Universidad de la República
Facultad de Ingeniería

BoyaFlores: Adquisidor de datos oceánicos con transmisión de datos inalámbrica utilizando red celular LTE-M

Memoria de proyecto presentada a la Facultad de Ingeniería de la Universidad de la República por

Diego Fraga, Rodrigo Bottero, Matías García

en cumplimiento parcial de los requerimientos para la finalización de la carrera de Ingeniería Eléctrica.

Tutor

Juan Pablo Oliver ..... Universidad de la República

Tribunal

Federico Favaro ..... Universidad de la República
Gabriel Gómez ..... Universidad de la República
Leonardo Steinfeld ..... Universidad de la República

Montevideo
viernes 8 agosto, 2025

BoyaFlores: Adquisidor de datos oceánicos con transmisión de datos inalámbrica utilizando red celular LTE-M, Diego Fraga, Rodrigo Bottero, Matías García.

Esta tesis fue preparada en LaTeX usando la clase iietesis (v1.2).
Contiene un total de 129 páginas.
Compilada el viernes 8 agosto, 2025.
http://iie.fing.edu.uy/

El desarrollo progresivo del hombre depende vitalmente de la invención; es el producto más importante de su cerebro creativo.

Nikola Tesla

Esta página ha sido intencionalmente dejada en blanco.

Agradecimientos

Agradecemos sinceramente a Antel por proporcionar la información técnica y la tarjeta SIM necesarias para el desarrollo de este proyecto, y especialmente a Linder Reyes por haber actuado de intermediario entre la empresa y nosotros, y a Marina Rogova por la gestión para el préstamo de la tarjeta SIM. Sin la colaboración de ambos, el análisis y la integración de la red LTE-M en nuestra plataforma no hubiera sido posible.

También agradecemos a Leonardo Steinfeld por prestarnos una placa Nordic nRF5340-dk la cuál permitió agilizar el proceso de familiarización con Zephyr durante los primeros meses de proyecto además de permitir la paralelización del trabajo.

Agradecemos a nuestro tutor Juan Pablo Oliver, por guiarnos durante todo este proceso.

Por último pero no menos importante queremos agradecer a nuestras familias y amigos por habernos apoyado incondicionalmente no solo en esta instancia sino que en toda la carrera.

Esta página ha sido intencionalmente dejada en blanco.

Resumen

El monitoreo de parámetros ambientales es cada vez más importante, más aún en áreas protegidas como la Isla de Flores, esto requiere soluciones tecnológicas robustas y energéticamente eficientes. Este proyecto desarrolla un nuevo adquisidor de datos capaz de medir temperatura y conductividad del agua (permitiendo calcular la salinidad a partir de estos parámetros), combinando autonomía energética, transmisión remota y bajo consumo, con miras a un funcionamiento continuo en entornos desafiantes.

Partiendo de un diseño previo, se estudiaron plataformas de hardware alternativas, seleccionando la Nordic nRF9160-dk, una solución de última generación ampliamente utilizada en la industria para aplicaciones de `IoT`, que integra módem `LTE-M` y permite mejoras sustanciales en tamaño y eficiencia energética. Basado en esta plataforma, se implementó un sistema alimentado por panel solar y batería de litio, el cual fue validado en pruebas de campo.

Mediciones del consumo realizadas con el Power Profiler Kit II de Nordic y un analizador Otii, permitieron caracterizar en detalle los distintos modos de operación y validar estrategias de optimización. Los resultados obtenidos demostraron las bondades de la plataforma elegida, registrándose un consumo promedio en modo de operación `NORMAL` (muestreo cada 1 minuto, transmisión cada 15 minutos) de $3,05mA$. Basándose en estas medidas se calculó una autonomía estimada de 42 días con una batería de $3,7V$ y $3200mAh$. Se validó la operación de forma ininterrumpida por más de 72 horas sin recarga (funcionando solo con la batería), así como la reconexión automática ante cortes de conectividad y energía, y la estabilidad de la transmisión incluso en condiciones de señal celular atenuada. Además, se verificó la estabilidad del sistema en todos los escenarios de uso y su escalabilidad futura, tanto a nivel de *hardware* como explorando mejoras en la eficiencia del *firmware* y mecanismos de actualización remota.

Se estudió también la viabilidad de nuevas configuraciones, como el funcionamiento exclusivo con baterías (sin panel solar), y el uso de antenas direccionales externas para entornos con baja cobertura `LTE-M`.

Finalmente, se identificaron oportunidades de expansión mediante herramientas para visualización remota de datos y despliegues prolongados en campo, consolidando así una base sólida para el desarrollo de redes de monitoreo ambiental autónomas, sostenibles y alineadas con las exigencias de aplicaciones reales.

Esta página ha sido intencionalmente dejada en blanco.

Tabla de contenidos

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 1

Introducción

1.1. Antecedentes

1.1.1. Equipo previo

Este proyecto tiene como antecedente un adquisidor de datos hecho por el Instituto de Ingeniería Eléctrica (IIE) en colaboración con la empresa Pyxis, el mismo busca resolver la problemática del monitoreo de áreas protegidas costeras. Con la creación de nuevas áreas protegidas surge la necesidad de evaluar el impacto ambiental de la actividad humana en las mismas y para ello es que desde el Ministerio de Ambiente buscan alternativas más económicas que las ofrecidas por el mercado.

El equipo previo fue probado de manera transitoria en un muelle del club Nautilus en la costa montevideana y posteriormente instalado en el primer punto de medida real: La Isla de Flores, área protegida desde 2018 ubicada a 17 km de la costa del Buceo.

Los parámetros de interés para evaluar la calidad de aguas oceánicas son la salinidad y la temperatura, entre otros, por lo tanto el equipo previo releva datos de temperatura y salinidad (la cual se calcula a partir de la conductividad y temperatura) del agua, así como temperatura interna y tensión de la batería, para tener un monitoreo interno del equipo.

El equipo previo, si bien es energéticamente autónomo, tiene una duración máxima de 3 días sin sol, la misma depende de la frecuencia de muestreo y envío de datos. Cuenta con un panel solar de $50W$ y un controlador de carga solar de $30A$ el cual carga una batería de ácido-plomo de $18Ah$. El sistema de carga alimenta una Raspberry Pi 3 [44] la cual se encarga de manejar tanto la adquisición de datos como la transmisión de los mismos utilizando un módem `LTE` conectado mediante `USB` a la placa.

En el equipo previo todo el sistema (batería, regulador, Raspberry, `PCB` y sensores internos), con excepción del panel solar y los sensores que van al agua, está contenido dentro de una caja estanca con certificación IP67 para asegurarse el no ingreso de agua ni partículas al sistema.

En cuanto a los sensores, el proyecto dispone de un `ADC` externo (más específi-

camente un ADS1115 de Texas Instrument) para medir el voltaje de la batería, dos sensores de temperatura DS18B20, uno de ellos con encapsulado a prueba de agua y otro junto a la Raspberry para verificar la temperatura interna en la caja estanca. Para medir la salinidad se contaba con un sensor de conductividad EZO-EC (luego los datos de conductividad y temperatura del agua se procesan para calcular el nivel de salinidad de la misma).

El proyecto fue exitoso: el sistema recolectó correctamente los datos de temperatura y conductividad del agua y los envió mediante peticiones `HTTP POST` a un *endpoint web*, incluyendo cada medida en formato `JSON`. Una vez recibidos, estos datos quedaron disponibles en el servidor para su tratamiento y análisis posteriores.

1.1.2. Aspectos a mejorar de esa primera implementación

Si bien el objetivo del proyecto original fue alcanzado con éxito, quedaron abiertos varios aspectos a mejorar del mismo.

En primer lugar, la plataforma elegida (Raspberry) está computacionalmente sobre-dimensionada respecto al procesamiento de datos necesario. Esto implica un consumo demasiado elevado (según estimativos entorno de 500 mA a 1 A). El consumo elevado genera la necesidad de acompañar el consumo con componentes capaces de suministrar la corriente necesaria resultando en el sobre-dimensionamiento de la batería, el regulador de carga y el panel solar.

El volumen y el peso de esta implementación fueron elevados, haciendo el sistema poco manejable y además requirieron de sistemas de sujeción muy robustos y voluminosos lo cuál es contraproducente cuando se quiere tomar una medida ambiental ya que lo deseable es interferir lo menos posible con el ambiente.

Otro punto en contra de la utilización de la placa Raspberry fue que necesitaba de módulos externos como ser el módem `LTE` para tener conectividad a internet y el `ADC` para monitorear el voltaje de la batería. Esto agrega complejidad ya que son dispositivos extra que hay que configurar y son más puntos de falla ante la eventualidad de que agua o humedad entren en el recipiente estanco.

En cuanto al software, este fue escrito en Python. Aunque el sistema permite acceso remoto por `SSH`, la ejecución del *script* en Python depende de tareas programadas en *crontab*, lo que obliga a editar manualmente las entradas cada vez que se quiere cambiar la frecuencia de muestreo, rutas de archivos o parámetros de transmisión.

Otra falencia respecto al software es que si bien la Raspberry se puede apagar remotamente, el sistema es incapaz completamente de volver a encenderse de manera remota, requiriendo que se prenda de forma manual. Esto hace que sea imposible ahorrar energía apagando la placa por lo que deberá permanecer encendida mientras que esté en uso, consumiendo energía constantemente.

Para maximizar la autonomía, es necesario desactivar puertos USB (el módem mas específicamente) mediante comandos de software a `GPIO`, pero esto anula la conexión a Internet mientras están apagados. Como consecuencia, no es posible reconectar de forma remota hasta que se reactive el puerto, lo que complica la

recuperación de datos y las tareas de mantenimiento. Estas limitaciones, sumadas a la gestión de *logs* y actualizaciones del sistema operativo, hacen que el uso de Raspberry Pi exija una planificación muy detallada de cada intervención presencial, reduciendo la flexibilidad de gestión remota.

1.2. Motivación

La motivación de este proyecto es corregir las deficiencias de la implementación anterior, evaluando un cambio de plataforma que ofrezca mayores funcionalidades que la Raspberry. Además, se busca mejorar la capacidad de realizar actualizaciones remotas de manera segura, prescindir de módulos externos (salvo el sensor de conductividad y el controlador de carga solar) y reducir el consumo del sistema.

Asociado al último punto, al reducir el consumo del sistema, se pueden disminuir considerablemente el tamaño de los elementos de alimentación (batería, panel solar y regulador) y, por tanto, las dimensiones generales del dispositivo. Esta baja en voluminosidad y peso dará un sistema mas manejable logísticamente y requerirá de sujeciones de menor porte, haciendo el equipo mas cómodo a la hora de instalar y/o mover.

En cuanto a mejoras en la adquisición de datos, el sistema debe ser más robusto ante pérdidas de energía. Se reforzará la capacidad para afrontar cortes de energía, sin pérdidas ni corrupción de información, de manera que al recuperar la conexión se pueda realizar la transmisión faltante, garantizando la continuidad del flujo de datos.

Además, el sistema deberá monitorizar el nivel de batería y, cuando éste caiga por debajo de cierto umbral definido, entrar en distintos modos de ahorro energético. Cada modo ajustará dinámicamente tanto la frecuencia de muestreo como la frecuencia de envío de datos, reduciendo progresivamente la actividad de adquisición y transmisión para optimizar el consumo y prolongar la autonomía.

1.3. Objetivos

1.3.1. Objetivo general

Diseñar una nueva versión del equipo previo, que sea de bajo consumo energético y apto para operar en lugares remotos. El equipo debe ser capaz de funcionar de forma autónoma mediante energía solar y baterías, idealmente, permitiendo la recolección y transmisión de datos a través de conectividad celular. Asimismo, debe lograr el mantenimiento mediante actualizaciones remotas de *firmware*, contemplar el recambio y gestión de sensores de forma práctica, y garantizar la operación continua frente a fallos de energía o conectividad. Todo esto protegido ante las condiciones ambientales adversas.

1.3.2. Objetivos específicos

- Diseñar un sistema de bajo consumo energético que optimice el uso de la energía disponible y maximice la autonomía operacional.
- Implementar un mecanismo de actualizaciones remotas de *firmware* que permita facilitar el mantenimiento del sistema en campo sin necesidad de intervención física.
- Diseñar un sistema de alimentación autónomo mediante baterías recargables con energía solar, asegurando su funcionamiento continuo en lugares sin acceso a la electricidad.
- Integrar conectividad celular al sistema para permitir el envío continuo de los datos recolectados desde ubicaciones remotas con cobertura móvil.
- Diseñar el equipo con un grado de protección adecuado, que asegure la resistencia frente al ingreso de agua, polvo o partículas del entorno.
- Garantizar la integridad de los datos ante fallos de energía, evitando su corrupción y permitiendo la reanudación normal del funcionamiento una vez recuperada la energía.
- Asegurar la continuidad del muestreo y almacenamiento de datos local incluso en caso de pérdida temporal de la conexión a internet.
- Integrar un sistema robusto que minimice la probabilidad de necesitar interacción humana para la recuperación de errores de *software/firmware*

1.4. Alcance

Este proyecto abarcará la implementación de un primer prototipo funcional del adquisidor de datos, con las mejoras y adiciones previamente dichas. El prototipo debe ser capaz de mantener la misma funcionalidad de su antecesor y demostrarlo en campo.

Resumiendo el alcance buscado para el desarrollo de esta segunda versión:

- Evaluación sobre *hardware* utilizado
  - Evaluación y selección de una plataforma de adquisición de datos alternativa que integre capacidades de procesamiento, conectividad celular y gestión de energía en una solución más eficiente y compacta
  - Evaluación de cambio de sensores y *hardware* auxiliar (dependiente de plataforma de desarrollo)
- En caso de cambio de plataforma de desarrollo
  - Reajustar la gestión de energía en concordancia con el nuevo *hardware*

    - Reajustar las protecciones ambientales elegidas para acomodar mejor el nuevo *hardware*

- Añadir actualizaciones de *firmware* `OTA` (*Over-The-Air*)
- Añadir funcionalidades que faciliten el mantenimiento en campo del dispositivo colocado
- Diseño base con visión a futuro, con posibilidades de crecimiento y expansión de funcionalidades
    - Dejar documentación en GitLab para que el proyecto pueda ser continuado
    - Modularizar todo lo que se pueda en el *firmware*, en caso que deba ser reemplazado o cambiado en el futuro

Si bien sería posible implementar una `PCB` diseñada específicamente para el proyecto, con los periféricos necesarios (como microprocesador, memoria externa y sistema de gestión de energía), esta alternativa se descarta por quedar fuera del alcance temporal del desarrollo.

También quedará por fuera del alcance de este proyecto cualquier caracterización a fondo de baterías o antenas de radiofrecuencia. En el caso de la batería se confiará en los valores nominales definidos por el fabricante y ya que tampoco es crucial tener estimativo exacto del nivel de carga de la misma no será necesario la caracterización completa.

Por otra parte, se utilizarán antenas que vengan incluidas con los kits comprados, no se comprarán antenas auxiliares ni tampoco se caracterizarán las propias de los kits.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 2

Hardware

2.1. Introducción

En esta sección se explicará el proceso de selección del hardware adquisidor de datos, los sensores utilizados y el sistema de gestión de energía, así como la elección de las protecciones físicas del equipo, todos ellos definidos en función de los requerimientos del proyecto.

2.2. Elección de hardware

2.2.1. Elección de adquisidor de datos

Como ya se explicó en Subsección 1.1.1 el proyecto antecesor utilizaba la plataforma Raspberry mas específicamente la Raspberry Pi 3. Si bien esta plataforma brinda prestaciones como la alta disponibilidad en plaza, precio accesible y un entorno amigable para desarrollar pudiendo correr Python sin mayores problemas, no fue hecha para una máxima eficiencia en el consumo cuando no está realizando tareas sin mucha exigencia. Tampoco cuenta con ciertos periféricos que deberán ser agregados externamente como ser un conversor analógico digital para monitorear el estado actual de la batería y por supuesto el módem para establecer una conexión a través de `LTE`.

Se evaluaron plataformas similares a la Raspberry como ser otras `SBC` (Single-Board Computer) de la misma marca y de la competencia, puesto que ofrecen la misma facilidad a la hora de reutilizar el contenido hecho en Python para el proyecto anterior pero ofreciéndonos en variedades con menores consumos.

A su vez se analizaron también otras alternativas a las `SBC` como son los microcontroladores estilo Arduino, que si bien tienen un entorno de desarrollo un poco menos amigable, teniendo que usarse lenguajes de programación como lo son C o C++ igualmente presentan bibliotecas ya armadas e inclusive ejemplos de implementación además de una comunidad grande de usuarios y foros activos.

La Tabla 2.1 resume las principales alternativas evaluadas, incluyendo aspectos técnicos y económicos relevantes para la decisión

Plataforma	Precio (USD)	Costo Total (USD)	Consumo LPM	Conectividad Celular	ADC Integrado	OTA Nativo	Antena Integrada
Raspberry Pi 3	60	158	100 mA	3G (modem ext.)	No	No	No
Raspberry Pi 4	109	234	100 mA	LTE-M (HAT)	No	No	No
ODROID N2L	59	157	∼100 mA	3G (modem ext.)	No	No	No
BeagleBoard BeaglePlay	99	197	∼100 mA	3G (modem ext.)	No	No	No
Arduino Portenta H7	114	216	670 μA	LTE-M (shield)	Sí	Sí	No
Nordic nRF9160 DK	**160**	**160**	$< 100\mu$**A**	**LTE-M nativa**	**Sí (12-bit)**	**Sí**	**Sí**
Sparkfun Thing Plus	139	139	$< 100\mu$A	LTE-M nativa	Sí (12-bit)	Sí	No

Tabla 2.1: Comparativa de plataformas de adquisición de datos evaluadas

Algunas SBC analizadas fueron la Raspberry Pi 5 [46] (ultima lanzada a la fecha) la cuál fue rápidamente descartada debido a su alto consumo energético. En su lugar se estudiaron alternativas de la competencia como ODROID N2L [31] ($59 USD) y BeagleBoard BeaglePlay [5] ($99 USD) llegando a la misma conclusión respecto al consumo. También se evaluó la Raspberry Pi 4 [45] con un costo total de $234 USD incluyendo el HAT celular necesario y consumo de $100mA$ en reposo con picos de $2 - 3A$. Debido al bajo requerimiento computacional que supone la recolección y envío de los datos adquiridos, se desestima la superioridad en poder de procesamiento de las SBC. Esto se traduce en que se descartan las mismas debido a su alto consumo energético.

Continuando investigando otros productos de la empresa Raspberry se estudió la posibilidad de utilizar una Raspberry Pi Pico 2 [47] la cuál presenta ciertas bondades como un consumo mucho menor además de la posibilidad de correr MicroPython lo que permitiría una fácil reutilización de los códigos del proyecto anterior además de añadir un ADC integrado a la misma placa. La gran desventaja que presenta es que no tiene manera directa de conectarse a internet mediante LTE (sólamente cuenta con WiFi). Esto supondría conseguir un módulo externo para la comunicación celular ya que no hay ninguna propuesta de la empresa para el caso de la Pico 2, por lo que también se decidió descartar esta opción.

Siguiendo la filosofía de la utilización de microcontroladores se estudiaron las opciones vigentes de la empresa Arduino las cuales contaban con la eficiencia energética ya mencionada, pero además contaban con opciones para conectividad celular a través de módulos llamados *shield* los cuales son convenientes ya que son plug-and-play con la plataforma e incluso tienen documentación, ejemplos y foros de soporte. Dentro de los microcontroladores analizados se encuentran 2 opciones, una de la gama consumidor (MKR NB 1500) y otra de gama profesional de Arduino (Portenta H7 [38] con *shield* CAT.M1/NB IoT GNSS). Como se muestra en la Tabla 2.1, la opción profesional presenta un consumo de $670\mu A$ y un costo total de $216 USD, mientras que ambas opciones ofrecen conversores analógico digital integrados para monitorear la batería y reguladores de tensión para poder alimentarse directamente a una batería de litio lo cual resulta conveniente para el proyecto. En ambos casos se requiere la compra de una antena externa, aunque esto no representa un obstáculo significativo dado que las antenas para estas aplicaciones están ampliamente disponibles y son compatibles con los conectores estándar de estos módulos. Además del problema de la antena, cualquiera de las dos soluciones provee soporte para la plataforma Cloud de Arduino, que provee al sistema de monitoreo remoto con interfaz gráfica multiplataforma (tanto en computadora

como en celulares), actualizaciones de *firmware over the air*, almacenamiento de datos en la nube y exportación de datos de manera fácil y rápida a formato `CSV`. El acceso completo a estos servicios requiere planes con precios diferenciados, lo que compromete la escalabilidad del proyecto debido al coste mensual de mantenimiento. Aunque ambas opciones incluyen conversores analógico-digital para el monitoreo de la batería y reguladores de tensión, lo que permitiría alimentar el dispositivo directamente desde una batería de litio, en ambos casos es necesario adquirir una antena externa, dejando de ser soluciones completamente integradas.

Por último se investigó sobre la plataforma de Nordic más específicamente la familia nRF91 que es la que cuenta con conectividad celular. La gama nRF91 incluye el Nordic nRF9160, que se presenta como un `SiP` (System in Package). Para facilitar el desarrollo, Nordic ofrece el *development kit* nRF9160-dk [27], que integra el `SiP` junto con reguladores de voltaje para poder usarse con batería de litio, flash externa para el almacenamiento de datos, antena LTE integrada con soporte para bandas de $700 - 2200MHz$ y además cuenta con `ADC` integrado de 12 bits. El costo del *development kit* es de $160 USD, representando una solución todo-en-uno sin necesidad de componentes adicionales. Una ventaja significativa del nRF9160-dk, claramente visible en la Tabla 2.1, el bajo consumo de energía, inferior a $100\mu A$ en modo de bajo consumo. Esto representa una reducción de consumo de 1000 veces comparado con las alternativas basadas en Raspberry Pi, permitiendo una operación prolongada con batería.

Las principales características del nRF9160 se resumen en la documentación oficial de Nordic Semiconductor [29].

- Procesador Arm Cortex M33
- 256KB de RAM
- 1MB de flash interna
- ADC de 12-bit hasta 200 ksps
- 4x UART
- 4x I2C
- 4x SPI
- 2x real-time counter (RTC)
- 3x 32-bit timer with counter mode
- 32 general purpose I/O pins

La abundante disponibilidad de periféricos de comunicación del nRF9160 (`UART`, `I2C`, `SPI`) junto con sus 32 pines de propósito general permite la expansión del sistema con sensores adicionales según las necesidades específicas de cada implementación. Esta flexibilidad es muy valiosa considerando que las distintas aplicaciones de monitoreo ambiental pueden requerir la medición de parámetros adicionales.

La arquitectura modular del sistema facilita estas expansiones sin necesidad de cambios significativos en el diseño base.

Otra ventaja del *development kit* es que sus conexiones de I/O son compatibles con un *footprint* de Arduino Uno Rev3 lo que facilita el uso de *shields* que son de amplia disponibilidad en plaza.

También se evaluó como alternativa el Sparkfun Thing Plus nRF9160 [51] ($139 USD) que utiliza el mismo `SiP` pero en un formato más compacto; sin embargo, el *development kit* oficial de Nordic ofrece mejor documentación y soporte, además de incluir antena integrada, como se observa en la tabla comparativaTabla 2.1.

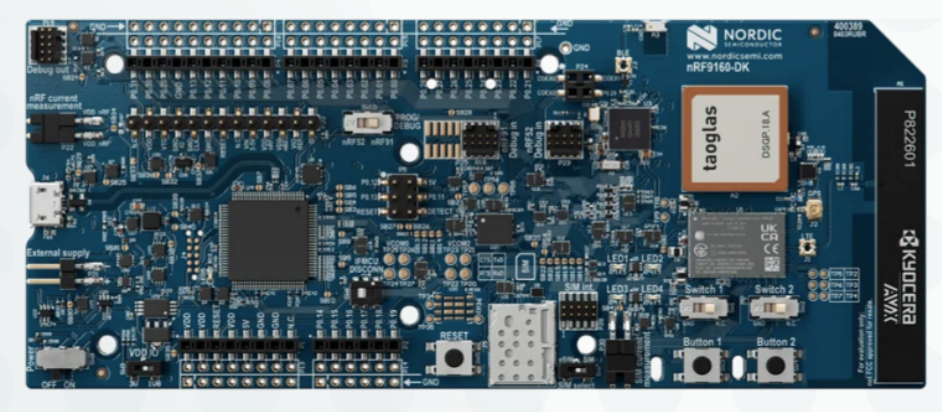


Figura 2.1: Diagrama de sensores y sus conexiones al equipo de desarrollo elegido. Tomada de [27].

Se decidió que la mejor opción para el proyecto sería cambiar de plataforma a la placa Nordic puesto que esta presentaba una solución todo en uno, cumple con los requisitos para llevar a cabo el proyecto y brindaba un bajo *footprint* de tamaño del sistema entero gracias a su bajo consumo.

2.2.2. Elección de sensores

Los sensores utilizados para el proyecto anterior serán reutilizados para este proyecto (con la excepción del `ADC` ya que la placa lo tiene incluido) puesto que dieron gran resultado en las pruebas de campo del proyecto antecesor.

Se utiliza el `DS18B20` como sensor de temperatura que cuenta con una interfaz digital 1-Wire. Estos sensores son muy asequibles y es posible conseguirlos en plaza facilitando su obtención y reemplazo. Una ventaja fundamental de esta clase de sensores de temperatura es que tienen disponible encapsulados a prueba de agua, lo que permite medir la temperatura del mar para estimar la salinidad de la misma.

El protocolo de comunicación 1-Wire es un protocolo digital de comunicación serial que se caracteriza por usar una sola línea para datos y tener modo de funcionamiento utilizando solamente dos cables. Es una solución de bajo consumo que sirve para largas distancias a costa de bajo *throughput* de datos lo cual en el caso del sensado de temperatura en los tiempos que se van a manejar no afectará

en nada. La comunicación es de tipo maestro-esclavo y la ventaja que tiene este protocolo es que se pueden conectar múltiples esclavos a un solo bus 1-Wire.

Figura 2.2: EZO-EC Conductivity Circuit.

Figura 2.3: Sonda Conductivity Probe K1.0 .

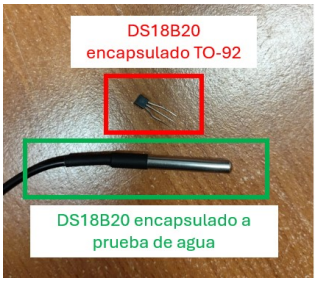


Figura 2.4: Sensor de temperatura DS18B20 junto a su versión con encapsulado a prueba de agua.

El sensor de conductividad utilizado es el EZO-EC Conductivity Circuit [16], acompañado de la sonda Conductivity Probe K1.0 [10], ambos de Atlas Scientific. El módulo EZO-EC opera entre $3{,}3V$ y $5V$ y consume aproximadamente $3{,}2mA$ en modo lectura activa, pudiendo bajar a $400\mu A$ si se coloca en *sleep* cuando no se requiere medir. Se comunica mediante `UART` o mediante `I2C`, lo cual facilita su integración con la placa Nordic. La sonda K1.0 permite medir conductividades en un rango de $0{,}07, \mu S/cm$ hasta $500,000, \mu S/cm$, lo cual cubre ampliamente los valores típicamente registrados en el entorno marino de la Isla de Flores según estudios previos [62]. Estas características hacen que el conjunto EZO-EC + K1.0 sea muy adecuado para este proyecto, ya que combina bajo consumo energético con un rango de medición compatible con las condiciones reales del sitio de despliegue.

2.2.3. Elección de sistema de gestión energética

Debido a los bajos requerimientos energéticos obtenidos tras el cambio de plataforma, fue posible reducir considerablemente el tamaño y la complejidad tanto del panel solar como del circuito de gestión de energía.

Panel solar

Para el panel solar se descartó reutilizar el módulo del proyecto anterior, ya que entregaba una potencia excesiva para los actuales requerimientos y tenía dimensiones significativamente mayores. Tras realizar una investigación y comparar las opciones disponibles en el mercado local, se determinó que la gran mayoría de los paneles que se comercializan en plaza están orientados a aplicaciones industriales o domésticas de gran escala. Estos paneles típicamente proporcionan potencias nominales que varían desde $50W$ hasta $300W$ (e incluso superan los $500W$ en algunos casos), y sus voltajes de operación en el punto de máxima potencia (V_{mp}) suelen ubicarse entre $17V$ y $38V$, dependiendo del número de celdas en serie. Di-

chos paneles están diseñados para sistemas de $12V$ o $24V$ nominales. En detalle:
Paneles de escala industrial/doméstica:

- *Potencia nominal*: 50–300W
- *Voltaje en circuito abierto (V_{oc})*: aproximadamente 21–48V, mientras que el voltaje en punto de máxima potencia (V_{mp}) suele estar en el rango de 17–38V
- *Corriente en punto de máxima potencia (I_{mp})*: entre $2A$ y $9A$, según la potencia nominal
- *Dimensiones típicas*: 300mm x 500mm a 670mm $\times$ 1000mm o mayores.
- *Encapsulado*: vidrio templado, marco de aluminio, soporta montajes fijos en techos o estructuras estáticas.
- *Protección IP*: generalmente IP65 o superior, adecuado para exteriores, pero con peso ($\geq 2kg$) y volumen que dificultan su montaje en un prototipo compacto.

Debido a que dichos paneles superan por mucho la demanda energética de nuestro sistema, considerando el tamaño y la necesidad de resistencia al agua, se tomó la decisión de importar un módulo de menor escala, pero con certificación adecuada para entornos húmedos. Finalmente, se eligió el Voltaic P105 [55], cuyas características principales son:

- Potencia nominal: $5W$
- Voltaje en circuito abierto (V_{oc}): $6V$
- Voltaje en punto de máxima potencia (V_{mp}): $5,25V$
- Corriente en punto de máxima potencia (I_{mp}): 0,95A
- Dimensiones: 137 mm (ancho) $\times$ 222 mm (largo) $\times$ 2.5 mm (grosor aproximado).
- Peso: 188 g.
- Certificación IP67: resistente a inmersiones ocasionales y al ingreso de polvo, ideal para instalación en proximidad al agua.

Su reducido tamaño y peso facilitan su montaje en estructuras ligeras sin comprometer la robustez requerida para operar en campo, lo que lo hace ideal para este proyecto.

En la Figura 2.5 se presenta una comparación visual entre el panel industrial previamente utilizado y el Voltaic P105, donde se aprecia claramente la diferencia de tamaños y formatos entre ambos.

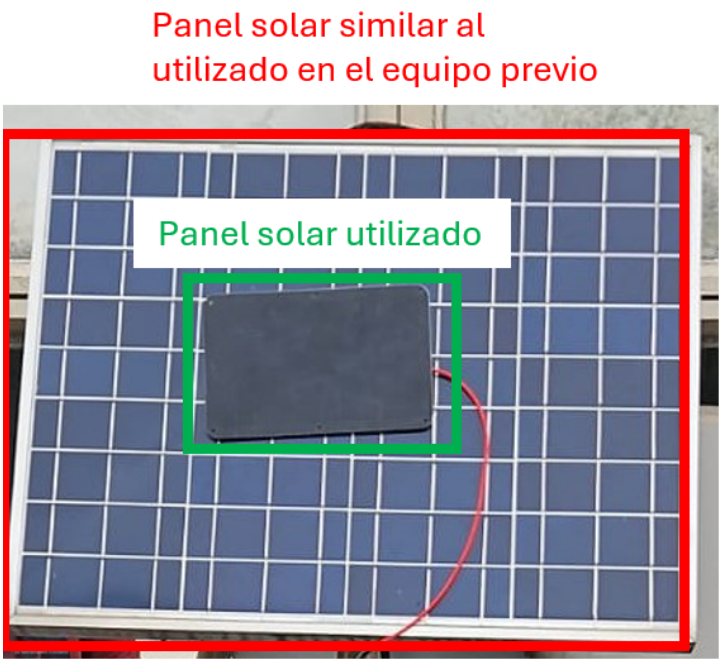


Figura 2.5: Comparación de tamaño entre el panel utilizado y uno similar al del equipo previo.

Cargador de batería

El cargador solar utilizado es el modelo BQ24074 de Adafruit [56], basado en el chip homónimo de Texas Instruments. Este dispositivo permite cargar una batería de litio y posee un rango de voltaje a la entrada de $5 - 10V$. El controlador incorpora un sistema inteligente que reduce automáticamente el consumo de corriente cuando el voltaje de entrada disminuye, garantizando así una carga estable mientras proporciona una salida de voltaje regulado para alimentar la placa.

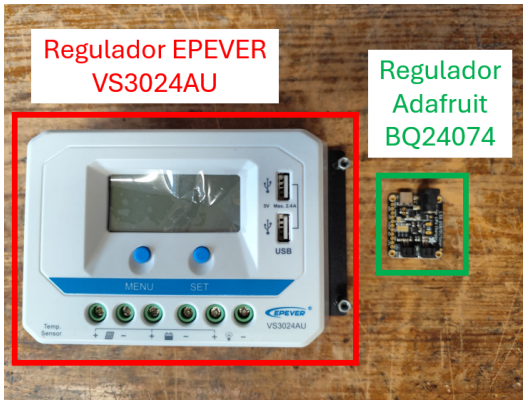


Figura 2.6: Comparación de los reguladores utilizados en el equipo previo (izquierda) con el utilizado en este proyecto (derecha).

Batería

Gracias al bajo consumo del sistema y con el afán de minimizar el volumen final del prototipo, se optó por una batería de ion de litio de formato 18650 para alimentar la placa. Este tipo de batería tiene una disponibilidad grande a nivel local y es un formato muy utilizado, por lo que permitió la fácil adquisición de la misma. Asimismo, ofrecen una elevada densidad energética, una tensión nominal de $3,7V$ (alcanzando $4,2V$ en plena carga) y una vida útil típica superior a 300 ciclos con la carga/descarga adecuada.

Esta batería de ion de litio que irá con un porta-pila conectada mediante un conector JST de 2 pines al cargador de batería.

Figura 2.7: Comparación de las baterías utilizadas en el equipo previo (izquierda) con la utilizada en este proyecto (derecha).

Impreso en el dorso de la batería elegida se muestra el valor $5800mAh$, este valor no fue considerado puesto a que no se pudo conseguir la hoja de datos de la batería utilizada. En su lugar, se utilizó la hoja de datos de una batería Panasonic NRC18650B [25] de $3200mAh$ como referencia para las respaldar las afirmaciones realizadas sobre la relación entre tensión y estado de carga de una batería de ion de litio, y también para cálculos de autonomía energética.

2.2.4. Esquema de conexiones a nivel hardware

A continuación (Figura 2.8) se presenta un esquema de conexiones con el *hardware* elegido, con sensores y periféricos sin el circuito de gestión de energía:

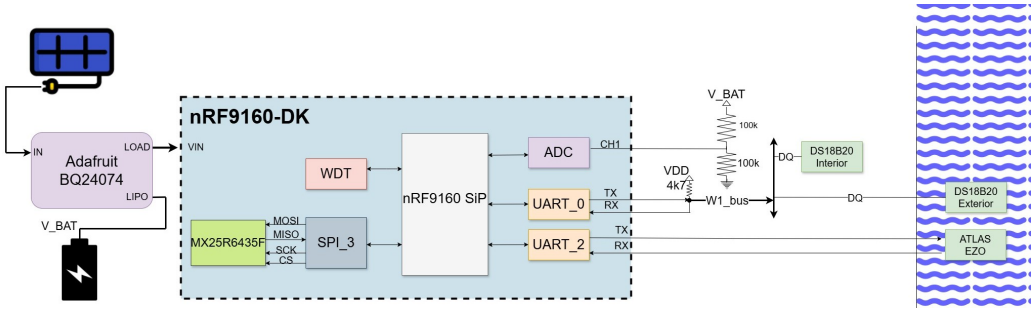


Figura 2.8: Diagrama de sensores y sus conexiones al equipo de desarrollo elegido.

En el esquema (Figura 2.8) se especifican además los periféricos utilizados del microprocesador nRF9160. También se detalla el divisor resistivo utilizado y el pull-up para el bus 1-Wire.

2.2.5. Gabinete

Se utilizó una caja estanca de medidas 145mm x 195mm x 40mm. La caja vino con tres agujeros ya hechos para pasar los cables, por lo que se utilizaron prensacables de tamaño adecuado para el sellado de los mismos y luego un tapón para el tercer agujero que queda en desuso en este proyecto.

Figura 2.9: Caja con prensacables y tapón instalados

La caja tenía otros agujeros extra para insertar alguna clase de tornillo (ver en la Figura 2.9, al lado del tapón) los cuales fueron tapados con sellador. Para verificar que los lugares donde se aplicó sellador hayan mantenido la estanqueidad de la caja, se probó sumergirla en su totalidad en un balde de agua, obteniendo resultados positivos.

2.3. Protocolo celular

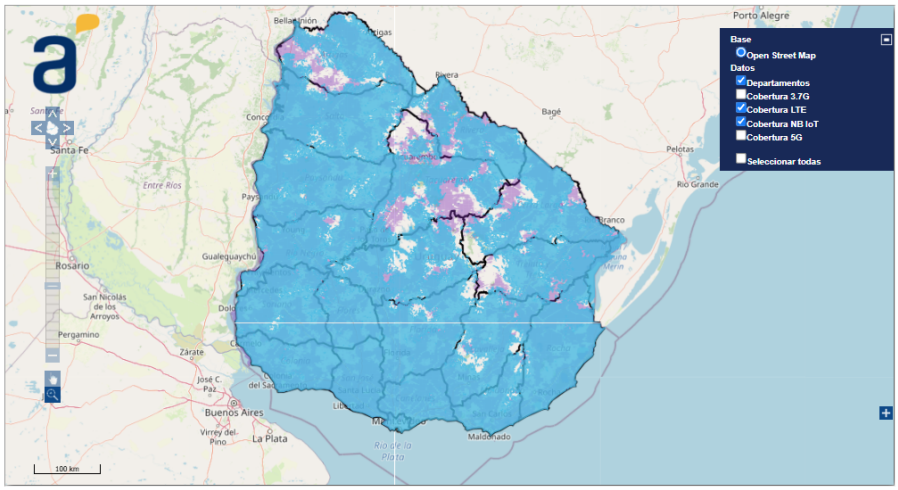
Se solicitó a la empresa nacional de telecomunicaciones Antel acerca de información y planes de sus redes específicas de IoT (Internet of Things), ya que no hay tanta información disponible al público como sí la hay de las redes para consumidores "domésticos" (es decir, las redes que utilizan los smartphones, como ser LTE y 5G). Conseguir información acerca de estas redes IoT fue un paso crucial al principio del proyecto, una vez se descartaron placas con módem celular externo como la Raspberry al haber seleccionado la plataforma Nordic; afortunadamente, la respuesta de la empresa fue positiva, brindando información técnica y una tarjeta *SIM* por la duración del proyecto.

Actualmente en Uruguay, Antel cuenta con servicios e infraestructura destinada a dispositivos IoT mediante los estándares NB-IoT (Narrow Band IoT) y LTE-M (Long Term Evolution for Machines). Ambos estándares tienen como objetivo ser utilizados por dispositivos de bajo consumo energético y de baja tasa de envío de datos, con mayor alcance que las redes LTE y 5G tradicionales pero con una capacidad bastante menor en cuanto a velocidades de transmisión de datos.

Las bandas que utilizan tanto LTE-M como NB-IoT en Antel son la 3 (1800MHz) y la 28 (700MHz), ambas se encuentran en el rango de bandas soportadas por el *development kit* elegido (de 700MHz a 2200MHz) por lo que no habrá problemas en cuanto a la compatibilidad en la conexión celular. El rango de cobertura de ambas es extenso y se encuentra presente en casi todo el territorio nacional, sin embargo, por las diferencias técnicas entre ambas, NB-IoT posee un mejor alcance, por lo tanto, mejor cobertura aunque hay bastante paridad entre ambas.

Además de la cobertura nacional (ver Figura 2.10), es pertinente destacar las diferencias prácticas de uso entre NB-IoT y LTE-M. NB-IoT suele ofrecer un consumo energético aún más bajo y mayor penetración en interiores, sacrificando velocidad de transmisión (ideal para tramas muy pequeñas cada horas). Por otro lado, LTE-M permite un ancho de banda suficiente para enviar paquetes más grandes y reduce la latencia. Esta característica es fundamental para implementar actualizaciones remotas del *firmware* y permite flexibilidad en la frecuencia de transmisión de datos según los requerimientos operativos del sistema. En nuestro caso, el SIM provisto por Antel estaba habilitado específicamente para LTE-M con el APN "antel.iot", por lo que se le dió prioridad a este estándar, aunque la plataforma Nordic permite cambiar dinámicamente a NB-IoT si fuese necesario.

Si bien el *hardware* elegido admite el uso de ambos estándares, se optó por LTE-M debido a que el SIM provisto por Antel fue prestado para el uso con esta tecnología.



Referencias

- Cobertura 3.7G: naranja
- Cobertura LTE: violeta
- Cobertura NB IoT: celeste
- Cobertura 5G: verde

Figura 2.10: Cobertura de redes `LTE-M` y `NB-IoT`. Tomada de [9].

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 3

Firmware

Este capítulo presenta el desarrollo del *firmware* del sistema implementado sobre el sistema operativo Zephyr utilizando C++. El proyecto adopta una arquitectura modular donde cada módulo se encarga de gestionar periféricos específicos y funcionalidades bien definidas, facilitando el desarrollo, testing y mantenimiento del código.

El capítulo aborda desde los fundamentos de Zephyr hasta la implementación completa de la arquitectura del sistema. Se describe la adquisición de datos, la gestión de almacenamiento a través del sistema de archivos `LittleFS`, y la implementación de la arquitectura de red basada en `MQTT`.

Además, se presenta el sistema de actualización remota de *firmware* que permite el mantenimiento del dispositivo sin intervención física, aspecto crítico para un sistema diseñado para operar en condiciones marítimas remotas donde la confiabilidad y la autonomía son fundamentales.

3.1. Fundamentos de Zephyr

3.1.1. Características de Zephyr

Zephyr [63] es un sistema operativo en tiempo real (`RTOS`) de código abierto, desarrollado como una iniciativa propia de la Fundación Linux [58] y lanzado en 2016 como proyecto colaborativo para dispositivos con recursos limitados. Está diseñado para sistemas con recursos limitados y sistemas embebidos, posee una gran comunidad activa que está constantemente desarrollando, mejorando y agregando soporte para múltiples arquitecturas, plataformas de desarrollo y distintos sensores.

Además de definir una estructura de *kernel* y un *scheduler* que soportan arquitecturas de ejecución en múltiples hilos (*multi-threading*) y también basados en eventos (*event-driven*) e interrupciones. Zephyr también ofrece servicios y `APIs` (Application Programming Interface) que facilita la integración de capas de abstracción de *hardware* (`HAL`) o implementación de aplicaciones con diferentes *stacks* de comunicación.

Una de sus principales ventajas frente a otros RTOS es su modularidad y capacidad de adaptación a distintas plataformas. Esto se logra gracias al uso del *Kconfig* [18] y del *Devicetree* [13].

El *Kconfig* permite seleccionar y compilar únicamente los módulos necesarios para el proyecto, esto incluye funcionalidades del *kernel*, subsistemas (*subsystems*) como ser sistemas de archivos (*file system*) y diferentes *drivers* los cuales implementan las interfaces para tratar con el *hardware* específico de cada dispositivo utilizados. Por otra parte, el *Devicetree* describe la configuración física del *hardware* utilizado, como ser los pines, buses o periféricos externos, permitiendo al compilador reconocer e identificar los dispositivos presentes para asociarlos con sus respectivos controladores (*drivers*) de forma automática durante el proceso de compilación.

La combinación entre ambas herramientas permite que el código sea portable entre distintas plataformas con mínimos cambios, al mismo tiempo que se optimiza el almacenamiento utilizado se evita la inclusión de módulos que no fueron configurados.

Otra ventaja es que Zephyr cuenta con la opción de que el *kernel* funcione en modo *tickless*, lo cual le brinda mayor eficiencia al no tener que despertar mediante interrupciones al CPU de manera periódica, pudiendo así tener un mejor desempeño energético ya que el CPU puede transicionar a modos de bajo consumo cuando no tiene tareas pendientes y se despierta sólo cuando recibe un evento. La interacción de esta funcionalidad del *kernel* de Zephyr con el sistema se detallará en la Subsección 3.4.3. La activación del CPU dada por eventos también es una ventaja cuando se trabaja con comunicaciones como ser la comunicación celular, ya que el *scheduler* puede elegir con mayor precisión los momentos cuando despertar al CPU y así poder implementar modos de bajo consumo que mantienen la conexión activa como ser *eDRX* (ver Subsección 3.4.1).

3.1.2. Conceptos y funcionalidades de Zephyr utilizados en el proyecto

Device driver model

Zephyr provee una capa de abstracción de *hardware* para interactuar y configurar diferentes periféricos mediante la interfaz llamada *device driver model* [12]. A partir de un nodo del *Devicetree*, se generan macros que vinculan al dispositivo de dicho nodo con el *driver* (controlador) correspondiente. Luego, en tiempo de ejecución se generan objetos de tipo *device* que pueden ser utilizados por la aplicación en que se esté trabajando mediante un *handle* (puntero que actúa como referencia al dispositivo sin exponer su implementación interna).

Threads (o hilos)

Un *thread* es una unidad de ejecución del *kernel* encargada de procesar instrucciones. Dado que el poder de procesamiento en un sistema embebido es limitado, Zephyr permite definir múltiples *threads*, ya sean creados por el usuario o por

el propio sistema operativo, cada uno con características y prioridades específicas. El *scheduler* se encarga de planificar cuál *thread* se ejecutará a continuación, asignándole la capacidad de procesamiento del procesador y retirándosela en caso de que ocurra una interrupción o se encole otro *thread* con mayor prioridad.

Esta funcionalidad es de gran utilidad para el proyecto ya que permite independizar la parte de muestreo de datos con la parte de comunicación, haciendo que errores en la comunicación no afecten el muestreo de datos y viceversa. Se profundizará en esto en Subsección 3.5.5.

Works (o trabajos)

Los trabajos son unidades de procesamiento diferido que se ejecutan en un hilo mediante encolado. Se utilizan para programar la ejecución de operaciones de alto costo de procesamiento que no requieren atención inmediata, por lo que normalmente tienen prioridad inferior tanto respecto a las rutinas de servicio de interrupciones (`ISRs`) como a *threads* de alta prioridad. Un uso común de los trabajos es procesar datos acumulados por `ISRs` en *buffers*. Su esquema de ejecución es similar al encolado de funciones en el desarrollo de sistemas embebidos, con el agregado de estar sincronizados por un *scheduler* que gestiona las prioridades.

Semáforos

Los semáforos son mecanismos del *kernel* utilizados principalmente para dos funciones: sincronizar *threads* y controlar el acceso concurrente a recursos compartidos. En Zephyr, los semáforos son por conteo, lo que permite que múltiples *threads* accedan a un recurso particular siempre que haya disponibilidad, limitando así el acceso simultáneo para evitar conflictos.

Como dato relevante para este proyecto, todos los semáforos definidos serán binarios, por lo que solo un *thread* podrá acceder al recurso a la vez. Esto es especialmente importante, ya que permitir múltiples accesos simultáneos a un único recurso puede provocar errores que podrían derivar en la corrupción de datos o incluso en la falla completa del sistema. Dado que se plantea utilizar el dispositivo en un entorno remoto, se debe minimizar cualquier situación que pueda provocar una falla total del sistema, ya que esta podría ser irrecuperable de forma remota. Este cuidado es esencial en todo caso y cobra especial relevancia en entornos donde el acceso físico es limitado.

WDT

Un `WDT` (*watchdog timer*) es un tipo especial de *timer* que en caso de expirar reinicia el *firmware* para recuperar el sistema de situaciones en las cuales el mismo podría estar atorado en un hilo de ejecución. Por ejemplo, si un `WDT` está configurado en 5 segundos y el programa entra en un *loop* del estilo while(true) , al expirar, el `WDT` reiniciará el sistema al pasar 5 segundos de haber entrado dicho *loop*.

La interfaz de Zephyr para configurar y utilizar el `WDT` permite la configuración de una cantidad virtualmente infinita de **canales** para tener monitoreados los diferentes hilos concurrentes de un programa. Cada canal virtual se comporta como un `WDT` hardware. La *API* interactúa con el `WDT` hardware para lograr la creación de canales.

Debido a la simplicidad de la `API` definida por Zephyr, no se implementó ninguna capa de abstracción adicional en el proyecto; por lo tanto, los módulos que utilizan el `WDT` interactúan directamente con dicha `API`.

3.2. Sensores

3.2.1. Temperatura

Para la temperatura se utilizaron dos sensores DS18B20 de la empresa Maxim, entre sus características y los motivos de elección se encuentran que funcionan a través de interfaz digital (1-Wire), son de amplia disponibilidad en plaza tanto en su encapsulado TO-92 como en su encapsulado a prueba de agua. Tiene un rango de medición de -55 °C a 125 °C con una precisión de $\pm 0{,}5$ °C (en un rango de temperatura de 10 °C a 85 °C) lo cual es un rango que para las condiciones climatológicas de Uruguay es apropiado.

La interfaz 1-Wire ofrece dos ventajas principales. Primero, al ser un protocolo digital, la longitud del cableado utilizado para medir la temperatura del agua tiene un impacto mínimo en la precisión de la medición, a diferencia de los sensores con salida analógica, que son más sensibles a la atenuación y al ruido inducido por la longitud del cable y las variaciones de temperatura ambiental. Esto se debe a que el protocolo digital cambia la relación señal/ruido (SNR) analógica por un margen de error digital más controlado, lo que permite alcanzar distancias mayores de conexión. Por ejemplo, en instalaciones típicas con cableado adecuado, 1-Wire puede funcionar confiablemente a distancias de hasta 100 metros, mientras que los sensores analógicos suelen degradar su señal notablemente a partir de unos pocos metros. La segunda ventaja ante estos sensores que no cuentan con interfaz digital es que en un mismo bus 1-Wire se pueden colocar varios sensores sin necesidad de ocupar más recursos de la placa y para saber la temperatura de cada uno por separado se necesita tan solo el número serial de 64 bits que es identificador único del sensor (escrito en la memoria `ROM` del dispositivo).

El desarrollo relacionado a los sensores de temperatura se dividió en dos módulos, uno dedicado a la interfaz con el bus 1-Wire (al cuál se lo llamó *onewire*) mientras que el segundo módulo (llamado *temperature*) fue enfocado a la interfaz con los sensores DS18B20.

Para obtener la temperatura de los sensores en el módulo *temperature* se utilizó la `API` *sensor.h* en conjunto con la biblioteca *w1_sensor.h*, ambas de Zephyr. La `API` de *sensor* facilita la comunicación, configuración y manejo de eventos producidos por todo tipo de sensores, inclusive con distinto nivel de complejidad y funcionalidad como ser acelerómetros, giroscopios, sensores de luz, de presión, y, en este caso, de temperatura.

Este nivel de abstracción a la hora de manejar distintos sensores mediante una única `API` se logra gracias a que en Zephyr, un sensor es representado como un objeto de tipo *device* asociado a la interfaz *sensor_driver_api* (ver Sección 3.1.2). Esta interfaz define una estructura estándar que incluye uno o más *channels* (canales), que representan las diferentes magnitudes que un sensor puede medir, propiedades configurables llamadas *attributes* (atributos) que pueden pertenecer a un canal específico o al sensor en general (como la escala, la frecuencia de muestreo, o las unidades), y finalmente los *triggers* (disparadores) que son eventos generados por el sensor y que pueden ser manejados por el microcontrolador.

Para el caso particular del sensor de temperatura DS18B20, puesto a que es un sensor que utiliza el bus 1-Wire, se necesita la ayuda de la biblioteca complementaria *w1_sensor.h*, la cuál compatibiliza la `API` de *sensor* con el manejo de objetos definidos por la `API` de 1-Wire de Zephyr (*w1.h*).

Estas bibliotecas se utilizan en la clase *temperature*, la cual en su constructor, recibe un objeto de tipo *device* del *Devicetree* (ver Figura 3.1) y un número entero sin signo de 64 bits (correspondiente al identificador único del sensor, obtenido durante la configuración o leído desde el *file system*) y realiza las conversiones correspondientes (de atributos y estructuras) para hacer uso de los métodos de la `API` de sensores. Una vez el objeto es instanciado, se pueden utilizar los métodos de la `API` de *sensor* de Zephyr como ser *sensor_sample_fetch* para iniciar una medición, y *sensor_channel_get* para obtener la temperatura medida.

Esta implementación remarca una gran ventaja derivada de la programación orientada a objetos frente a un código basado en un lenguaje como C, la cuál es que se pueden cambiar los sensores en tiempo de ejecución. Esto se logra consiguiendo el número identificador del sensor nuevo, luego destruyendo el objeto de tipo *temperature* original y posteriormente instanciando un nuevo objeto creado a partir del nuevo número identificador.

Lo segundo destacable de esta implementación es la escalabilidad conseguida, ya que si bien actualmente el resto del *firmware* esta pensado para funcionar con sólamente dos sensores, si en un futuro se quisieran agregar más sensores de temperatura sólamente consiguiendo sus números identificadores se pueden instanciar más objetos de tipo *temperature*.

Para conseguir el número identificador de un nuevo sensor se utiliza la clase *onewire* en conjunto con el modo de funcionamiento *CONFIG*, el cuál se detallará más adelante (Subsección 3.5.4). Esta clase toma como entrada el objeto de tipo *device* que tiene la definición del bus 1-Wire (ver Figura 3.1), y luego mediante la función *w1_search_bus* se comanda una búsqueda de todos los *slaves* (esclavos) 1-Wire que hayan en el bus siguiendo el algoritmo propuesto por Analog Devices en su nota de aplicación 187 [2], que se encuentra integrada dentro de la `API` del subsystem de 1-Wire de Zephyr.

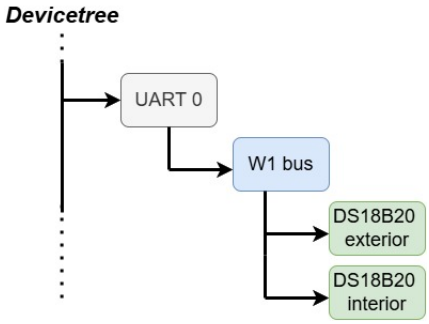


Figura 3.1: Diagrama del *Devicetree* conteniendo la definición de los nodos relacionados a los módulos de temperatura.

La ventaja de utilizar la función *w1_search_bus* es que permite definir una función de *callback* en caso de que se encuentre un *slave* en el bus, esto permitió guardar los números identificadores de los sensores encontrados en una variable de tipo estructura de la siguiente forma:

```
struct SensorROM {
    uint64_t rom[MAX_SENSORS];
    int count = 0;
};
```

En donde la constante `MAX_SENSORS` en esta ocasión fue definido como 2.

Esta variable que contiene los números identificadores (nombrados como *rom* en la estructura mencionada), si bien esta definida en la parte privada de la clase *onewire*, se puede consultar mediante el métodos implementados en la clase. Esto permite (una vez consultado los valores de los números identificadores de los sensores) la creación de los nuevos objetos de *temperature* y también sirve para el almacenamiento de dichos números en el *file system*.

3.2.2. Sensor de conductividad

Como fue mencionado en la sección 2.2.2 se mantuvo el sensor de conductividad *EZO-EC conductivity circuit* del proyecto predecesor (de aquí en adelante se refiere al mismo como `EZO-EC`). Diferentes referencias al código desarrollado durante el proyecto se refieren a este sensor como *sensor de salinidad* ya que el motivo de medir la conductividad es conseguir en última instancia la salinidad del agua.

El `EZO-EC` define su interfaz de configuración, reporte de estado y reporte de datos a través de un protocolo basado en *strings* (cadena de caracteres) sobre una interfaz serial. La especificación de protocolo declara que los comandos pueden ser leídos (operación *poll*) o escritos. El protocolo también define mensajes que determinan el estado del dispositivo y que no están directamente relacionados a los comandos enviados. Entre ellos, *OK* (respuesta a un comando cuando es recibido de forma exitosa), *ERR* (respuesta a un comando cuando es recibido de forma

errónea) y *UV* (mensaje recibido cuando el voltaje de alimentación del sensor está por debajo de los 3,3 V).

El protocolo de comandos puede ser accedido a través de la interfaz serial `UART` o `I2C`. Ambas interfaces comparten los mismos `GPIOs` por lo que no se pueden usar simultáneamente ni paralelamente; la interface `UART` es la usada por defecto. La hoja de datos del `EZO-EC` declara dos formas de conmutar protocolos: una manual y otra a través de comandos. Para intercambiar el protocolo de forma manual se debe alimentar el módulo y cortocircuitar dos de los `GPIOs` desconectando físicamente la interfaz serial con el *host*. Para intercambiar el protocolo a través de comandos, se debe en primer lugar enviar un comando por el protocolo serial inicial y luego aplicar un *reboot* para que la actualización se materialice.

Por más que el uso de la interfaz `I2C` hubiese sido provechoso en términos de aprovechamiento de `GPIOs`, se decidió no utilizarla ya que ambos métodos de transición entre protocolos seriales se consideraron demasiado riesgosos como para ser integrados al sistema. En primer lugar, el intercambio manual entre protocolos implicaría agregar un paso crítico a la instalación de los dispositivos e implicaría tener un mecanismo de verificación previo a la misma. El método manual también es problemático ya que en caso de que el `EZO-EC` sufra un fallo que lo retorne a las configuraciones de fábrica, se necesitaría intervención manual para re-instalar el mismo. El método de intercambio de protocolos usando comandos, implicaría implementar la interfaz de comandos para enviar el comando inicial para luego intercambiar de protocolo serial. Esto implicaría implementar *driver* para ambas interfaces e implementar un mecanismo para re-configurar los pines del *host* a la vez del periférico serial del mismo. En resumen, el uso de la comunicación `UART` por defecto simplifica los requerimientos de desarrollo del *driver*.

La arquitectura del firmware para la abstracción del sensor utiliza una capa de abstracción de la interfaz `UART` implementada mediante la biblioteca `salinidad_phy`, desarrollada dentro del proyecto. Además, cuenta con un conjunto de comandos definidos mediante una adaptación del patrón de diseño *builder* [8], que representa cada comando como un objeto independiente que hereda de la clase `atlas_sci_command`. Todos los comandos deben implementar los siguientes métodos:

1. `get_header()`: Retorna el *string* que define el comando.
2. `handle_reception()`: Maneja el evento en el cual se recibe un comando desde el sensor. En el ejemplo de la respuesta a un comando de *poll*, este método se usa para confirmar el seteo del valor.
3. `handle_event()`: Maneja uno de los eventos declarados por el sensor de conductividad en su especificación del protocolo.

Para cada comando se pueden definir métodos adicionales que devuelvan el comando necesario para realizar una operación específica no generalizable entre todos los comandos. Por ejemplo, para configurar el tipo de *probe* que el sensor tiene conectado, se debe enviar el *string*:

$$K, < tipo >$$

Para esto, en el comando *set_probe* define el método *set_type*, que no es compartido con el resto de comandos definidos. Esta implementación permite la generalización de las funcionalidades comunes de los comandos manteniendo la posibilidad de distinción entre los mismos aplicando sus funcionalidades particulares especificadas por el protocolo.

Retomando la definición de la abstracción del sensor de conductividad, la biblioteca *salinidad* usa la capa física y los comandos para realizar las funcionalidades específicas a la aplicación. La inicialización configura el sensor al *probe* usado y configura el sensor para estar en modo *sleep* con la motivación de ahorrar energía cuando no se está usando. Otro método definido en la interfaz pública permite tomar una medida; lo cual es realizado enviando un comando de *wake up* para tener la medición de conductividad disponible y requiere una medida única. Luego de que la medida es recibida, se envía un comando de *sleep* para volver al estado de reposo.

3.2.3. Medida tensión

El `SiP` utilizado cuenta con un `ADC` interno por lo que no será necesario hacer uso de un módulo externo, simplemente se deberá configurar. El `ADC` es de tipo `SAADC` (Successive Aproximations Analog-to-Digital Converter) [49] con una resolución variable entre 8, 10 y 12 bits (también 14 bits si se utiliza *oversampling*) de 8 canales para medidas *single-ended* (4 para medidas diferenciales), con una ganancia configurable, elección entre referencia interna de voltaje y `VDD`, con la capacidad de comunicar los resultados al `CPU` utilizando la tecnología *EasyDMA* [14] que guarda las medidas directamente en la memoria `RAM` como un arreglo de enteros de 16 bits en complemento a dos (Figura 3.2).

Debido a las características y funcionalidades de este `ADC`, la empresa Nordic provee de una `API` específica para su uso en donde se tiene la opción de aplicar funcionalidades extra como modo de muestreo por timer interno con interrupción o modo ráfaga (en donde se le pasa uno o más *buffers* para que el `ADC` llene de medidas); sin embargo se optó utilizar la `API` estándar de Zephyr ya que contiene las funcionalidades necesarias y cuenta con una interface más simple de utilizar.

En el proyecto se utilizó el `ADC` para medir la tensión en bornes de la batería en conjunto con un divisor resistivo para acondicionar el rango de voltaje de la batería usada al rango aceptable por la entrada al `ADC`. La medida del voltaje de la batería se utilizó para realizar transiciones en la máquina de estados sobre la cual se desarrolla en la sección 3.5.

Se realizaron las estimaciones relacionadas a la batería partiendo de la hoja de datos de la batería Panasonic NCR18650B [25]. El rango de voltaje reportado por la hoja de datos es de $2,5\ V$ totalmente descargada a $4,2\ V$ totalmente cargada. Dado que la tensión mínima para que el dispositivo esté encendido es de $3,2\ V$, se toma el rango de voltaje posible al medir es de $3,2\ V$ a $4,2\ V$. Respecto a la capacidad efectiva de la batería, tomando el rango útil de voltaje nombrado anteriormente, un consumo medio en el orden de las decenas de mA y una temperatura de entre 0 °C a 30 °C, se consideró que la carga efectiva de la batería es de 3200 mAh.

En el caso de este proyecto se configuró el ADC para tomar medidas *single-ended* en el canal 1, utilizando como referencia la interna de $0,6V$, con una resolución de 12 bits y teniendo una ganancia de $\frac{1}{6}$.

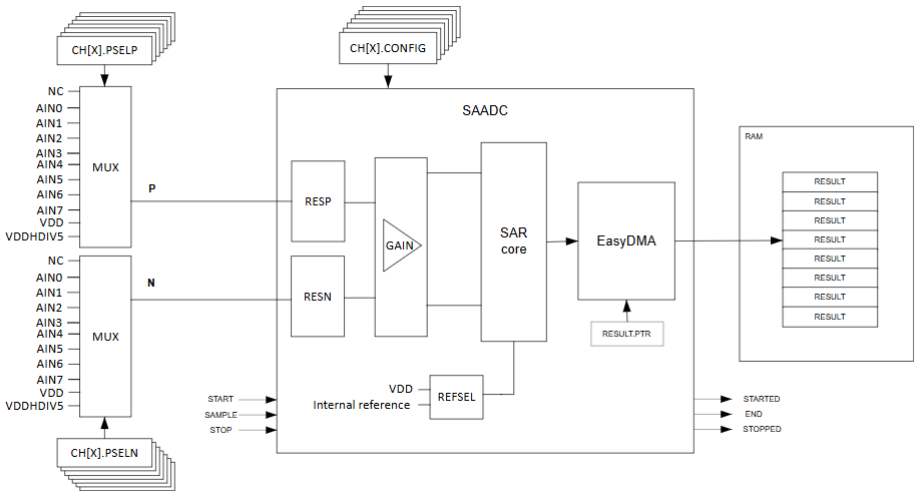


Figura 3.2: Diagrama de bloques del ADC del *development kit*. Tomado de [49].

El rango total de medición $V_{rangeADC}$ está dado por la ganancia y referencia elegidas, que en este caso, $\frac{1}{6}$ y $0,6V$ respectivamente, por lo que el rango de entrada queda definido como $V_{rangeADC} = \frac{0,6V}{gain} = \frac{0,6V}{\frac{1}{6}} = 3,6V$. Por otra parte, dados los 12 bits de resolución, el rango total $V_{rangeADC}$ se divide en niveles discretos de $\frac{3,6V}{2^{12}} = 878,9\mu V$.

Para adaptar la tensión de la batería al rango de entrada, se implementó un divisor resistivo compuesto por resistencias iguales de valor nominal 100 $k\Omega$ (ver Figura 3.3). Esta configuración divide por dos la tensión de la batería, lo que implica que el nodo intermedio de medición alcanza un máximo de $2,1$ V cuando la batería está totalmente cargada y un mínimo de $1,6$ V cuando la batería se encuentra en el mínimo umbral de funcionamiento de la placa, cumpliendo con el rango de entrada calculado anteriormente.

El valor medido de la batería posteriormente será utilizado para definir los diferentes modos de funcionamiento del sistema, de acuerdo a lo explicado en la Subsección 3.5.4.

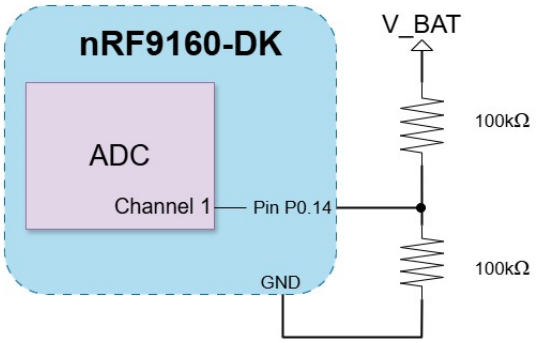


Figura 3.3: Diagrama de conexión del `ADC` del *development kit* con el divisor resistivo.

3.3. Implementación y configuración del file system

3.3.1. Hardware usado y configuración en Zephyr

El almacenamiento temporal de las mediciones se realizará en la memoria flash interna de la nRF9160-DK, una MX25R6435F [1] de la empresa Macronix, soldada directamente en la placa. Esta memoria, con una capacidad de 64 Mb, es lo suficientemente grande para almacenar una cantidad extensa de datos. Se utilizará para optimizar las transmisiones, guardando temporalmente las mediciones antes de transmitirlas, lo que contribuye a reducir el consumo energético del sistema.

En esta ocasión se utilizó el *driver* de `SPI` de Zephyr ya que la memoria utiliza este protocolo para su comunicación, mientras que también se habilitaron los *drivers* específicos para el trato con memorias flash de tipo NOR (*SPI_NOR*). A su vez, se configuraron otros ajustes que permitirán el uso de un *file system* (como *LittleFS*), y se separó la parte de memoria de datos de la del *bootloader* del cuál se hablará mas adelante (ver Sección 3.4.4).

3.3.2. Introducción a LittleFS

Para el *file system* se optó por `LittleFS` [59] debido a sus ventajas frente a otros sistemas de manejo de memoria no volátil. Su principal fortaleza frente a sistemas similares es que está especialmente diseñado y optimizado para microcontroladores de escasos recursos computacionales y de bajo consumo de energía, lo que lo hace más eficiente que soluciones similares.

LittleFS se basa en la división de la memoria en bloques de tamaño fijo y en el uso de un sistema de *logs* para almacenar los metadatos del sistema manejador de archivos. Estos metadatos contienen información sobre los archivos, como su ubicación en memoria física, tamaño y nombre, entre otros aspectos, pero no incluye el contenido de los archivos en sí.

Lo que sucede cuando se modifica un archivo en `LittleFS` es que no sobreescribe directamente los bloques que estaban en uso, sino que escribe la nueva versión del archivo en bloques libres (de la memoria física). Este mecanismo evita escrituras repetidas sobre los mismos sectores, lo cual es fundamental para prolongar la

vida útil de la flash, al distribuir uniformemente las escrituras.

Otra gran ventaja y la principal razón para elegir `LittleFS` se debe a que operaciones como las escrituras son atómicas, éstas se realizan en un sistema llamado *copy-on-write* lo que hace al sistema casi inmune a corrupciones por pérdidas energéticas, un caso de uso ideal para el proyecto actual. Esto es debido a que los metadatos no son modificados a menos que la operación realizada sea exitosa (ver Figura 3.4).

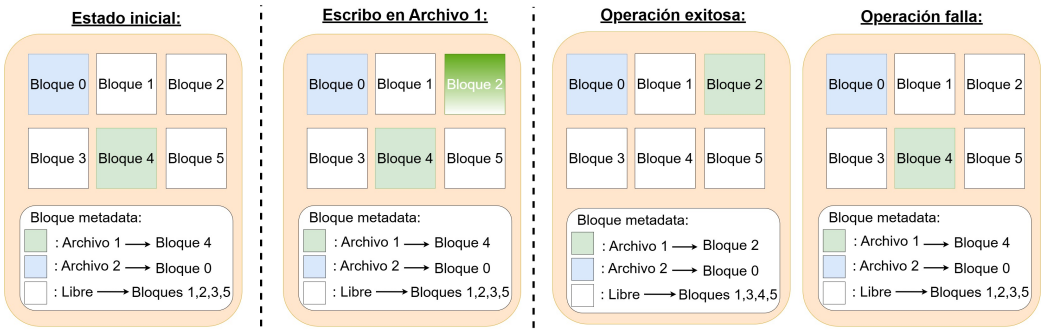


Figura 3.4: Esquema abstracto de funcionamiento de *copy-on-write* de *LittleFS*. Al modificarse el archivo 1 se comienza a escribir en un nuevo bloque de memoria física, en caso de que sea exitosa la escritura, se actualizan los metadatos marcando la nueva ubicación de el archivo 1, y en caso que no sea exitosa la escritura, simplemente no se actualizan los metadatos por lo que el bloque 2 seguirá marcado como libre.

Gracias a esto, en caso de pérdida en la alimentación de la placa el sistema no se corrompe y queda en un estado estable. A lo sumo perdiendo la información que se iba a guardar al momento de la pérdida pero no corrompiendo los datos que ya fueron escritos.

3.3.3. Implementación en el proyecto

La implementación del sistema de almacenamiento de datos se da por la necesidad de tener almacenamiento temporal para conservar energía y no estar transmitiendo contínuamente a medida que se tome una medida. Esto resultaría ineficiente energéticamente sobre todo en casos de que no haya conexión celular. Tener la capacidad de almacenar datos para posteriormente transmitirlos desliga la frecuencia de muestreo de la frecuencia de transmisión, mejorando la eficiencia del sistema y su robustez.

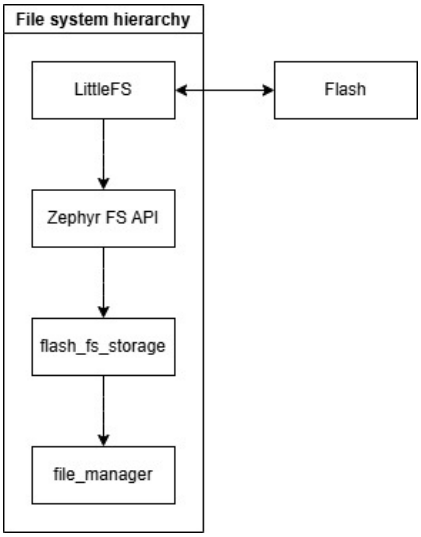


Figura 3.5: Esquema de jerarquías del File System

La interacción con `LittleFS` se da utilizando dos intermediarios, el primero es la `API` de *file system* de Zephyr [64] y el segundo es una clase llamada *flash_fs_storage*.

La `API` de *virtual file system* de Zephyr permite definir uno o varios puntos de montaje para un *file systems* como `LittleFS` o `FATFs`. Esta `API` permite la operación del *file system* montado, mediante comandos generales como lo son **fs_mount**, **fs_read**, **fs_write**, **fs_seek**, entre otros, de manera básica, debiendo especificar no solo punto de montaje al cuál se le realiza la acción sino también la cantidad de bytes a escribir o leer, además de la posición en donde se realizan esas escrituras o lecturas.

Para simplificar el proceso de lectura y escritura es que se implementó la clase **flash_fs_storage** que agrega un extra a las funciones propias de `LittleFS` como por ejemplo funciones que devuelven el tamaño en bytes de un archivo, o formatean el *file system*. Esta clase también tiene definida en su interfaz privada el punto de montaje para que sea de uso exclusivo dentro de la clase.

Luego, para simplificar el almacenamiento de datos tomados se hizo uso de una estructura llamada *SensorData* la cuál se encuentra detallada a continuación:

```
struct SensorData {
    int32_t temp1;            // Temperatura del sensor 1
    int32_t temp2;            // Temperatura del sensor 2
    int32_t battery_voltage;  // Voltaje de la bateria en mV
    float   salinidad;        // Medida de salinidad/conductividad
    int64_t timestamp_unix;   // Timestamp en formato unix
};
```

Al ser una estructura se tiene una misma cantidad fija de bytes (24 para ser exactos) por cada medida que se tome lo que facilita el indexado al momento de que se guardan en los archivos, es decir conociendo el tamaño de los mismos se puede fácilmente calcular el número de medidas guardadas en el mismo. Es gracias a esta forma particular de guardar datos que el intermediario cobra sentido ya que hay mucha funcionalidad repetida en la clase *flash_fs_storage* que Zephyr brinda en su `API` como ser funciones estilo *fs_mkdir* (para crear directorios) o *fs_unlink* (para borrar directorios y/o archivos), sin embargo los métodos de lectura y escritura son propias de la clase. Estos métodos son basados en índices, dados por la cantidad de bytes que existen en un archivo los cuales siempre serán un múltiplo de 24 bytes, por lo que si un archivo tiene 240 bytes de tamaño se sabe que tiene 10 medidas guardadas.

Para llamar a estos métodos primeramente el `LittleFS` deberá estar montado, para ello se hizo uso de la función de la `API` de Zephyr *fs_mount* durante la inicialización del sistema. Por otra parte una vez se cuente con un directorio válido y su path correspondiente se podrá comenzar a escribir en un archivo los datos.

Hay dos modos de escritura dependiendo del índice con el que se llame al método *flash_fs_write_data*, en caso de que el parámetro de entrada *index* sea -1 se estará usando la modalidad *append* la cuál simplemente agrega el dato al final del archivo que fue introducido con el parámetro de entrada *fname* (nombre del archivo).
El siguiente caso de uso es cuando se introduce un número en el parámetro *index*. Este número es verificado que no sea mayor al tamaño máximo de medidas por archivo denotado por la constante *FILE_AMOUNT_MEASURES* y luego se procederá a escribir en la posición indicada dado el archivo. También se deberá especificar el tamaño en bytes de lo que se piensa escribir con el parámetro de entrada *data_size_bytes*.

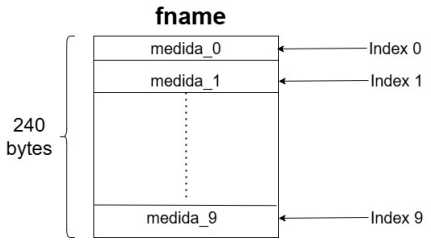


Figura 3.6: Ejemplo de un archivo de datos (llamado **fname**) de tamaño 10 medidas.

Es importante recalcar que la previa existencia o no del archivo a escribir no es importante puesto a que el mismo será creado gracias a los parámetros de la función *fs_open* en conjunto con la flag *FS_O_CREATE*, sin embargo el directorio donde se ubica el archivo mandatoriamente deberá ser creado previamente.

Un segundo tipo de dato es almacenado en un path aparte de los datos recopilados a modo de registro auxiliar para atender el caso de una pérdida energética. Esta estructura auxiliar guarda en qué archivo y en cuál índice de archivo se estaba

escribiendo al momento y también el archivo que se estaba mandando y el índice del último dato enviado del archivo.

La estructura está compuesta de la siguiente manera:

```
struct AuxFields{
    uint64_t  exterior_sensor;          // Identificador del sensor de
                                            temperatura exterior
    uint64_t  interior_sensor;          // Identificador del sensor de
                                            emperatura interior
    uint32_t  index_file_being_written; // Índice del archivo
                                            que se está escribiendo
    uint32_t  index_file_being_sent;    // Indice del archivo
                                            que está siendo mandado
    uint32_t  file_being_written;       // Número de archivo
                                            que se está escribiendo
    uint32_t  file_being_sent;          // Número de archivo
                                            que está siendo mandado
    float     fs_version;               // Versión del file system
};
```

Esta estructura atiende las necesidades ante una pérdida energética, permitiendo que el sistema pueda continuar escribiendo y mandando donde retomando desde su último estado previo al corte energético. Para hacer uso de esta estructura auxiliar, se implementó una segunda capa de abstracción a modo de facilitar el uso de este sistema desde el *main* del proyecto mediante la implementación de otra clase auxiliar llamada *file_manager*.

Esta clase instancia el objeto *flash_fs_storage*, haciéndose única intermediaria entre el *file system* y el *main*. Implementa métodos que tratan con la actualización del archivo auxiliar y además el manejo de los índices para mantener al día la lectura y escritura con la transmisión de datos.

Esta clase también tiene definidos los *paths* que se utilizarán para el almacenamiento de datos y el prefijo por defecto que llevarán los archivos, por defecto el *path* para guardar los datos se eligió como */lfs/sensor_data/* mientras que el prefijo será */lfs/sensor_data/file_**X**.dat* de donde la **X** infiere que el número de archivo el cuál se deberá definir también para dar lugar a la rotación de archivos. La rotación de archivos implica la sobreescritura de datos si se realiza una vuelta entera al sistema de datos, por ejemplo si tengo 3 archivos de 10 medidas cada uno, la medida numero 31 reemplazará la primer medida tomada (ver Figura 3.7), por lo que es importante la definición de tamaño de archivo y número de medidas para garantizar que no se sobrescriban medidas que todavía no fueron enviadas.

Para este proyecto se configuró por defecto el *file_manager* con 3 archivos de 1440 medidas cada uno, lo que representa 3 días tomando medidas cada 1 minuto. Esto es para alinearse con el objetivo planteado de una autonomía del sistema de 72 horas, sin embargo estos 3 archivos solamente ocupan 103.68kB de los 8MB de flash disponible, siendo el valor máximo teórico de medidas que se pueden registrar $\frac{8MB}{24B} \approx 333$ mil medidas.

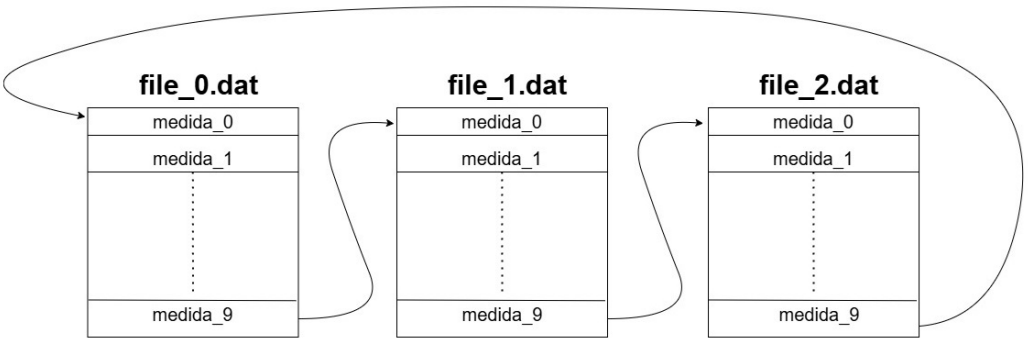


Figura 3.7: Ejemplo de rotación con 3 archivos de 10 medidas cada uno

Luego de cada escritura, el archivo auxiliar será actualizado automáticamente por el *file_manager* con los nuevos índices y/o número de archivo (en caso que corresponda). En cambio, en el caso de una lectura se deberá hacer uso de una función adicional para habilitar la actualización del archivo auxiliar. Esto es debido a que pueden ocurrir situaciones en donde se realiza la lectura correctamente pero debido a fallos en la conexión los datos no llegan a ser recibidos exitosamente. Por esta razón, es fundamental esperar la confirmación de recepción exitosa de todos los mensajes antes de modificar los índices. De lo contrario, si los índices se actualizaran antes de esta confirmación, los datos podrían considerarse enviados y nunca se intentaría el reenvío de los mismos.

Para solventar eso se implementó un método extra en la clase llamado *commit_read* en donde el módulo consumidor de datos (en caso de que los datos hayan sido transmitidos de manera exitosa) puede confirmarle al *file_manager* que es momento de actualizar el índice de datos enviados.

La gran desventaja de esta implementación de archivo auxiliar es que al ser una estructura lo que se escribe, obligatoriamente para cualquier actualización se deberá escribir los (en este caso) 40 bytes de la estructura `Aux_Fields` completos. Esto es ineficiente para escrituras singulares de medidas puesto a que la escritura de la medida son simplemente 24 bytes pero la actualización del archivo auxiliar son 40 extra, es decir 64 bytes en total.

Una posible solución a implementar en un futuro para mitigar este problema consiste en prescindir de la estructura `Aux_Fields` y en su lugar implementar métodos específicos de lectura y escritura para cada índice de archivo de forma independiente. Por ejemplo, en caso de actualizar únicamente el índice `file_being_sent`, se modificarían exclusivamente los 4 bytes correspondientes a dicho campo, evitando así la escritura innecesaria del resto de la estructura. De manera complementaria (y potencialmente en un archivo separado), los campos de menor frecuencia de actualización, como la versión del *file system* o los números identificadores de los sensores de temperatura (de aquí en adelante llamados `ROM`), podrían gestionarse mediante métodos de escritura específicos, distintos de los utilizados para los índices.

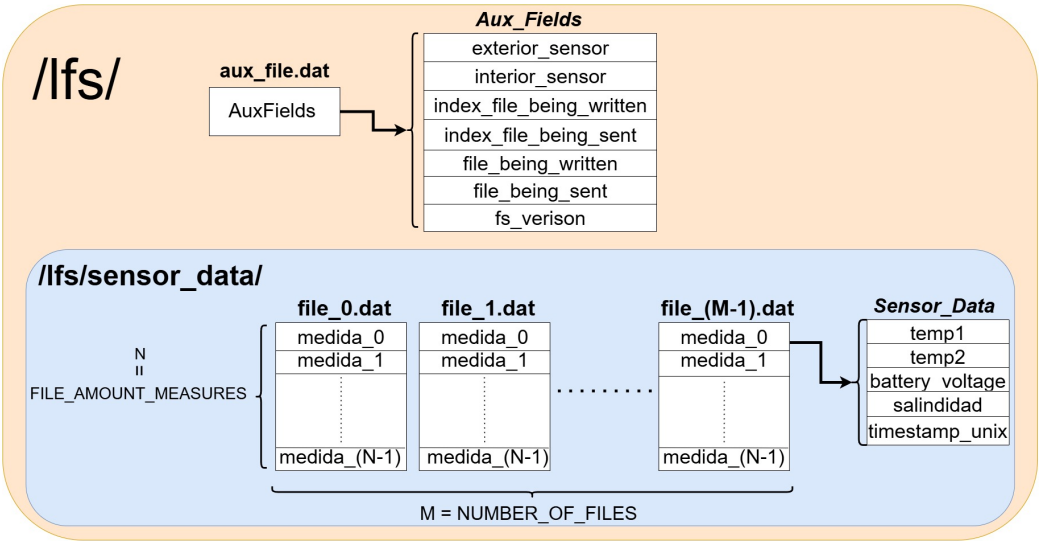


Figura 3.8: Diagrama de archivos en *file_manager* con los paths por defecto del proyecto.

3.4. Arquitectura de red y comunicaciones

A continuación se especifica el proceso de desarrollo de las diferentes bibliotecas que comprenden las comunicaciones en la aplicación.

3.4.1. Set up de capa física

Se implementó una biblioteca para configurar el módem, conectándolo a la red y recuperar la conexión en casos en que la misma se pierda. Esta clase es la responsable de llevar a cabo las configuraciones específicas de la aplicación respecto a la capa celular. En términos de diseño, la clase es *singleton* [50] ya que representa el único módem al cual tiene acceso el *hardware*.

Para entender la configuración del módem, se debe entender dos características negociables de la red celular.

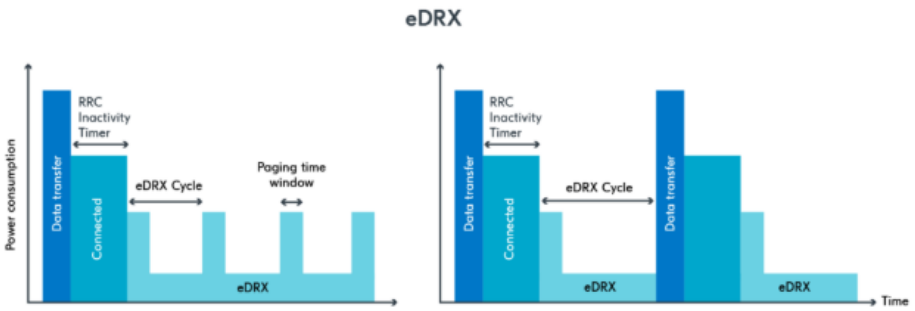


Figura 3.9: Consumo de corriente en un dispositivo conectado a la red celular configurado para eDRX. Fuente: [42].

El `eDRX` (extended delayed reception) es un parámetro negociado con la red celular que determina el retardo máximo de los mensajes *downstream* (de la red al dispositivo usuario). La mantención del link depende de la coordinación de ventanas de paginación (`PTW` o *paging time window* en la figura 3.9) para que los mensajes arriben al dispositivo. Entre ventanas de paginación, la red debe retener los mensajes *downstream* para que la comunicación se dé de forma exitosa. El `eDRX` define el período de tiempo entre períodos de paginación. La característica `eDRX` es importante para reducir el consumo de batería ya que entre períodos de paginación, el modem celular puede apagar su radio y antena disminuyendo el ciclo de trabajo de los mismos y así reduciendo la corriente media consumida. En términos de ahorro de batería, un mayor período de `eDRX` resulta en un menor consumo de batería bajo el riesgo de aumentar significativamente la latencia de recepción de datos.

Al necesitar coordinar los períodos donde el nodo entra en paginación, la red soporta una lista dada de posibles períodos de `eDRX` y `PTW`. Estos períodos se negocian en el momento de conexión entre el nodo y el proveedor de servicio. En el momento en que un nodo solicita unirse a la red, en el mensaje enviado requiere un período de `eDRX` y `PTW` que es aceptado o no por el proveedor. Los valores aceptados por la red de Antel se muestran en la Tabla 3.1.

eDRX (s)	**PTW** (s)
20,48	1,28
40,96	2,56
81,92	3,84
163,84	5,12
327,68	6,4
655,36	7,68
1310,72	8,96
2621,44	10,24
5242,88	11,52
10485,76	12,8
	14,08
	15,36
	16,64
	17,92
	19,20
	20,48

Tabla 3.1: Valores disponibles de configuración de conexión celular para la red de Antel. Extraído de especificaciones técnicas de la implementación (ver Apéndice A).

El `PTW` (*Paging Time Window*) es el período a través del cual el dispositivo es capaz de recibir datos desde la red sin encender la radio en su máxima capacidad. Si es usado en simultáneo a `eDRX`, el período `PTW` determina el ciclo de trabajo

de recepción siendo el tiempo durante el cual el nodo es alcanzable por la red inmediatamente. Un PTW más bajo reduce el consumo de batería bajo el riesgo de disminuir el *throughput* de datos *downstream*.

La red celular tiene definido el modo PSM (Power Saving Mode) similar al mecanismo eDRX. Al entrar en PSM, un dispositivo permanece registrado en la red aunque no sea alcanzable por las radiobase; esto es, que el nodo apaga su radio, antena y módem hasta el siguiente período de conexión en donde entre en paginación. Mientras un dispositivo está en PSM, a diferencia de eDRX, no recibe mensajes ni se retienen por la red hasta su próxima paginación. Este modo de ahorro de energía es ideal para sensores distribuidos con comunicaciones solamente *upstream*. Un período alto en PSM se traduce a una reducción drástica en el consumo de batería aunque se traduce también a la existencia de períodos en los cuales no se alcanza a enviar información al mismo.

En un principio, la capa celular de la aplicación se inicializó sin configuración de eDRX ni PSM. Tras las primeras pruebas de consumo del dispositivo, se observó que mantener la conexión celular era la mayor fuente de consumo por lo que se optó por optimizar el consumo relacionado a la misma. Debido a que la aplicación necesita de una capacidad de respuesta asíncrona a datos *downstream*, se descartó en primera instancia la posibilidad de usar PSM, por lo que se optó por configurar la conexión para usar el modo eDRX.

Se estableció como requerimiento que el mayor tiempo entre el envío de un comando y la recepción del mismo por el dispositivo debía ser de 1 minuto. A partir de los requerimientos se eligieron los valores más ceñidos al cumplimiento de los mismos utilizando los valores disponibles por la red desplegados en la tabla 3.1; resultando en un eDRX de 40,96 segundos y un PTW de 2,56 segundos.

La configuración anteriormente mencionada resulta en una velocidad de respuesta de mensajes *downstream* de un retardo máximo de 40,96 segundos teóricamente. El impacto en el consumo es detallado en el Sección 5.2.3.

3.4.2. MQTT

MQTT es un protocolo de comunicación de capa de aplicación sobre TCP con un esquema de publicador-subscriptor en el cual entidades publican o se suscriben a tópicos y sub-tópicos especificados por un *string*. El ruteo de mensajes entre publicadores y subscriptores es manejado por un servidor llamado *broker*. Un tópico es similar a un canal virtual de comunicación; todos aquellos clientes que estén subscritos a un tópico determinado, recibirán el mensaje que otro cliente envíe al mismo. El *broker* retiene un listado de los tópicos a los cuales está subscrito cada cliente y en caso de que otro cliente publique a un tópico donde hay clientes subscritos, replica el mensaje para el resto de subscriptores. [22]

El protocolo MQTT tiene disponible el uso de publicación y subscripción bajo diferentes calidades de servicio (QoS por sus siglas en inglés). Los diferentes niveles de calidad de servicio indican si el *broker* fue capaz de enviar el mensaje a todos los suscriptores del tópico a donde el mensaje fue publicado. Dichos niveles de calidad de servicio se dividen en 3 niveles explicados a continuación.

En primer lugar, se tiene QoS 0, también referido como *at most once*. Para esta calidad de servicio, el publicador envía un dato al *broker* y no se realiza ningún *handshake* a nivel capa de aplicación para validar que el mensaje haya sido recibido correctamente por los subscriptores. En caso de que un subscriptor esté fuera de alcance al ser enviado este mensaje, el mismo no será re-enviado a dicho subscriptor. En segundo lugar, QoS 1, también referido como *at least once* es un nivel de calidad de servicio que implementa un *hanshake* simple de reconocimiento de envío de mensaje desde los subscriptores al publicador, lo que asegura que el mensaje el entregado a los subscriptores. Debido a la simplicidad del *handshake*, en caso de que un reconocimiento de recepción no sea entregado al publicador por ejemplo, el mismo puede re-enviar el mensaje siendo recibido de múltiples veces por los subscriptores; por lo que con QoS 1 se asegura que el mensaje llega **al menos** una vez. Esta calidad de servicio permite asegurar que los mensajes son recibidos por los subscriptores bajo el precio de un mayor *overhead* en procesamiento por parte del publicador. Finalmente, se define QoS 2, también referido como *exactly once* el cual implementa un *handshake* complejo entre publicador y subscriptor para asegurar que el subscriptor recibe un mensaje exactamente una vez. Ya que esta calidad de servicio requiere de transacciones entre ambas partes, requiere de una lógica más compleja de ambos lados y consume mayor cantidad de recursos computacionales generando el mayor *overhead* de las calidades de servicio disponible. En resumen, una mayor calidad de servicio asegura una mayor confiabilidad en términos de recepción de mensajes

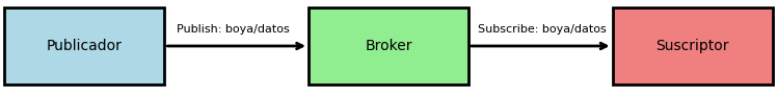


Figura 3.10: Esquema de conexión MQTT

Para el proyecto se utilizó la conexión MQTT para transmitir los datos medidos y enviar comandos al dispositivo. Es de interés que el dispositivo sea capaz de confirmar que las medidas enviadas hayan sido recibidas por el servidor. Esto implica que las medidas deben ser enviadas con QoS 1 o QoS 2. Debido a que se asume que el servidor es capaz de descartar mensajes repetidos, se optó por el uso de QoS 1 para el envío de datos. Esto permite tener la posibilidad de reenviar los datos en caso de no haber sido recibidos por el servidor sin tener el *overhead* que supone QoS 2.

Respecto al envío de comandos, es de interés no recibirlos más de una vez ya que sería necesaria la implementación de lógica para evitar condiciones de carrera en ejecución que podrían llegar a inestabilidades en el *firmware*. Para lograr lo anteriormente planteado, se tiene disponible usar `QoS` 0 o `QoS` 2. Ya que se supone que los comandos son enviados por un operario que es capaz de analizar si el mensaje es recibido por el dispositivo o no, se optó por utilizar `QoS` 0 ya que supone el algoritmo más simple posible para recibir los comandos. Para evitar la posibilidad de que los *logs* que envía el dispositivo en respuesta a los comandos no se envíen resultando en el operario re-enviando el comando y asumiendo que no es problemático recibir un mismo *log* múltiples veces se usó `QoS 1` para los mismos.

Para el diseño y desarrollo del proyecto se creó un *broker* `MQTT` sobre `TLS` utilizando el servicio de `EMQX` [15]. Esta plataforma permite la implementación de *brokers* específicos de aplicación, otorgando una plataforma a través de la cual se determinan los usuarios con acceso al *broker* y donde se puede consultar el uso del mismo para *debugging*.

La implementación del módulo `MQTT` estuvo basado en la biblioteca nativa de Zephyr para conexiones `MQTT`, manejada por una clase de tipo *singleton* la cual abstrae la conexión `MQTT` y el manejo de publicación y subscripción a los tópicos a las necesidades específicas de la aplicación: publicar medidas tomadas por el dispositivo, descifrar los comandos recibidos por el tópico correspondiente y enviar *logs* de traza de la aplicación.

Para el manejo de comandos recibidos se implementó un módulo separado de tipo *singleton* y *observer* [30], el cual permite a diferentes secciones del código definir *handlers* para diferentes comandos. El módulo también tiene expuesta una interfaz de *parsing* de comandos. Este módulo permite desligar la capa de transporte de los comandos; en este caso `MQTT`. Esto permite utilizarla también con interfaces seriales o con otros protocolos inalámbricos manteniendo el funcionamiento de los comandos, los *handlers* y la interface a la aplicación.

Al recibir un mensaje `MQTT`, se invoca el *parsing* de comandos del módulo, quien luego define qué comando fue recibido e invoca el *handler* correspondiente para el comando recibido.

En contraparte, el envío de las medidas tomadas por el módulo se realiza a nivel del hilo de manejo de envío de datos. Este hilo periódicamente envía mensajes comunicando las medidas tomadas. El período de envío de datos depende del modo del dispositivo. En modo de bajo consumo, se envía con períodos más largos de tiempo para ahorrar batería. En modo de batería crítica se evita el envío de datos. Los períodos de envío de datos se desarrollan en la sección 3.5.

3.4.3. Time management

Zephyr tiene un manejo de tiempo llamado *Zephyr Time* [65] el cual está basado en un contador de *ticks* que inicia en cero cuando el sistema comienza a funcionar. Si bien esta implementación es esperable en este tipo de sistemas, su utilidad está pensada para manejo de tiempos relacionados a actividades en el contexto del funcionamiento del firmware en sí, como por ejemplo utilizando la

función **k_sleep(1000)**, en la cual se le indica al *kernel* que debe poner a dormir un *thread* por 1000 *ticks* (por defecto los *ticks* se configuran para que tengan una duración aproximada de un milisegundo).
Sin embargo la utilidad que se le va a dar al sistema de recolección de datos requiere la obtención de tiempo en formato fecha y hora es decir no relativo al funcionamiento del sistema, por lo que se pedirá la hora actual aprovechando el hecho de que la transmisión de datos requiere de conexión a internet activa.

Nordic provee una `API` (*date_time* [11]) específica para facilitar la actualización de la hora actual mediante diversos métodos como ser peticiones al módem mediante comandos AT o a un servidor `NTP` (una vez establecida la conexión LTE), también admite actualizaciones de hora de un *Real Time Clock* externo.

Para este proyecto se optó por no contar con un *Real Time Clock* externo ya que se pretende que la placa tenga conexión el mayor tiempo posible. Debido a que el proyecto no necesita una precisión temporal mayor al orden de los 10 minutos y que la precisión de la hora obtenida a través de la red es órdenes de magnitud menor (precisión por debajo de 1 segundo) no se incluyó un estudio respecto a la precisión temporal.

La obtención de la fecha y hora se hará mediante una petición al módem celular, y en los momentos donde la conexión a internet del módem se pierda el tiempo se contará mediante el *Real Time Counter* [48], el cuál es el periférico que la placa Nordic utiliza por defecto para contar *ticks* (y por consiguiente, el tiempo). Ya que el *kernel* de Zephyr es *tickless* (es decir no está basado en interrupciones periódicas sino que está basado en eventos programados), cuando el sistema se va a modo de bajo consumo, Zephyr le programa al *Real Time Counter* el número de *ticks* correspondientes al próximo evento relevante (por ejemplo el próximo muestreo de los sensores o próxima paginación de la red celular). Una vez el *Real Time Counter* alcanza el valor programado, le hace una interrupción al microcontrolador el cual se despierta del modo de bajo consumo y ejecuta las tareas correspondientes al evento. Antes de volver al modo de bajo consumo, el *kernel* actualiza la hora actual del sistema sumándole a la última hora guardada el tiempo transcurrido correspondiente a los *ticks* medidos por el *Real Time Counter*, el cuál es programado de vuelta con el número de *ticks* del siguiente evento relevante. De este modo, el sistema puede mantenerse en reposo con el contador en funcionamiento, minimizando el consumo energético sin perder el seguimiento del tiempo.

Pasando a la implementación concreta del manejo del tiempo en el proyecto, y dado que las peticiones de hora al módem son asincrónicas y no inmediatas, la `API` **date_time** está basada en eventos. Por esta razón, fue necesario implementar un módulo adicional y registrar un *handler* encargado de gestionar dichos eventos, más precisamente, cuando se obtiene una hora válida el *handler* se ocupa de cargarla inmediatamente utilizando la función de la `API` *date_time_set* al módulo de tiempo de Zephyr para que este pueda continuar contando el tiempo internamente.

Por otra parte, se desarrolló una función para forzar la petición de hora al módem de forma asíncrona que se utiliza durante la inicialización del sistema como

forma de asegurar la obtención de una hora válida luego de conseguida la conexión celular. Esta hora conseguida tiene un período de confiabilidad (de 60 minutos por defecto) en donde se considera lo suficientemente fiable como para no tener que realizar otra petición asíncrona al módem. Pasado este período de tiempo se deberá consultar al módem nuevamente para evitar el desfasaje entre la hora que lleva el módulo de tiempo de Zephyr (el cuál está sujeto a la precisión del *Real Time Counter* y por consiguiente el oscilador RC [37] que tiene como entrada), comparado con la hora real provista por la torre celular. El motivo detrás de esto es no realizar consultas innecesarias ya que estas requieren un tiempo considerable para realizarse.

Cuando se carga la hora válida al módulo de tiempo de Zephyr utilizando *date_time_set*, una vez pasado el período de validez de una hora y en caso de que no haya conexión a internet activa condicionando a que el módem no pueda devolver una hora válida, la hora que se devolverá será la que proviene del módulo de tiempo.

Ante una pérdida energética se deberá re-inicializar todo el sistema, realizar una petición al módem y cargarlo en el módulo de tiempo de Zephyr. El único caso donde la hora que se reciba del módulo de *time_management* sea incorrecta será cuando la placa se inicialice luego de una pérdida total energética y no se pueda establecer correctamente la conexión, por lo que nunca se va a poder recibir y cargar correctamente la hora en el *kernel* de Zephyr.
Si bien el sistema no requiere una precisión absoluta de la hora, se estima que para el propósito de timestamping de datos y sincronización básica, una precisión del orden de uno o dos minutos es aceptable. Durante el funcionamiento normal, cuando la conexión celular está activa, la hora obtenida del módem proporciona una precisión mejor a ± 1 segundo. En los períodos sin conexión, el tiempo se mantiene mediante el Real Time Counter utilizando el oscilador RC interno, el cual tiene una precisión de aproximadamente $\pm 500 ppm$, lo que puede implicar una desviación máxima de ± 3 minutos por día. En caso de reinicio sin conexión a internet, no se puede garantizar ninguna precisión hasta obtener una nueva hora válida. Por esta razón, se implementa un valor de error conocido (1969-12-31 23:59:59) que permite detectar estos eventos. Cabe aclarar que la hora que se solicita siempre es la hora local por lo que no hay que hacer conversiones de tiempo, simplemente cambiar su formato.

Además se proveyó un método para devolver la hora en formato *string* legible (AAAA-MM-DD HH:MM:SS) puesto que las funciones de la `API` de Nordic devuelven el tiempo en milisegundos en una estructura *tm* (de C), este paso a formato AAAA-MM-DD HH:MM:SS se hace mediante la función de C *gmtime_r* la cuál tomando de entrada devuelve una estructura compuesta por segundos, minutos, horas, días, meses y años que luego son fácilmente pasados a un *string* mediante la función *snprintf()*. Este método descrito es sumamente útil a la hora de *debugguear*, ya que permite imprimir en pantalla la hora y fecha que tiene la placa, lo cual hace más fácil detectar errores del módulo, por ejemplo, en caso que la petición de la hora no se haya realizado con éxito, debido a la arquitectura del código realizado,

se mandará una fecha y hora de error marcada como $1969 - 12 - 31\ 23 : 59 : 59$ que es la hora que se obtiene si se pasa por las funciones de C el entero -1. Este último ejemplo se dará solamente cuando no haya ninguna conexión establecida.

3.4.4. DFU

DFU (*Device firmware update*) refiere al proceso de actualizar la imágen de aplicación del *firmware* de un dispositivo desplegado sin la necesidad de re-programar por completo la memoria del microcontrolador. Las aplicaciones con capacidades de realizar `DFU` permiten el desarrollo continuo de las mismas, suponiendo una mayor seguridad frente a errores de *firmware*, los cuales al ser detectados pueden ser solucionados y desplegados a los dispositivos sin necesidad de un programador ni de un operario. Un término ampliamente usado para procesos de `DFU` inalámbricos es "Firmware por el aire" (`FOTA`).

El dispositivo fue programado para tener la capacidad de realizar procesos de `FOTA` a través de `HTTP` sobre la red celular. Para esto se usaron bibliotecas nativas de Zephyr para actualizaciones a través de servicios de *buckets* de `AWS` (*Amazon Web Services*) del lado del *firmware* y el servicio de `AWS` *Bucket S3* por el lado del servidor `HTTP`, el cual se desarrolla en la sección 4.1.

Para entender el proceso de actualización de *firmware* de un dispositivo, se debe entender primero los diferentes pasos implicados y los requerimientos de la aplicación para poder efectuarla.

Requerimientos:

Para que un dispositivo sea capaz de realizar actualizaciones en tiempo de ejecución, es necesario tener un transporte para el binario de actualización, que el dispositivo sea capaz de escribir su propia memoria flash y que se tenga particionada la memoria en al menos 3 sectores en los cuales, uno debe ser ocupado con un *bootloader* con conocimiento de las particiones de memoria, otro debe ser ocupado por la aplicación actual y el último debe ser un sector libre de memoria en dónde descargar el binario de actualización.

El *bootloader* utilizado para el proyecto es *MCUBoot* [60]; el cual es un *bootloader* de código abierto y soportado nativamente por el entorno de desarrollo de Zephyr. *MCUBoot* es compilado con los puntos de entrada a la partición primaria de código, donde es ejecutada normalmente la aplicación y el punto de entrada a la partición secundaria que es donde se descarga el binario de actualización. *MCUBoot* usa como interfaz con la aplicación banderas en flash, las cuales indican qué partición ejecutar al comenzar la ejecución de la imágen de *firmware* y si es necesario ejecutar un *memory swap*, lo cual será explicado posteriormente junto al proceso de una actualización de *firmware*. *MCUBoot* permite las actualizaciones seguras, verificación de imágenes a partir de firmas, control de reversiones y *testing* de nuevas imágenes previo a hacer una actualización definitiva. En la Figura 3.11 se muestra un esquema simple de las diferentes particiones recientemente nombradas.

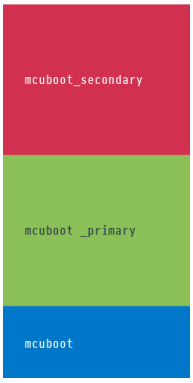


Figura 3.11: Esquema de particiones de una aplicación con MCUBoot como bootloader y capacidades de FOTA. Tomado de la lección 9 del curso intermedio de Nordic Dev Academy [20].

Procedimiento de FOTA:

Para realizar una actualización `FOTA` es necesario que la aplicación comience el proceso de descarga a través del *stack* de comunicación elegido y que lo posicione ordenadamente en la partición de memoria secundaria. En la Figura 3.12 se muestra un esquema simple del proceso de descarga.

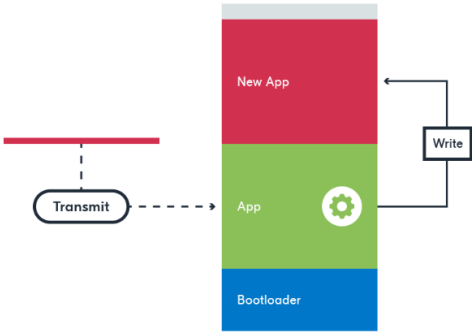


Figura 3.12: Esquema de proceso de descarga de imagen de actualización. Tomado de la lección 9 del curso intermedio de Nordic Dev Academy [20].

Luego de completada descargada la nueva imágen de *firmware*, se indica a *MCUBoot* escribiendo en sus banderas de inicialización que se debe ejecutar la imágen secundaria y se reinicia el dispositivo. La imágen de *firmware* descargada contiene un *hash* como firma del binario descargado. *MCUBoot* se compila con la llave privada que verifica que el *hash* corresponde a la imágen descargada. Al reiniciarse el dispositivo, antes de ejecutar la imágen nueva de *firmware* se verifica el *hash*. En caso de que no corresponda con la imágen recibida se vuelve a reiniciar el sistema en la versión de *firmware* anterior. Este procedimiento evita problemas por errores en la descarga de la imágen nueva de *firmware* y protege el dispositivo frente a imágenes no autorizadas mediante un sistema de verificación criptográfica.

Cada imagen debe estar firmada digitalmente con una clave privada conocida por el desarrollador. El cargador de arranque verifica esta firma usando la clave pública correspondiente antes de permitir la ejecución de la imagen. Si la verificación falla, la imagen es rechazada y no se ejecuta, impidiendo la instalación de firmware malicioso o no autorizado.

Una vez verificado el *hash*, se procede a ejecutar la imágen nueva (en los esquemas definido como *New App*). Este paso del procedimiento se conoce como etapa de *Test* ya que se valida el funcionamiento correcto de la nueva aplicación antes de su ejecución (previo a hacer la actualización definitiva). Es responsabilidad de la aplicación confirmar la imágen para que se realice el *swap* de memoria. De reiniciarse el dispositivo antes de confirmarse la imágen, el *bootloader* vuelve a ejecutar la aplicación anterior, revirtiendo la actualización. La necesidad de confirmar la imágen blinda a la aplicación de actualizaciones que dejen inoperantes a los dispositivos. En caso de que una actualización introduzca un *bug* que genera que el *firmware* se congele; al dispararse el *watchdog timer* la placa volvería a la imágen anterior resolviendo el *bug* temporalmente. Una vez la nueva imágen es confirmada usando la `API` de *MCUBoot*, se realiza un *memory swap*, el cual es un procedimiento tras el cual la imagen escrita en la partición secundaria es rotada a la partición primaria dejando libre de nuevo la partición secundaria para ser re-escrita y así realizar nuevos procesos de `FOTA`. El paso final está esquematizado en la Figura 3.13

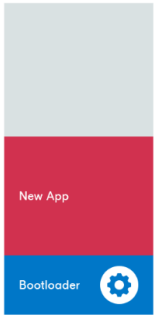


Figura 3.13: Escritura de la nueva imágen de *firmware* en el *slot* primario de flash tras la confirmación. Tomado de la lección 9 del curso intermedio de Nordic Dev Academy. [20]

Específicamente para la aplicación, el transporte usado es `HTTP` sobre la red celular. El procedimiento de `FOTA` se comienza a través de un comando `MQTT`. En ese momento se comienza la descarga del binario de actualización utilizando la biblioteca de actualización a través de `HTTP` de Zephyr *zephyr/net/fota_download.h*. De esta manera se comienza la descarga de la imágen del *bucket*.

El módulo *fota_handler*, encargado de abstraer el procedimiento al inicio de la descarga y aplicación de la misma, es una máquina de estados la cual indica el proceso de la actualización. Una vez terminada la descarga, el módulo pasa al estado correspondiente y permite aplicar la actualización reiniciando la placa. El reinicio fue diseñado diferido de la descarga para permitir que los procesos que se estén llevando a cabo tal como la escritura en memoria flash de medidas o

transmisión de las mismas se puedan completar previo al reinicio requerido. Luego de reiniciado el dispositivo, el *bootloader* se encuentra en estado de *Test* y la imágen se confirma al inicializar el módulo *fota_handler*.

Nótese que, debido al proceso de actualización, la imagen en el *slot* secundario corresponde a la versión previa del *firmware*, permitiendo revertir a esa versión siempre y cuando no se inicie un nuevo proceso de `FOTA`. Aprovechando esta característica, *fota_handler* soporta también revertir la imágen; indicando al *bootloader* volver a la versión anterior sin tener que pasar por un nuevo proceso de descarga.

3.5. Arquitectura de firmware

Se diseñó el sistema con una arquitectura de máquina de estado concurrente. Una máquina de estados rige sobre el sistema, el cuál tiene diferentes módulos ejecutados en sus propios hilos. Para el desarrollo escalable y reutilizable de la arquitectura, se implementó el ***Modular State Machine Framework*** descrito en la sección 3.5.5.

A continuación se explican los diferentes aspectos y definiciones necesarias para abordar la arquitectura de *firmware*.

3.5.1. Módulos

Se definió como módulo a la abstracción de un conjunto de responsabilidades similares a ejecutar. Dichos módulos son los encargados de disparar los métodos definidos por los diferentes *driver* para llevar a cabo los requerimientos de la aplicación así como de generar los eventos que generan las transiciones de la máquina de estados. Particularmente, se definió un módulo de recolección de datos y un módulo de consumo de datos.

El módulo de recolección de datos es el encargado de tomar las medidas de los sensores en el período de tiempo determinado por el estado actual del sistema y de guardarlas en el sistema de archivos. El módulo de consumo de datos es el encargado de enviar paquetes de datos con las medidas recolectadas, mantener la conexión con el servidor `MQTT` activa y verificar la disponibilidad de actualizaciones de *firmware* en caso de que sea capaz de aplicarlas.

3.5.2. Recursos compartidos entre módulos

Secciones de código se ejecutan transversalmente entre módulos generando potencialmente conflictos de datos compartidos. Los módulos que describen este funcionamiento son el sistema de memoria no volátil (*file system*) y el observador de la máquina de estados.

Para evitar corrupción de datos por la concurrencia de los módulos, se necesitó utilizar un mecanismo de control de acceso. Las opciones evaluadas fueron el *mutex* y el semáforo booleano (semáforo de cuenta máxima 1). Debido a la familiaridad de uso de los semáforos en otras secciones de código, se optó por utilizar semáforos de cuenta única.

Se tuvo en cuenta dos posibles esquemas para tomar los recursos. En primer lugar, permitir el acceso externo para el manejo de los semáforos y en segundo lugar, manejar los semáforos de forma privada por parte de los módulos que necesitan de este recurso.

Semáforos de interfaz pública

El uso de semáforos de interfaz pública implica que el módulo usuario es responsable de la toma y liberación del recurso. Por ejemplo, el *file system* debe contener métodos estáticos de acceso público para tomar el recurso. Los módulos deben tomar el semáforo del *file system* antes de invocar sus métodos y liberarlo una vez que hayan terminado. La principal ventaja de esta aproximación es que el módulo usuario puede definir el tiempo máximo de espera para obtener el semáforo en caso de que el recurso esté ocupado. Si transcurre ese tiempo sin poder tomar el semáforo, el módulo puede manejar esta situación de forma adecuada, como reintentar, devolver un error o realizar acciones alternativas para evitar bloqueos o esperas indefinidas.

Semáforos de interfaz privada

El uso de semáforos de interfaz privada implica que el recurso es responsable de la protección contra concurrencia. En el mismo ejemplo usado para los semáforos de interfaz pública, los métodos del módulo *file system* deben tomar un semáforo para evitar los datos compartidos y en caso de no poder hacerlo en el período de toma del recurso, retornar un error.

La principal ventaja de esta implementación es que el acople de responsabilidades entre el módulo y el dato compartido es mínima. El requerimiento de resiliencia ante la recurrencia es cumplido por el mismo objeto que es compartido; por lo que los diferentes módulos pueden usar el recurso sin tomar acciones extra para asegurar el funcionamiento.

Esquema elegido para el proyecto

Para el proyecto se decidió implementar semáforos de interfaz privada. Esto permite mantener la modularización del código y la escalabilidad sin la necesidad de que los módulos necesiten ser diseñados teniendo en cuenta que otros módulos usen el recurso; esto es, para escalar el proyecto no es necesario volver a resolver la problemática de los datos compartidos para un posible nuevo módulo.

3.5.3. Unidades de ejecución diferida

Diferentes funcionalidades del código necesitan un esquema de ejecución asíncrona respecto al resto de módulos. Particularmente la interfaz de recepción y ejecución de comandos `MQTT` y la aplicación de los procesos de `FOTA`. Como fue explicado en la sección 3.1.2, los *work* son una buena herramienta para la ejecución de dichas

unidades.

La interfaz de interacción entre las unidades de ejecución diferida con los módulos siguió un esquema específico a las características de cada una.

3.5.4. Estados

Se implementaron 4 estados del sistema que abstraen los modos de funcionamiento del mismo. Los estados son los siguientes:

1. NORMAL
2. LOW_POWER
3. ULP (Ultra Low Power)
4. CONFIG

El estado `CONFIG` es utilizado para el despliegue o mantenimiento del sistema; particularmente, este estado se centra en permitir llevar a cabo el procedimiento a través del cual se configuran las direcciones `ROM` de los sensores de temperatura del bus *1-Wire*.

El resto de estados describen el funcionamiento estacionario del sistema y transicionan entre ellos basados en las medidas tomadas del entorno; particularmente, transicionan entre ellos dependiendo del valor de la batería.

Esta transición se da acorde a la imagen Figura 3.14, donde se aprecia la existencia de dos umbrales ($V_{UmbralLow} = 3,5V$ y $V_{UmbralHigh} = 3,8V$) con un histéresis configurable que por defecto tiene su valor fijado en $75mV$.

Los umbrales fueron decididos basados en la curva de descarga de la batería Panasonic NCR18650B [25], a una tasa de descarga 0.2C y 25 °C, en donde $V_{UmbralHigh} = 3,8V$ equivale a un 68 % de carga disponible y $V_{UmbralLow} = 3,5V$ equivale a un 28 % de carga restante. Por otra parte, el margen está para evitar cambios innecesarios de estado dados por las diferencias encontradas entre medidas consecutivas en el `ADC`, y su valor fue impuesto luego de observar que podía haber diferencias de hasta $60mV$ entre medidas que no reflejaban la tasa de cambio real del voltaje en la batería.

Para transicionar a estados de mayor actividad (es decir desde `ULP` a `LOW_POWER` y desde `LOW_POWER` a `NORMAL`) se debe medir $V_{Umbral} + 75mV$, mientras que para descender a estados de menor actividad (`NORMAL` a `LOW_POWER` y de `LOW_POWER` a `ULP`) se necesita medir $V_{Umbral} - 75mV$.

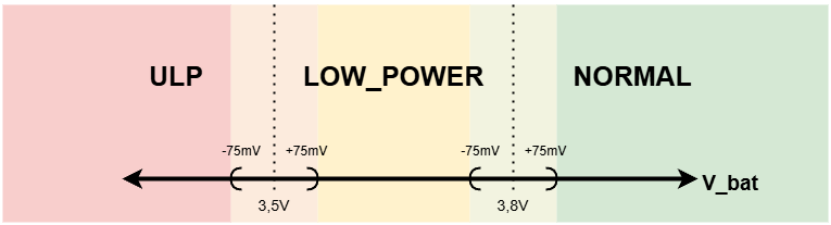


Figura 3.14: Transición de estados según el voltaje medido por el `ADC` en la batería.

Estado Normal

Este es el estado de más alta *performance*; todas las *features* programadas son utilizadas, y se configura en este estado cuando los niveles de batería son máximos. La configuración de funcionamiento en este estado toma medidas del entorno cada 1 minuto y envía los datos al servidor cada 15. Esto permite el seguimiento de las mediciones en tiempo real relativo a la variabilidad del entorno medido. Este es el único estado en donde el sistema puede realizar procesos de `FOTA` automáticamente debido al consumo que estos procesos implican.

Estado Low Power

El estado `LOW_POWER` se da en niveles medios de batería. En este estado, el sistema comienza el compromiso entre funcionalidad y consumo. En primer lugar, la toma de medidas se realiza cada 30 minutos, que relativo a la frecuencia de los cambios en los niveles de conductividad y temperatura, se mantiene un seguimiento ceñido a los valores reales (tomando como referencia trabajos previos en contextos similares [62]).

La transmisión de datos se aplaza a una vez cada 4 horas; lo que disminuye el consumo a ser 16 veces menor al consumo asociado en estado `NORMAL`. Conjugar la medida y transmisión de esta forma, permite disminuir significativamente el consumo de batería sin perder la característica de seguimiento en tiempo real en las medidas; aunque en recepción el procesamiento de las mismas se pueda ver atrasado hasta por 4 horas.

Estado ULP

El estado `ULP` (*Ultra Low Power*) se ejecuta cuando la batería está en niveles bajos y críticamente bajos. La motivación de la inclusión de este estado, es mantener las funcionalidades mínimas del proyecto (medida y eventual comunicación) aún en caso de que tormentas de invierno no permitan la carga de la batería por encima del consumo diario durante largos períodos de tiempo. En este estado, las medidas se toman cada media hora al igual que en el estado `LOW_POWER`, manteniendo una frecuencia acorde a los cambios de conductividad y temperatura en el agua, y no se envían datos al servidor. Al mantener el módulo de transmisión apagado, se ahorra el consumo asociado al envío de datos. Al acumularse los datos en flash, los mismos serán enviados una vez se esté en un estado de mayor capacidad de carga.

Dada la configuración por defecto del *file system* desarrollada en la sección Subsección 3.3.3 y considerando que en este estado se muestrea cada 30 minutos, se espera contar con un período aproximado de 90 días sin transmitir medidas ni sobrescribir datos almacenados.

Estado CONFIG

El estado *CONFIG* se ejecuta cuando un operario sitúa el *switch* de confi-

guración (ver Figura 3.15) en posición activa y tiene el objetivo de registrar las direcciones `ROM` en el bus *1-Wire* de cada sensor de temperatura indexado a su posición (interior o exterior).

Este estado funciona como una sub-máquina de estados a través de los cuales se registra cada sensor de temperatura. Los sub-estados representan los pasos de configuración necesarios y las transiciones entre los mismos se llevan a cabo usando el botón 1 de la placa (ver Figura 3.15).

El operario es guiado en el proceso utilizando *logs* en línea de comandos como interfaz de usuario, lo que le permite conectar en el orden correcto los sensores.

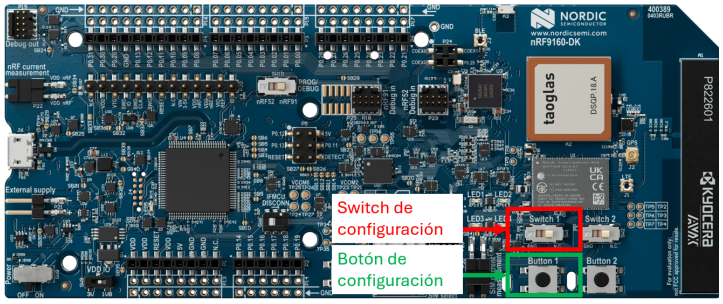


Figura 3.15: Switch y botón elegidos para el uso del modo *CONFIG*.

Etapas del proceso de configuración.

- **SENSOR_1:** El sistema solicita al usuario que conecte el sensor de temperatura interior y que a posteriori presione el botón. En el momento en que el botón es presionado, se registra la `ROM` interior en el bus de *1-Wire*.*
- **SENSOR_2:** El sistema solicita al usuario que conecte el sensor de temperatura exterior y que a posteriori presione el botón. En el momento en que el botón es presionado, se registra la `ROM` exterior en el bus de *1-Wire**
- **FIN:** El sistema fue capaz de leer correctamente las direcciones de los sensores y está pronto para salir del estado de configuración. Se indica al operario que devuelva el *switch* a la posición original para terminar el proceso.
- **ERROR:** En caso de que haya ocurrido un error durante la configuración, se transiciona a este sub-estado, en el cual se pide al operario que comience de vuelta el proceso de configuración.

* En caso de que se escanee una cantidad de direcciones no correspondiente a la esperada en el sub-estado (por ejemplo, 2 direcciones o ninguna en el sub-estado **SENSOR_1**, se transiciona al estado **ERROR** en lugar de continuar con los sub-estados de forma normal).

A continuación (Figura 3.16) un diagrama de estados de la configuración de los sensores de temperatura:

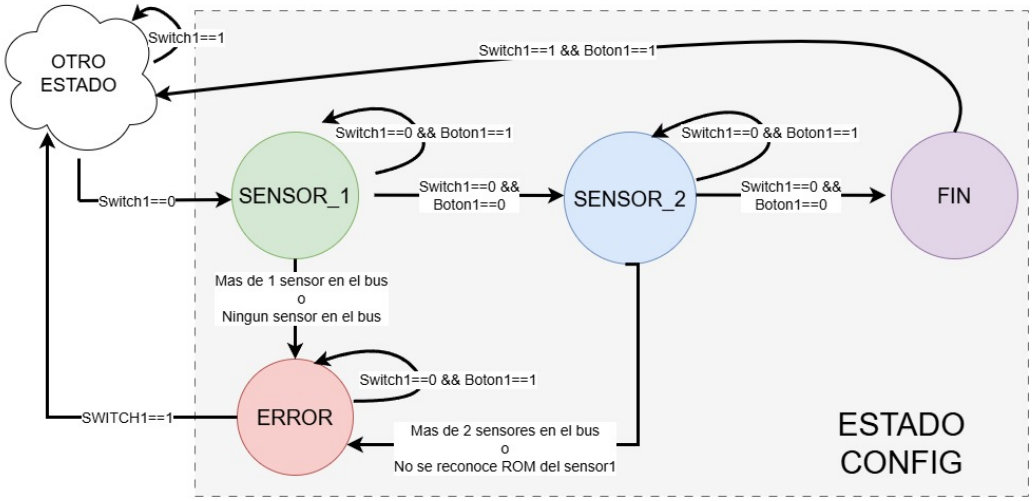


Figura 3.16: Diagrama de transición de estados del modo `CONFIG`.
Nota: Tanto los sensores como los switches funcionan por nivel bajo en la placa utilizada.

3.5.5. Modular State Machine Framework

Para construir una máquina de estados escalable y reutilizable en próximos proyectos basados en Zephyr, se implementó un *framework* para crear máquinas de estado con módulos concurrentes basado en el *framwork* de máquina de estados otorgado por Zephyr. Las máquinas de estados generadas por el `SMF` (*state machine framework*) están dispuestas para ser ejecutadas en un solo hilo, por lo que el valor agregado del *framework* desarrollado en el proyecto es la capacidad de ser utilizado en sistemas concurrentes de forma directa.

State Machine Framework

Zephyr provee un *framework* para definir máquinas de estados simple de usar, versátil y moldeable; por lo que se decidió usarlo para definir la máquina de estados del sistema. El *framework* de máquina de estados de Zephyr, permite para cada estado la definición de una función de ejecución, una función de entrada y una función de salida del estado (ver Figura 3.17) lo cual es muy útil para el requerimiento del proyecto de ajustar los *timeouts* de los *timers* que determinan el momento de tomar medidas o de enviar datos en las transiciones. Estos métodos y transiciones se ejecutan en el hilo nativo de la máquina de estados.

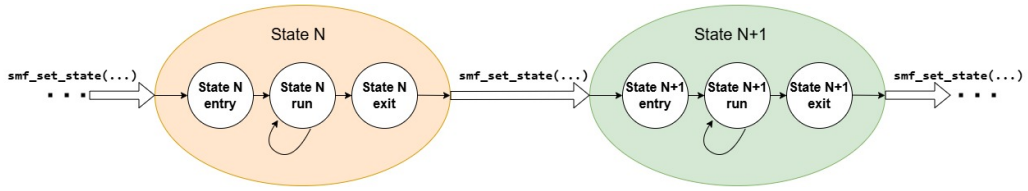


Figura 3.17: Esquema equivalente de transición entre estados utilizando el State Machine Framework de Zephyr.

Implementación de MSMF

Para definir un *framework* de máquina de estados modular, se necesitó definir un núcleo que invoque las funciones del *framework* de máquina de estados de Zephyr, una estructura replicable que represente un módulo y una interfaz para que los módulos interactúen con la máquina de estados. Como núcleo, se implementó el **"state_machine_observer"** que ejecuta la máquina de estados y sus transiciones; y se definió una clase **"module"** a partir de la cuál, se construyen los módulos. La interfaz de interacción desde el *observer* a los módulos se da por el registro de los mismos al ser inicializados. El registro de los módulos le da acceso al observador a su interfaz privada en la cual están definidos los métodos de ingreso, ejecución y salida de cada estado.

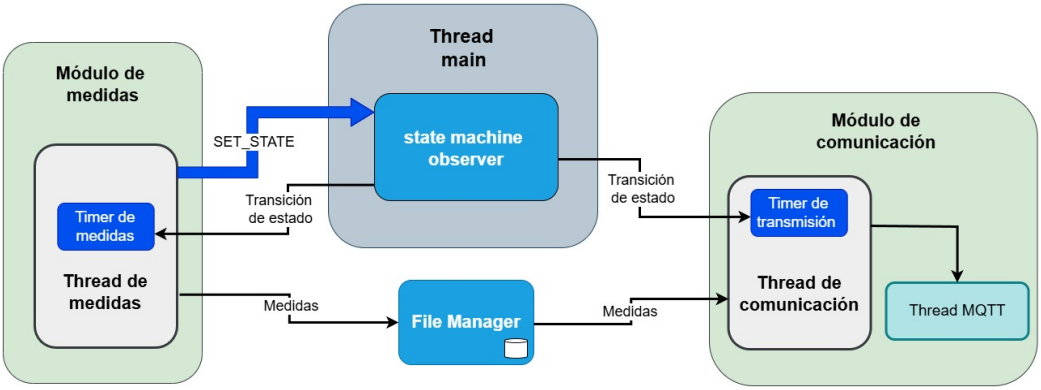


Figura 3.18: Esquema de la arquitectura de firmware usando MSMF.

Implementación de módulos

El objeto módulo define las características de la interfaz utilizada por el núcleo de la máquina de estados. Un módulo se compone por un set de ***state_handlers*** que definen las interacciones necesarias para cada estado. Un *state_handler* es un objeto que abstrae la entrada, salida y ejecución síncrona de un estado. Los módulos tienen un puntero a *state_handler* por cada estado definido a nivel sistema y su constructor debe rutear los punteros a las instancias de *state_handler* a su estado correspondiente.

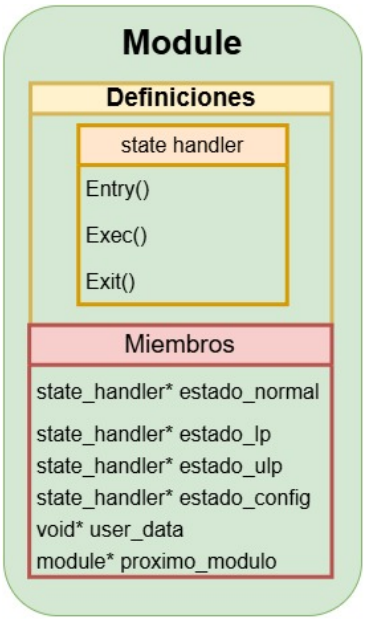


Figura 3.19: Descripción gráfica de la implementación de un módulo.

Esta implementación permite usar un mismo *state_handler* para diferentes estados de un mismo módulo. Esto es provechoso ya que para módulos que se deben comportar de una misma forma para diferentes estados, al usar el mismo *state_handler* para ambos estados no se necesita implementar diferentes objetos y métodos con un comportamiento igual.

Por la implementación del *framework*, el *state_machine_observer* guarda los módulos registrados como una lista en cascada, por lo que cada módulo tiene como miembro un puntero al próximo módulo para ejecutar los procesos síncronos.

Módulo de medidas

El módulo de medidas define *handlers* para los estados `NORMAL`, `ULP` y `CONFIG`. Como el intervalo de medida de los periféricos en estado `LOW_POWER` y `ULP` son iguales, el *handler* de estado utilizado es el mismo.

El módulo de medidas define un hilo de ejecución que, en un *loop* infinito, toma una medida e intenta adquirir un semáforo booleano. La cadencia de las mediciones es controlada por un *timer* que, al expirar, habilita el semáforo y dispara la toma de un conjunto de mediciones del sensor de conductividad y de los sensores de temperatura. Además, al hilo se le incorporó un canal de `WDT` para monitorear que la toma de medidas no se interrumpa por períodos prolongados debido a errores en el firmware. La configuración del `WDT` para este módulo es de 45 minutos, lo que permite que, incluso en el mayor intervalo entre mediciones (30 minutos), no se active el reinicio del dispositivo.

Los *state_handlers* del módulo de medidas configuran el tiempo de expiración del *timer* de flujo de medidas en la entrada a los estados. La ventaja del uso del mecanismo de *timer* + *semáforo* en lugar de *delay* del hilo es que normalmente

los ciclos de *delay* dependen de polling a nivel de *kernel* de los *ticks* del sistema, generando un mayor procesamiento y por lo tanto un mayor consumo de batería asociado al uso de `CPU`. Además, el uso de un *timer* permite el disparo periódico de medidas sin el retraso que supone la acumulación del tiempo de medida.

Otra ventaja de esta implementación es que se tiene una completa desvinculación del módulo de comunicaciones. En caso de que dicho módulo falle, el muestreo de datos no será interrumpido independientemente del estado de la conexión actual con el *broker*.

El *state_handler* de `CONFIG` implementa el mecanismo de manejo de la configuración, incluyendo las instrucciones al operario y el manejo de *switch* y botón. La ejecución de la máquina de estados de configuración se da de forma síncrona con el núcleo de la máquina de estados, por lo que es ejecutada en el hilo de *main*.

Módulo de comunicaciones

El módulo de comunicaciones define *handlers* para los estados `NORMAL`, `LOW_POWER` y `ULP`. Como tanto en estado `ULP` como en el estado `CONFIG` no se envían datos al *broker* `MQTT`, se utiliza el mismo *handler* para ambos estados.

De la misma forma que el módulo de medidas, éste módulo usa un hilo, un *timer* y un semáforo para controlar el flujo de envío de datos. Las transiciones de estado controlan los períodos de envío mediante la determinación del período del *timer*. El *timer*, al expirar, libera el semáforo para que el hilo ejecute el envío de todos los datos almacenados en memoria no volátil.

Las medidas se organizan para ser enviadas en grupos de a 10. Para confirmar que las medidas son recibidas por el *broker*, se utiliza el *callback* de confirmación de envío de *mqtt_connection* en conjunto con la confirmación de lectura de *file_manager* además de un semáforo que evita condiciones de carrera relacionadas a múltiples envíos previos a la confirmación del envío de un dato. En caso de que un dato no sea enviado correctamente, se vuelve a intentar el envío del mismo conjunto de datos.

El módulo de comunicaciones también se encarga de monitorear y mantener el buen estado de la conexión `MQTT` sobre la red celular. En caso de detectar un error en el envío o de reportarse un error desde el módulo `MQTT`, se re-intenta la conexión volviendo a la etapa de inicialización del hilo. Esto es particularmente importante ya que cada 12 horas se da un evento de desconexión relacionado con el período `TAU` (Tracking Area Update) de la capa celular que obliga a tener un mecanismo de reacción capaz de funcionar periódicamente sin intervenir con la funcionalidad de medición.

Dicho evento de desconexión se detectó experimentalmente y según los logs se origina exactamente a las 12 horas de que el dispositivo se conecte a la red. El error se origina en la capa de conexión a la red celular y según foros de Nordic, el error se podía trazar a una desconfiguración en los *timers* de largo plazo; fundamentales para mantener la conexión (`T3412` correspondiente a tiempos entre `TAU`) [28].

Diagramas de flujo de los *threads* de comunicaciones y medidas

A continuación (Figura 3.20) se presenta un diagrama de flujo de los *threads* de comunicación y medidas, detallando también el *thread* de `MQTT` y su mecanismo de reconexión.

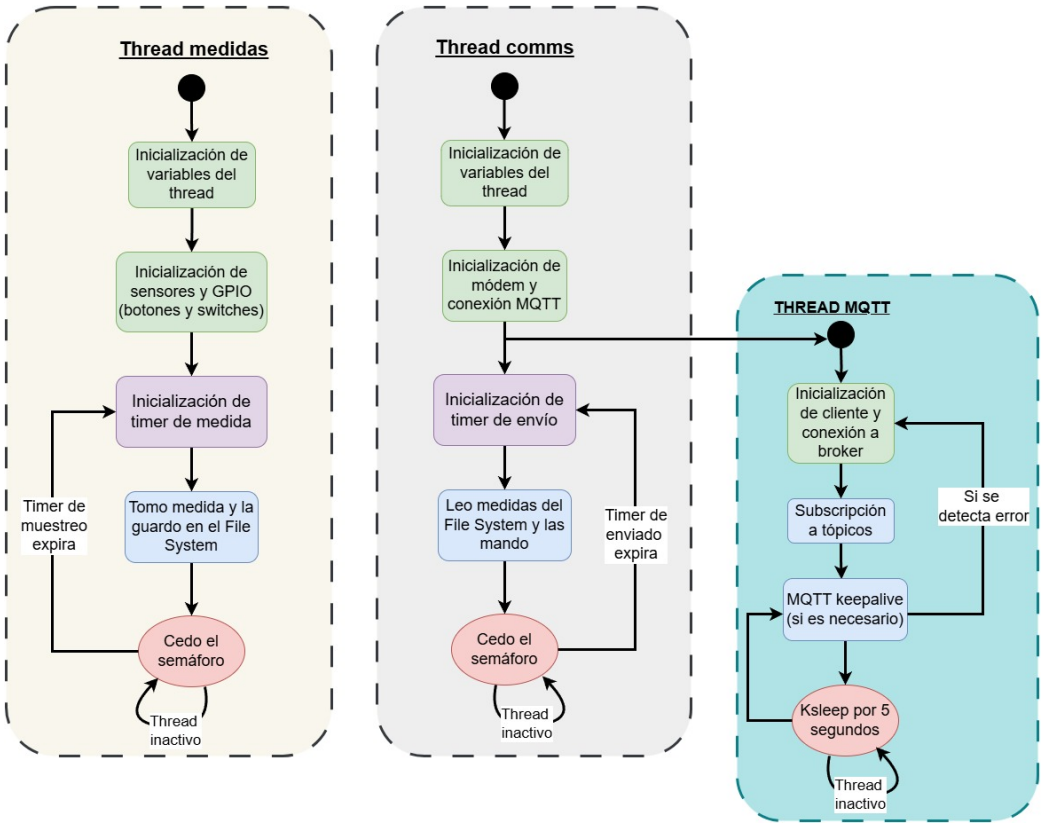


Figura 3.20: Esquema de *threads* (hilos) de comunicación y de medidas.

Esta página ha sido intencionalmente dejada en blanco.

Capítulo 4

Herramientas complementarias

Este capítulo presenta las herramientas de soporte desarrolladas para complementar el funcionamiento del sistema. Se aborda la implementación de la infraestructura necesaria para el despliegue, mantenimiento y monitoreo del dispositivo, aspectos fundamentales para garantizar la operación autónoma en las condiciones de instalación mencionadas en capítulos anteriores.

El capítulo describe la implementación del servidor de actualizaciones utilizando `AWS S3`, que permite la distribución segura de *firmware* mediante `HTTP` con entornos separados para pruebas y despliegue oficial. Se detalla el sistema de integración y despliegue continuo (`CI/CD`) basado en GitLab, que automatiza la compilación, verificación y distribución del código, incluyendo diferentes versiones optimizadas según el propósito específico (desarrollo, medición de consumo, producción).

Además, se presenta la solución desarrollada para la recolección y visualización de datos mediante *scripts* de Python, que supera las limitaciones del *broker* `MQTT` proporcionando almacenamiento permanente, interfaz de control remoto y capacidades de análisis gráfico. Por último, se aborda la documentación técnica orientada a futuros desarrolladores, incluyendo guías de instalación, tutoriales específicos y documentación de arquitectura, asegurando la continuidad del proyecto.

4.1. Servidor de actualizaciones

Como se explicó en la sección 3.4.4, las actualizaciones de *firmware* se realizaron mediante `HTTP`; por lo que es preciso disponer de un servidor capaz de procesar *requests* `HTTP` para descargar archivos.

Para crear el servidor `HTTP` se utilizaron los servicios de *Amazon web Services* (`AWS` de aquí en adelante); particularmente el servicio de *Bucket S3*. Los *buckets S3* [3] son servidores `HTTP/HTTPS` que manejan archivos, permitiendo guardarlos o descargarlos a través de comandos *PUT* y *GET*. El servicio incluye la obtención de un *host-name* ligado al servidor `DNS` de `AWS`.

Una de las características de los *Buckets S3* es su orientación a servicios máquina a máquina (`M2M` por sus siglas en inglés). Esto es: que las interacciones están

orientadas a puntos de acceso y credenciales para máquinas, lo que permite el acceso desde diferentes aplicaciones como lo es el *firmware* del dispositivo. El diseño del servidor incluye la definición de los servicios otorgados por el mismo y del mecanismo y permisos de acceso a esos servicios.

4.1.1. Servicios definidos

En vistas de tener la posibilidad de probar las actualizaciones antes de desplegarlas oficialmente, se crearon 2 *buckets*; uno dedicado al despliegue oficial de imágenes de *firmware* y otro dedicado al despliegue pre-oficial de versiones. Esto permite probar una imágen en dispositivos fácilmente accesibles previo a que los dispositivos en campo se expongan a las versiones actualizadas.

Respecto al archivo de actualización *app_update.bin*, se permite el comando *GET* a cualquiera que lo requiera, lo que se traduce a que cualquier dispositivo o usuario es capaz de descargar el archivo siempre y cuando disponga de la `URL` del recurso. Esta característica permite que los dispositivos desplegados descarguen el archivo de actualización sin definición de credenciales. Los permisos de *POST* sobre el recurso están limitado a un usuario con acceso `M2M` usado en las acciones de GitLab para publicar el archivo más nuevo para que las placas sean capaces de obtener la última versión de *firmware* de una forma automática.

4.1.2. Usuarios definidos

Las cuentas de `AWS` pueden definir diferentes usuarios para acceder a los recursos desplegados. Cada cuenta tiene un usuario *root*, y es el asociado a la persona que crea y maneja los recursos. Como fue introducido en la sección 4.1.1, `AWS` permite la definición de usuarios junto a sus derechos de acceso a diferentes recursos; dichos usuarios se denominan `IAM`.

Un usuario `IAM` es un objeto definido en la nube que abstrae un rol en la misma. El funcionamiento de los usuarios `IAM` es comparable con el funcionamiento de llaves digitales de acceso a un edificio. Una llave con derechos de visitante podría acceder a los espacios públicos de las instalaciones mientras que una llave con derechos administrativos podría acceder a los lugares públicos como a salas de servidores y de mantenimiento. Naturalmente, el *root user* de la cuenta cuenta con derecho de acceso ilimitado a cualquier recurso. Un usuario `IAM` puede ser accedido a través de usuario y contraseña (orientado a humanos) o a través de *ACCESS_ID* y *SECRET_ACCESS_KEY* (orientado a interacciones `M2M`).

Para el proyecto, se definió un usuario `IAM` orientado al acceso del repositorio de GitLab para publicar las imágenes de *firmware*. El usuario del repositorio tiene permisos de *PUT* a los *buckets* en donde se publica el *firmware*. El procedimiento a través del cual se generan y publican imágenes de *firmware* a los *buckets* se profundiza en la sección 4.2.

4.2. CI/CD

Las siglas `CI/CD` provienen de *Continuous Improvement/Continuous Development* y refiere a aquellos procedimientos que se realizan en el servidor de Git utilizado por un proyecto para establecer un desarrollo continuo o mejora continua. Estos procesos incluyen compilado de código, *testing* del mismo y despliegue del código. Para el proyecto estas acciones se utlizaron para compilar versiones oficiales de código. Tomando un enfoque más generalista respecto a la mejora continua y desarrollo continuo, se consideró como parte del proyecto plantar las bases para la continuación posterior del mismo documentando localmente el *set up* del proyecto, así como tutoriales sobre cómo utilizar las diferentes herramientas desarrolladas en archivos *README* y en páginas de *wiki* del proyecto.

4.2.1. Herramientas de mejora continua

Para el desarrollo del proyecto y producción de archivos de actualización se utilizó la herramienta de *CI/CD* de Gitlab. En esta sección se explican los diferentes aspectos de dichas acciones y cómo fueron utilizadas en el proyecto.

Gitlab permite definir *scripts* para ser ejecutados al momento de realizar *commits*, *Merge Requests* (`MR` de aquí en adelante) y *tags*. Estos *scripts* son normalmente utilizados para realizar chequeos sobre el código, realizar *tests* y acciones de despliegue de código.

Pipelines, Jobs y steps

El *script* de *CI/CD* en GitLab define un ***pipeline*** que contiene ***jobs*** que se ejecutan en orden de distintos ***steps*** diferentes.

Esta estructura jerárquica permite la compilación de código, *testing* del mismo en diferentes pasos y despliegue del mismo para actualizar dispositivos en campo.

Los trabajos definidos son ejecutados en ***runners***, las cuales son máquinas distribuidas capaces de actuar como servidores para ejecutar las tareas definidas para el trabajo. Los *runners* son registrados en GitLab con nombres (para que personas puedan discernir entre distintos *runners*) y con etiquetas para ser discriminados dependiendo de los requerimientos del trabajo a ser ejecutado. Además, los *runners* definen su ejecutor, el cual determina el tipo de interfaz de línea de comandos usada por el trabajo realizado. En síntesis, la lista de instrucciones definida para cada trabajo, es interpretada por el *runner* como una lista de comandos ingresados por terminal.

Arquitectura de red para el desarrollo continuo

La arquitectura para el desarrollo continuo comprende el relacionamiento entre el proceso de desarrollo y el proceso de despliegue. Para el proyecto, el despliegue

del código creado se basa en los *commits* realizados a la rama *main* y *test* de git. Cada *commit* a estas ramas resulta en la actualización del binario correspondiente a cada una; para *main*, el *bucket* oficial mientras que para *test* el *bucket* de pruebas. Ambos *jobs* generan el binario completo de la aplicación disponible (incluyendo *bootloader*, *firmware* propietario del modem e imágen de la aplicación) para programar la placa usando el programador partiendo de una versión desplegada y sin la necesidad de instalar las herramientas de desarrollo del proyecto.

La rama *main* se protegió para que ningún usuario pueda hacer *commits* a la misma directamente. La única forma de integrar cambios (que serán desplegados a campo) a la rama son a través de *merge requests*, los cuales deben ser aprobados por los *reviewers* asignados. Esto permite que el proyecto avance con el consenso explícito de los desarrolladores lo que obliga al aporte conjunto sobre el código integrado; tanto en la funcionalidad del código como en la legibilidad para desarrollo futuro sobre el módulo integrado o modificado. Esto se condensa a que el desarrollo se realiza en ramas separadas lo que permite una mayor trazabilidad del código.

Pipeline desarrollado en el proyecto

El *pipeline* desarrollado para el proyecto se configuró para ser ejecutado en todos aquellos *commits* a la *branch main* o hacia la branch *test*, y sobre cualquier *Merge Request* (ver Figura 4.4).

El *pipeline* primeramente compila el código a modo de una primera instancia de verificación. En caso de no tener éxito la compilación del mismo, el *pipeline* falla y en caso de ser sobre un *Merge Request*, el mismo no puede ser integrado a *main* (o *test*) hasta que el *pipeline* no se complete exitosamente. Esto da una primera línea de defensa ante actualizaciones de *firmware* con código compilado fuera del *CI*, puesto a que se compila de manera independiente al entorno y equipo de todos los desarrolladores.

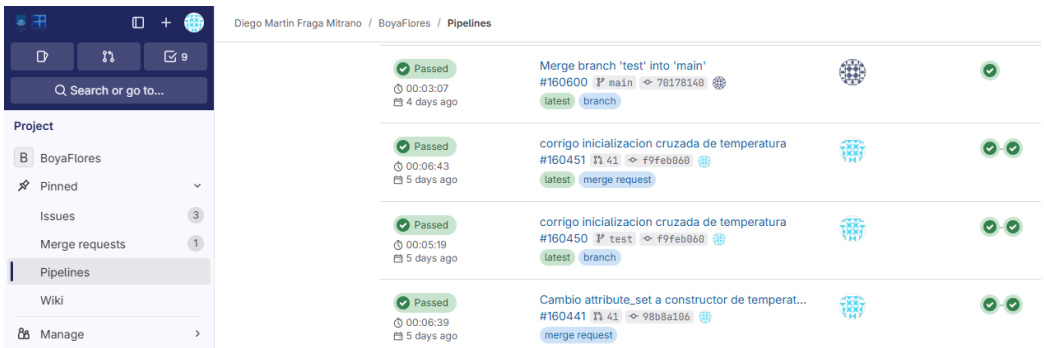


Figura 4.1: Muestra de pipelines corridos en la pestaña *Pipeline* en GitLab.

En caso de que la compilación del código sea exitosa, se guardan en forma de *artifact* tanto el binario de actualización como el binario completo del proyecto para su disponibilidad. El *artifact* llamado *app_update.bin* se utiliza para las actualizaciones de *firmware* de forma remota mientras que para programarlo mediante

USB se puede utilizar el *artifact* de tipo *.hex* con la herramienta **Programmer** de *Nordic* que se puede instalar desde el *nRFConnect* [26].

Cuando se realiza un *commit* a una *branch* que tiene un `MR` abierto hacia cualquiera de las dos *branches* se tiene la particularidad de que se lanzan varios *jobs* en simultáneo separados en dos etapas o *stages* (llamados *build* y *build_y_deploy*) que compilan el código. Estos *jobs* difieren en los archivos extra de configuración (*.conf*) a la hora de la compilación y también difieren en el **EXTRAVERSION** presente en el archivo *VERSION* del proyecto el cuál indica si es una versión oficial (llamada *deploy*), o si es una versión dedicada a *testing*, en cuyo caso se divide a su vez en cuatro posibilidades: *debug*, *normal*, *lp* y *ulp*.

Las últimas tres versiones mencionadas fuerzan el estado de la placa (a `NORMAL`, `LOW_POWER` y `ULP` respectivamente) para tomar mediciones de consumo en cada uno de ellos sin tener activado el *logger* mientras que *debug* mantiene la funcionalidad de cambiar de estado de la placa dinámicamente pero teniendo el módulo *logger* activado.

Para lograr la compilación de los diferentes perfiles de *testing*, se utilizó la capacidad de definir *overlays* en los archivos de configuración. Esto es, que para cada perfil se creó un archivo que define las configuraciones necesarias para cada perfil. Estos archivos son denominados $overlay\_ < feature\ agregada > .conf$. De esta forma, desde el comando de compilación se puede integrar y configurar código sin cambios intrusivos en el repositorio. Por ejemplo, en los archivos de configuración específicos que fuerzan la placa a cada modo, se configura al código para forzar el nivel de batería (no usar el nivel medido para las transiciones de estado del sistema) y se configura el nivel de batería forzado a uno correspondiente al estado que se quiere forzar. Para los *commits* que se realizan sobre la rama *test*, se usa un *overlay* de configuración que habilita los *logs* en consola `RTT` (Real Time Transfer) hasta el nivel *debug*.

Esto es de gran ayuda a la hora del *testing*, puesto que el *logger* si bien es una herramienta imprescindible a la hora de hacer *debugging*, consume recursos energéticos debido a que despierta al sistema en reiteradas ocasiones con el motivo de mostrarle al desarrollador en pantalla los *logs*. Esto perjudica el consumo de la placa, por lo tanto en una versión oficial del *firmware* no deberían estar presentes los *logs* ni tampoco cuando se esté comprobando el consumo en los distintos estados.

Cuando el *commit* se hace sobre la rama *main*, sólamente se ejecutará en el *stage build_y_deploy* la compilación con el argumento **EXTRAVERSION = deploy**. De los binarios producidos llamados *merged.hex* y *app_update.bin*, sólo el último es publicado en el *bucket* de `AWS` llamado **s3://boya-flores-app-fw** del proyecto permitiendo la actualización de *firmware* del dispositivo desplegado (ver Figura 4.2).

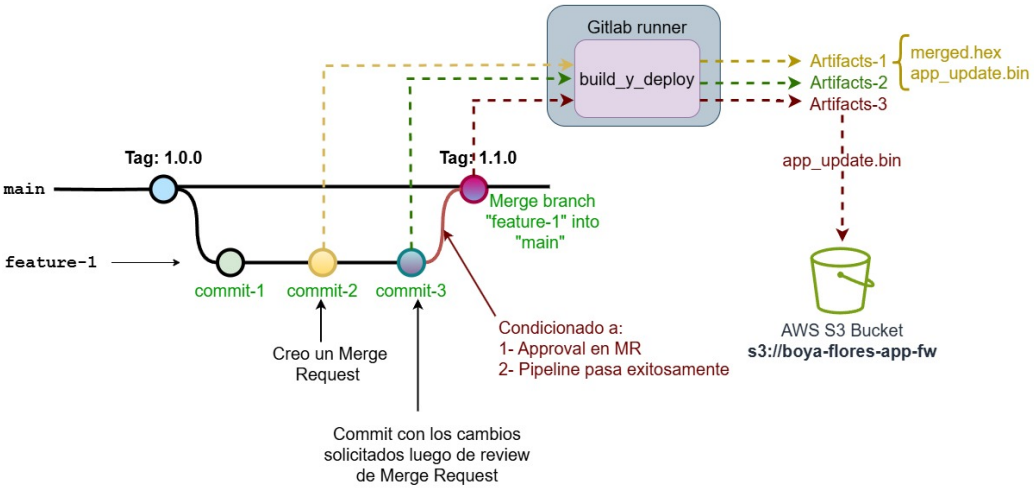


Figura 4.2: Esquema de funcionamiento de CI/CD cuando se tiene un *Merge Request* hacia la branch *main* en caso de que se agregue una *feature* (funcionalidad) nueva.

Por otra parte, en caso que el *commit* sea sobre la rama *test*, la compilación se ejecutará tal y como cuando se tiene un *merge request*, con la diferencia de que en este caso sólamente se subirá al segundo *bucket* llamado **s3://boya-flores-debug** el archivo *app_update.bin* del *step build_y_deploy* con el argumento **EXTRAVERSION = debug**.

Esto permite que la *branch test* sea lo suficientemente versátil, teniendo a disposición todas las versiones de *artifacts* disponibles para ser programadas e incluso con la utilización del *bucket* de *debugging*, poder probar *firmware* que será aplicado mediante actualización remota utilizando el comando `TEST`.

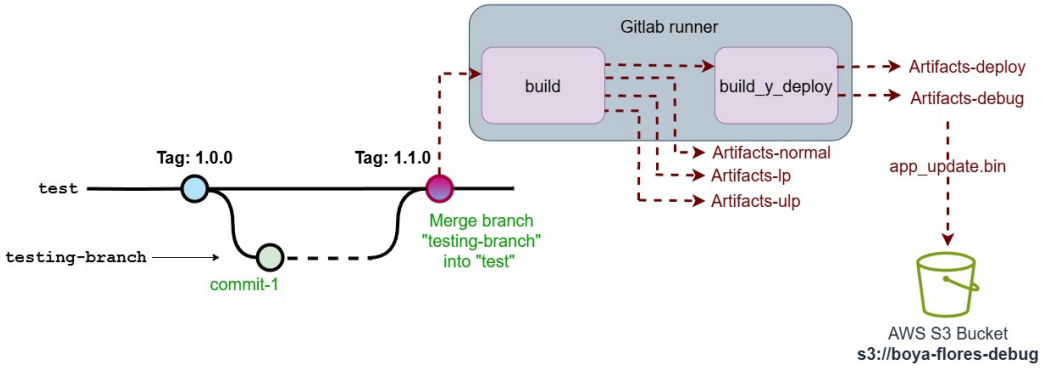


Figura 4.3: Esquema de funcionamiento de `CI/CD` cuando se tiene un *commit* hacia la *branch test*.
Se puede apreciar como se tienen todos los *artifacts* compilados cuando se termina la ejecución del pipeline pero sólamente se sube al bucket el archivo *app_update.bin* proveniente de *debug*.

Como política interna del grupo para el desarrollo de este proyecto, ante un *merge request* siempre se asigna a un segundo integrante del grupo como *reviewer*,

el cuál sugiere cambios a través de un *review*. Esto es para tener un control cuando se aplican cambios, además de proveer retroalimentación o recomendaciones para implementarse en el *merge request* previo a la integración del código. Una vez los cambios fueron revisados y aceptados por el segundo compañero se aprueban los cambios para que se pueda realizar el *merge* de la *branch* a *main* tal y como se ve en la Figura 4.2.

Para la *branch test* se consideró que no era necesaria la aprobación de un segundo compañero por lo que el único requerimiento para integrar cambios a esa *branch* es que el código compile correctamente (ver Figura 4.3).

Resumiendo, el siguiente diagrama de flujo refleja el accionar del *pipeline* de este repositorio (ver Figura 4.4):

4.3. Script de recolección de datos

Actualmente, el plan utilizado de *broker* `EMQX` [15] para este proyecto no contempla el almacenamiento de datos, por lo que el *broker* se limita a distribuir los mensajes enviados a todos los subscriptores de los distintos tópicos.

Si bien la aplicación utilizada para el monitoreo (`MQTTx` [23]) de los datos tiene la opción de importar las medidas tomadas, solo soporta la exportación en ciertos formatos estándares como ser `CSV`, `JSON`, `YAML`, entre otros. Es decir los datos que se exportan deben estar escritos en alguno de estos formatos de antemano.

Además mediante pruebas experimentales, se confirmó que si la computadora donde se está ejecutando la aplicación se deja sin actividad durante un tiempo indefinido la aplicación da de baja la conexión automáticamente. Esto puede resultar ineficiente si se quiere dejar a la aplicación adquiriendo datos constantemente lo cual es el objetivo del proyecto, por lo que se evaluaron otras opciones para continuar con el propósito de guardar los datos de manera continua y eficiente.

Se tomó la decisión de implementar *scripts* en Python para la recolección automática de datos, para esto se utilizó como base el código de ejemplo provisto por `EMQX` en su repositorio de GitHub [57] para establecer una conexión `TLS` como subscriptor a uno de sus *broker*, y se siguieron los pasos planteados en el tutorial provisto por la empresa [61].

Una vez se ejecutó exitosamente el ejemplo se procedió a la adaptación del *script* para las necesidades de este proyecto. Se optó por modularizar el trabajo, dejando un *script* para la recolección de datos y manejo de conexión con el servidor, un segundo código para graficar los datos tomados y por último para facilitar la interfaz con el usuario se implementó un tercer *script* para poder insertar comandos en la terminal. De esta forma se tiene independencia para poder cambiar cada módulo de manera independiente, de forma que si se desea, se puede por ejemplo implementar una interfaz gráfica en lugar de interfaz mediante línea de comandos y el resto del código no se verá afectado.

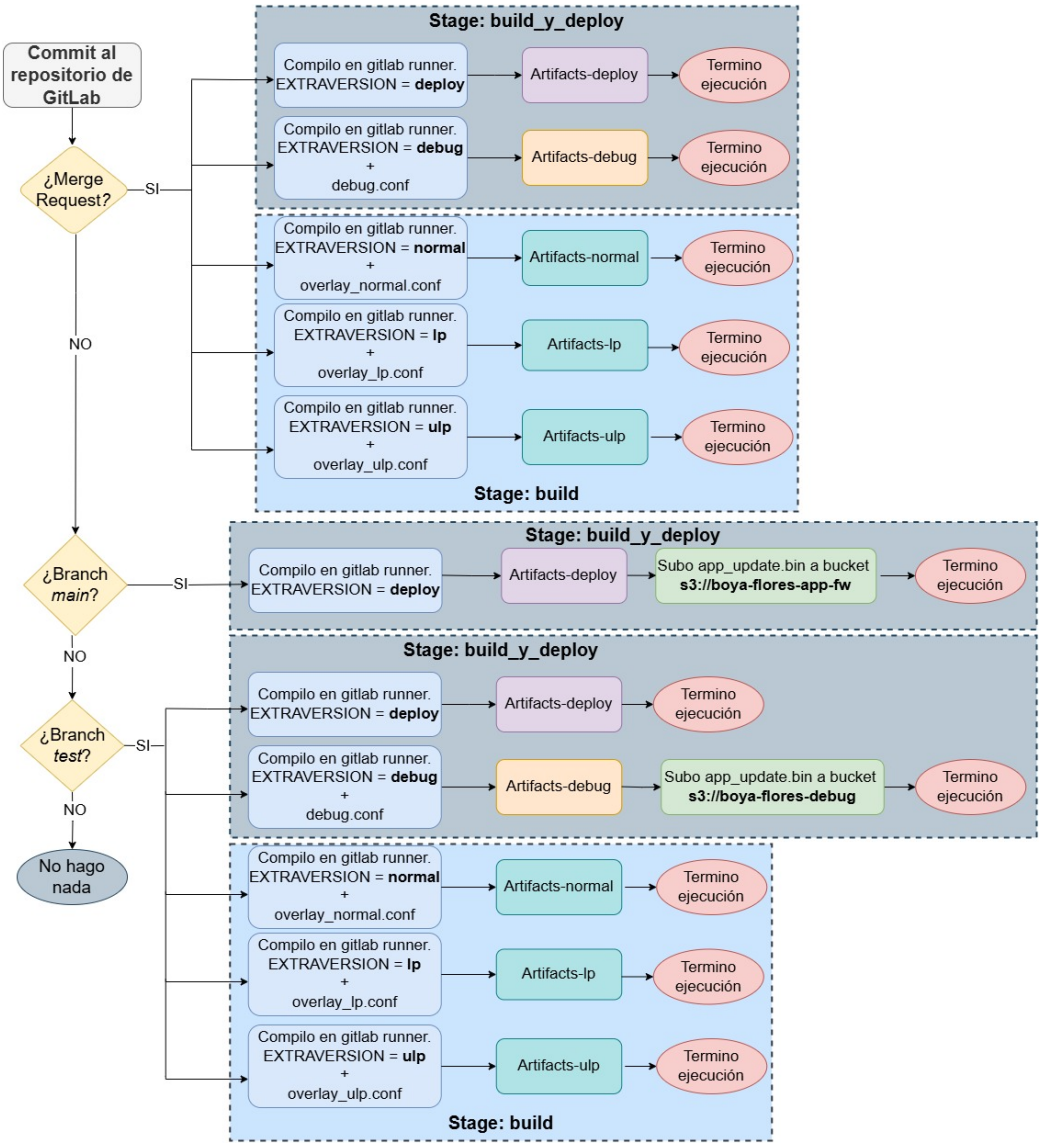


Figura 4.4: Diagrama de flujo de funcionamiento del gitlab-ci.yml cuando se realiza un commit al repositorio.

Manejo de conexión, almacenamiento y graficado de datos

El *script* de conexión y almacenamiento llamado *mqtt_sub_boya.py* utiliza la librería *paho-mqtt* [35] de Python para establecer la conexión con el *broker* y subscribirse a los tópicos, mientras que usa otras librerías estándar de Python como ser *os*, *csv* y *time* para las funciones de almacenamiento.

Una vez se establece la conexión con el *broker* y se suscribe al tópico de *comandos*, la librería *paho-mqtt* provee mediante *callbacks* un método fácil para manejar las acciones a tomar en caso de recibir un mensaje `MQTT`, lo que lo hace ideal para la situación requerida. Cuando se recibe un mensaje, se ejecuta el *callback* que

maneja la recepción del mismo. Allí se clasifica dependiendo del tópico a donde fue enviado, en este caso las únicas dos opciones que importan son `boya/datos` o `boya/logs`.

En el caso de que el mensaje recibido corresponda al tópico `boya/datos`, las medidas se transmiten como texto plano en un formato delimitado por comas, conteniendo: temperatura interior, temperatura exterior, voltaje de batería, conductividad y el *timestamp* de la medición. Al recibirse, este *string* se separa en sus componentes y luego se almacena en un archivo de tipo **.csv**, donde cada columna corresponde a una de las variables mencionadas y cada fila representa una medición.

En el directorio generado *data_files/* se guarda un archivo con el *timestamp* de cuando se recibió la primera medida cuyo nombre sigue el siguiente patrón: *sensor_data_YYYY-MM-DD_HH-MM-SS*. Este archivo no se borra cuando se realiza una conexión nueva sino que perdura en el directorio y queda a modo de registro para tener una base de datos de medidas (ver Figura 4.5).

Para el caso de los mensajes que llegan al tópico *boya/logs* ocurre un proceso similar. Se tiene un directorio llamado *logs/* en donde se registra de la misma manera los *logs* recibidos en un archivo con formato *logs_YYYY-MM-DD_HH-MM-SS* para que quede un registro por conexión realizada (ver Figura 4.5).

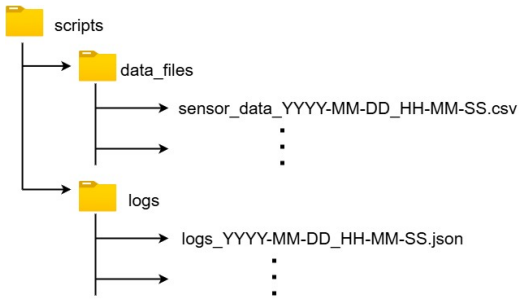


Figura 4.5: Esquema de directorios donde se guardan los datos recolectados.

Los *logs* en esta ocasión son almacenados en formato *JSON*, el cuál posee una sintaxis que resulta visualmente fácil de interpretar, ideal para este caso.

Como los *logs* enviados por la placa tienen un formato variable y distinto a los datos, la primera estructura o *key* que se guarda es el *timestamp* conteniendo la fecha y hora exacta de cuando se recibió el *log*, seguido por una segunda `key` llamada **mensaje** que va a contener el *payload* recibido. Dicho *payload* puede ser el mensaje proveniente de un comando `STATUS` como también de aviso mensajes provenientes de la placa luego de una acción como puede ser una actualización de *firmware*.

Es importante recalcar que ambos archivos tanto el de los datos como el de los *logs* son creados independientemente, es decir el archivo de *log* puede haber sido creado antes que el archivo contenedor de datos y viceversa, todo en la misma sesión de conexión `MQTT` (ver Figura 4.6). Cuando esta conexión se interrumpe se

generarán nuevos archivos (en cuanto se reciba el primer *log* y la el primer dato respectivamente).

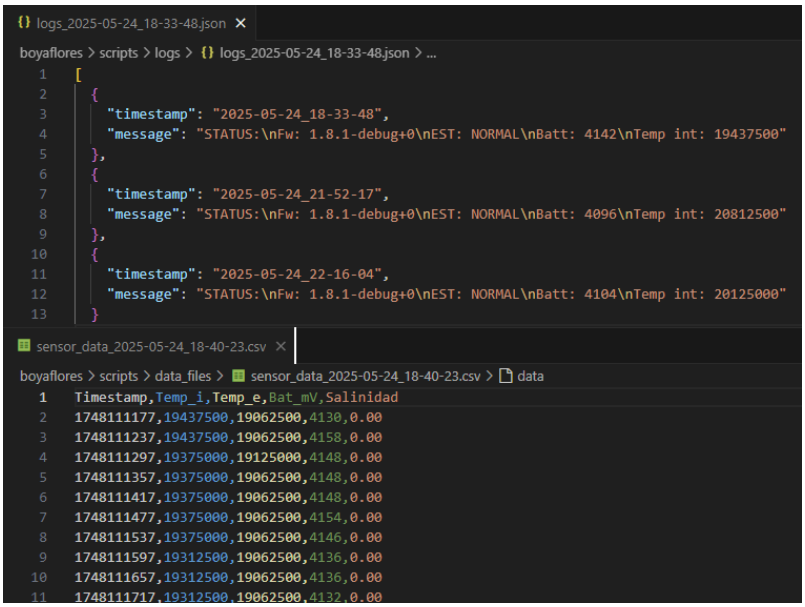


Figura 4.6: Archivos de logs y datos provenientes de la misma conexión MQTT pero creados en distinto tiempo.

La librería *paho-mqtt* también proporciona métodos para realizar publicaciones (*publish*) en diferentes tópicos, lo cual resulta especialmente útil cuando se desea extender la funcionalidad del sistema más allá de la simple recolección de datos. Dado que el *firmware* desarrollado permite la interacción con el dispositivo (por ejemplo, para modificar su estado con el fin de ahorrar energía o iniciar la descarga de una actualización) se aprovecharán las capacidades de esta librería para enviar comandos al tópico correspondiente. Esto permite interactuar con la placa directamente desde el código, facilitando así el control remoto y la gestión del dispositivo.

Por otra parte para el graficado de datos adquiridos y almacenamiento en los archivos **.csv** contenidos en el directorio *data_files/* se utiliza el *script* de Python llamado *graph_data.py*. Este código emplea las librerías Pandas [36] para importar los datos desde el archivo mientras que para graficarlos se utiliza la librería *Matplotlib* [19].

Interfaz de usuario

Para juntar la funcionalidad de los códigos previamente mencionados de recolección y graficado de datos se empleó un tercer *script* de Python el cuál unifica todo y provee al usuario con una interfaz definida en la cual mediante comandos se pueden realizar distintas operaciones. A este código se lo llamó ***wrapper*** o envoltorio, y su nombre es *mqtt_data_collector_wrapper.py*.

Cuando se activa la interfaz llamándolo desde la terminal, se despliega un menú de comandos disponibles (ver Figura 4.7) y el usuario tan solo deberá escribir el comando y presionar *enter* para ejecutarlo.

```
Comandos disponibles:
  c       → Conectar al broker MQTT
  q       → Desconectar del broker
  comando → Enviar comando al tópico boya/comandos (menú interactivo)
  plot    → Graficar la ultima data adquirida del dir /data_files/
              → Por defecto se grafican temperatura, bateria y salinidad (conductividad)
  plot --name filename.csv
          → Graficar la data de nombre filename.csv, puede combinarse con --show
  plot --show t,b,s
          → Elijo graficar temperatura, bateria y salinidad (conductividad), puede combinarse con --name
             → Opciones disponibles de --show: t,b,s
               (pueden usarse solo una letra o combinarse t,b t,s etc.)
  flush   → Borrar todos los archivos de sensor_data/ y logs/
  help    → Mostrar el menu de comandos
  quit    → Desconectar del broker y salir del programa
```

Figura 4.7: Menú desplegado al correr la interfaz de *mqtt_data_collector_wrapper.py*.

Estos comandos comprenden funcionalidades básicas como establecer y terminar la conexión con el *broker*, así como también otros tienen funcionalidades específicas dentro de la funcionalidad del graficado y almacenamiento de datos.

Para el caso del comando *plot* se puede seleccionar el archivo a ser graficado mediante el agregado de el argumento *–name* seguido del nombre del archivo a graficar. Esta bandera a su vez, es compatible con la de `--show` la cual permite solamente graficar los datos de interés, por ejemplo si quisiera graficar sólamente la temperatura y la conductividad y no es de interés la batería.

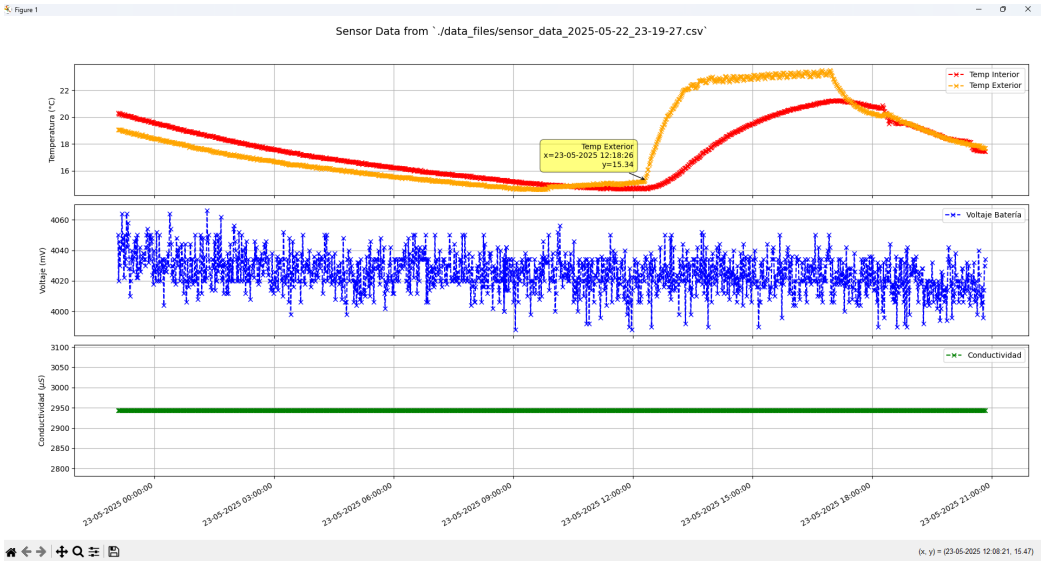


Figura 4.8: Ejemplo de figura desplegada por el comando **plot** sin argumento *–show* es decir mostrando la temperatura (arriba, en rojo la temperatura exterior y en naranja la interior), voltaje de la batería (en el medio en azul) y conductividad (abajo, en color verde).

En Figura 4.8 se puede apreciar en la esquina inferior izquierda que se tienen distintos íconos. El usuario puede interactuar con estos íconos los cuales cumplen distintas funciones en la ventana del gráfico, como ser guardar la imagen, hacer *zoom* en áreas de importancia en el gráfico e inclusive cambiar los anchos de los márgenes de las gráficas en la ventana. Otra funcionalidad interactiva es que al mover el cursor sobre las distintas gráficas se muestra un cuadro con el valor del punto para facilitar la lectura de los datos (ver en Figura 4.8 en temperatura en la mitad de la gráfica).

Por otra parte si el usuario precisa comunicarse con la placa para saber su *status* o modificar el estado actual a uno de ahorro de energía puede escribir `comando` en la terminal lo cual despliega un menú en el cuál se puede elegir el comando a enviar al tópico `boya/comandos` (ver Figura 4.9).

```
comando

Selecciona un comando para enviar a 'boya/comandos':
  0 - Regresar al menú principal
  1 - STATUS
  2 - DFU
  3 - TEST
  4 - RESET
  5 - REVERTIR
  6 - EST_NORMAL -> WIP
  7 - EST_LP -> WIP
  8 - EST_ULP -> WIP
  9 - EST_CONF -> WIP
  10 - CONFIRM_IMG -> WIP
  11 - MKFS
Elige una opción (0-10):
```

Figura 4.9: Menú desplegado tras poner comando en la terminal.

Ya que se posee un número limitado de comandos programados se decidió utilizar una interfaz con números como elección ya que de esta forma el usuario no tiene que recordar los *string* correspondientes a todos los comandos implementados sino que simplemente escribir el número correspondiente y presionar la tecla *enter*. El *script* no permite ingresar a este segundo menú si no hay una conexión activa con el *broker* y tampoco permite salir a menos que se seleccione una opción válida o se presione el número 0.

Una desventaja de esta la implementación es que si el usuario no sale del menú de comandos no se recibirán datos en esa ventana de tiempo por lo que se deberá procurar enviar el comando rápidamente o en su defecto regresar al menú principal.

En caso de que se desee agregar más comandos a la interfaz, simplemente deben añadirse al diccionario de Python denominado **commands**, el cual contiene los *strings* de los comandos definidos en el método *comando_menu* dentro del archivo *mqtt_data_collector_wrapper.py*.

Este diccionario está compuesto por *keys*, que corresponden a cadenas de texto que representan números. Estas claves son los valores que el usuario ingresará por terminal para seleccionar un comando. El valor asociado a cada clave es el *string* del comando que se enviará al *broker* `MQTT`.

Es importante destacar que las *keys* deben ser de tipo *string* o cadenas de texto y no de tipo entero, incluso si representan números. Esto se debe a que la función `input()` de Python, utilizada para leer desde la terminal, siempre devuelve un valor de tipo `str`, sin importar si el usuario ingresa un número.

A continuación se muestra un ejemplo de cómo agregar un nuevo comando al diccionario:

```
commands = {
    "0": "Regresar al menú principal",
    "1": "STATUS",
    "2": "DFU",
    "3": "TEST",
    ...
    "11": "MKFS",
    "12": "NUEVO_COMANDO"
}
```

Por último, en caso de que se desee eliminar los archivos ya procesados, el usuario puede llamar al comando *flush* el cuál borra todos los *logs* y datos recibidos. Este comando requiere una confirmación mediante terminal una vez invocado escribiendo la palabra *yes* para evitar borrados accidentales (Figura 4.10).

Figura 4.10: Petición de confirmación para el borrado de datos.

Por último se tiene un comando para terminar la conexión `MQTT` con el *broker* y salir del *script* de forma correcta sin necesidad de una interrupción forzosa (`Ctrl + C`).

4.4. Documentación orientada a próximos desarrolladores

Uno de los objetivos de este proyecto es que pueda ser reutilizado o al menos usado como base para otro proyecto a futuro, para esto es imperativo tener documentación necesaria para poder continuar con el desarrollo del proyecto o para la reutilización de alguno de sus módulos.

README

Primeramente, en el repositorio del proyecto se encuentra un archivo README.md [6] el cuál especifica los primeros pasos a seguir para los futuros desarrolladores en cuanto a la instalación de las herramientas de software utilizadas para el proyecto.

Entre esas herramientas se destaca el uso de las extensiones para *VScode* de *nRFConnect*, sumado a la aplicación *nRF Connect for desktop* [26] la cuál brinda manejo y rápido acceso a otras herramientas como *Cellular Monitor* (utilizada para monitorear las interacciones a nivel capa física del módem) o *Power Profiler* (utilizada en conjunto con el `PPK2` para tomar medidas de consumo de dispositivos *Nordic*).

En el archivo README.md se tiene una guía de desarrollo y políticas de versionamiento para el *firmware*, lo cual es importante para mantener la consistencia en las futuras versiones del código y la trazabilidad del desarrollo.

Este documento también especifíca el uso de las herramientas periféricas de desarrollo tales como los *scripts* explicados anteriormente en la sección 4.3 o cómo configurar una computadora como *runner* para ejecutar los *pipelines* de gitlab.

También se especifica el software utilizado durante el principio del proyecto para conectarse al *broker* `MQTT` para poder interactuar con la placa, ver los mensajes que está mandando e incluso descargarlos.

Wiki de GitLab

Además del archivo *README* previamente mencionado, el proyecto cuenta con una *wiki* [7] que incluye explicaciones más detalladas y modularizadas sobre el funcionamiento del *firmware* desarrollado. Entre los contenidos disponibles en esta *wiki*, se destaca una página dedicada exclusivamente a la arquitectura de los módulos y submódulos implementados. Esta sección ofrece descripciones más acotadas, concisas y orientadas a la práctica que las presentadas en la Sección 3.5, con el objetivo de facilitar una puesta en marcha rápida y eficaz, centrada en las estructuras e ideas principales del código.

Por otro lado, se incluyen tres tutoriales complementarios. El primero detalla el procedimiento para agregar nuevos comandos `MQTT` al conjunto de funcionalidades del *firmware* mientras que el segundo constituye una guía completa para la utilización del *script* de recolección de datos, el cual permite recibir y almacenar la información enviada por la placa de forma sencilla. El tercero y último tutorial corresponde a la creación e implementación de un *gitlab-runner* para compilación del proyecto.

Dado que los comandos `MQTT` son fundamentales para habilitar la comunicación remota con la placa y facilitar su control, es imprescindible disponer de documentación clara que sirva como referencia para futuros desarrolladores. Cabe destacar que el procedimiento para añadir estos comandos está estrechamente ligado a la arquitectura específica del *firmware* desarrollado, sin depender de librerías externas.

Respecto al tutorial del *script* de recolección de datos, se procuró dotar al proyecto de una solución accesible en cuanto a versatilidad para la recepción y almacenamiento de los datos provenientes del dispositivo, contribuyendo así a una mayor autonomía y facilidad de uso en el proceso de monitoreo.

Capítulo 5

Resultados

En este capítulo se presentan los resultados obtenidos de las pruebas de validación del sistema desarrollado, enfocándose en tres aspectos fundamentales: funcionalidad, consumo energético y *performance* de comunicación.

La validación funcional incluyó pruebas críticas de autonomía energética y se validaron los mecanismos de reconexión automática al *broker* `MQTT` tras pérdidas de conectividad.

En cuanto al análisis de consumo, se realizaron mediciones precisas utilizando equipamiento especializado de medición de corriente en los tres modos de operación definidos (`NORMAL`, `LOW_POWER` y `ULP`), empleando tanto el Power Profiler Kit II de Nordic como el analizador Otii para pruebas específicas.

Finalmente, se evaluó el *performance* de la señal `LTE` mediante mediciones de parámetros críticos como `SNR` (Signal to Noise Ratio), `RSRP` (Reference Signal Received Power) y `RSRQ` (Reference Signal Received Quality).

5.1. Validación funcional

Se diseñaron diversos escenarios de prueba para comprobar el funcionamiento del sistema diseñado según los objetivos planteados del proyecto.

5.1.1. Prueba de autonomía de 72 horas

Partiendo de una batería cargada pero no llena (su voltaje medido aproximado al inicio de la prueba era de $4{,}05V$) se simuló una obstrucción en el panel haciendo que este no reciba luz durante 72 horas para validar que el sistema tenga la autonomía suficiente para durar ese tiempo sólamente con la batería sin recibir carga.

La prueba fue exitosa ya que no sólo se cumplió el objetivo de las 72 horas, sino que se superó con holgura, manteniendo un nivel de carga significativo en la batería, asumiendo que el voltaje nominal de la misma es $3{,}7V$.

Asumiendo un ciclo de descarga estándar de una batería de ion de litio, el hecho de no haber bajado del voltaje nominal indicaría que se está lejos de valores de carga críticos.

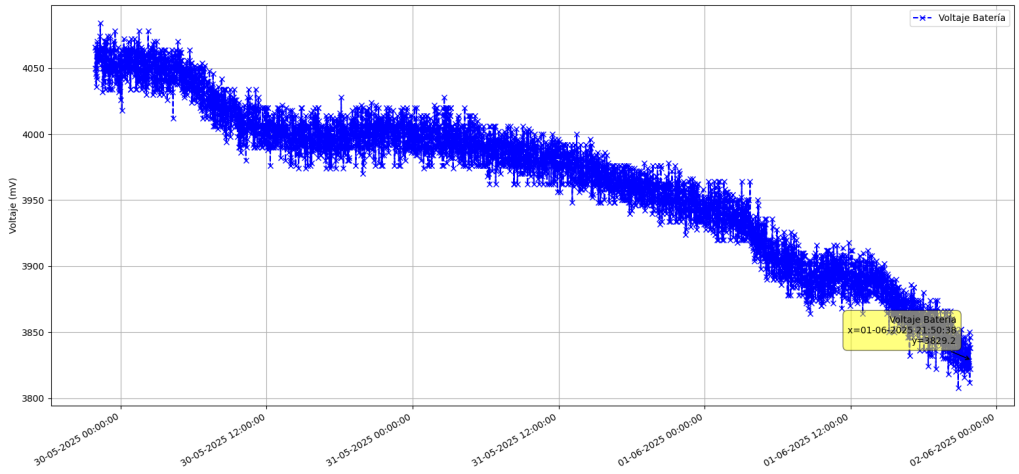


Figura 5.1: Datos de voltaje de batería que la placa mandó durante la prueba de 72 horas.

5.1.2. Reestablecimiento desde batería descargada

Partiendo de una batería que se encontraba en aproximadamente $2{,}75V$ (basándose en una curva estándar de descarga de una batería de ion de litio esto implicaría un nivel de carga muy bajo), se colocó el equipo al sol posicionando el panel solar correctamente (orientación norte y con un ángulo aproximado de 30°) en un día parcialmente nublado al mediodía.

El resultado de esta prueba fue exitoso, logrando en apenas 1 hora y 15 minutos subir el nivel de tensión de la batería hasta $3,56V$ (lo cuál implica modo `ULP`) y antes de cumplir las dos horas ya se había reestablecido el modo `NORMAL` de funcionamiento.

Como dato complementario sacado de la parte de descarga de esta prueba es que se logró obtener el voltaje mínimo de funcionamiento del sistema, el cuál es aproximadamente $3{,}26V$. Este fue el último voltaje muestreado antes de que el sistema se quedara sin energía.

Otro dato de importancia obtenido es que a pesar de haber permanecido durante más de 12 horas entre que se quedó sin energía y que fue puesto al sol, no se sufrió ningún desperfecto de *hardware* ni tampoco hubo una corrupción del sistema a nivel de *firmware* una vez recuperada la energía, lo cual demuestra la robustez del diseño del sistema ante interrupciones de energía, retomando su funcionamiento normal de conexión y muestreo sin inconvenientes.

5.1.3. Validación de reconexión automática

Durante el desarrollo del *firmware* se implementaron métodos de reconexión para poder reestablecer la comunicación con el *broker* en caso de que esta se pierda. Esta prueba fue realizada al notarse que estando en modo `ULP` luego de 12 horas de conexión el *broker* cortaba la conexión, por lo que se registró utilizando *logs* la desconexión y posterior reconexión.

```
[12:00:53.124,298] <wrn> mqtt: Thread MQTT
[12:00:58.129,516] <wrn> mqtt: Thread MQTT
[12:00:58.141,571] <dbg> mqtt: evt_handler: Cliente MQTT desconectado
[12:00:58.148,834] <err> mqtt: Error manteniendo conexion MQTT viva. -115
[12:00:58.162,994] <err> mqtt: Address info get fallo: -11
[12:00:58.169,555] <err> mqtt: Error inicializando broker.
[12:00:58.176,055] <err> mqtt: Error conectando al broker
[12:01:03.182,556] <wrn> mqtt: Thread MQTT
[12:01:03.187,683] <err> mqtt: Error manteniendo conexion MQTT viva. -128
```

```
[12:12:08.974,121] <wrn> mqtt: Thread MQTT
[12:12:08.979,248] <err> mqtt: Error manteniendo conexion MQTT viva. -128
[12:12:08.987,152] <err> mqtt: POLL no value
[12:12:16.035,888] <wrn> mqtt: Thread MQTT
[12:12:16.041,137] <dbg> mqtt: evt_handler: Cliente conectado a broker

[12:12:16.048,583] <inf> mqtt: Iniciando subscripcion a: boya/comandos
[12:12:21.056,610] <wrn> mqtt: Thread MQTT
[12:12:21.061,859] <dbg> mqtt: evt_handler: Subscrito al topico!

[12:12:26.068,817] <wrn> mqtt: Thread MQTT
[12:12:34.943,817] <inf> cellular_hal: RRC: Idle
[12:13:16.203,857] <inf> cellular_hal: RRC: Conectado
```

Figura 5.2: Arriba a las 12 horas de funcionamiento de la placa en `ULP` se pierde la conexión. Abajo 12 minutos después se logra reconectar al *broker*.

5.2. Análisis de Consumo Energético

5.2.1. Estimaciones preliminares

Previo a tener el development kit, se realizaron estimaciones de consumo en base a una herramienta online de Nordic llamada Online Power Profiler [32], la cuál permite tener la certeza del orden de magnitud de consumo en un ciclo de 10 minutos aproximadamente.

Para calcular un peor caso se desactivó la opción de `PSM` (Power Saving Mode) que ofrece Nordic, dejando sólamente `RRC` (Radio Resource Control) *idle mode* activado con configuraciones por defecto. Se activo la subida de datos con un periodo de transmisión de datos de 10 minutos con una carga de 100 bytes a modo de ejemplo.

Gracias a esta herramienta y los resultados que se obtuvieron de ella (ver Figura 5.3 y Figura 5.4) se compró con mayor seguridad el panel solar, la batería y el regulador, sabiendo que iban a estar en un rango aceptable para los objetivos propuestos.

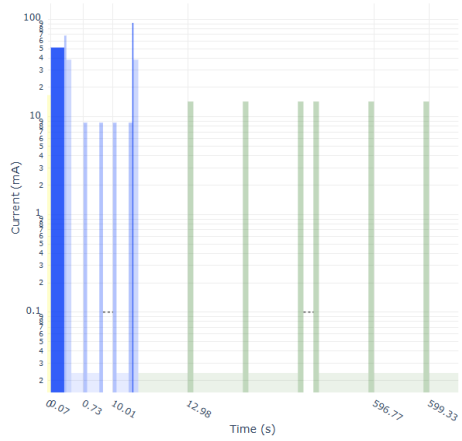


Figura 5.3: Grafica obtenida en el Online Power Profiler

General			
Chip	nRF9160 rev2	**Current consumption**	
Voltage	3.7		
RRC connected mode			
Data upload interval	10.00 min		
Data upload charge	2.94 mC	LTE event total charge	47.74 mC
cDRX average current	2.02 mA		
cDRX charge	20.08 mC		
Connection management charge	17.01 mC	iDRX idle current	655.50 µA
Total charge	19.96 mC		
RRC idle mode			
Time in iDRX	9.82 min		
iDRX average current	655.50 µA	Total average current	722.96 µA
Total charge	385.96 mC		

Figura 5.4: Tabla de consumo arrojada por el Online Power Profiler

Para sintetizar un breve análisis de los resultados obtenidos en esta fase inicial, se obtuvo que para un ciclo de 10 minutos se consumen aproximadamente en promedio 722,96μA. En caso que se tenga una batería con una capacidad de 1200mAH (número razonable y bastante conservador para una batería de litio de tipo 18650) se tendría una autonomía de casi 300 horas (276.6 horas o aproximadamente 11.5 días). De todas formas estos números fueron tomados con precaución, y sirvieron solamente para tener una estimación base a la hora de comprar el *hardware* de gestión de energía.

5.2.2. Equipamiento de medición

Para las mediciones de consumo reales se utilizaron dos equipos especializados que proporcionaron diferentes capacidades de análisis y validación cruzada de los resultados obtenidos.

Power Profiler Kit II

Para las mediciones de consumo reales se aprovechó de la oferta de la empresa Nordic la cuál brinda una placa para medición llamado Power Profiler Kit II [41]. Este *hardware* brinda una manera fácil y rápida de tomar el consumo de la nRF9160-dk, siendo la herramienta principal utilizada para la caracterización del comportamiento energético del sistema.

Además de proveer una manera simple para la medición de consumo la placa, se cuenta también con una interfaz gráfica llamada *Power Profiler*, capaz de graficar

el consumo y hacer ciertos cálculos como ser promedio de consumo y devolver fácilmente los picos del mismo.

También permite exportar los datos en formato `.csv` para su análisis en otros programas de tratamiento de datos y a su vez también permite exportarlos en formato `.ppk2` el cuál es un formato propietario de Nordic para utilizar dentro del software *Power Profiler*.

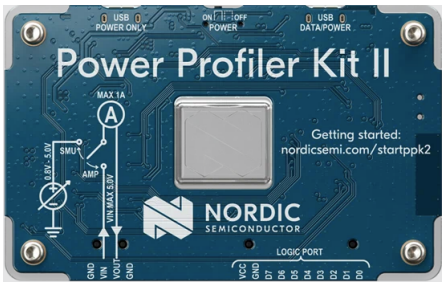


Figura 5.5: Power Profiler Kit II de Nordic

Analizador Otii

Complementariamente, para pruebas específicas y validación de mediciones críticas se utilizó el analizador Otii [34], un equipo especializado en análisis de consumo energético para dispositivos `IoT` (Internet of Things) y sistemas embebidos. El Otii ofrece capacidades avanzadas de medición con alta precisión y resolución temporal, permitiendo capturar tanto el comportamiento de consumo en régimen estacionario como los transitorios rápidos que pueden ocurrir durante las transmisiones `LTE`.

Este equipo se caracteriza por su capacidad de realizar mediciones simultáneas de voltaje y corriente con una alta frecuencia de muestreo, lo que resulta particularmente útil para analizar los picos de consumo durante las operaciones de transmisión del módem celular.

Además, cuenta con funcionalidades de análisis estadístico avanzado y capacidades de exportación de datos en múltiples formatos para análisis posterior.

La utilización de ambos equipos permitió obtener una caracterización más completa y confiable del comportamiento energético del sistema, proporcionando validación cruzada de los resultados obtenidos y mayor confianza en las mediciones realizadas.

Figura 5.6: Analizador Otii

5.2.3. Mediciones con Power Profiler Kit II

Configuración y pruebas preliminares

Una vez se pudo probar la funcionalidad de transmisión de datos al principio del proyecto se procedió a tomar medidas de consumo preliminares para corroborar el acierto en el orden de los componentes de batería y regulador.

Estas pruebas fueron realizadas con un peor caso de transmisión, con un período de envío de datos de 60 segundos en un lapso de tiempo de unos 10 minutos. El período de 60 segundos entre transmisiones fue tomado ya que era una frecuencia mayor a la necesaria para el proyecto, por lo que la media de consumo debería ser mayor a la media del funcionamiento final permitiendo realizar una medida de 10 ciclos de envío en solo 10 minutos. La elección fue suficiente para una aproximación del futuro consumo de la placa.

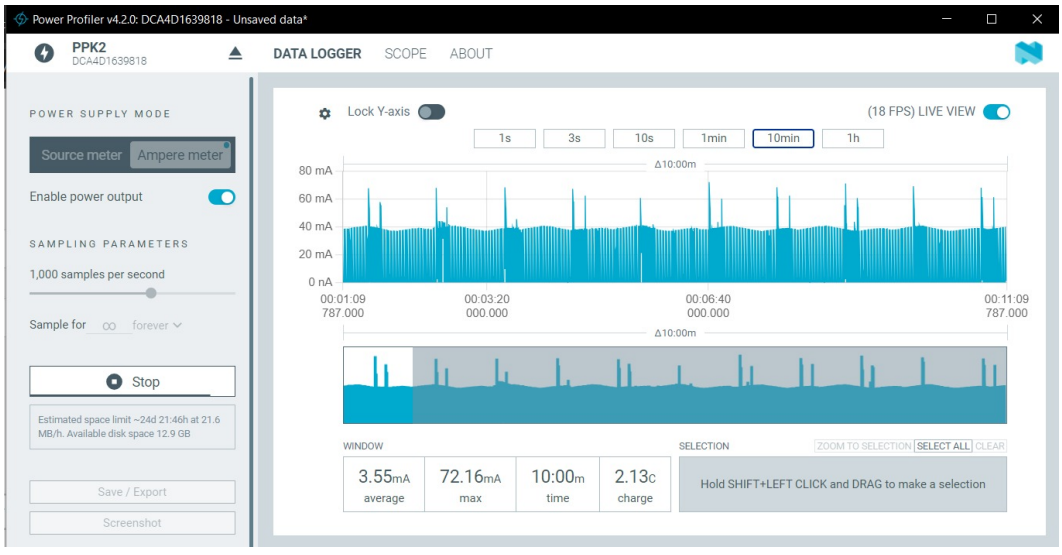


Figura 5.7: Medida preliminar tomada desde el Power Profiler, nótese que en la parte inferior se puede apreciar el promedio de consumo, el pico máximo, el tiempo transcurrido y la carga usada

Resultados por modo de operación

Se midió el consumo en los 3 (`NORMAL`, `LOW_POWER` y `ULP`) de los cuatro modos de funcionamiento definidos en Subsección 3.5.4, ya que no se espera que la placa permanezca en el modo restante ya que se usa en situaciones particulares y por un breve tiempo por lo que no vale la pena su testeo.

El hardware de medición (Power Profiler Kit II) fue utilizado en modo amperímetro [21] para las medidas, haciendo uso de los pines **P22 VDD_nRF'** y **P22 VDD_nRF** de medición que provee el development kit [43].

Las mediciones se hicieron en régimen estacionario, es decir, con la comunicación establecida y los sensores inicializados.

Se estimó que para la medida en el estado `NORMAL` con 5 ciclos de enviado de medidas (15 minutos entre envíos) alcanzaba para caracterizar el consumo a largo plazo en ese estado, es decir 80 minutos aproximadamente para dar cierto margen de holgura.

Para el modo de `LOW POWER` se realizó la misma medida en el mismo período (80 minutos) pero desfasados temporalmente desde la inicialización de la placa unos 200 minutos (es decir 2 horas y 20 minutos). Esto es para asegurar que se capten 3 mediciones de datos (recordando que el muestreo es cada 30 minutos en modo `LOW POWER` y en modo `ULP`) además de una transmisión (en modo `LOW POWER` se dan cada 4 horas).

Por último, para el modo `ULP`, se utilizó el mismo razonamiento que en el modo `LOW POWER`, tomando consumo durante 80 minutos solo que esta vez el desfasaje respecto a la inicialización fueron 15 minutos aproximadamente ya que esto permite captar 3 mediciones de datos y gracias a que en modo `ULP` no hay transmisión de datos no tenemos la limitante de esperar un envío.

Las mediciones de consumo se presentan a continuación en la Tabla 5.1:

Modo de consumo	**Corriente promedio consumida (mA)**	**Tiempo de medición (minutos)**	**Notas**
NORMAL	3.05	80	Frec. de muestreo 1 minuto. Frec. de transmisión 15 minutos.
LOW POWER	2.93	80	Frec. de muestreo 30 minutos. Frec. de transmisión 240 minutos. Se capturó una transmisión.
ULP	2.92	80	Frec. de muestreo 30 minutos No hay transmisión.

Tabla 5.1: Tabla de consumo de Energía

La poca diferencia en los valores de los distintos modos de funcionamiento apreciados en la Tabla 5.1 se debe a que la principal fuente de consumo energético se da en la conexión celular. Si bien la transmisión y el muestreo afectan, la actividad en el módem varía casi de la misma manera en los 3 estados de manera cíclica (por el `eDRX`), ya que no se apaga el módem por completo en ninguno de los estados.

Otras medidas de interés

Como prueba de impacto en el consumo energético que tiene el modo `eDRX` se realizó una medida de 20 minutos en modo `NORMAL`.

En la imagen Figura 5.8, puede observarse picos de consumo entre los ciclos de muestreo y paginación `LTE`, indicando mayor actividad en el módem.

En contraste, la imagen Figura 5.9 se observa que entre ciclos de paginación y muestreo casi no se registra actividad.

Esto se traduce en una diferencia significativa de consumo promedio cuando no se tiene `eDRX` (de $3,22mA$) contra los $2,89mA$ cuando sí se tiene activado.

Figura 5.8: Medida de consumo sin `eDRX` activado.

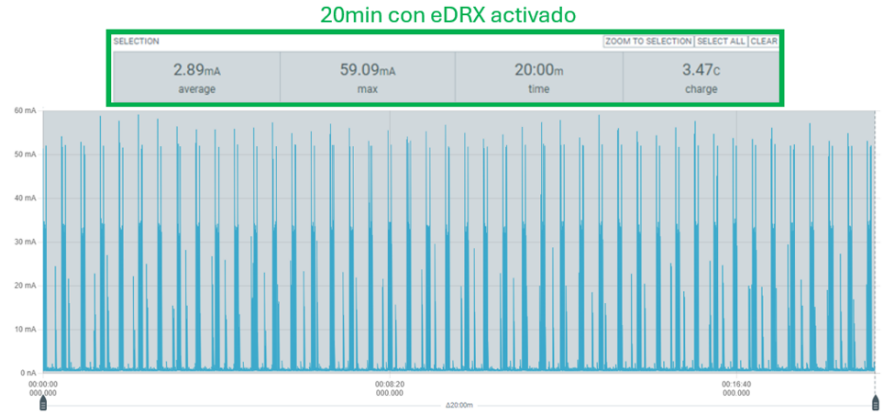


Figura 5.9: Medida de consumo con `eDRX` activado.

Otra mejora relacionada con la optimización del consumo energético se implementó en el sensor de salinidad. Este sensor, por defecto, opera en modo de muestreo continuo (una medición por segundo) y no entra automáticamente en modo *sleep*. Debido a su elevado consumo en comparación con el resto del sistema, su funcionamiento constante impactaba significativamente en el consumo general, como se observa en la Tabla 5.2:

Modo de operación	Consumo promedio (mA)	Notas
Sin modo *sleep* implementado	6,83	Medida de 30 minutos en modo `NORMAL`
Con modo *sleep* implementado	3,05	Tomado de Sección 5.2.3

Tabla 5.2: Comparación del consumo de corriente con y sin modo *sleep* en el sensor.

5.2.4. Mediciones con Analizador Otii

Configuración de conexión

La conexión se realizó siguiendo las recomendaciones de Nordic para mediciones de corriente cuando se utiliza un instrumento de tipo `SMU` (Source Measure Unit) en el nRF9160-dk [17]. Siguiendo las instrucciones brindadas, se conectó terminal negativo del el Otii al pin negativo del *header External Supply* de la placa, y luego en el terminal **VDD_nRF** del *header* **P22**. Se conectó el cable USB a la placa (ver Figura 5.10) ya que este alimenta al *debugger*, el cual no debe ser tenido en cuenta para la medición de consumo, ya que cuando la placa se encuentre en actividad este no estará habilitado.

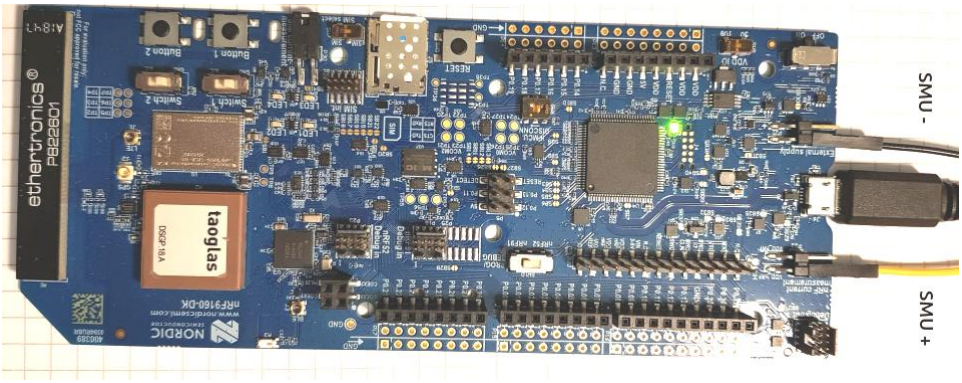


Figura 5.10: Esquema de conexión en el que fue utilizado el Otii. Tomada de [17].

La configuración del Otii se estableció con los siguientes parámetros:

- Voltaje de alimentación: $3{,}75V$ (simulando el voltaje nominal de la batería)
- Frecuencia de muestreo: $4kHz$ para capturar adecuadamente los picos de transmisión

Análisis detallado por actividades

Se realizaron mediciones en puntos específicos de interés con el objetivo de caracterizar el equipo con mayor precisión que la obtenida mediante el uso del Power Profiler Kit II. Posteriormente, se comparan los resultados obtenidos, lo que permite validar las mediciones de consumo realizadas con dicho instrumento. Cabe destacar que el Power Profiler Kit II presenta un error del $10\,\%$ (como se observa en la documentación oficial [40]), mientras que el analizador Otii tiene un error del $0,1\,\%$ (como se detalla en la documentación oficial [33]). En la Figura 5.11

se observa el consumo medido con el analizador Otii, cuyo valor es de $3,03mA$. Este resultado es prácticamente idéntico al obtenido con el Power Profiler Kit II, que registró un consumo de $3,05mA$ en estado normal. La elección de la ventana de tiempo utilizada para estas mediciones se detalla más adelante.

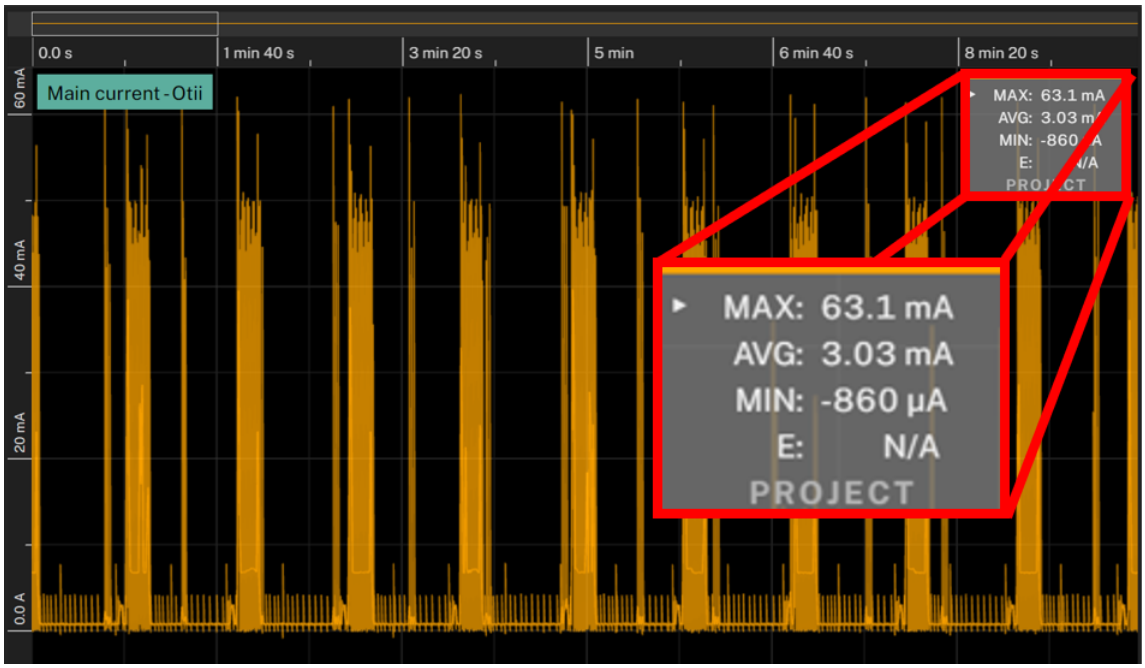


Figura 5.11: Consumo medido con Otii. **Nota:** La ventana de tiempo de la medición abarca la inicialización del dispositivo por lo que el valor mínimo mostrado en la gráfica no es representativo.

Los consumos más representativos del sistema se analizan en el estado `NORMAL`, ya que en este se presentan tanto las mediciones como las transmisiones con mayor frecuencia. En este estado, se realiza una medición cada 1 minuto y una transmisión cada 15 minutos. Además, se pueden observar eventos de *paging*, que ocurren aproximadamente cada 40 segundos. Entre estos eventos activos, el sistema permanece en reposo, en un modo de bajo consumo. Este consumo en reposo constituye una parte importante del promedio general y debe ser considerado para obtener una caracterización completa del comportamiento energético.

Para obtener una estimación precisa del consumo energético, se definió una ventana de observación de 30 minutos. Este período permite capturar múltiples ocurrencias de todas las fases relevantes del funcionamiento del sistema: medición, transmisión, *paging* y reposo. La periodicidad de los eventos de *paging* puede observarse en la Figura 5.13, mientras que la frecuencia de las mediciones se muestra en la Figura 5.12. Esta duración garantiza que todas las situaciones ocurran al menos una vez, permitiendo así calcular un promedio de consumo que sea representativo.

De este modo, se garantiza un muestreo de los distintos momentos de actividad del sistema, lo que permite calcular no solo un promedio general de consumo, sino también estimaciones específicas para cada una de las fases consideradas.

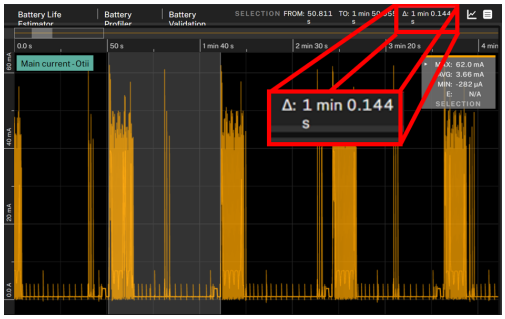


Figura 5.12: Tiempo entre medidas en modo *NORMAL*.

Figura 5.13: Tiempo entre paginaciones LTE.

A partir de las mediciones realizadas durante la ventana de observación de 30 minutos dentro del estado normal, se obtuvieron los valores específicos de consumo para cada actividad.

Como se observa en la Figura 5.14, el consumo en modo de ahorro de energía es de $827\mu A$. La Figura 5.15 muestra que durante las operaciones de medición el consumo alcanza $10,3mA$, mientras que la Figura 5.16 revela un consumo de $1,53mA$ durante los eventos de paginación. El mayor consumo se observa en la Figura 5.17, donde las transmisiones de datos alcanzan $12,5mA$.

La Tabla 5.3 compara los valores obtenidos con ambos instrumentos:

Actividad	**PPK II (mA)**	**Otii (mA)**	**Diferencia (%)**
Ahorro de energía (idle)	0,732	0,827	13,0
Paginación	1,420	1,530	7,4
Realizando medición	11,43	10,3	9,9
Transmitiendo datos	12,22	12,5	2,3
Promedio estado NORMAL	**3,05**	**3,03**	**0,7**

Tabla 5.3: Comparación de consumos entre Power Profiler Kit II y Otii

La comparación muestra una buena correlación entre ambos instrumentos, con diferencias que oscilan entre $2,3\,\%$ y $13\,\%$ para las diferentes actividades. El consumo promedio en estado normal muestra una excelente concordancia con menos del $1\,\%$ de diferencia, validando la precisión de ambas mediciones.

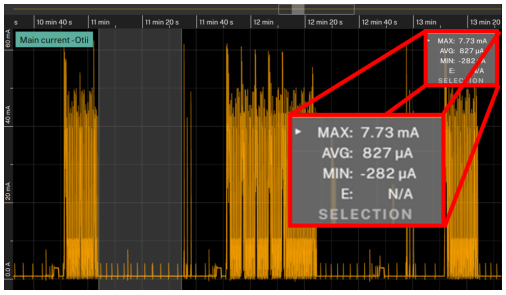


Figura 5.14: Consumo en ahorro de energía

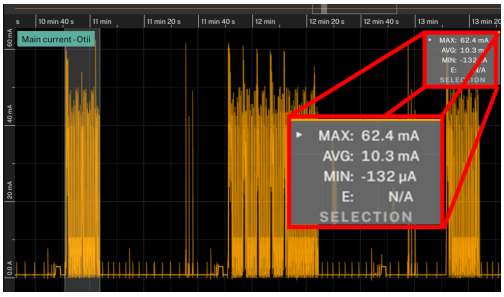


Figura 5.15: Consumo realizando una medida

Figura 5.16: Consumo durante una paginación

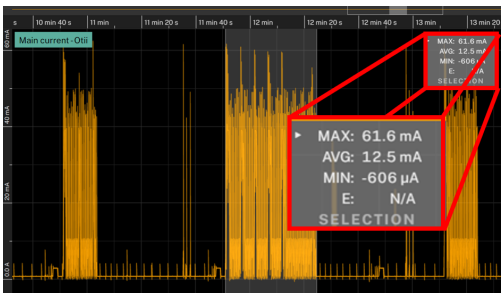


Figura 5.17: Consumo durante una transmisión

Las mediciones realizadas con el Otii validan y complementan los resultados obtenidos con el Power Profiler Kit II, proporcionando una caracterización más detallada del comportamiento energético del sistema.

Validación cruzada de resultados

La validación cruzada entre ambos equipos de medición mostró una correlación excelente en los valores promedio de consumo, con diferencias menores al 15 % en todas las medidas. Esta consistencia en los resultados proporciona alta confianza en la caracterización energética realizada. Los datos obtenidos confirman que el sistema cumple con los requerimientos de autonomía establecidos y valida las decisiones de diseño tomadas en cuanto a la selección de batería y panel solar.

5.2.5. Pruebas en condiciones de señal atenuada

Metodología experimental

Con el objetivo de simular una condición de baja señal celular, similar a la que podría encontrarse en la instalación final, se realizó una prueba colocando el equipo dentro de un contenedor metálico. Este tipo de material genera una atenuación significativa de la señal, lo cual permite analizar el comportamiento del consumo energético del sistema ante dificultades en la comunicación con la red celular. El setup experimental se muestra en las figuras Figura 5.18, Figura 5.19 y Figura 5.20, donde se puede observar la progresión desde el contenedor abierto

hasta la configuración de máxima atenuación con el contenedor completamente cerrado. La Figura 5.21 muestra el setup completo de medición con el analizador Otii conectado al equipo funcionando dentro del contenedor metálico, permitiendo el monitoreo continuo del consumo energético durante toda la prueba.

Figura 5.18: Vista del contenedor metálico abierto, mostrando su estructura

Figura 5.19: Equipo instalado dentro del contenedor

Figura 5.20: Contenedor completamente cerrado, simulando la situación de máxima atenuación

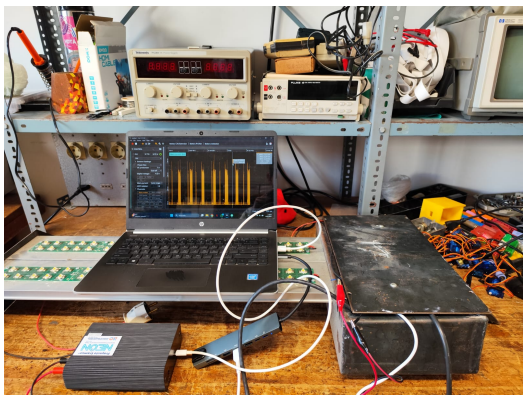


Figura 5.21: Prueba del equipo midiendo dentro del contenedor

Resultados de consumo energético

Durante esta prueba se midió el consumo de energía en distintos momentos del ciclo de funcionamiento del equipo, particularmente durante los intentos de conexión celular y las transmisiones de datos. Para ello se utilizó el analizador Otii configurado según el procedimiento descrito en la sección anterior, manteniendo los mismos parámetros de medición para garantizar la correcta comparación de los resultados. Los resultados obtenidos muestran valores específicos que confirman la estabilidad energética del sistema. Como se observa en la Figura 5.22, el consumo durante los eventos de transmisión alcanza $12,5mA$, valor que se mantiene consistente con las mediciones realizadas en condiciones normales de señal presentadas en la sección 5.3. Por otro lado, la Figura 5.23 revela un consumo promedio durante el ciclo completo de operación de $3,15mA$.

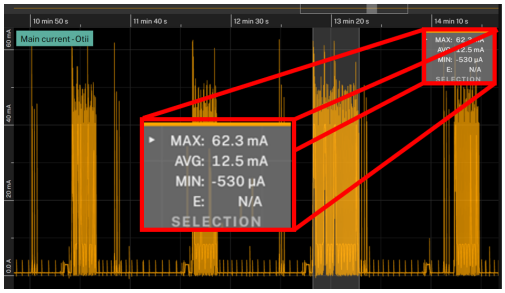


Figura 5.22: Consumo de la transmisión con la señal atenuada

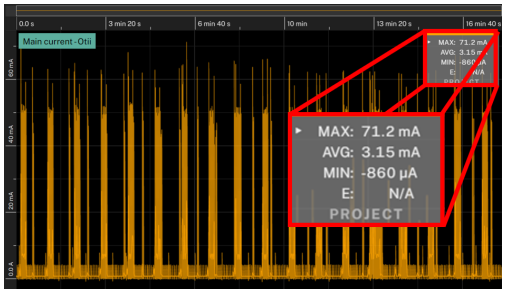


Figura 5.23: Consumo con la señal atenuada

Interpretación de resultados y comparación

Al comparar estos valores con los obtenidos en condiciones normales de señal, se observa una notable consistencia en el comportamiento energético del sistema. El consumo durante las transmisiones de $12,5mA$ en condiciones de señal atenuada es idéntico al valor de $12,5mA$ registrado con el analizador Otii en condiciones normales (ver Tabla 5.3), lo que indica que el módem `LTE` mantiene un nivel de potencia de transmisión estable independientemente de la calidad de la señal recibida. El consumo promedio de $3,15mA$ en condiciones de señal atenuada muestra únicamente una ligera variación respecto al valor de $3,03mA$ obtenido en condiciones normales, representando una diferencia de apenas $4\,\%$ que se encuentra dentro del margen de error esperado para este tipo de mediciones. La estabilidad del consumo energético observada valida que el sistema mantendrá su autonomía energética planificada aún en las condiciones más desfavorables de señal celular que puedan presentarse en la aplicación real, puesto que se obtendrá una duración de la batería de $\frac{3200mAh}{3,15mA} \approx 42$ días. Esto confirma la robustez del diseño y la adecuada selección de los componentes de gestión de energía.

5.3. Evaluación de Performance de Comunicación

Mediciones de calidad de señal LTE

Para la medición de calidad se utilizó el software provisto por Nordic llamado *Cellular Monitor*, el cuál tiene *firmware* especializado precargado para monitorear mediante comandos AT el módem LTE de la placa de manera fácil y rápida, con una interfaz gráfica para su fácil comprensión.

La antena que se utilizó para tomar las mediciones es la incluida por el *development kit*, la cuál es una antena omnidireccional.

Como base para luego ir a medir a la isla, se tomó una medida de referencia en la Facultad de Ingeniería Eléctrica en Montevideo, dando como resultado los siguientes valores presentados en la siguiente imagen:

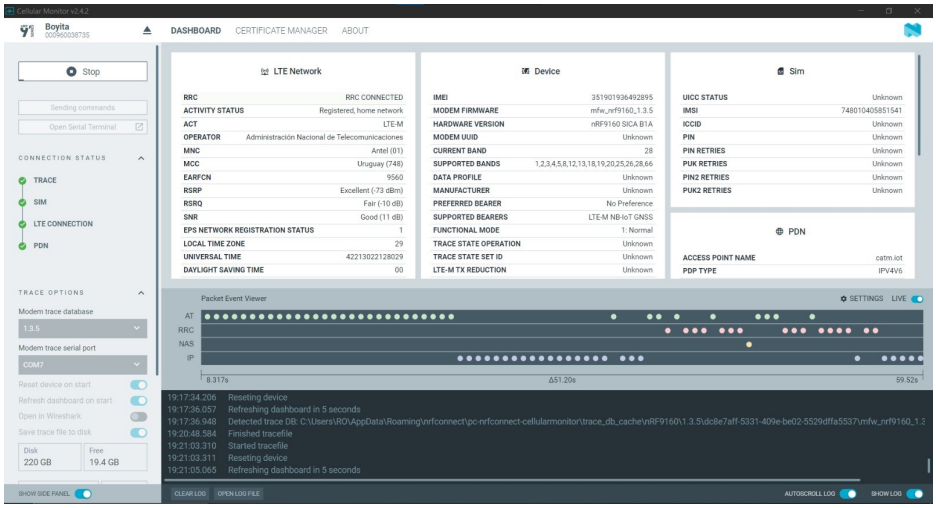


Figura 5.24: Medida de referencia de señal LTE tomada desde facultad de ingeniería.

Los valores fundamentales para saber la calidad y fuerza de la señal que llega son los dados por los parámetros: SNR (Signal to Noise Ratio) que dice cuál es la relación señal a ruido de lo que llega a la placa expresado en dB, RSRP (Reference Signal Received Power) lo que expresa la potencia media de las señales de referencia de LTE medida en dBm y por último RSRQ (Reference Signal Received Quality) la cuál da la pauta de la calidad de aquellas señales de referencia de LTE medidas en dB.

Comparativa: Facultad vs Isla

	Medida en Facultad de Ingeniería	**Medida en la Isla**
SNR (dB)	11	Unknown (Error al medir)
RSRP (dBm)	-73	-109
RSRQ (dB)	-10	-15.5

Tabla 5.4: Mediciones de Parámetros de Señal LTE

Consideraciones para mejora de antena

Se concluyó que la antena omnidireccional que posee el *development kit* no es suficiente para tener una señal que permita una conexión relativamente estable estando en la isla, por lo que si se quisiera establecer en la isla el proyecto se deberá instalar una antena direccional y enfocar su patrón de radiación hacia la costa. Estas antenas direccionales tienen un parámetro característico llamado *dBi* (decibels relative to isotropic) el cuál es una medida de cuánto más ganancia tiene la antena direccional comparado contra una antena isotrópica ideal, con las antenas de plaza teniendo una ganancia de aproximadamente *+11 dBi* lo que supondría más de 10 veces la ganancia sobre la antena omnidireccional del *development kit* (en la dirección del lóbulo principal).

Si se quisiera sustituir la antena por una direccional el fabricante del *development kit* Nordic da especificaciones mínimas las cuales están dadas por la siguiente Tabla 5.5:

Parametro	**Requerimiento**
VSWR (Voltage Standing Wave Ratio)	$< 3:1$
RL (Return Loss)	$> 6{,}0dB$
Efficiency	$> 50\,\%$
Minimum power handling	1 W

Tabla 5.5: Requerimientos de la antena externa, todas las condiciones de están dadas para una impedancia de 50Ω. Tomada de [4].

Además de estas características, la interfaz con la placa precisa de un adaptador para llevar el conector coaxial de la placa a uno `SMA` para conectar con la antena.

Esto supone un trabajo más profundo de mediciones de radiofrecuencia, de evaluación e investigación de antenas además de suponer otro dispositivo a montar, lo cuál quedó por fuera del alcance de este proyecto.

Capítulo 6

Conclusiones y trabajo a futuro

6.1. Conclusiones

El presente trabajo logró desarrollar exitosamente una nueva versión del adquisidor de datos para monitoreo ambiental, superando las limitaciones identificadas en el diseño previo. La migración hacia la plataforma Nordic nRF9160-DK resultó en mejoras significativas en múltiples aspectos críticos del sistema.

6.1.1. Logros principales del proyecto

Eficiencia energética mejorada

Se logró una reducción significativa del consumo energético, pasando de un sistema con consumo del orden de los amperes (típicamente $1 - 2A$ en operación normal) a un consumo de $3,05mA$ en modo de funcionamiento `NORMAL`, representando una reducción de aproximadamente tres órdenes de magnitud respecto al diseño anterior. Esta gran optimización permitió diseñar un sistema significativamente más compacto y liviano, viable para alimentación autónoma con paneles solares pequeños y baterías de capacidad reducida. La caracterización de los diferentes estados de funcionamiento mediante equipamiento especializado validó la solidez del diseño de bajo consumo.

Autonomía energética

La integración de un sistema de alimentación autónomo basado en panel solar compacto y batería de ion de litio tipo 18650 fue validada mediante pruebas prolongadas. Según los cálculos teóricos vistos en la Sección 5.2.5, el sistema podría funcionar continuamente durante aproximadamente 42 días con una batería de $3200mAh$ y un consumo promedio de $3,15mA$. Asimismo, como se ve en la Subsección 5.1.1, el sistema demuestra funcionar de manera continua durante 72 horas, confirmando su robustez ante escenarios de baja disponibilidad solar. De esta manera, se confirma que los requerimientos planteados son ampliamente superados.

Robustez operativa

Se implementaron y verificaron mecanismos de reconexión automática ante pérdida de conectividad `LTE`, así como el correcto restablecimiento desde estados de batería descargada. Estas funcionalidades aseguran la continuidad operativa en entornos remotos y condiciones adversas.

Estabilidad del sistema

Incluso en condiciones adversas de señal celular, el sistema mostró comportamiento estable con consumo energético prácticamente inalterado, evidenciando una adecuada selección de componentes y arquitectura robusta.

6.2. Trabajo a futuro

6.2.1. Hardware

Usando este proyecto como base, en un futuro es posible reutilizar y adaptar fácilmente el código desarrollado a otro *hardware*, gracias a las prestaciones de abstracción y portabilidad brindadas por Zephyr. Además, la plataforma `nRF91` de Nordic ofrece la flexibilidad de disponer de otros kits de desarrollo, con prestaciones distintas al *development kit* elegido como lo puede ser la línea `Thingy:91 X` [53]. En caso de optar por la línea `Thingy:91 X` para continuar con el proyecto, será necesario cambiar el sensor de temperatura por uno que se adapte mejor a la interfaz de la placa, la cual no soporta de forma nativa el protocolo *1-Wire* pero sí `I2C`. Más aún, en el caso del sensor de salinidad y en caso de utilizar el mismo circuito de adquisición de medidas, se debería cambiar su modo de funcionamiento de `UART` a `I2C` para que los `GPIOs` sean suficientes para acaparar los periféricos.

Fabricación personalizada del hardware

Actualmente, el sistema utiliza un *development kit* comercial (`nRF9160-DK`), lo cual es ideal para un prototipo, pero presenta limitaciones cuando se piensa en una implementación a mayor escala o en un despliegue prolongado en campo. El siguiente paso en la evolución del dispositivo es diseñar una placa de circuito impreso (`PCB`) propia, adaptada específicamente a las necesidades del proyecto. Los beneficios de una solución hecha a medida son múltiples:

- Reducción del tamaño y peso: se eliminan componentes innecesarios del *development kit* y se integra solamente lo estrictamente requerido.
- Aumento en la confiabilidad: una solución compacta y dedicada es más robusta frente a vibraciones, humedad y corrosión que un sistema armado con múltiples módulos conectados por pines o jumpers.
- Integración completa: se puede incluir en la misma `PCB` el cargador de batería, el regulador de voltaje, las conexiones a sensores, la interfaz del panel

solar y cualquier otra lógica auxiliar, reduciendo la cantidad de cables y puntos de falla.

Para abordar esta tarea será necesario definir el *layout* esquemático y la distribución de pistas de la PCB en software especializado. Este trabajo, si bien queda fuera del alcance del presente proyecto, representa un camino claro hacia la confiabilidad y la escalabilidad del sistema, siendo clave para cualquier despliegue como red de sensores inalámbricos o producción en serie.

Reducción de consumo y alternativa de alimentación

Según los consumos teóricos mínimos de los componentes del dispositivo (sensor de salinidad: $0,4mA$ en reposo, nRF9160 $< 10\mu A$ en estados de ahorro profundo de energía [39]) se tiene un límite teórico de consumo mínimo de $0,6mA$ estimados. Un consumo similar a uno tan reducido en conjunto con una batería de mayor capacidad podría suponer la posibilidad de prescindir del panel solar y su circuito de carga.

Los valores de consumo caracterizados en la Sección 5.2 abren la posibilidad de considerar una estrategia de alimentación alternativa basada exclusivamente en baterías, prescindiendo del panel solar. Los niveles de corriente promedio obtenidos, particularmente en los modos de menor consumo, sugieren que el sistema podría operar durante períodos prolongados utilizando únicamente la energía proporcionada por la batería.

Recurriendo nuevamente a la hoja de datos de la batería Panasonic NCR18650B, con un valor nominal de capacidad de $3200mAh$ podría alimentar (teóricamente) al sistema en modo NORMAL de manera continua durante aproximadamente 42 días (asumiendo el peor caso de consumo promedio registrado de $3,15mA$ tomado de Tabla 5.1). Esta duración se asocia a cronogramas típicos de mantenimiento en el sistema (calibración de sensores, limpieza de equipos, verificación de integridad mecánica), donde las visitas de campo suelen ser mensuales. La implementación de esta estrategia permitiría simplificar significativamente el diseño del sistema, eliminando la complejidad asociada al panel solar y el regulador de carga. Prescindir del panel solar y el circuito de carga del dispositivo supondría un costo mucho menor por unidad (en torno a 50 dólares menos por unidad).

Adicionalmente, se podría evaluar la incorporación de baterías de mayor capacidad, como celdas de $10000mAh$ a $15000mAh$, que extenderían la autonomía a 4-6 meses, reduciendo aún más la frecuencia de intervenciones. El reemplazo de baterías durante las tareas rutinarias de mantenimiento optimizaría los recursos, sincronizando la duración de la batería con los períodos de mantenimiento de los sensores. De esta forma, con baterías recargables, un operario podría cambiarlas cada vez que se realiza mantenimiento sobre los dispositivos.

Antena direccional externa

Una limitante clara fue la calidad de señal LTE en ubicaciones remotas, como la Isla de Flores. La antena omnidireccional incluida en el *development kit* de Nordic,

si bien funcional en entornos urbanos, no proporciona la ganancia necesaria para mantener una conexión confiable en contextos de baja cobertura. Una solución concreta consiste en el reemplazo de la antena actual por una antena direccional de alta ganancia. Este tipo de antena permite concentrar la energía de transmisión y recepción en una dirección específica, lo cual mejora significativamente los parámetros críticos de comunicación como el `RSRP` y el `SNR`. La implementación de esta mejora requerirá abordar varios desafíos:

- Montaje seguro y duradero: la antena deberá instalarse en una estructura que soporte condiciones de viento y humedad sin comprometer la orientación ni la integridad del sistema.
- Interfaz física: será necesario incorporar un adaptador `SMA` [24] y realizar una modificación mecánica del gabinete para garantizar una salida estanca del cable coaxial hacia el exterior.
- Orientación precisa: el patrón de radiación estrecho de una antena direccional exige una alineación cuidadosa hacia la torre de telecomunicaciones más cercana.
- Selección adecuada de la antena: asegurando que cumpla con los parámetros de impedancia, `VSWR` y potencia mínima definidos por Nordic.

Esta mejora aumentaría considerablemente la robustez del sistema frente a variaciones de señal celular, permitiendo operar con mayor estabilidad en locaciones aisladas, donde se prevé instalar dispositivos como este a futuro.

Un ejemplo de antena a ser evaluada a futuro puede ser la Tupavco TP545 [54], la cual cumple con los requerimientos provistos por Nordic, tiene una ganancia de *11dBi* y además fue diseñada para uso en exteriores lo que la hace una gran contendiente a ser utilizada. En cuanto a la conexión con el *development kit*, Nordic recomienda el uso del adaptador MXHS83QE3000 [24] para pasar de conector coaxial a `SMA`. Este conector coaxial tiene un interruptor integrado que desconecta la antena del *development kit* y conecta la radio del `SiP` al `SMA` automáticamente cuando es utilizado, sin necesidad de implementar cambios a nivel de *hardware* ni de *firmware*.

6.2.2. Firmware

Durante el transcurso del proyecto, la constante investigación dio lugar a mejoras que lograron bajar substancialmente el consumo de la placa como ser el uso de `eDRX` o la implementación de modo *sleep* para el sensor de salinidad. Sin embargo, Zephyr y en particular la plataforma `nRF9160` da lugar a más mejoras en ese aspecto.

Una de esas mejoras a realizar sería validar el modo `PSM` implementado en la última versión del *firmware*, ya que este provee un menor consumo comparado con `eDRX`. La implementación de `PSM` implica manejar la discontinuidad de conexión a partir de las capas altas del *stack* de comunicación ya que los mensajes enviados mientras el dispositivo se encuentra en la etapa inactiva del modem no serán

recibidos por el dispositivo. Una vía implementada en conjunto con la versión de *firmware* experimental es que el servidor debe enviar los comandos a través de mensajes `MQTT` retenidos. Debido a que este mecanismo fue implementado luego de las validaciones documentadas y debido a la incapacidad de realizar un análisis acorde al considerado necesario para determinar una versión como estable por causa de limitaciones de tiempo, se decidió publicar y documentar esta versión como parte de un prototipo futuro en pos del espíritu de desarrollo vivo que defiende este proyecto.
Otro aspecto del *firmware* a mejorar tiene que ver con la variabilidad entre medidas del `ADC`. Se implementó (pero no se validó) una mejora en una *branch* nueva en el repositorio del proyecto que comprende el muestreo y promediado de varias medidas consecutivas del `ADC` para tratar de mitigar lo más posible esas diferencias en los valores obtenidos.

Para facilitar el *testing* y la medición en los distintos estados de consumo de la placa, se deberá terminar la implementación los comandos de cambio de modo para no tener que *flashearlos* directamente del *artifact* generado por el *pipeline* como se explicó en Sección 4.2.

Por último, en caso de que se quiera escalar este proyecto a múltiples dispositivos, se tienen dos caminos a seguir, el primero es tener un *firmware* independiente en cada unidad, en donde cada una se suscribe a un tópico de comandos distinto en el *broker* (por ejemplo *comandos1*, *comandos2*, etc.) y luego deja como responsabilidad al usuario mandar el comando al tópico correspondiente. Esto implicaría modificar el *pipeline* implementado y agregar más *buckets* o en su defecto, tener la capacidad de subir distintos archivos de actualización en un mismo *bucket*, de todas maneras no debería tomar demasiado tiempo de desarrollo para su implementación.

La segunda manera de obtener esta escalabilidad en los comandos sería mediante una revisión completa de la implementación, esto implicaría un mayor tiempo de desarrollo pero se podrá dotar con métodos para distinguir entre dispositivos. Un ejemplo de esto puede ser averiguar mediante peticiones al módem el identificador único `IMEI` (International Mobile Equipment Identity por sus siglas en inglés) y en base a él poder distinguir desde que *bucket* o desde cual archivo del *bucket* se va a descargar la actualización.

6.2.3. Herramientas complementarias

Una de las principales mejoras a implementar respecto a las herramientas complementarias utilizadas durante el proyecto tiene que ver con el *pipeline*, más precisamente con los *runner*. Durante el desarrollo, se utilizaron como *runners* las computadoras personales de los integrantes del equipo, lo cual resulta poco conveniente, ya que requiere activación manual y no hay disponibilidad continua del servicio. Idealmente el *runner* no debería ser un equipo particular sino que debería estar alojado en un servidor dedicado, de esta forma no se ocupan equipos que pueden ser usados para tareas de desarrollo y se tiene asegurada la disponibilidad de

al menos un *runner* en cualquier a cualquier hora del día.

Otro aspecto a mejorar es la parte relacionada con la recolección y el almacenamiento de datos. Al igual que con los *runners*, la recolección de datos fue realizada utilizando equipos personales, y sería de gran utilidad tener el *script* corriendo en un servidor. Además de la ventaja ya mencionada respecto a la disponibilidad continua del servicio, los servidores usualmente cuentan con respaldo energético y ofrecen la posibilidad de escalar el almacenamiento, permitiendo un manejo creciente de datos.

Contar con un servidor desplegado en también permitiría mejorar el *script* de recolección de datos implementando el acceso a los datos de manera remota, facilitando la lectura y consulta de estado de la placa. Además, se podría llegar a implementar una interfaz gráfica de tipo *dashboard* (por ejemplo utilizando la librería *open source* de Python llamada *Streamlit* [52]), para que usuarios no técnicos tengan la posibilidad de acceder y visualizar los datos recolectados de manera más intuitiva y fácil.

6.2.4. Validación y despliegue

Validación a largo plazo

Si bien se realizaron pruebas para verificar que el dispositivo desarrollado es capaz de funcionar de forma ininterrumpida por largos períodos de tiempo, es necesario confirmar dicha hipótesis mediante pruebas prolongadas e ininterrumpidas en condiciones reales.

Las pruebas de validación deberían contemplar un estudio estadístico que caracterice la confiabilidad del sistema, incluyendo:

- Probabilidad de pérdida de medidas basada en la disponibilidad energética del dispositivo
- Análisis del balance energético durante condiciones ambientales adversas, como tormentas prolongadas de invierno que reduzcan la generación solar
- Caracterización de la degradación de componentes a lo largo del tiempo
- Evaluación del comportamiento térmico en diferentes estaciones del año

Despliegue en campo

Como se mencionó en la sección Sección 6.1, no se pudo realizar el despliegue del dispositivo desarrollado. Un paso fundamental a futuro es implementar el dispositivo en condiciones reales de operación y monitorear su funcionamiento para completar la validación del sistema.

El despliegue en campo permitirá:

- Validar el funcionamiento del sistema bajo condiciones ambientales reales
- Identificar posibles puntos de falla no detectados en las pruebas de laboratorio

- Evaluar la interfaz de usuario y los procedimientos de mantenimiento
- Determinar requerimientos adicionales para mejorar la resiliencia del sistema

Los resultados obtenidos del despliegue servirán para adaptar la implementación actual o definir mejoras necesarias para optimizar las capacidades de operación autónoma y confiabilidad a largo plazo.

Esta página ha sido intencionalmente dejada en blanco.

Referencias

[1] Mx25r6435f: Wide range, 64mb, ultra low power serial nor flash memory, August 2022. [Online]. Available: https://www.macronix.com/Lists/Datasheet/Attachments/8868/MX25R6435F,%20Wide%20Range,%2064Mb,%20v1.6.pdf. Accessed: May 11, 2025.

[2] 1-wire search algorithm — application note 187, May 2025. [Online]. Available: https://www.analog.com/media/en/technical-documentation/app-notes/1wire-search-algorithm.pdf. Accessed: June 3, 2025.

[3] Amazon s3 — scalable object storage for data storage, retrieval, backup and disaster recovery, June 2025. [Online]. Available: https://aws.amazon.com/s3/. Accessed: Jun. 13, 2025.

[4] Antenna requirements for gateway reference design — nwp-033 white paper, June 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/nwp_033/page/WP/nwp_033/nwp_033_antenna_req.html. Accessed: Jun. 22, 2025.

[5] Beagleplay — portable linux gaming, media and general-purpose embedded platform, June 2025. [Online]. Available: https://www.beagleboard.org/boards/beagleplay. Accessed: Jun. 22, 2025.

[6] Boyaflores: Readme, May 2025. [Online]. Available: https://gitlab.fing.edu.uy/diego.fraga/boyaflores/-/blob/main/README.md?ref_type=heads. Accessed: May. 18, 2025.

[7] Boyaflores: Wiki, May 2025. [Online]. Available: https://gitlab.fing.edu.uy/diego.fraga/boyaflores/-/wikis/home. Accessed: May. 18, 2025.

[8] Builder design pattern, June 2025. [Online]. Available: https://refactoring.guru/design-patterns/builder. Accessed: Jun. 21, 2025.

[9] Cobertura móvil, April 2025. [Online]. Available: https://www.antel.com.uy/personas/servicios/movil/cobertura. Accedido el: 22 de abril de 2025.

[10] Conductivity probe k 1.0, April 2025. [Online]. Available: https://atlas-scientific.com/probes/conductivity-probe-k-1-0/. Accessed: Apr. 9, 2025.

[11] Date-time api, March 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/nrf-apis-latest/page/group_date_time.html#details. Accessed: Mar. 18, 2025.

[12] Device driver model — zephyr project documentation, May 2025. [Online]. Available: https://docs.zephyrproject.org/latest/kernel/drivers/index.html. Accessed: June 3, 2025.

[13] Devicetree — build and configuration systems, June 2025. [Online]. Available: https://docs.zephyrproject.org/latest/build/dts/index.html. Accessed: Jun. 22, 2025.

[14] Easydma — nrf9161 product specification, April 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ps_nrf9161/page/easydma.html. Accessed: Apr. 12, 2025.

[15] Emqx: The unified mqtt and ai platform, April 2025. [Online]. Available: https://www.emqx.com/en. Accessed: Apr. 9, 2025.

[16] Ezo™ conductivity circuit, April 2025. [Online]. Available: https://atlas-scientific.com/embedded-solutions/ezo-conductivity-circuit/. Accessed: Apr. 9, 2025.

[17] Getting started with current measurements on the nrf9160, 2025. [Online]. Available: https://devzone.nordicsemi.com/guides/hardware-design-test-and-measuring/b/nrf9x/posts/getting-started-with-current-measurements-on-the-nrf9160. Accessed: June 9, 2025.

[18] Kconfig language, June 2025. [Online]. Available: https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html. Accessed: Jun. 14, 2025.

[19] Matplotlib: Visualization with python, May 2025. [Online]. Available: https://matplotlib.org/. Accessed: May 11, 2025.

[20] Mcuboot, mcumgr, and dfu target — nrf connect sdk intermediate, lesson 9, June 2025. [Online]. Available: https://academy.nordicsemi.com/courses/nrf-connect-sdk-intermediate/lessons/lesson-9-bootloaders-and-dfu-fota/topic/mcuboot-mcumgr-and-dfu-target/. Accessed: Jun. 13, 2025.

[21] Measuring current in ampere meter mode — power profiler kit ii user guide, May 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ug_ppk2/page/UG/ppk/measure_current_ampere_meter.html. Accessed: May 29, 2025.

[22] Mqtt version 5.0 specification, June 2025. [Online]. Available: https://mqtt.org/mqtt-specification/. Accessed: Jun. 21, 2025.

[23] Mqttx – powerful mqtt 5.0 desktop client, April 2025. [Online]. Available: https://mqttx.app/. Accessed: Apr. 9, 2025.

[24] Mxhs83qe3000 – 50 test probe for swd/swf connectors, June 2025. [Online]. Available: https://www.mouser.com/ProductDetail/Murata-Electronics/MXHS83QE3000?qs=QnQa2dTVfy7u6%252B1BCmlR%252Bw%3D%3D. Accessed: Jun. 22, 2025.

[25] Ncr18650b lithium-ion rechargeable battery datasheet, June 2025. [Online]. Available: https://www.batteryspace.com/prod-specs/ncr18650b.pdf?srsltid=AfmBOorclI6Lh79CtlJBEyTQ3minjwbAyKTKLW2ET3o0fV5cJhBUg9NB. Accessed: Jun. 13, 2025.

[26] nrf connect for desktop, April 2025. [Online]. Available: https://www.nordicsemi.com/Products/Development-tools/nRF-Connect-for-Desktop. Accessed: Apr. 9, 2025.

[27] nrf9160 dk, March 2025. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/nRF9160-DK. Accessed: Apr. 8, 2025.

[28] nrf9160 psm, June 2025. [Online]. Available: https://devzone.nordicsemi.com/f/nordic-q-a/62333/nrf9160-psm. Accessed: Jun. 21, 2025.

[29] nrf9160 – key features, March 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ps_nrf9160/page/nRF9160_html5_keyfeatures.html. Accessed: Apr. 8, 2025.

[30] Observer design pattern, June 2025. [Online]. Available: https://refactoring.guru/design-patterns/observer. Accessed: Jun. 21, 2025.

[31] Odroid-n2l with 4gb ram – high-performance embedded linux board, June 2025. [Online]. Available: https://www.hardkernel.com/shop/odroid-n2l-with-4gbyte-ram/. Accessed: Jun. 22, 2025.

[32] Online power profiler for lte, April 2025. [Online]. Available: https://devzone.nordicsemi.com/power/w/opp/3/online-power-profiler-for-lte. Accessed: Apr. 9, 2025.

[33] Otii arc pro current measurement accuracy, 2025. [Online]. Available: https://docs.qoitech.com/user-manual/otii/hardware/otii-arc-pro. Accessed: June 13, 2025.

[34] Otii products — power analyzer, battery simulator & testing, May 2025. [Online]. Available: https://www.qoitech.com/products/. Accessed: May 29, 2025.

[35] paho-mqtt: Mqtt python client library, May 2025. [Online]. Available: https://pypi.org/project/paho-mqtt/. Accessed: May 11, 2025.

[36] pandas: Python data analysis library, May 2025. [Online]. Available: https://pandas.pydata.org/. Accessed: May 11, 2025.

[37] Pmu oscillators — nrf9160 product specification, June 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ps_nrf9160/page/pmu_oscillators.html#unique_590298043. Accessed: Jun. 12, 2025.

[38] Portenta h7 – high-performance industrial microcontroller board, June 2025. [Online]. Available: https://docs.arduino.cc/hardware/portenta-h7/. Accessed: Jun. 22, 2025.

[39] Power optimization for cellular applications — nrf91 series, June 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ncs-3.0.0/page/nrf/test_and_optimize/optimizing/power_nrf91.html#cellular_applications. Accessed: Jun. 12, 2025.

[40] Power profiler kit ii - measurement accuracy, 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ug_ppk2/page/UG/ppk/ppk_measure_accuracy.html. Accessed: June 13, 2025.

[41] Power profiler kit ii — development hardware, May 2025. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/Power-Profiler-Kit-2. Accessed: May 29, 2025.

[42] Power saving techniques — cellular iot fundamentals, lesson 1, 2025. [Online]. Available: https://academy.nordicsemi.com/courses/cellular-iot-fundamentals/lessons/lesson-1-cellular-fundamentals/topic/lesson-1-power-saving-techniques/. Accessed: June 3, 2025.

[43] Prepare the dk for current measurements — nrf9160 dk user guide, May 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ug_nrf9160_dk/page/UG/nrf91_DK/measuring_current/prepare_board.html. Accessed: May 29, 2025.

[44] Raspberry pi 3 model b – compact, versatile single-board computer, June 2025. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-3-model-b/. Accessed: Jun. 22, 2025.

[45] Raspberry pi 4 model b – powerful single-board computer, June 2025. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. Accessed: Jun. 22, 2025.

[46] Raspberry pi 5 – next-generation single-board computer, June 2025. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-5/. Accessed: Jun. 22, 2025.

[47] Raspberry pi pico 2 – microcontroller board with rp2040, June 2025. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-pico-2/. Accessed: Jun. 22, 2025.

[48] Rtc — real time counter — nrf9160 product specification, June 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ps_nrf9160/page/rtc.html. Accessed: Jun. 12, 2025.

[49] Saadc — successive approximation analog-to-digital converter, May 2025. [Online]. Available: https://docs.nordicsemi.com/bundle/ps_nrf52840/page/saadc.html. Accessed: June 3, 2025.

[50] Singleton design pattern, June 2025. [Online]. Available: https://refactoring.guru/design-patterns/singleton. Accessed: Jun. 21, 2025.

[51] Sparkfun thing plus nrf9160 – cellular iot development board, June 2025. [Online]. Available: https://www.sparkfun.com/sparkfun-thing-plus-nrf9160.html. Accessed: Jun. 22, 2025.

[52] Streamlit – turn data scripts into shareable web apps in minutes, June 2025. [Online]. Available: https://streamlit.io/. Accessed: Jun. 13, 2025.

[53] Thingy:91-x – development hardware for cellular iot, June 2025. [Online]. Available: https://www.nordicsemi.com/Products/Development-hardware/Nordic-Thingy-91-X. Accessed: Jun. 13, 2025.

[54] Yagi directional roof antenna – 3g/4g/lte wide-band signal booster, June 2025. [Online]. Available: https://www.tupavco.com/products/yagi-directional-roof-antenna-3g4glte-wide-band-cell-signal-booster. Accessed: Jun.22,2025.

[55] Adafruit Industries. 5v 5w solar panel – etfe [voltaic p105], April 2025. [Online]. Available: https://www.adafruit.com/product/5367. Accessed: Apr. 9, 2025.

[56] Adafruit Industries. Adafruit bq24074 universal usb/dc/solar charger breakout, April 2025. [Online]. Available: https://learn.adafruit.com/adafruit-bq24074-universal-usb-dc-solar-charger-breakout/overview. Accessed: Apr. 9, 2025.

[57] EMQX. Mqtt client examples: Python3 pub_sub_tls.py, May 2025. [Online]. Available: https://github.com/emqx/MQTT-Client-Examples/blob/master/mqtt-client-Python3/pub_sub_tls.py. Accessed: May 11, 2025.

[58] Linux Foundation. About the linux foundation, April 2025. [Online]. Available: https://www.linuxfoundation.org/about. Accessed: Apr. 5, 2025.

[59] littlefs-project. littlefs: A little fail-safe filesystem, March 2025. [Online]. Available: https://github.com/littlefs-project/littlefs. Accessed: Mar. 18, 2025.

[60] MCUboot Project. Mcuboot: Secure bootloader for 32-bit mcus, March 2025. [Online]. Available: https://github.com/mcu-tools/mcuboot. Accessed: Mar. 18, 2025.

[61] Dekun Tao. Mqtt in python with paho client: Beginner's guide 2025, March 2025. [Online]. Available: https://www.emqx.com/en/blog/how-to-use-mqtt-in-python. Accessed: May 11, 2025.

[62] R. Trinchin, G. Manta, R. Santana, L. Rubio, S. Horta, C. Passadore, C. de Mello, M. N. Szephegyi, and M. Barreiro. Hacia un monitoreo continuo de variables oceanográficas en el parque nacional isla de flores, uruguay, October 2020. [Internet]. Vol. 21, ene–jun; pp. 89–108. Disponible en: https://ojs.latu.org.uy/index.php/INNOTEC/article/view/544. Accessed: Jun. 21, 2025.

[63] Zephyr Project. About the zephyr project, April 2025. [Online]. Available: https://www.zephyrproject.org/learn-about/. Accessed: Apr. 5, 2025.

[64] Zephyr Project. File system api — zephyr project documentation, April 2025. [Online]. Available: https://docs.zephyrproject.org/latest/services/file_system/index.html#file-system-api. Accessed: Apr. 9, 2025.

[65] Zephyr Project. Time utility apis, March 2025. [Online]. Available: https://docs.zephyrproject.org/latest/kernel/timeutil.html#. Accessed: Mar. 18, 2025.

Apéndice A

Hoja de datos de la red LTE-M

A continuación se adjunta la hoja de datos de la red LTE-M provista por Antel.

Internet of Things

Especificaciones Técnicas de la Implementación

Setiembre 2018

ESPECIFICACIONES TÉCNICAS

	NB-IoT
Modos de Operación	In-band LTE
Bandas de Operación	3 (1800MHz), 28 (700MHz)
Duplexación	HD-FDD
Modulación	DL: QPSK (OFDMA) UL: BPSK, QPSK (SC-FDMA)
Espaciamiento de Sub-portadoras	DL: 15 kHz UL: 15 kHz
Transmisiones Multi-Tone	Soportado[1]
HARQ	DL: Asíncrono UL: Asíncrono
Cobertura	- CE0 (144 dBm MCL) - CE1 (154 dBm MCL) - CE2 (hasta 164 dBm MCL)
Movilidad	- IDLE: soportado (intra e inter frecuencia) - CONNECTED: no soportado
Power Class	- PC3 (23 dBm) - PC5 (20 dBm)
Velocidad de Transferencia de Datos (máx)[2]	DL: 22,7 kbps UL: 62,5 kbps
Funcionalidades	- Soporte para DoNAS (CP CIoT EPS Optimization) - Power Saving Mode (PSM)[2] - Extended Discontinuous Reception (eDRX)[3]

	LTE-M
Bandas de Operación	3 (1800MHz), 28 (700MHz)
Duplexación	HD-FDD
Modulación (máx)	DL: 16QAM UL: 16QAM
Espaciamiento de Sub-portadoras	DL: 15 kHz UL: 15 kHz
HARQ	DL: Asíncrono UL: Asíncrono
Cobertura	Mode A: CE0 (MCL en el entorno de 144 dBm), CE1 (MCL entre 149 y 154 dBm) [4]
Movilidad	- IDLE: soportado (intra e inter frecuencia) - CONNECTED: soportado (intra frecuencia)
Power Class	- PC3 (23 dBm) - PC5 (20 dBm)
Velocidad de Transferencia de Datos (máx) [1]	DL: 300 kbps UL: 375 kbps
Funcionalidades	- Power Saving Mode (PSM)[2] - Extended Discontinuous Reception (eDRX)[3]

[1] *Single-tone (8ms) o multi-tone (3, 6 y 12 subportadoras) con duración de 4ms, 2ms y 1ms, respectivamente, considerando subportadoras de 15kHz.*

[2] *El soporte de esta funcionalidad depende de las capacidades del dispositivo. Los valores soportados por la red son: T3324: 0-11160 (segundos); T3412 extendido: 0-9912 (horas).*

[3] *El soporte de esta funcionalidad depende de las capacidades del dispositivo. Los valores soportados por la red NBIoT son: eDRX cycle: {20,48; 40,96; 81,92; 163,84; 327,68; 655,36; 1310,72; 2621,44; 5242,88; 10485,76} (segundos); PTW: {2,56; 5,12; 7,68; 10,24; 12,8; 15,36; 17,92; 20,48; 23,04; 25,60; 28,16; 30,72; 33,28; 35,84; 38,40; 40,96} (segundos).*
Los valores soportados por la red LTE-M son: eDRX cycle: {20,48; 40,96; 81,92; 163,84; 327,68; 655,36; 1310,72; 2621,44; 5242,88; 10485,76} (segundos); PTW: {1,28; 2,56; 3,84; 5,12; 6,4; 7,68; 8,96; 10;24; 11,52; 12,8; 14,08; 15,36; 16,64; 17,92; 19,20; 20,48} (segundos).

[4] *Valores teóricos.*

Glosario

- **Artifact:** Archivo o conjunto de archivos generados por un job dentro de un pipeline de GitLab, que pueden almacenarse y compartirse entre etapas del proceso de integración y despliegue.
- **Bootloader:** Programa que se ejecuta al reiniciar para cargar y arrancar la aplicación principal.
- **Branch:** Línea de desarrollo independiente en un repositorio Git.
- **Broker:** Servidor que enruta mensajes entre publicadores y suscriptores MQTT.
- **Bug:** Defecto de software que causa un comportamiento incorrecto o inesperado.
- **Callback:** Función que se registra para ser ejecutada automáticamente en respuesta a un evento específico.
- **Commit:** Instantánea de cambios registrada en el historial de un repositorio Git.
- **Copy-on-write:** Optimización que pospone la copia de datos hasta que se produce una modificación.
- **Devicetree:** Estructura de datos que describe el hardware al sistema operativo en tiempo de compilación.
- **Development Kit:** Placa de desarrollo que integra un microcontrolador u otro sistema programable junto con interfaces de comunicación, conectores de depuración y otros recursos para facilitar la creación y prueba de aplicaciones embebidas.
- **Driver:** Software de bajo nivel que controla un dispositivo de hardware y expone una API estándar.
- **Features:** Capacidades o funcionalidades específicas ofrecidas por un sistema, módulo o componente.
- **File System:** Método y estructuras que usa un sistema operativo para organizar archivos en un medio de almacenamiento.

- **Firmware:** Software de bajo nivel almacenado en memoria no volátil que proporciona control al hardware.
- **Flash:** Memoria no volátil que puede ser borrada y reprogramada eléctricamente.
- **Footprint:** Cantidad de recursos que un sistema, aplicación o componente ocupa, como memoria, CPU o espacio físico.
- **Framework:** Conjunto de herramientas, librerías y convenciones que facilitan el desarrollo estructurado de software.
- **GitLab Runner:** Agente que ejecuta los trabajos CI/CD definidos en un pipeline de GitLab.
- **Handler:** Función o rutina que responde a un evento, interrupción o callback.
- **Hash:** Valor de longitud fija generado por una función hash, usado para sumas de comprobación o búsqueda rápida.
- **Kernel:** Componente central de un sistema operativo que gestiona hardware, memoria y planificación.
- **Keys:** Objetos inmutables utilizados para indexar y recuperar valores en un diccionario Python.
- **Kconfig:** Sistema de configuración en tiempo de compilación (menuconfig) usado por Zephyr y Linux.
- **LittleFS:** Sistema de archivos ligero para memoria flash con nivelación de desgaste y tolerancia a pérdidas de energía.
- **Log:** Registros de eventos o mensajes generados por software o hardware.
- **Logger:** Componente o biblioteca que genera y escribe mensajes de log.
- **Merge:** Operación de Git que integra historiales divergentes.
- **Merge request:** Es una propuesta de merge revisable.
- **Metadatos:** Datos que describen otros datos (como por ejemplo, marcas de tiempo, permisos, etc.).
- **Mutex:** Objeto de sincronización que permite acceso exclusivo a un recurso compartido entre múltiples hilos.
- **Overhead:** Carga adicional de tiempo, memoria u otros recursos que implica una operación o abstracción.
- **Performance:** Medida de cuán eficientemente un sistema ejecuta tareas (velocidad, uso de recursos, etc.).

- **Poll:** Mecanismo que permite verificar periódicamente el estado de una conexión, evento o recurso (por ejemplo, para mantener vivo un canal MQTT).
- **Power Management:** Conjunto de técnicas y mecanismos para reducir el consumo energético de un sistema sin comprometer su funcionalidad.
- **Reboot:** Reinicio del sistema, que detiene toda ejecución y vuelve a comenzar desde el bootloader.
- **Root:** Cuenta de usuario con privilegios máximos o directorio raíz en un sistema de archivos.
- **Scheduler:** Componente del sistema operativo que decide qué hilo o tarea se ejecuta en cada momento.
- **Script:** Archivo con una secuencia de comandos interpretados por un lenguaje de scripting.
- **Shield:** Placa de expansión diseñada para agregarse a una placa base (como Arduino) y extender sus funcionalidades.
- **Slave:** Dispositivo en un bus de comunicación controlado por un maestro; responde a sus órdenes (por ejemplo, en buses 1-Wire o I2C).
- **String:** Secuencia de caracteres tratada como dato textual.
- **Subsystem:** Conjunto de módulos que ofrecen una capacidad específica dentro de Zephyr.
- **Thread:** Secuencia mínima de instrucciones que el sistema operativo puede planificar de forma independiente.
- **Throughput:** Cantidad de datos transmitidos exitosamente a través de un canal de comunicación por unidad de tiempo.
- **Tick:** Interrupción de un timer o contador que sirve de base de tiempo al scheduler del sistema operativo.
- **Timeout:** Evento que se activa cuando una operación excede el tiempo asignado.

Esta página ha sido intencionalmente dejada en blanco.

Siglas

- **ADC:** *Analog to Digital Converter*. Circuito que convierte señales analógicas continuas en valores digitales.
- **API:** *Application Programming Interface*. Interfaz de software que permite comunicación entre módulos.
- **APN:** *Access Point Name*. Nombre del punto de acceso que configura la conexión entre una red celular y una red IP externa, como Internet; es necesario para establecer conectividad de datos móviles.
- **AWS:** *Amazon Web Services*. Plataforma de computación en la nube bajo demanda de Amazon.
- **CI/CD:** *Continuous Integration / Continuous Delivery*. Práctica DevOps para automatizar pruebas, construcción y despliegue.
- **DFU:** *Device Firmware Update*. Proceso para actualizar la imagen de firmware.
- **eDRX:** *Extended Discontinuous Reception*. Característica de LTE que extiende los intervalos de paginación.
- **FOTA:** *Firmware Over-the-Air*. Actualización de firmware entregada inalámbricamente a dispositivos embebidos.
- **GPIO:** *General Purpose Input/Output*. Pines de entrada/salida digital configurables en un microcontrolador.
- **HAL:** *Hardware Abstraction Layer*. Capa de software que abstrae el hardware del código de alto nivel.
- **HAT:** *Hardware Attached on Top*. Placa de expansión diseñada para conectar directamente a un Raspberry Pi y extender sus funcionalidades.
- **HTTP:** *Hypertext Transfer Protocol*. Protocolo de aplicación para comunicación cliente-servidor en la web.
- **IAM:** *Identity and Access Management*. Servicio de AWS para gestionar usuarios, roles y permisos.

- **IMEI:** *International Mobile Equipment Identity*. Número único que identifica un dispositivo móvil en redes celulares.
- **IPXX:** *Ingress Protection Code*. Clasificación que indica el grado de protección de un equipo frente a la entrada de polvo (primer dígito) y agua (segundo dígito).
- **I2C:** *Inter-Integrated Circuit*. Bus serie de dos cables (SDA, SCL) para comunicación entre dispositivos.
- **JSON:** *JavaScript Object Notation*. Formato ligero y legible para intercambio de datos basado en JavaScript.
- **LTE:** *Long-Term Evolution*. Es un estándar definido por 3GPP para banda ancha móvil de alta velocidad.
- **LTE-M:** *LTE-Machine Type Communication*. Variante de bajo consumo de LTE para dispositivos IoT, también conocida como Cat-M1.
- **MQTT:** Protocolo liviano de mensajería tipo publicar/suscribir para IoT.
- **MSMF:** *Modular State Machine Framework*. Marco de trabajo de Zephyr para construir máquinas de estados jerárquicas.
- **NB-IoT:** *Narrowband Internet of Things*. Tecnología de radio LPWAN estandarizada por 3GPP para implementaciones masivas de IoT.
- **NTP:** *Network Time Protocol*. Protocolo usado para sincronizar el reloj de un sistema con servidores de tiempo en red.
- **OTA:** *Over-the-Air*. Distribución inalámbrica de actualizaciones de firmware o configuración.
- **PCB:** *Printed Circuit Board*. Placa laminada con trazas conductoras que conecta componentes electrónicos.
- **POST:** *HTTP POST Method*. Método HTTP para enviar datos al servidor creando recursos subordinados.
- **PSM:** *Power Saving Mode*. Modo de inactividad que permite largos períodos de suspensión sin perder conexión.
- **PTW:** *Paging Time Window*. Intervalo dentro de eDRX donde el dispositivo escucha la red.
- **PUT:** *HTTP PUT Method*. Método HTTP para reemplazar o crear un recurso con los datos proporcionados.
- **QoS:** *Quality of Service*. Nivel de garantía de entrega asociado a cada mensaje MQTT (0,1,2).

- **RAM:** *Random Access Memory*. Memoria volátil de lectura/escritura usada durante la ejecución del programa.
- **RL:** *Return Loss*. Medida (en dB) de la potencia reflejada hacia la fuente debido a desadaptación de impedancias en una línea de transmisión de radiofrecuencia.
- **RRC:** *Radio Resource Control*. Protocolo de control en LTE que maneja estados de conexión y movilidad.
- **RSRP:** *Reference Signal Received Power*. Potencia media recibida de las señales de referencia en LTE; indica la intensidad de la señal.
- **RSRQ:** *Reference Signal Received Quality*. Relación entre RSRP y la interferencia total (RSSI); mide la calidad de la señal LTE.
- **RTC (Contador):** *Real-Time Counter*. Periférico de bajo consumo que cuenta tics y puede generar alarmas.
- **RTC (Reloj):** *Real-Time Clock*. Circuito con batería que mantiene la hora calendario incluso apagado el sistema.
- **RTOS:** *Real-Time Operating System*. Sistema operativo que garantiza latencias acotadas en tareas críticas.
- **SAADC:** *Successive Approximation Analog-to-Digital Converter*. Convertidor analógico-digital basado en aproximaciones sucesivas, utilizado comúnmente en SoCs como los de Nordic por su eficiencia y precisión.
- **SBC:** *Single-Board Computer*. Computadora completa construida sobre una sola placa de circuito.
- **SIM:** *Subscriber Identity Module*. Tarjeta que identifica al abonado en redes móviles y permite la autenticación con el operador.
- **SiP:** *System-in-Package*. Paquete que integra múltiples chips y componentes pasivos en un solo sustrato.
- **SMA:** *SubMiniature version A*. Conector coaxial roscado utilizado comúnmente para señales de radiofrecuencia, especialmente en antenas y módulos de comunicación.
- **SMU:** *Source Measure Unit*. Instrumento que combina una fuente de voltaje y corriente con la capacidad de medir ambas simultáneamente mientras las suministra.
- **SNR:** *Signal-to-Noise Ratio*. Relación entre la potencia útil de la señal y el ruido; expresa la claridad de la recepción.
- **SPI:** *Serial Peripheral Interface*. Bus serie síncrono de doble vía con arquitectura maestro-esclavo.

- **SSH:** *Secure Shell*. Protocolo seguro para acceso remoto y ejecución de comandos cifrados.
- **TLS:** *Transport Layer Security*. Protocolo criptográfico que brinda confidencialidad e integridad de los datos en tránsito entre aplicaciones de red.
- **UART:** *Universal Asynchronous Receiver/Transmitter*. Comunicación serie que usa bits de inicio/parada para transmisión asíncrona.
- **VSWR:** *Voltage Standing Wave Ratio*. Relación de onda estacionaria de voltaje que cuantifica la desadaptación de impedancias en una línea de transmisión de radiofrecuencia.
- **WDT:** *Watchdog Timer*. Temporizador que reinicia el sistema si no es atendido periódicamente, utilizado para recuperación ante fallos.
- **YML:** *YAML Ain't Markup Language*. Extensión de archivos YAML.

Índice de tablas

Esta página ha sido intencionalmente dejada en blanco.

Índice de figuras

Esta es la última página.
Compilado el viernes 8 agosto, 2025.
http://iie.fing.edu.uy/