Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Diseño, Implementación y Validación del Software de Vuelo para el Nanosatélite UPMSat-3

Autor: María Muñoz Nieto
Tutor: Ángel Grover Pérez Muñoz

Madrid, Junio 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Diseño, Implementación y Validación del Software de Vuelo para el Nanosatélite UPMSat-3

Junio 2025

Autor: María Muñoz Nieto
Tutor: Ángel Grover Pérez Muñoz
Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

UPMSat-3 es un satélite desarrollado por el Instituto de Microgravedad Ignacio da Riva (IDR-UPM), junto a otros equipos de investigación de la Universidad Politécnica de Madrid (UPM), principalmente de las Escuelas Técnicas Superiores de Telecomunicación, Informática y Aeroespacial.

El tercer satélite desarrollado por este equipo tiene como objetivo servir de plataforma de demostración tecnológica. Este está diseñado para ejecutar diversos experimentos de investigación en vuelo y además resulta útil en ámbitos académicos, principalmente ayudando a formar a aquellos estudiantes que están participando en el desarrollo del proyecto.

El nanosatélite UPMSat-3 sigue el estándar de diseño de satélites *CubeSat* y se trata de un sistema compuesto por varios computadores de vuelo que se comunican principalmente haciendo uso del protocolo *Cubesat Space Protocol* (CSP). El software que se encuentra en estos computadores está desarrollado por el grupo de investigación en Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos (STRAST) de la Escuela Técnica Superior de Telecomunicación.

Este trabajo de fin de grado implica el desarrollo del software de vuelo del satélite, con el objetivo de que este sea capaz de comunicarse con tierra y de coordinar las operaciones realizadas por todos los computadores que lo compone. Para ello será necesario realizar diversas actividades como el análisis de los requisitos software del satélite, análisis del entorno de desarrollo, diseño de la arquitectura software, implementación, verificación y validación.

Primero se analizarán y detallarán las funciones que se deben diseñar e implementar en el software de vuelo. Seguidamente, se describirá el entorno software y herramientas usadas para implementar dichas funciones. Por último, se detallará el diseño y desarrollo de las funciones y su implementación y validación tanto en modelos de ingeniería como de vuelo.

Las pruebas realizadas sobre los sistemas resultaron exitosas, ya que el software probado cumplió las funciones previamente estudiadas y detalladas. Además, se realizó una validación adecuada y satisfactoria de los diferentes componentes de vuelo.

Abstract

UPMSat-3 is a satellite developed by the "Ignacio Da Riva" Institute for Microgravity Research (IDR-UPM), together with other research teams from the Universidad Politécnica de Madrid (UPM), from the schools of Telecommunications, Computer Science and Aeronautical and Space Engineering.

The third satellite developed the team is intended to serve as a technology demonstration platform. It is designed to perform various in-flight research experiments and is also useful in academic environments, mainly to help train the students who are participating in the development of the project.

The UPMSat-3 nanosatellite follows the CubeSat satellite design standard and is a system composed of several computers that communicate using the Cubesat Space Protocol (CSP). The software in these computers is developed by the Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos (STRAST) research group of the UPM school of Telecommunications.

This TFG is centered on the development of the satellite on-board software, with the objective of making it capable of both communicating with the ground station and coordinating the operations performed by all the computers that compose it. To achieve this, it will be required to perform several activities such as the analysis of the satellite software requirements, analysis of the framework, design of the software architecture, implementation, verification, and validation.

First, the functions to be designed and implemented in the flight software will be analyzed and clarified. Next, the framework and tools used to implement these functions will be described. Finally, further details will be provided regarding the design and development of the software functions and their implementation and validation in both the engineering model and the flight model.

The tests performed on the systems were successful since the software tested on them fulfilled the functions previously studied and detailed. In addition, an adequate and satisfactory validation of the flight components was performed.

Tabla de contenidos

Capítulo 1

Introducción

1.1. Misión UPMSat-3

UPMSat-3 es el tercer satélite diseñado por la Universidad Politécnica de Madrid y cuyo desarrollo está dirigido por el Instituto de Microgravedad Ignacio da Riva (IDR-UPM) de la Escuela Técnica Superior de Ingeniería Aeronáutica y del Espacio (ETSIAE) [1, 2, 3].

Diversos equipos de investigación de diferentes escuelas de la UPM son los encargados de desarrollar el UPMSat-3, en concreto, el grupo de investigación en Sistemas de Tiempo Real y Arquitectura de Servicios Telemáticos (STRAST, ETSIT-UPM) se ocupa del desarrollo del software, tanto el que se cargará en los computadores del satélite, como el presente en la estación de tierra.

El principal objetivo de la misión UPMSat-3 es desarrollar y lanzar un satélite capaz de ejecutar una serie de diferentes experimentos en vuelo, para posteriormente almacenar los datos generados por dichos experimentos y enviarlos a tierra donde pueden ser analizados. Debido a la naturaleza de estos experimentos podemos definir que la plataforma está diseñada principalmente para fines académicos y de investigación.

El microsatélite o nanosatélite UPMSat-3, denominado de esta forma debido a su pequeño tamaño, con 22 kg de masa y dimensiones de $0{,}25 \times 0{,}25 \times 0{,}3$m, está formado por una red de sistemas empotrados o embebidos. Es decir, el sistema está compuesto de microcontroladores que realizan funciones con un propósito específico, centrándose solo en realizar tareas específicas.

1.2. Motivación

Una de las principales motivaciones del proyecto UPMSat-3 es, como ya mencionado previamente, servir como plataforma de demostración tecnológica. Empresas y equipos de investigación podrán probar sus proyectos en órbita, ejecutando en el satélite lo que se denominan como experimentos. Dichos experimentos ejecutarán las pruebas definidas por estas empresas y equipos y, usando los

computadores del satélite, se recogerá la información de estos experimentos y se devolverá a tierra para su posterior análisis.

Las empresas y equipos para los que se han diseñado experimentos son el grupo de investigación GREMA de la Universidad Carlos III de Madrid con experimentos centrados en probar una serie de radiómetros fotónicos, DHV Technology aportando un sistema de potencia creado para su uso en microsatélites, HYDRA Space para probar su propio transceptor de comunicaciones miniaturizado, OC-CAM Space con mecanismos de despliegue y el propio IDR con el sistema de determinación de actitud del UPMSat-3.

La segunda motivación del proyecto es ofrecer a los estudiantes implicados en la misión la experiencia laboral que se obtiene al trabajar en un proyecto real. Debido a que el satélite está desarrollado por múltiples grupos de diferentes escuelas de la Universidad Politécnica de Madrid hay un alto número de estudiantes trabajando en la misión y esta les ofrece experiencia profesional y oportunidad de aplicar los conocimientos obtenidos a lo largo de la carrera en un entorno práctico.

1.3. Objetivos

El objetivo principal de este trabajo de fin de grado es **desarrollar el software de misión del microsatélite UPMSat-3**. Para cumplir con este objetivo, se ha definido una serie de objetivos específicos (OE) que se listan a continuación:

OE1. **Analizar y diseñar los requisitos software del satélite**. Para la obtención de los requisitos software es necesario hacer un análisis de la misión y los requisitos a nivel de sistema.

OE2. **Analizar el framework del software de la misión**. Este está compuesto por el sistema operativo FreeRTOS, protocolo CSP y otras librerías de soporte.

OE3. **Diseñar el software de misión del satélite**. Esto implica diseñar las funcionalidades del sistema, incluyendo la recolección y almacenamiento de datos, mensajes intercambiados entre tierra y satélite, y los modos de operación.

OE4. **Implementar y validar el software en los computadores de ingeniería y de vuelo.** Estas pruebas deberán demostrar que los componentes se comunican y realizan los requisitos establecidos adecuadamente.

1.4. Estructura del trabajo de fin de grado

Obviando este capítulo introductorio, este trabajo de fin de grado se ha estructurado en 6 capítulos resumidos a continuación:

- En el capítulo 2 se analizarán y detallarán las funciones que se deben diseñar e implementar y los componentes del satélite que realizarán dichas funciones.

- A continuación, en el capítulo 3, se describirá el entorno software y herramientas usadas para implementar las funciones del satélite.
- Posteriormente, en los capítulos 4 y 5 se detallará el diseño y desarrollo de las funciones y su implementación y validación tanto en modelos de ingeniería como de vuelo.
- Por último, en los capítulos 6 y 7 se desarrollará un análisis del impacto del proyecto y las conclusiones y resultados obtenidos de este trabajo.

Capítulo 2

Análisis y funcionalidades de la misión

Antes de comenzar a explicar el desarrollo del software de misión del microsatélite UPMSat-3 es necesario entender que funciones se quieren que cumpla y cuáles son las partes que lo componen.

El software de misión del UPMSat-3 se ha desarrollado con el propósito de monitorizar y controlar las operaciones del satélite, para lograr este objetivo, podemos definir las siguientes funcionalidades principales que debe implementar el software de misión:

- Recepción y decodificación de Telecomandos (TC), es decir, los mensajes recibidos de la estación de tierra, para su ejecución en vuelo.
- Recolección y almacenamiento de datos de Housekeeping (HK), datos cuyo contenido describe el estado actual del satélite.
- Ejecución de experimentos y obtención de los datos generados.
- Registro de eventos generados por el satélite.
- Codificación y transmisión de Telemetría (TM), es decir, los mensajes generados por el satélite que serán enviados a la estación de tierra.
- Control de los modos de operación del satélite.

Para entender cómo se realizarán estas actividades se explicará el cometido de cada componente del satélite y definiremos en detalle cada requisito software a desarrollar.

2.1. Componentes del UPMSat-3

El software de vuelo se encuentra desplegado en dos computadores diferentes, el **Computador de Misión** y el **Computador de Experimentos**. El Computador de Misión contiene el *Mission On-Board Software* (OBSW), que implementa operaciones a nivel de misión y comunicación con tierra. Por otro lado, el Compu-

tador de Experimentos implementa el *Experiments On-Board Software* (OBSW) que permite controlar experimentos realizados durante el vuelo, obtener los datos adquiridos y enviarlos al computador de la misión cuando este lo solicite.

Estos dos computadores se comunican a través de una interfaz *Controller Area Network* (CAN) empleando el protocolo de comunicación *CubeSat Space Protocol* (CSP) diseñado específicamente para la comunicación entre componentes de satélites *CubeSat* [4]. Esta conexión entre el Computador de Misión y Computador de Experimentos se denomina sistema de gestión de datos (On-Board Data Handling, OBDH) ya que se encarga de tratar los datos generados en vuelo. El OBDH se puede ver en la Figura 2.1.

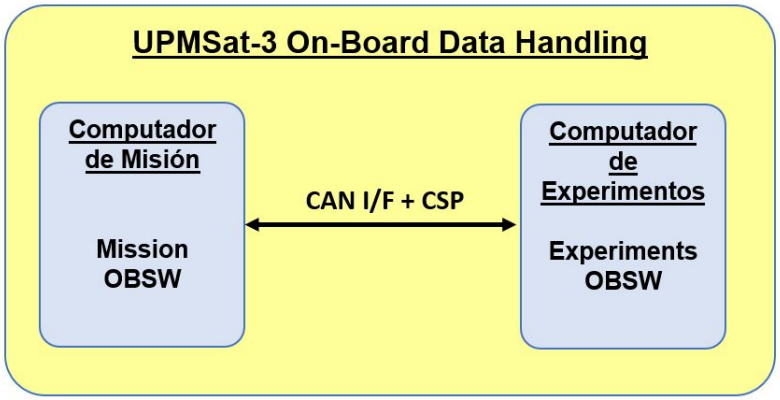

Figura 2.1: UPMSat-3 On-Board Data Handling System.

El OBDH, formado por el Computador de Misión y Computador de Experimentos, también tiene conexión con los siguientes subsistemas:

- **Electrical Power Subsystem (EPS):** Este componente diseñado por *DHV Technology* se encarga de alimentar los computadores del satélite, además de aportar sus propios datos de TM. Entre estos datos se encuentra el nivel de batería, que es imprescindible para el control de modos de operación.

- **Radio del satélite:** Esta radio es la responsable de unir el satélite con la estación de tierra. Recibe y codifica TM del Computador de Misión para enviarla a tierra y a su vez recibe y decodifica TC de tierra y se los envía al Computador de Misión, para que este pueda procesarlos y realizar las acciones necesarias.

- **Experimentos o cargas de pago:** El UPMSat-3 es capaz de realizar una serie de experimentos en vuelo, para ello, el Computador de Experimentos se comunicará con los diferentes computadores de cada experimento para mandarles instrucciones (como por ejemplo, para solicitar comenzar un experimento) y para recibir los datos de los experimentos en ejecución. El satélite tiene un total de 3 experimentos posibles a realizar, y cada uno de estos experimentos tiene su propio computador encargado de realizarlo:

- HYDRA-AMSAT Telemetry, Tracking and Command (TTC): Esta radio de la compañía HYDRA Space se encenderá en vuelo y funcionará a la vez que la radio principal del satélite, enviando a tierra sus propios datos.
- Radiómetros UC3M-GREMA: En este experimento, los radiómetros fotónicos del grupo de investigación GREMA de la Universidad Carlos III de Madrid se usarán para probar el concepto de detección de alta sensibilidad.
- Attitude Determination and Control Subsystem (ADCS): Estos códigos de control y determinación de actitud desarrollados por el IDR controlarán la orientación del satélite en vuelo.

- **Data Acquisition Board (DAB):** El computador de experimentos se comunicará con este computador para leer señales analógicas y realizar el control térmico del satélite.

Las conexiones entre estos subsistemas, y los protocolos de comunicación se encuentran representados en la Figura 2.2.

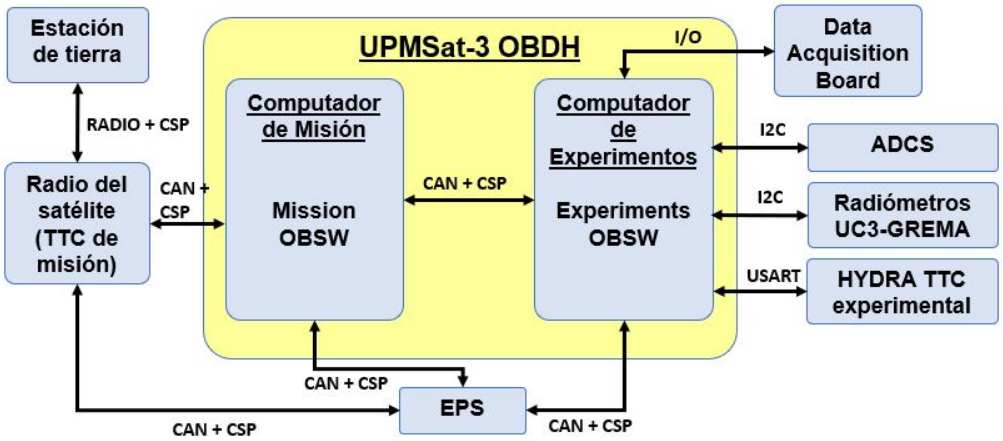

Figura 2.2: Diagrama de la arquitectura de UPMSat-3.

Una vez explicada la relación entre los subsistemas se va a describir detalladamente las características y funciones de aquellos con los que más se han trabajado a la hora de desarrollar el software de vuelo.

2.1.1. Computador de Misión NanoMind A3200

El Computador de Misión es el NanoMind A3200, fabricado por la compañía *GomSpace* [5]. Es un pequeño computador diseñado específicamente para su uso en satélites de pequeña escala basados en el estándar de diseño de satélites *CubeSat*. Este componente se encuentra representado en la Figura 2.3.

El NanoMind contiene un microcontrolador AT32UC3C0512C y posee múltiples interfaces compatibles con el protocolo CSP, entre ellas una interfaz CAN con

una velocidad máxima de bus de 1 Mbit/s. Esta característica del computador es esencial para permitir al NanoMind comunicarse a través de CAN bus con otros componentes del satélite. Entre otras interfaces de interés, este computador contiene una interfaz JTAG, que permite cargar software y una interfaz *Universal Synchronous Asynchronous Receiver Transmitter* (USART), usada como una interfaz de depuración.

Figura 2.3: Foto de NanoMind A3200.

Con respecto a la memoria externa, NanoMind posee 128 MB de NOR flash, 32 kB de FRAM y 32 MB de SDRAM. Esta memoria externa nos permite guardar datos que no queremos que se pierdan en caso de que el computador se reinicie.

Por último, centrándonos en el software del computador, tenemos el **A3200 *Command & Managment Software Development Kit* (SDK)** y el **A3200 *Mission Library*** [6]. El software de misión se ha escrito con ayuda de este SDK y librerías, y entre las funciones más importantes de este software se incluye:

- *LibCSP* versión 1.6.
- Un sistema de parámetros con *libparam* que permite almacenar datos y la configuración del sistema con facilidad.
- *GomSpace Shell* (GOSH), una terminal accesible a través de la interfaz USART, que es de gran ayuda para probar y depurar el NanoMind.
- Librería *libhk*, que envía a tierra paquetes de datos de housekeeping.

Estas, y otras funciones se explorarán más en detalle en el Capítulo 3.

2.1.2. Radio NanoCom AX100

NanoCom AX100 es la radio del satélite, encargada de enviar y recibir mensajes de tierra. Este componente, al igual que el Computador de Misión NanoMind, es de la compañía *GomSpace*, por lo que poseen funciones software que son compatibles entre sí [7]. Se incluye una fotografía de este computador en la Figura 2.4.

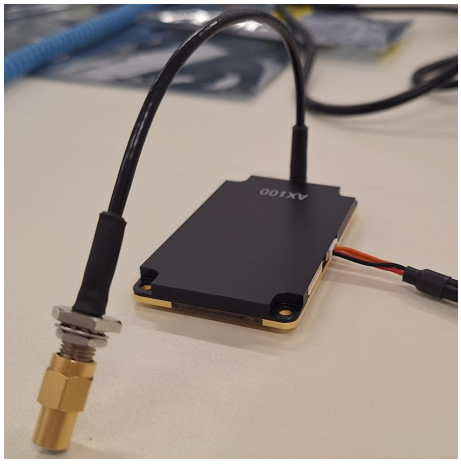

Figura 2.4: Foto de NanoCom AX100 con una *dummy load* conectada.

Para hacer pruebas, la radio NanoCom AX100 cuenta también con la terminal GOSH, accesible desde la interfaz USART. Además, cuenta con el sistema de parámetros de la librería *libparam* para guardar su configuración, por lo que, a través de CSP, el NanoMind es capaz de entender dichos parámetros y editarlos, permitiéndonos así cambiar la configuración de la radio desde el computador de misión.

Es importante recalcar que, a diferencia del NanoMind, no se dispone del código fuente del software incluido en la radio, por lo que es imposible cargar nuestro propio software en el componente. Realizar la configuración de los parámetros de este computador es posible desde el NanoMind.

2.1.3. NanoDock DMC-3

NanoDock DMC-3 es un componente pasivo del satélite, diseñado por *GomSpace* para conectar dos de sus computadores, el NanoCom AX100 y NanoMind, tal y como se muestra en la Figura 2.5 [8]. Los conectores que sirven para conectar estos dos componentes al NanoDock DMC-3 poseen, entre otras, interfaces CAN y USART, por lo que, a través del NanoDock DMC-3 se puede acceder a las terminales GOSH de ambos componentes además de permitir que se comuniquen entre ellos gracias a la interfaz de comunicación CAN.

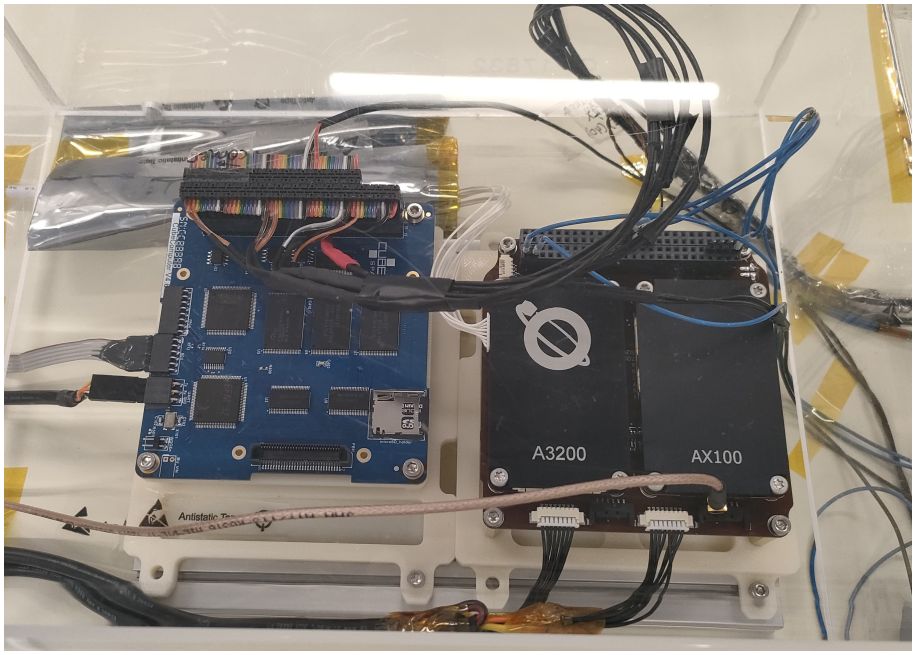

Figura 2.5: Foto de AX100 y NanoMind conectados a través de NanoDock, junto al computador de experimentos.

2.1.4. Computador de Experimentos CubeComputer

El computador de experimentos es el CubeComputer V4.1 de la compañía *CubeSpace*. Este computador hace uso de un microcontrolador EFM32GG280F1024 y dispone de interfaces de comunicación CAN, *Universal Asynchronous Receiver-Transmitter* (UART) e I^2C permitiendo así la comunicación con NanoMind y los computadores de cada uno de los experimentos. Cabe destacar que a través de la interfaz UART se puede depurar el software cargado.

Al igual que con el NanoMind, en el CubeComputer se puede cargar nuestro propio software. Es en este software en el que también se ha incluido *libCSP* con la versión 1.6 para que se pueda comunicar sin dificultades con el NanoMind.

2.1.5. Estación de tierra

La estación de tierra o *Ground Station* (GS) es, como indica su nombre, una estación situada en tierra para recoger y enviar datos al satélite. Es por esto que la estación de tierra debe disponer de una radio capaz de comunicarse con el UPMSat-3 y un computador con el que se puedan almacenar datos, procesarlos y crear TCs para enviarlos.

Para simplificar la conexión entre componentes lo máximo posible, se decidió seguir usando los productos de *GomSpace*, para garantizar que la radio del satélite y la de tierra sean compatibles y se puedan comunicar sin dificultades. Estos componentes usados para la estación son el **NanoCom GS100** y **NanoCom MS100**, que se pueden observar en la Figura 2.6.

El NanoCom GS100 [9] está diseñado específicamente para ser una radio de estación de tierra que sea capaz de comunicarse con la radio NanoCom AX100, es por esto que el NanoCom GS100 está compuesto por dos radios NanoCom AX100 ya que así nos asegurarnos que la forma de comunicarse de la radio del satélite y de la radio de tierra es idéntica, compartiendo las mismas características.

Mientras tanto el NanoCom MS100 [10] es el computador encargado de analizar los datos recibidos del satélite, crear TCs, cambiar la configuración de las radios del NanoCom GS100, etc. Este computador contiene un sistema operativo GNU / Linux que ejecuta una serie de servicios, entre ellos, servicios que conectan el NanoCom MS100 con las radios del NanoCom GS100, de forma que se puede acceder a su terminal GOSH desde este computador.

También se ha adquirido el **MS100 *Command & Management* SDK** y **MS100 *Mission Library***, que ofrecen librerías similares a aquellas del NanoMind, como recolección de housekeeping y el mismo sistema de almacenamiento de datos por parámetros.

Figura 2.6: Foto de GS, con MS100 a la derecha del monitor y GS100 debajo del portátil.

2.2. Funcionalidades software

2.2.1. Housekeeping

El housekeeping es una función fundamental del satélite. Consiste en recolectar información y datos sobre el estado actual del satélite y sus componentes de forma periódica y almacenar dicha información para enviarla a tierra. El computador de misión o NanoMind, debe de ser el encargado de recoger datos de los distintos componentes del microsatélite y de guardarlos.

En concreto, el NanoMind va a recoger datos periódicamente de los siguientes componentes:

- Del propio NanoMind.
- De la radio del satélite NanoCom AX100.
- Del computador de experimentos CubeComputer.
- Del EPS.

A continuación, se va a especificar qué datos se van a obtener de cada componente en las tablas Tabla 2.1, Tabla 2.2, Tabla 2.3 y Tabla 2.4.

Cuadro 2.1: Datos de HK de NanoMind.

Nombre	**Bytes**	**Tipo**	**Descripción**
Mission time	4	int32	Número de segundos transcurridos desde que el NanoMind se encendió.
System mode	1	uint8	Modo de operación actual del satélite.
RAM image	1	bool	Indica si el OBSW está ejecutándose desde una RAM image.
CPU temperature	2	int16	Temperatura del CPU en grados Celsius.
RAM temperature	2	int16	Temperatura de la RAM en grados Celsius.
Flash current	2	uint16	Consumición de corriente del dispositivo flash medido en miliamperios.
Boot count	2	uint16	Número de veces que el NanoMind se ha encendido.

Cuadro 2.2: Datos de HK de NanoCom AX100.

Nombre	**Bytes**	**Tipo**	**Descripción**
CPU temperature	2	int16	Temperatura del CPU en grados Celsius.
PA temperature	2	int16	Temperatura del *Power Amplifier* (PA) en grados Celsius.
Last RSSI	2	int16	Último *Recieved Signal Strength Indicator* (RSSI) recibido.
Last contact	4	uint32	Timestamp del último paquete válido, medido en segundos desde el encendido del AX100.

Cuadro 2.3: Datos de HK de CubeComputer.

Nombre	**Bytes**	**Tipo**	**Descripción**
Runtime (segundos)	2	uint16	Número de segundos transcurridos desde encendido del CubeComputer.
Runtime (milisegundos)	2	uint16	Número de milisegundos (después de los segundos) transcurridos desde encendido del CubeComputer.

Cuadro 2.4: Datos de HK de EPS.

Nombre	**Bytes**	**Tipo**	**Descripción**
Voltaje de salida de 3,3V, 5V y 12V	6	3 x int16	Voltaje de salida de los tres buses regulados de 3,3V, 5V y 12V, medido en milivoltios.
Corriente de salida de 3,3V, 5V y 12V	6	3 x int16	Corriente de salida de los tres buses regulados de 3,3V, 5V y 12V, medido en miliamperios.
Voltaje de salida de bus de potencia no regulado	2	int16	Voltaje de salida del bus de potencia no regulado, medido en milivoltios.
Corriente de salida de bus de potencia no regulado	2	int16	Corriente de salida del bus de potencia no regulado, medido en miliamperios.
Voltaje de salida de PDMs	32	16 x int16	Voltaje de salida de los 16 *Power Distribution Modules* (PDM), medido en milivoltios.
Corriente de salida de PDMs	32	16 x int16	Corriente de salida de los 16 *Power Distribution Modules* (PDM), medido en miliamperios.
Voltaje de batería	2	int16	Voltaje de la batería, medido en milivoltios.
Corriente de batería	2	int16	Corriente de la batería, medido en miliamperios.
Temperatura de batería	24	12 x int16	Temperatura de la batería, medida en grados centígrados.
Temperatura interna	2	int16	Temperatura interna del EPS, medida en grados centígrados.
Modo de operación	2	int16	Modo de operación actual del EPS (no del satélite).

2.2.2. Experimentos

Durante la ejecución de los experimentos, el computador de misión debe avisar al computador de experimentos para que inicie los experimentos solicitados. El NanoMind enviará una solicitud, indicando que experimento quiere iniciar y el CubeComputer se encargará de empezarlo, contactando con los computadores correspondientes.

Mientras el experimento esté activo, el CubeComputer deberá recoger los datos generados para enviarlos al NanoMind cuando este los solicite periódicamente. A su vez, en el NanoMind se almacenarán todos los datos recogidos. Cuando se termine un experimento y el satélite este en cobertura con la estación de tierra, el NanoMind mandará estos datos para que puedan ser analizados.

2.2.3. Modos de operación

Los modos de operación del satélite dictan que operaciones deben realizar el computador de misión y el computador de experimentos en una determinada etapa o estado de la misión. Estos modos y como el satélite pasa de uno a otro se encuentran reflejados en la Figura 2.7.

Inicialización

Este será el modo en el que se encuentren el NanoMind y CubeComputer cuando se enciendan al recibir potencia del EPS. En este modo ambos computadores realizarán las acciones necesarias para configurar los componentes del satélite y que este se encuentre preparado para ejecutar las operaciones encontradas en el resto de los modos.

En concreto, en este modo el NanoMind seguirá los siguientes pasos:

1. Configurar e inicializar servicios básicos como el reloj y servicios CSP.
2. Usando el EPS, se debe confirmar que se está alimentando a los computadores.
3. Mandar mensajes CSP al EPS, CubeComputer y radio y esperar respuesta para confirmar que están operativos.
4. Configurar la radio.
5. Comenzar a recolectar datos de housekeeping y mandarlos a tierra periódicamente.
6. Esperar el mensaje de CubeComputer que indica que el *detumbling* o el proceso de estabilización del satélite ha sido completado.
7. Desplegar los paneles solares haciendo uso de los comandos del EPS.
8. Esperar a que CubeComputer haya terminado el proceso de inicialización.

Una vez se terminen de realizar estos pasos, el NanoMind pasará automáticamente a modo commissioning y enviará un mensaje al CubeComputer indicando que este también debe cambiar de modo.

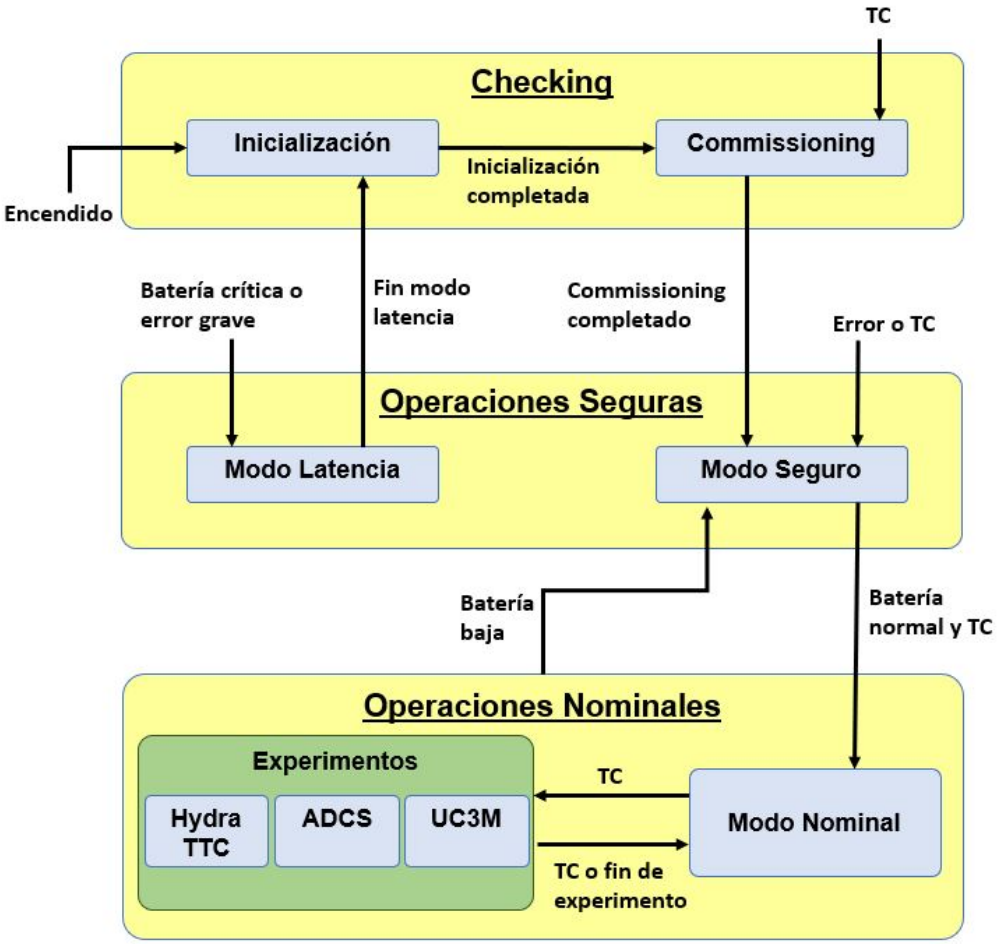

Figura 2.7: Modos de operación.

Commissioning

Se accederá a este modo automáticamente cuando se termine la fase de inicialización o cuando se reciba un telecomando que pida al satélite cambiar a este modo. Cuando el NanoMind se encuentre en este modo realizará las siguientes acciones:

1. Comprobar que el EPS está alimentando al ADCS, DAB y CubeComputer.
2. Mandar mensajes CSP a el EPS, CubeComputer y radio para comprobar que aún hay conexión con ellos.
3. Comprobar que la radio está configurada correctamente.
4. Esperar a que el CubeComputer termine de verificar que el ADCS está funcionando correctamente.

Al terminar estos pasos, tanto NanoMind como CubeComputer automáticamente pasarán al modo seguro.

Modo Seguro

Tras la finalización del modo commissioning se accederá a este modo, donde se comprueba que hay conexión con el resto de computadores antes de comenzar a realizar las funciones básicas de la misión, como la recolección y envío de datos de housekeeping y eventos. Además se puede acceder a este modo de otras formas, ya que se puede usar un telecomando para pasar a este modo desde cualquier otro, o, si se está alcanzando niveles bajos de batería en el modo nominal o de experimentos o ha ocurrido algún error también se regresará al modo seguro.

Modo Nominal y Experimentos

Una vez la batería del satélite se encuentre a niveles normales, se puede acceder al modo nominal desde el modo seguro usando un telecomando desde tierra. Este modo es idéntico al modo seguro, siendo la única diferencia que en modo nominal se pueden ejecutar experimentos. Al recibir un telecomando indicando que experimento se quiere ejecutar, el NanoMind ordenará al CubeComputer comenzar el experimento y a recoger los datos que genera. El CubeComputer mandará estos datos al NanoMind y este los almacenará. Por último, cuando un experimento finalice o cuando se reciba un telecomando para parar el experimento se volverá al modo nominal.

Modo Latencia

En caso de que la batería alcance niveles críticos u ocurra un error grave que impida el correcto funcionamiento del satélite, se accederá automáticamente a este modo sin importar en que modo se encontraba el satélite en ese momento. El objetivo de este modo es consumir poca energía para lograr cargar la batería, esto se logra apagando todos los computadores del satélite salvo la radio, NanoMind y EPS para que el satélite pueda recibir telecomandos pero consumir la menor cantidad de energía posible. Al llegar a este modo, el computador de misión creará un temporizador que indica el tiempo que se debe esperar antes de salir de modo latencia y una vez ese temporizador llegue a cero se alimentará de nuevo al resto de computadores y se volverá al modo de inicialización.

Cobertura

Para especificar si el satélite se encuentra en cobertura y es capaz de mandar mensajes a la estación de tierra, se hace uso de un modo separado al resto de modos explicados en esta sección y que funciona de forma paralela a estos llamado modo cobertura. Para saber si se está en cobertura o no se envían una serie de mensajes entre la estación de tierra y el microsatélite, mostrados en la Figura 2.8, siendo este método el mismo usado en el UPMSat-2 [11].

Para acceder a este modo el NanoMind debe recibir un mensaje de tierra que indica que el satélite se encuentra en cobertura y cuánto tiempo se calcula que lo estará. Una vez enviado este mensaje, NanoMind debe confirmar que lo ha recibido y procede a esperar a que se terminen de enviar telecomandos desde

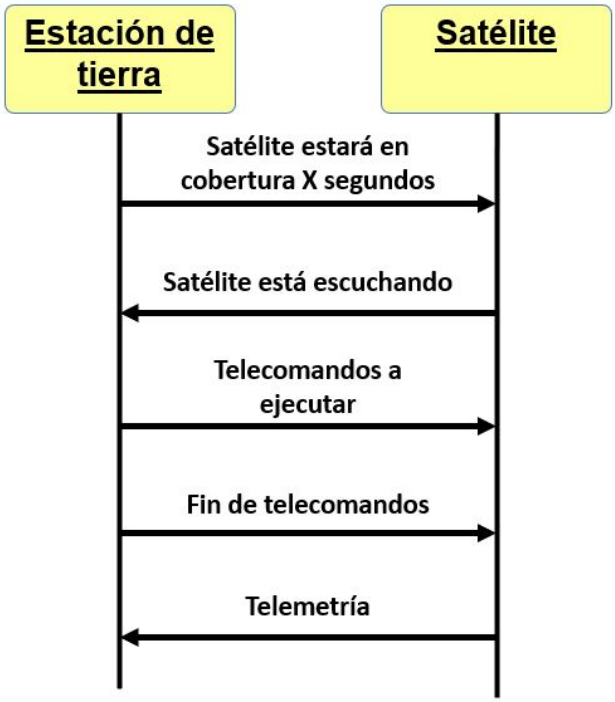

Figura 2.8: Intercambio de mensajes de modo cobertura.

tierra. Posteriormente, para indicar que se terminó de enviar telecomandos, se envía un mensaje desde la estación de tierra confirmando que el computador de misión no se debe de ejecutar más telecomandos en esta órbita y que puede enviar toda la telemetría este ha almacenado.

2.2.4. Eventos

El satélite cuenta con un sistema para almacenar eventos o sucesos significativos ocurridos durante vuelo para luego enviarlos a tierra. Estos eventos pueden consistir en cambios de modo, cambios en los niveles de batería, experimentos en ejecución y terminados, entre otros.

Los eventos se almacenarán en el NanoMind, por lo que cualquier evento que haya ocurrido fuera de este computador debe de ser enviado al NanoMind para que este lo almacene y envíe. Además, estos eventos pueden generar acciones como cambios de modo, por ejemplo, si se detecta batería baja se enviará un evento al NanoMind y se pasará a un modo de operación de bajo consumo.

A continuación, en la Tabla 2.5 se muestra la lista de eventos que se pueden generar:

Cuadro 2.5: Lista de eventos.

Evento	**Descripción**
Change mode	Indica que ha ocurrido un cambio de modo.
Lost Communication	El satélite ha perdido la conexión con tierra.
TC Invalid	Se ha recibido un telecomando no válido.
Detumbling completed	Se ha terminado el detumbling del *Attitude, Determination, and Control System* (ADCS).
Initialization completed	Se ha completado la fase de inicialización.
Commissioning completed	Se ha completado la fase de commissioning.
Normal Battery	La batería ha alcanzado niveles normales.
Low Battery	La batería ha alcanzado niveles bajos.
Critical Battery	La batería ha alcanzado niveles críticos.
Experiment ADCS started	El experimento ADCS se ha iniciado.
Experiment ADCS completed	El experimento ADCS se ha completado.
Experiment ADCS aborted	El experimento ADCS se ha acabado debido a un error.
Experiment TTC started	El experimento HYDRA-AMSAT TTC se ha iniciado.
Experiment TTC completed	El experimento HYDRA-AMSAT TTC se ha completado.
Experiment TTC aborted	El experimento HYDRA-AMSAT TTC se ha acabado debido a un error.
Experiment TTC rebooted	El experimento HYDRA-AMSAT TTC se ha reiniciado debido a un error.
Experiment UC3M started	El experimento de radiómetros UC3M-GREMA se ha iniciado.
Experiment UC3M completed	El experimento de radiómetros UC3M-GREMA se ha completado.
Experiment UC3M aborted	El experimento de radiómetros UC3M-GREMA se ha acabado debido a un error.
Change mode CC	Indica que ha ocurrido un cambio de modo en el CubeComputer.
Initialization completed CC	Se ha completado la fase de inicialización en el CubeComputer.
Commissioning completed CC	Se ha completado la fase de commissioning en el CubeComputer.

2.2.5. Envío de Telemetría

Tras almacenar los datos de housekeeping, eventos y experimentos estos se deben de enviar a tierra. Los datos se deben enviar en dos casos diferentes:

- **Asíncronamente**: Cuando el satélite se encuentra en cobertura, es decir, la estación de tierra se puede comunicar con el satélite, se enviarán datos de eventos, experimentos y housekeeping en este orden. Primero se envían todos los eventos almacenados, empezando por los más recientes, a con-

tinuación, se envían los datos de experimentos, empezando por los más recientes también y por último los datos de housekeeping.

- **Periódicamente**: Mientras el satélite no se encuentre en cobertura, se enviarán datos de housekeeping. Este paquete de datos se denomina beacon o baliza, y tiene el mismo formato que el paquete de datos de housekeeping. Mientras el satélite no se encuentre en cobertura, se recolectarán estos datos y se enviarán inmediatamente a tierra.

Los paquetes que contienen los datos a enviar a tierra seguirán el siguiente formato, detallado en la Tabla 2.6:

Cuadro 2.6: Formato de paquete de telemetría.

Nombre	**Bytes**	**Tipo**	**Descripción**
Sequence Number	4	uint32	Este número indica el número de paquetes de telemetría enviados desde el arranque del computador de misión.
Telemetry Type	1	uint8	Indica cual tipo de datos de telemetría se encuentran en este paquete. ▪ 0 - Beacon ▪ 1 - Eventos ▪ 2 - Housekeeping ▪ 3 - Experimentos
Telemetry Data		uint8[]	Datos de telemetría. Dependiendo del tipo de telemetría del paquete esta parte del paquete tomará formatos diferentes. ▪ Beacon y Housekeeping: Los datos de telemetría serán la unión de las tablas Tabla 2.1, Tabla 2.2, Tabla 2.3 y Tabla 2.4 en este orden. ▪ Eventos: Se seguirá el formato mostrado en la Tabla 2.7. ▪ Experimentos: Se seguirá el formato mostrado en la Tabla 2.8.

Cuadro 2.7: Formato de datos de telemetría de evento.

Nombre	**Bytes**	**Tipo**	**Descripción**
Mission time	4	uint32	Número de segundos transcurridos desde que el NanoMind se encendió hasta que ocurrió este evento.
System mode	1	uint8	Modo de operación del satélite cuando se creó el evento.

Nombre	Bytes	Tipo	Descripción
Event contents	1	uint8	Número que indica cual es el evento que ha ocurrido, empezando por 0. El número de cada evento se corresponde con su posición en la Tabla 2.5.

Cuadro 2.8: Formato de datos de telemetría de experimentos.

Nombre	Bytes	Tipo	Descripción
Mission time	4	uint32	Número de segundos transcurridos desde que el NanoMind se encendió hasta que se guardaron estos datos de experimento.
Experiment type	1	uint8	Número que indica al experimento que pertenecen estos datos. Puede tomar los siguientes valores: ▪ 0 - HYDRA-AMSAT TTC. ▪ 1 - Radiómetros UC3M-GREMA. ▪ 2 - ADCS.
Experiment data		uint8[]	Datos del experimento.

2.2.6. Telecomandos

El UPMSat-3 puede recibir una serie de comandos enviados desde la estación de tierra denominados como telecomandos. Dependiendo del comando enviado, el NanoMind deberá realizar una función u otra, siendo algunos de estos comandos los siguientes:

- Petición de cambio de modo para ir a modo commissioning, seguro o nominal.
- Petición de inicio de experimento.
- Cambio de valor de los diferentes umbrales de los niveles batería.

Todos los telecomandos se podrán ejecutar de manera inmediata, cuando estos sean recibidos por el satélite, o de manera diferida, especificando en que momento se quiere que el comando se ejecute.

Capítulo 3

Entorno y herramientas de desarrollo software

Al trabajar con sistemas embebidos y de tiempo real, el framework o entorno de desarrollo del que se tiene que hacer uso para desarrollar el software de misión debe de adaptarse a las características de estos sistemas. Teniendo en cuenta este hecho, el entorno de desarrollo utilizado para este proyecto se compone principalmente de tres elementos de vital importancia mostrados en la Figura 3.1.

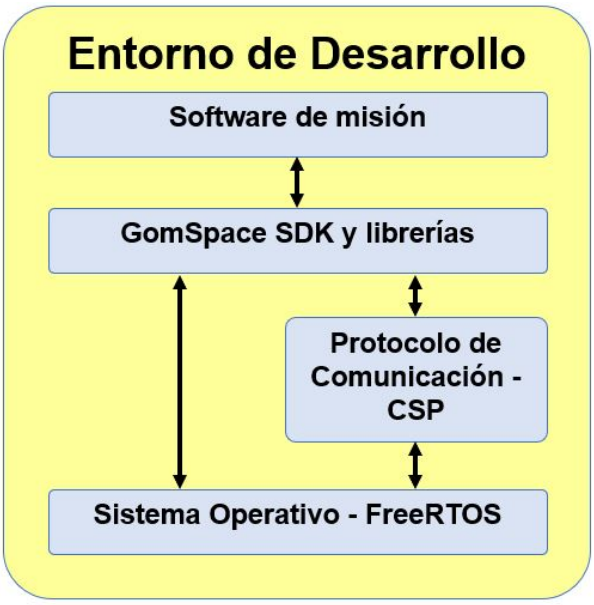

Figura 3.1: Esquema del entorno de desarrollo.

Comenzando por el componente de más bajo nivel, tenemos el sistema operativo **FreeRTOS**. FreeRTOS se trata de un *Real Time Operating System* (RTOS) o sistema operativo de tiempo real diseñado para su uso en microcontroladores y, en concreto, sistemas embebidos. Haciendo uso de este sistema operativo, se pue-

den ejecutar varios hilos o *tasks* en el software de misión además de implementar sistemas de concurrencia como mutex y semáforos. El resto de componentes de los que vamos a hablar en este apartado hacen uso de FreeRTOS.

Segundo, el protocolo de comunicación usado por la mayoría de los componentes del satélite, como mencionado en el apartado anterior, es el protocolo CSP, implementado en la librería **libCSP**. Esta pila de comunicaciones ofrece comunicación entre diferentes sistemas embebidos de una red de pequeña escala y posee un diseño similar al del protocolo TCP/IP. CSP está adaptado para que funcione en FreeRTOS.

Por último, y a más alto nivel, se dispone de las librerías y funciones aportadas por el **Command & Managment Software SDK** y el **Mission Library** de los componentes NanoMind y NanoCom MS100. Estas librerías hacen uso de las funciones aportadas por FreeRTOS y *libCSP* para implementar su propio software centrado en ofrecer operaciones de utilidad para desarrollar el software de misión del satélite.

El objetivo de este apartado es explicar, en detalle, el funcionamiento y cometido de estos tres componentes y por qué son piezas fundamentales para desarrollar el software de vuelo.

3.1. Sistema operativo FreeRTOS

FreeRTOS se trata de un sistema operativo de tiempo real, diseñado para ser utilizado en sistemas embebidos debido a su simplicidad y pequeño tamaño. Este sistema operativo ofrece una serie de operaciones que garantizan la correcta ejecución de varios procesos, denominados por el sistema como *tasks* o tareas, junto herramientas como semáforos, mutex y temporizadores para crear comunicación entre ellas [12].

Los sistemas operativos de tiempo real como FreeRTOS son utilizados para ofrecer un patrón de ejecución determinista, que asegura que el comportamiento del sistema es predecible. En el caso de FreeRTOS esta característica se alcanza asignando a cada tarea o *task* una prioridad. Las prioridades de cada tarea son decididas por el propio usuario del sistema.

Es importante aclarar que, a pesar de que FreeRTOS afirma poder ejecutar varias tareas a la vez, este sistema no ofrece la ejecución de tareas de forma paralela, sin embargo, FreeRTOS soporta la ejecución concurrente de varias tareas, donde la tarea que está siendo ejecutada por el procesador cambia rápidamente, ejecutando así varios fragmentos de cada una de las tareas en una ventana pequeña de tiempo.

Estos momentos en los que se cambia la tarea a ejecutar son denominados como *ticks*. En cada *tick* el planificador o *scheduler*, responsable de decidir que tarea se debe ejecutar siguiendo una serie de reglas, se encarga de parar la tarea en ejecución si es necesario para ejecutar otra (cambio de contexto). En el caso de FreeRTOS, el *scheduler* elegirá que tarea se debe de ejecutar usando un algoritmo Round-robin, además de tener en cuenta la prioridad de las tareas

[13]. El intervalo en el que se cambia de *task* suele tomar un valor entre 1 y 10 milisegundos.

Es importante destacar que el computador de misión NanoMind, donde se ha desarrollado la mayor parte del software de misión, hace uso de la versión de FreeRTOS 10.2.0 [14] y, para simplificar, esta será la única versión de la que se hablará en el resto del apartado. Seguidamente, se explicarán las funciones que más se van a usar de FreeRTOS. Cabe destacar que estas no son las únicas, pero son las que más se usarán en el protocolo CSP y las librerías de *GomSpace*.

3.1.1. Creación de tareas

El aspecto más fundamental de FreeRTOS y del resto de los sistemas operativos de tiempo real son las tareas o *tasks*. En FreeRTOS se ofrecen varias funciones que facilitan la creación de estas.

La primera de estas funciones es **xTaskCreate**. Haciendo uso de esta función podemos crear una nueva instancia de una tarea, para ello se necesita indicar a la función los siguientes elementos:

- Nombre de la tarea.
- Puntero a la función C que va a realizar. Este tipo de funciones están implementadas de tal forma que forman un bucle infinito del cual no se debe salir.
- Tamaño de la pila de la tarea.
- Prioridad de la tarea. Este parámetro es un número entero, siendo 0 la prioridad más baja y -1 simboliza la máxima prioridad.

Al crear una tarea con la función xTaskCreate, la RAM necesaria para crear la pila de la tarea y para guardar el estado actual de la tarea (*Task Control Block* o TCB) se asigna automáticamente del heap de FreeRTOS. Si el objetivo del usuario es que él mismo sea el que realice esta asignación de memoria, se debe usar la función **xTaskCreateStatic**. Esta función, aunque similar a la anterior, necesita dos parámetros adicionales:

- Un puntero a la pila de la tarea. La pila es un array de variables de tipo StackType_t de tamaño decidido por el usuario.
- Un puntero a la variable en donde guardar el TCB de la tarea. En concreto, la variable debe de ser de tipo StaticTask_t.

Para crear tareas en el NanoMind se va a usar la función xTaskCreate o xTaskCreateStatic dependiendo de la configuración usada. En este proyecto, el NanoMind hará uso de la función xTaskCreate. Por otro lado, CubeComputer hace uso de la función xTaskCreateStatic para crear las tareas de forma estática.

3.1.2. Esperas de tareas

En muchas ocasiones se desea que una tarea se detenga durante un periodo de tiempo determinado antes de continuar. Por ejemplo, al solicitar datos de un experimento al CubeComputer, el NanoMind periódicamente mandará solicitudes a este. Cuando los datos se han recibido, el NanoMind esperará el tiempo indicado antes de volver a mandar una solicitud. Este funcionamiento se puede aplicar con **vTaskDelay** y **vTaskDelayUntil**.

Usando estas dos funciones se puede bloquear la tarea durante un número de *ticks* determinado. Usando la función **vTaskDelay** se debe indicar el número de *ticks* en los que queremos que la tarea esté bloqueada mientras que con la función **vTaskDelayUntil** se debe usar un tiempo absoluto con el que se quiere que la tarea se quede bloqueada hasta llegar a dicho tiempo.

3.1.3. Semáforos

FreeRTOS nos permite crear una variedad de semáforos con diferentes características. Entre estos semáforos tenemos los **Counting Semaphore**, los **Binary Semaphore** y los **Mutex Semaphore**.

Los **Counting Semaphores** son aquellos semáforos que pueden tomar un valor entero mayor o igual a cero. Para crear estos semáforos podemos usar dos funciones diferentes: Primero tenemos **xSemaphoreCreateCounting**. Para llamar a esta función se deben indicar cual es el valor inicial del semáforo y cual es el máximo valor que puede alcanzar. Para crear un semáforo se debe de guardar un pequeño espacio en RAM para almacenar el estado de cada semáforo, es por esto que, de manera similar a la creación de tareas, si queremos que sea el propio usuario el que se encargue de reservar dicho espacio se debe de usar la función **xSemaphoreCreateCountingStatic**.

En segundo lugar, tenemos los **Binary Semaphores**. Estos semáforos solo tienen dos estados, cero o uno y, de manera similar a los semáforos anteriores, se pueden crear con las funciones **xSemaphoreCreateBinary** y **xSemaphoreCreateBinaryStatic**. Siempre que se cree un semáforo binario su valor inicial será 0, no se le puede dar un valor inicial.

Por último, están los **Mutex Semaphore**. Estos semáforos son muy similares a los semáforos binarios, pero tienen pequeñas diferencias. Los semáforos binarios son mejores a la hora de implementar sincronización entre tareas mientras que los semáforos mutex son mejores para implementar funciones de exclusión mutua, ya que la prioridad de la tarea que tiene el mutex aumentará temporalmente si una tarea con prioridad superior intenta acceder a la sección crítica protegida por el mutex, haciendo así que el mutex se desbloquee solo cuando salga la tarea que originalmente estaba en la sección crítica.

Estos semáforos mutex se pueden crear con las funciones **xSemaphoreCreateMutex** y **xSemaphoreCreateMutexStatic** de la misma forma que se crean los semáforos binarios, pero también se dispone de las funciones **xSemaphoreCreateRecursiveMutex** y **xSemaphoreCreateRecursiveMutexStatic** crean-

do así semáforos mutex recursivos que permiten a una tarea bloquear el mutex varias veces a la vez, pero luego lo debe de desbloquear el mismo número de veces.

Para mandar una señal y hacer una espera con cualquiera de estos semáforos se usan las siguientes operaciones:

- xSemaphoreGive y xSemaphoreTake
- xSemaphoreGiveRecursive y xSemaphoreTakeRecursive: Estas funciones se deben usar en los semáforos mutex recursivos.
- xSemaphoreGiveFromISR y xSemaphoreTakeFromISR: Se deben usar en una rutina de interrupción o *Interrupt Service Routine* (ISR).

3.1.4. Colas

Para almacenar datos de forma ordenada FreeRTOS ofrece funciones para facilitar la creación de *queues*, o colas FIFO. Estas colas se deben crear con **xQueueCreate**, indicando cual es el número máximo de elementos que puede poseer la cola y el tamaño de cada elemento. Si se desea que el espacio en RAM sea reservado automáticamente de el FreeRTOS heap se debe usar xQueueCreate, pero si por el contrario es el propio usuario el que desea reservar este espacio se debe usar **xQueueCreateStatic**.

Si se desea obtener datos de la cola se hace uso de **xQueueReceive** y **xQueueReceiveFromISR**, especificando de que cola se quieren obtener los datos y en donde guardarlos. Si se quieren obtener datos en una rutina de interrupción se debe usar xQueueReceiveFromISR.

Por el contrario, si se quiere introducir datos en las colas, se usan las funciones **xQueueSendToFront** y **xQueueSendToBack**, para introducir un elemento en el inicio o final de la cola. Además, en caso de querer realizar estas acciones en una rutina de interrupción se usan las funciones **xQueueSendToFrontISR** y **xQueueSendToBackISR**.

3.2. CubeSat Space Protocol y libCSP

El satélite UPMSat-3, al igual que otros muchos microsatélites, están compuestos por un conjunto de computadores diferentes que deben de ser capaces de poder comunicarse entre sí para realizar diversas operaciones. Para facilitar la comunicación en este tipo de sistemas, es decir, en redes pequeñas compuestas por sistemas empotrados, se utiliza el protocolo CubeSat Space Protocol, o CSP.

Este protocolo fue diseñado específicamente para este tipo de sistemas, en concreto, como su nombre indica, para CubeSats, pequeños satélites en forma de cubo [4], ofreciendo así un sistema de comunicación relativamente simple para pequeñas redes de computadores como las encontradas en estos satélites.

La librería que implementa el protocolo CSP es la librería *libCSP* [15]. Esta librería está completamente escrita en el lenguaje de programación C y está diseñada

para implementar una pila de protocolos con similar diseño a la pila TCP/IP, pero con paquetes de cabeceras pequeñas de 32 bits.

Esta librería puede correr en una serie de diferentes sistemas operativos como Linux, MacOS y Windows, pero para este proyecto nos interesa que la librería *libCSP* puede correr en el sistema operativo FreeRTOS.

En este apartado se va a hablar específicamente de la versión de **libCSP 1.6**, ya que es la versión que utilizan todos los componentes del UPMSat-3 que tienen esta librería, por lo que hay que considerar que elementos como la estructura de un paquete CSP puede variar con respecto la última versión.

3.2.1. Funcionamiento básico

El protocolo CSP se basa en el envío de paquetes de datos entre diferentes nodos de la red. Cada nodo es uno de los computadores de la red y es representado por un número entero. Para saber la dirección de cada elemento de la red, cada nodo posee su propia tabla de enrutamiento.

Si se quiere que un **nodo envíe un paquete CSP a otro nodo** se deben de seguir los siguientes pasos:

- Establecer una conexión con el nodo al que se va a enviar el mensaje y a que puerto de dicho nodo.
- Obtener espacio del buffer para el paquete CSP.
- Preparar los datos del paquete.
- Enviar paquete.
- Cerrar conexión.

Por otro lado, si se quiere **recibir los datos enviados por otro nodo** se deben de realizar los siguientes pasos:

- Se crea un socket CSP, estableciendo su puerto y cuantos mensajes se guardarán en el backlog.
- Se acepta la conexión desde el socket cuando se vaya a recibir un paquete.
- Se lee paquete y se libera su buffer.
- Se cierra conexión.

Juntando estas dos acciones también se pueden realizar **transacciones**, en las que se abre una conexión para enviar una solicitud y con esa misma conexión se reciben los datos solicitados o, en el caso contrario, se acepta una conexión del socket cuando se recibe una petición y con esa misma conexión se envían los datos que han sido solicitados.

A continuación se definirán en detalle cada paso de estos procedimientos, además de incluir ejemplos en el Apéndice A.

Conexión

Al iniciar el protocolo CSP en un computador, este asigna espacio para un conjunto limitado de conexiones denominado como *connection pool*. Cada conexión CSP dispone de su propia cola de mensajes recibidos de tamaño también limitado. Tanto el tamaño de las colas de mensajes recibidos como el número de conexiones disponibles son modificables.

Para obtener una conexión de la *connection pool* se debe de usar una de las siguientes funciones:

En el nodo **cliente** se debe de usar la función **csp_connect** para iniciar la conexión, para ello se necesitan especificar los siguientes parámetros:

- El nivel de prioridad con el que se mandará el mensaje, tal y como se muestra en la Tabla 3.1.
- Nodo y puerto de destino de los mensajes CSP que van a ser enviados.
- Opciones de configuración de la conexión. Estas opciones afectan principalmente a la estructura y formato de los paquetes a enviar. Estas opciones se encuentran en la Tabla 3.2.

Cuadro 3.1: Niveles de Prioridad para mensajes CSP.

Nombre	**Valor**
CSP_PRIO_CRITICAL	0
CSP_PRIO_HIGH	1
CSP_PRIO_NORM	2
CSP_PRIO_LOW	3

Cuadro 3.2: Opciones de configuración para conexiones CSP.

Nombre	**Valor**	**Significado**
CSP_O_NONE	0x00	Con ninguna opción de configuración.
CSP_O_RDP	0x01	Activar envío con RDP, *reliable datagram protocol*.
CSP_O_NORDP	0x02	En caso de que en la configuración CSP por defecto ya esté activado RDP, desactivarlo para esta conexión.
CSP_O_HMAC	0x04	Activar encriptación HMAC.
CSP_O_NOHMAC	0x08	En caso de que en la configuración CSP por defecto ya esté activado HMAC, desactivarlo para esta conexión.
CSP_O_XTEA	0x10	Activar encriptación XTEA.
CSP_O_NOXTEA	0x20	En caso de que en la configuración CSP por defecto ya esté activado XTEA, desactivarlo para esta conexión.
CSP_O_CRC32	0x40	Activar encriptación CRC32.

Nombre	Valor	Significado
CSP_O_NOCRC32	0x80	En caso de que en la configuración CSP por defecto ya esté activado CRC32, desactivarlo para esta conexión.

Mientras tanto, en el nodo **servidor**, como explicado previamente, se debe crear un socket que se encargue de escuchar para luego aceptar las conexiones entrantes, para ello se deben usar las siguientes funciones en este orden:

- Se crea un socket con la función **csp_socket**, especificando sus opciones de configuración, especificadas en la Tabla 3.3.
- Enlazamos un puerto establecido por el usuario al socket con la función **csp_bind**.
- El socket comienza la escucha con **csp_listen**, indicando el número de conexiones que puede guardar en el backlog para aceptarlas más adelante.
- Se aceptan las conexiones recibidas, de una en una, con **csp_accept**. Al usar esta función se debe establecer cuanto tiempo se va a esperar a que el socket reciba una conexión antes de continuar, pudiendo establecer CSP_MAX_TIMEOUT para que se espere indefinidamente a que se reciba una conexión por el socket.

Cuadro 3.3: Opciones de configuración para sockets CSP.

Nombre	Valor	Significado
CSP_SO_NONE	0x00	Con ninguna opción de configuración.
CSP_SO_RDPREQ	0x01	Requiere envío con RDP, *reliable datagram protocol*, por parte del cliente.
CSP_SO_RDPPROHIB	0x02	Prohíbe recibir mensajes con RDP, *reliable datagram protocol*.
CSP_SO_HMACREQ	0x04	Requiere envío con encriptación HMAC por parte del cliente.
CSP_SO_HMACPROHIB	0x08	Prohíbe recibir mensajes con HMAC.
CSP_SO_XTEAREQ	0x10	Requiere envío con encriptación XTEA por parte del cliente.
CSP_SO_XTEAPROHIB	0x20	Prohíbe recibir mensajes con XTEA.
CSP_SO_CRC32REQ	0x40	Requiere envío con encriptación CRC32 por parte del cliente.
CSP_SO_CRC32PROHIB	0x80	Prohíbe recibir mensajes con CRC32.
CSP_SO_CONN_LESS	0x100	Activa recibo de paquetes enviados sin conexión.
CSP_SO_INTERNAL_LISTEN	0x1000	Flag interna: listen called on socket.

Buffer

Al crear o recibir paquetes CSP estos se almacenan en el buffer. Al iniciar CSP se reserva memoria para los buffers, siendo el número de buffers y tamaño de cada uno definido por el usuario. Estos buffers se almacenan en una cola y se pueden acceder a ellos usando **csp_buffer_get**.

Si se quiere enviar un paquete se debe de llamar a esta función, indicando el espacio mínimo que se necesita reservar para dicho paquete. Esta función devolverá un puntero al paquete que se completará con los datos necesarios y se enviará. De la misma forma, al recibir un paquete la librería deberá reservar espacio en el buffer.

Hay que considerar que también se debe vaciar los buffers cuando sea necesario, usando **csp_buffer_free**, por ejemplo, cuando se recibe un paquete CSP y ya se ha terminado de leer.

Enviar y recibir

Para realizar el envío de mensajes CSP se hará uso principalmente de la función **csp_send**. Dada la conexión y el paquete CSP que enviar, esta función primero preparará el paquete, añadiendo los cambios necesarios en función de las opciones de configuración de la conexión elegidas por el usuario. Tras terminar de aplicar las configuraciones necesarias al paquete se mandará al siguiente nodo de la cadena o *nextstep* a través de la interfaz que se señala en la tabla de enrutamiento del nodo.

Aunque csp_send es la función principal que se usa para enviar mensajes, también se puede usar la función **csp_sendto**. Esta función, al contrario que csp_send, envía paquetes CSP sin conexión, es debido a esto que todos los parámetros que se usarían para crear una conexión se deben usar al llamar a csp_sendto, ya que no han sido previamente identificados al no haberse creado la conexión. A pesar de realizarse el envío sin crear una conexión, esta función funciona de manera casi idéntica a la explicada previamente, usando la tabla de enrutamiento para buscar el siguiente nodo al que se debe enviar el mensaje.

Por último, si se quiere recibir un mensaje se debe de usar **csp_read**. Esta función esperará a recibir un mensaje CSP de una conexión por un tiempo establecido por el usuario, llegando a ser incluso infinito si este lo desea. Este mensaje se encuentra en la Rx Queue de la conexión, una cola FIFO creada por funciones de FreeRTOS, por lo que está haciendo csp_read realmente es llamar a xQueueReceive de FreeRTOS para recuperar el mensaje.

3.2.2. Pila de protocolo

Como se mencionó previamente, el protocolo CSP posee un diseño similar al protocolo TCP/IP, por lo que el diseño de su pila de protocolo también tiene las mismas características a la pila del protocolo TCP/IP. Esta pila se compone de cuatro capas, tal como se especifica en la Figura 3.2. A continuación se explicarán estas capas, comenzando por la más baja.

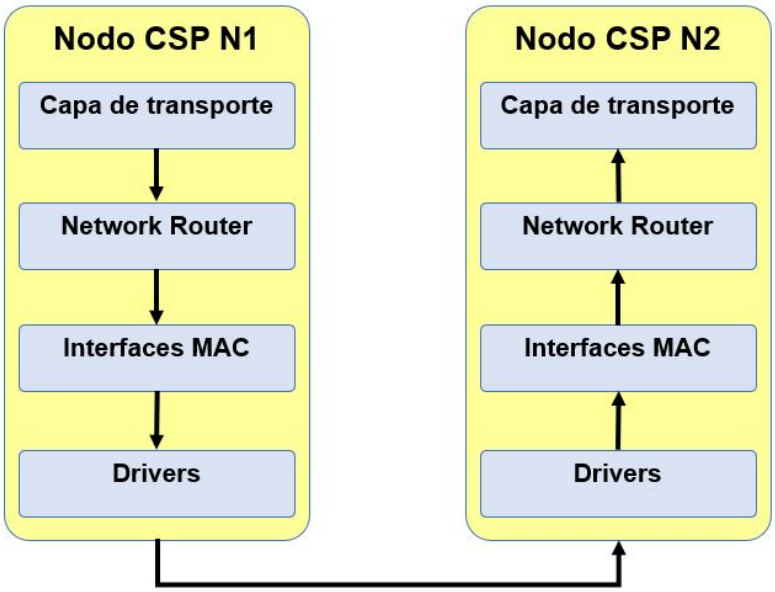

Figura 3.2: Pila de protocolo CSP.

Capa 1: Drivers

Esta capa, al igual que en el protocolo TCP/IP, es la encargada de recibir y enviar paquetes CSP a través de la red. La librería *libCSP* incluye dos drivers, driver socket CAN y driver USART, y aunque estos sean los únicos, se pueden incluir más como se verá cuando se analicen las librerías aportadas por *GomSpace* ya que la compañía añade también sus propios drivers de CAN, I^2C y KISS.

Capa 2: Interfaces MAC

En esta versión de *libCSP* se dispone de las siguientes interfaces: I^2C, CAN, KISS, LoopBack y ZMQHub. En la inicialización de CSP, se crearán todas estas interfaces y se añadirán a una linked list.

De estas interfaces podemos destacar la interfaz LoopBack, que es la interfaz que se usa al enviar un mensaje CSP de un nodo a él mismo. Esta interfaz se utiliza mucho sobre todo para pruebas.

También tenemos la interfaz ZMQHub, que hace uso de la librería open-source de mensajería ZeroMQ, que se usará sobre todo en la estación de tierra, para recoger datos de las dos radios y ordenarlos.

Capa 3: Network router

Esta capa mira la cabecera CSP de cada paquete para ver hacia que nodo y puerto se debe dirigir para enviar el mensaje al siguiente nodo que le acerque a su destino. Esto se logra gracias a las tablas de enrutamiento presentes en cada nodo.

Para prevenir que el paquete CSP se envíe entre los mismos nodos en bucle sin que pueda llegar a su destino, *libCSP* fuerza a los paquetes a salir por una interfaz distinta por la que se ha entrado al nodo.

Capa 4: Capa de transporte

Para que el usuario pueda decidir entre aplicar un protocolo de comunicación fiable o uno más simple la capa de transporte ofrece dos protocolos diferentes, el protocolo Reliable Datagram Protocol (**RDP**) y User Datagram Protocol (**UDP**).

El protocolo RDP implementa elementos como handshakes, reordenación y retransmisión de paquetes, pero como deseamos que el satélite UPMSat-3 tenga un sistema de comunicación lo más simple posible, se ha decidido sacrificar la fiabilidad que ofrece este protocolo y usar el protocolo UDP.

El uso del protocolo UDP no garantiza que lleguen todos los paquetes enviados, o que estos se reciban en orden ya que no implementa ninguna función como las que implementa RDP, pero de esta forma el envío y recepción de datos será más rápido. Esto es esencial ya que el UPMSat-3 debe enviar una gran cantidad de datos en un tiempo limitado, es decir, cuando se encuentra en cobertura, y que se pierdan una pequeña cantidad de paquetes no afectará a las operaciones del satélite.

3.2.3. Routing table

Para saber hacia dónde dirigir los paquetes CSP se debe usar la routing table o tabla de enrutamiento. Esta tabla contiene una serie de rutas que deben tener el siguiente formato: **<dirección>[/máscara] <interfaz>[siguiente paso]**. El significado de cada uno de estos elementos se encuentra explicado en la Tabla 3.4.

Cuadro 3.4: Elementos de ruta CSP.

Nombre	**Descripción**
Dirección	Nodo de destino. Se comparará con la dirección presente en la cabecera del paquete CSP.
Máscara	Indica cual es el número de bits de la dirección de la cabecera del paquete que deben coincidir con el campo de dirección, siendo 5 una coincidencia exacta y 0 una ruta por defecto, en caso de que no coincida con ninguna otra. Este campo es opcional.
Interfaz	El nombre de la interfaz a usar al enrutar el paquete.
Siguiente paso	El nodo al que se debe dirigir el paquete en el siguiente paso. Si es 255 significa que el siguiente paso es el destino final del paquete. Este campo es opcional.

CSP ofrece dos implementaciones distintas de esta tabla, primero una implementación estática que no hace uso de máscaras distintas de 0 o 5, ofreciendo así solo rutas que tienen que coincidir por completo y rutas por defecto, siendo la opción más rápida a la hora de buscar rutas, pero esto hace que las tablas sean

más costosas de configurar, ya que requieren más rutas diferentes. La segunda implementación es *Classless Inter-Domain Routing* (CIDR). Esta implementación sí que permite usar diferentes máscaras, haciendo las búsquedas más lentas pero ofreciendo una configuración más simple. Está última implementación es la que se usa en los computadores del UPMSat-3.

3.2.4. Small Fragmentation Protocol SFP

Si se quiere enviar una gran cantidad de datos a través de CSP se debe de usar el protocolo *Small Fragmentation Protocol* o SFP. Usando este protocolo, un mensaje CSP grande se fragmentará y se mandará en varios paquetes CSP más pequeños y de un mismo tamaño propuesto por el usuario.

Para enviar y recibir paquetes SFP se usan las funciones **csp_sfp_send** y **csp_sfp_recv** respectivamente. Usando estas funciones se mandarán todos los paquetes CSP que forman el mensaje, incluyéndose en cada paquete una cabecera SFP que indica el offset de los datos de cada paquete y el tamaño total del mensaje.

Este tipo de envío conlleva riesgos, ya que si se pierde un solo paquete de los que forman el mensaje completo el cliente SFP dejará de recibir el resto de paquetes y no formará el mensaje, perdiendo así todos los datos enviados.

3.2.5. Estructura de un paquete CSP

Los paquetes CSP se caracterizan por su simplicidad, conteniendo cabeceras pequeñas. La estructura de un paquete CSP estándar es la que se muestra en la Tabla 3.5.

Cuadro 3.5: Estructura estándar de paquete CSP.

	Cabecera	**Longitud de datos (en bytes)**	**Datos**
Longitud	32 bits	16 bits	Variable

En donde la cabecera sigue la estructura de la Tabla 3.6.

Cuadro 3.6: Cabecera de paquete CSP.

	Cabecera					
	Prioridad	Nodo de origen	Nodo de destino	Puerto de destino	Puerto de origen	Flags
Longitud	2 bits	5 bits	5 bits	6 bits	6 bits	8 bits

Se ha estado hablando de la estructura paquetes CSP estándar ya que, dependiendo de las flags establecidas, la estructura puede variar. Los valores de las flags son aquellos encontrados en la Tabla 3.7. Hay que tener en cuenta que puede haber varias flags activas a la vez.

Cuadro 3.7: Flags de paquetes CSP.

Nombre	**Valor**	**Significado**
CSP_FFRAG	0x10	Uso de fragmentación SFP.
CSP_FHMAC	0x08	Uso de verificación *Hash-based Message Authentication Code* (HMAC).
CSP_FXTEA	0x04	Uso de encriptación *Extended Tiny Encryption Algorithm* (XTEA).
CSP_FRDP	0x02	Uso de protocolo de comunicación RDP.
CSP_FCRC32	0x01	Uso de checksum CRC32.

En concreto, lo que va a variar de la estructura es el campo de datos. Este elemento del paquete, que contiene los datos establecidos por el usuario, puede incluir bits extras dependiendo de las flags. También hay que considerar que el valor del campo 'longitud de datos' aumentará debido a estos bits extra.

Si la flag de fragmentación SFP está activa se añadirá 32 bits que indican el offset de los datos del paquete con respecto al mensaje entero y 32 bits que muestran el tamaño total del mensaje, mostrado en la Tabla 3.8.

Cuadro 3.8: Paquete CSP de protocolo SFP.

	Cabecera	**Longitud de datos**	**Datos**		
			Datos de mensaje	Offset	Tamaño total
Longitud	32 bits	16 bits	Variable	32 bits	32 bits

Si se activa la verificación HMAC se van a sumar 32 bits al campo de datos que contengan el valor del HMAC calculado por la librería. Se puede ver el paquete en la Tabla 3.9.

Cuadro 3.9: Paquete CSP con verificación HMAC activa.

	Cabecera	**Longitud de datos**	**Datos**	
			Datos de mensaje	HMAC
Longitud	32 bits	16 bits	Variable	32 bits

Usando la encriptación XTEA no solo se van a encriptar los datos del paquete, sino que además se añadirán 32 bits al final de este indicando el número aleatorio *nonce* que se usa para asegurar que el mensaje sea único. En la Tabla 3.10 se encuentra la estructura de un paquete con esta flag activa.

Cuadro 3.10: Paquete CSP con encriptación XTEA activa.

	Cabecera	**Longitud de datos**	**Datos**	
			Datos de mensaje	XTEA Nonce
Longitud	32 bits	16 bits	Variable	32 bits

De manera similar al resto de flags, si se usa CRC32 se añadirá un checksum de 32 bits al final del paquete tal y como se muestra en la Tabla 3.11.

Cuadro 3.11: Paquete CSP con CRC32 checksum.

	Cabecera	**Longitud de datos**	**Datos**	
			Datos de mensaje	CRC32 Checksum
Longitud	32 bits	16 bits	Variable	32 bits

Por último, si se usa el protocolo RDP los paquetes contendrán una cabecera RDP. Esta cabecera ocupa un total de 5 bytes o 40 bits como se puede visualizar en la Tabla 3.12. Dado que esta cabecera es un poco más compleja que las cabeceras añadidas por otras flags y que este protocolo no se va a usar, no se va a entrar en más detalle al respecto.

Cuadro 3.12: Paquete CSP con protocolo RDP activo.

	Cabecera	**Longitud de datos**	**Datos**	
			Datos de mensaje	Cabecera RDP
Longitud	32 bits	16 bits	Variable	40 bits

3.3. SDK y librerías de soporte

Los productos de *GomSpace* encontrados en el UPMSat-3 y en la estación de tierra poseen su propio software y librerías desarrolladas por la compañía. Con el MS100 Command & Management SDK, MS100 Mission Library, A3200 Command & Managment SDK y el A3200 Mission Library, tanto el MS100 de la estación de tierra y el computador de misión NanoMind ofrecen su propio código y librerías que se pueden editar para adaptarse a la misión [6].

Además, aunque no se pueda editar, las radios AX100 presentes tanto en el satélite como en la estación de tierra también tienen software desarrollado por la compañía con funciones compatibles y similares a aquellas encontradas en las librerías del MS100 y NanoMind.

En esta sección se hablarán de las librerías y funciones más utilizadas en el proyecto.

3.3.1. GOSH

GomSpace Shell o GOSH, es una interfaz de comandos presente en todos los productos de *GomSpace*. Frecuentemente esta interfaz es accesible directamente a través de un puerto en serie USART encontrado en cada dispositivo, debido a que se usa para realizar pruebas en tierra y no en vuelo. Una imagen de la interfaz se puede observar en la Figura 3.3.

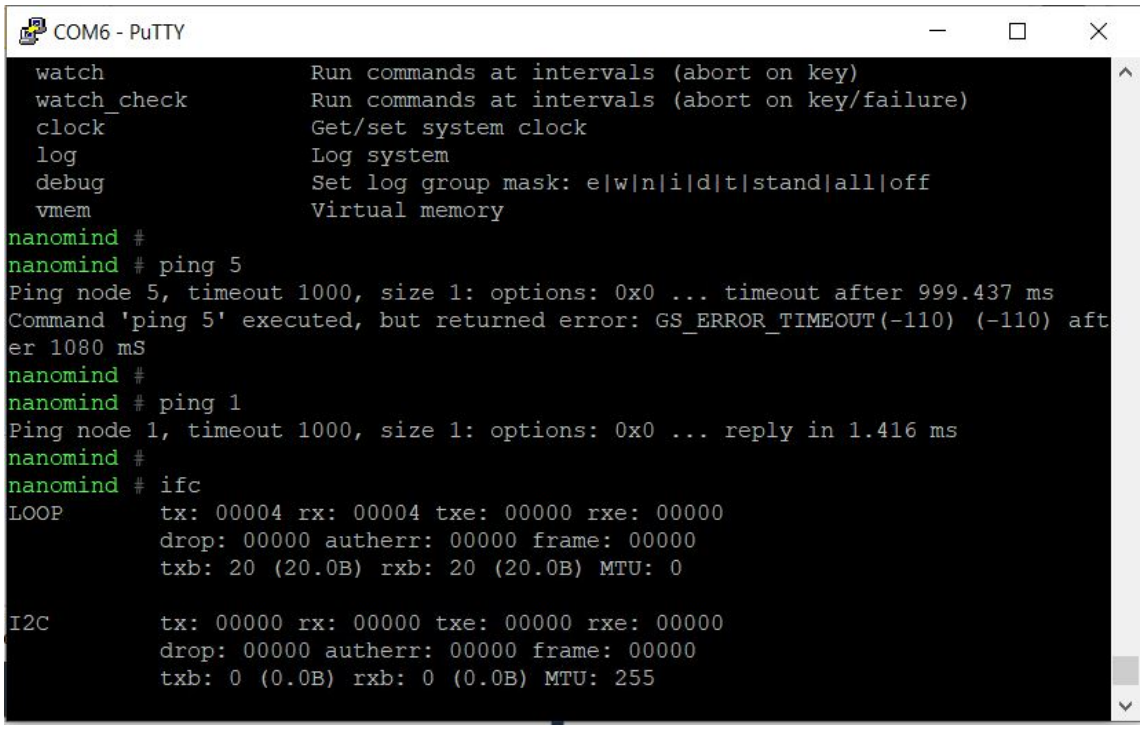

Figura 3.3: Terminal GOSH.

Usando esta interfaz se dispone de diversos comandos de las librerías que permiten alterar la configuración del sistema, ver logs, visualizar telemetría y otras muchas funciones ideales para realizar pruebas. Además, en el MS100 y NanoMind se pueden crear comandos gracias al SDK para poder usarlos posteriormente en esta interfaz.

Tanto el MS100, NanoMind y AX100 cuentan con esta interfaz, pero hay que tener en cuenta que los comandos presentes en cada uno de los dispositivos no son idénticos ya que no todos poseen las mismas funciones.

3.3.2. libgscsp

Para adaptar la librería *libcsp* a los productos de *GomSpace* se hace uso de la librería *libgscsp*. Esta librería, además de las funciones ya encontradas en la librería *libcsp*, ofrece otras funciones como drivers de CAN, I^2C y KISS y comandos para su uso en la interfaz GOSH que sirven para inspeccionar y analizar el tráfico CSP.

3.3.3. Flight Planner (libfp)

La librería *libfp* de NanoMind y MS100 se encarga de ejecutar comandos diferidos en timestamps indicados por el usuario. Tras indicar que comandos se quieren ejecutar y cuando, el servicio denominado como *Flight Planner Executor* se encarga de ejecutarlos. Este servicio está compuesto por un único hilo, por lo que solo se puede ejecutar un comando a la vez, pero se pueden programar hasta un máximo de 5000 peticiones o entradas, conteniendo cada una un solo comando.

Se puede pedir que dichos comandos se ejecuten en un momento específico ya sea absoluto o relativo al actual. Si se establece que un comando se ejecute en un momento relativo, por ejemplo, dentro de 10 segundos, además se puede indicar que se repita varias veces.

3.3.4. Sistema de parámetros (libparam)

Todos los productos de la compañía *GomSpace* hacen uso del sistema de parámetros para guardar datos de telemetría y configuración. Este es un sistema intuitivo donde cada dato o parámetro se encuentra guardado en una tabla de parámetros. Cada tabla posee varios parámetros, cada uno con una dirección dentro de dicha tabla que se corresponde a una dirección en memoria, de esta forma, con uso de operaciones GET y SET se puede editar y acceder a cada parámetro a través de este sistema, sin que el usuario tenga que acceder directamente a memoria.

Cada una de estas tablas puede ser guardada en diferentes sitios, normalmente se tiene una versión de la tabla en memoria volátil y otras en memoria persistente, ya que, en el caso de que los datos de una tabla no se puedan leer ya sea porque están corruptos o erróneos, se cargue otra de las tablas guardadas.

Para facilitar el acceso a los parámetros, la librería *libparam* además ofrece una serie de comandos GOSH, que permiten editar y visualizar parámetros, tanto localmente como en remoto.

3.3.5. Scheduler

El computador NanoMind ofrece un hilo denominado *scheduler* diseñado para realizar tareas periódicamente. Este hilo de baja prioridad llamará a las funciones indicadas por el usuario en intervalos regulares. Los intervalos de tiempo a los que se realizará cada función se deben de configurar usando el sistema de parámetros del computador.

3.3.6. Sistema de ficheros (libstorage y libftp)

Haciendo uso de la librería *libstorage*, se aporta a el NanoMind un sistema de ficheros en la memoria flash. Para acceder al sistema de ficheros remotamente, NanoMind y MS100 hacen uso de la librería FTP (*File Transfer Protocol*). Usando esta librería el sistema se puede conectar a un servidor FTP mediante CSP, y una vez conectados se puede modificar, leer y crear archivos en el sistema de ficheros de un computador remoto.

3.3.7. HouseKeeping (libhk)

Para recolectar, enviar y recibir de datos de housekeeping, *GomSpace* ofrece la librería *libhk*, presente en el computador de misión y MS100. Esta librería hace uso de las tablas de parámetros de los diferentes computadores para recoger los datos.

Para especificar qué datos se quieren recolectar y enviar a tierra y cada cuanto tiempo, se debe crear una especificación de beacon. Estas especificaciones se deben de guardar en forma de archivos .json en el sistema de ficheros del computador. En este archivo se debe de indicar también si se enviarán los beacons automáticamente a tierra tras ser recolectados o no. Las muestras pedidas y recolectadas por esta librería se pueden guardar en memoria persistente además de una memoria caché para no perderse en caso de que el sistema se apague o reinicie.

Capítulo 4

Software desarrollado

Una vez establecidas las funcionalidades que debe cumplir el software y en que entorno se va a desarrollar, se lleva a cabo el proceso de desarrollo del software de la misión. Este capítulo se centra en este aspecto del proyecto, específicamente en la definición de una arquitectura software que satisface los requisitos de la misión. En primer lugar, se procederá a explicar en la Sección 4.1 la arquitectura de alto nivel donde se describen los componentes diseñados para el software y sus interacciones generales. A continuación, en la Sección 4.2, se profundizará en las funciones y características de dichos componentes, presentando una arquitectura detallada de estos.

4.1. Arquitectura software de alto nivel

En la siguiente sección va a presentar la arquitectura software estática y dinámica del software de misión. Para diseñar estas arquitecturas se han seguido las directrices del estándar de software espacial "ECSS-E-ST-40C" de la *European Commission for Space Standardization* (ECSS) [16].

4.1.1. Arquitectura estática

La arquitectura estática de un sistema, tal como se indica en el estándar de software "ECSS-E-ST-40C" [16], define los componentes software principales de dicho sistema y sus relaciones. Estos componentes software se comunican e interactúan entre sí con el objetivo de ofrecer las funciones asignadas al sistema, siendo en este proyecto las funcionalidades definidas en el Capítulo 2.

Con la finalidad de implementar las funcionalidades de la misión, se ha diseñado el módulo Mission Manager. Este módulo se ejecuta en el NanoMind y se encargará de gestionar el estado del satélite, analizando la información recogida de sus diferentes componentes y realizando las funciones necesarias dependiendo de dicho estado. Además, este módulo deberá de ser capaz de comunicarse con tierra, enviando información y ejecutando instrucciones. Seguidamente, se detalla el funcionamiento de cada subcomponente del Mission Manager.

Recolección y almacenamiento de datos

Como se muestra en la Figura 4.1, Mission Manager contiene cuatro módulos para la funcionalidad de obtención y almacenamiento de datos. Estos se resumen a continuación:

- **Data Storage**: Representa el almacén donde se depositan los datos de housekeeping, experimentos y eventos. Se leerá el contenido de este almacén para enviarlos a tierra. Data Storage se divide en diferentes almacenes, cada uno asociado a un tipo diferente de datos.
- **Data Collector**: Recoge datos de housekeeping del EPS y del computador de experimentos y datos de experimentos para almacenarlos en el Data Storage. Data Collector también recogerá datos del propio NanoMind, pero por simplicidad, este aspecto no se encuentra representado en el diagrama.
- **Event Manager**: Recoge los eventos generados por el CubeComputer y el propio NanoMind. Dichos eventos se guardan en el Data Storage.
- **Mode Manager**: Implementa los modos de operación del satélite, es decir, la máquina de estados para transicionar entre modos. Guarda el modo de operación actual en el Data Storage.

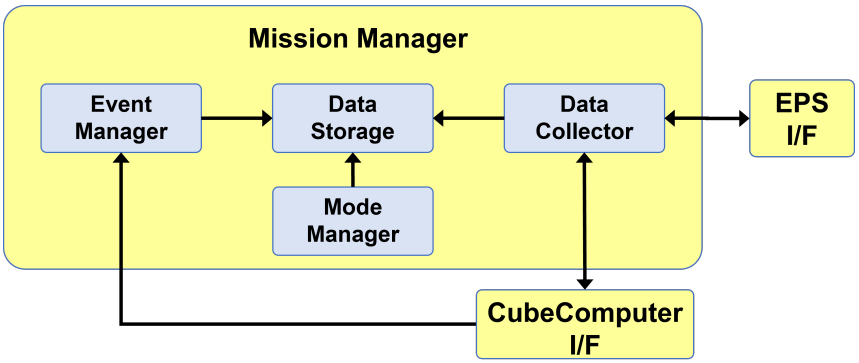

Figura 4.1: Arquitectura estática del manejo de datos de TM.

Recepción y envío de telemetría a tierra

Como se muestra en la Figura 4.2, Mission Manager incluye los módulos Telecommand Receiver y Telemetry Sender para **mandar datos a tierra** cuándo se encuentra el **satélite en cobertura**. Estos módulos se resumen a continuación:

- **Telemetry Sender**: Recoge los datos guardados en Data Storage y los envía a la estación de tierra cuando el satélite se encuentre en cobertura o se enviarán periódicamente en forma de beacon cuando no haya conexión con la estación de tierra.
- **Telecommand Receiver**: Recibe mensajes de tierra que confirmarán que el satélite está en cobertura. Tras recibir estos mensajes, el módulo avisará al Telemetry Sender que se encuentra en cobertura y que este debe enviar todos los datos almacenados.

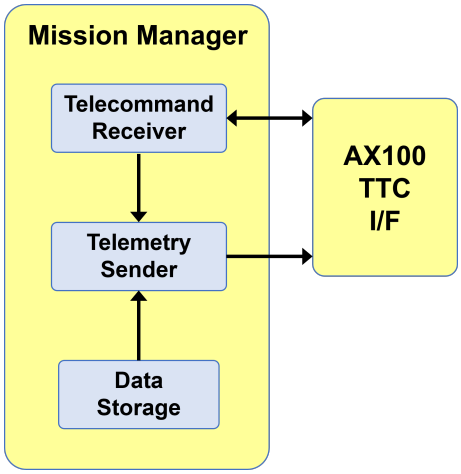

Figura 4.2: Arquitectura estática de intercambio de datos con tierra.

Modos de operación

El módulo Mode Manager también implementa los **modos de operación** del satélite. Como se muestra en la Figura 4.3, este módulo colabora con los módulos Event Manager y Data Collector para llevar a cabo los cambios de modo.

- **Mode Manager**: Realiza los cambios de modo del satélite y ejecuta las funciones definidas en la Subsección 2.2.3. Estos cambios de modo se pueden ocasionar por un telecomando recibido desde tierra o debido a un evento. En función del modo, este puede modificar el comportamiento del Data Collector y la configuración del EPS, AX100 y CubeComputer.
- **Event Manager**: Al recibir eventos que pueden ocasionar un cambio de modo, como cambios en el nivel de batería, este módulo informará al Mode Manager que se debe de realizar el cambio.
- **Data Collector**: La recolección de datos depende del modo en el que se encuentra el satélite. Por ejemplo, en el modo latencia, no se recogerán datos de experimentos ni de housekeeping. En otros modos, como en modo de experimentos, si se debe recoger información de estos.

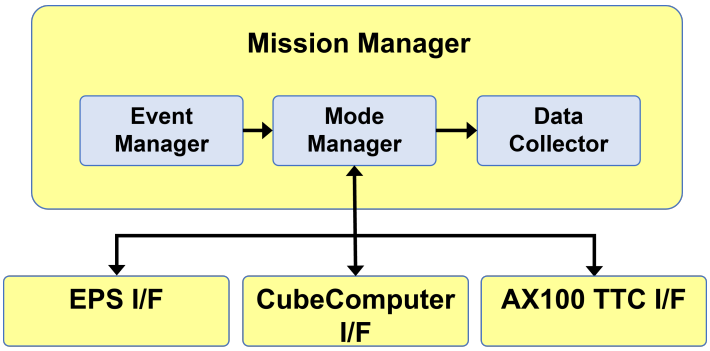

Figura 4.3: Arquitectura estática de gestión de modos de operación.

4.1.2. Arquitectura dinámica

La arquitectura dinámica de un sistema, tal como se indica en el estándar de software "ECSS-E-ST-40C" [16], describe los aspectos del diseño concurrentes y que afrontan las limitaciones del tiempo real del software. Estos aspectos se dividen en tareas o hilos, siendo estas secuencias de instrucciones o tareas que se ejecutan de forma concurrente, y recursos compartidos, objetos protegidos por semáforos u otros mecanismos de sincronización que permiten a las tareas comunicarse entre sí. Para implementar la arquitectura dinámica se ha hecho uso de tareas *FreeRTOS* que se ejecutan de forma concurrente y de almacenes de datos protegidos por mutex (mecanismo de exclusión mutua). Estos mutex, también de *FreeRTOS*, aseguran que el acceso y modificación de los datos se realiza de forma segura, evitando condiciones de carreara.

En los diagramas que representan la arquitectura dinámica del sistema, los paralelogramos representan las tareas o hebras y los rectángulos los datos protegidos por mutex. La función que se ejecuta en cada tarea puede ser periódica, o esporádica, indicado por el icono de la esquina superior izquierda de cada tarea.

Recolección y almacenamiento de datos

La Figura 4.4 muestra la arquitectura dinámica que implementa la recolección y almacenamiento de datos entre Nanomind, EPS y CubeComputer. Para recolectar datos de housekeeping, se usarán las tareas CC, NM y EPS Data Collector, que guardarán datos periódicamente en el almacén de datos del NanoMind.

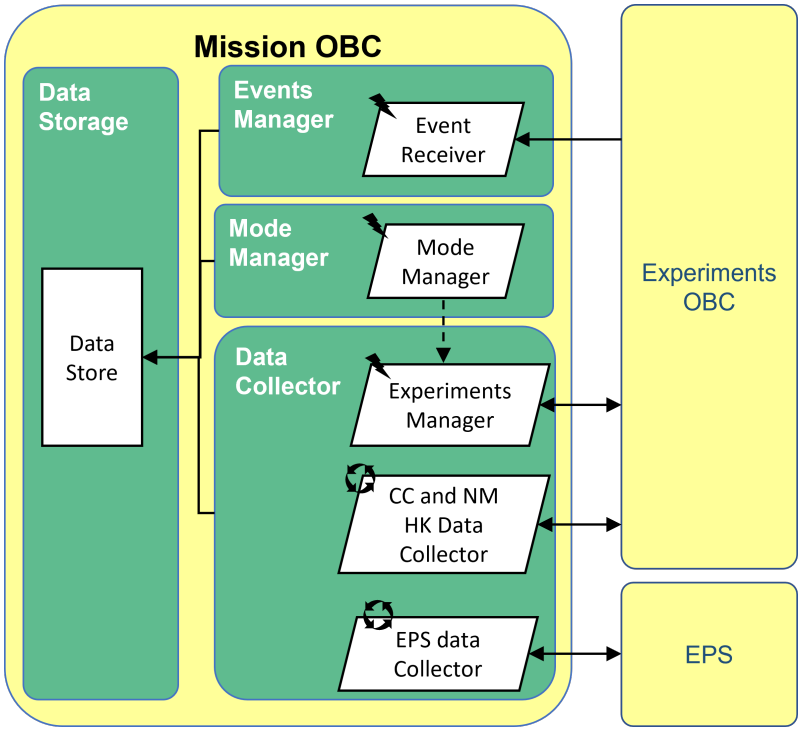

Figura 4.4: Tareas de recolección de datos.

El Experiments Manager recogerá y almacenará los datos de experimentos cuando estos estén activos. Específicamente, recogerá estos datos sólo cuando el Mode Manager notifique de que se está en modo de experimento, en este caso, la tarea Experiments Manager solicitará datos del CubeComputer de forma periódica.

Por último, el propio Mode Manager añadirá el modo de operación actual a la Data Store y el Event Receiver recogerá y almacenará los eventos enviados por el CubeComputer y el propio NanoMind.

Recepción y envío de telemetría a tierra

La Figura 4.5 muestra los módulos encargados de la recepción y envío de los datos de telemetría a tierra. Estos son el Telecommand Receiver y el Telemetry Sender, respectivamente. Para saber si se encuentra en cobertura, el Telecommand Receiver y la estación de tierra transmitirán entre ellos los mensajes mostrados en la Figura 2.8. Existen dos flujos de ejecución:

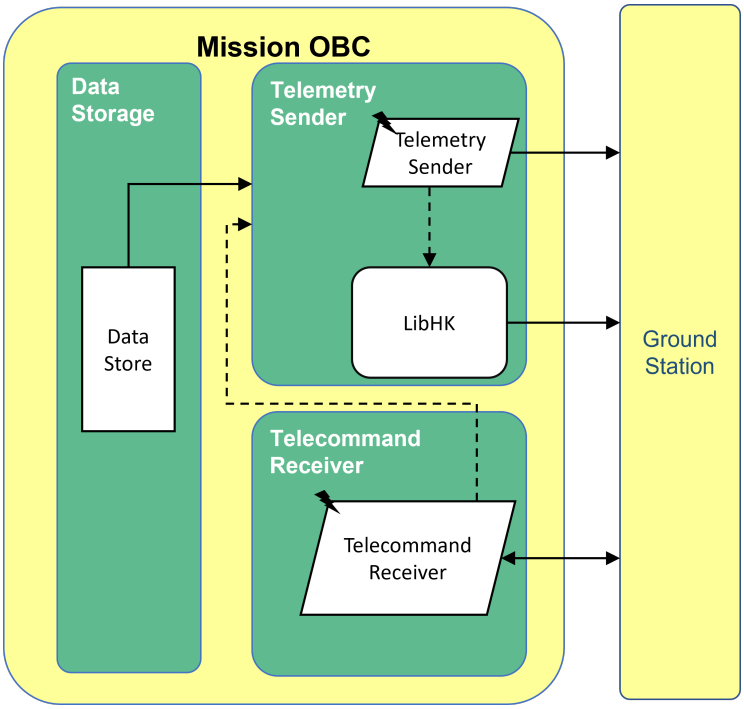

Figura 4.5: Tareas de comunicación con estación de tierra.

- Envío de telemetrías: Cuando se indique desde la estación de tierra que se puede comenzar a enviar datos desde el satélite, el Telecommand Receiver avisará al Telemetry Sender y este enviará datos de eventos, de experimentos y los datos de housekeeping (obtenidos con la librería *libhk*). Estos datos se obtienen de los almacenes presentes en Data Store. Cuando no se está en cobertura, las tareas de *libhk* seguirán enviando beacons de housekeeping.

- Recepción de telecomandos: Las tareas encargadas de ejecutar los telecomandos dependerán del contenido a ejecutar. Cada tarea que ejecute un telecomando leerá mensajes CSP de un puerto diferente. Por ejemplo, si se comanda cambiar el modo del satélite desde tierra, se enviará un mensaje CSP al puerto del que está leyendo la tarea Mode Manager. Tras recibir el mensaje, esta tarea se encargará de leerlo y realizar las instrucciones indicadas. El software del NanoMind no tiene ninguna tarea específica para leer y ejecutar todos los telecomandos posibles.

Modos de operación

Por último, para **gestionar los diferentes modos de operación** y las funciones a realizar en cada uno de ellos se hace uso del Mode Manager, presente en la Figura 4.6. Esta tarea actualizará el modo actual guardado en la Data Store cada vez que se cambie de modo, y, dependiendo del modo, el Experiments Manager, EPS Data Collector y los Housekeeping Collectors deben dejar de recolectar datos si el modo actual lo requiere. Además, el Mode Manager también realizará diferentes operaciones sobre el EPS, CubeComputer y NanoCom AX100 dependiendo del modo. Un cambio de modo puede ocurrir por un evento recibido desde el Event Receiver o por un telecomando de tierra.

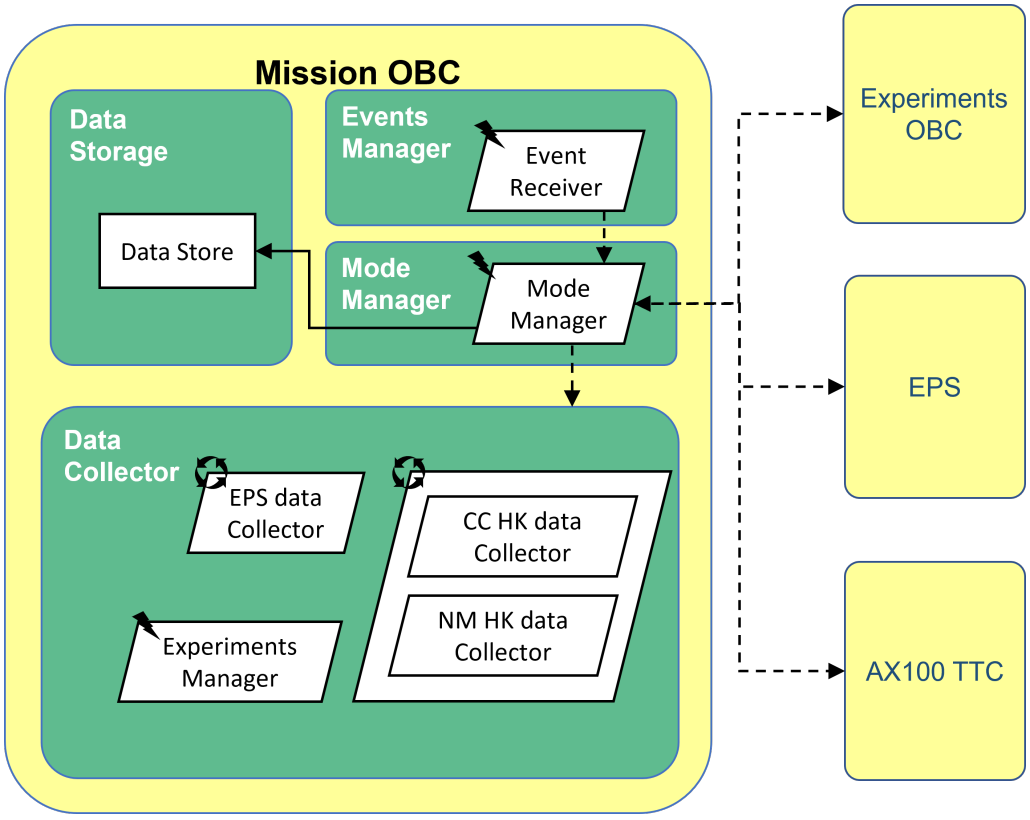

Figura 4.6: Tareas de cambio y gestión de modos de operación.

4.2. Arquitectura software detallada

4.2.1. Componente de HouseKeeping y Experimentos

Para realizar la recolección y almacenamiento de datos de housekeeping y experimentos se hace uso del componente **Data Collector y sus almacenes de datos**. Este componente recoge datos del NanoMind, CubeComputer y EPS y está dividido en tres tareas diferentes: Experiments Manager, EPS Data Collector y el conjunto de CC HK Data Collector y NM HK Data Collector, siendo estas dos últimas funciones ejecutadas en una misma tarea. Este componente y las tareas que lo componen se muestran en la Figura 4.7.

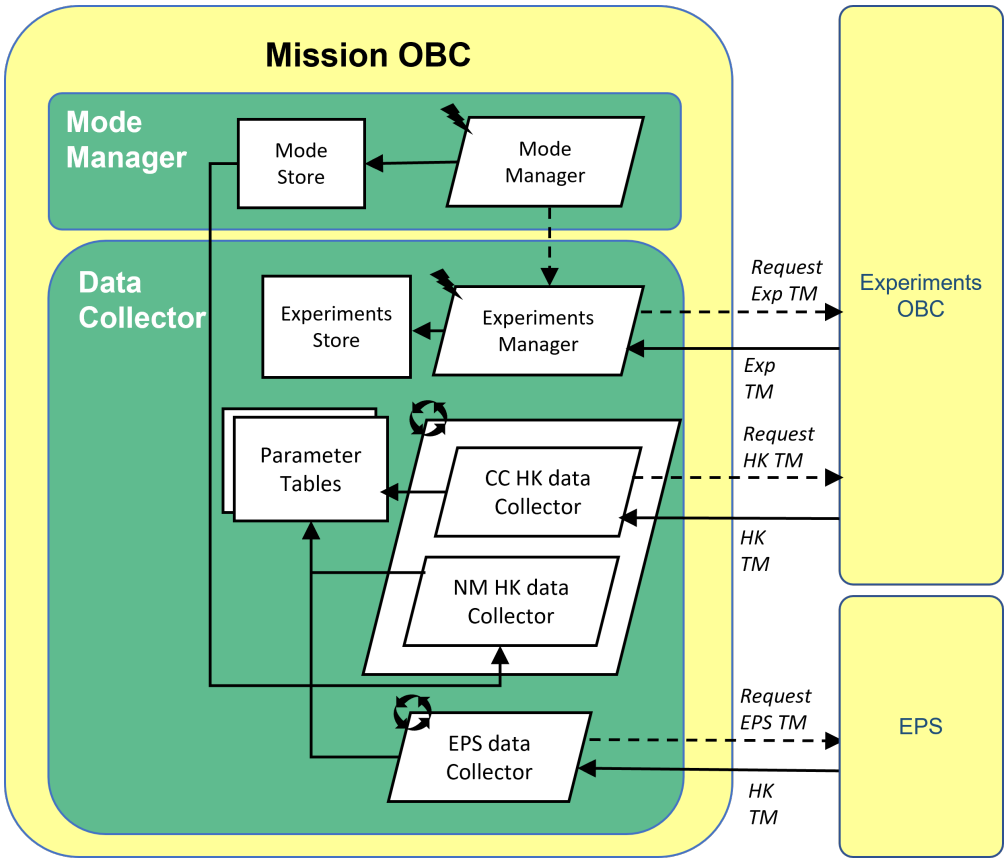

Figura 4.7: Esquema detallado de la arquitectura del componente de recolección de datos de housekeeping y experimentos.

Comenzando por el **EPS Data Collector**, esta tarea periódica se encarga de recoger datos de housekeeping del EPS. Para realizar esta función, la tarea envía una solicitud al EPS vía CSP dirigida al puerto de housekeeping ya abierto en el EPS y este contesta mandando sus datos. Tras recibir los datos estos se almacenan y se comprueba el nivel de batería del EPS, que se puede calcular usando los parámetros presentes en su telemetría de housekeeping. Si este nivel ha variado con respecto a la última vez que se calculó, se debe de enviar un mensaje CSP a través de la interfaz *loopback* a la tarea Event Receiver, para que esta genere un evento.

En este componente también se encuentran el **NM y CC HK Data Collectors**. Estas dos funciones se ejecutan dentro de la tarea **Scheduler** del SDK de NanoMind. Como se ha definido en la Subsección 3.3.5, esta tarea de baja prioridad ejecuta funciones de manera periódica. Es por la baja prioridad de la tarea que el EPS Data Collector, a pesar de ser muy similar al CC y NM HK Data Collectors, se ejecuta en una tarea diferente y de mayor prioridad, ya que el control de la batería que realiza esta tarea es una función prioritaria.

Como su nombre indica, CC HK Data Collector recoge datos de houskeeping del CubeComputer usando el mismo sistema que el EPS Data Collector, recibiendo los datos por medio de un sistema de solicitudes. Por otro lado, NM HK Data Collector recoge datos del propio NanoMind, en concreto, cada vez que se llama a la función, esta almacena el modo de operación actual del sistema, encontrado en la Mode Store. El resto de datos de housekeeping del NanoMind son recogidos y almacenados por la librería *libparam* y *libhk*, por lo que no se debe implementar la recolección de estos datos en este componente.

Tanto el EPS Data Collector como las funciones CC y NM HK Data Collectors ejecutadas por el Scheduler almacenan los datos en las **tablas de parámetros** de la librería *libparam*, explicadas en detalle en la Subsección 3.3.4. Tras crearse al inicio del programa tres tablas de parámetros diferentes, una para datos de NanoMind, otra para datos de CubeComputer y una última para datos del EPS, todos los datos obtenidos se almacenarán en estas tablas para que la librería *libhk* del SDK pueda acceder a ellos sin dificultad cuando se quieran enviar a tierra.

Con respecto a la recolección de datos de experimentos, la tarea **Experiments Manager** recogerá datos de manera periódica, mandando solicitudes al CubeComputer que enviará de vuelta los datos de cada experimento, pero al contrario de las otras tareas del componente, esta solo se activará cuando indique el Mode Manager que el satélite se encuentra en un modo de experimento. Más detalles sobre este comportamiento se pueden encontrar en el componente de modos de operación.

Los datos de los experimentos no se almacenan en las tablas de parámetros habituales, sino que se guardan en el **Experiments Store**. Este almacén de datos se comporta de manera diferente a las tablas de parámetros ya que este almacén sigue una estructura LIFO (*Last In First Out*) y al leer un elemento de esta estructura este se borrará de ella, siendo esta la principal razón por la que se tuvo que optar por crear otro almacén diferente.

Junto a los datos del experimento, también se introducirá en el almacén el número de segundos transcurridos desde que el NanoMind se encendió hasta que se introdujeron dichos datos a el Experiments Store y a qué tipo de experimento pertenecen dichos datos. Estos dos elementos se deben de incluir para poder ser enviados a tierra con el formato indicado en la Tabla 2.8.

4.2.2. Componente de Gestión de Eventos

El cometido del componente de gestión de eventos, representado en la Figura 4.8 es recoger los eventos generados por el CubeComputer y NanoMind, almacenarlos y, si se recibe un evento que requiere un cambio de modo, informar al Mode Manager. La tarea principal de este componente es el **Event Receiver**, ya que en esta tarea, se leerá en bucle los mensajes CSP recibidos por el puerto de eventos del NanoMind. Cada paquete que se mande a este puerto contiene un número entero que se corresponde a uno de los eventos encontrados en la Tabla 2.5. Al leer el número, este se introducirá en el **Event Store**, un almacén de datos muy similar en estructura al Experiments Store, siendo una estructura LIFO cuyos datos se eliminan de memoria al ser leídos.

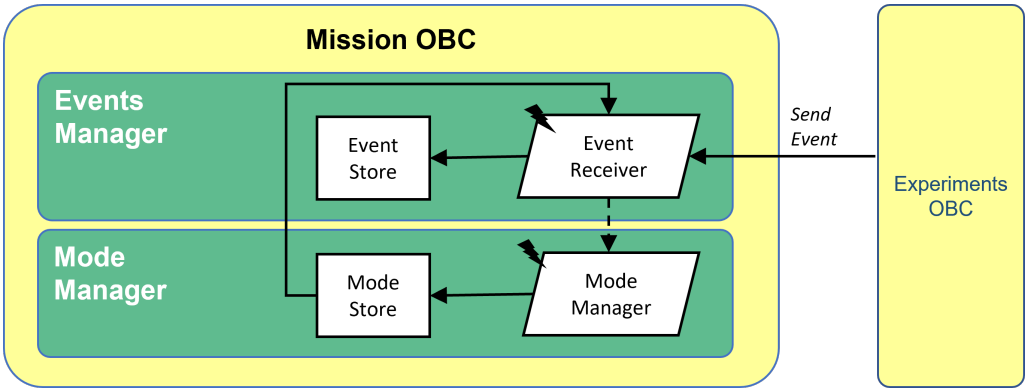

Figura 4.8: Esquema detallado de la arquitectura del componente de gestión de eventos.

Junto al número que identifica al evento, también se almacenará el número de segundos transcurridos desde que se encendió el computador hasta que se insertó el evento en el almacén y el modo de operación del satélite cuando se generó dicho evento, obtenido de el Mode Store. Con estos datos, la telemetría de eventos seguirá el mismo formato que el encontrado en la Tabla 2.7 cuando se envíe a tierra.

Los mensajes CSP de eventos que recibe la tarea Event Receiver pueden proceder del propio NanoMind a través la interfaz *Loopback* o del CubeComputer por la interfaz CAN. De esta forma, enviando todos los eventos por CSP sin importar que se envíen desde el NanoMind o CubeComputer, se recibirán a través de la cola de mensajes CSP ordenados, garantizando así que se lean en orden de llegada sin importar de que computador provenga el evento.

Si el evento recibido por el Event Receiver requiere cambiar de modo de operación, como por ejemplo, si se ha recibido un evento del EPS Data Collector que indica que los niveles de batería son críticos, Event Receiver deberá comunicarse con **Mode Manager** para realizar este cambio. Esta acción provocará que la tarea Event Receiver mande un mensaje CSP de cambio de modo mediante la interfaz *Loopback* al Mode Manager. Hay otros eventos que pueden ocasionar que el Event Receiver interactúe con el Mode Manager sin uso de mensajes CSP, principalmente, existen eventos que interactúan con semáforos *FreeRTOS*

como el evento *Detumbling Completed* que llama a la función *post* del semáforo *Detumbling Semaphore* para asegurar que el Mode Manager continúe con la inicialización solo si el *detumbling* ya está completado.

4.2.3. Componente de Modos de Operación

Este componente gestiona los modos de operación del satélite, usando principalmente la tarea Mode Manager, que se comunicará con el resto de tareas y computadores mostrados en la Figura 4.9. El funcionamiento de estos componentes varía en función del modo de operación del satélite.

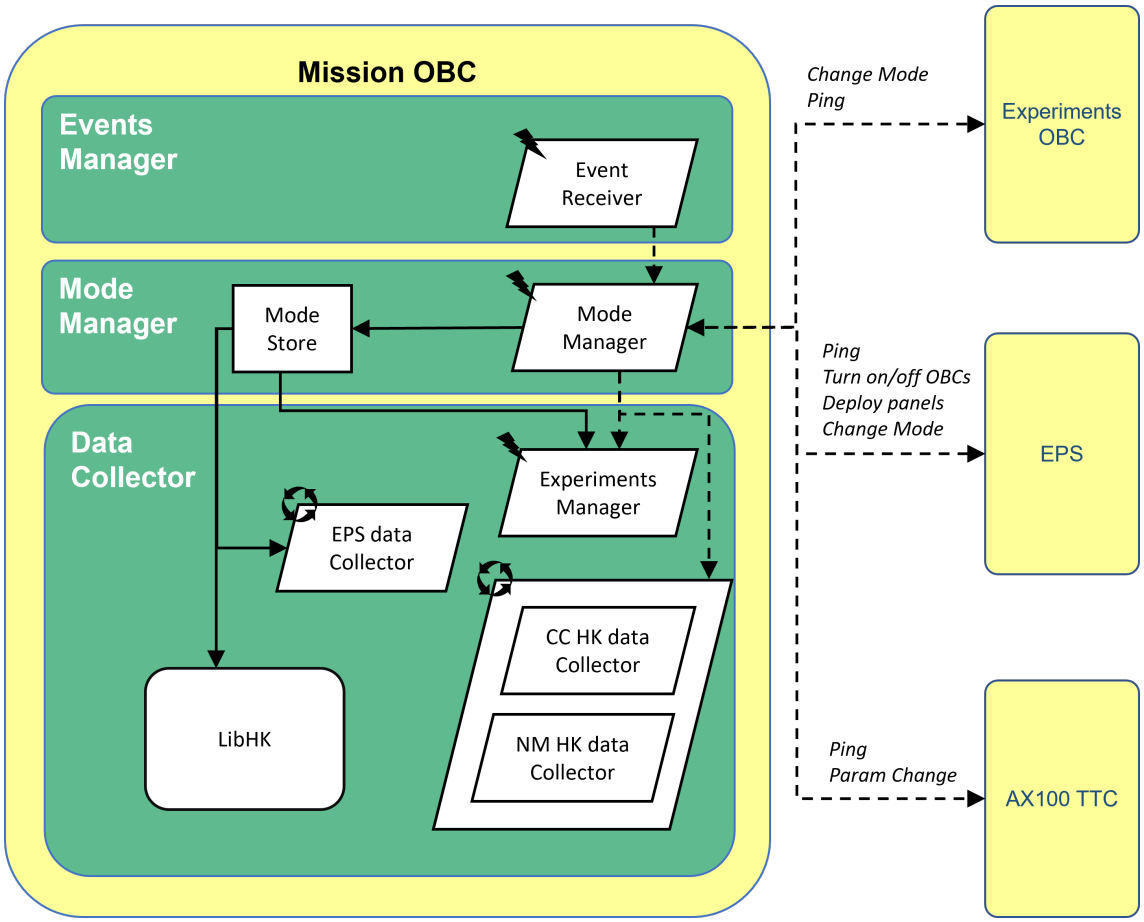

Figura 4.9: Esquema detallado de la arquitectura del componente de cambio y gestión de modos de operación.

El **Mode Manager**, de forma similar a como el Event Receiver lee eventos, leerá solicitudes de cambio de modo en bucle que pueden provenir de tierra, del Event Receiver o, en algunos casos puntuales, de otras tareas del software de misión.

Una vez recibida una solicitud, el Mode Manager decide si este cambio de modo se puede realizar o no, usando la lógica de la Figura 2.7 y dependiendo del modo y nivel de batería actuales. Si se confirma que este cambio se puede realizar, Mode Manager procederá a actualizar el modo de operación actual guardado en

Mode Store. Por último, mandará un evento *Mode Changed* al Event Receiver y comenzará a ejecutar las funciones asociadas a cada modo, mostradas en el Subsección 2.2.3 y detalladas a continuación:

- **Modo inicialización**: Si el modo actual ha pasado a ser el modo inicialización significa que el computador se acaba de inicializar o acaba de salir del modo latencia, por lo que debe de realizar una serie de inicializaciones y comprobaciones básicas, como arrancar las tareas de *libhk*, comprobar que el CubeComputer está encendido usando comandos CSP del EPS, probar la conexión entre el resto de computadores mandando mensajes ping, comprobar que la configuración de la radio es correcta y cambiarla si necesario, accediendo a sus tablas de parámetros de forma remota y comenzar a recolectar datos de housekeeping configurando el scheduler.

  Tras terminar de realizar estas operaciones se esperará, haciendo uso de un semáforo, a que el CubeComputer termine de realizar el proceso de *detumbling* y envíe un evento *Detumbling completed* antes de comandar al EPS que se desplieguen los paneles solares. Finalmente, la tarea esperará, también con un semáforo, a que el CubeComputer termine el proceso de inicialización. Una vez terminados estos pasos, la tarea Mode Manager enviará un evento *Initialization Completed* que hace que el Event Receiver automáticamente envíe una solicitud para pasar a modo commissioning.

- **Modo commissioning**: Las primeras operaciones que se realizarán en este modo será comprobar que el ADCS, DAB y CubeComputer siguen siendo alimentados y cambiar el modo del CubeComputer a modo commissioning. A continuación se ejecutarán funciones idénticas a las del modo de inicialización, como probar la conexión entre el resto de computadores enviando mensajes ping y comprobar que la configuración actual de la radio es la correcta.

  También se debe de esperar, mediante el uso de semáforos, a que se reciba un evento *CubeComputer Commissioning Completed* desde el CubeComputer que indica que este ha terminado su commissioning. Después, Mode Manager enviará un evento que indica que el NanoMind lo ha completado también. Este evento provocará que el Event Receiver mande una solicitud para realizar un cambio a modo seguro.

- **Modo seguro**: Primero, se debe comprobar que el ADCS, DAB y CubeComputer siguen siendo alimentados y cambiar el modo del CubeComputer a modo seguro. Después se procederá a mandar mensajes ping al EPS y radio, incluyendo una comprobación de la configuración de esta.

  En este modo, a diferencia del modo inicialización o commissioning, no se debe esperar ningún evento del CubeComputer ya que para salir de este modo se debe recibir un telecomando desde tierra que cambie el modo.

- **Modo nominal**: El Mode Manager no debe de realizar operaciones en este modo, solo confirmar que el EPS está alimentando a los computadores y cambiar el modo de operación del CubeComputer a modo nominal. Esto se debe a que las operaciones que se realizan en este modo como recolección

de datos y envío automático de beacons se ha iniciado previamente en el modo de inicialización.

- **Modo latencia**: El objetivo principal de este modo de operación es que el satélite consuma la menor cantidad de energía posible, es por esto que el Mode Manager indica al EPS que apague todos los computadores del satélite salvo él mismo, la radio AX100 y el NanoMind, para que se pueda seguir recibiendo telecomandos desde el satélite.

  Tras dejar de alimentar a los computadores, se cambiará al EPS a un modo seguro, que desactiva todas sus *Power Distribution Units* o PDUs y se activará un temporizador con la librería *libfp*, mostrada en el Subsección 3.3.3, que automáticamente mandará una solicitud para cambiar a modo inicialización pasado un tiempo determinado. Cuando el satélite vuelva al modo de inicialización, se cambiará el modo del EPS y se volverá a alimentar al CubeComputer.

  Mientras el satélite se encuentra en modo latencia, no se recolectarán ni enviarán datos de telemetría. Por este motivo Mode Manager parará el scheduler para que no se ejecuten las funciones CC y NM HK Data Collector. Además las tareas de la librería *libhk* y EPS Data Collector comprobarán el modo del satélite accediendo a la Mode Store antes de recolectar y enviar datos de las tablas de parámetros y no realizando estas operaciones si se encuentra en modo latencia.

- **Modo experimentos**: Si se cambia el modo de operación actual a un modo de experimento significa que se debe de iniciar dicho experimento avisando al CubeComputer y recoger los datos generados por este. Primero se cambiará el modo de operación del CubeComputer. Una vez cambiado el modo, este computador arrancará el experimento y comenzará a recolectar datos.

  Tras avisar al CubeComputer, el Mode Manager usará el semáforo *Experiment Semaphore* para indicar al Experiments Manager que este puede comenzar a pedir y almacenar datos de forma periódica. Esta tarea comprobará el Mode Store para saber que experimento se está ejecutando cada vez que se vayan a pedir datos al CubeComputer. Si el modo almacenado deja de ser un modo de experimentos se saldrá de la función periódica y se deberá a esperar a que el Mode Manager desbloquee la tarea otra vez.

4.2.4. Componente de Gestión de Cobertura

Para comunicar el satélite con la estación de tierra y que este sepa si se encuentra en cobertura o no para mandar datos de telemetría se hará uso de las tareas Telemetry Sender y Telecommand Receiver, junto a librerías procedentes del NanoMind SDK. Estos elementos se encuentran reflejados en la Figura 4.10.

Para comprender como se comportan estos componentes, se detallará paso por paso las acciones ejecutadas por cada uno de ellos cuando el satélite se encuentra en cobertura:

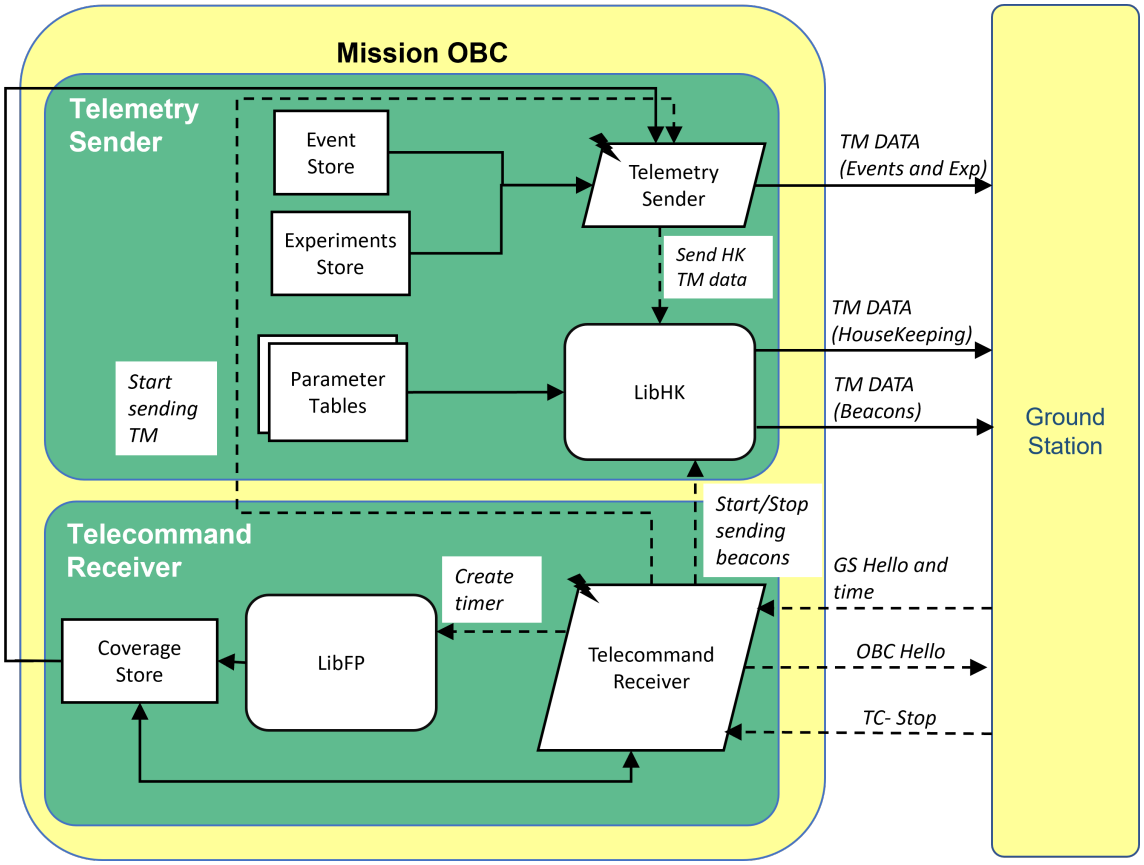

Figura 4.10: Esquema detallado de la arquitectura del componente de gestión de cobertura y envío de telemetría.

1. Siguiendo los pasos definidos en la Figura 2.8, primero el Telecommand Receiver recibirá un mensaje “Hello” o saludo de la estación de tierra, que indica al satélite que se encuentra en cobertura y el tiempo que va a estar en cobertura. Al leer el mensaje y confirmar que el formato sea el correcto se cambiará el valor guardado en la **Coverage Store**. El parámetro guardado en este almacén es un booleano que indica si el satélite se encuentra en cobertura o no, por lo que este parámetro tomará el valor "verdadero".

2. Con ayuda de la librería *libfp* la tarea creará un temporizador que, cuando pase el tiempo indicado por tierra, volverá a cambiar el parámetro almacenado en la Coverage Store, pasando a ser "falso".

3. Telecommand Receiver indicará a la tarea encargada de mandar beacons cuando no hay cobertura, la tarea de servicio de la librería *libhk*, que deje de mandar beacons durante el tiempo que el satélite va a estar en cobertura.

4. La tarea responderá a tierra con su propio mensaje "Hello", indicando que el satélite está escuchando y esperando a los telecomandos que se deben de ejecutar.

5. Telecommand Receiver esperará a que se envíen telecomandos desde la

estación de tierra a otras tareas. El satélite no tiene manera de saber si se han dejado de mandar telecomandos hasta que se recibe de tierra el mensaje "TC Stop", que indica que ha parado de enviar telecomandos y que el computador de la estación de tierra está listo para recibir telemetría.

6. Una vez recibido el mensaje, se debe comprobar que el satélite sigue en cobertura leyendo el valor guardado en la Coverage Store y, si lo sigue estando, se indicará a la tarea Telemetry Sender mediante un semáforo que esta debe de mandar los datos de eventos, experimentos y housekeeping almacenados. Al recibir este aviso, Telemetry Sender abrirá una conexión CSP con el computador de la estación de tierra.

7. Telemetry Sender procederá a enviar todos los datos de telemetría en paquetes CSP individuales, añadiendo su *Sequence Number*, un número que indica el número del paquete de telemetría, tal como se indica en la Tabla 2.6. Algunos mensajes de telemetría como datos de algunos experimentos ocuparán mucho espacio. Por este motivo todos los paquetes que contengan telemetría se enviarán mediante CSP con protocolo SFP y manteniendo de esta forma cohesión entre los mensajes. Es importante destacar que los beacons se envían como mensajes SFP también, pero sin conexión ya que no se están enviando a ningún nodo CSP real.

8. Primero se enviarán todos los datos de eventos, comenzando por los más recientes y comprobando antes de enviar cada dato que se sigue en cobertura.

9. A continuación se enviarán todos los datos de experimentos, comenzando por los más recientes y comprobando antes de enviar cada dato que se sigue en cobertura.

10. Por último se debe de enviar todos los datos de housekeeping que no han sido enviados aún a la estación de tierra, usando la librería *libhk*, comenzando por los datos más recientes, comprobando antes de enviar cada dato que se sigue en cobertura. Esto se logra guardando el timestamp de los últimos datos de telemetría enviados en la última pasada, de esta forma se puede localizar los datos que aún no se han mandado, viendo si la timestamp de los datos que se van a enviar son posteriores al timestamp de los últimos datos enviados.

11. Telemetry Sender cerrará la conexión CSP con tierra, finalizando así el proceso.

Capítulo 5

Verificación y validación del sistema

Con el objetivo de probar el software desarrollado para el UPMSat-3, se ha hecho uso de dos modelos de los componentes del satélite. Primero, se tiene el modelo de vuelo, constituido por componentes que se encontrarán en el microsatélite cuando este se lance. Estos componentes son muy costosos porque están diseñados para entornos espaciales. Por ello, en este proyecto se usan modelos de ingeniería. Estos modelos poseen características similares a los modelos de vuelo, pero son más accesibles y menos costosos ya que no están destinados a ser utilizados en los mismos entornos que los componentes del modelo de vuelo.

Los componentes del modelo de vuelo se describen en la Sección 2.1. Estos componentes eran habitualmente manipulados en el interior de una sala limpia, salvo los computadores que componen la estación de tierra, y requerían elaborar documentos que definen las pruebas realizadas en ellos y sus resultados. Por otro lado, se podía acceder a los componentes del modelo de ingeniería en cualquier momento y lugar, haciendo uso de un escritorio remoto y sin ser necesario tener que elaborar documentación de las pruebas realizadas en ellos.

A pesar de que el modelo de ingeniería debe ser similar al modelo de vuelo para probar el software de misión, hay suficientes diferencias entre los dos que dificultan probar algunas funciones. Principalmente, el modelo de ingeniería no posee equivalentes a todos los componentes del modelo de vuelo. Concretamente, no existen modelos de ingeniería para el NanoCom AX100, la Data Acquisition Board y los computadores de cada experimento. Asimismo, los componentes del modelo de ingeniería presentan ligeras diferencias con respecto a los de vuelo. Por ejemplo, las librerías que acceden a la memoria persistente del NanoMind, no se pueden utilizar en este modelo.

El modelo de ingeniería se compone principalmente de dos placas equivalentes al computador de misión NanoMind y computador de experimentos CubeComputer, además de un modelo de ingeniería del EPS aportado por DHV. Este último no alimenta a las placas como en el modelo de vuelo, ya que estas se alimentan conectándose directamente al PC mediante USB, pero es de utilidad para

probar la interacción de el NanoMind del modelo de ingeniería con la interfaz y comandos CSP del EPS. A continuación, se detallará principalmente el manejo del modelo de ingeniería y de vuelo del NanoMind, ya que este es el computador que contiene el software de misión, además de las pruebas realizadas sobre los modelos de ingeniería y vuelo de diversos componentes.

5.1. Modelos de ingeniería y de vuelo

5.1.1. Modelo de ingeniería

Con el objetivo de probar el software de misión que se cargará en el NanoMind se adquirió la placa **Atmel AT32UC3C-EK** [17] como modelo de ingeniería del NanoMind debido a que este componente es un kit de evaluación o *Evaluation Kit* (EK) para microcontroladores AT32UC3C0512C siendo este el mismo microcontrolador encontrado en el NanoMind de *GomSpace*. Esta placa, además de contener el microcontrolador, posee otra serie de dispositivos como LEDs, interfaces CAN y conector JTAG para cargar y depurar software.

Para alimentar la placa AT32UC3C-EK y poder acceder a la interfaz GOSH que se mostrará cuando el SDK del NanoMind esté cargado se usa el puerto USB *Virtual Com Port* o J27 de la tarjeta de evaluación, como se puede ver en la Figura 5.1. Al conectar la placa a un PC a través de este puerto USB esta recibirá corriente y se podrá, a través del puerto serie del PC, leer y mandar comandos a través de su interfaz GOSH. Una vez conectada la placa y cargado el software, se puede leer la interfaz usando programas que nos permitan acceder a los puertos serie del PC como *TeraTerm* [18] o *Putty* [19], configurando la velocidad de lectura del puerto serie a 9600 bps y configurando el puerto en modo asíncrono, estableciendo 8 bits de datos, no bit de paridad y 1 bit de parada (8n1). La Figura 3.3, encontrada en el capítulo anterior, muestra la interfaz GOSH desde el programa *Putty*.

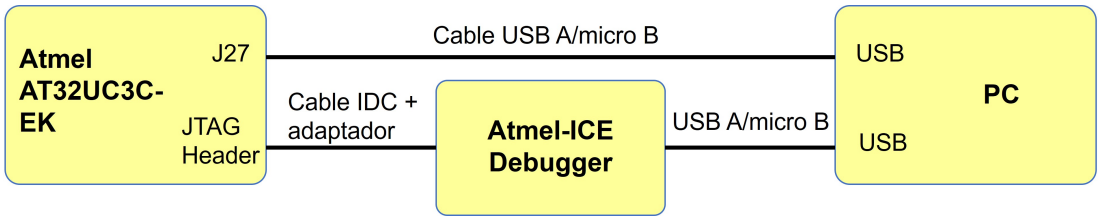

Figura 5.1: Diagrama de conexión de modelo de ingeniería de NanoMind.

Si se quiere cargar software en la tarjeta de evaluación es necesario usar un *debugger* que se conecte a la interfaz JTAG de la tarjeta. En este proyecto se hace uso del **Atmel-ICE Debugger** [20], una herramienta que permite cargar nuestro software haciendo uso del programa *Microchip Studio* [21]. Este programa permite liberar la memoria de la tarjeta y cargar archivos *Extensible Linkable Format* (ELF) en esta, como se puede ver en la Figura 5.2, que muestra la interfaz de *Microchip Studio*. La conexión de este componente con la tarjeta de evaluación y el PC se muestra en la Figura 5.1.

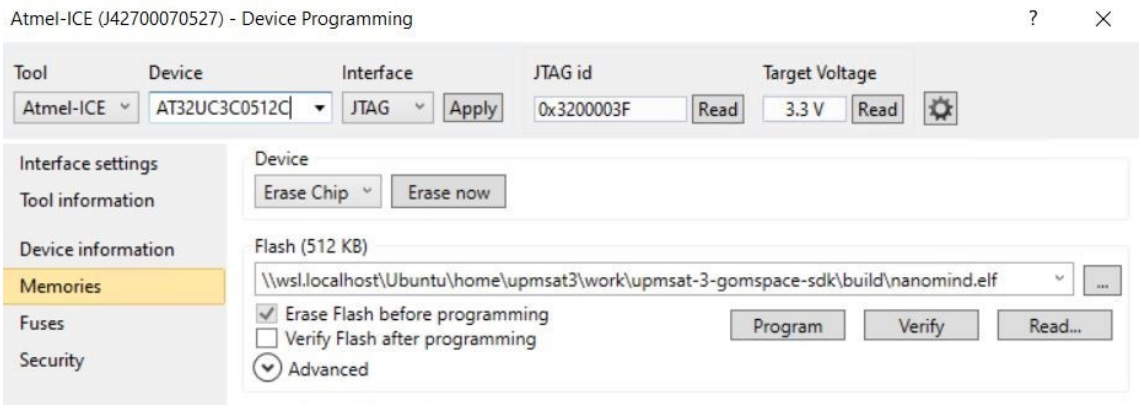

Figura 5.2: Interfaz de Microchip Studio.

Adicionalmente, es posible depurar los mensajes CAN que envían los computadores utilizando el dispositivo **Ixxat SimplyCAN** [22]. Este adaptador se conecta al bus CAN que usan los componentes del modelo de ingeniería y, conectándolo a un puerto USB del PC se pueden visualizar todos los mensajes enviados por el bus. Con esta herramienta y el software *BusMonitor* que incluye, se puede realizar un análisis de los paquetes CSP enviados entre diferentes componentes. En la Figura 5.3 se puede visualizar el programa *BusMonitor* mostrando los mensajes CSP enviados entre el computador de misión y el computador de experimentos del modelo de ingeniería, en concreto siendo dos mensajes CSP, el primero un mensaje ping enviado por el computador de misión y el segundo su respuesta, enviada por el computador de experimentos.

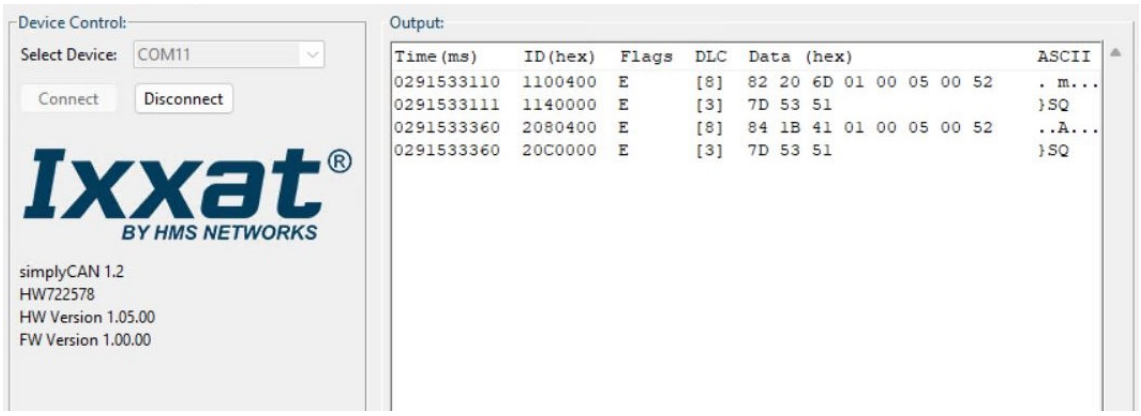

Figura 5.3: Interfaz de BusMonitor mostrando dos mensajes CSP.

Finalmente, la Figura 5.4 muestra el modelo de ingeniería completo. En esta imagen se observa el modelo de ingeniería del CubeComputer, una placa EFM32GG [23] en la esquina superior izquierda conectado a un adaptador CAN MCP2515 [24]. En la esquina superior derecha se halla el modelo de ingeniería del EPS junto al del NanoMind, que se encuentra conectado al Atmel-ICE Debugger, el dispositivo blanco que se sitúa debajo de este. Por último, debajo del modelo de ingeniería del CubeComputer se puede ver el adaptador Ixxat Simply-

CAN, que se encuentra conectado al bus CAN junto al resto de computadores.

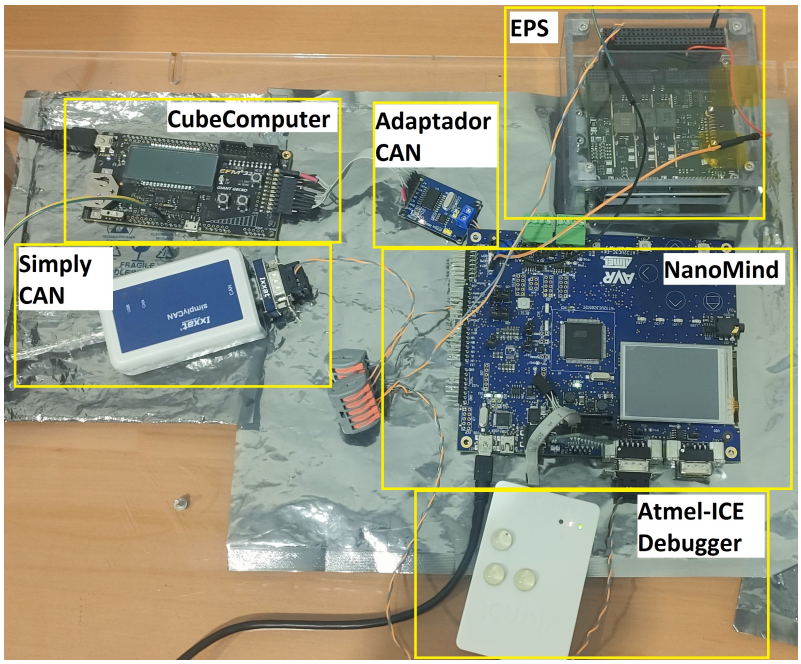

Figura 5.4: Modelo de ingeniería completo.

5.1.2. Modelo de vuelo

El manejo y conexión del modelo de vuelo es muy similar al del modelo de ingeniería, pero hay varias diferencias fundamentales. La primera diferencia está en la forma de alimentar al NanoMind, ya que esta puede cambiar dependiendo de si el NanoMind se encuentra conectado al NanoDock, mostrado en la Subsección 2.1.3. En primer lugar, si no se está haciendo uso del NanoDock, se alimenta de forma muy similar al modelo de ingeniería. En este caso, el NanoMind se debe conectar al PC a través de su puerto P1 mediante un cable FTDI/USB. Estos pasos son los definidos en el *datasheet* del componente [5] y se encuentra representado en la Figura 5.5.

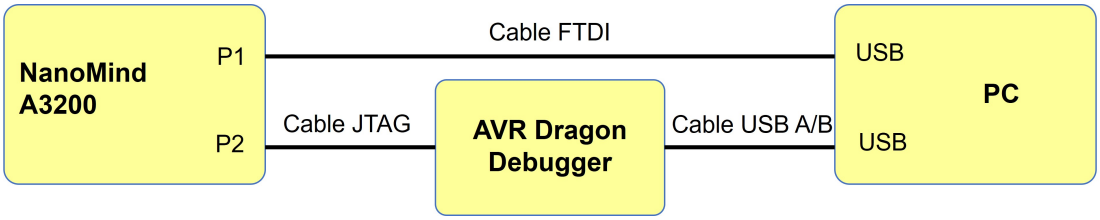

Figura 5.5: Diagrama de conexión de NanoMind sin NanoDock.

Si se desea que el NanoMind se alimente a través del NanoDock se procederá a realizar la conexión representada en la Figura 5.6. Primero se conecta el NanoMind al NanoDock de forma directa como se indica en su *datasheet* [8], introduciendo los conectores X1 y X3 del NanoMind a los conectores X1 y X1-2 del

NanoDock. Una vez realizada esta conexión, se podrá leer la interfaz GOSH del computador a través del conector PicoBlade P1 del NanoDock. Para comenzar a alimentar a los computadores que se encuentran en el NanoDock, este se debe conectar a una fuente externa haciendo uso de sus conectores H2.

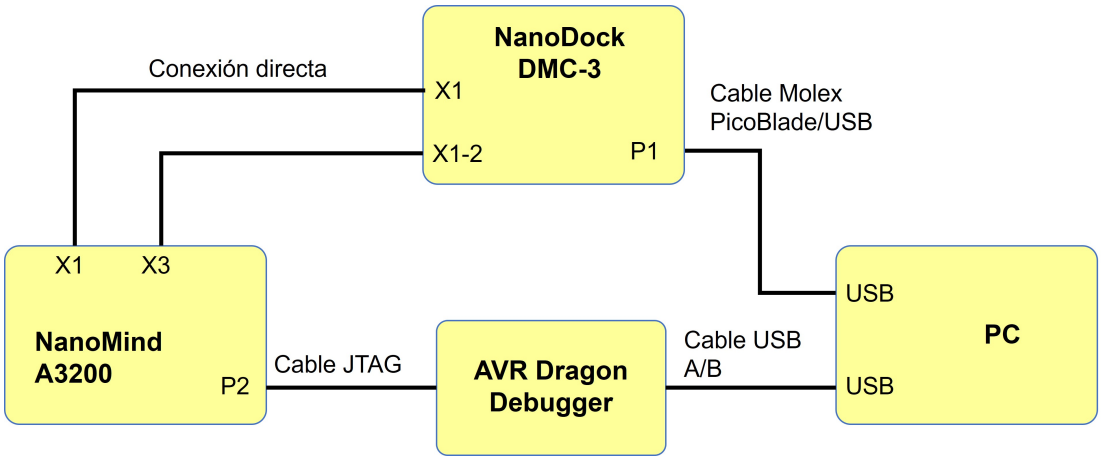

Figura 5.6: Diagrama de conexión de NanoMind con NanoDock.

Una vez realizada la conexión, es necesario usar las mismas herramientas mencionadas en la subsección anterior para acceder a la interfaz GOSH. La configuración que usar para acceder al puerto serie será la misma salvo que la velocidad será de 500000 bps en vez de 9600 bps.

Por último, en comparación con su modelo de ingeniería, la manera de cargar software será diferente. Comenzando con el dispositivo que realiza la escritura en el NanoMind, se hará uso del **Atmel AVR Dragon Debugger** [25], ya que este es el dispositivo que *GomSpace* indica que se debe usar para cargar el software a los computadores NanoMind A3200. Como ya mostrado en la Figura 5.5 y Figura 5.6, se conecta el Atmel AVR Dragon al NanoMind usando un cable JTAG por el puerto P2 del NanoMind y a su vez se conecta el Atmel AVR Dragon al PC mediante un cable USB. Para cargar el software no se hará uso del programa Microchip Studio, sino que se usará la toolchain incluida en el SDK.

5.2. Pruebas

Como explicado al inicio del capítulo, para verificar y validar el software desarrollado este se debe de probar tanto en modelos de ingeniería como de vuelo. No obstante, al encontrarse el proyecto UPMSat-3 aún en desarrollo en el momento que se escribió este documento, no se ha tenido la oportunidad de poder probar todo el software en el modelo de vuelo, ya que solo se han podido realizar pruebas de naturaleza más básica en estos modelos. La mayoría de estas pruebas consistían en arrancar los componentes por primera vez, y comprobar que estos funcionaban correctamente.

Sin embargo, todo el software se ha podido probar en el modelo de ingeniería, salvo algunas funciones puntuales que deben de hacer uso de componentes a los

que no se tiene acceso desde el modelo de ingeniería, como la radio NanoCom. A continuación, se especificarán que pruebas se han realizado en el modelo de ingeniería y de vuelo y los pasos que se han seguido.

5.2.1. Pruebas de conexión entre NanoMind y CubeComputer

A lo largo de la realización del proyecto se ha usado el modelo de ingeniería para probar todo el software de misión, principalmente para probar la comunicación entre el NanoMind y el CubeComputer mediante CSP. Durante estas pruebas se ha verificado el funcionamiento de las siguientes funciones:

- **Recolección de datos de housekeeping**: Esta es una de las primeras pruebas realizadas entre los dos computadores. En esta se probó que el NanoMind pudiera solicitar telemetría de housekeeping de forma periódica al CubeComputer, recolectarla y almacenarla en tablas de parámetros que se podían visualizar a través de la interfaz GOSH. Estas pruebas se han realizado con datos de telemetría provisionales.

- **Cambio de modo**: En esta prueba se verificó los cambios de modo entre NanoMind y CubeComputer. Haciendo uso de comandos GOSH de NanoMind, este cambiaba de modo al recibir el comando y mandaba un mensaje al CubeComputer para que este también cambiara al mismo modo.

- **Recolección de datos de experimentos**: Una vez finalizadas las pruebas de cambio de modo, se procedió a desarrollar y validar la recolección de datos de experimentos. Al no disponer de los computadores que gestionan cada experimento y no poder generar datos de estos, al entrar en un modo experimento el CubeComputer generaba datos provisionales que enviaba periódicamente al NanoMind cuando este los solicitaba. Una vez obtenidos y guardados los datos, con un comando GOSH se visualizaba el almacén de datos de experimentos, verificando así su funcionamiento.

- **Gestión de eventos**: Por último, se probó la gestión y recolección de eventos. Para esta prueba se implementó el envío de eventos de CubeComputer a NanoMind cuando este empezaba o finalizaba un experimento. También se implementaron eventos del propio NanoMind como fin de inicialización o fin de commissioning, que refinó la función de cambio de modos ya que de esta forma se generaban cambios de modo automáticos al recibir estos eventos. Para verificar que los eventos se estaban guardando se podía acceder al almacén de eventos usando un comando GOSH.

Inicialmente, debido a que las solicitudes de housekeeping y de cambio de modo funcionan de manera similar, estas se enviaban usando el mismo servidor y con un mismo puerto, pero esto generó problemas ya que ambas operaciones no podían funcionar en paralelo de esta forma. Tras decidir usar puertos CSP distintos para cada una de las operaciones, el resto de las funciones se lograron implementar sin mayores inconvenientes.

5.2.2. Pruebas de envío de telemetría

Estas pruebas se centran en el envío de datos de experimentos, eventos y housekeeping a tierra, no obstante, al ser desarrolladas en el modelo de ingeniería no se disponía de un computador que cumpliera la función de estación de tierra, por lo que esta parte del software de misión se validó completamente en el NanoMind del modelo de ingeniería.

Con el propósito de realizar estas pruebas, se diseñó, en el propio NanoMind, una tarea provisional que actuaba como la estación de tierra. Esta tarea era la encargada de enviar los mensajes de la estación de tierra mostrados en la Figura 2.8, recolectar los datos recibidos y mostrarlos por la terminal GOSH para poder verificar si se han enviado correctamente.

Otro de los principales obstáculos al realizar esta prueba en el modelo de ingeniería del NanoMind es la falta de acceso a la memoria persistente del computador y, por lo tanto, al sistema de ficheros del SDK. Como se menciona en la Subsección 3.3.7, para que las tareas de la librería *libhk* recojan y envíen los datos se necesita guardar en el sistema de ficheros un archivo json que especifique cuales son dichos datos, por lo que, para poder hacer uso de esta librería en el modelo de ingeniería, se requiere editar algunas funciones de *libhk* para que cargue la especificación deseada sin acceder al sistema de ficheros.

```
Sending Hello message to NM (coverage:30 seconds)
Waiting to receive hello back from NM
Received Hello message from NM
Sending telecommands....
Sending TC-Stop message to NM
GS Received event: SeqN:6   Type:1   Mission_time: 98   OP_mode: 5   Data: 17
GS Received event: SeqN:7   Type:1   Mission_time: 96   OP_mode: 5   Data: 19
GS Received event: SeqN:8   Type:1   Mission_time: 95   OP_mode: 5   Data: 0
GS Received event: SeqN:9   Type:1   Mission_time: 44   OP_mode: 8   Data: 16
GS Received event: SeqN:10   Type:1   Mission_time: 44   OP_mode: 8   Data: 19
GS Received event: SeqN:11   Type:1   Mission_time: 44   OP_mode: 8   Data: 0
GS Received experiment: SeqN:12   Type:3   Mission_time: 95   ExperimentType:1
  Radiometer_temperature: 0   Radiometer_duty_cycle: 0
GS Received experiment: SeqN:13   Type:3   Mission_time: 85   ExperimentType:1
  Radiometer_temperature: 0   Radiometer_duty_cycle: 0
GS Received experiment: SeqN:14   Type:3   Mission_time: 75   ExperimentType:1
  Radiometer_temperature: 0   Radiometer_duty_cycle: 0
GS Received experiment: SeqN:15   Type:3   Mission_time: 65   ExperimentType:1
 Radiometer_temperature: 0   Radiometer_duty_cycle: 0
GS Received experiment: SeqN:16   Type:3   Mission_time: 54   ExperimentType:1
 Radiometer_temperature: 0   Radiometer_duty_cycle: 0
GS Received experiment: SeqN:17   Type:3   Mission_time: 44   ExperimentType:1
 Radiometer_temperature: 0   Radiometer_duty_cycle: 0
Receiving HK DATA
seq_numb: 18
tm_type: 2
```

Figura 5.7: Interfaz GOSH mostrando datos de telemetría.

En la Figura 5.7 se muestra los resultados de esta prueba desde la interfaz GOSH del computador, donde se pueden visualizar los mensajes que serían enviados desde la estación de tierra y los datos de telemetría recibidos.

5.2.3. Pruebas del interfaz del EPS

Para validar la interfaz del EPS se hizo uso del modelo de ingeniería de NanoMind y EPS, conectados a través del bus CAN. Siguiendo la documentación aportada por DHV, se crearon una serie de comandos GOSH que, al ser llamados desde el NanoMind, enviaban diferentes solicitudes al EPS a través de CSP. Cada solicitud se enviaba hacia un puerto del EPS distinto, que indica a este que comando específico debe ejecutar. Estos comandos pueden notificar al EPS que se realice un envío de telemetría al NanoMind, que se realicen cambios en su configuración y otros comandos útiles para la misión.

Las pruebas realizadas con el EPS no fueron exitosas en un inicio debido a que comandos como *ping* o *reset* no seguían el formato propuesto por la propia librería *libCSP*, lo que causó algunos comportamientos no intencionados en el sistema al intentar usarlos. Con ayuda de DHV se logró solventar estos errores, aplicando la estructura de paquete correcta en las solicitudes de estos comandos.

5.2.4. Registro y documentación de pruebas en modelos de vuelo

Debido a la importancia de los componentes a manejar al realizar pruebas en modelos de vuelo, se deben de realizar una serie de documentos antes de cada prueba especificando los siguientes puntos:

- Lista de equipamiento a usar, tanto los computadores en los que se va a centrar cada prueba como otros elementos, por ejemplo, cables.
- Pasos que realizar para preparar la prueba antes de comenzar. Estos pasos suelen describir como conectar los computadores y como leer los datos que estos muestran.
- Especificar, paso por paso, como realizar cada prueba y que se espera que ocurra en cada paso.

Mientras se esté realizando la prueba también se debe elaborar documentación, anotando que está ocurriendo en cada paso, si la prueba está aportando los resultados esperados o si se ha saltado o añadido algún paso. En conclusión, tras la finalización de la prueba se debe de tener un documento que indique todos los pasos planteados antes de la realización de la prueba y cuales han sido las acciones realizadas y resultados obtenidos a lo largo del proceso.

Una vez definido que contenido debe de contener esta documentación, se va a describir las pruebas realizadas en los modelos de vuelo.

5.2.5. Pruebas del NanoCom AX100

Las primeras pruebas realizadas con la radio NanoCom AX100 consistían en encenderla y, con diferentes configuraciones y niveles de potencia que se podían cambiar a través de la interfaz GOSH, hacerla transmitir señales simples, como patrones constantes de ceros y unos, para analizar la salida de la radio. Además, en esta prueba también se probaron las capacidades de recepción de la radio,

haciendo uso de un generador de señales y, usando la interfaz GOSH del AX100, para verificar los datos recibidos.

Para alimentar al AX100 se hará uso del conector J3, tal como se indica en su *datasheet* [7]. A través de este conector USART se puede alimentar al computador y acceder a su interfaz GOSH. A pesar de que este computador se alimenta de la misma forma que el NanoMind, conectándose a un PC a través de un cable FTDI/USB, al realizar las pruebas usando este método se comprobó que el AX100 no recibía suficiente potencia para garantizar su adecuado funcionamiento.

Con el objetivo de lograr que llegue una mayor cantidad de corriente al AX100 se procede a usar una fuente de alimentación externa, por lo que se deben extraer los pines 1 y 2 del conector J3 y seguir el esquema mostrado en la Tabla 5.1 para realizar la conexión. Esta fue la configuración final utilizada para realizar todas las pruebas planteadas.

Cuadro 5.1: Pines de conector USART J3 de NanoCom AX100.

Pin	**Descripción**	**Conectado a**
1	Tierra	Fuente de alimentación externa (Tierra)
2	Voltaje 3,3V	Fuente de alimentación externa (3,3V)
3	USART de Recepción	PC vía cable FTDI
4	USART de Transmisión	PC vía cable FTDI

Tras lograr alimentar a la radio, se realizaron todas las pruebas planteadas y con los resultados obtenidos se pudo establecer parte de la configuración que se usará en vuelo, principalmente la potencia de transmisión, que será la máxima que el AX100 permite, la frecuencia y la tasa de baudios de transmisión y recepción.

5.2.6. Pruebas del NanoMind A3200

Estas fueron las primeras pruebas realizadas con el NanoMind A3200, por lo que se siguieron unos pasos básicos para cargar software por primera vez en el computador y usar la terminal GOSH para comprobar que se estaba comportando adecuadamente. En concreto se siguió el siguiente procedimiento:

1. Primero se conecta el computador siguiendo el esquema de la Figura 5.5.
2. A continuación, se compila y carga nuestro programa en el NanoMind haciendo uso del AVR Dragon Debugger. Este programa es idéntico a la versión original del SDK de *GomSpace* salvo que se añadió una línea de código que imprimía una frase en la terminal GOSH. De esta forma se podía confirmar que se había cargado software modificado sin alterar la funcionalidad del programa original.
3. Tras cargar el programa se visualizan todas las tablas de parámetros, comprobando que los valores de los datos contenidos en estas sean correctos.

4. Usando los parámetros del NanoMind, se confirma que el sistema de ficheros y el reloj del computador están funcionando correctamente.

5. Seguidamente, se envía al propio NanoMind mensajes ping por la interfaz loopback, comprobando así que es capaz de mandar y recibir mensajes CSP.

6. Por último, usando la librería *libfp*, se crea un temporizador que envíe un mensaje ping pasado un tiempo determinado.

Estas pruebas se realizaron correctamente sin incidentes, demostrando así que el componente y su software se estaba comportando adecuadamente.

5.2.7. Pruebas de la estación de tierra

En la estación de tierra, compuesta por GS100 y MS100, es importante verificar que las dos radios AX100 encontradas en el GS100 funcionan adecuadamente, es por ello que se llevaron a cabo las mismas pruebas realizadas sobre el AX100 del satélite en las dos radios contenidas en el GS100.

Junto a las pruebas de transmisión y recepción, se realizaron también pruebas de conexión CSP entre los componentes de la estación de tierra. Considerando que cada una de las dos radios del GS100 y el propio MS100, más específicamente su aplicación *csp-term* que ejecuta el SDK, tienen su propio nodo CSP, se enviaron mensajes ping entre todos ellos, verificando de esta forma que se pueden comunicar a través de este protocolo.

Las pruebas de conexión CSP no aportaron los resultados que se esperaban en un inicio, debido a que no se podían realizar pings desde el MS100 a las dos radios GS100. Esto se solucionó alterando las tablas de enrutado de las radios, ya que sus tablas por defecto no contenían la entrada correspondiente a la dirección del MS100.

5.2.8. Pruebas del NanoDock DMC-3

El objetivo de esta prueba era lograr que la radio AX100 y NanoMind se pudieran comunicar a través de CSP/CAN, conectando ambos al NanoDock DMC-3. Para conseguir este propósito se conectó el NanoMind siguiendo el esquema mostrado en la Figura 5.6. Por otro lado, se conecta el AX100 de forma similar, usando su conector J1 encontrado en la parte inferior del computador para conectarlo al conector X2 del NanoDock.

No obstante, una vez que los computadores se encontraran en el NanoDock no se tendría acceso a los conectores laterales, necesarios para visualizar la interfaz GOSH, sino que se accedería a esta interfaz mediante los puertos P1 y P2 del NanoDock. Por este motivo, como en la prueba era importante seguir teniendo acceso a esta interfaz se han tenido que configurar los computadores para cambiar el conector por el que muestran dicha interfaz antes de conectarlos con el NanoDock.

Para lograr esto se tuvo que cambiar la configuración guardada en el sistema de parámetros de cada computador. En el NanoMind es simple, ya que se puede editar en el SDK antes de compilarlo y cargarlo, pero en el AX100, al no poder cargar software en este computador, se tuvo que modificar el parámetro haciendo uso de la interfaz GOSH y guardarlo en memoria persistente, ya que, aunque se reinicie, debe seguir mostrando la terminal GOSH por el puerto del NanoDock. Una vez modificados estos parámetros se conectan los computadores al NanoDock.

Usando las interfaces GOSH, se modificaron las tablas de enrutado de ambos computadores y otros parámetros como la velocidad del bus CAN para que se pudieran comunicar entre sí. Tras realizar este cambio, se enviaron mensajes ping entre ellos, los cuales funcionaron correctamente. Además, se prueba a configurar remotamente el AX100 desde el NanoMind, modificando sus parámetros, ya que esta es una función esencial en vuelo y también funciona como esperado.

La mayor dificultad encontrada a la hora de realizar la prueba fue configurar la radio AX100 para que esta pudiera mostrar su interfaz GOSH a través del NanoDock. Al haberse guardado su configuración en memoria persistente pero no como configuración por defecto del sistema, al reiniciarse el computador y no poder cargar la configuración que se estaba usando, se intentó cargar la configuración por defecto, que en aquel momento no estaba modificada. Debido a esto, se dejó de mostrar la interfaz GOSH por el puerto del NanoDock obligando a desconectar la radio del NanoDock y cargar la configuración por defecto.

Tras cargar la configuración y conectar la radio al NanoDock otra vez se terminaron de realizar las pruebas planteadas confirmando así que los dos computadores se pueden comunicar entre sí.

5.2.9. Pruebas de conexión entre el NanoMind y la estación de tierra

Esta prueba es la más compleja realizada en el modelo de vuelo, ya que involucra todos los componentes probados hasta el momento: NanoMind y AX100 conectados a través del NanoDock y la estación de tierra completa, MS100 y GS100. El objetivo de la prueba es enviar mensajes CSP SFP del NanoMind al MS100 y que este los lea e imprima en su terminal.

La dificultad de esta prueba se encuentra en la configuración de los componentes, ya que no solo se deben de editar los parámetros de las tres radios para que estas se puedan comunicar entre sí, sino que también se deben de editar las tablas de enrutado de todos los componentes, especificando a través de que nodos se debe enviar cada mensaje, por ejemplo, el NanoMind debe de enviar sus mensajes a tierra a través de la radio del satélite.

Tras enviar paquetes CSP SFP de diversos tamaños se llegó a la conclusión que, con la configuración de la radio que se iba a usar en vuelo, el tamaño máximo de datos que se puede encontrar en cada paquete fragmentado es de 203 bytes, por lo que los paquetes de telemetría que se envíen a tierra tendrán este tamaño máximo.

Capítulo 6

Análisis de impacto

A pesar de que el proyecto UPMSat-3 no se encuentra finalizado, este proyecto ya está causando impacto en diferentes contextos, principalmente entre los participantes de esta iniciativa. Una gran parte de los integrantes del proyecto son alumnos, por lo que este les aporta conocimientos únicos y experiencia laboral en un campo poco común. Asimismo, una vez lanzado el microsatélite, se espera poder usarse como plataforma educacional y aportar diferentes iniciativas para que alumnos de diferentes escuelas de la UPM puedan interactuar con el UPMSat-3 en órbita.

Como resultado, se puede considerar que el principal impacto que tendrá este proyecto es un impacto positivo sobre la educación de los alumnos. Si tenemos en cuenta los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030 [26] es razonable afirmar que este proyecto aporta de manera positiva al **Objetivo 4: Educación de calidad**. Este objetivo tiene como metas aportar las competencias necesarias, técnicas y profesionales, para acceder al empleo. Asimismo, también afirma que la financiación de la educación debe convertirse en una prioridad de inversión nacional, por lo tanto, al estar financiado este proyecto en gran parte con dinero proveniente de la Comunidad de Madrid, se puede observar las consecuencias positivas que tienen alcanzar las metas de este objetivo.

En este sentido, también se puede vincular el proyecto con el **Objetivo 8: Trabajo decente y crecimiento económico**, que promueve la creación de puestos de trabajo innovativos y ofrecer puestos de trabajo a jóvenes.

A su vez, este proyecto también beneficia a las empresas como HYDRA Space o DHV Technology, cuyos productos están siendo probados en el propio satélite, ya sea en forma de experimentos como la radio de HYDRA, o como una parte íntegra del sistema como el EPS de DHV. Estos componentes y otros como el ADCS del IDR y los radiómetros de UC3M-GREMA contribuyen al cumplimiento del **Objetivo 9 del ODS: Industria, innovación e infraestructura**, fomentando el progreso tecnológico e innovación de la industria.

No solo los alumnos y empresas se verán impactados positivamente con el desarrollo del microsatélite, sino que otros grupos externos como *Open Source Global Network of Satellite Ground-Stations* (SatNOGS) [27] y la comunidad de radioafi-

cionados podrán leer datos de beacons del satélite cuando este se encuentre en órbita, beneficiando así al equipo del UPMSat-3, que obtienen datos recogidos de todo el mundo. Por este motivo, el formato de la telemetría será público y accesible a quién esté interesado por saber más acerca del satélite.

Por último, para lograr que el microsatélite pueda operar durante un largo periodo de tiempo se ha diseñado este usando una serie de estándares profesionales y herramientas comunes en este sector ya probadas en otros satélites, por ejemplo, en el caso del software, el protocolo CAN, el sistema operativo FreeRTOS y el protocolo de comunicaciones LibCSP. A pesar de esto, el UPMSat-3 no podrá estar operativo durante muchos años, siendo previsto que lo estará un total de 3 años. Una vez transcurrido este periodo de tiempo, el satélite pasará a un estado de fin de vida (EOL), en el que se desactivarán las fuentes de energía para evitar explosiones o roturas. Este modo deja el satélite fuera de servicio, lo que minimiza el riesgo de basura espacial y posibles colisiones. Este último punto tiene un impacto positivo en relación con el **ODS 12: Producción y Consumo Responsables**, cuya meta principal es reducir la generación de residuos y prevenir su uso.

Capítulo 7

Conclusiones y trabajo futuro

7.1. Conclusiones

Para cumplir el objetivo general de este trabajo de fin de grado (diseñar, desarrollar, verificar y validar el software de misión de UPMSat-3), a lo largo del proyecto se ha trabajado en la consecución de los objetivos específicos definidos en la Sección 1.3. En esta sección final, se resumen los objetivos alcanzados y la forma en que se han logrado.

- **OE1. Analizar y diseñar los requisitos software del satélite**: Durante el desarrollo del proyecto se ha estado realizando un exhaustivo análisis de la misión y los requisitos que se deben cumplir a nivel software. Los resultados de este análisis han contribuido a definir las funciones de la misión, mostradas en el Capítulo 2. Aunque las funciones definidas pueden sufrir ligeros cambios a medida que sigue avanzando el proyecto, ya se encuentran totalmente definidas.

- **OE2. Analizar el framework del software de la misión**: Para realizar el análisis del entorno de desarrollo del software de la misión, detallado en el Capítulo 3, se ha procedido a estudiar documentación relacionada con todos sus componentes, *FreeRTOS*, el protocolo CSP y *libCSP* y el software de *GomSpace*, teniendo de esta manera una base sólida no solo para realizar el desarrollo de este proyecto, sino para otros posibles proyectos de este campo, ya que el uso de estos es común en otros microsatélites.

- **OE3. Diseñar el software de misión del satélite**: Este objetivo también se ha alcanzado de forma satisfactoria. El diseño software y la definición de las tareas que lo componen, mostrado en el Capítulo 4, permite desarrollar e implementar software en la placa NanoMind que realiza las funciones definidas previamente durante el objetivo OE1. Al igual que las propias funciones, el software se puede ver ligeramente alterado durante el resto del proyecto, pero se espera que la estructura principal, mostrada y estudiada en esta tesis no se modifique.

- **OE4. Implementar y validar el software en los computadores de ingeniería y de vuelo**: Para demostrar que las funciones definidas se realizaban correctamente, se ha probado en el modelo de ingeniería del satélite el software diseñado en el objetivo OE3. Asimismo, se han realizado pruebas en diferentes componentes de vuelo, verificando que estos funcionan correctamente y algunas funcionalidades básicas entre ellos, como el envío de datos desde el NanoMind a la estación de tierra mediante el AX100. Estas pruebas, detalladas en el Capítulo 5, justifican que se ha validado el software por completo en el modelo de ingeniería y parcialmente en el de vuelo.

Al cumplir todos los objetivos específicos del trabajo de fin de grado, se considera que el objetivo principal de esta tesis, "desarrollar el software de misión del microsatélite UPMSat-3"se ha alcanzado también, finalizando de esta forma el desarrollo software del computador de misión del satélite.

7.2. Trabajo futuro

Al no estar finalizado el proyecto UPMSat-3, todavía hay trabajo por realizar. Comenzando por las funcionalidades del satélite, aún se deben de añadir datos como más **eventos** que definan con exactitud errores o comportamientos anómalos ocurridos en vuelo y más datos de **housekeeping**, que muestren más aspectos del estado del satélite.

También se debe **validar el sistema completo** en el modelo de vuelo, probando en este todo el software desarrollado en el NanoMind, ya que no ha sido posible probar el software probado en el modelo de ingeniería en el modelo de vuelo. Además, no se han podido realizar pruebas entre el NanoMind y el CubeComputer de vuelo, y validar el sistema completo es una tarea fundamental que se debe de realizar antes de lanzar el satélite.

Con respecto a los **experimentos** que se van a realizar en órbita, estos también se encuentran en desarrollo. Por esta razón, a pesar de que la funcionalidad de recoger datos de experimentos ya se encuentra desarrollada en el computador de misión, se debe definir en este cuáles serán los datos exactos a recolectar, almacenar y enviar a tierra.

Por otra parte, se debe de desarrollar y validar el **software de la estación terrena**. Esta parte del desarrollo software de la misión, aunque parcialmente estudiada al realizar pruebas en el NanoMind como la definida en la Subsección 5.2.2, aún se debe de analizar y diseñar en detalle.

Estas sugerencias de posibles trabajos futuros están relacionadas con el UPMSat-3. No obstante, fuera del proyecto también pueden abordarse tareas vinculadas con esta tesis, como **generalizar la arquitectura software** desarrollada para que pueda aplicarse en misiones similares.

Bibliografía

[1] IDR/UPM, “Instituto Universitario de Microgravedad Ignacio Da Riva,” [En línea]. Disponible en: https://www.idr.upm.es/es/.

[2] IDR/UPM, “Instituto Universitario de Microgravedad Ignacio Da Riva - The UPMSat-3 project,” Nov 2024, [En línea]. Disponible en: https://www.idr.upm.es/index.php/es/upmsat-3?view=article&id=296:1el-proyecto-upmsat-3&catid=40.

[3] IDR/UPM, “Instituto Universitario de Microgravedad Ignacio Da Riva - Satélites UPM-SAT,” [En línea]. Disponible en: https://sat.idr.upm.es/.

[4] “CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers,” NASA CubeSat Launch Initiative, 2017, [En línea]. Disponible en: https://www.nasa.gov/wp-content/uploads/2017/03/nasa_csli_cubesat_101_508.pdf?emrc=05d3e2.

[5] GomSpace, “NanoMind A3200,” datasheet, 17 Feb 2021, [En línea]. Disponible en: https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanomind-a3200_1006901-117.pdf.

[6] GomSpace, “Software Development Kits,” [En línea]. Disponible en: https://gomspace.com/UserFiles/Subsystems/flyer/gomspace_software_A3200_sdk_flyer.pdf.

[7] GomSpace, “NanoCom AX100,” datasheet, 26 Nov 2019, [En línea]. Disponible en: https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanocom-ax100.pdf.

[8] GomSpace, “NanoDock DMC-3,” datasheet, 25 Mar 2021, [En línea]. Disponible en: https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanodock-dmc-3-112.pdf.

[9] GomSpace, “NanoCom GS100,” datasheet, 23 Abr 2020, [En línea]. Disponible en: https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanocom-gs100-22.pdf.

[10] GomSpace, “NanoCom MS100,” datasheet, 16 Feb 2021, [En línea]. Disponible en: https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanocom-ms100-13.pdf.

[11] J. Garrido, J. Zamorano, A. Alonso, and J. A. de la Puente, "Timing Analysis of the UPMSat-2 Communications Subsystem." *IFAC-PapersOnLine*, vol. 51, no. 10, pp. 217–222, 2018, doi: 10.1016/j.ifacol.2018.06.265.

[12] FreeRTOS, "What is FreeRTOS?" [En línea]. Disponible en: https://www.freertos.org/Why-FreeRTOS/What-is-FreeRTOS.

[13] FreeRTOS, "RTOS Fundamentals," 2025, [En línea]. Disponible en: https://www.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/01-RTOS-fundamentals.

[14] *The FreeRTOS Reference Manual, version 10.0.0*, Amazon Web Services, 2017, [En línea]. Disponible en: https://www.freertos.org/media/2018/FreeRTOS_Reference_Manual_V10.0.0.pdf.

[15] CSP, "The Cubesat Space Protocol," 2025, [En línea]. Disponible en: https://libcsp.github.io/libcsp/.

[16] European Commission for Space Standardization, "ECCS-E-ST-40C Rev.1 Space Engineering — Software," 30 Abr 2025, [En línea]. Disponible en: https://ecss.nl/wp-content/uploads/2025/05/ECSS-E-ST-40C-Rev.1(30April2025).pdf.

[17] Atmel, "AVR32919: UC3C Evaluation Kit User Guide," May 2015, [En línea]. Disponible en: https://ww1.microchip.com/downloads/aemDocuments/documents/MPU32/ProductDocuments/UserGuides/Atmel-32151-UC3C-Evaluation-Kit_UserGuide.pdf.

[18] "TeraTerm Project," 05 May 2025, [En línea]. Disponible en: https://teratermproject.github.io/index-en.html.

[19] "PuTTY: a free SSH and Telnet client," 08 Feb 2025, [En línea]. Disponible en: https://www.chiark.greenend.org.uk/~sgtatham/putty/.

[20] Atmel, "Atmel-ICE User Guide," Oct 2016, [En línea]. Disponible en: https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/UserGuides/Atmel-ICE_UserGuide.pdf.

[21] Microchip Technology Inc., "Microchip Studio for AVR® and SAM Devices," [En línea]. Disponible en: https://www.microchip.com/en-us/tools-resources/develop/microchip-studio.

[22] HMS Networks., "Ixxat simplyCAN USB-to-CAN Adapter User Manual," [En línea]. Disponible en: https://hmsnetworks.blob.core.windows.net/nlw/docs/default-source/products/ixxat/pc-interface-cards/manuals-and-guides---manuals/simplycan-manual-english.pdf?sfvrsn=85e648d7_13.

[23] Silicon Labs, "EFM32GG Data Sheet," [En línea]. Disponible en: https://www.silabs.com/documents/public/data-sheets/efm32gg-datasheet.pdf.

[24] Microchip Technology Inc., "MCP2515 Stand-Alone CAN Controller with SPI Interface Data Sheet," 15 Ag 2018, [En línea]. Dis-

ponible en: https://ww1.microchip.com/downloads/en/DeviceDoc/MCP2515-Stand-Alone-CAN-Controller-with-SPI-20001801J.pdf.

[25] Atmel, "AVR Dragon User Guide," Abr 2016, [En línea]. Disponible en: https://ww1.microchip.com/downloads/en/devicedoc/atmel-42723-avr-dragon_userguide.pdf.

[26] Naciones Unidas, "Objetivos de Desarrollo Sostenible, Agenda 2030," [En línea]. Disponible en: https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/.

[27] SatNOGS, "Open Source global network of satellite ground-stations SatNOGS," [En línea]. Disponible en: https://satnogs.org/.

Anexos

Apéndice A

Anexo 1 - Ejemplos CSP

A.1. Envío de paquete CSP

En el siguiente fragmento de código se muestra como mandar un mensaje CSP. Es importante tener en cuenta que, en caso de error, se debe cerrar la conexión y liberar el buffer. En el final del código no se libera el buffer, ya que las funciones de las interfaces son las que se encargan de liberarlo cuando transmiten el mensaje.

```
// Preparamos datos
csp_packet_t * packet = csp_buffer_get(100);
if (packet == NULL)
        return 1;

// Nos conectamos al nodo 2 a través del puerto 10
csp_conn_t * conn = csp_connect(CSP_PRIO_NORM, 2, 10, 1000, CSP_O_NONE);
if (conn == NULL) {
        csp_buffer_free(packet);
        return 1;
}

// Escribimos los datos
packet->data[0] = 0x01;
packet->length = sizeof(uint8_t);

// Enviamos el paquete
if (!csp_send(conn, packet, 1000)){
    csp_buffer_free(packet);
    csp_close(conn);
        return 1;
}

csp_close(conn);
```

```
return 0;
```

A.2. Recibo de paquete CSP

Para recibir paquetes primero se debe de crear un socket al que se puedan enviar paquetes. En este ejemplo creamos un socket en el puerto 10 y este socket podrá guardar hasta 5 paquetes a la vez para ser leídos.

```
csp_socket_t* socket;
socket = csp_socket(CSP_SO_NONE);
csp_bind(socket, 10);
csp_listen(socket, 5);
```

Una vez creado el socket se puede aceptar conexiones a través de este y leer los datos de los paquetes CSP recibidos.

```
// Aceptamos conexión
csp_conn_t * conn;
conn = csp_accept(socket, CSP_MAX_TIMEOUT);
if (conn == NULL) {
    return 1;
}

// Recogemos el paquete recibido
csp_packet_t *packet = csp_read(conn, 0);
if(packet == NULL){
    csp_close(conn);
    return 1;
}

// Leemos los datos (Suponemos que el dato es un solo uint8)
uint8_t num = packet->data[0];

// Liberamos paquete y cerramos conexión
csp_buffer_free(packet);
csp_close(conn);

return 0;
```

A.3. Envío y recibo de mensaje SFP

La forma de mandar mensajes SFP es muy similar a mandar paquetes CSP no fragmentados. En este ejemplo se enviará un mensaje de tamaño 500 bytes subdividido en paquetes de 180 bytes de datos.

```
int size = 500;
uint8_t data[size];
```

```
uint32_t encoded_size = sizeof(data);

// Abrimos conexión
csp_conn_t *conn = csp_connect(CSP_PRIO_NORM, 2, 10, 10000, CSP_O_NONE);
if (conn == NULL)
{
   return 1;
}

// Inicializamos los datos a enviar
for (int i = 0; i < size; i++)
{
    data[i] = i;
}

// Enviamos por SFP
int result = csp_sfp_send(conn, data, encoded_size, 180, 1000);
if (result != CSP_ERR_NONE)
{
    csp_close(conn);
    return 1;
}

// Cerramos conexión
csp_close(conn);

return 0;
```

Para recibir paquetes se debe crear primero un socket, como se ha descrito en el apartado anterior y seguir los siguientes pasos:

```
// Aceptamos conexión
csp_conn_t *conn = csp_accept(socket, CSP_MAX_DELAY);
if (conn == NULL)
{
    return 1;
}

// Reservamos espacio para el paquete
uint8_t* packet_data = (uint8_t*)malloc(size * sizeof(uint8_t));
if (packet_data == NULL)
{
    csp_close(conn);
    return 1;
}

int packet_size = 0;
```

```
// Recibimos el paquete en packet_data y en packet_size su tamano
int ret = csp_sfp_recv(conn, (void**)&packet_data, &packet_size, 3000);
if (ret != CSP_ERR_NONE)
{
    free(packet_data);
    csp_close(conn);
    return 1;
}

free(packet_data);
csp_close(conn);

return 0;
```

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jun 03 05:46:22 CEST 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)