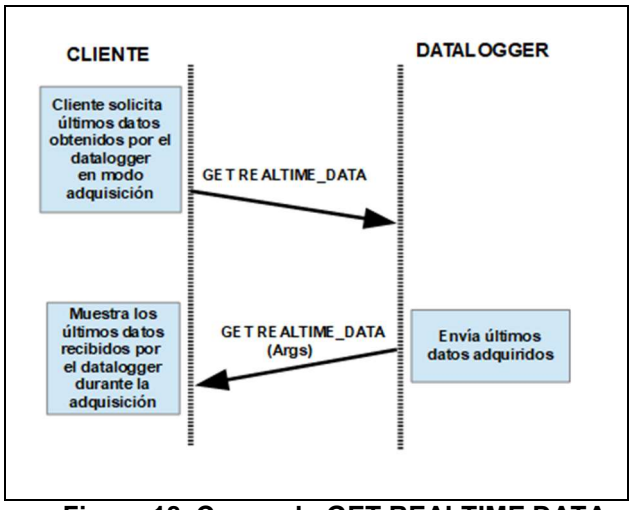


Figura 18. Comando GET REALTIME DATA

3.2 Implementación

3.2.1 Implementación del *firmware*

Tal y como se indica en el apartado del diseño, en el *firmware* del *datalogger*, los procesos y modos de funcionamiento se ejecutan en *threads* con diferentes niveles de prioridad.

La figura 19 contiene un esquema de los *threads* que componen el *firmware* del *datalogger*, las relaciones entre ellos y los componentes externos (cliente e instrumentos de medida), y los módulos del *firmware* a los que se acceden desde los diferentes *threads*.

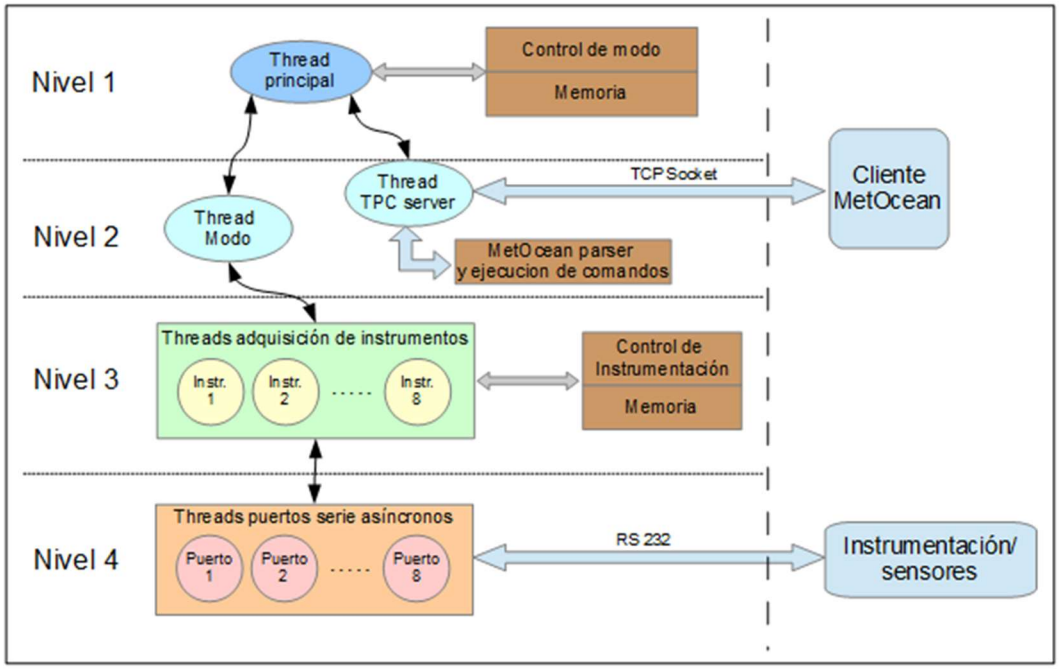


Figura 19. Esquema de threads del firmware

El funcionamiento de estos *threads*, las relaciones entre ellos y los módulos de los que hacen uso se describen en los siguientes apartados.

2.2.1.1 *Thread* principal

El *thread* principal inicia su ejecución tras un reset del sistema, su diagrama de flujo se muestra en la figura 20.

Tras el reset, se inicializan las funciones propias del SDK de WICED. Esta inicialización es necesaria para cualquier aplicación que haga uso de WICED. Una vez inicializado WICED se pasa a la inicialización de módulos y estructuras de datos propias del *datalogger*. La inicialización del *datalogger* se muestra en el diagrama de la figura 20 como un proceso predefinido. En la figura 21 se descompone el proceso de inicialización del *datalogger* en procesos más sencillos. Su descripción se detalla más adelante en este apartado.

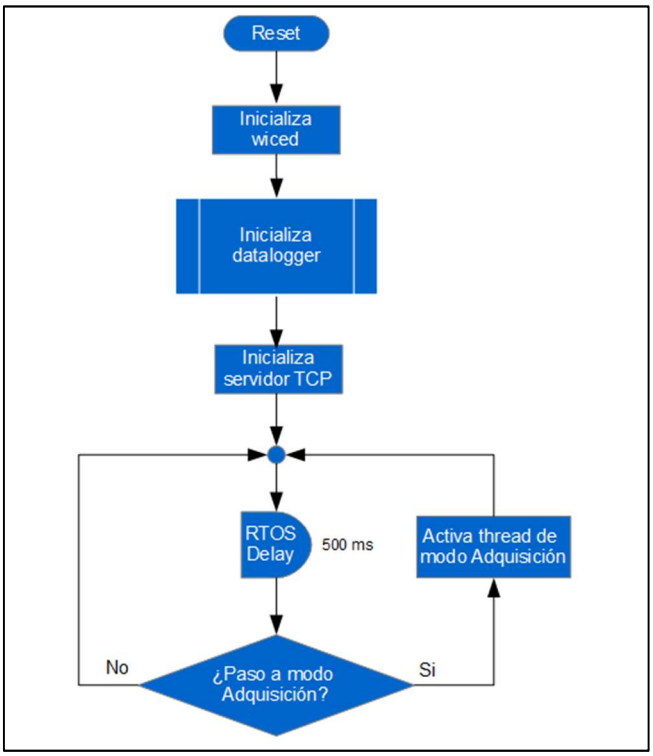


Figura 20. Thread principal

Durante el proceso de inicialización del servidor TCP, se crea el punto de acceso *WiFi* del *datalogger* y se inicializa el servidor TCP que permanece a la escucha del puerto 7777 en espera de la llegada de comandos desde el cliente.

Finalmente, el *thread* principal entra en un bucle sin fin en el que controla el paso del modo de funcionamiento de administración al modo de funcionamiento de adquisición. En el caso de detectar la llegada de un comando para el paso al modo de adquisición, libera el semáforo que mantenía el *thread* de modo adquisición en estado suspendido.

Con el objetivo de liberar al microprocesador de carga de trabajo y dar la opción de recibir tiempo de procesado a otros *threads* del sistema, el bucle

se ejecuta con una periodicidad de 500 ms, de modo que el tiempo transcurrido entre las sucesivas ejecuciones del bucle, el *thread* principal permanece en modo *sleep.*

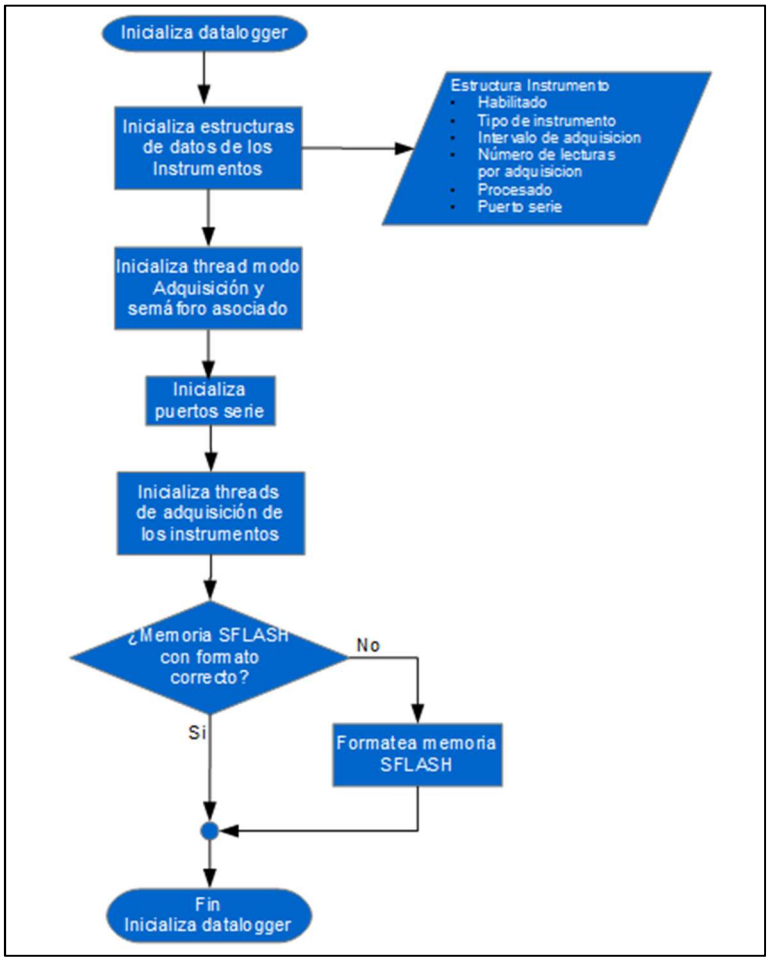


Figura 21. Inicialización del datalogger

En lo que respecta a la inicialización del *datalogger*, en la figura 21 se puede observar su diagrama de flujo. Este proceso parte con la inicialización de la estructura de datos que contiene la configuración actual de los instrumentos conectados al *datalogger*. Esta estructura de datos guarda la información introducida por el usuario en el cliente del *datalogger* cuando el sistema está en modo administración, y es empleada a la hora de ejecutar la adquisición de datos. Tras la inicialización todos los instrumentos están por defecto deshabilitados.

En segundo lugar, el proceso de inicialización del *datalogger* pasa a la creación y ejecución del *thread* de modo de adquisición y su semáforo asociado. El funcionamiento del *thread* del modo adquisición se detalla más adelante en este documento.

La inicialización de los puertos serie consiste en la habilitación de las lineas GPIO del PSoC 62 en función de si serán empleadas como líneas de transmisión de datos o líneas de recepción de datos de cada uno de los puertos serie asíncronos definidos en el *datalogger*. Se crean además

los *threads* y semáforos empleados en los procesos de recepción de datos de los puertos serie.

En la inicialización de los *threads* de adquisición de los instrumentos se crean los *threads* en los que se ejecutarán los procesos de adquisición de cada uno de los instrumentos que pueden ser habilitados en el *datalogger*. Se crean, por lo tanto, ocho *threads* de igual prioridad y de ejecución concurrente.

Como último paso en el proceso de inicialización se comprueba el estado de la memòria *flash* externa en la que se guardarán los datos adquiridos. Si se detecta que la memoria no está formateada según se describe en el apartado 3.1.2.4 de esta memoria, se procede a su borrado y formateo.

3.2.1.2 *Thread* de modo adquisición

Dentro del segundo nivel de ejecución de los *threads* del *firmware* del *datalogger* se encuentra el *thread* de modo de adquisición, que se encarga de activar y desactivar el modo de funcionamiento de adquisición.

El diagrama de flujo del *thread* de modo adquisición se muestra en la figura 22. Una vez inicializado, el *thread* chequea si el *datalogger* está en modo administración. Existe una variable empleada a modo de *flag* que indica el modo de funcionamiento actual del *datalogger*. En caso de estar en modo administración, se hacen pasar los *threads* de adquisición de los instrumentos a modo *sleep* y se pone el propio *thread* de modo adquisición en modo *sleep.* El paso a modo *sleep* se efectúa haciendo uso del semáforo de modo adquisición, de manera que el *thread* permanece en espera de que se active el citado semáforo. El encargado de activar el semáforo cuando se pasa al modo de adquisición es el *thread* principal (figura 20), en el proceso “Activa *thread* de modo adquisición”.

Por otro lado, si no está activo el modo administración, se chequea si estamos en la primera iteración tras la activación del modo adquisición. En caso afirmativo se activan los *threads* de adquisición de los instrumentos que han sido habilitados por el usuario durante la configuración, para lo que accede a la “estructura instrumentos” que almacena la configuración de cada instrumento disponible descrita en el apartado anterior. Por último y antes de iniciar cada nueva iteración del bucle, se suspende el *thread* durante un periodo de 200 ms por medio de un retardo del RTOS. De esta manera se descarga al microprocesador de la ejecución constante de este bucle para el empleo del tiempo de microprocesador en otros *threads* en ejecución.

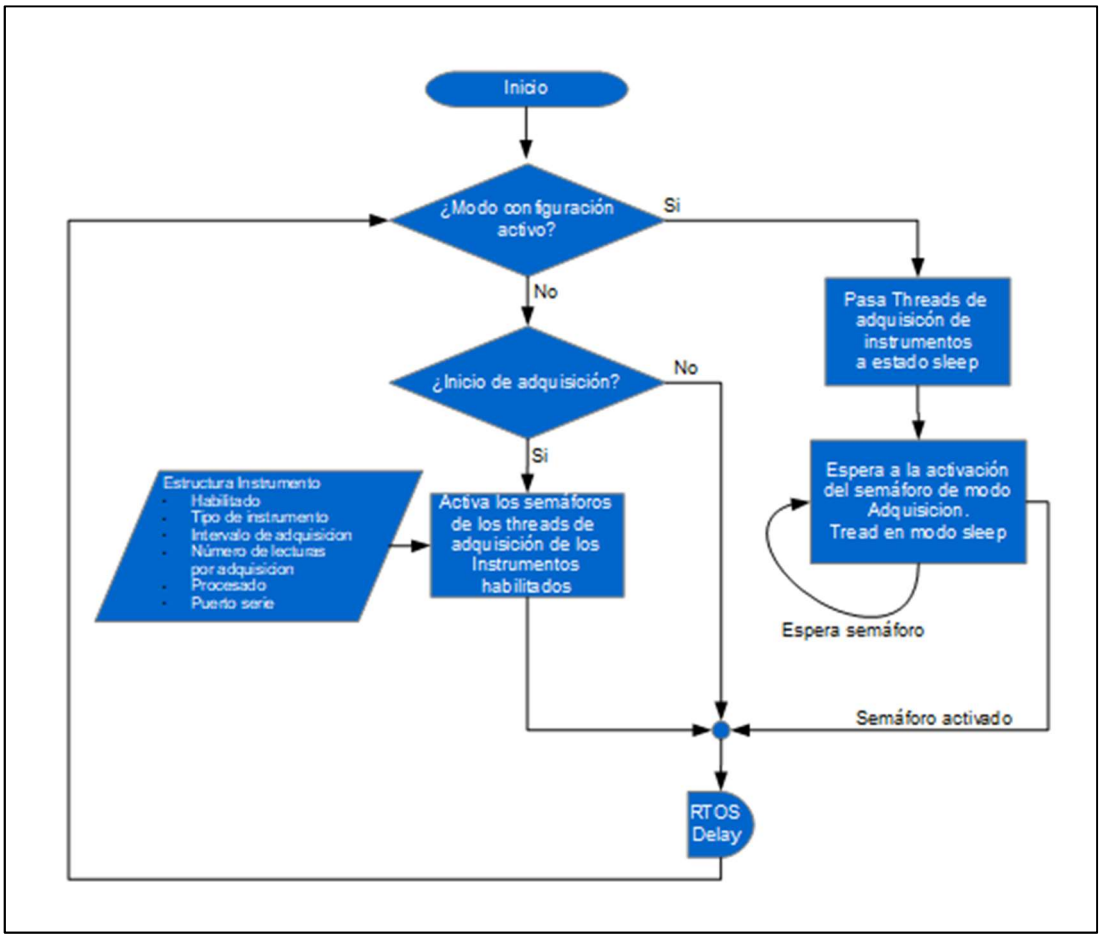


Figura 22. Thread de modo adquisición

2.2.1.3 Servidor TCP

En el mismo nivel de ejecución que el *thread* de modo adquisición se encuentran los *threads* del servidor TCP del *datalogger*. El servidor TCP tiene definidos tres *threads* cuya ejecución se asocia a los eventos de "cliente conectado al servidor", "cliente desconectado del servidor" y "datos recibidos".

La función de soporte del evento de "cliente conectado al servidor" es muy sencilla y tiene como objetivo la aceptación de la conexión del cliente por parte del servidor.

Con similar nivel de complejidad a la anterior, la función de soporte del evento "cliente desconectado del servidor" vuelve a configurar el puerto de conexión del servidor en modo escucha, en espera de la llegada de una nueva conexión del servidor.

Por su parte, la función ejecutada en el *thread* activado con el evento de "datos recibidos" es mucho más compleja que las dos anteriores, su diagrama de flujo se muestra en la figura 23.

El *firmware* del *datalogger* cuenta con un módulo que actúa a modo de analizador sintáctico o *parser* de los comandos recibidos desde el cliente,

y con un módulo encargado de la ejecución de estos comandos tras ser procesados por el *parser*. La mayor parte de la complejidad de la función de soporte al evento de datos recibidos reside en sus interacciones con los módulos de *parser* y ejecución del comando.

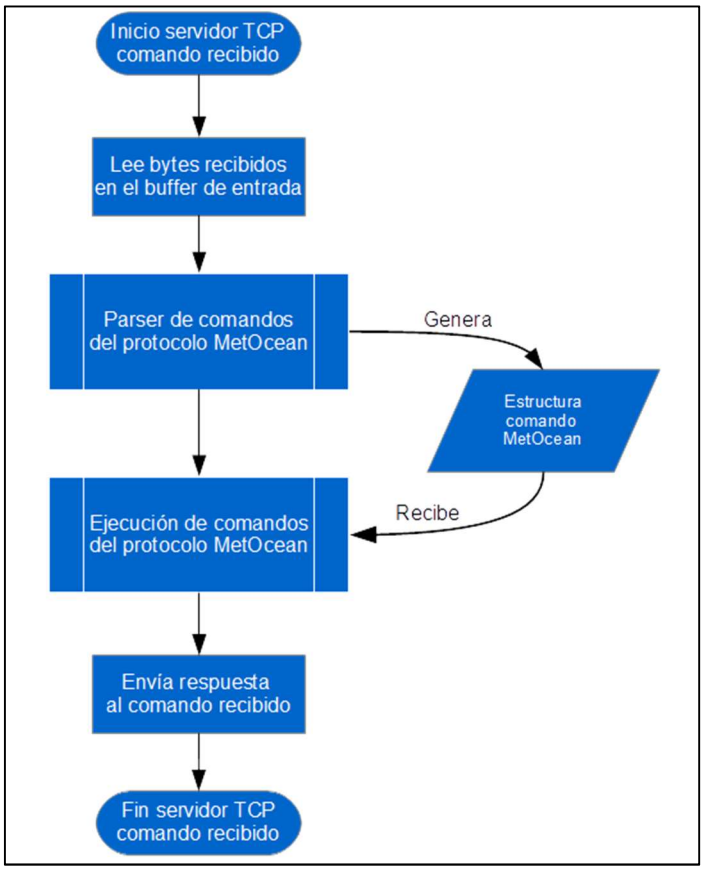


Figura 23. Comando recibido por TCP

Una vez activado el evento de recepción de datos en el servidor TCP, se pasa a leer el *buffer* de entrada del puerto de escucha. El contenido del *buffer,* por diseño, debe ser un comando del protocolo *MetOcean* enviado por el cliente. El comando recibido se pasa al módulo *parser.* El objetivo del *parser* es la extracción de la información contenida en cada uno de los comandos y generar una estructura de datos en la que toda la información extraída está ordenada de forma homogénea y accesible por el módulo de ejecución de comandos de manera independiente al tipo de comando y número de argumentos.

El tratamiento efectuado por el *parser* al comando queda descrito a alto nivel en el diagrama de la figura 24 y se describe en los siguientes párrafos.

El *parser* comienza a recorrer cada uno de los caracteres que componen el comando. Teniendo en cuenta los caracteres de control que separan cada uno de los campos del comando, en primer lugar extrae los códigos del comando primario y del comando secundario. A continuación entra en un bucle en el que en cada iteración extrae el código del tipo de argumento y el valor del argumento. En función de cada tipo de argumento el *parser*

efectúa diferentes acciones para la extracción de su valor, ya que dependiendo del tipo de argumento, el tipo de dato empleado para guardar su valor varía. En el diagrama de flujo de la figura 24, a modo de ejemplo y para simplificar el esquema, sólo se muestran tres tipos de argumentos.

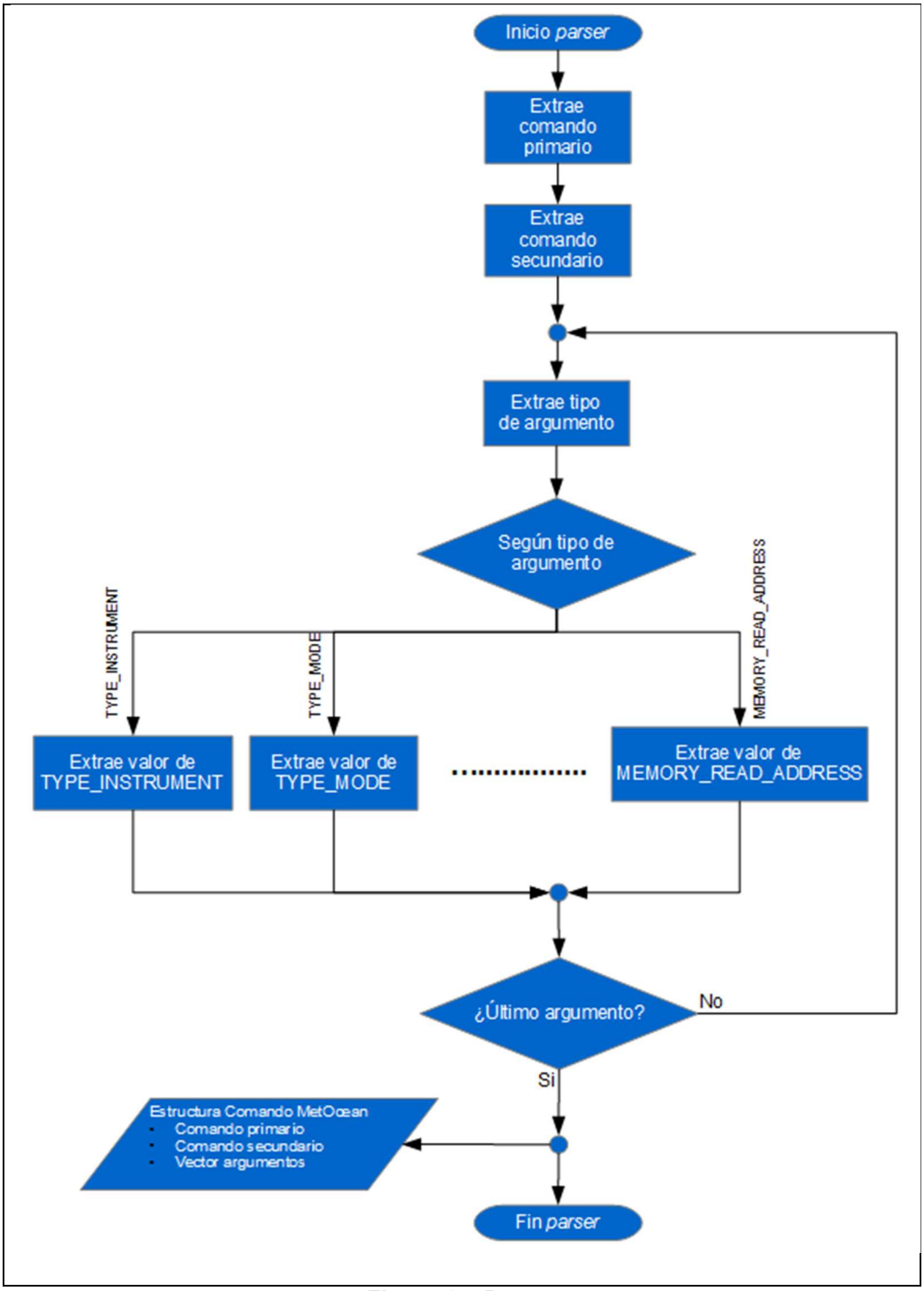


Figura 24. Parser

A continuación de la ejecución del *parser* se procede a la ejecución del comando recibido. Las figuras 25 y 26 muestran el diagrama de flujo del módulo del *firmware* encargado de ejecución del comando.

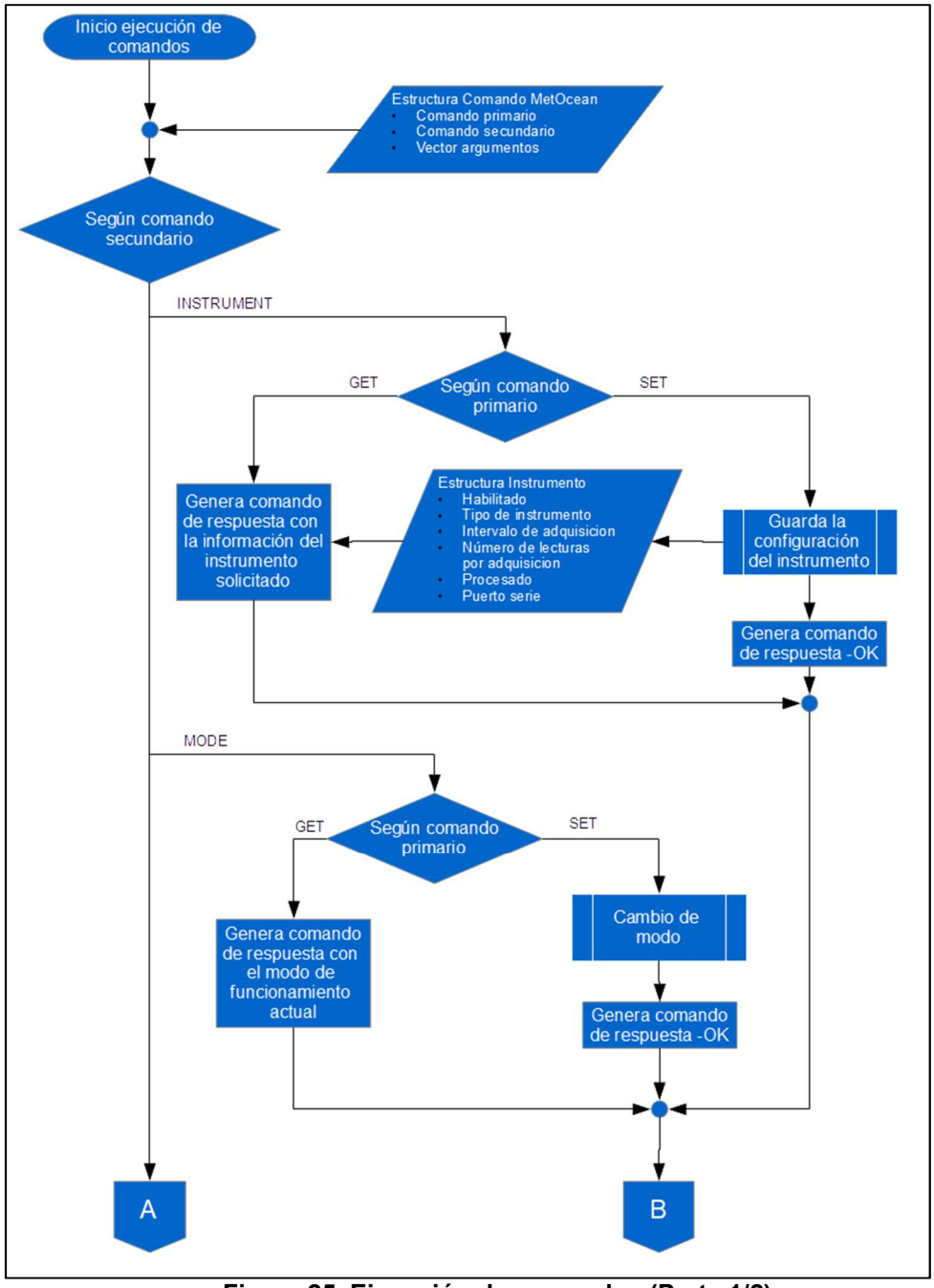


Figura 25. Ejecución de comandos (Parte 1/2)

El objetivo del módulo de ejecución de comandos es efectuar las operaciones correspondientes a los comandos recibidos y generar un comando de respuesta en el formato del protocolo *MetOcean*.

Al inicio del módulo de ejecución se chequea, en primer lugar, el comando secundario recibido. En el caso de que el comando secundario pueda estar acompañando a cualquiera de los dos comandos primarios disponibles (*SET* y *GET*), se pasa al chequeo del comando primario. Si el comando secundario recibido sólo está disponible con uno de los comandos primarios no es necesario el chequeo del comando primario.

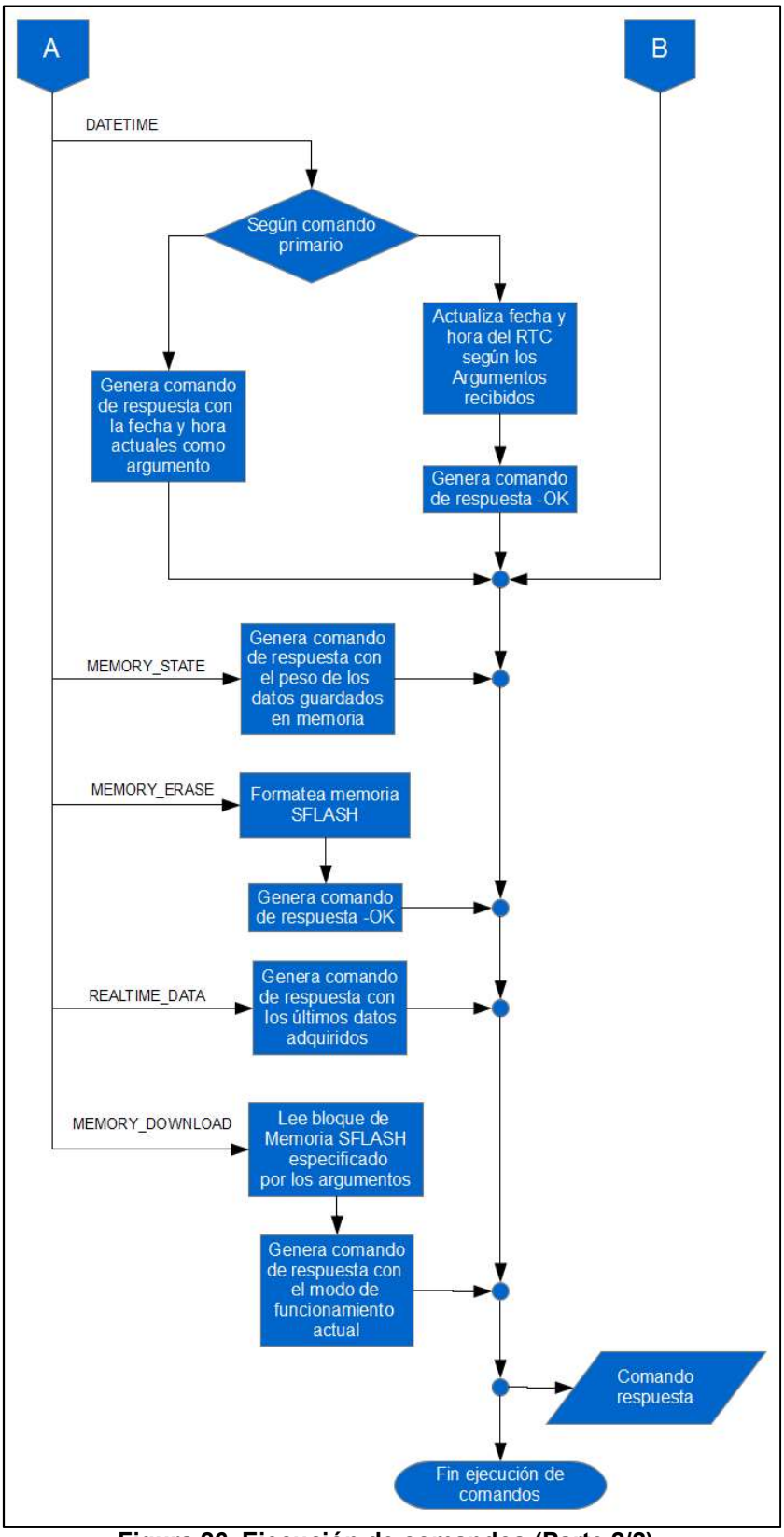


Figura 26. Ejecución de comandos (Parte 2/2)

Una vez discriminado el tipo de comando secundario y, en caso de ser necesario, el primario, se pasa a la ejecución de las acciones pertinentes en función del tipo de comando.

Cuando el comando secundario es *INSTRUMENT*, las acciones ejecutadas son la actualización de la estructura que contiene la configuración de los instrumentos,en el caso de comando primario tipo *SET*, o la lectura del contenido de esta estructura para la composición del comando de respuesta con la información leída en el caso de que el comando primario sea *GET*.

En el caso de que el comando secundario sea *MODE*, cuando el comando primario es *GET* se genera un comando en formato del protocolo *MetOcean* con la información del modo actual de funcionamiento. Cuando el comando primario es *SET* se actualiza el valor de la variable que contiene el modo de funcionamiento del *datalogger* según los argumentos del comando recibido. Tanto el *thread* de modo de adquisición como el *thread* principal se coordinan para pasar de un modo a otro en función del valor de esta variable.

Si el comando secundario recibido es *DATETIME*, y cuando acompaña al comando primario *GET*, se genera un comando de respuesta incluyendo en los argumentos la fecha y hora actuales del RTC del *datalogger* en formato ISO 8601. Si *DATETIME* acompaña al comando primario *SET*, se actualiza la fecha y hora del RTC con el valor de segundos UTC recibido en los argumentos.

Ante el comando *MEMORY_STATE*, el módulo de ejecución de comandos, obtiene el número de bytes de datos almacenados en memoria y genera el correspondiente comando *MetOcean* con la información obtenida.

Al recibir el comando secundario *MEMORY_ERASE,* se llama a la función de formateo del módulo de memoria, de modo que se inicializan los bytes de memoria destinados a guardar la dirección del siguiente acceso de escritura para que apunten a la dirección de base de la memoria de datos.

Cuando el comando secundario recibido es *REALTIME_DATA*, la respuesta del módulo de ejecución es la generación del comando *MetOcean* con los últimos datos adquiridos por los instrumentos habilitados.

Por último, en el caso en que el comando secundario sea *MEMORY_DOWNLOAD*, el módulo de ejecución accede al módulo de control de memoria para leer el bloque de memoria especificado por los argumentos recibidos. Los datos leídos desde la memoria pasan directamente a formar parte del comando *MetOcean* de respuesta. El máximo tamaño del bloque de memoria a leer está definido en 1024 Bytes en cada comando.

El comando respuesta generado para cada caso en el módulo de ejecución de comandos es directamente enviado al cliente como respuesta al comando inicial recibido.

3.2.1.4 *Threads* de adquisición

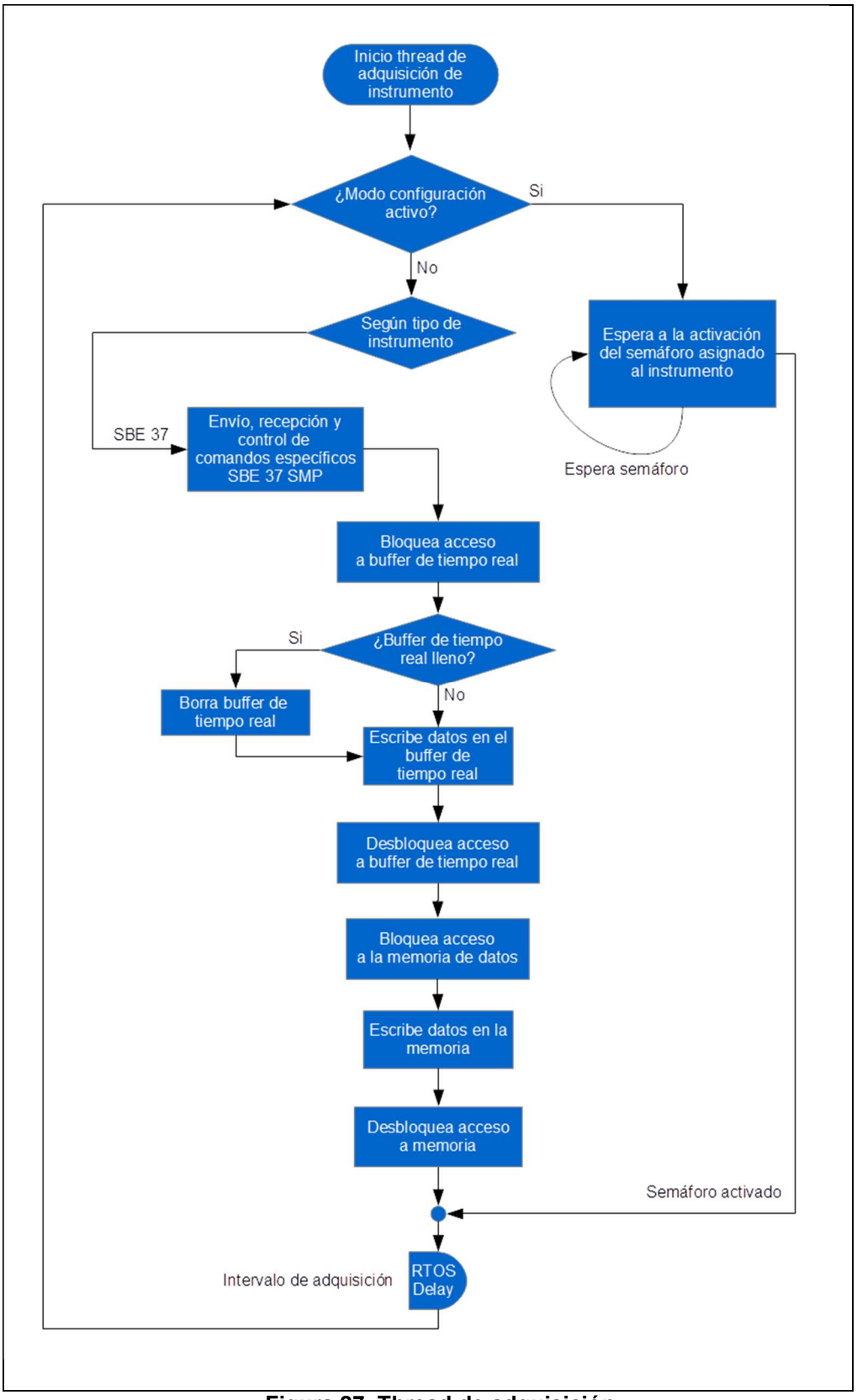


Figura 27. Thread de adquisición

En el tercer nivel de ejecución se encuentran los *threads* de adquisición. Se trata de ocho *threads* que se ejecutan en paralelo y que son creados y puestos en funcionamiento durante la inicialización del *datalogger*.

El diagrama de flujo de la función ejecutada en los *threads* de adquisición de instrumentos se muestra en la figura 27.

Durante el tiempo en el que el *datalogger* se encuentra en modo administración, los *threads* de adquisición de los instrumentos permanecen en estado suspendido en espera de que se activen sus respectivos semáforos. Cuando el *datalogger* entra en modo adquisición sólo son activados los semáforos de los *threads* de los instrumentos que han sido habilitados durante la administración.

Una vez activado el *thread* de adquisición de un instrumento, comienza la ejecución de un bucle infinito con periodo de repetición igual al intervalo de la toma de muestras programado para el instrumento. Dentro de este bucle se envían y reciben los comandos propios de cada sensor o instrumento por el puerto sèrie, empleando el protocolo RS232-C. En esta operación se efectúan las adquisiciones de datos según la configuración establecida por el usuario en lo referente al número de adquisiciones por intervalo y tipo de procesado aplicado a los datos.

3.2.1.5 Sensor SBE37 SMP

Actualmente en el *firmware* del *datalogger* está implementado el módulo para el envío, recepción y control del instrumento SBE 37 SMP (37).

El SBE 37 SMP es un instrumento configurable con capacidad para trabajar en modo autocontenido con un periodo de muestreo determinado, alimentado por sus propias baterías y almacenando los datos obtenidos en su propia memoria. Sin embargo para trabajar con el *datalogger* implementado, el SBE 37 SMP trabaja en modo *polled*, el instrumento permanece a la espera de la llegada de un comando de adquisición de datos, adquiere los datos y los transmite por el puerto serie.

La configuración del instrumento integrado en la adquisición del datalogger se muestra a continuación:

```
vMain =  9.95, vLith =  2.86
samplenumber = 0, free = 399457
not logging, never started
sample interval = 60 seconds
data format = converted engineering
output temperature, Celsius
output conductivity, S/m
output pressure, Decibar
output oxygen, ml/L
transmit real time data = yes
sync mode = no
minimum conductivity frequency = 3140.7
adaptive pump control disabled, pump on time 1.0 * 4.0 = 4.0 sec
<Executed/>
```

En modo *polled* el comando para que el SBE 37 SMP adquiera un dato y lo transmita por el puerto serie es la cadena de caracteres *“TS\n”*. La respuesta del instrumento ante este comando se muestra a continuación:

```
20.2753,  0.00004,   -0.233,  6.192, 14 Dec 2019, 09:27:14
<Executed/>
```

El contenido de la respuesta es una cadena de caracteres ASCII en la que los valores están separados por el caracter ',' y finaliza con la subcadena '<Executed/>'. Siguiendo lo indicado en la configuración del instrumento antes mostrada, el primer valor es la temperatura medida en ºC, el segundo es la conductividad en S/m, el tercero es la presión dado en decibares y el cuarto es el oxígeno disuelto en mL/L.

La figura 28 muestra el diagrama de flujo del código implementado para en envío, la recepción de comandos y el control del SBE 37 SMP.

El código implementado en primer lugar envía por el puerto serie asignado el comando para iniciar una toma de datos del SBE 37 SMP, esto es, envía un 'TS\n' y a continuación entra en un bucle en el que chequea si hay nuevos bytes recibidos en el puerto y si el último byte recibido se corresponde con el carácter '<', que indica que ya se han recibido todos los datos de interés.

En el caso de ejecutar el bucle durante determinado tiempo, en este caso establecido en 5 segundos aproximadamente, sin que se haya recibido el carácter '<', se da por finalizada la adquisición sin datos.

Una vez recibido el caracter '<', se lee el *buffer* de entrada del puerto, que contiene todos los caracteres recibidos hasta ese momento. Al leer el *buffer*, el código de control del puerto borra su contenido.

Tras la lectura del *buffer* se procesa la cadena de caracteres, separando los campos por ',', y el valor de cada parámetro es almacenado en un vector. Existe un vector para almacenar los datos de cada uno de los parámetros medidos.

Tras el procesado de la cadena recibida y si aún no se han efectuado todas las lecturas establecidas por el usuario para cada adquisición, se vuelve a iniciar una nueva lectura. En caso contrario, se procesan los datos obtenidos según la configuración dada por el usuario. Esto es, para cada parámetro se puede calcular la media, la mediana o no hacer ningún procesado.

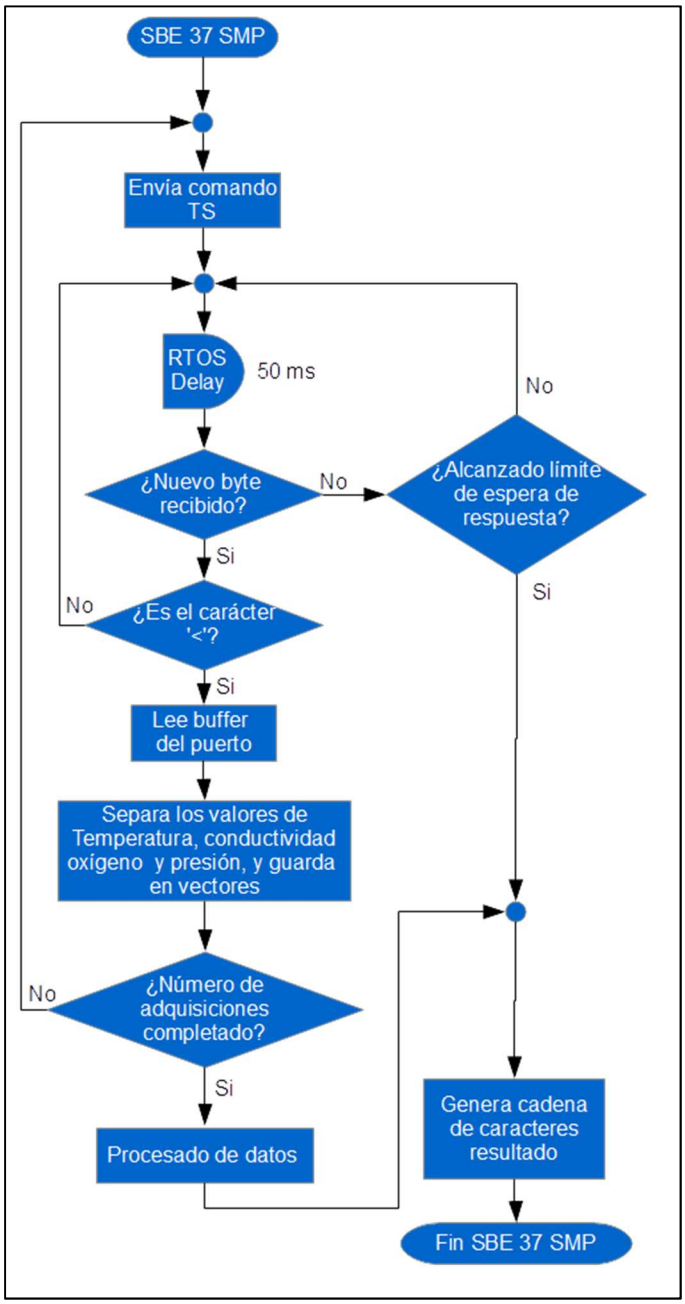


Figura 28. Adquisición del SBE37 SMP

Finalmente se genera la cadena de caracteres del resultado de la adquisición, añadiendo a los datos obtenidos la fecha y hora del *datalogger*.

3.2.1.6 Puertos serie asíncronos

Tal y como se indica en el apartado de diseño, el *firmware* desarrollado cuenta con un módulo de puertos serie creado específicamente para hacer trabajar a dos lineas generales de entrada y salida del PSoC 62 (GPIO) como un puerto serie RS232-C. De las dos líneas, una línea es ocupada para la recepción de datos (Rx) y la otra para la transmisión de datos (Tx). Cada uno de los ocho instrumentos configurables del

datalogger está asociado a dos líneas GPIO del PSoC 62 según la tabla 9.

Cada línea GPIO empleada como transmisor de datos está configurada como salida con resistencia de *pull up*, las líneas GPIO empleadas como receptor de datos está configurada como entrada con alta impedancia.

En el diagrama de flujo de la figura 29 se describe el funcionamiento de la función de envio de datos por el puerto serie RS232-C. Esta función tiene como argumento de entrada un vector de bytes (codificados como *unsigned integer* de 8 bits) y envía por la correspondiente linea de transmisión todos los bytes almacenados en el vector de entrada. En ella se asume un esquema de la trama a enviar de ocho bits de datos, un bit de stop sin bit de paridad y sin protocolo de *handshake*. El *endianess* del puerto es *big-endian*.

Instrumento	Línea del puerto	PSoC 6 GPIO
Instrumento 1	Tx	WICED_GPIO_61
	Rx	WICED_GPIO_62
Instrumento 2	Tx	WICED_GPIO_63
	Rx	WICED_GPIO_65
Instrumento 3	Tx	WICED_GPIO_66
	Rx	WICED_GPIO_67
Instrumento 4	Tx	WICED_GPIO_69
	Rx	WICED_GPIO_70
Instrumento 5	Tx	WICED_GPIO_71
	Rx	WICED_GPIO_72
Instrumento 6	Tx	WICED_GPIO_98
	Rx	WICED_GPIO_99
Instrumento 7	Tx	WICED_GPIO_93
	Rx	WICED_GPIO_94
Instrumento 9	Tx	WICED_GPIO_39
	Rx	WICED_GPIO_40

Tabla 9. Líneas GPIO asociadas a los puertos RS232 del datalogger

Los *threads* en los que se ejecutan las funciones del puerto serie cuentan con el mayor nivel de prioridad entre todos los que componen el *firmware* del *datalogger*. De esta manera se evita que *threads* que ejecutan otros servicios menos críticos en lo que a temporización se refiere puedan interferir en el correcto desempeño de los puertos serie asíncronos del *datalogger*.

Tal y como se muestra en la figura 29, el envio de datos por el puerto serie comienza con la entrada en un bucle en el que se recorren todos los bytes a enviar. Al comienzo del envio de cada byte se establece el bit de start

de la transmisión del byte. El envio del bit de start supone establecer el GPIO correspondiente a nivel bajo y a continuación, haciendo uso de un retardo del RTOS, se mantiene el tiempo de bit de start en el puerto. Al finalizar el tiempo del bit de start comienza la ejecución de un bucle que recorre todos los bits del byte que se està enviando, siguiendo el *endianess big-endian*.

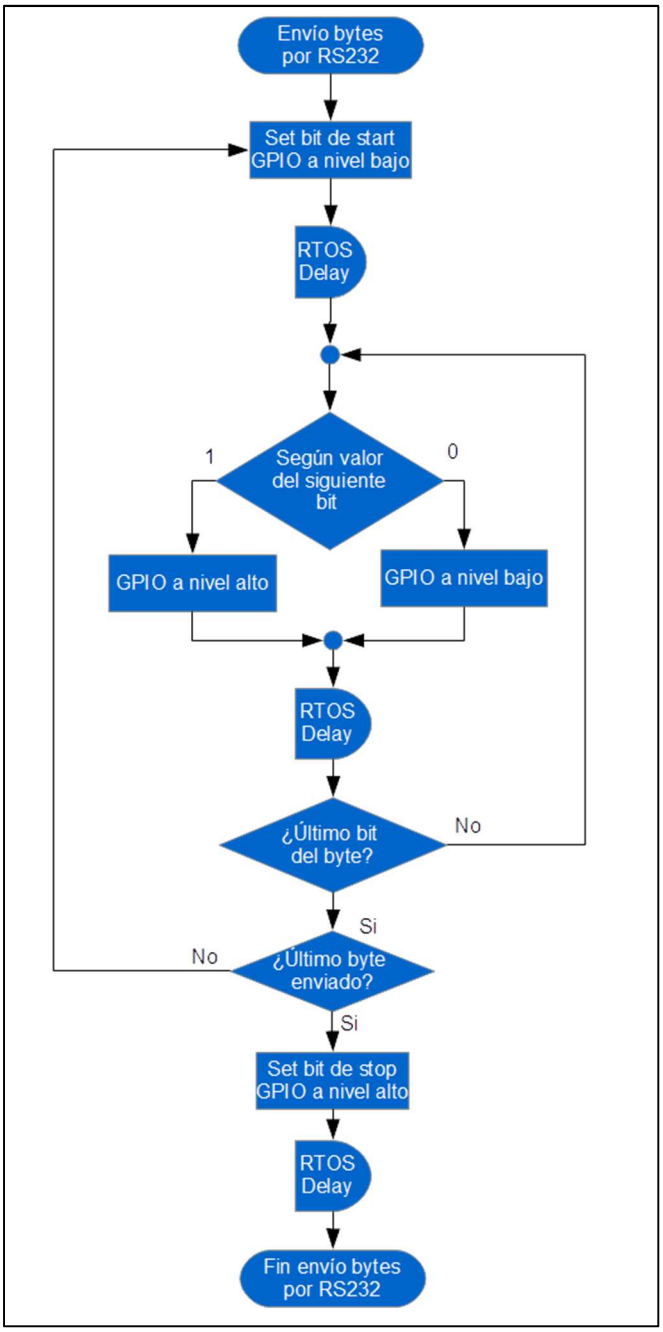


Figura 29. Envío de bytes por el puerto serie RS232-C

Si el bit a enviar es un “1” lógico, se establece la línea GPIO correspondiente a nivel alto. En el caso que el bit a enviar sea un “0” lógico, la línea GPIO empleada como transmissora se situa a nivel bajo. Al igual que en el caso del bit de start, tras situar el nivel lógico adecuado

en la GPIO, el *thread* pasa a estado *sleep* por medio de un retardo del RTOS.

El valor del retardo del RTOS empleado para cada bit se estableció inicialmente a nivel teórico para un puerto serie de 9600 baudios como la inversa de la velocidad del puerto. Es decir, el tiempo de bit para 9600 baudios es de 10,416 ms. Siguiendo un procedimiento empírico se ajustó el valor del retardo para tener en cuenta los tiempos empleados en el resto de instrucciones ejecutadas durante el tiempo de bit. El ajuste empírico del retardo se estableció por medio de la visualización de las señales generadas durante la transmisión del byte 0xAA en un osciloscopio y efectuando un ajuste fino de la duración de cada bit.

Al finalizar el tiempo de bit del último bit del byte se pasa al envio del bit de stop. Para esto se pone el GPIO de transmisión a nivel bajo y se vuelve a pasar el *thread* a modo *sleep* durante el tiempo del bit de stop.

Cuando se han enviado todos los bytes, finaliza la ejecución de la función de envio por el puerto serie RS232-C.

La recepción de datos por el puerto serie RS232-C está compuesta por dos funciones: por un lado el *thread* encrgado de la recepción de los datos desde la correspondiente línea GPIO, y por otro lado, una función de respuesta a la interrupción associada a la linea GPIO empleada como recepción de datos.

Cada tipo de sensor tiene un tiempo de respuesta diferente ante una solicitud de efectuar una medida. Este tiempo de respuesta puede variar desde el orden de unos milisegundos hasta el orden de varios segundos, por lo que, desde el punto de vista del *datalogger*, la llegada de los datos de los sensores se producen de manera asíncrona y aleatoria.

La llegada de una trama de datos a una línea de recepción del puerto serie RS232-C del *datalogger* es un evento critico en el dominio temporal. El *datalogger* necesita dar respuesta lo sufientemente rápida ante estos eventos para evitar la pérdida de datos enviados por los sensores.

Para dar respuesta a los eventos de llegada de datos desde los sensores, cada línea GPIO empleada como receptora de datos tiene habilitada una interrupción que se dispara ante un evento de flanco de bajada. La llegada de un flanco de bajada en el puerto representa la llegada de un bit de start de una trama de datos.

La figura 30 muestra el diagrama de flujo de la función de recepción de datos por el puerto serie y de la rutina de servicio de la interrupción acriva por flanco de bajada en la linea GPIO.

Mientras el *datalogger* se encuentra en modo adquisición, mantiene un *buffer* que almacena de forma temporal los últimos datos adquiridos. El tamaño máximo del buffer está definido en 100 bytes y su función es la de

proporcionar una respuesta rápida cuando el cliente solicita datos en tiempo real por medio del comando *MetOcean GET REALTIME_DATA* . Debido a que todos los *threads* de los instrumentos activos tienen acceso al *buffer*, es necesario el uso de mecanismos de seguridad para evitar el acceso concurrente al mismo por varios *threads*. En este caso el mecanismo empleado es un *mutex* del RTOS. Cuando se va a acceder al *buffer*, el *thread* bloquea el *mutex* asociado al *buffer* de tiempo real, si este ya estaba bloqueado por otro *thread*, se pasa a modo suspendido hasta que el *mutex* haya sido desbloqueado por el resto de *threads*. Cuando el *thread* bloquea el *mutex* pasa a verificar si el *buffer* se llenará con la nueva información de tiempo real a añadir. En caso afirmativo se borra el *buffer* y se guarda los nuevos datos, en caso negativo simplemente se añaden los nuevos datos al contenido del *buffer*.

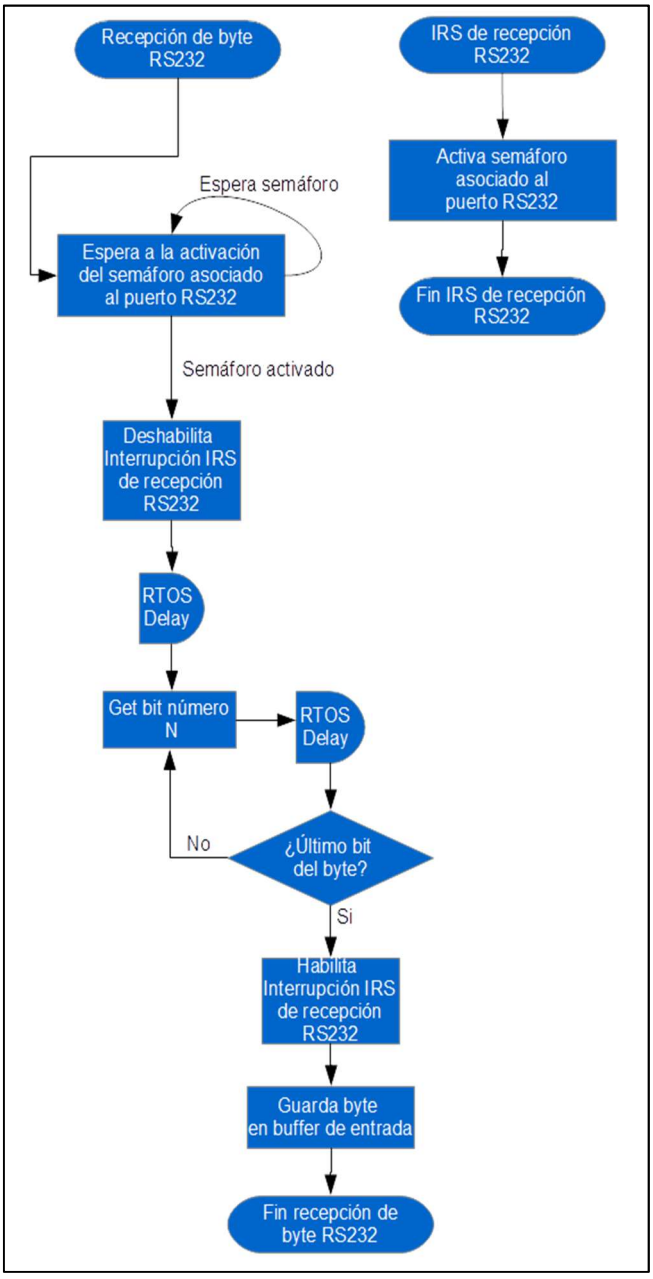


Figura 30. Recepción de bytes por el puerto serie RS232-C

Idéntico recurso se emplea durante las operaciones de escritura de datos en la memoria. Se emplea un *mutex* que está asociado a la escritura en memoria, de manera que cuando un *thread* pasa a guardar datos en memoria, bloquea el acceso al resto de *threads* por medio del *mutex* y en caso de que el *mutex* se encuentre bloqueado por otro *thread*, se pasa a modo suspendido hasta que quede desbloqueado.

La figura 31 muestra una imágen de la pantalla del osciloscopio tomada durante el ajuste de los tiempos de espera de la función de recepción de datos del puerto RS232-C. La señal del canal 2 del osciloscopio (en azul) es una señal generada a modo de depuración durante la implementación del *firmware* del puerto. Esta señal proviene de una GPIO del PSoC 62 programada como salida que cambia de nivel en el momento en que el puerto lee el dato de la línea Rx. La señal del canal 1 es la señal recibida en la línea Rx del puerto implementado.

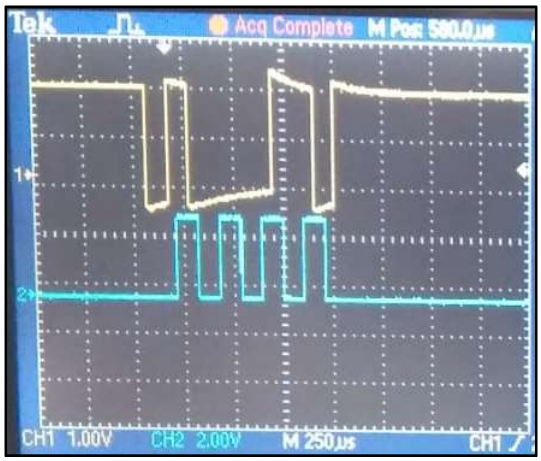


Figura 31. Ajuste de recepción RS212-C con osciloscopio

3.2.2 Implementación del cliente

El cliente del *datalogger* es una aplicación desarrollada con el lenguaje de programación *python* con la que el usuario interactúa para la configuración y descarga de datos del *datalogger*.

La GUI del cliente cuenta con dos pantallas: una en la que el usuario configura los instrumentos conectados al *datalogger* (figura 32) y otra en la que el usuario configura la fecha y hora del datalogger, accede a los datos de memoria y a datos en tiempo real (figura 33).

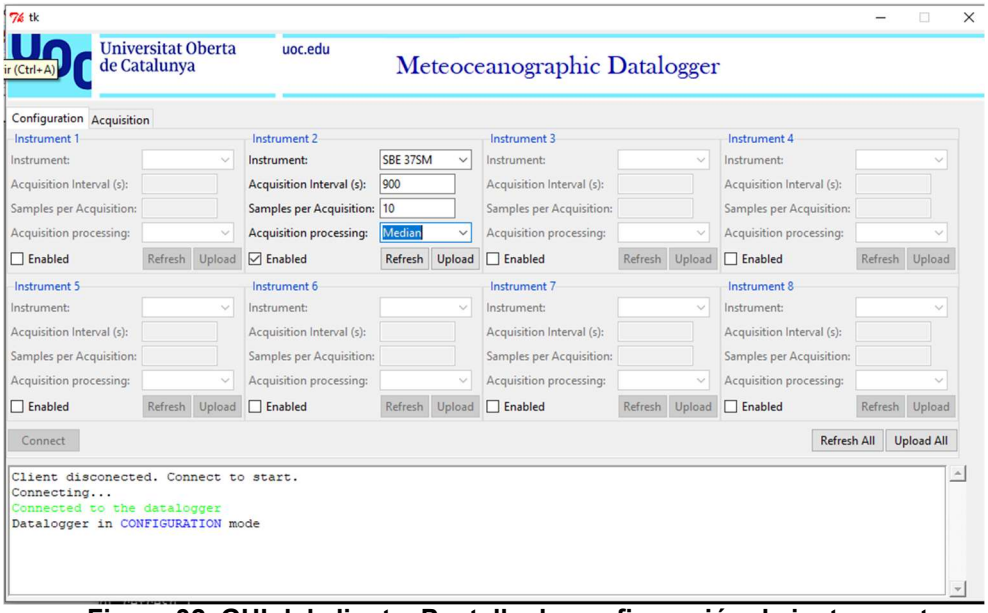


Figura 32. GUI del cliente. Pantalla de configuración de instrumentos

Para dotar al cliente de la funcionalidad necesaria para dar respuesta a las acciones del cliente, se ha implementado el código en *python* cuyo funcionamiento se describe a continuación.

Tras la creación de la GUI, el usuario sólo tiene habilitado el botón *"Connect"*. Cuando el usuario pulsa este botón, el cliente trata de enviar el comando *MetOcean GET MODE* para obtener el modo de funcinamiento actual del *datalogger*. Si no recibe respuesta desde el datalogger se indica al usuario que no se pudo establecer conexión. Si, por el contrario, se recibe respuesta, se actualiza el estado de la GUI incluyendo información del modo actual del *datalogger* y actualizando el estado de los controles e indicadores de la GUI. El diagrama de flujo de la función asociada al botón "*Connect*" se muestra en la figura 34.

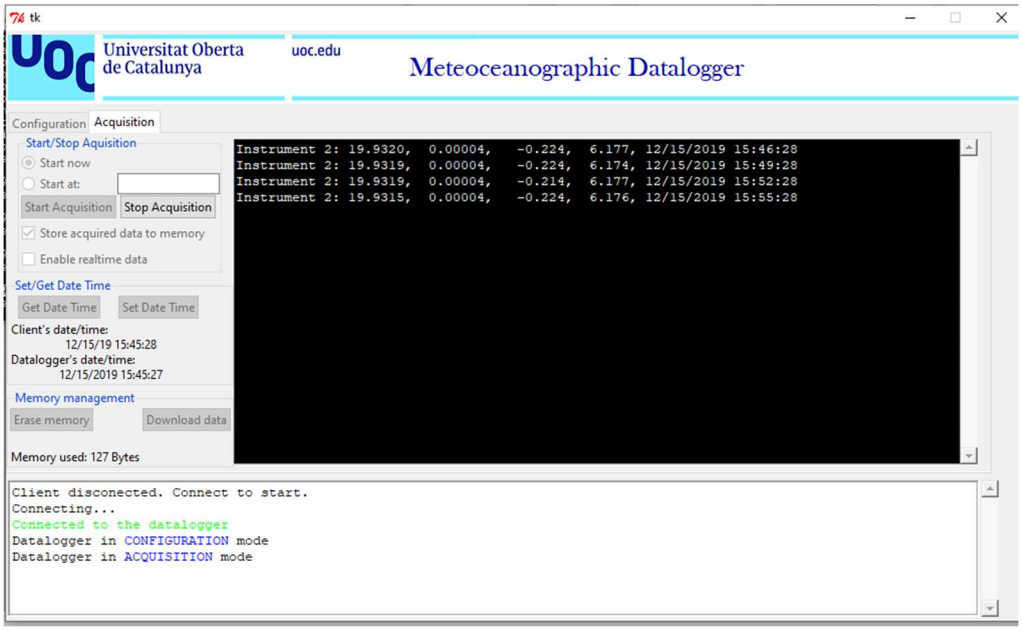


Figura 33. GUI del cliente. Pantalla de adquisición con datos en tiempo real

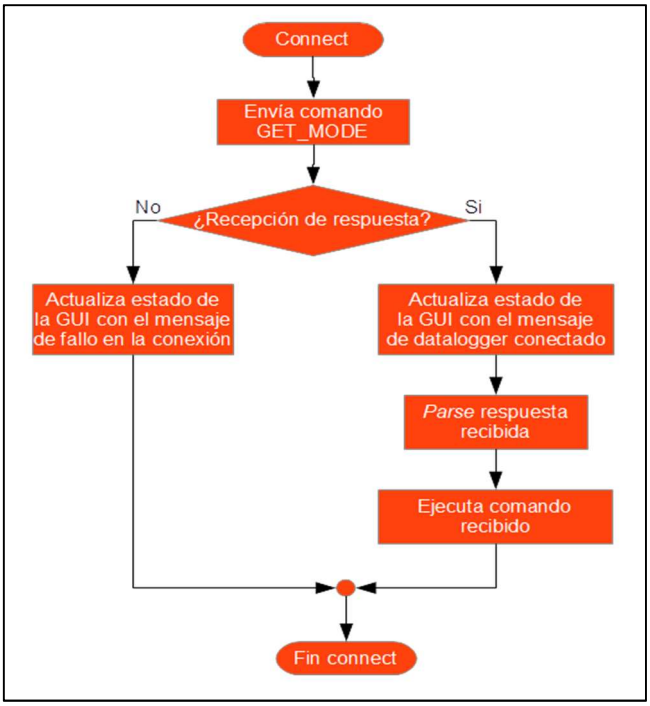


Figura 34. Función "Connect"

Como se observa en la figura 34, el cliente cuenta con un módulo *parser* que se encarga de recibir la cadena de caracteres con el comando *MetOcean* de respuesta recibido desde el *datalogger* y genera un objeto de tipo *MetOcean Command* que contiene, ordenada por campos, toda la información recibida en el comando. El diagrama de flujo del módulo parser del cliente se muestra en la figura 35.

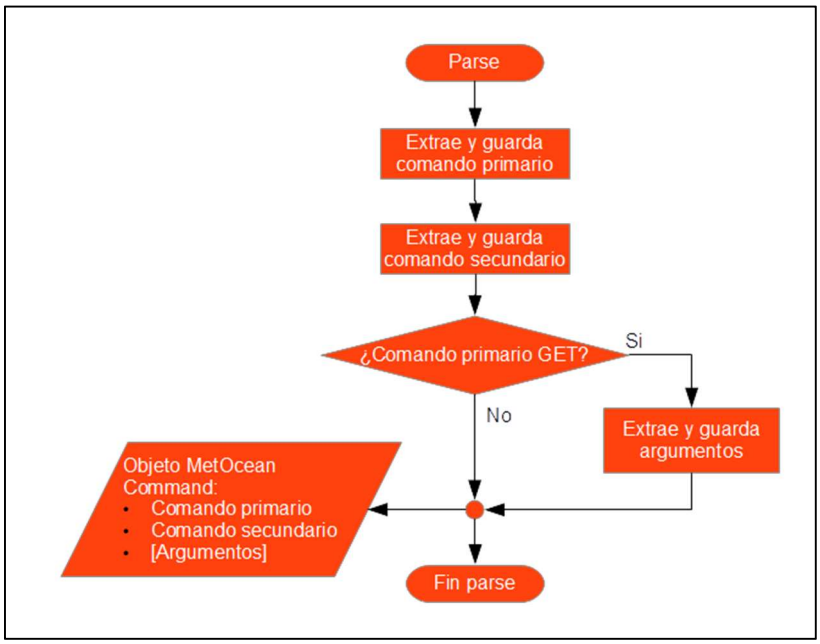


Figura 35. Parser del cliente

El objeto resultado del *parser* es recibido por el módulo de ejecución del comando. El diagrama de flujo del módulo de ejecución se muestra en la figura 36.

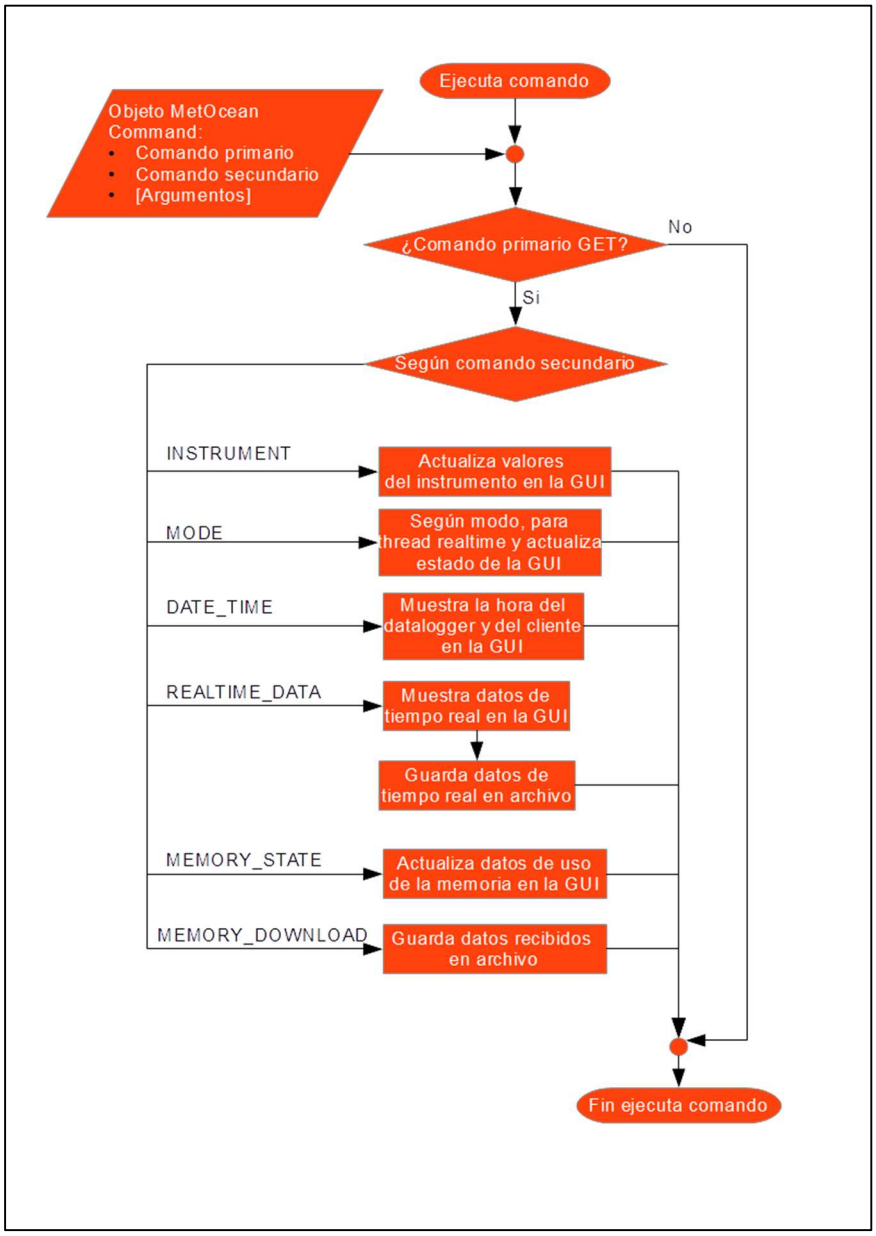


Figura 36. Ejecución de comandos del cliente

Cuando el usuario inicia una adquisición pulsando el botón *"Start Acquisition"* se ejecuta en el cliente la función para poner el *datalogger* en modo adquisición cuyo diagrama de flujo se muestra en la figura 37.

En la función de activación del modo adquisición del *datalogger*, en primer lugar se chequea si el usuario ha programado una fecha y hora de inicio de la adquisición. En caso afirmativo se valida la información de fecha y hora introducida por el usuario. La validación incluye el chequeo del formato de los datos introducidos y el chequeo de valores de fecha y hora válido, incluyendo chequeo de año bisiesto. En el caso de que el usuario haya introducido un dato sin seguir el formato adecuado o una fecha u

hora inexistentes, se muestra una ventana de error indicando el problema detectado.

Cuando la fecha y hora introducidos por el usuario son correctos, se envía el comando de lectura de fecha y hora del *datalogger*. Se compara la fecha y hora del datalogger con la fecha y hora introducida por el usuario. En el caso de que la fecha y hora intoducida por el usuario para comenzar la adquisición sea anterior a la fecha y hora del datalogger, se muestra una ventana de error al usuario indicando el problema detectado.

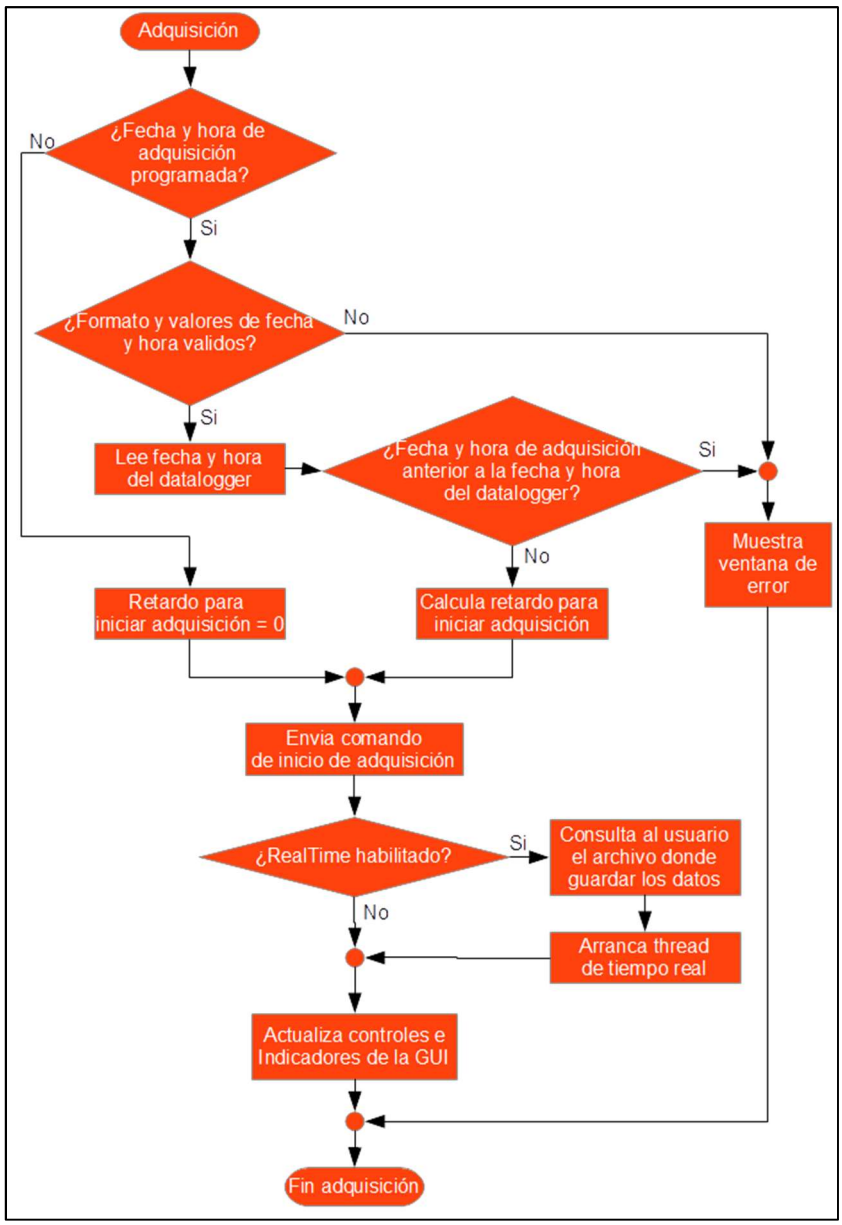


Figura 37. Paso a modo adquisición del cliente

Una vez superadas las comprobaciones anteriores, el cliente calcula el tiempo en milisegundos que debe transcurrir para iniciar la adquisición.

Si el usuario no ha programado una fecha y hora de comienzo de la adquisición, el tiempo que debe transcurrir para iniciar la adquisición es nulo.

El número de milisegundos que deben transcurrir hasta que se inicia la adquisición es uno de los argumentos del comando *MetOcean* para el paso a modo adquisición del *datalogger*.

A continuación se envía el comando *MetOcean* al *datalogger*, y se chequea si el usuario desea visualizar y adquirir los datos en tiempo real. En caso afirmativo se le solicita al usuario que introduzca el nombre y localización del archivo en el que desea que se guarden los datos a medida que se van recibiendo y se arranca el *thread* de tiempo real del cliente.

El *thread* de tiempo real del cliente ejecuta una función en segundo plano cuyo diagrama de bloques se muestra en la figura 38.

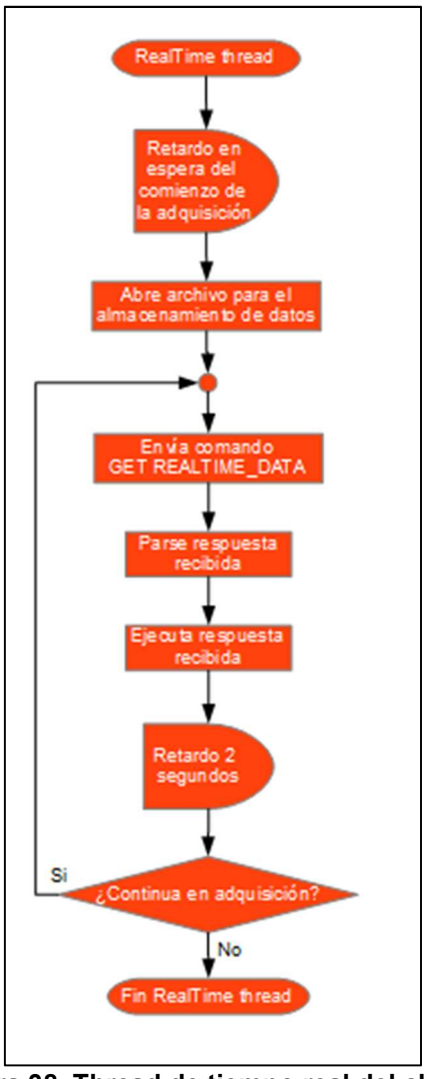


Figura 38. Thread de tiempo real del cliente

Una vez iniciada la función del *thread* de datos en tiempo real, la primera acción es esperar a que comience la adquisición, el valor del tiempo a esperar es el mismo que se ha enviado al *datalogger* en el comando de paso a modo adquisición. Si el usuario no ha programado una fecha y hora para el inicio, el valor del tiempo de espera es cero.

Cuando se cumple el tiempo de espera, en primer lugar se crea y abre el archivo en el que se guardarán los datos recibidos, para a continuación entrar en un bucle en el que se envía el comando *GET REALTIME_DATA* al *datalogger* para recibir los últimos datos adquiridos, guardarlos en el archivo y mostrarlos en la GUI del cliente.

Cuando se pasa al modo andministración, se sale del bucle y finaliza el *thread*.

3.2.3 Manual de funcionamiento del cliente

Para la ejecución del cliente del *datalogger* se requiere un computador con adaptador de red inalámbrico con capacidad de trabajar según el estándar lEEE 802.11n y con un intérprete de *python* versión 2.7 instalado.

La distribución del cliente está compuesta por tres archivos:

- main.py: script principal de la aplicación.
- MetOceanParser: script que contiene código relativo al parser de comandos MetOcean del .liente.
- logo_v4: imagen de cabecera de la GUI

Para iniciar una sesión de trabajo con el cliente del *datalogger* se debe ejecutar el script *main.py.*

Una vez inicializada, la única opción disponible dentro de la GUI es el establecimiento de conexión con el *datalogger*, por medio del botón *“Connect”* (figura 39).

Para el establecimiento de la comunicación el *datalogger* debe estar alimentado y el computador en el que se ejecuta el cliente debe estar conectado a la red *MetOcean Datalogger*.

En el caso de pulsar el botón *Connect* sin estar dentro de la red *MetOcean Datalogger*, el cliente no será capaz de establecer la conexión con el *datalogger* y mostrará el mensaje de error en la conexión (figura 40).

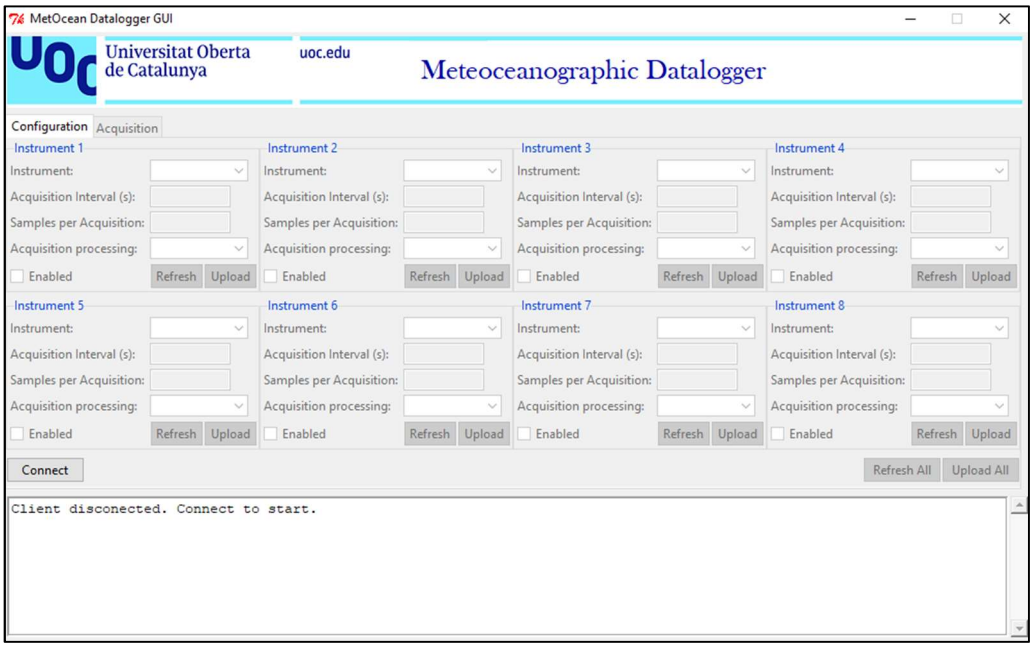


Figura 39. Pantalla inicial del cliente

Cuando el cliente establece la conexión con el *datalogger*, indica que está conectado y muestra el modo de funcionamiento actual del *datalogger*.

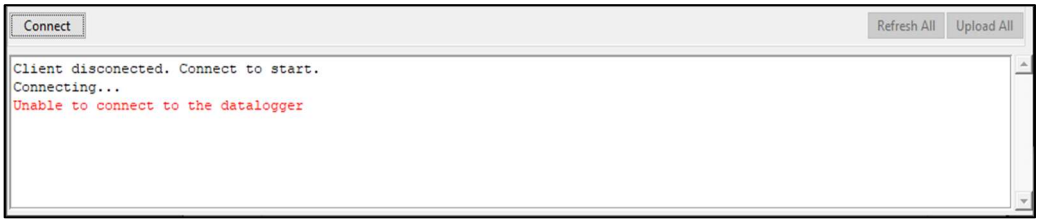


Figura 40. Cliente sin establecer conexión con el datalogger

Una vez conectado, si el *datalogger* se encuentra en modo administración, el usuario puede interaccionar con todos los controles de configuración de los instrumentos. El *datalogger* tiene capacidad de adquirir datos de hasta ocho instrumentos en paralelo, cada instrumento tiene asociado un puerto de comunicaciones RS232-C. La pantalla de configuración de instrumentos del cliente permite la configuración de cada uno de estos ocho instrumentos por separado, para ello cuenta con un conjunto de controles de configuración para cada instrumento (figura 41).

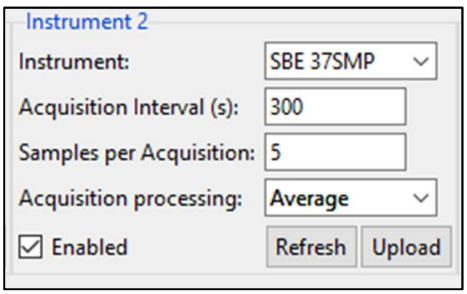


Figura 41. Configuración de instrumento

3.2.3.1 Configuración de instrumento

Cada instrumento cuenta con los siguientes controles de configuración:

- *Enabled checkbutton*. Por medio de este control se habilita o deshabilita el instrumento para la adquisición de datos según el resto de parámetros de configuración.

- Selección del instrumento, mediante el control *Instrument*. El usuario selecciona el instrumento del que se desea adquirir datos.

- Intervalo de adquisición. El usuario debe introducir el intervalo de tiempo, en segundos, que transcurrirá entre adquisiciones sucesivas.

- En *Samples per Aquisición* se establece el número de adquisiciones de datos que se tomarán en cada intervalo de adquisición.

- En *Acquisition processing* el usuario establece si desea o no efectuar algún tipo de procesado, de entre los disponibles, al conjunto de datos adquiridos en cada periodo de adquisición. Los procesados disponibles son el cálculo de la media o el cálculo de la mediana.

Al pulsar el botón *Upload*, los datos de configuración del instrumento son enviados al *datalogger* para su configuración.

Los datos introducidos en *Acquisition Interval* y en *Samples per Acquisition* deben ser enteros positivos. Para asegurar que los datos introducidos por el usuario son válidos, el cliente efectúa el correspondiente chequeo de sus valores antes de enviar los datos al *datalogger*. En caso de detectar un valor inadecuado, muestra la correspondiente ventana de aviso al usuario y los datos de configuración del instrumento no son enviados al *datalogger* (figura 42).

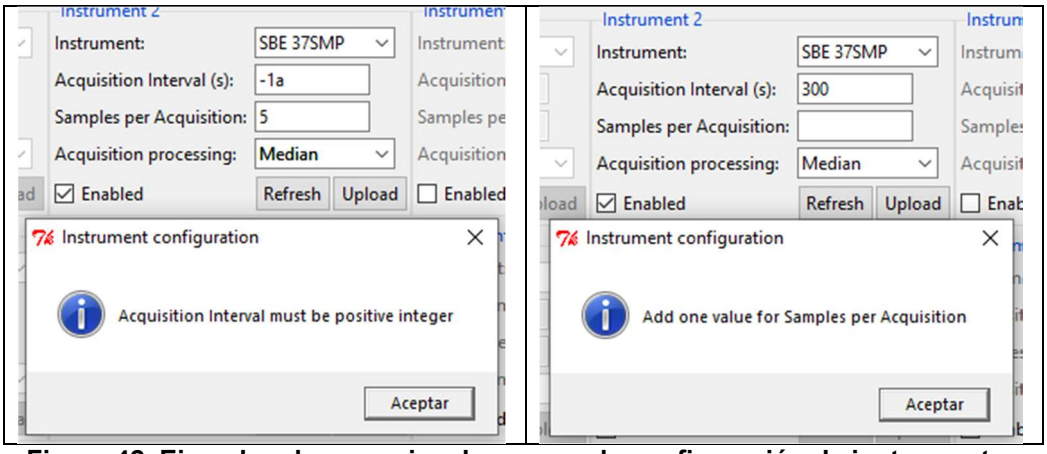


Figura 42. Ejemplos de mensajes de error en la configuración de instrumentos

Al pulsar el botón *Refresh*, el cliente solicita al *datalogger* la información de configuración que tiene almacenada para el instrumento seleccionado.

Con la información recibida desde el *datalogger* el cliente actualiza el contenido de los controles de configuración.

El botón *Upload All* envía al *datalogger* para su almacenamiento los datos de configuración de todos los instrumentos.

El botón *Refresh All* solicita al datalogger los datos que tiene almacenados de configuración de todos los instrumentos y actualiza el contenido de los controles de la GUI de cada uno de los instrumentos.

3.2.3.2 Pantalla de adquisición

La pantalla de adquisición del cliente se muestra en la figura 43

Figura 43. Pantalla de adquisición del cliente

La pantalla de adquisición dispone de controles e indicadores agrupados por funcionalidades que se describen en los siguientes apartados

Start/Stop Acquisition

Actuando sobre este grupo de controles se pueden iniciar y finalizar las adquisiciones de datos. Sus componentes y funciones se describen a continuación.

- *Start now*. Cuando está seleccionada la opción *Start now*, la adquisición comienza inmediatamente después de pulsar el botón *Start Acquisition*.

- *Start at*. Cuando está seleccionada esta opción, el usuario debe introducir la fecha y hora en la que desea que inicie la adquisición. La información de fecha y hora debe estar en el formato MM/DD/AA hh:mm:ss.

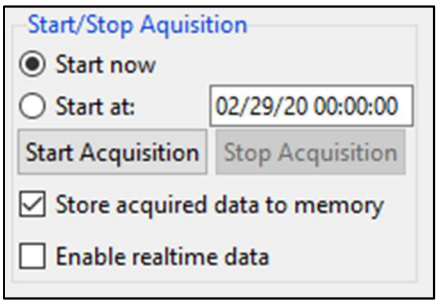


Figura 44. Start/Stop Acquisition

- *Botón Start Acquisition*. Al pulsar este botón, el cliente envía al *datalogger* el comando de paso al modo adquisición. En caso de estar seleccionada la opción de "*Start at*", el cliente chequea, en primer lugar, que el formato de fecha y hora introducidos por el usuario es el adecuado, para posteriormente chequear que cada uno de los campos tienen valores válidos. En este chequeo se tienen en cuenta también los años bisiestos. Finalmente se chequea que la fecha y hora introducido por el usuario son posteriores a la fecha y hora actuales del *datalogger*, para lo que se efectúa una solicitud de fecha y hora al propio *datalogger*.

  En caso de detectarse algún dato no válido, se muestra al usuario un mensaje indicando el problema detectado y no se envía el comando de inicio de adquisición al *datalogger*.

Figura 45. Mensaje de fecha de inicio de adquisición incorrecta

- *Store acquired data to memory*. Cuando esta opción está seleccionada, el cliente indica al *datalogger* que los datos tomados durante la adquisición deben ser almacenados en la memoria de datos.

- *Enable realtime data*. Cuando está habilitada esta opción, al iniciar la adquisición de datos, el cliente recibe de forma periódica los últimos datos adquiridos por el *datalogger*. Estos datos son mostrados en la zona de visualización de datos en tiempo real y almacenados en el archivo seleccionado por el usuario. La zona de visualización de datos en tiempo real es la de zona de fondo negro localizada a la derecha de los controles e indicadores de la pantalla de adquisición.

Memory management

Está compuesto por dos botones y un indicador.

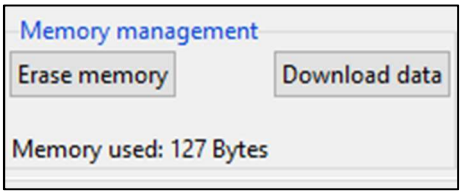


Figura 46. Memory management

- *Erase Memory*. Al pulsar este botón, el cliente envía el comando de borrado de memoria de datos al *datalogger*. Antes de efectuar el envío del comando al *datalogger* solicita confirmación al usuario para evitar posibles pérdidas de datos inadvertidas (figura 47)

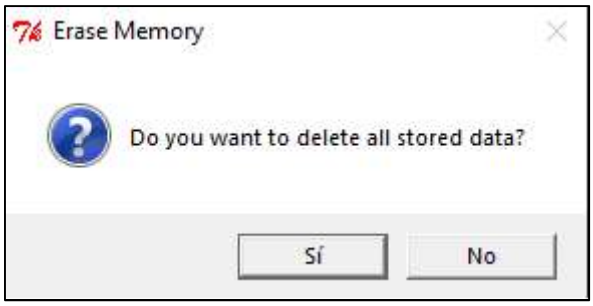


Figura 47. Confirmación de borrado de memoria

- *Download data*. Por medio de este botón el usuario puede descargar en un archivo en formato ASCII todos los datos almacenados en la memoria del *datalogger*. El cliente solicita al usuario el nombre y ubicación del archivo de datos y posteriormente envía los comandos de descarga de datos de memoria al *datalogger* para recibir la información almacenada en su memoria de datos.

- *Memory used*. este indicador muestra el número de bytes de datos almacenados en la memoria del *datalogger*. Cada vez que el *datalogger* pasa del modo adquisición al modo administración, el cliente solicita al *datalogger* la información del número de bytes almacenados en memoria para actualizar el valor de *Memory uded*. De igual manera se actualiza cuando se inicializa la memoria por medio del botón *Erase Memory.*

Set/Get Datetime

Este grupo está destinado al control y visualización del reloj del *datalogger*, está compuesto por los siguientes controles e indicadores:

- Botón *Get Date Time*. Al pulsar este botón, el cliente solicita al *datalogger* información sobre la fecha y hora de su RTC y la muestra

bajo la etiqueta “*Datalogger's date/time*”, al mismo tiempo muestra la fecha y hora del computador donde se ejecuta el cliente y lo muestra bajo la etiqueta “*Client's date/time*”

- Botón *Set Date Time*. Al pulsar este botón, el cliente envía al *datalogger* la fecha y hora del computador donde se ejecuta. El *datalogger* actualiza su RTC con la fecha y hora recibida desde el cliente.

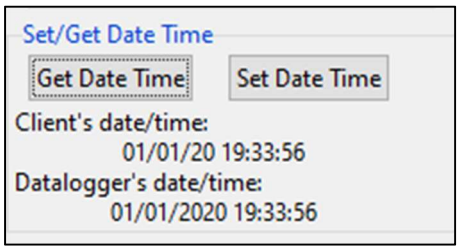


Figura 48. Set/Get date time

3.4 Resultados

El nivel de desarrollo alcanzado ha permitido efectuar una prueba de funcionamiento del sistema adquiriendo datos reales de un SBE37 SMP. Este es un instrumento habitual en observatorios oceánicos a nivel global con capacidad para proporcionar datos del máximo nivel de exactitud alcanzable en observaciones de temperatura, conductividad, presión y oxígeno disuelto en agua marina.

3.4.1 Configuración de la prueba de funcionamiento

Se ha configurado el *datalogger* para la adquisición de una lectura de datos del SBE 37 SMP cada cinco minutos. La prueba se efectuó durante un día aproximadamente de forma ininterrumpida. Los datos adquiridos fueron almacenados en la memoria interna del *datalogger*.

Durante el periodo de adquisición el SBE 37 SMP fue introducido en un recipiente con agua marina para la simulación de condiciones reales de uso.

Durante la prueba de funcionamiento el *datalogger* fue alimentado con una batería de plomo ácido de 12 Vdc y 1.200 mAH conectada en el conector J9 del *datalogger* (ver anexo B). El SBE 37 SMP fue alimentado por su propio sistema de baterías interna, basado en un *pack* de pilas no recargables de Li-$SOCl_2$.

La figura 49 muestra el montaje efectuado para la prueba de funcionamiento con el *datalogger* adquiriendo datos del SBE37 SMP.

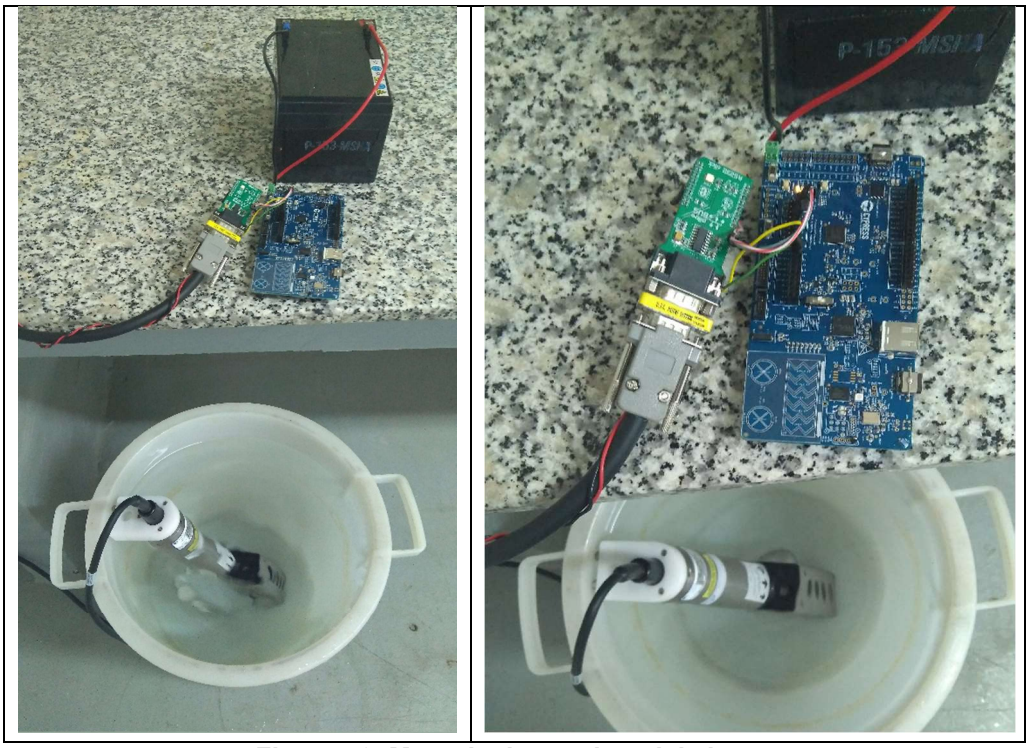


Figura 49. Montaje de pruebas del sistema

3.4.2 Resultado de la prueba de funcionamiento

La figura 50 muestra parte de los datos descargados de la memoria del *datalogger*. Los datos pueden ser visualizados desde cualquier editor de texto con capacidad para mostrar caracteres ASCII.

```
Instrument 0:  20.2380,  5.02305,    0.319,  2.307,  36.6952, 12/21/2019 13:55:10
Instrument 0:  20.2358,  5.02277,    0.344,  2.312,  36.6947, 12/21/2019 14:00:10
Instrument 0:  20.2377,  5.02291,    0.319,  2.316,  36.6942, 12/21/2019 14:05:10
Instrument 0:  20.2332,  5.02251,    0.319,  2.319,  36.6949, 12/21/2019 14:10:10
Instrument 0:  20.2325,  5.02241,    0.319,  2.324,  36.6947, 12/21/2019 14:15:10
Instrument 0:  20.2312,  5.02250,    0.319,  2.326,  36.6966, 12/21/2019 14:20:10
Instrument 0:  20.2268,  5.02205,    0.330,  2.330,  36.6966, 12/21/2019 14:25:10
Instrument 0:  20.2278,  5.02205,    0.330,  2.337,  36.6958, 12/21/2019 14:30:10
Instrument 0:  20.2313,  5.02229,    0.333,  2.339,  36.6947, 12/21/2019 14:35:10
Instrument 0:  20.2284,  5.02192,    0.333,  2.344,  36.6942, 12/21/2019 14:40:10
Instrument 0:  20.2290,  5.02186,    0.330,  2.349,  36.6932, 12/21/2019 14:45:10
Instrument 0:  20.2200,  5.02147,    0.344,  2.351,  36.6978, 12/21/2019 14:50:10
```

Figura 50. Muestra de datos adquridos en prueas de funcionamiento

Donde la primera columna indica el instrumento del que proceden los datos. La segunda columna muestra el valor de temperatura en ºC, la

tercera columna muestra la conductividad en S/m, la cuarta columna es la presión a la que se encuentra el instrumento en dbar, la quinta columna es el valor de oxígeno disuelto en ml/L y la sexta columna contiene el valor de la salinidad del agua en PSU. Todos estos datos son acompañados de la fecha y hora del RTC del *datalogger* en el momento en que fueron adquiridos.

Como muestra de las capacidades de adquisición del dispositivo *datalogger*, se han representado gráficamente los datos adquiridos durante el periodo de prueba.

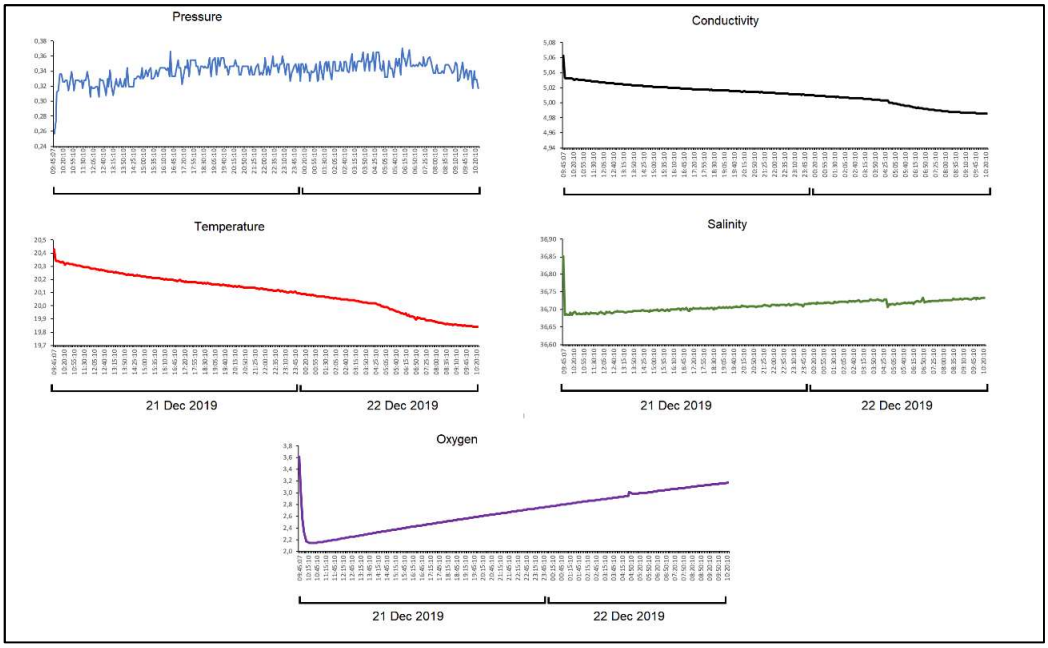


Figura 51. Representación gráfica de datos adquiridos

En la representación gráfica de los datos adquiridos se observa un pequeño intervalo, de unos 10 minutos, al inicio de la adquisición en la que todos los sensores estabilizan los valores de los parámetros adquiridos. Este periodo de estabilización es habitual en instrumentos al inicio de las adquisiciones. A partir del periodo inicial de estabilización, los valores adquiridos se corresponden con valores normales de temperatura, conductividad y salinidad en una muestra de agua marina de la región y su posible evolución a lo largo de un día en un recipiente. Se puede observar una tendencia a la disminución de la temperatura, debido a la variación de la temperatura ambiente en la que se encuentra la muestra de agua. Al disminuir la temperatura, disminuye la conductividad del agua, ya que la movilidad de los iones disminuye con la temperatura. El pequeño aumento en la salinidad es debido a la evaporación de moléculas de agua, que deriva en un aumento en la concentración de sales en la muestra. Por útimo, el aumento en la concentración de oxígeno disuleto es debida a la presencia de organismos fotosintéticos

Esta prueba del funcionamiento del *datalogger* representa una muestra de la potencialidad del sistema implementado.

4. Conclusiones y líneas de trabajo futuras

4.1 Conclusiones

Se ha diseñado e implementado un sistema *datalogger* con capacidad para adquirir, guardar y enviar en tiempo real a un programa cliente, datos de hasta ocho instrumentos oceanográficos y meteorológicos.

El *datalogger* diseñado e implementado proporciona al usuario un sistema sencillo y de configuración flexible, en el que sólo es necesario establecer los parámetros propios de sus necesidades de adquisición (intervalo de medida, número de adquisiciones por intervalo, etc).

En el estado de desarrollo actual, el sistema es capaz de adquirir y procesar datos de instrumentos SBE37 SMP. Se trata de instumentos habituales en los observarorios científicos oceanográficos a nivel global para la medición de temperatura, conductividad, salinidad y oxígeno disuelto en agua marina. De esta manera el *datalogger* es ya un sistema que puede ser integrado en cualquier plataforma que le suministre alimentación de entre 5 a 12 Vdc en un compartimento estanco que lo dote de protección frente al entorno marino. En este contexto el *datalogger* puede actuar como sistema de adquisición de datos en modo autocontenido. Estos datos pueden ser descargados de la memoria del *datalogger* una vez finalizado el periodo de adquisición.

La funcionalidad de adquisición de datos en tiempo real está indicada para entornos en los que el cliente puede permanecer conectado a la red inalámbrica del *datalogger* durante la adquisición de datos. Se trata de aplicaciones de adquisición de datos en ambientes portuarios, costeros o adquisición de datos oceánicos bordo de buques en las que el cliente pueda estar conectado al *datalogger*.

En lo que al consumo energético respecta, el sistema de comunicaciones por el estándar IEEE802.11n es el más demandante en el aspecto energético. El diseño del *datalogger* puede ser optimizado en este sentido si se añade la opción de deshabilitar y habilitar las comunicaciones inalámbricas a petición del usuario, ya que en aplicaciones de adquisición de datos de modo autocontenido sin envío de información en tiempo real no es necesario el mantenimiento del punto de acceso a la red inalámbrica.

A nivel académico, la ejecución de este TFM supone la primera incrusión del autor en el diseño y desarrollo de sistemas electrónicos embebidos. La selección de la tarjeta CY8CKIT-062-WIFI-BT como tarjeta base del *datalogger*, el SDK de WICED, y la documentación disponible tanto de la tarjeta como del SDK, han permitido al autor el seguimiento de una curva de aprendizaje, partiendo desde cero, que ha posibilitado la implementación del sistema *datalogger* descrito en este documento dentro de la planificación temporal establecida.

De igual manera, la implementación del programa cliente ha precisado del seguimiento de un proceso de aprendizaje tanto del lenguaje de programación *python* como del módulo ttk, cuyos componentes se emplean en la GUI del cliente.

La selección de la tarjeta base del *datalogger*, el seguimiento de las correspondientes curvas de aprendizaje y el diseño e implementación de todo el sistema se ha podido efectuar dentro de los dos meses destinados al diseño e implementación, según la planificación establecida por la asignatura de TFM. La prueba de funcionamiento descrita en el apartado de resultados se ha efectuado dentro del periodo destinado a la redacción de esta memoria.

4.2 Líneas de trabajo futuras

Las líneas de trabajo futuras indentificadas para este trabajo se citan a continuación.

- Integración de nueva instrumentación meteorológica y oceanográfica. Para dotar al sistema de mayor versatilidad y ubicuidad en cuanto a plataformas en las que puede ser usado, es necesario añadir la capacidad de trabajar con el mayor número de sensores oceanográficos y meteorológios. En el capítulo del estado del arte de esta memoria se citan lo sensores e instrumentos más comunes en los observatorios desplegados por el platena cuya integración en el *datalogger* implementado es de gran interés.

- Integración de capacidad de comunicaciones satélite. Las comunicaciones por satélite dotarán al sistema de capacidad para el envío de datos en tiempo real desde localizaciones remotas. En este sentido la constelación *Iridium* proporciona cobertura global, su servicio *Short Burst Data (SBD)[referencia]* es una buena opción para la transmisión de datos de los sensores y existe en el mercado dispositivos MODEM[referencia] con antena integrada y API sencilla que pueden ser integrados en el *datalogger*.

- Optimización del factor de forma con el diseño de una tarjeta PCB específica para el *datalogger*. La tarjeta CY8CKIT-062-WIFI-BT tiene integrados varios módulos que no son empleados en el *datalogger*. Se plantea el diseño de una tarjeta únicamente con los módulos necesarios para el *datalogger*. Estos módulos son los incluidos en los esquemáticos del anexo B.

  Como sistema adaptador RS232-C se puede prescindir de las tarjetas RS232 click e integrar directamente los circuitos integrados MAX3232 en la nueva PCB.

- Diseño de un compartimento estanco que permita al dispositivo ser directamente puesto en contacto con el agua marina e incluso ser sumergido en un fondeo.

- Se plantea la adaptación del dispositivo datalogger para que, además de poder ser empleado diferentes plataformas y estaciones fijas y móviles, pueda ser adherido a animales marinos con sensores minuaturizados para la monitorización de sus movimientos y de parámetros oceanográfícos. En este sentido existe literatura en la que se emplean mamíferos marinos para la monitorización de parámetros medioambientales (38) (39).

- Implementar en el *firmware* del *datalogger* un modo específico de bajo consumo en el que se deshabilite la red WiFi cuando no se vaya a hacer uso de ella durante un periodo de adquisición de datos. En caso de que el usuario desee volver a disponer de la red WiFi para interactuar con el *datalogger* podrá pulsar un botón que haga al datalogger habilitar la red *MetOcean Datalogger*.

- Dotar al cliente de la capacidad de mostrar los datos recibidos en tiempo real de forma gráfica.

5. Glosario

Las siguientes tablas contienen la descripción de los acrónimos y unidades de medida empleadas a lo largo de esta memoria

Aconimo	Definición
ADC	*Analog to Digital Converter*
AES	*Advanced Encryption Standard*
AP	*Access Point*
API	*Application Programming Interface*
ASCII	*American Standard Code for Information Interchange*
BSSID	*Basic Service Set Identifier*
CCK	*Complementary Code Keying*
CCMP	*Counter Mode Cipher Block Chaining Message Authentication Code Protocol*
CMOS	*Complementary Metal Oxide Semiconductor*
CPU	*Central Processing Unit*
DHCP	*Dynamic Host Configuration Protocol*
DMA	*Direct Memory Access*
DSSS	*Direct Sequence Spread Spectrum*
EMI	*Electromagnetic Interference*
FIFO	*First In First Out*
FLL	*Frequency Locked Loop*
GND	*Ground*
GPIO	*General Purpose Input Output*
GPRS	*General Packet Radio Service*
GPS	*Global Positioning System*
GSM	*Global System for Mobile*
GUI	*Graphical User Interface*
HSIOM	*High Speed Input Output Matrix*
IDE	*Integrated Development Environment*
IEEE	*Institute of Electrical and Electronic Engineer*
ILO	*Internal Low-speed Oscillator*
IMO	*Internal Main Oscillator*
IoT	*Internet of things*
IP	*Internet Protocol*
IPv4	*Internet Protocol version 4*
ISR	*Interrupt Service Rutine*
LoRa	*Long Range*
LSB	*Less Significant Byte*
LTE	*Long Term Evolution*
LVTTL	*Low Voltage Transistor to Transistor Logic*
MAC	*Media Access Control*
MCU	*Microcontroller Unit*
MIC	*Message Integrity Check*
MPU	*Memory Protection Unit*
MSB	*Most Significant Byte*
NVIC	*Nested Vector Interrupt Control*
OFDM	*Orthogonal Frequency Division Multiplexing*
PC	*Personal Computer*
PCB	*Printed Circuit Board*
PILO	*Precision Internal Low-speed Oscillator*
PLL	*Phase Locked Loop*
PSoC	*Programable Systen on Chip*
QSPI	*Quad Serial Peripheral Interface*
ROM	*Read Only Memory*
RTOS	*Real Time Operating System*
SDIO	*Secure Digital Input Output*
SDK	*Software Development Kit*
SMIF	*Serial Memory Interface*
SPI	*Serial Peripheral Interface*
SRAM	*Static Random Access Memory*
SSID	*Service Set Identifier*
TCP	*Transmission Control Protocol*
TFM	Trabajo Fin de Máster
TKIP	*Temporal Key Integrity Protocol*
UART	*Universal Asynchronous Receiver/Transmitter*
USB	*Universal Serial Bus*
WEP	*Wired Equivalent Privacy*
WIC	*Wakeup Interrupt Controller*
WICED	*Wireless Internet Connectivity for Embedded Devices*
WCO	*Watch Crystal Oscillator*
WiFi	*Wireless Fidelity*
WLAN	*Wireless Local Area Network*
WPA	*Wireless Protected Access*
WPA2-PSK	*WiFi Protected Access 2- Pre Shared Key*

Tabla 10. Definición de acrónimos

Acronimo	Unidad de medida
ºC	Grado centígrado
GB	Giga Byte
GHz	Giga Hercio
Hz	Hercio
kB	Kilo Byte (1024 Bytes)
kg	Kilogramo
mA	Miliamperio
mAH	Miliamperio Hora
mL/l	Mililitros litro
ms	Milisegundo
Mb	Mega bit
Mbps	Mega bits por segundo
MB	Mega Byte
MHz	Mega Hercio (10^6 Hercios)
PSU	*Practical Salinity Unit*
S/m	Siemens metro
V	Voltios
Vdc	*Volts direct current - Tensión contínua*

Tabla 11. Definición de unidades

6. Bibliografía

1. **Comission, European.** Copernicus. Europe's eye on Earth. [En línea] [Citado el: 5 de 10 de 2019.] https://www.copernicus.eu/es.
2. **European Comission.** Copernicus Marine Environment Monitoring System. [En línea] [Citado el: 5 de 10 de 2019.]
3. **FixO3.** Fixed-Point Open Ocean Observatories. [En línea] [Citado el: 5 de 10 de 2019.]
4. **EMSO ERIC.** European Multidisciplinary Seafloor and water comlumn Observatory. [En línea] [Citado el: 5 de 10 de 2019.] http://emso.eu/.
5. **EMSODev.** European Multidisciplinary seafloor and water column observatorie. [En línea] [Citado el: 5 de 10 de 2019.] http://www.emsodev.eu/index.html.
6. **AtlantOS.** Optimising and Enhancing the Integrated Atlantic Ocean Observing Systems. [En línea] 2019. [Citado el: 5 de 10 de 2019.] https://www.atlantos-h2020.eu/.
7. **PIRATA.** Preducción and Research Moored Array in the Tropical Atlantic. [En línea] [Citado el: 5 de 10 de 2019.] http://www.brest.ird.fr/pirata/.
8. **National Oceanic and Atmospheric Administration.** Global Tropical Moored Buoy Array. *Ocean RAMA.* [En línea] [Citado el: 5 de 10 de 2019.] https://www.pmel.noaa.gov/gtmba/pmeltheme/indian-ocean-rama.
9. Global Tropical Moored Buoy Array. *Pacific Ocean TAO.* [En línea] [Citado el: 5 de 10 de 2019.] https://www.pmel.noaa.gov/gtmba/pmeltheme/pacific-ocean-tao.
10. *Technology Refresh of NOAA's Tropical Atmosphere Ocean (TAO) Buoy System.* **Chung-Chu Teng, Landry.** s.l. : IEEE, 2006.
11. **TPOS 2020.** Tropial Pacific Observing Systems. [En línea] 2016. [Citado el: 5 de 10 de 2019.] http://tpos2020.org/.
12. **OceanSITES.** Taking the pulse of the global ocean. [En línea] [Citado el: 8 de 10 de 2019.] http://www.oceansites.org/.
13. **Deep Ocean Observing Strategy.** [En línea] 2016. [Citado el: 8 de 10 de 2019.] http://deepoceanobserving.org/.
14. **Fugro.** Seawatch Metocean Buoys and Sensors. [En línea] 2019. [Citado el: 9 de 10 de 2019.] https://www.fugro.com/about-fugro/our-expertise/technology/seawatch-metoceanbuoys-and-sensors.
15. **Hellenic Centre for Marine Research.** Poseidon System. [En línea] 2012. [Citado el: 9 de 10 de 2019.] http://poseidon.hcmr.gr/index.php.
16. **Campbell Scientific.** Dataloggers y Sistemas de Adquisición de Datos. [En línea] 2019. [Citado el: 9 de 10 de 2019.] https://www.campbellsci.es/data-loggers.
17. **PLOCAN.** Observatorio ESTOC. [En línea] 2019. [Citado el: 9 de 10 de 2019.] http://siboy.plocan.eu/ESTOC.
18. **NexSens Technology.** Real-time Environmental Data. [En línea] 2019. [Citado el: 10 de 10 de 2019.] https://www.nexsens.com/.
19. **Observator group.** OMC-045-III GPRS Data Logger. [En línea] [Citado el: 10 de 10 de 2019.] https://observator.com/en/meteo-hydro/products/data-loggers-signal-conditioning/.

20. **Skye.** DataHog Datalogger. [En línea] 2019. [Citado el: 10 de 10 de 2019.] https://www.skyeinstruments.com/products/light-sensors-systems/metersdataloggers/datahog-datalogger/.
21. **KUNAK.** Sensing Anywhere. [En línea] 2019. [Citado el: 10 de 10 de 2019.] https://www.kunak.es/productos/data-loggers/datalogger-2g-4g-lte-nb-iot-para-exteriores/.
22. **ONSET.** HOBO RX3000 Remote Monitoring Station Data Logger. [En línea] 2019. [Citado el: 10 de 10 de 2019.] https://www.onsetcomp.com/products/data-loggers/rx3000.
23. **Cypress Semiconductor.** *PSoC 6 WiFi-BT Pioneer Kit Guide.* San Jose, CA : Cypress Semiconductor Corporation, 2019.
24. —. *PSoC 6 MCU: CY8C62x6, CY8C62x7 Architecture Technical Reference Manual.* San Jose, CA : Cypress Semiconductor Corporation, 2019.
25. **Developer, ARM.** Cortex-M4 Documentation. [En línea] [Citado el: 15 de 11 de 2019.] https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4/docs.
26. —. Cortex-M0 Plus Documentation. [En línea] [Citado el: 15 de 11 de 2019.] https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0-plus/docs.
27. **Cypress Semiconductor.** *S25FL512S. 512 Mbit (64 Mbyte), 3.0 V SPI Flash Memory.* San José, CA : Cypress Semiconductor Corporation, 2019.
28. **Murata.** Type 1DX. [En línea] 2019. [Citado el: 20 de 11 de 2019.] https://wireless.murata.com/type-1dx.html.
29. **Cypress Semiconductor.** *CYW4343W. Single-Chip 802.11 b/g/n MAC/Baseband/Radio with Bluetooth 4.1.* San Jose, CA : Cypress Semiconductor Corporation, 2017.
30. **SD Card Association.** *SDIO Simplified Specifications. Version 3.0.* s.l. : SD Card Association, 2018.
31. **Mikroe.** RS232 Cliick. [En línea] [Citado el: 28 de 11 de 2019.] https://www.mikroe.com/rs232-click.
32. **Texas Instruments.** *MAX3232 3-V to 5.5-V Multichannel RS-232 Line Driver/Receiver With ±15-kV ESD Protection.* s.l. : Texas Instruments, 2000.
33. —. Pump it up with charge pumps. [En línea] [Citado el: 20 de 12 de 2019.] https://e2e.ti.com/blogs_/b/powerhouse/archive/2016/02/01/charge-it-up-with-charge-pumps-part-1.
34. —. *bq24266 3-A, 30-V Standalone Single-Input, Single-Cell Switchmode Li-Ion Battery Charger.* s.l. : Texas Instruments, 2015.
35. **Cypress Semiconductor.** WICED Studio. [En línea] [Citado el: 15 de 10 de 2019.] https://www.cypress.com/products/wiced-software.
36. **Microsoft.** Express Logic. [En línea] [Citado el: 3 de 11 de 2019.] https://rtos.com/.
37. **Sea Bird Scientific.** *SBE 37-SMP-ODO MicroCAT C-T-ODO (P) Recorder. User Manual.* Bellevue, Washington : SeaBird Electronics, 2015.

38. *Animal-borne CTD-Satellite Relay Data Loggers for real-time oceanographic data collection.* **Boheme L. et al.** 5, s.l. : Ocean Science, 2009.
39. *Salinity sensors on seals:use of marine predators to carry CTD dataloggers.* **Sascha K. Hooker.** s.l. : Elseiver, 2003.

7. Anexos

Anexo A. Código implementado

A.1 *Firmware*

Archivo datalogger.mk

```
#
# $ Copyright José Raúl Santana Jiménez $
#
NAME := app_METEOCEANOGRAPHIC_DATALOGGER

$(NAME)_SOURCES    := datalogger.c \
                                                    datalogger_MetOcean_TCP_protocol.c \
                                                    datalogger_Sensors.c \
                                                    datalogger_MetOcean_Commands.c \
                                                    datalogger_Acquire.c \
                                                    datalogger_Serial_Port.c \
                                                    datalogger_memory.c

$(NAME)_COMPONENTS += inputs/gpio_button
WIFI_CONFIG_DCT_H := wifi_config_dct.h
```

wifi_config_dct.h

```
#pragma once

#ifdef __cplusplus
extern "C" {
#endif


/* This is the soft AP available for normal operation */
#define SOFT_AP_SSID         "MetOcean Datalogger"
#define SOFT_AP_PASSPHRASE   "abcd1234"
#define SOFT_AP_SECURITY     WICED_SECURITY_WPA2_AES_PSK
#define SOFT_AP_CHANNEL      1



#ifdef __cplusplus
} /*extern "C" */
#endif
```

Archivo datalogger.c

```
/*
 * datalogger.c
 *
 *  Created on: 3/11/2019
 *      Author: José Raúl Santana Jiménez
 */
#include "wiced.h"
#include "datalogger_MetOcean_TCP_protocol.h"
#include "datalogger_MetOcean_Commands.h"
#include "datalogger_Acquire.h"
#include "datalogger_general.h"
#include "datalogger_Sensors.h"
#include "datalogger_Serial_Port.h"
#include "datalogger_memory.h"
/*****************************************************
```

```
 *                      Macros
 ******************************************************/

#define TCP_SERVER_LISTEN_PORT             (7777)
#define TCP_PACKET_MAX_DATA_LENGTH         (30)


#define THREAD_PRIORITY_ACQUISITION_MODE     (10)
#define THREAD_STACK_SIZE_ACQUISITION_MODE   (1024)



/******************************************************
 *                    Enumerations
 ******************************************************/
typedef enum
{
    SBE37_SM,
    OPTODE_3835,
    WETLABS_FLNTU
}datalogger_instruments_t;

/******************************************************
 *               Static Function Declarations
 ******************************************************/

static wiced_result_t client_connected_callback   ( wiced_tcp_socket_t* socket, void* arg );
static wiced_result_t client_disconnected_callback( wiced_tcp_socket_t* socket, void* arg );
static wiced_result_t received_data_callback      ( wiced_tcp_socket_t* socket, void* arg );

/******************************************************
 *               Variable Definitions
 ******************************************************/

static wiced_tcp_socket_t tcp_server_socket;
static datalogger_protocol_command_t struct_command;
datalogger_instrument_t datalogger_available_instruments[MAX_NUMBER_OF_INSTRUMENTS];

wiced_utc_time_t datalogger_datetime;
wiced_iso8601_time_t datalogger_datetime_iso;

static wiced_thread_t ThreadHandle_Acquisition_Mode;

static wiced_semaphore_t semaphoreHandle_Acquisition_Mode;

static const wiced_ip_setting_t device_init_ip_settings =
{
    INITIALISER_IPV4_ADDRESS( .ip_address, MAKE_IPV4_ADDRESS(192, 168, 10,  1) ),
    INITIALISER_IPV4_ADDRESS( .netmask,    MAKE_IPV4_ADDRESS(255, 255, 255, 0) ),
    INITIALISER_IPV4_ADDRESS( .gateway,    MAKE_IPV4_ADDRESS(192, 168, 10,  1) ),
};

/******************************************************
 *               Function Definitions
 ******************************************************/
void datalogger_enable_tcp_server(void) {
    wiced_result_t result;

    /* Bring up the network interface */
    wiced_network_up( WICED_AP_INTERFACE, WICED_USE_INTERNAL_DHCP_SERVER,
                                                    &device_init_ip_settings );

    /* Create a TCP server socket */
    if ( wiced_tcp_create_socket( &tcp_server_socket, WICED_AP_INTERFACE ) != WICED_SUCCESS )
    {
        WPRINT_APP_INFO( ("TCP socket creation failed\r\n") );
    }

    /* Register callbacks to handle various TCP events */
    result = wiced_tcp_register_callbacks( &tcp_server_socket, client_connected_callback,
                            received_data_callback, client_disconnected_callback, NULL );

    if ( result != WICED_SUCCESS )
    {
```

```
        WPRINT_APP_INFO( ("TCP server socket initialization failed\r\n") );
    }

    /* Start TCP server to listen for connections */
    if ( wiced_tcp_listen( &tcp_server_socket, TCP_SERVER_LISTEN_PORT ) != WICED_SUCCESS )
    {
        WPRINT_APP_INFO( ("TCP server socket initialization failed\r\n") );
        wiced_tcp_delete_socket( &tcp_server_socket );
        return;
    }

    WPRINT_APP_INFO( ("Async tcp server started. Listening on port %d \r\n",
                                            TCP_SERVER_LISTEN_PORT) );

}


/* Define the thread function that will set the aquisition mode of the datalogger */
void acquisition_mode_thread(wiced_thread_arg_t arg){
    datalogger_mode_satates_t datalogger_prev_mode;
    wiced_rtos_delay_milliseconds(500);
    datalogger_prev_mode = datalogger_actual_mode;
    while(1){
        if (datalogger_actual_mode == CONFIGURING){
            //Here stops the acquisition threads
            //instruments_acquiring_stop = 1;
            set_stop_instrument_acquiring(1);
            datalogger_prev_mode = CONFIGURING;
            wiced_rtos_get_semaphore(&semaphoreHandle_Acquisition_Mode, WICED_WAIT_FOREVER);
            wiced_rtos_delay_milliseconds(1000);
        }
        if ((datalogger_prev_mode == CONFIGURING)&& (datalogger_actual_mode == ACQUIRING)) {
            //instruments_acquiring_stop=0;
            set_stop_instrument_acquiring(0);
            acquisition_mode_start(&datalogger_available_instruments);
        }
        datalogger_prev_mode = datalogger_actual_mode;
        wiced_rtos_delay_milliseconds(100);
    }
}


void datalogger_init(void){
    //Initialize the datalogger_available_instruments array
    for (int i = 0; i<MAX_NUMBER_OF_INSTRUMENTS; i++){
        datalogger_available_instruments[i].enabled = 0;
        datalogger_available_instruments[i].instrument_type = NONE_It;
    }

    //Set up the datalogger mode semaphore handle
    wiced_rtos_init_semaphore(&semaphoreHandle_Acquisition_Mode);
    /* Initialize and start Acquisition MODE thread */
    wiced_rtos_create_thread(&ThreadHandle_Acquisition_Mode, THREAD_PRIORITY_ACQUISITION_MODE,
"Thread_Acquisition_Mode", acquisition_mode_thread, THREAD_STACK_SIZE_ACQUISITION_MODE, NULL);
    /* Initialize the datalogger serial ports */
    datalogger_serial_ports_init();
    /* Set up the instruments semaphore handlers and Initialize and start instruments threads */
    acquisition_mode_init();
    datalogger_get_memory_size();
    if (!datalogger_memory_ready()){
        datalogger_memory_init();
    }
    wiced_time_get_utc_time(&datalogger_datetime);
    wiced_time_get_iso8601_time(&datalogger_datetime_iso);
}

void application_start( void )
{
    datalogger_mode_satates_t datalogger_prev_mode;
    /* Initialize the device and WICED framework */
    wiced_init( );
    datalogger_init();
    datalogger_enable_tcp_server();
    datalogger_prev_mode = datalogger_actual_mode;
```

```
    while (1) {
        if((datalogger_actual_mode == ACQUIRING)&&(datalogger_prev_mode == CONFIGURING)){
            // Set the Acquiring state semaphore
            wiced_rtos_delay_milliseconds(datalogger_programmed_acquire_mode_delay);
            wiced_rtos_set_semaphore(&semaphoreHandle_Acquisition_Mode);
        }
        datalogger_prev_mode = datalogger_actual_mode;
        wiced_rtos_delay_milliseconds(200);
    }
}

static wiced_result_t client_connected_callback( wiced_tcp_socket_t* socket, void* arg )
{
    wiced_result_t      result;
    wiced_ip_address_t  ipaddr;
    uint16_t            port;

    UNUSED_PARAMETER( arg );

    /* Accept connection request */
    result = wiced_tcp_accept( socket );
    if( result == WICED_SUCCESS )
    {
        /* Extract IP address and the Port of the connected client */
        wiced_tcp_server_peer( socket, &ipaddr, &port );

        WPRINT_APP_INFO(("Accepted connection from :: "));

        WPRINT_APP_INFO ( ("IP %u.%u.%u.%u : %d\r\n", (unsigned char) ( (
GET_IPV4_ADDRESS(ipaddr) >> 24 ) & 0xff ),
                                                      (unsigned char) ( (
GET_IPV4_ADDRESS(ipaddr) >> 16 ) & 0xff ),
                                                      (unsigned char) ( (
GET_IPV4_ADDRESS(ipaddr) >>  8 ) & 0xff ),
                                                      (unsigned char) ( (
GET_IPV4_ADDRESS(ipaddr) >>  0 ) & 0xff ),
                                                      port ) );

        return WICED_SUCCESS;
    }
    return WICED_ERROR;
}

static wiced_result_t client_disconnected_callback( wiced_tcp_socket_t* socket, void* arg )
{
    UNUSED_PARAMETER( arg );

    WPRINT_APP_INFO(("Client disconnected\r\n\r\n"));

    wiced_tcp_disconnect(socket);
    /* Start listening on the socket again */
    if ( wiced_tcp_listen( socket, TCP_SERVER_LISTEN_PORT ) != WICED_SUCCESS )
    {
        WPRINT_APP_INFO( ("TCP server socket re-initialization failed\r\n") );
        wiced_tcp_delete_socket( socket );
        return WICED_ERROR;
    }

    return WICED_SUCCESS;
}

static wiced_result_t received_data_callback( wiced_tcp_socket_t* socket, void* arg )
{
    wiced_result_t      result;
    wiced_packet_t*     tx_packet;
    char*               tx_data;
    wiced_packet_t*     rx_packet = NULL;
    char*               request;
    uint16_t            request_length;
    uint16_t            available_data_length;
    char*               command_received;

    result = wiced_tcp_receive( socket, &rx_packet, WICED_WAIT_FOREVER );
    if ( result != WICED_SUCCESS )
```

```
    {
        return result;
    }

    wiced_packet_get_data( rx_packet, 0, (uint8_t**) &request, &request_length,
                                                        &available_data_length );

    if (request_length != available_data_length)
    {
        WPRINT_APP_INFO(("Fragmented packets not supported\r\n"));
        return WICED_ERROR;
    }
    printf("AVAILABLE DATA LENGTH: %d\r\n", available_data_length);

    /* Null terminate the received string */
    request[request_length] = '\x0';
    WPRINT_APP_INFO(("Received data: %s \r\n", request));

    /* Send echo back */
    if ( wiced_packet_create_tcp( socket, TCP_PACKET_MAX_DATA_LENGTH, &tx_packet,
                (uint8_t**)&tx_data, &available_data_length ) != WICED_SUCCESS )
    {
        WPRINT_APP_INFO(("TCP packet creation failed\r\n"));
        return WICED_ERROR;
    }

    // Parse the received MetOcean command
    struct_command = datalogger_metocean_parser(request);
    // Execute the received MetOcean Command
    metocean_command_execute(&struct_command, datalogger_available_instruments);
    request_length = command_out_length;
    tx_data[request_length] = '\x0';
    memcpy( tx_data, &command_out[0], request_length );
    /* Set the end of the data portion */
    wiced_packet_set_data_end( tx_packet, (uint8_t*)tx_data + request_length );

    /* Send the TCP packet */
    if ( wiced_tcp_send_packet( socket, tx_packet ) != WICED_SUCCESS )
    {
        WPRINT_APP_INFO( ("TCP packet send failed\r\n") );

        /* Delete packet, since the send failed */
        wiced_packet_delete( tx_packet );
    }
    WPRINT_APP_INFO(("Echo data: %s\r\n", tx_data));

    /* Release a packet */
    wiced_packet_delete( rx_packet );
    return WICED_SUCCESS;
}
```

Archivo datalogger_general.h

```
/*
 * datalogger_general.h
 *
 *  Created on: 12/11/2019
 *      Author: José Raúl Santana Jiménez
 */

#ifndef APPS_DATALOGGER_DATALOGGER_GENERAL_H_
#define APPS_DATALOGGER_DATALOGGER_GENERAL_H_

#define MAX_NUMBER_OF_INSTRUMENTS 8
#include "datalogger_MetOcean_TCP_protocol.h"
typedef struct
{
    datalogger_type_instruments_values_t instrument_type;
    metocean_tcp_protocol_command_arguments_t acquisition_interval;
    metocean_tcp_protocol_command_arguments_t samples_per_acquisition;
    metocean_tcp_protocol_command_arguments_t processing;
    int enabled;

}datalogger_instrument_t;
```

```
extern char* command_out;

#endif /* APPS_DATALOGGER_DATALOGGER_GENERAL_H_ */
```

Archivo datalogger_memory.h

```
/*
 * datalogger_memory.h
 *
 *  Created on: 21/11/2019
 *      Author: José Raúl Satana Jiménez
 */

#ifndef APPS_DATALOGGER_DATALOGGER_MEMORY_H_
#define APPS_DATALOGGER_DATALOGGER_MEMORY_H_

#include "wiced.h"
#include "spi_flash.h"
#include "stdio.h"
#include "string.h"

typedef union
{
uint32 next_write_address;
uint8 next_write_address_bytes[4];
}SFlash_memory_pointer_t;

uint32_t datalogger_get_data_bytes_saved(void);

void datalogger_memory_init(void);

wiced_bool_t datalogger_memory_ready();

void datalogger_get_memory_size(void);

void datalogger_memory_write(uint8_t* buffer, int buffer_size);

void datalogger_memory_read(uint8_t* buffer, int buffer_size, uint32_t address_to_start);


#endif /* APPS_DATALOGGER_DATALOGGER_MEMORY_H_ */
```

Archivo datalogger_memory.c

```
/*
 * datalogger_memory.c
 *
 *  Created on: 21/11/2019
 *      Author: José Raúl Santana Jiménez
 */

#include "datalogger_memory.h"
#include <inttypes.h>

#define PACKET_SIZE (100u)


sflash_handle_t qspi_flash_handle;
sflash_write_allowed_t qspi_flash_allowed_write_in = SFLASH_WRITE_ALLOWED;
void* qspi_flash_peripheral_id=0;

unsigned long flash_size=0;
uint32_t extMemAddress = 0x1000000; // base address of the external memory
uint32_t lastMemAddress = 0x3FFFFFC;
uint32_t pointer_next_MemAddress = 0x1000000;

SFlash_memory_pointer_t memory_pointer_bytes_uint_32;
```

```
uint8_t txBuffer[PACKET_SIZE];
uint8_t rxBuffer[PACKET_SIZE];

// Check if the memory has the correct format
wiced_bool_t datalogger_memory_ready(){
    int ret=-1;
    uint8_t valid_address_indicator;
    ret = sflash_read(&qspi_flash_handle, lastMemAddress+4, &valid_address_indicator, 1);
    if (valid_address_indicator == (uint8_t)0xAA) {
        return WICED_TRUE;
    } else {
        return WICED_FALSE;
    }
}

// Initialize the memoru with the correct format
void datalogger_memory_init(void){
    int ret=-1;
    uint8_t init_valid_address_indicator[5];
    init_valid_address_indicator[0] = 0x00;
    init_valid_address_indicator[1] = 0x00;
    init_valid_address_indicator[2] = 0x00;
    init_valid_address_indicator[3] = 0x01;
    init_valid_address_indicator[3] = 0xAA;
    ret = -1;
    ret = sflash_sector_erase(&qspi_flash_handle, 16773119); // Erase the last sector on the
                                                                                    SFLASH
    sflash_sector_erase(&qspi_flash_handle, extMemAddress); // Erase the first data sector on
                                                                        the SFLASH
    //Initialize last data address and the valid address indicator 16773119
    ret = sflash_write(&qspi_flash_handle, lastMemAddress,
(uint8_t*)init_valid_address_indicator, 5);
    WPRINT_APP_INFO(("Erase Sector | status: %d\r\n", ret));
}

void datalogger_get_memory_size(void){
    int ret=-1;
    ret = init_sflash(&qspi_flash_handle, qspi_flash_peripheral_id,
                                        qspi_flash_allowed_write_in);
    sflash_get_size(&qspi_flash_handle, &flash_size);
    WPRINT_APP_INFO(("Memory Size: %lu | status: %d\r\n", flash_size, ret));
}

void datalogger_memory_write(uint8_t* buffer, int buffer_size){
    uint8_t aux_byte = 0xAA;
    int ret=-1;
    ret = -1;
    // Get the direction of the last byte written
    sflash_read(&qspi_flash_handle, lastMemAddress,
                        &memory_pointer_bytes_uint_32.next_write_address_bytes[0], 4);
    ret = sflash_write(&qspi_flash_handle, memory_pointer_bytes_uint_32.next_write_address,
                                                            buffer, buffer_size);

    memory_pointer_bytes_uint_32.next_write_address =
                        memory_pointer_bytes_uint_32.next_write_address + buffer_size;
    sflash_sector_erase(&qspi_flash_handle, 16773119);
    sflash_write(&qspi_flash_handle, lastMemAddress,
                        &memory_pointer_bytes_uint_32.next_write_address_bytes[0], 4);
    sflash_write(&qspi_flash_handle, lastMemAddress+3, &aux_byte, 1);
    ret = -1;
}

void datalogger_memory_read(uint8_t* buffer, int buffer_size, uint32_t address_to_start){
    int ret=-1;
    ret = -1;
    ret = sflash_read(&qspi_flash_handle, pointer_next_MemAddress + address_to_start, buffer,
                                                                        buffer_size);
}

uint32_t datalogger_get_data_bytes_saved(void){
    sflash_read(&qspi_flash_handle, lastMemAddress,
                                &memory_pointer_bytes_uint_32.next_write_address_bytes[0], 4);
    return memory_pointer_bytes_uint_32.next_write_address - extMemAddress;
```

```
}
```

Archivo datalogger_MetOcean_Commands.h

```
/*
 * datalogger__MetOcean_Commands.h
 *
 *  Created on: 12/11/2019
 *      Author: José Raúl Santana Jiménez
 */

#ifndef APPS_DATALOGGER_DATALOGGER_METOCEAN_COMMANDS_H_
#define APPS_DATALOGGER_DATALOGGER_METOCEAN_COMMANDS_H_
#include "datalogger_MetOcean_TCP_protocol.h"
#include "datalogger_general.h"

//command out contains the string command that will be sent to the datalogger client
extern char* command_out;
extern uint32_t command_out_length;
extern datalogger_mode_satates_t datalogger_actual_mode;
extern wiced_utc_time_ms_t datalogger_programmed_acquire_mode_delay;
extern wiced_bool_t tcp_client_connected;


void metocean_download_data(uint16_t buffer_length, uint32_t memory_address);
void metocean_realtime_data(void);
void datalogger_get_mode(void);
void datalogger_get_date_time(void);
void datalogger_set_instrument(int* num_args, datalogger_protocol_argument_t* args,
datalogger_instrument_t instruments[MAX_NUMBER_OF_INSTRUMENTS]);
void datalogger_get_instrument(int instrument_id, datalogger_instrument_t
instruments[MAX_NUMBER_OF_INSTRUMENTS]);
void metocean_command_execute(datalogger_protocol_command_t* str_command,
datalogger_instrument_t instruments[MAX_NUMBER_OF_INSTRUMENTS]);

void prueba(void);

#endif /* APPS_DATALOGGER_DATALOGGER_METOCEAN_COMMANDS_H_ */
```

Archivo datalogger_MetOcean_Commands.c

```
/*
 * datalogger__MetOcean_Commands.c
 *
 *  Created on: 12/11/2019
 *      Author: José Raúl Santana Jiménez
 */
#include "datalogger_MetOcean_Commands.h"
#include "datalogger_MetOcean_TCP_protocol.h"
#include "datalogger_general.h"
#include "datalogger_Sensors.h"
#include "wiced.h"
#include <stdio.h>
#include <string.h>
#include <inttypes.h>

char* command_out = "";
uint32_t command_out_length = 12;
datalogger_mode_satates_t datalogger_actual_mode = CONFIGURING;
wiced_utc_time_ms_t datalogger_programmed_acquire_mode_delay =0;
wiced_bool_t tcp_client_connected = WICED_FALSE;



void datalogger_get_mode(){
    unsigned char* aux_command;
    char* buffer;
    aux_command = (unsigned char *)malloc(8 * sizeof(unsigned char));
    aux_command[0] = (unsigned char)'1';
    aux_command[1] = (unsigned char)'-';
    aux_command[2] = (unsigned char)'1';
```

```
    aux_command[3] = (unsigned char)'(';
    aux_command[4] = (unsigned char)'2'; // "Instrument ID" type argument
    aux_command[5] = (unsigned char)',';
    aux_command[6] = (unsigned char)('0'+datalogger_actual_mode);
    aux_command[7] = (unsigned char) ')';
    command_out_length = 8;

    buffer=(char *)malloc(10 * sizeof(char));
    sprintf(buffer, "%s", aux_command);
    command_out = buffer;

}

// Function for Get Instrument Command.
void datalogger_get_instrument(int instrument_id, datalogger_instrument_t
instruments[MAX_NUMBER_OF_INSTRUMENTS]){
    unsigned char* aux_command;
    unsigned char* sample_interval_string = (unsigned char *)malloc(10 * sizeof(unsigned char));
    unsigned char* samples_per_acquisition_string = (unsigned char *)malloc(10 * sizeof(unsigned
                                                                                    char));
    unsigned char* processing_string = (unsigned char *)malloc(2 * sizeof(unsigned char));
    char* buffer=(char *)malloc(100 * sizeof(char));;
    aux_command = (unsigned char *)malloc(50 * sizeof(unsigned char));
    aux_command[0] = (unsigned char)'1';
    aux_command[1] = (unsigned char)'-';
    aux_command[2] = (unsigned char)'0';
    aux_command[3] = (unsigned char)'(';
    aux_command[4] = (unsigned char)'0'; // "Instrument ID" type argument
    aux_command[5] = (unsigned char)',';
    aux_command[6] = (unsigned char)('0'+instrument_id);
    aux_command[7] = (unsigned char) ';';
    aux_command[8] = (unsigned char)'1'; // "Type instrument" type argument
    aux_command[9] = (unsigned char)',';
    aux_command[10] = (unsigned char)('0'+ instruments[instrument_id].instrument_type);
    aux_command[11] = (unsigned char) ';';
    aux_command[12] = (unsigned char) '1';
    aux_command[13] = (unsigned char) '6';
    aux_command[14] = (unsigned char) ',';
    command_out_length = unsigned_to_decimal_string  (
                instruments[instrument_id].acquisition_interval, sample_interval_string, 1, 10);
    command_out_length = command_out_length + unsigned_to_decimal_string  (
instruments[instrument_id].samples_per_acquisition, samples_per_acquisition_string, 1, 10);

    command_out_length = command_out_length + unsigned_to_decimal_string  (
                instruments[instrument_id].processing, processing_string, 1, 1);
    sprintf(buffer, "%s", aux_command);
    strcat(buffer, sample_interval_string);
    strcat(buffer, ";17,");
    strcat(buffer, samples_per_acquisition_string);
    strcat(buffer, ";18,");
    strcat(buffer, processing_string);
    strcat(buffer,")");
    command_out_length = command_out_length+24;
    command_out = buffer;
}

// Function for the Set Instrument Command.
void datalogger_set_instrument(int* num_args, datalogger_protocol_argument_t args[],
datalogger_instrument_t instruments[MAX_NUMBER_OF_INSTRUMENTS]){
    datalogger_instrument_t aux_instrument;
    int index_instrument;
    for (int i = 0; i< num_args; i++){
        switch (args[i].argument_type){
            case INSTRUMENT_ID:{
                index_instrument = args[i].argument_value.instrument_id;
                break;
            }
            case TYPE_INSTRUMENT:{
                aux_instrument.instrument_type = args[i].argument_value.instrument_type;
                break;
            }
            case INTERVAL:{
                aux_instrument.acquisition_interval = args[i].argument_value.sample_interval;
                break;
```

```
            }
            case SAMPLES_PER_INTERVAL:{
                aux_instrument.samples_per_acquisition =
                                                args[i].argument_value.samples_per_interval;
                break;
            }
            case PROCESSING:{
                aux_instrument.processing = args[i].argument_value.processing;
                break;
            }
            default:{
                printf("Undefined argument\n");
                break;
            }
        }

    }
    aux_instrument.enabled = 1;
    instruments[index_instrument] = aux_instrument;
}

void datalogger_get_date_time(void){
    unsigned char* aux_command;
    char* buffer;
    wiced_iso8601_time_t datalogger_datetime_iso;
    wiced_time_get_iso8601_time(&datalogger_datetime_iso);
    aux_command = (unsigned char *)malloc(40 * sizeof(unsigned char));
    aux_command[0] = (unsigned char)'1';
    aux_command[1] = (unsigned char)'-';
    aux_command[2] = (unsigned char)'2';
    aux_command[3] = (unsigned char)'(';
    aux_command[4] = (unsigned char)'3';
    aux_command[5] = (unsigned char)',';
    for (int i = 0; i < 4; i++){
        aux_command[i+6]=(unsigned char)datalogger_datetime_iso.year[i];
    }
    aux_command[10] = (unsigned char)';';
    aux_command[11] = (unsigned char)'4';
    aux_command[12] = (unsigned char)',';
    aux_command[13] = (unsigned char)datalogger_datetime_iso.month[0];
    aux_command[14] = (unsigned char)datalogger_datetime_iso.month[1];
    aux_command[15] = (unsigned char)';';
    aux_command[16] = (unsigned char)'5';
    aux_command[17] = (unsigned char)',';
    aux_command[18] = (unsigned char)datalogger_datetime_iso.day[0];
    aux_command[19] = (unsigned char)datalogger_datetime_iso.day[1];
    aux_command[20] = (unsigned char)';';
    aux_command[21] = (unsigned char)'6';
    aux_command[22] = (unsigned char)',';
    aux_command[23] = (unsigned char)datalogger_datetime_iso.hour[0];
    aux_command[24] = (unsigned char)datalogger_datetime_iso.hour[1];
    aux_command[25] = (unsigned char)';';
    aux_command[26] = (unsigned char)'7';
    aux_command[27] = (unsigned char)',';
    aux_command[28] = (unsigned char)datalogger_datetime_iso.minute[0];
    aux_command[29] = (unsigned char)datalogger_datetime_iso.minute[1];
    aux_command[30] = (unsigned char)';';
    aux_command[31] = (unsigned char)'8';
    aux_command[32] = (unsigned char)',';
    aux_command[33] = (unsigned char)datalogger_datetime_iso.second[0];
    aux_command[34] = (unsigned char)datalogger_datetime_iso.second[1];
    aux_command[35] = (unsigned char)')';
    command_out_length = 36;
    buffer=(char *)malloc(50 * sizeof(char));
    sprintf(buffer, "%s", aux_command);
    command_out = buffer;
    tcp_client_connected = WICED_TRUE;
}

void metocean_realtime_data(void){
    char* buffer;
    char aux_buffer[100];
    aux_buffer[0] = '1';
    aux_buffer[1] = '-';
```

```
    aux_buffer[2] = '3';
    aux_buffer[3] = '(';
     if (length_buffer_realtime_out != 0) {
        for (int i = 0; i< length_buffer_realtime_out; i++){
            aux_buffer[4 + i] = internal_buffer_realtime_out[i];
        }
        buffer=(char *)malloc(100 * sizeof(char));
        length_buffer_realtime_out = length_buffer_realtime_out +4;
        command_out_length=length_buffer_realtime_out;
        command_out = buffer;
    }
    else {
        // No real time data to send
        command_out = "1-3(0,0)";
        command_out_length = 8;
    }
     //If real time buffer is full, clean it
    datalogger_sensors_clean_realtime_buffer();
}

void metocean_download_data(uint16_t buffer_length, uint32_t memory_address){

    char* memory_buffer;
    char* aux_buffer;
    char* aux_command;
    char* buffer;
    aux_command = (char *)malloc(2*(buffer_length + 5) * sizeof(char));
    aux_buffer = (char *)malloc((buffer_length + 5) * sizeof(char));
    aux_buffer = "1-4(";

    memory_buffer=(char *)malloc(buffer_length * sizeof(char));
    buffer=(char *)malloc(buffer_length * sizeof(char));
    strcpy(aux_command, aux_buffer);
    datalogger_memory_read(memory_buffer, buffer_length, memory_address);
    sprintf(buffer, "%s", memory_buffer);
    strcat(aux_command, buffer);
    command_out = aux_command;
    command_out_length = buffer_length + 4;
}

void metocean_command_execute(datalogger_protocol_command_t* str_command,
datalogger_instrument_t instruments[MAX_NUMBER_OF_INSTRUMENTS]){
    switch (str_command->s_command){
    case INSTRUMENT:{
        if(str_command->p_command == 0){
            datalogger_set_instrument(str_command->number_of_args, str_command->command_args,
                                                                                    instruments);
        } else {
            if (str_command->command_args[0].argument_type == INSTRUMENT_ID) {
                datalogger_get_instrument(str_command-
>command_args[0].argument_value.instrument_id, instruments);

            }

        }
        break;
    } case MODE:{
        if (str_command->p_command == SET) {
            wiced_utc_time_ms_t datalogger_actual_utc_ms;
            wiced_time_get_utc_time_ms(&datalogger_actual_utc_ms);
            if (str_command->command_args[1].argument_value.datalogger_acquire_utc_datetime ==
                                                                                        500) {
                datalogger_programmed_acquire_mode_delay=0;
            } else {
                datalogger_programmed_acquire_mode_delay = (str_command->
command_args[1].argument_value.datalogger_acquire_utc_datetime) - datalogger_actual_utc_ms;
            }
            datalogger_sensors_clean_realtime_buffer();
            datalogger_set_write_data_to_memory((wiced_bool_t)str_command->
                        command_args[2].argument_value.acquisition_memory_write_flag);
            datalogger_actual_mode = str_command->
                                        command_args[0].argument_value.datalogger_mode;
        } else {
            datalogger_get_mode();
```

```
        }
        break;
    } case DATE_TIME: {
        if (str_command->p_command == SET) {
            //Set the date and time of the datalogger
            wiced_time_set_utc_time_ms(&str_command->
                                command_args[0].argument_value.datalogger_utc_datetime);
        } else {
            datalogger_get_date_time();
        }
        break;
    } case REALTIME_DATA: {
        metocean_realtime_data();
        break;
    } case MEMORY_DOWNLOAD: {
        char* buffer;
        metocean_download_data(str_command->
command_args[1].argument_value.memory_read_buffer_length_value, str_command->
                        command_args[0].argument_value.memory_read_address_value);
        break;
    } case MEMORY_STATE: {
        uint32_t bytes_in_memory;
        char* buffer;
        char* aux;
        char* aux_command;
        aux_command=(char *)malloc(100 * sizeof(char));
        buffer=(char *)malloc(50 * sizeof(char));
        aux = (char *)malloc(50 * sizeof(char));
        aux = "1-5(12,";
        bytes_in_memory = datalogger_get_data_bytes_saved();
        unsigned_to_decimal_string  ( bytes_in_memory, buffer,0,5);
        strcpy(aux_command, aux);
        strcat(aux_command, buffer);
        aux = ")";
        strcat(aux_command, aux);
        command_out = aux_command;
        command_out_length = strlen(command_out);
        break;
    } case MEMORY_ERASE:{
        datalogger_memory_init();
        break;
    }
    default:
        // Undefined command
        break;
    }
}
```

Archivo datalogger_MetOcean_TCP_Protocol.h

```
/*
 * datalogger_MetOcean_TCP_protocol.h
 *
 *  Created on: 10/11/2019
 *      Author: José Raúl Santana Jiménez
 */

#ifndef APPS_DATALOGGER_DATALOGGER_METOCEAN_TCP_PROTOCOL_H_
#define APPS_DATALOGGER_DATALOGGER_METOCEAN_TCP_PROTOCOL_H_
#include <string.h>
#include <stdlib.h>
#include "wiced.h"

#define MAX_NUMBER_OF_TCP_PROTOCOL_ARGUMENTS 5
#define MAX_COMMAND_RESPONSE_LENGTH 10

// Commands defined for the MetOcean TCP protocol
typedef enum
{
    SET,
    GET
} metocean_tcp_protocol_primary_commands_t;
```

```
typedef enum
{
    INSTRUMENT,
    MODE,
    DATE_TIME,
    REALTIME_DATA,
    MEMORY_DOWNLOAD,
    MEMORY_STATE,
    MEMORY_ERASE
}metocean_tcp_protocol_secondary_commands_t;

// Arguments defined for the MetOcean TCP protocol commands
typedef enum{

    INSTRUMENT_ID,
    TYPE_INSTRUMENT,
    TYPE_MODE,
    YEAR_t,
    MONTH_t,
    DAY_t,
    HOUR_T,
    MINUTE_t,
    SECOND_t,
    SECONDS_UTC_t,
    SECONDS_UTC_ACQUIRE_t,
    REALTIME_DATA_arg,
    MEMORY_STATE_Arg,
    MEMORY_WRITE_FLAG,
    MEMORY_READ_ADDRESS,
    MEMORY_READ_BUFFER_LENGTH,
    INTERVAL,
    SAMPLES_PER_INTERVAL,
    PROCESSING
}metocean_tcp_protocol_command_arguments_t;

// Values defined for TYPE_INSTRUMENT argument
typedef enum{
    NONE_It,
    SBE_37SM
}datalogger_type_instruments_values_t;

// Values defined for MODE  secundary command
typedef enum{
    CONFIGURING,
    ACQUIRING
} datalogger_mode_satates_t;

/**
 * MetOcean TCP protocol Result Type
 */
typedef enum
{
    PARSER_SUCESS,
    COMMAND_NOT_SUPPORTED
} metocean_tcp_protocol_result_t;

typedef enum
{
    NONE_p,
    AVERAGE,
    MEDIAN
}metocean_tcp_protocol_processing_values;

//This union defines all arguments that can be used in all commands
typedef union
{
    int instrument_id;
    int instrument_type;
    int datalogger_mode;
    wiced_utc_time_ms_t datalogger_utc_datetime;
    wiced_utc_time_ms_t datalogger_acquire_utc_datetime;
    wiced_bool_t acquisition_memory_write_flag;
```

```
    uint32_t memory_read_address_value;
    uint16_t memory_read_buffer_length_value;
    uint16_t sample_interval;
    uint8_t samples_per_interval;
    metocean_tcp_protocol_processing_values processing;

}datalogger_protocol_arg_t;

//this struct contains one pair type argument, argument value
typedef struct
{
    metocean_tcp_protocol_command_arguments_t argument_type;
    datalogger_protocol_arg_t argument_value;
}datalogger_protocol_argument_t;

//This is the struct obtained after parsing one MetOcean TCP protocol Word.
//After parsing, this struct will be passed to the execute command pipeline.
typedef struct
{
    metocean_tcp_protocol_primary_commands_t p_command;
    metocean_tcp_protocol_secondary_commands_t s_command;
    int number_of_args;
    datalogger_protocol_argument_t command_args[MAX_NUMBER_OF_TCP_PROTOCOL_ARGUMENTS];
    char* response[MAX_COMMAND_RESPONSE_LENGTH];

}datalogger_protocol_command_t;


datalogger_protocol_command_t datalogger_metocean_parser(char *command);

typedef enum
{
    DATALOGGER_STATE_SET,
    DATALOGGER_STATE_GET
} metocean_tcp_protocol_commands_t;

#endif /* APPS_DATALOGGER_DATALOGGER_METOCEAN_TCP_PROTOCOL_H_ */
```

Archivo datalogger_MetOcean_TCP_Protocol.c

```
/*
 * datalogger_MetOcean_TCP_protocol.c
 *
 *  Created on: 10/11/2019
 *      Author: José Raúl Santana Jjiménez
 */
#include "datalogger_MetOcean_TCP_protocol.h"
#include <string.h>
#include <stdio.h>
#include <inttypes.h>

datalogger_protocol_command_t datalogger_metocean_parser(char *command)
{
    char* numeric_argument_value;
    char aux_value[30];
    int aux_value_length = 0;
    numeric_argument_value=(char *)malloc(20 * sizeof(char));
    char copy_command[40];
    unsigned char actual_char;
    memset(copy_command, '\0', sizeof(copy_command));
    strcpy(copy_command, &command[0]);
    datalogger_protocol_command_t aux_struct_command;
    int aux_int = 0;
    int pointer = 0;
    int num_args = 0;
    int tup_component = 0;
    int arg_magnitude_order = 1;
    actual_char=(unsigned char)copy_command[pointer];
    // Get the primary command
    while (actual_char != '-') {
        aux_int = aux_int*arg_magnitude_order + ((int)(actual_char - '0'));
        arg_magnitude_order = arg_magnitude_order*10;
```

```
    pointer++;
    actual_char=(unsigned char)copy_command[pointer];
};
aux_struct_command.p_command = aux_int;
arg_magnitude_order = 1;
pointer++;
aux_int = 0;
actual_char=(unsigned char)copy_command[pointer];
// Get the secondary command
while (actual_char != '(') {
    aux_int = aux_int*arg_magnitude_order + ((int)(actual_char - '0'));
    arg_magnitude_order = arg_magnitude_order*10;
    pointer++;
    actual_char=(unsigned char)copy_command[pointer];
};
aux_struct_command.s_command = aux_int;
do {
    aux_int =0;
    arg_magnitude_order = 1;
    pointer++;
    actual_char=(unsigned char)copy_command[pointer];
    while (actual_char != ','){
        aux_int = aux_int*arg_magnitude_order + ((int)(actual_char - '0'));
        arg_magnitude_order = arg_magnitude_order*10;
        pointer++;
        actual_char=(unsigned char)copy_command[pointer];
    };
    pointer ++;
    actual_char=(unsigned char)copy_command[pointer];
    switch ((unsigned int)aux_int) {
    case INSTRUMENT_ID: {
        aux_struct_command.command_args[num_args].argument_type = INSTRUMENT_ID;
        arg_magnitude_order = 1;
        aux_int =0;
        while ((actual_char != ';') && (actual_char != ')')){
            aux_int = aux_int*arg_magnitude_order + ((int)(actual_char - '0'));
            arg_magnitude_order = arg_magnitude_order*10;
            pointer++;
            actual_char=(unsigned char)copy_command[pointer];
        }
        aux_struct_command.command_args[num_args].argument_value.instrument_id = aux_int;
        break;
    }
    case TYPE_INSTRUMENT:{
        arg_magnitude_order = 1;
        aux_int =0;
        while ((actual_char != ';') && (actual_char != ')')){
            aux_int = aux_int*arg_magnitude_order + ((int)(actual_char - '0'));
            arg_magnitude_order = arg_magnitude_order*10;
            pointer++;
            actual_char=(unsigned char)copy_command[pointer];
        }
        aux_struct_command.command_args[num_args].argument_type = TYPE_INSTRUMENT;
        aux_struct_command.command_args[num_args].argument_value.instrument_type = aux_int;
        break;
    }
    case TYPE_MODE:{
        // Only one char for defining the TYPE MODE value
        aux_struct_command.command_args[num_args].argument_type = TYPE_MODE;
        actual_char=(unsigned char)copy_command[pointer];
        aux_struct_command.command_args[num_args].argument_value.datalogger_mode =
                                                    (int)(actual_char - '0');
        pointer++;
        actual_char=(unsigned char)copy_command[pointer];
        break;
    } case SECONDS_UTC_t:
      case SECONDS_UTC_ACQUIRE_t:
        printf("OBTENIENDO LOS SEGUNDOS EN UTC\n");
        char* buffer;
        char utc_seconds_aux[30];
        wiced_utc_time_ms_t datalogger_datetime;
        actual_char=(unsigned char)copy_command[pointer];
        int seconds_length = 0;
        while ((unsigned char )actual_char != (unsigned char)')' && (unsigned char
```

```
                                            )actual_char != (unsigned char)';'){
                    utc_seconds_aux[seconds_length] = actual_char;
                    seconds_length++;
                    pointer++;
                    actual_char=(unsigned char)copy_command[pointer];
                }
                utc_seconds_aux[seconds_length] = '5';
                utc_seconds_aux[seconds_length+1] = '0';
                utc_seconds_aux[seconds_length+2] = '0';
                buffer=(char *)malloc((seconds_length+3) * sizeof(char));
                sprintf(buffer, "%s", utc_seconds_aux);
                datalogger_datetime = (wiced_utc_time_ms_t)atof(buffer);
                if (aux_int == SECONDS_UTC_t) {
                    aux_struct_command.command_args[num_args].argument_type= SECONDS_UTC_t;
                    aux_struct_command.command_args[num_args].argument_value.datalogger_utc_datetime
                                                                        = datalogger_datetime;
                } else if(aux_int == SECONDS_UTC_ACQUIRE_t) {
                    aux_struct_command.command_args[num_args].argument_type= SECONDS_UTC_ACQUIRE_t;

                    aux_struct_command.command_args[num_args].argument_value.datalogger_acquire_utc_
                    datetime = datalogger_datetime;
                }
                break;
              case MEMORY_WRITE_FLAG: {
                  aux_struct_command.command_args[num_args].argument_type = MEMORY_WRITE_FLAG;

aux_struct_command.command_args[num_args].argument_value.acquisition_memory_write_flag =
(wiced_bool_t)copy_command[pointer];
                actual_char=(unsigned char)copy_command[pointer];
                pointer++;
                actual_char=(unsigned char)copy_command[pointer];
                break;
            } case MEMORY_READ_ADDRESS:
              case MEMORY_READ_BUFFER_LENGTH:
              case INTERVAL:
              case SAMPLES_PER_INTERVAL:
                actual_char=(unsigned char)copy_command[pointer];
                aux_value_length = 0;
                while ((unsigned char )actual_char != (unsigned char)')' && (unsigned char
                                                    )actual_char != (unsigned char)';'){
                    aux_value[aux_value_length] = actual_char;
                    aux_value_length++;
                    pointer++;
                    actual_char=(unsigned char)copy_command[pointer];
                }
                sprintf(numeric_argument_value, "%s", aux_value);
                if(aux_int == MEMORY_READ_ADDRESS) {
                    aux_struct_command.command_args[num_args].argument_type = MEMORY_READ_ADDRESS;

        aux_struct_command.command_args[num_args].argument_value.memory_read_address_value
                                                    =(uint32_t)atof(numeric_argument_value);

                } else if (aux_int == MEMORY_READ_BUFFER_LENGTH) {
                    aux_struct_command.command_args[num_args].argument_type =
                                                                MEMORY_READ_BUFFER_LENGTH;

                  aux_struct_command.command_args[num_args].argument_value.memory_read_buffer_leng
                                        th_value =(uint16_t)atof(numeric_argument_value);
                } else if(aux_int == INTERVAL){
                    aux_struct_command.command_args[num_args].argument_type = INTERVAL;
                    aux_struct_command.command_args[num_args].argument_value.sample_interval =
                                                     (uint16_t)atof(numeric_argument_value);
                } else if (aux_int == SAMPLES_PER_INTERVAL){
                    aux_struct_command.command_args[num_args].argument_type =
                                                                        SAMPLES_PER_INTERVAL;
                    aux_struct_command.command_args[num_args].argument_value.samples_per_interval
                                                    = (uint8_t)atof(numeric_argument_value);
                }
                break;
              case PROCESSING:{
                  // Only one char for defining the PROCESSING value
                  aux_struct_command.command_args[num_args].argument_type = PROCESSING;
                  actual_char=(unsigned char)copy_command[pointer];
                  aux_struct_command.command_args[num_args].argument_value.processing =
```

```
                                                        (int)(actual_char - '0');
                pointer++;
                actual_char=(unsigned char)copy_command[pointer];
                break;
            }
          default:
            // Undefined value
                break;
        };
        num_args++;
    }while (actual_char != ')') ;
    aux_struct_command.number_of_args = num_args;
    return aux_struct_command;
}
```

Archivo datalogger_Acquire.h

```
/*
 * datalogger_Acquire.h
 *
 *  Created on: 18/11/2019
 *      Author: José Raúl Santana Jiménez
 */

#ifndef APPS_DATALOGGER_DATALOGGER_ACQUIRE_H_
#define APPS_DATALOGGER_DATALOGGER_ACQUIRE_H_

#define THREAD_PRIORITY_INSTRUMENT           (6)
#define THREAD_STACK_SIZE_INSTRUMENT       (1024)
#include "wiced.h"
#include "datalogger_general.h"

void datalogger_instrument_acquire(int Instrument_Id);

void acquisition_mode_init(void);
void acquisition_mode_start(datalogger_instrument_t* avaliable_instruments);

void aquisition_instrument(uint32_t arg);
void set_stop_instrument_acquiring(int value);

#endif /* APPS_DATALOGGER_DATALOGGER_ACQUIRE_H_ */
```

Archivo datalogger_Acquire.c

```
/*
 * datalogger_Aquire.c
 *
 *  Created on: 18/11/2019
 *      Author: José Raúl Santana Jiménez
 */
# include "datalogger_Acquire.h"
# include "datalogger_Sensors.h"
# include "wiced.h"

// Threads and semaphores for handling the instruments acquisition
static wiced_semaphore_t semaphoreHandle_Instrument[MAX_NUMBER_OF_INSTRUMENTS];
static wiced_thread_t ThreadHandle_Instrument[MAX_NUMBER_OF_INSTRUMENTS];
// Variables for instruments threads arguments
static int argument_instrument[MAX_NUMBER_OF_INSTRUMENTS];

static datalogger_instrument_t
acquisition_datalogger_available_instruments[MAX_NUMBER_OF_INSTRUMENTS];

// Flag used to start and stop instruments acquisition
static int stop_instrument_aquiring;

void datalogger_instrument_acquire(int Instrument_Id){
    switch (acquisition_datalogger_available_instruments[Instrument_Id].instrument_type){
    case NONE_It:{
```

```
        break;
    }
    case SBE_37SM: {
        datalogger_SBE37_acquire(&acquisition_datalogger_available_instruments[Instrument_Id],
Instrument_Id);
        break;
    }
    }
}

void acquisition_instrument(uint32_t arg) {
    wiced_rtos_delay_milliseconds(500);
    while(1) {

        if (stop_instrument_aquiring == 1) {
            wiced_rtos_get_semaphore(&(semaphoreHandle_Instrument[arg]), WICED_WAIT_FOREVER);
        }

wiced_rtos_delay_milliseconds(acquisition_datalogger_available_instruments[arg].acquisition_inte
                                                                                    rval);

        datalogger_instrument_acquire(arg);
        wiced_rtos_delay_milliseconds(5000);
        SBE_print_buffer(arg);
        wiced_rtos_delay_milliseconds(1000);
    }
}


void set_stop_instrument_acquiring(int value){
    stop_instrument_aquiring = value;
}

void acquisition_mode_init(){
/* Set up the instruments semaphore handlers and Initialize and start instruments threads */
    stop_instrument_aquiring = 1;
    for (int i = 0; i< MAX_NUMBER_OF_INSTRUMENTS; i++){
        argument_instrument[i]=i;
        wiced_rtos_init_semaphore(&(semaphoreHandle_Instrument[i]));
        wiced_rtos_create_thread(&(ThreadHandle_Instrument[i]), THREAD_PRIORITY_INSTRUMENT,
NULL, acquisition_instrument, THREAD_STACK_SIZE_INSTRUMENT, argument_instrument[i]);
    }
}

// Start running all threads for acquiring enabled instruments data and store in memory
void acquisition_mode_start(datalogger_instrument_t* avaliable_instruments){
    for (int i = 0; i < MAX_NUMBER_OF_INSTRUMENTS; i++){
        acquisition_datalogger_available_instruments[i] = avaliable_instruments[i];
        if (avaliable_instruments[i].enabled == 1){
                wiced_rtos_set_semaphore(&(semaphoreHandle_Instrument[i]));
        }
    }
}
```

Archivo datalogger_sensors.h

```
/*
 * datalogger_Sensors.h
 *
 *  Created on: 19/11/2019
 *      Author: José Raúl Santana Jiménez
 */

#ifndef APPS_DATALOGGER_DATALOGGER_SENSORS_H_
#define APPS_DATALOGGER_DATALOGGER_SENSORS_H_
#include "datalogger_general.h"
#include "wiced.h"

#define BUFFER_REALTIME_OUT_MAX_LENGTH 100
#define MAX_RECEIVED_DATA_LENGTH_SBE37 100
#define MAX_NUMBER_OF_SAMPLES_PER_ACQUISITION 50
```

```
// Defines for the SBE 37 available parameters*******
#define TEMPERATURE 0
#define CONDUCTIVITY 1
#define PRESSURE 2
#define OXYGEN 3
#define SALINITY 4
//************************************************


#define MEDIAN_t 2
#define AVERAGE_t 1


extern char* buffer_realtime_out;
extern char internal_buffer_realtime_out[200];
extern int length_buffer_realtime_out;

void SBE_print_buffer(uint32 arg);

float datalogger_median(float *vector_values, int vector_length);

float datalogger_average(float *vector_values, int vector_length);

wiced_bool_t datalogger_get_write_data_to_memory(void);

void datalogger_set_write_data_to_memory(wiced_bool_t write_data);

void datalogger_check_realtime_buffer_max_length(void);

void datalogger_sensors_clean_realtime_buffer(void);

void datalogger_sensors_init(void);

void datalogger_SBE37_acquire(datalogger_instrument_t* instrument, int instrument_id);


#endif /* APPS_DATALOGGER_DATALOGGER_SENSORS_H_ */
```

Archivo datalogger_sensors.c

```
/*
 * datalogger_Sensors.c
 *
 *  Created on: 19/11/2019
 *      Author: José Raúl Santana Jiménez
 */
#include "datalogger_Sensors.h"
#include "datalogger_Serial_Port.h"
#include "datalogger_memory.h"

// Mutex for locking the memory access between the sensor threads
static wiced_mutex_t memory_access_mutex;


char* buffer_realtime_out = "";
char internal_buffer_realtime_out[200];
int length_buffer_realtime_out = 0;
wiced_bool_t write_data_to_memory = WICED_TRUE;


float datalogger_median(float *vector_values, int vector_length){
  float median = 0.0;
  float aux_value = 0.0;
  int minimun_value_pointer = 0;
  int median_pointer = 0;
  // sort vector_values from min to max values
  for (int i = 0; i< vector_length; i++){
      minimun_value_pointer = i;
      for (int j = minimun_value_pointer+1; j<vector_length; j++) {
          if (vector_values[j] < vector_values[i]){
              minimun_value_pointer = j;
          }
```

```
      }
      aux_value = vector_values[i];
      vector_values[i] = vector_values[minimun_value_pointer];
      vector_values[minimun_value_pointer] = aux_value;
  }
  median_pointer = vector_length / 2;
  median = vector_values[median_pointer];
  return median;
}

float datalogger_average(float *vector_values, int vector_length){
    float average = 0.0;
    for (int i = 0; i< vector_length; i++) {
        average = average + vector_values[i];
    }
    average = average /vector_length;
    return average;
}


wiced_bool_t datalogger_get_write_data_to_memory(void){
    return write_data_to_memory;
}

void datalogger_set_write_data_to_memory(wiced_bool_t write_data){
    write_data_to_memory = write_data;
}

void datalogger_sensors_init(void) {
    buffer_realtime_out=(char *)malloc(100 * sizeof(char));
    buffer_realtime_out = "";
     wiced_rtos_init_mutex(&memory_access_mutex);

}

void datalogger_sensors_clean_realtime_buffer(void){
    buffer_realtime_out = "";
    for (int i = 0; i < length_buffer_realtime_out; i++){
        internal_buffer_realtime_out[length_buffer_realtime_out] = NULL;
    }
    length_buffer_realtime_out = 0;
}

void datalogger_check_realtime_buffer_max_length(void){
    if (length_buffer_realtime_out >= BUFFER_REALTIME_OUT_MAX_LENGTH){
        datalogger_sensors_clean_realtime_buffer();
    }
}

void SBE_print_buffer(uint32 arg){
    print_buffer(arg);
}

void datalogger_obtain_datetime_string(char *datetime_buffer){
    wiced_iso8601_time_t datalogger_datetime_iso;
    char *year = (char *)malloc(5 * sizeof(char));
    char *month = (char *)malloc(3 * sizeof(char));
    char *day = (char *)malloc(3 * sizeof(char));
    char *hour = (char *)malloc(3 * sizeof(char));
    char *minutes = (char *)malloc(3 * sizeof(char));
    char *seconds = (char *)malloc(3 * sizeof(char));
    wiced_time_get_iso8601_time(&datalogger_datetime_iso);
    sprintf(year, "%s", datalogger_datetime_iso.year);
    sprintf(month, "%s", datalogger_datetime_iso.month);
    sprintf(day, "%s", datalogger_datetime_iso.day);
    sprintf(hour, "%s", datalogger_datetime_iso.hour);
    sprintf(minutes, "%s", datalogger_datetime_iso.minute);
    sprintf(seconds, "%s", datalogger_datetime_iso.second);
    strcpy(datetime_buffer, month);
    strcat(datetime_buffer, "/");
    strcat(datetime_buffer, day);
    strcat(datetime_buffer, "/");
    strcat(datetime_buffer, year);
    strcat(datetime_buffer, " ");
```

```
    strcat(datetime_buffer, hour);
    strcat(datetime_buffer, ":");
    strcat(datetime_buffer, minutes);
    strcat(datetime_buffer, ":");
    strcat(datetime_buffer, seconds);
}

void datalogger_SBE37_acquire(datalogger_instrument_t* instrument, int instrument_id){
    char SBE37_command[3];
    char *string_data_result = (char *)malloc(100 * sizeof(char));
    char *instrument_string = (char *)malloc(20 * sizeof(char));
    char *instrument_id_char = (char *)malloc(sizeof(char));
    char * parameter_string = (char *)malloc(10 * sizeof(char));
    int number_of_bytes_received = 0;
    float temperature[MAX_NUMBER_OF_SAMPLES_PER_ACQUISITION] = {0.0};
    float conductivity[MAX_NUMBER_OF_SAMPLES_PER_ACQUISITION] = {0.0};
    float pressure[MAX_NUMBER_OF_SAMPLES_PER_ACQUISITION] = {0.0};
    float oxygen[MAX_NUMBER_OF_SAMPLES_PER_ACQUISITION] = {0.0};
    float salinity[MAX_NUMBER_OF_SAMPLES_PER_ACQUISITION] = {0.0};
    float base = 10.0;
    float temperature_final = 0.0;
    float conductivity_final = 0.0;
    float pressure_final = 0.0;
    float oxygen_final = 0.0;
    float salinity_final = 0.0;
    int sign = 1;
    int frames_received = 0;
    int data_parameter = TEMPERATURE;
    float aux_data_int_part = 0.0; //integer part of one value
    float aux_data_frac_part = 0.0; //fractional par of one value
    float aux_data = 0.0; // auxiliar value used during conversion from string to float
    wiced_bool_t frame_end = WICED_FALSE;
    wiced_bool_t fractional = WICED_FALSE;

    char* datalogger_datetime = (char *)malloc(20* sizeof(char));
    int k = -1;
    SBE37_command[0] = 't';
    SBE37_command[1] = 's';
    SBE37_command[2] = '\r';
    int memory_write_buffer_length = 0;
    uint8 input_data[MAX_RECEIVED_DATA_LENGTH_SBE37];
    datalogger_check_realtime_buffer_max_length();
    for (int l = 0; l< instrument->samples_per_acquisition; l++) {
        serial_send_string(SBE37_command, 3, instrument_id, 9600);
        wiced_rtos_delay_milliseconds(4000); //time to allow the SBE37 to send the acquired data
        if (serial_new_byte_received(instrument_id)){

                number_of_bytes_received = get_serialport_buffer_length(instrument_id);
                if (number_of_bytes_received <= MAX_RECEIVED_DATA_LENGTH_SBE37){ //search the
                                                                                 '<' char
                    k = -1;
                    do {
                        k++;
                        input_data[k] = serial_port_buffer_get_byte(instrument_id, k);
                        if (input_data[k] == '<') {
                            frame_end = WICED_TRUE;
                        }
                    }while(frame_end == WICED_FALSE);
                }
                if (frame_end == WICED_TRUE) {
                    k--;
                    frames_received++;
                    for (int i = 0; i<k; i++){
                        if ((unsigned char)input_data[i] != ','){
                            if((unsigned char)input_data[i] != ' '){
                                if((unsigned char)input_data[i] == '-'){
                                    sign = -1;
                                }
                                if ((unsigned char)input_data[i] == '.'){
                                    fractional = WICED_TRUE;
                                    base = 0.1;
                                }
                                if (fractional) {// Obtain the fractional part of the data
                                    aux_data_frac_part = aux_data_frac_part +
```

```
                                                ((float)(input_data[i] - '0')*base);
                                    base = base*0.1;
                                } else { //Obtain the integer part of the data
                                    aux_data_int_part = aux_data_int_part*base +
                                                                ((float)(input_data[i] - '0'));
                                    base = base*10;
                                }
                            }
                        } else {
                            fractional = WICED_FALSE;
                            base = 10.0;
                            aux_data = ((float)sign*aux_data_int_part)+aux_data_frac_part;
                            switch (data_parameter) {
                            case TEMPERATURE:{
                                temperature[l] = aux_data;
                                data_parameter = CONDUCTIVITY;
                                break;
                            }
                            case CONDUCTIVITY:{
                                conductivity[l] = aux_data;
                                data_parameter = PRESSURE;
                                break;
                            }
                            case PRESSURE:{
                                pressure[l] = aux_data;
                                data_parameter = OXYGEN;
                                break;
                            }
                            case OXYGEN:{
                                oxygen[l] = aux_data;
                                data_parameter = SALINITY;
                                break;
                            }
                            case SALINITY:{
                                salinity[l] = aux_data;
                                break;
                            }
                            default:
                                break;
                            }
                            sign = 1;
                            aux_data = 0.0;
                            aux_data_frac_part = 0.0;
                            aux_data_int_part = 0.0;
                        }
                    }
                    datalogger_clear_serial_port_buffer(instrument_id);
                }
            }
        }
        switch (instrument->processing) {
        case MEDIAN_t:{
            temperature_final = datalogger_median(&temperature[0], frames_received);
            conductivity_final = datalogger_median(&conductivity[0], frames_received);
            pressure_final = datalogger_median(&pressure[0], frames_received);
            oxygen_final = datalogger_median(&oxygen[0], frames_received);
            salinity_final = datalogger_median(&salinity[0], frames_received);
            break;
        }
        case AVERAGE_t:{
            temperature_final = datalogger_average(&temperature[0], frames_received);
            conductivity_final = datalogger_average(&conductivity[0], frames_received);
            pressure_final = datalogger_average(&pressure[0], frames_received);
            oxygen_final = datalogger_average(&oxygen[0], frames_received);
            salinity_final = datalogger_average(&salinity[0], frames_received);
            break;
        }
        default:{
            temperature_final = temperature[0];
            conductivity_final = conductivity[0];
            pressure_final = pressure[0];
            oxygen_final = oxygen[0];
            salinity_final = salinity[0];
            break;
```

```
    }
    }

    instrument_string  = "Instrument ";
    unsigned_to_decimal_string ((uint32_t)instrument_id, instrument_id_char, 1, 1);
    strcpy(string_data_result, instrument_string);
    strcat(string_data_result, instrument_id_char);
    strcat(string_data_result, ": ");
    memory_write_buffer_length = 14 + float_to_string(parameter_string, 10, temperature_final
                                                                                    ,4);
    strcat(string_data_result, parameter_string);
    parameter_string = ", ";
    memory_write_buffer_length = memory_write_buffer_length + 2;
    strcat(string_data_result, parameter_string);
    parameter_string = "";
    memory_write_buffer_length = memory_write_buffer_length + float_to_string(parameter_string,
                                                        10, conductivity_final ,5);
    strcat(string_data_result, parameter_string);
    parameter_string = ", ";
    memory_write_buffer_length = memory_write_buffer_length + 2;
    strcat(string_data_result, parameter_string);
    parameter_string = "";
    memory_write_buffer_length = memory_write_buffer_length + float_to_string(parameter_string,
                                                            10, pressure_final ,3);
    strcat(string_data_result, parameter_string);
    parameter_string = ", ";
    memory_write_buffer_length = memory_write_buffer_length + 2;
    strcat(string_data_result, parameter_string);
    parameter_string = "";
    memory_write_buffer_length = memory_write_buffer_length + float_to_string(parameter_string,
                                                            10, oxygen_final ,3);
    strcat(string_data_result, parameter_string);
    parameter_string = ", ";
    memory_write_buffer_length = memory_write_buffer_length + 2;
    strcat(string_data_result, parameter_string);
    parameter_string = "";
    memory_write_buffer_length = memory_write_buffer_length + float_to_string(parameter_string,
                                                             10, salinity_final ,3);
    strcat(string_data_result, parameter_string);
    parameter_string = ", ";
    memory_write_buffer_length = memory_write_buffer_length + 2;
    strcat(string_data_result, parameter_string);
    parameter_string = "";
    datalogger_obtain_datetime_string(datalogger_datetime);
    strcat(string_data_result,datalogger_datetime);
    memory_write_buffer_length = memory_write_buffer_length+19;
    strcat(string_data_result, "\n");
    memory_write_buffer_length++;
    strcat(internal_buffer_realtime_out, string_data_result);
    length_buffer_realtime_out = length_buffer_realtime_out + memory_write_buffer_length;
    if (write_data_to_memory == '1'){
        wiced_rtos_lock_mutex(&memory_access_mutex); // Block the access to the memory
        datalogger_memory_write(string_data_result, memory_write_buffer_length);
        wiced_rtos_unlock_mutex(&memory_access_mutex); // Un block the access to the memory
    }
}
```

Archivo datalogger_Serial_Port.h

```
/*
 * datalogger_Serial_Port.h
 *
 *  Created on: 19/11/2019
 *      Author: José Raúl Santana Jiménez
 */

#ifndef APPS_DATALOGGER_DATALOGGER_SERIAL_PORT_H_
#define APPS_DATALOGGER_DATALOGGER_SERIAL_PORT_H_

#include "wiced.h"
#include "platform_peripheral.h"
#include "platform_init.h"
```

```
// Definitions for datalogger serial port management
typedef enum{
    SERIAL1,
    SERIAL2,
    SERIAL3,
    SERIAL4,
    SERIAL5,
    SERIAL6,
    SERIAL7,
    SERIAL8
}datalogger_asinchronous_serial_ports_t;

typedef struct {
    datalogger_asinchronous_serial_ports_t port;
    wiced_gpio_t Tx;
    wiced_gpio_t Rx;
}datalogger_asinchronous_serial_port_t;

int get_serialport_buffer_length(int instrument_id);

wiced_bool_t serial_new_byte_received(uint32_t arg);

void print_buffer(uint32_t arg);

void serial_start_isr(uint32_t arg);

void datalogger_receive(uint32_t arg);

void datalogger_send(wiced_bool_t byte[], int serial_port_id);

void datalogger_char_to_byte(char ch, wiced_bool_t byte[]);

void serial_send_string(char* string_to_send, int number_of_chars, int serial_port_id, int
                                                                                    baud_rate);

void datalogger_serial_ports_init(void);

#endif /* APPS_DATALOGGER_DATALOGGER_SERIAL_PORT_H_ */
```

Archivo datalogger_Serial_Port.c

```
/*
 * datalogger_Serial_Port.c
 *
 *  Created on: 19/11/2019
 *      Author: José Raúl Santana Jiménez
 */
#include "datalogger_general.h"
#include "datalogger_Serial_port.h"

#define THREAD_PRIORITY_SERIAL_PORT           (5)
#define THREAD_STACK_SIZE_SERIAL_PORT       (1024)
#define MAX_SERIAL_INPUT_BUFFER_LENGTH      (1024)

static datalogger_asinchronous_serial_port_t
available_instruments_serial_ports[MAX_NUMBER_OF_INSTRUMENTS];

// Threads and semaphores for handling the serial ports
static wiced_semaphore_t semaphoreHandle_Serial_Port[MAX_NUMBER_OF_INSTRUMENTS];
static wiced_thread_t ThreadHandle_Serial_Port[MAX_NUMBER_OF_INSTRUMENTS];
wiced_bool_t
serial_buffer_input[MAX_NUMBER_OF_INSTRUMENTS][MAX_SERIAL_INPUT_BUFFER_LENGTH]={WICED_TRUE};
int serial_buffer_input_pointer[MAX_NUMBER_OF_INSTRUMENTS]={0};
wiced_bool_t serial_port_new_byte_received[MAX_NUMBER_OF_INSTRUMENTS];


wiced_bool_t serial_new_byte_received(uint32_t arg){
    return serial_port_new_byte_received[arg];
}
```

```
uint8 serial_port_buffer_get_byte(int port_id, int index){
    int bit_LSB = (index * 8) + 7;
    int bit_MSB = (index * 8);
    int actual_bit;
    uint8 aux = 0;
    int base = 1;
    for(int i = bit_LSB; i<= bit_MSB; i--){
        actual_bit = serial_buffer_input[port_id][i];
        aux = aux + (actual_bit*base);
        base = base *2;
    }
    return aux;
}

int get_serialport_buffer_length(int instrument_id){
    return serial_buffer_input_pointer[instrument_id];
}

void datalogger_clear_serial_port_buffer(instrument_id){
    serial_buffer_input_pointer[instrument_id] = 0;
}

void serial_start_isr(uint32_t arg)
{
    wiced_rtos_set_semaphore(&semaphoreHandle_Serial_Port[arg]); /* Set the semaphore */
}

void print_buffer(uint32_t arg){
    int j = 0;
    for (int i = 0; i < serial_buffer_input_pointer[arg]; i++){
        if (j == 7) {
            j = 0;
            printf("\n");
        } else {
            j++;
        }
    }
    serial_buffer_input_pointer[arg] = 0;
}


void datalogger_receive(uint32_t arg)
{
    wiced_bool_t byte[8];
    wiced_bool_t estado = WICED_FALSE;
    int index = (int)arg;
    while(1)
    {
        wiced_rtos_get_semaphore(&semaphoreHandle_Serial_Port[index], WICED_WAIT_FOREVER); //
                                                                    whait for the start bit
        wiced_gpio_input_irq_disable(available_instruments_serial_ports[index].Rx); // Unable
                                                                                    the IRS

        for (int i = 0; i <= 7; i++){
            if (estado == WICED_FALSE) {
                estado = WICED_TRUE;
            } else {
                estado = WICED_FALSE;
            }
            byte[i] =  wiced_gpio_input_get(available_instruments_serial_ports[index].Rx);
            serial_buffer_input[arg][serial_buffer_input_pointer[arg]] = byte[i];
            serial_buffer_input_pointer[arg] =serial_buffer_input_pointer[arg]+1;
            serial_port_new_byte_received[index]=WICED_TRUE;
            wiced_rtos_delay_microseconds( 97 );
        }
        wiced_gpio_input_irq_enable(available_instruments_serial_ports[arg].Rx,
                                    IRQ_TRIGGER_FALLING_EDGE, serial_start_isr, arg);
    }
}

void datalogger_send(wiced_bool_t byte[], int serial_port_id) {
    int aux = 1;
    wiced_gpio_output_low(available_instruments_serial_ports[serial_port_id].Tx); // start bit
    wiced_rtos_delay_microseconds( 100 );
    for (int i = 0; i <= 7; i++) {
```

```
        if (byte[i] == WICED_FALSE) {
             wiced_gpio_output_low(available_instruments_serial_ports[serial_port_id].Tx);
        } else {
            wiced_gpio_output_high(available_instruments_serial_ports[serial_port_id].Tx);
        }
        wiced_rtos_delay_microseconds( 100 );
    }
    wiced_gpio_output_high(available_instruments_serial_ports[serial_port_id].Tx); // bit de
                                                                                      stop
    wiced_rtos_delay_microseconds( 100 );
}


void datalogger_char_to_byte(char ch, wiced_bool_t byte[]) {
    int integer_ch;
    int j = 0;
    integer_ch = (unsigned int)ch;
    while(integer_ch >= 2) {
        byte[j] = integer_ch % 2;
        integer_ch = integer_ch/2;
        j=j+1;
    }
    byte[j] = integer_ch;
    j = j+1;
    while (j <= 7){
        byte[j] = 0;
        j = j+1;
    }
}

void serial_send_string(char* string_to_send, int number_of_chars, int serial_port_id, int
                                                                                    baud_rate){
    wiced_bool_t byte[8];
    for (int i = 0; i< number_of_chars; i++){
        datalogger_char_to_byte(string_to_send[i], byte);
        datalogger_send(byte, serial_port_id);
    }
}

void datalogger_serial_ports_init(void){
    available_instruments_serial_ports[0].Tx = WICED_GPIO_61;
    available_instruments_serial_ports[1].Tx = WICED_GPIO_63;
    available_instruments_serial_ports[2].Tx = WICED_GPIO_66;
    available_instruments_serial_ports[3].Tx = WICED_GPIO_69;
    available_instruments_serial_ports[4].Tx = WICED_GPIO_71;
    available_instruments_serial_ports[5].Tx = WICED_GPIO_98;
    available_instruments_serial_ports[6].Tx = WICED_GPIO_93;
    available_instruments_serial_ports[7].Tx = WICED_GPIO_39;
    available_instruments_serial_ports[0].Rx = WICED_GPIO_62;
    available_instruments_serial_ports[1].Rx = WICED_GPIO_65;
    available_instruments_serial_ports[2].Rx = WICED_GPIO_67;
    available_instruments_serial_ports[3].Rx = WICED_GPIO_70;
    available_instruments_serial_ports[4].Rx = WICED_GPIO_72;
    available_instruments_serial_ports[5].Rx = WICED_GPIO_99;
    available_instruments_serial_ports[6].Rx = WICED_GPIO_94;
    available_instruments_serial_ports[7].Rx = WICED_GPIO_40;

    for (int i = 0; i< MAX_NUMBER_OF_INSTRUMENTS; i++){
        available_instruments_serial_ports[i].port = i;
        wiced_gpio_init(available_instruments_serial_ports[i].Tx, OUTPUT_PUSH_PULL);
        wiced_gpio_init(available_instruments_serial_ports[i].Rx, INPUT_HIGH_IMPEDANCE);
        serial_buffer_input_pointer[i]=0;
        serial_buffer_input[i][0]=WICED_FALSE;

        wiced_gpio_output_high(available_instruments_serial_ports[i].Tx);

        wiced_rtos_init_semaphore(&(semaphoreHandle_Serial_Port[i]));
        wiced_rtos_create_thread(&(ThreadHandle_Serial_Port[i]), THREAD_PRIORITY_SERIAL_PORT,
                                 NULL, datalogger_receive, THREAD_STACK_SIZE_SERIAL_PORT, i);

        wiced_gpio_input_irq_enable(available_instruments_serial_ports[i].Rx,
IRQ_TRIGGER_FALLING_EDGE, serial_start_isr, i);

    }
```

}

A.2 Software del cliente

Archivo Main.py

```
#coding=utf-8
#Main.py
# Created on: 20/10/2019
#     Author: Jose Raul Santana Jimenez
#!/usr/bin/env Python

import socket
import optparse
import datetime
import time
import sys
import threading
import PIL.Image
import PIL.ImageTk
import tkFont
import MetOceanParser
from tkinter import ttk
import tkMessageBox, tkFileDialog
from MetOceanParser import *

# Global variables
sck = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
MetOceanCommand = MetOcean_object_command(-1,-1,{})
# Max number of instruments that can be configured in the GUI
NUMBER_OF_INSTRUMENTS = 8
# Instruments that are available per row
NUMBER_OF_INSTRUMENTS_PER_ROW = 4

BUFFER_SIZE = 1024
COMMAND = ""

# IP details for the WICED TCP server
DEFAULT_IP   = '192.168.10.1'    # IP address of the WICED TCP async server
DEFAULT_PORT = 7777              # Port of the WICED TCP server


# Labels for error handling while connecting to the datalogger##
SUCCESS = 0
NETWORK_MISTMATCH = 1
UNABLE_TO_CONNECT_TO_SERVER = 2
UNABLE_TO_GET_HOST_IP = 3


DEFAULT_KEEP_ALIVE = 0           # Keep the TCP connection alive (=1), or close the connection
(=0)

received_MetOcean_data = "-1" # Global variable for keeping the last received MetOcean data from
the datalogger
realtime_file_path = ""

# Arrays for managing the GUIs widgets ************************##
# list containing all the "Enabled" Checkbuttos of the GUI*****##
instruments_checkbuttons_list = []
# list of lists containing all widget names inside the instruments frames
widget_instruments_array=[]
widget_instruments_tuple_array=[]

memory_used = 0

realtime_data_flag = 0
```

```
# Function for getting the memory state from the datalogger
def get_memory_state():
    send_command = GET + "-" + MEMORY_STATE + "(0,0)"
    TCP_Tx_Rx(send_command)
    MetOceanCommand.parseMetOceanCommand(received_MetOcean_data)
    execute_command()


# Function to start the acquisition in real time mode
def realtime_data(time_to_start_realtime):
    global DEFAULT_KEEP_ALIVE
    global realtime_data_flag
    global realtime_file_path
    global MetOceanCommand
    global realtime_file
    if (time_to_start_realtime != 0):
        now = datetime.datetime.utcnow()
        uct_seconds = (now - datetime.datetime(1970, 1, 1)).total_seconds()
        time_to_start_realtime = float(time_to_start_realtime) - float(uct_seconds)
    time.sleep(float(time_to_start_realtime))
    realtime_file = open(realtime_file_path, "w")
    realtime_data_flag = 1
    while (realtime_data_flag == 1):
        send_command = GET + "-" + REALTIME_DATA + "(9,0)"
        TCP_Tx_Rx(send_command)
        MetOceanCommand.parseMetOceanCommand(received_MetOcean_data)
        execute_command()

        time.sleep(2)
    realtime_file.close()

# Function to erase the datalogger memory
def erase_memory():
    if tkMessageBox.askyesno("Erase Memory", "Do you want to delete all stored data?"):
        send_command = SET + "-" + MEMORY_ERASE + "(0,0)"
        TCP_Tx_Rx(send_command)
        get_memory_state()

# Function to download the data stored in the datalogger's memory
def save_data_file():
    datafile = tkFileDialog.asksaveasfilename(title = "Select file to save data")
    if (datafile != ""):
        global delayed_data_file
        global memory_used
        get_memory_state()
        if (memory_used > 0):
            delayed_data_file = open(datafile, "w")
            first_address_to_read = 0
            if(int(memory_used) > 1023):
                last_address_to_read = 1023
                while (1):
                    send_command = GET + "-" + MEMORY_DOWNLOAD + "(" + MEMORY_READ_ADDRESS + ","
+ str(first_address_to_read) + ";" + MEMORY_READ_BUFFER_LENGTH + "," + str(last_address_to_read
                                                                    - first_address_to_read) + ")"
                    TCP_Tx_Rx(send_command)
                    MetOceanCommand.parseMetOceanCommand(received_MetOcean_data)
                    execute_command()
                    first_address_to_read = last_address_to_read + 1
                    last_address_to_read = first_address_to_read + 1023
                    if (last_address_to_read > int(memory_used)):
                        break
            send_command = GET + "-" + MEMORY_DOWNLOAD + "(" + MEMORY_READ_ADDRESS + "," +
str(first_address_to_read) + ";" + MEMORY_READ_BUFFER_LENGTH + "," + str(int(memory_used) -
                                                                  first_address_to_read) + ")"
            TCP_Tx_Rx(send_command)
            MetOceanCommand.parseMetOceanCommand(received_MetOcean_data)
            execute_command()
            delayed_data_file.close()


# Function for execute MetOcean protocol commands
def execute_command():
    global realtime_file
```

```
global delayed_data_file
if (MetOceanCommand.primary_command == GET):
    if(MetOceanCommand.secondary_command == INSTRUMENT):
        #Actualize all selected instrument widgets as stated in the argument list
        for argument in MetOceanCommand.arguments:
            argument_pair = argument.split(",")
            if (argument_pair[0] == INSTRUMENT_ID):
                instrument_id = argument_pair[1]
            elif (argument_pair[0] == TYPE_INSTRUMENT):
                if (argument_pair[1] == SBE_37SM):
                    instrument_type = "SBE 37SMP"
                    instrument_value = SBE_37SM
                elif (argument_pair[1] == NONE):
                    instrument_type = "NONE"
                    instrument_value = NONE
            elif (argument_pair[0] == INTERVAL):
                interval = argument_pair[1]
            elif (argument_pair[0] == SAMPLES_PER_INTERVAL):
                samples_interval = argument_pair[1]
            elif (argument_pair[0] == PROCESSING):
                if(argument_pair[1] == NONE):
                    processing_type  = "NONE"
                    processing_value = NONE_p
                elif (argument_pair[1] == AVERAGE):
                    processing_type = "Average"
                    processing_value = AVERAGE_p
                elif (argument_pair[1] == MEDIAN):
                    processing_type = "Median"
                    processing_value = MEDIAN_p
        for actual_widget in widget_instruments_array[int(instrument_id)]:
            if actual_widget[0] == 'Instrument':
                get_command ="type =" + actual_widget[1] + ".set( \"" + instrument_type +
                                                                                "\")"
                exec(get_command)
            if actual_widget[0] == 'Acquisition_Interval':
                get_command = "type =" + actual_widget[1] + ".set( \"" + interval + "\")"
                exec(get_command)
            if actual_widget[0] == 'Samples_Acquisition':
                get_command = "type =" + actual_widget[1] + ".set( \"" + samples_interval +
                                                                                "\")"
                exec(get_command)
            if actual_widget[0] == 'Processing':
                get_command = "type =" + actual_widget[1] + ".set( \"" + processing_type +
                                                                                "\")"
                exec(get_command)
    elif(MetOceanCommand.secondary_command == MODE):
        argument_pair = MetOceanCommand.arguments[0].split(",")
        if (argument_pair[1] == CONFIGURING):
            for instrument_chb in instruments_checkbuttons_list:
                command = instrument_chb +".config(state=NORMAL)"
                exec(command)
            stop_bt.config(state = DISABLED)
            printLineState("Datalogger in ", "black")
            printLineState("ADMINISTRATION ", "blue")
            printLineState("mode \n","black")
        else:
            printLineState("Datalogger in ", "black")
            printLineState("ACQUISITION ", "blue")
            printLineState("mode \n","black")
    elif(MetOceanCommand.secondary_command == DATE_TIME):
        for argument in MetOceanCommand.arguments:
            argument_pair = argument.split(",")
            if (argument_pair[0] == YEAR):
                year = argument_pair[1]
            elif (argument_pair[0] == MONTH):
                month = argument_pair[1]
            elif(argument_pair[0] == DAY):
                day = argument_pair[1]
            elif(argument_pair[0] == HOUR):
                hours = argument_pair[1]
            elif(argument_pair[0] == MINUTE):
                minutes = argument_pair[1]
            else:
                seconds = argument_pair[1]
```

```
            global datalogger_date_time
            datalogger_date_time = datetime.datetime(int(year),int( month), int(day),
                                                     int(hours), int(minutes), int(seconds))
            datetime_variable.set("Client's date/time: \n\t  " +
datetime.datetime.now().strftime("%c") + "\nDatalogger's date/time: \n\t" + month + "/" + day +
                                  "/"+ year+ " " + hours + ":" + minutes + ":" + seconds)
        elif(MetOceanCommand.secondary_command == REALTIME_DATA):
            if (MetOceanCommand.arguments[0] != "0,0"):
                text_data.insert(END, MetOceanCommand.arguments[0])
                realtime_file.write(MetOceanCommand.arguments[0])
        elif(MetOceanCommand.secondary_command == MEMORY_STATE):
            memory_arguments = MetOceanCommand.arguments[0].split(",")
            global memory_used
            memory_used = memory_arguments[1]
            memory_used = 127
            memory_state_text = "\nMemory used: " + str(memory_used) + " Bytes"
            memory_state_variable.set(memory_state_text)
        elif(MetOceanCommand.secondary_command == MEMORY_DOWNLOAD):
            delayed_data_file.write(MetOceanCommand.arguments[0])


def command_send(command):
    sck.send(command)


# Function for creating one string containing the command for a new widget creation in the
configuration tab of the GUI
def create_New_Widget_Command(aux_widget_name, function, arguments):
    aux_arguments = "("
    number_of_arguments =len(arguments)
    i = 1;
    for arg in arguments:
        if i != number_of_arguments:
            aux_arguments = aux_arguments + arg  + ","
        else:
            aux_arguments = aux_arguments + arg
        i = i+1
    aux_arguments = aux_arguments + ")"
    return(aux_widget_name + "=" + function + aux_arguments)


# Function for the checkbuttons of the sensors configuration screen
def checkFunc(inst):
    command = "cbstate = cb_bool_{}.get()".format(inst)
    exec(command)
    widget_name = "cb_enable_{}".format(str(inst))
    for actual_widget in widget_instruments_array[inst]:
        if cbstate == 1:
            try:
                command_text = actual_widget[1] + ".config(state = NORMAL)"
                exec(command_text)
            except:
                continue
        else:
            try:
                command_text = actual_widget[1] + ".config(state = DISABLED)"
                exec(command_text)
            except:
                continue
        for instrument_chb in instruments_checkbuttons_list:
           command = instrument_chb +".config(state=NORMAL)"
           exec(command)

# Function to print text in the state widget of the GUI
def printLineState(line, tag):
    text_state.config(state = NORMAL)
    text_state.insert(END, line, (tag))
    text_state.yview_moveto(1)

# Main function of the TCP client
def tcp_client( server_ip, server_port, test_keepalive, command ):
    message_count=0;
    global realtime_data_flag
    try:
```

```
        host_name = socket.gethostname()
        host_ip = socket.gethostbyname(host_name)
        return_value = SUCCESS
    except:
        return_value = UNABLE_TO_GET_HOST_IP
    finally:
        if return_value == UNABLE_TO_GET_HOST_IP:
            return return_value
    server_ip_split = server_ip.split(".")
    host_ip_split = host_ip.split(".")
    if (server_ip_split[0] != host_ip_split[0]) or (server_ip_split[1] != host_ip_split[1])or
                                                   (server_ip_split[2] != host_ip_split[2]):
        return NETWORK_MISTMATCH
    sck = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sck.connect((server_ip, server_port))
    sck.send(command)
    data = sck.recv(BUFFER_SIZE)
    global received_MetOcean_data
    received_MetOcean_data = data
    sck.close()
    return return_value

# Function to start one TCP connection with the datalogger
def connectFunction():
    printLineState("Connecting... \n", "black")
    send_command = GET + "-" + MODE + "(" + TYPE_MODE + ",0)"
    if (TCP_Tx_Rx(send_command) == SUCCESS):
        printLineState("Connected to the datalogger \n", "green")
        MetOceanCommand.parseMetOceanCommand(received_MetOcean_data)
        execute_command()
        button_upload.config(state=NORMAL)
        button_refresh.config(state=NORMAL)
        button_connect.config(state=DISABLED)
        principal1.tab(1, state="normal")
        get_datetime()
        get_memory_state()
    else:
        printLineState("Unable to connect to the datalogger \n", "red")

# Function to start a TCP transmission
def TCP_Tx_Rx(command):
    parser = optparse.OptionParser()
    parser.add_option("--hostip", dest="hostip", default=DEFAULT_IP, help="Hostip to listen
                                                                          on.")
    parser.add_option("-p", "--port", dest="port", type="int", default=DEFAULT_PORT, help="Port
                                                  to listen on [default: %default].")
    parser.add_option("--test_keepalive", dest="test_keepalive", type="int",
                      default=DEFAULT_KEEP_ALIVE, help="Test keepalive capability")
    (options, args) = parser.parse_args()
    connect = tcp_client(options.hostip, options.port, options.test_keepalive, command)
    return connect

# Function to send one instrument configuration to the datalogger. This function
# uses the Set Instrument MetOcean Command
def single_instrument_upload(inst):
    send_command = SET + "-"+ INSTRUMENT + "("
    for actual_widget in widget_instruments_array[inst]:
        try:
            if actual_widget[0] == 'Instrument':
                get_command ="type =" + actual_widget[1] + ".get()"
                exec(get_command)
                if type == "SBE 37SMP":
                    typevalue = SBE_37SM
                if type == "NONE":
                    typevalue = NONE
                if type == "":
                    value_processing = "Select one value for Type Instrument"
                    raise ValueError
                send_command = send_command + INSTRUMENT_ID + ","+ str(inst) + ";" +
                                                  TYPE_INSTRUMENT + "," + typevalue
            if actual_widget[0] == 'Acquisition_Interval':
                get_command = "interval =" + actual_widget[1] + ".get()"
                exec(get_command)
                send_command = send_command + ';'
```

```
                send_command = send_command  + INTERVAL + ',' + str(interval)
                if interval == "":
                    value_processing = "Add one value for Acquisition Interval"
                    raise ValueError
                if (interval.isdigit() == 0):
                    value_processing = "Acquisition Interval must be positive integer"
                    raise ValueError
            if actual_widget[0] == 'Samples_Acquisition':
                get_command = "samples_per_acquisition =" + actual_widget[1] + ".get()"
                exec(get_command)
                send_command = send_command + ';'
                send_command = send_command + SAMPLES_PER_INTERVAL + ',' +
                                              str(samples_per_acquisition)
                if samples_per_acquisition == "":
                    value_processing = "Add one value for Samples per Acquisition"
                    raise ValueError
                if (samples_per_acquisition.isdigit() == 0):
                    value_processing = "Samples per Acquisition must be  positive integer"
                    raise ValueError
            if actual_widget[0] == 'Processing':
                get_command = "processing =" + actual_widget[1] + ".get()"
                exec(get_command)
                send_command = send_command + ';'
                if processing == "NONE":
                    type_processing = NONE_p
                if processing == "Median":
                    type_processing = MEDIAN_p
                if processing == "Average":
                    type_processing = AVERAGE_p
                    send_command = send_command + PROCESSING + ',' + type_processing
                if processing == "":
                    value_processing = "Select one value for Acquisition processing"
                    raise ValueError
        except ValueError:
            tkMessageBox.showinfo("Instrument configuration", value_processing)
            return

    send_command = send_command + ")"
    TCP_Tx_Rx(send_command)

# Function to send the configuration of all the actived instruments
# to the datalogger
def upload_all():
    for i in range(8):
        cb_boolean_variable = "boolean_value = cb_bool_{}".format(str(i)) + ".get()"
        exec(cb_boolean_variable)
        if boolean_value == 1:
            single_instrument_upload(i)

# Function to get the stored configuration that the datalogger has for
# the selected instrument, and upload the GUI content
def single_instrument_refresh(inst):
    send_command = GET + "-" + INSTRUMENT + "(" + INSTRUMENT_ID + "," +  str(inst) + ")"
    TCP_Tx_Rx(send_command)
    MetOceanCommand.parseMetOceanCommand(received_MetOcean_data)
    execute_command()

# Function to get the stored configuratoin that the datalogger has for
# all instrument, and upload the GUI content
def refresh_all():
    for i in range(8):
        single_instrument_refresh(i)

# Send the command for setting the datalogger in acquisition mode
def setAquisitionMode():
    if (start_rd.get() == 1):
        try:
            delayed_acquire_date_time = start_en.get()
            value_processing = "Please, insert a valid date & time value:\n MM/DD/YY hh:mm:ss"
            aux_fields_delayed_acquire_date = delayed_acquire_date_time.split("/")
            if(len(aux_fields_delayed_acquire_date) != 3):
                raise ValueError
            month = aux_fields_delayed_acquire_date[0]
            day = aux_fields_delayed_acquire_date[1]
```

```
            aux_fields_delayed_acquire_date = aux_fields_delayed_acquire_date[2].split(" ")
            if(len(aux_fields_delayed_acquire_date) != 2):
                raise ValueError
            year = aux_fields_delayed_acquire_date[0]
            aux_fields_delayed_acquire_date = aux_fields_delayed_acquire_date[1].split(":")
            if(len(aux_fields_delayed_acquire_date) != 3):
                raise ValueError
            hours = aux_fields_delayed_acquire_date[0]
            hours_int = int(hours)
            if ((hours_int < 0) or (hours_int > 23)):
                value_processing = "The hours value must be between 00 and 23"
                raise ValueError
            minutes = aux_fields_delayed_acquire_date[1]
            minutes_int = int(minutes)
            if ((minutes_int < 0)or(minutes_int > 59)):
                value_processing = "The minutes value must be between 00 and 59"
                raise ValueError
            seconds=aux_fields_delayed_acquire_date[2]
            seconds_int = int(seconds)
            if ((seconds_int < 0)or(seconds_int > 59)):
                value_processing = "The seconds value must be between 00 and 59"
                raise ValueError
            year_int = int(year)
            if ((year_int < 0)or(year_int > 99)):
                value_processing = "The year value must be between 00 and 99"
                raise ValueError
            month_int = int(month)
            if ((month_int < 0)or(month_int > 12)):
                value_processing = "The month value must be between 01 and 12"
                raise ValueError
            day_int = int(day)
            if((month_int == 1)or(month_int == 3)or(month_int == 5)or(month_int ==
                    7)or(month_int == 8)or(month_int == 10)or(month_int == 12)):
                if((day_int < 0)or(day_int > 31)):
                    value_processing = "The day value for month {} must be between 01 and
                                                                        31".format(month)
                    raise ValueError
            elif((month_int == 4)or(month_int == 6)or(month_int == 9)or(month_int == 11)):
                if((day_int < 0)or(day_int > 30)):
                    value_processing = "The day value for month {} must be between 10 and
                                                                        30".format(month)
                    raise ValueError
            else:
                if(((year_int % 4) == 0) and ((year_int % 100) != 0) or ((year_int % 400) ==
                                                                                0)):
                    if((day_int < 0)or(day_int > 29)):
                        value_processing = "The day value for month {} and year {} must be
                                                between 01 and 29".format(month, year)
                        raise ValueError
                else:
                    if((day_int < 0)or(day_int > 28)):
                        value_processing = "The day value for month {} and year {} must be
                                                between 01 and 28".format(month, year)
                        raise ValueError

        except ValueError:
            tkMessageBox.showinfo("Delayed acquisition", value_processing)
            return
        get_datetime()
        valid_delayed_acquisition_date_time = datetime.datetime(year_int+2000, month_int,
                                    day_int, hours_int, minutes_int, seconds_int)
        if (valid_delayed_acquisition_date_time < datalogger_date_time):
            tkMessageBox.showinfo("Delayed acquisition", "Acquisition date and time is earlier
                                                 that the datalogger's date and time")
            return
        aux_utc_seconds_start = str((valid_delayed_acquisition_date_time -
                                    datetime.datetime(1970, 1, 1)).total_seconds())
        utc_seconds_start = aux_utc_seconds_start.split(".")[0]
    else:
        utc_seconds_start = "0"
    if (enable_realtime_variable.get() == 1):
        global realtime_file_path
        realtime_file_path = tkFileDialog.asksaveasfilename(title = "Select file to save real
```

```
                                time data", filetypes = [("MetOcean data files", "*.mod")])
        if (realtime_file_path == ""):
            return
    send_command = SET + "-" + MODE + "(" + TYPE_MODE + "," + ACQUIRING + ";" +
UTC_SECONDS_ACQUIRE_START + "," + utc_seconds_start + ";" + MEMORY_WRITE_FLAG + "," +
                                        str(store_acquired_variable.get()) +  ")"
    TCP_Tx_Rx(send_command)
    if (enable_realtime_variable.get() == 1):
        t = threading.Thread(target=realtime_data, args=(float(utc_seconds_start),))
        t.start()
    printLineState("Datalogger in ", "black")
    printLineState("ACQUISITION ", "blue")
    printLineState("mode \n","black")
    if (start_rd.get() == 1):
        printLineState("\t Waiting to start on ", "blue")
        printLineState(month + "/" + day + "/20" + year + " at " + hours + ":" + minutes + ":" +
                                                                        seconds + "\n", "blue")
    principal1.tab(0, state = DISABLED)
    start_bt.config(state = DISABLED)
    stop_bt.config(state = NORMAL)
    start_rb_1.config(state = DISABLED)
    start_rb_2.config(state = DISABLED)
    getdatetme_bt.config(state = DISABLED)
    setdatetme_bt.config(state = DISABLED)
    erasememory_bt.config(state  = DISABLED)
    downloaddata_bt.config(state  = DISABLED)
    enable_realtime_cb.config(state = DISABLED)
    store_data_memory.config(state = DISABLED)

# Send the command for setting the datalogger in administration mode
def setConfigurationMode():
    global realtime_data_flag
    realtime_data_flag = 0
    send_command = SET + "-" + MODE + "(" + TYPE_MODE + "," + CONFIGURING + ")"
    TCP_Tx_Rx(send_command)
    printLineState("Datalogger in ", "black")
    printLineState("ADMINISTRATION ", "blue")
    printLineState("mode \n","black")
    principal1.tab(0, state = NORMAL)
    start_bt.config(state = NORMAL)
    stop_bt.config(state = DISABLED)
    start_rb_1.config(state = NORMAL)
    start_rb_2.config(state = NORMAL)
    getdatetme_bt.config(state = NORMAL)
    setdatetme_bt.config(state = NORMAL)
    erasememory_bt.config(state  = NORMAL)
    downloaddata_bt.config(state  = NORMAL)
    enable_realtime_cb.config(state = NORMAL)
    store_data_memory.config(state = NORMAL)
    get_memory_state()

# Function to get the date and time from the RTC of the datalogger
# and update the GUI screen
def get_datetime():
    send_command = GET + "-" + DATE_TIME + "(0,0)"
    TCP_Tx_Rx(send_command)
    MetOceanCommand.parseMetOceanCommand(received_MetOcean_data)
    execute_command()

# Function to send the client's date and time to the datalogger
# the datalogger will update it's date and time
def set_datetime():
    now = datetime.datetime.utcnow()
    uct_seconds = str((now - datetime.datetime(1970, 1, 1)).total_seconds())
    send_command = SET + "-" + DATE_TIME + "(" + UTC_SECONDS + "," + uct_seconds.split(".")[0]
                                                                                +")"
    TCP_Tx_Rx(send_command)



# Main function. This function creates the client's GUI
if __name__ == '__main__':
    # Create the GUI
    top = Tk()
```

```
    top.resizable(False, False)
    principal1 = ttk.Notebook(top)
    tab1 = ttk.Frame(principal1)
    tab2 = ttk.Frame(principal1)
    principal1.add(tab1, text="Configuration")
    principal1.add(tab2, text="Acquisition", state = DISABLED)
    # Creation and placement of the instruments configuration widgets in the GUI
    frame_row = 0
    frame_column = 0
    for instrument in range(NUMBER_OF_INSTRUMENTS):
        widget_name = "frame_" + str(instrument)
        widget_command = create_New_Widget_Command(widget_name, "ttk.Labelframe", ["tab1", "text
                                    = \"Instrument {}\".format(str(instrument+1))"])
        # Create the list of this instrument widgets
        widget_list_name = "widget_list_" + str(instrument)
        update_list_command = widget_list_name + "=[]"
        exec(update_list_command)
        # Add new widget to the widget list
        instrument_tuple = ('main_frame', widget_name)
        #update_list_command = widget_list_name+".append({})".format(widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        #Place the main frame of the instrument configuration
        if frame_column == NUMBER_OF_INSTRUMENTS_PER_ROW:
            frame_column = 0
            frame_row = frame_row +1
        exec(widget_name + ".grid(row = {}, column = {}, sticky = W, pady =
                                    2)".format(frame_row, frame_column))
        frame_column = frame_column +1
        # Create and place the enable checkbutton
        cb_boolean_variable = "cb_bool_{}".format(str(instrument)) + "= IntVar()"
        exec(cb_boolean_variable)
        widget_name = "cb_enable_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "text = \"Enabled\"", "command = " +
                    "lambda: checkFunc({})".format(str(instrument)), "variable =" +
                    "cb_bool_{}".format(str(instrument)), "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Checkbutton",
                                                                    widget_options)
        instrument_tuple = ('Enable', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 5, column = 0, sticky = W, pady = 2)")
        instruments_checkbuttons_list.append(widget_name)
        # Create and place the instrument selecction label
        widget_name = "lb_sensor_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "text = \"Instrument: \"", "state =
                                                                    DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Label", widget_options)
        instrument_tuple = ('Ins_Label', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 0, column = 0, sticky = W, pady = 2)")
        # Create and place the instrument selecction combo
        cm_text_variable = "cm_instrument_{}".format(str(instrument)) + "= IntVar()"
        exec(cm_text_variable)
        widget_name = "cm_instrument_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "values = [\"NONE\", \"SBE 37SMP\"]",
"width = 12", "textvariable =" + "cm_instrument_{}".format(str(instrument)), "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Combobox", widget_options)
        instrument_tuple = ('Instrument', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 0, column = 1, columnspan = 3, sticky = W, pady = 2)")
        # Create and place the acquisition interval entry
        widget_name = "en_interval_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "width = 12", "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Entry", widget_options)
        instrument_tuple = ('Acquisition_Interval', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
```

```
        exec(update_list_command)
        exec(widget_name + ".grid(row = 2, column = 1, columnspan = 3, sticky = W, pady = 2)")
        # Create and place the samples per acquisition entry
        widget_name = "en_samples_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "width = 12", "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Entry", widget_options)
        instrument_tuple = ('Samples_Acquisition', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 3, column = 1, columnspan = 3, sticky = W, pady = 2)")
        # Create and place the acquisition interval label
        widget_name = "lb_acquisition_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "text = \"Acquisition Interval (s): \"",
                                                                "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Label", widget_options)
        instrument_tuple = ('Acquisition_Label', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 2, column = 0, sticky = W, pady = 2)")
        # Create and place the samples per acquisition label
        widget_name = "lb_samples_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "text = \"Samples per Acquisition: \"",
                                                              "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Label", widget_options)
        instrument_tuple = ('Samples_Label', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 3, column = 0, sticky = W, pady = 2)")
        # Create and place the processing method selecction combo
        widget_name = "cm_processing_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "values = [\"NONE\", \"Median\",
                                    \"Average\"]", "width = 12", "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Combobox", widget_options)
        instrument_tuple = ('Processing', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 4, column = 1, columnspan = 3, sticky = W, pady = 2)")
        # Create and place the processing method label
        widget_name = "lb_processing_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "text = \"Acquisition processing: \"",
                                                              "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Label", widget_options)
        instrument_tuple = ('Processing_Label', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 4, column = 0, sticky = W, pady = 2)")
        # Create and place the upload button
        widget_name = "bt_upload_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "text = \"Upload\"","width = 7", "command
= " + "lambda: single_instrument_upload({})".format(str(instrument)), "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Button", widget_options)
        instrument_tuple = ('Upload', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 5, column = 3, sticky = W, pady = 2)")
        # Create and place the refresh button
        widget_name = "bt_refresh_{}".format(str(instrument))
        widget_options = ["frame_" + str(instrument), "text = \"Refresh\"","width = 7", "command
 = " + "lambda: single_instrument_refresh({})".format(str(instrument)), "state = DISABLED"]
        widget_command = create_New_Widget_Command(widget_name, "ttk.Button", widget_options)
        instrument_tuple = ('Refresh', widget_name)
        update_list_command = widget_list_name+".append(instrument_tuple)"
        exec(widget_command)
        exec(update_list_command)
        exec(widget_name + ".grid(row = 5, column = 2, sticky = W, pady = 2)")
        # Update the array with the last created instruments widgets list
        update_array_command = "widget_instruments_array"+".append({})".format(widget_list_name)
        exec(update_array_command)
```

```
frame_text = ttk.Frame(top)
scrollb = Scrollbar(frame_text)
text_state = Text(frame_text, height=8, width = 120, yscrollcommand=scrollb.set, borderwidth
                                                                    = 3, relief = GROOVE)
scrollb.config(command=text_state.yview)
text_state.insert(END, "Client disconected. Connect to start.\n")
text_state.config(state = DISABLED)
text_state.tag_config("green", foreground="green")
text_state.tag_config("red", foreground="red")
text_state.tag_config("black", foreground="black")
text_state.tag_config("blue", foreground="blue")
frame_text.grid(row = 3, column = 0, columnspan = 5, sticky = W, pady = 2)
text_state.grid(row = 0, column = 0, sticky = W, pady = 2, padx = 2)
scrollb.grid(row = 0, column = 1, sticky = NS, pady = 2)
frame_buttons = ttk.Frame(tab1)
button_connect = ttk.Button(tab1, text="Connect", command = connectFunction)
button_upload = ttk.Button(frame_buttons, text="Upload All", state = DISABLED, command =
                                                                        upload_all)
button_refresh = ttk.Button(frame_buttons, text="Refresh All", state = DISABLED, command =
                                                                        refresh_all)
button_connect.grid(row = 3, column = 0, sticky = W, pady = 2)
button_upload.grid(row = 0, column = 1, sticky = E, pady = 2)
button_refresh.grid(row = 0, column = 0, sticky = W, pady = 2)
frame_buttons.grid(row = 3, column = 3, sticky = E, pady = 2)
logo = PIL.ImageTk.PhotoImage(PIL.Image.open("logo_v3.png"))
Logo_Label = Label(top, image=logo)
Logo_Label.grid(row = 0, column = 0, columnspan = 3, sticky = W, pady = 2)
principal1.grid(row = 1, column = 0, sticky = W, pady = 2)
###############################################################################################
##                                Acquisition tab
###############################################################################################
frame_acq = ttk.Frame(tab2)
frame_acq.grid(column = 0, row = 0)
frame_start= ttk.Labelframe(frame_acq, text = "Start/Stop Aquisition")
frame_start.grid(column = 0, row=0, pady = 2)
start_rd = IntVar()
start_rd.set(0)
start_rb_1 = ttk.Radiobutton(frame_start, variable=start_rd, value=0, text = "Start now")
start_rb_2 = ttk.Radiobutton(frame_start, variable=start_rd, value=1, text = "Start at: ")
start_rb_1.grid(column = 0, row = 0, sticky = W)
start_rb_2.grid(column = 0, row = 1, sticky = W)
start_en = ttk.Entry(frame_start, width =16)
start_en.grid(column = 1, row = 1)
start_bt = ttk.Button(frame_start, text = "Start Acquisition", command = setAquisitionMode)
stop_bt = ttk.Button(frame_start, text = "Stop Acquisition" , command =
                                                setConfigurationMode)
start_bt.grid(column = 0, row =2)
stop_bt.grid(column = 1, row =2)
store_acquired_variable = IntVar()
store_data_memory = ttk.Checkbutton(frame_start, text = "Store acquired data to memory",
                                                    variable = store_acquired_variable)
store_acquired_variable.set(1)
store_data_memory.grid(column = 0, columnspan = 2, row = 3, pady = 3, sticky = W)
enable_realtime_variable = IntVar()
enable_realtime_cb = ttk.Checkbutton(frame_start, text = "Enable realtime data", variable =
                                                                    enable_realtime_variable)
enable_realtime_cb.grid(column = 0, columnspan = 2, row = 4, pady = 2, sticky = W)
enable_realtime_variable.set(0)
realtime_fr = ttk.Frame(frame_start)
realtime_rd = IntVar()
realtime_rb_1 = ttk.Radiobutton(realtime_fr, variable=realtime_rd, value=0, text = "Real
                                                                        Time mode")
realtime_rb_2 = ttk.Radiobutton(realtime_fr, variable=start_rd, value=1, text = "Delayed
                                                                            mode")
realtime_rb_1.grid(column = 0, row = 0)
realtime_rb_2.grid(column = 0, row = 1)
frame_datetime = ttk.Labelframe(frame_acq, text ="Set/Get Date Time")
frame_datetime.grid(column = 0, row = 1, sticky = EW, pady = 2)
getdatetme_bt = ttk.Button(frame_datetime, text = "Get Date Time", command = get_datetime)
getdatetme_bt.grid(column = 0, row = 0, padx = 8)
setdatetme_bt = ttk.Button(frame_datetime, text = "Set Date Time", command = set_datetime)
setdatetme_bt.grid(column = 1, row = 0, padx = 8)
datetime_variable=StringVar()
datetime_lb = ttk.Label(frame_datetime, textvariable = datetime_variable, justify = LEFT)
```

```
    datetime_variable.set("Client's date/time: \n\nDatalogger's date/time: \n")
    datetime_lb.grid(column = 0, row = 1, columnspan = 2, sticky = W)
    frame_memory = ttk.Labelframe(frame_acq, text ="Memory management")
    frame_memory.grid(column = 0, row = 2, pady = 2, sticky = EW)
    erasememory_bt = ttk.Button(frame_memory, text = "Erase memory", command = erase_memory)
    erasememory_bt.grid(column = 0, row = 0, sticky = W)
    downloaddata_bt = ttk.Button(frame_memory, text = "Download data", command = save_data_file)
    downloaddata_bt.grid(column = 1, row = 0, sticky = E)
    memory_state_variable = StringVar()
    memorystate_lb = ttk.Label(frame_memory, textvariable = memory_state_variable, justify =
                                                                                    CENTER)
    memory_state_variable.set("\nMemory used")
    memorystate_lb.grid(column = 0, row = 1)
    frame_text_data = ttk.Frame(tab2)
    scrollb_data = Scrollbar(frame_text_data)
    text_data = Text(frame_text_data, height=20, width = 90, background = "black", foreground =
                                                "white", yscrollcommand=scrollb_data.set)
    scrollb_data.config(command=text_data.yview)
    frame_text_data.grid(row = 0, column = 1, columnspan = 5, sticky = W, pady = 2)
    text_data.grid(row = 0, column = 0, sticky = W, pady = 2)
    scrollb_data.grid(row = 0, column = 1, sticky = NS, pady = 2)
    top.mainloop()
```

Archivo MetOceanParser.py

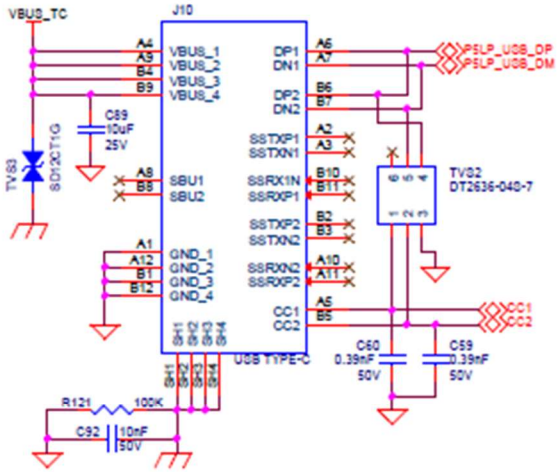
```
#coding=utf-8
#MetOceanParser.py
# Created on: 16/11/2019
#     Author: Jose Raul Santana Jimenez


# This file contains functions and declarations for the MetOcean protocol commands parsing

#!/usr/bin/env python

#####################################################################
# Labels for MetOcean principal commands definition
SET = '0'
GET = '1'
# Labels for MetOcean secondary commands definition
INSTRUMENT = '0'
MODE = '1'
DATE_TIME = '2'
REALTIME_DATA = '3'
MEMORY_DOWNLOAD = '4'
MEMORY_STATE = '5'
MEMORY_ERASE = '6'


#Labels for MetOcean commands arguments definition
INSTRUMENT_ID = '0'
TYPE_INSTRUMENT = '1'
TYPE_MODE = '2'
YEAR = '3'
MONTH = '4'
DAY = '5'
HOUR = '6'
MINUTE = '7'
SECOND = '8'
UTC_SECONDS = '9'
UTC_SECONDS_ACQUIRE_START = '10'
REALTIME_DATA_arg = '11'
MEMORY_STATE_Arg = '12'
MEMORY_WRITE_FLAG = '13'
MEMORY_READ_ADDRESS = '14'
MEMORY_READ_BUFFER_LENGTH = '15'
INTERVAL = '16'
SAMPLES_PER_INTERVAL = '17'
PROCESSING = '18'

# Labels for TYPE_INSTRUMENT argument possible values
```

```
NONE = '0'
SBE_37SM = '1'

# Labels for PROCESSING argument possible values

NONE_p = '0'
AVERAGE_p = '1'
MEDIAN_p = '2'

# Labels for TYPE_MODE argument possible values
CONFIGURING = '0'
ACQUIRING = '1'

class MetOcean_object_command():
    def __init__(self, primary_command, secondary_command, arguments):
        self.principal_command = -1
        self.secondary_command = -1
        self.arguments = {}

#Function to parse one MetOcean command received from the datalogger
    def parseMetOceanCommand(self, received_command):

        #Obtain primary command
        substrings = str(received_command).split("-")
        auxiliar_command = substrings[1];
        self.primary_command = substrings[0];

        #Obtain secundary command
        substrings = auxiliar_command.split("(")
        self.secondary_command = substrings[0]
        auxiliar_command = substrings[1]

        if (self.primary_command == GET):
            #Recover all arguments
            substrings = auxiliar_command.split(")")
            auxiliar_command = substrings[0]
            self.arguments = auxiliar_command.split(";")
```

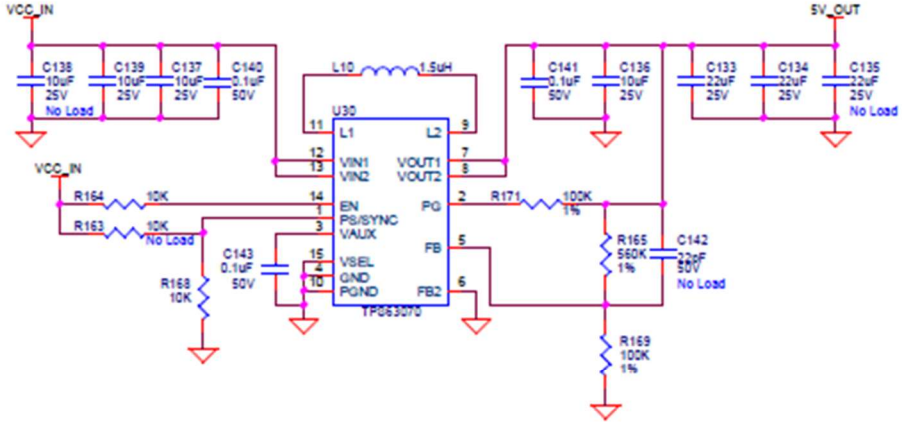
Anexo B. Esquemáticos del datalogger

En este anexo se recopilan los diagramas esquemáticos de los componentes usados en el *datalogger*. Los esquemáticos han sido extraídos de la guía de usuario de la tarjeta CY8CKIT-062-WIFI-BT y del *datasheet* de la tarjeta *RS232 Click*

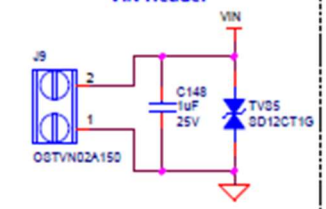
KitProg2 USB Type-C Connector

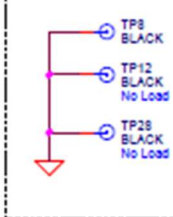


5V Buck-Boost Regulator

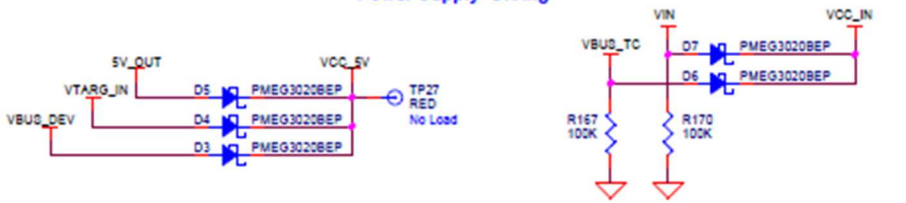


VIN Header

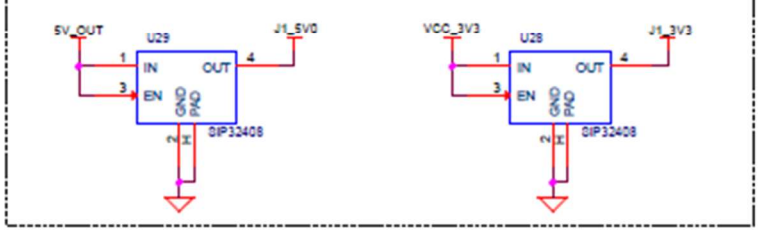




Power Supply 'OR'ing



Arduino Header O/P



CYPRESS SEMICONDUCTOR
198 CHAMPION COURT
SAN JOSE, CA 95134
(408) 943-2600

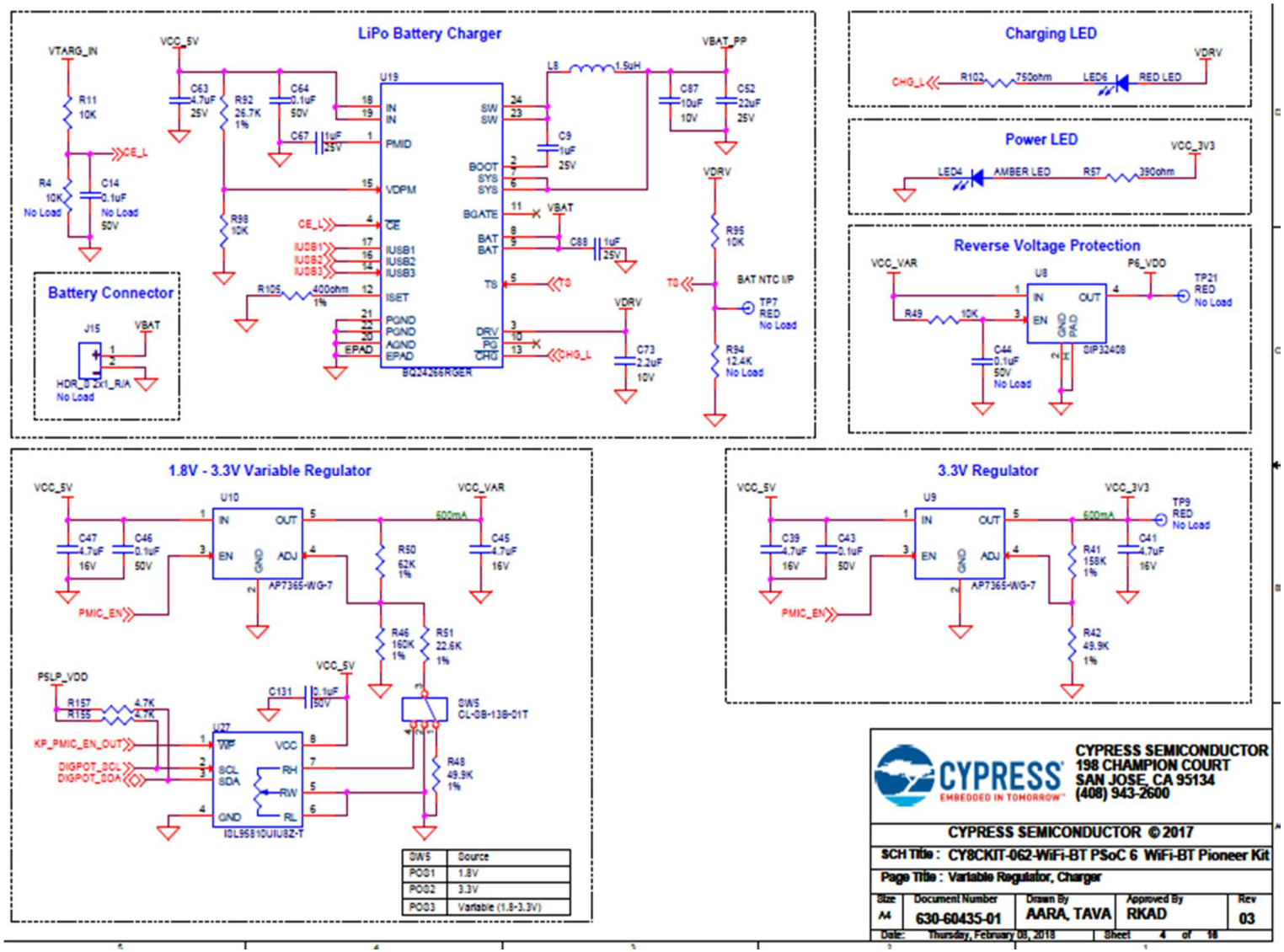
CYPRESS SEMICONDUCTOR ©2017

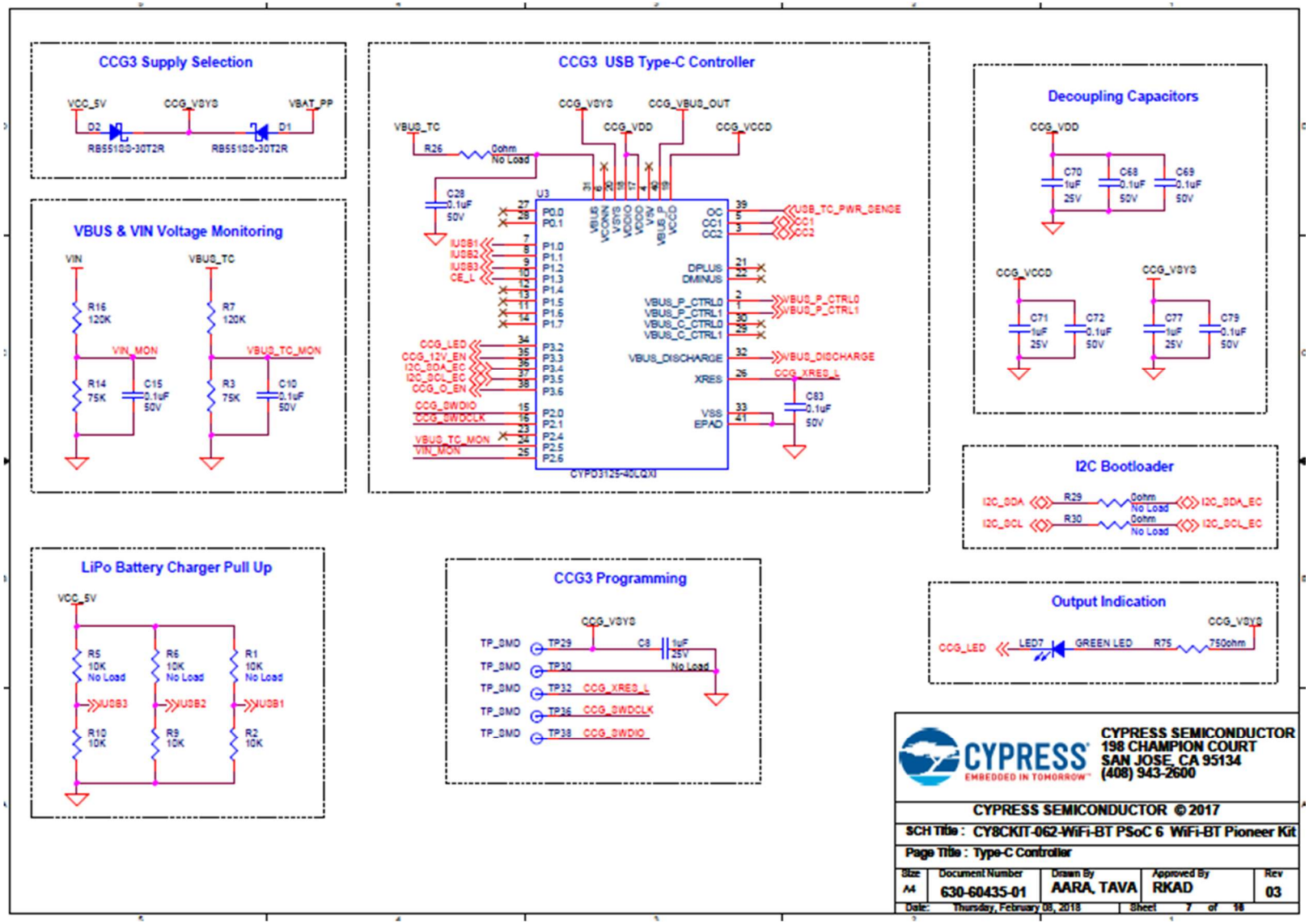
SCH Title : CY8CKIT-062-WiFi-BT PSoC 6 WiFi-BT Pioneer Kit

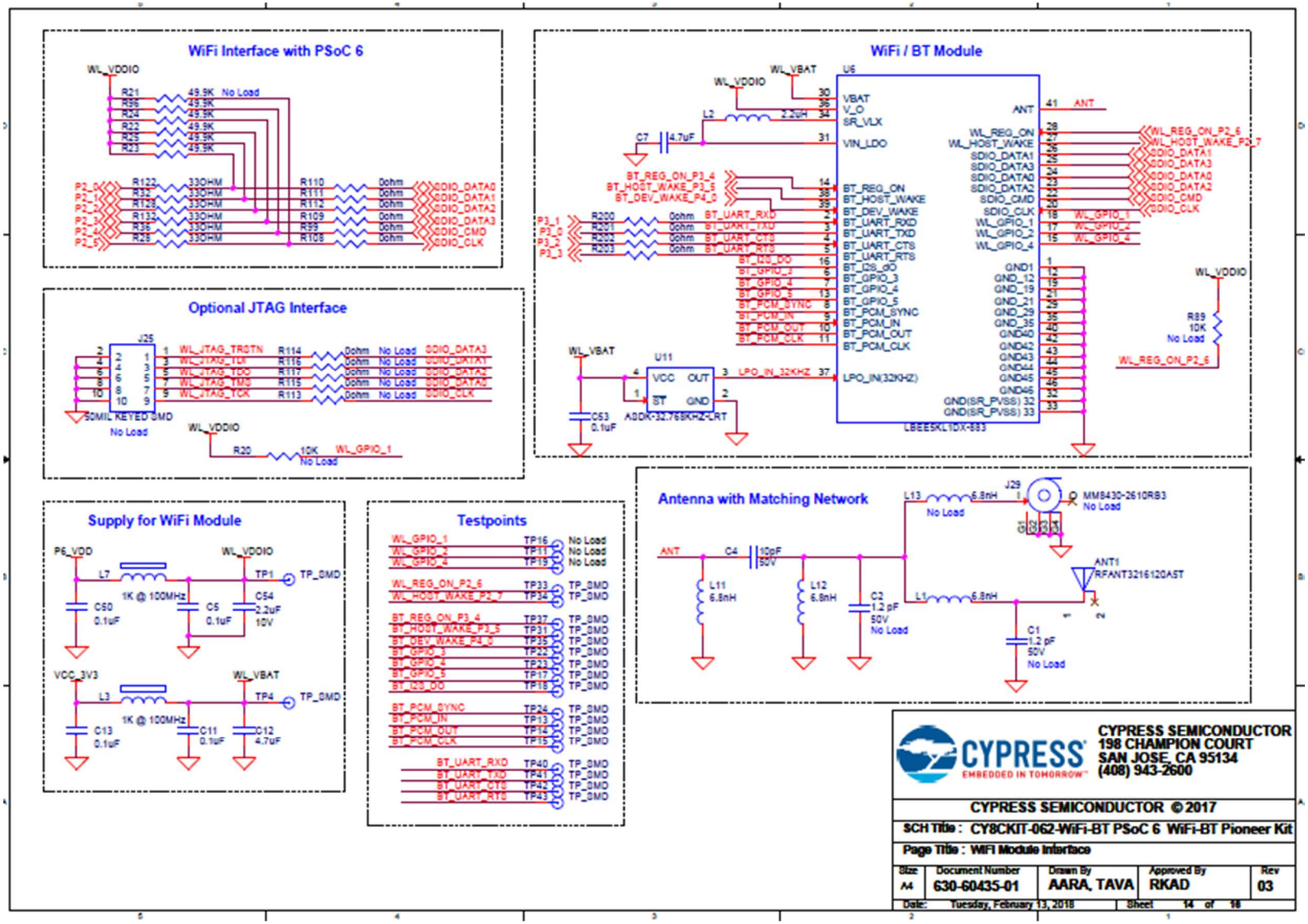
Page Title : 5V Buck/Boost Regulator

Size	Document Number	Drawn By	Approved By	Rev
A4	630-60435-01	AARA, TAVA	RKAD	03

Date: Thursday, February 08, 2018 | Sheet 3 of 18

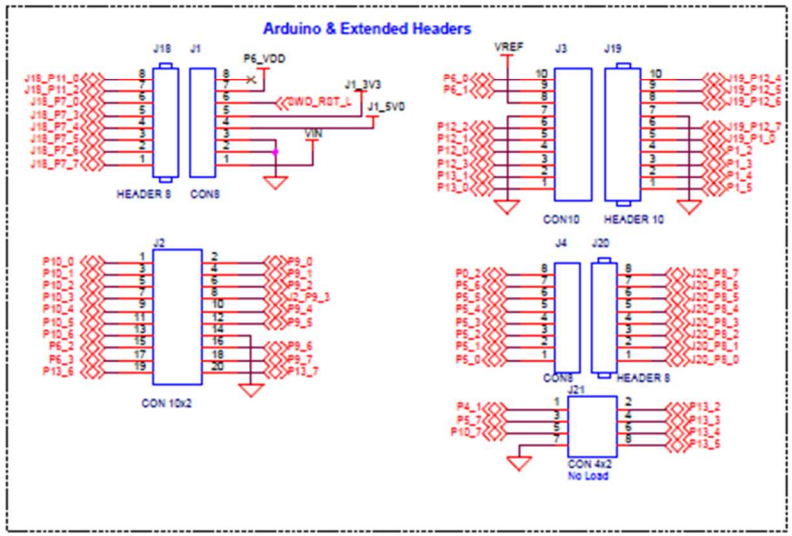

LiPo Battery Charger
Battery Connector
Charging LED
Power LED
Reverse Voltage Protection
1.8V - 3.3V Variable Regulator
3.3V Regulator
CYPRESS SEMICONDUCTOR
198 CHAMPION COURT
SAN JOSE, CA 95134
(408) 943-2600
CYPRESS SEMICONDUCTOR © 2017
SCH Title : CY8CKIT-062-WiFi-BT PSoC 6 WiFi-BT Pioneer Kit
Page Title : Variable Regulator, Charger
Size A4
Document Number 630-60435-01
Drawn By AARA, TAVA
Approved By RKAD
Rev 03
Date: Thursday, February 08, 2018
Sheet 4 of 18

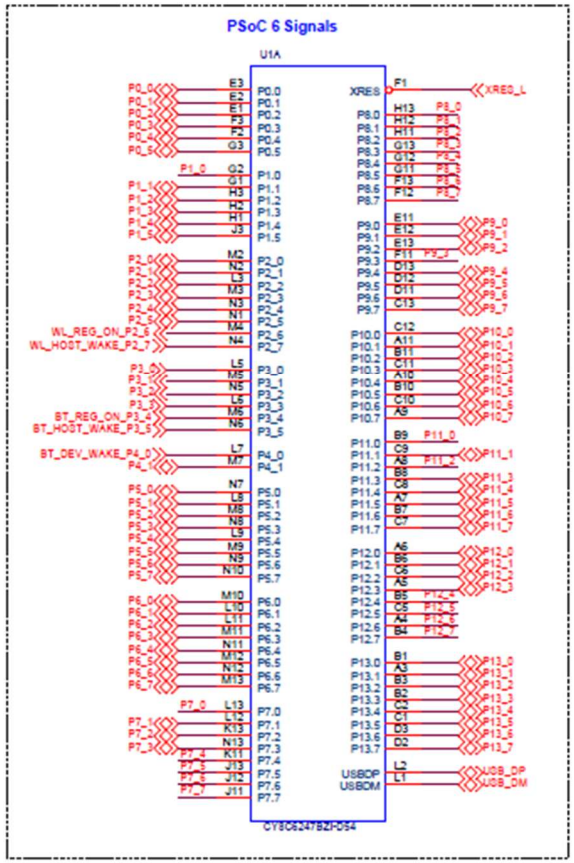

CCG3 Supply Selection
VCC_5V
CCG_VDD
VBAT_PP
D2
D1
RB551D0-30T2R
RB551D0-30T2R
VBUS & VIN Voltage Monitoring
VIN
VBUS_TC
R16
120K
R7
120K
VIN_MON
VBUS_TC_MON
R14
75K
C15
0.1uF
50V
R3
75K
C10
0.1uF
50V
CCG3 USB Type-C Controller
CCG_VDD
CCG_VBUS_OUT
VBUS_TC
R26
0ohm
No Load
CCG_VCCD
C28
0.1uF
50V
U3
P0.0
P0.1
P1.0
P1.1
P1.2
P1.3
P1.4
P1.5
P1.6
P1.7
P3.2
P3.3
P3.4
P3.5
P3.6
P2.0
P2.1
P2.4
P2.5
P2.6
VBUS
VCONN
VSYS
VDDIO
VIN
VBUS_P
VCCD
OC
CC1
CC2
USB_TC_PWR_SENSE
CC1
CC2
IUSB1
IUSB2
IUSB3
CE_L
DPLUS
DMINUS
VBUS_P_CTRL0
VBUS_P_CTRL1
VBUS_C_CTRL0
VBUS_C_CTRL1
VBUS_P_CTRL0
VBUS_P_CTRL1
VBUS_DISCHARGE
VBUS_DISCHARGE
XRES
CCG_XRES_L
C83
0.1uF
50V
VSS
EPAD
CCG_LED
CCG_12V_EN
I2C_SDA_EC
I2C_SCL_EC
CCG_O_EN
CCG_SWDIO
CCG_SWDCLK
VBUS_TC_MON
VIN_MON
CYPD3125-40LQXI
Decoupling Capacitors
CCG_VDD
C70
1uF
25V
C68
0.1uF
50V
C69
0.1uF
50V
CCG_VCCD
CCG_VSYS
C71
1uF
25V
C72
0.1uF
50V
C77
1uF
25V
C79
0.1uF
50V
I2C Bootloader
I2C_SDA
R29
0ohm
No Load
I2C_SDA_EC
I2C_SCL
R30
0ohm
No Load
I2C_SCL_EC
LiPo Battery Charger Pull Up
VCC_5V
R5
10K
No Load
R6
10K
No Load
R1
10K
No Load
IUSB3
IUSB2
IUSB1
R10
10K
R9
10K
R2
10K
CCG3 Programming
CCG_VSYS
TP_SMD
TP29
C8
1uF
25V
TP_SMD
TP30
No Load
TP_SMD
TP32
CCG_XRES_L
TP_SMD
TP36
CCG_SWDCLK
TP_SMD
TP38
CCG_SWDIO
Output Indication
CCG_VSYS
CCG_LED
LED7
GREEN LED
R75
750ohm
CYPRESS SEMICONDUCTOR
198 CHAMPION COURT
SAN JOSE, CA 95134
(408) 943-2600
CYPRESS SEMICONDUCTOR © 2017
SCH Title : CY8CKIT-062-WiFi-BT PSoC 6 WiFi-BT Pioneer Kit
Page Title : Type-C Controller
Size
A4
Document Number
630-60435-01
Drawn By
AARA, TAVA
Approved By
RKAD
Rev
03
Date:
Thursday, February 08, 2018
Sheet
7
of
16

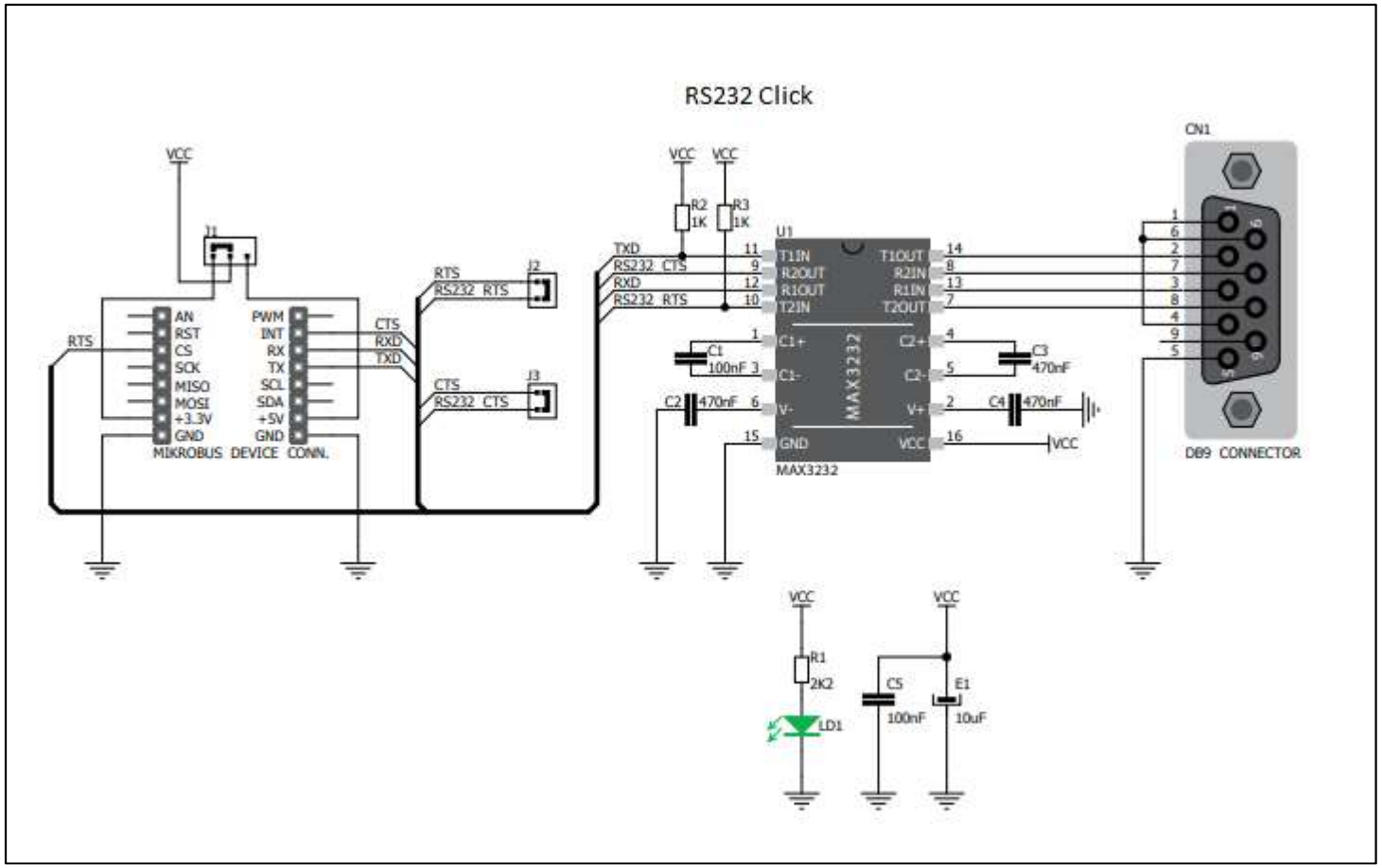

WiFi Interface with PSoC 6
Optional JTAG Interface
Supply for WiFi Module
Testpoints
WiFi / BT Module
Antenna with Matching Network
CYPRESS SEMICONDUCTOR
198 CHAMPION COURT
SAN JOSE, CA 95134
(408) 943-2600
CYPRESS SEMICONDUCTOR © 2017
SCH Title : CY8CKIT-062-WiFi-BT PSoC 6 WiFi-BT Pioneer Kit
Page Title : WiFi Module Interface
Size A4
Document Number 630-60435-01
Drawn By AARA, TAVA
Approved By RKAD
Rev 03
Date: Tuesday, February 13, 2018
Sheet 14 of 16


WCO
C85
33pF
50V
Y1
32.768KHz
ECS-.327-12.5-34B-TR
C86
18pF
50V
P0_1
P0_0

Reset Button
VDDD
R147
4.7K
XRES_L
SW1
EVQ-PE105K
C121
0.1uF
50V


Arduino & Extended Headers
J18
J1
P6_VDD
J1_3V3
SWD_RST_L
J1_5V0
VIN
HEADER 8
CON8
VREF
J3
J19
CON10
HEADER 10
J2
CON 10x2
J4
J20
CON8
HEADER 8
J21
CON 4x2
No Load


PSoC 6 Signals
U1A
XRES
XRES_L
USBDP
USBDM
USB_DP
USB_DM
CY8C6247BZI-D54


RS232 Click
VCC
J1
AN
RST
CS
SCK
MISO
MOSI
+3.3V
GND
PWM
INT
RX
TX
SCL
SDA
+5V
GND
MIKROBUS DEVICE CONN.
RTS
CTS
RXD
TXD
J2
RTS
RS232 RTS
J3
CTS
RS232 CTS
R2
1K
R3
1K
U1
TXD
RS232 CTS
RXD
RS232 RTS
T1IN
R2OUT
R1OUT
T2IN
T1OUT
R2IN
R1IN
T2OUT
C1+
C1-
V-
GND
C2+
C2-
V+
VCC
MAX3232
C1
100nF
C2
470nF
C3
470nF
C4
470nF
CN1
DB9 CONNECTOR
R1
2K2
LD1
C5
100nF
E1
10uF