

Grado Universitario en Ingeniería Informática  
2020-2021

*Trabajo Fin de Grado*

# “Diseño e implementación de un banco de pruebas para el desarrollo de nanosatélites”

---

Marcelo Miquel Juan

Tutor

Javier Fernández Muñoz

Leganés, 2021



Esta obra se encuentra sujeta a la licencia Creative Commons **Reconocimiento**  
– **No Comercial** – **Sin Obra Derivada**



# Resumen

La exploración espacial es atractiva tanto para empresas como para organizaciones gubernamentales. La última tendencia en el sector es la de enviar nanosatélites al espacio: satélites pequeños, baratos y accesibles. Este movimiento ha permitido reducir la barrera de entrada al espacio: ya no solo grandes corporaciones tienen acceso a él, ahora también pequeñas empresas y universidades se pueden permitir tener su propio satélite en órbita.

Este proyecto pertenece a la cátedra UC3M-SENER: un proyecto conjunto entre ambas entidades que reúne estudiantes en su último año de carrera de múltiples disciplinas (aeroespacial, telecomunicaciones, informática) con el fin de desarrollar y lanzar un nanosatélite al espacio.

En un proyecto de esta envergadura, el proceso de *testing* es una de las actividades más importantes para detectar errores a tiempo y garantizar que se cumplen los requisitos predefinidos. Este proyecto de fin de grado pretende implementar un banco de pruebas para probar el software de vuelo que usará el nanosatélite de la cátedra: el cFS. La arquitectura de software empleada es el NOS<sup>3</sup>, proporcionado por la NASA, que permite probar el software de vuelo junto a simuladores de componentes hardware para crear un entorno de ejecución realista.

**Palabras clave:** Nanosatélite, banco de pruebas, simulador, software de vuelo.



# Abstract

Space exploration attracts both corporations and government entities. The latest trend in the industry is to build and send nanosatellites into space: small, cheap and accesible satellites. This trend has reduced the barrier to entry into space: no longer only large corporations have access to it, now also small companies and universites can afford to have their own satellite in orbit.

This project belongs to the UC3M-SENER chair: a joint project between both entities that brings together students in their last year of career from multiple disciplines (aerospace, telecommunications, computing) in order to develop and launch a nanosatellite into space.

In this kind of projects, testing is one of the most important activities to detect errors early and ensure that the predefined requirements are met. This end-of-degree project aims to implement a test bench to test the flight software that will be used by the nanosatellite of the chair: the cFS. The software architecture used is the NOS<sup>3</sup>, provided by NASA, which allows the flight software to be tested together with simulated hardware component to create a realistic runtime environment.

**Keywords:** Nanosatellite, test bench, simulator, flight software.



# Índice general

<b>1. Introducción .....</b>	<b>1</b>
1.1 Motivación .....	1
1.2 Objetivos .....	2
1.3 Estructura del documento.....	3
<b>2. Estado del arte .....</b>	<b>5</b>
2.1 Historia.....	5
2.2 Sistemas Empotrados .....	6
2.3 Sistemas de tiempo real .....	11
2.4 cFS .....	15
2.5 Bancos de prueba .....	21
2.6 NOS3.....	25
<b>3. Análisis y diseño.....</b>	<b>30</b>
3.1 Requisitos de usuario .....	30
3.2 Casos de uso.....	36
3.3 Requisitos de software .....	41
3.4 Diseño de la solución .....	48
3.5 Matrices de trazabilidad .....	53
<b>4. Implementación. ....</b>	<b>55</b>
4.1 Simuladores.....	55
4.2 Aplicación “Keep in Orbit”.....	60
<b>5. Pruebas y evaluación.....</b>	<b>63</b>
5.1 Pruebas .....	63
5.2 Matriz de trazabilidad .....	67
5.3 Evaluación.....	67
<b>6. Plan de proyecto.....</b>	<b>69</b>
6.1.Tareas del proyecto .....	69

6.2 Planificación .....	70
6.3 Presupuesto del proyecto .....	72
<b>7. Marco regulador y entorno socio-económico .....</b>	<b>75</b>
7.1 Marco regulador.....	75
7.2 Entorno socio-económico. ....	78
<b>8. Conclusiones y trabajos futuros .....</b>	<b>80</b>
8.1 Objetivos .....	80
8.2 Conclusiones del proyecto .....	81
8.3 Trabajos futuros .....	82
<b>A. Summary .....</b>	<b>83</b>
A.1 Introduction.....	83
A.2 Objectives.....	84
A.3 NOS3.....	84
A.4 System architecture .....	86
A.5 Implementation .....	87
A.6 Project plan .....	90
A.7 Conclusions .....	92
<b>B. Glosario.....</b>	<b>95</b>
<b>C. Manual de usuario .....</b>	<b>96</b>
<b>D. Bibliografía.....</b>	<b>101</b>





# Índice de figuras

Figura 2.1: Sputnik-1 .....	6
Figura 2.2: Funcionamiento de un sistema empotrado .....	7
Figura 2.3: Arquitectura de la memoria caché .....	9
Figura 2.4: Ejemplo de arquitectura de un sistema empotrado .....	10
Figura 2.5: Gráfico de las revoluciones industriales .....	11
Figura 2.6: Multitask .....	13
Figura 2.7: Arquitectura del cFS .....	17
Figura 2.8: Arquitectura del cFS II .....	18
Figura 2.9: Servicios ejecutivos del cFE .....	19
Figura 2.10: ETAS ES4440 .....	24
Figura 2.11: ETAS ES4441 .....	25
Figura 2.12: Cubesat STF-1 .....	26
Figura 2.13: Arquitectura NOS <sup>3</sup> .....	28
Figura 2.14 Consola de NOS Engine .....	29
Figura 3.1: Plantilla de requisito de usuario .....	31
Figura 3.2: Diagrama UML .....	37
Figura 3.3: Plantilla de caso de uso .....	38
Figura 3.4: Plantilla de requisito de sistema .....	42
Figura 3.5: Plantilla de componente .....	48
Figura 3.6: Diagrama de componentes .....	49
Figura 4.1: Configuración XML del sensor .....	56
Figura 4.2: Configuración XML del motor .....	58
Figura 4.3: Diagrama de comunicación .....	61
Figura 4.4: Algoritmo de la aplicación .....	62
Figura 5.1: Plantilla de casos de pruebas .....	64
Figura 6.1: Diagrama Gantt .....	71
Figure A.1: NOS <sup>3</sup> architecture .....	86
Figure A.2: Component diagram of the solution .....	87
Figure A.3: Communication diagram .....	90
Figura C.1: Entorno de ejecución de NOS <sup>3</sup> .....	96
Figura C.2: Uso de git para clonar el repositorio NOS3 .....	97
Figura C.3: Opciones de configuración .....	98
Figura C.4: Ejecución de vagrant .....	98
Figura C.5: Escritorio inicial del NOS3 .....	99
Figura C.6: Configuración de carpeta compartida .....	100



# Índice de tablas

Tabla 2.1: Servicios ejecutivos de cFE .....	20
Tabla 2.2: Comparativa de modelos OPAL-RT.....	22
Tabla 2.3: Comparativa de modelos d-SPACE.....	23
Tabla 2.4: Componentes que forman el NOS3 .....	27
Tabla 3.1: Trazabilidad entre requisitos funcionales y requisitos de capacidad.....	53
Tabla 3.2: Trazabilidad entre requisitos de restricción y requisitos no funcionales	54
Tabla 3.3: Trazabilidad entre componentes y requisitos funcionales .....	54
Tabla 4.1: Estructura del mensaje enviado por el sensor .....	57
Tabla 4.2: Descripción de los campos del mensaje enviado por el sensor .....	57
Tabla 4.3: Valores del header y el trailer.....	57
Tabla 4.4: Estructura del mensaje recibido por el motor .....	59
Tabla 4.5: Descripción de los campos del mensaje recibido por el motor.....	59
Tabla 4.6: Codificación del campo "payload" .....	59
Tabla 4.7: Valores del header y del trailer.....	60
Tabla 5.1: Trazabilidad entre casos de prueba y requisitos funcionales.....	67
Tabla 5.2: Evaluación del tiempo de respuesta del sistema.....	68
Tabla 6.1: Distribución de las horas del proyecto.....	72
Tabla 6.2: Costes de personal .....	72
Tabla 6.3: Costes de material.....	73
Tabla 6.4: Costes indirectos .....	74
Tabla 6.5: Coste total .....	74
Tabla 7.1: Atributos de registros.....	77
Table A.1: NOS3 dependency list.....	85
Table A.2: Message structure sent by the sensor .....	88
Table A.3: Description of the fields of the message sent by the sensor .....	88
Table A.4: Structure of the message received by the engine .....	89
Table A.5: Description of the fields of the message received by the engine .....	89
Table A.6: Coding of the payload.....	89
Table A.7: Distribution of project hours .....	91
Table A.8: Personnel costs.....	91
Table A.9: Indirect costs .....	92
Table A.10: Total costs .....	92
Tabla C.1: Dependencias de NOS <sup>3</sup> .....	97



## Capítulo 1.

# Introducción

Este proyecto de fin de grado ha sido posible gracias a la colaboración entre la empresa SENER y la universidad Carlos III de Madrid. Tras un largo y fructífero vínculo de colaboración en tareas de investigación y docentes, en 2018 se creó la Cátedra UC3M-SENER Aeroespacial, un nuevo marco en donde poder continuar y potenciar la relación entre ambas empresas.

Fundada en 1956 por Enrique de Sendagorta Aramburu como oficina técnica naval, SENER es la primera empresa española de ingeniería registrada como tal [1]. Actualmente, SENER agrupa las actividades propias de Aeroespacial y de Ingeniería, además de participaciones industriales en compañías que trabajan en energía. SENER Aeroespacial cuenta con más de 50 años de experiencia y es un proveedor de primer nivel para Espacio, Defensa y Ciencia.

Dicha cátedra reúne estudiantes de múltiples disciplinas de la ingeniería: aeroespacial, informática, de telecomunicaciones, etc. Y tiene como objetivo establecer las bases de la colaboración en tareas de investigación relacionadas con el desarrollo de un nanosatélite.

## 1.1 Motivación

Desde el lanzamiento del primer satélite al espacio, la tendencia ha sido de construir y lanzar satélites cada vez más complejos, grandes y caros. El espacio estaba reservado a grandes corporaciones dependientes del ámbito gubernamental y militar. Es lo que se denomina el *Old Space* [2].

Pero el panorama actual ha cambiado mucho. La nueva ola de colonización está siendo llevada a cabo por sistemas pequeños, baratos y rápidos que favorecen que entren en juego aquellas empresas que necesitan el espacio para alcanzar sus objetivos y ampliar sus servicios [2]. Esta filosofía es a la que llamamos *New Space*.

Los nanosatélites son la última tendencia en el New Space. Pequeños, con una masa de entre 1 y 10kg permiten ofrecer una gran precisión. Además, gracias a los avances en microelectrónica y al estándar CubeSat, su fabricación permite una producción masiva

y de bajo costo. El estándar CubeSat [3] fue creado en 1999 por la Universidad Politécnica de California y la Universidad de Stanford con el fin de ampliar el acceso a la industria espacial a instituciones educativas. Se basa en una unidad estándar de Cubesat (1 U): un cubo de 10 centímetros de lado con una masa de 1 a 1,33 kilogramos. Actualmente podemos encontrar configuraciones de 1, 2, 6 o incluso 12 U.

Este movimiento ha permitido democratizar el acceso al espacio para todo tipo de empresas, ya sean grandes, pequeñas, organizaciones gubernamentales o instituciones educativas. En los últimos años la popularidad de los nanosatélites no ha dejado de crecer, atrayendo cada vez a más empresas e investigadores, por lo que este proyecto es muy relevante en la actualidad.

Aunque estos avances han reducido considerablemente el precio de construir y poner en órbita un satélite, la dificultad de tal operación o los posibles riesgos (como que no entre en la órbita correcta o que colisione con otro astro) siguen presentes. Se necesitan aplicar múltiples pruebas y herramientas de simulación para asegurar que todo sale como se espera. Este proyecto de fin de grado se enfoca en este aspecto: pretende implementar un banco de pruebas software para simular los componentes y el comportamiento de un nanosatélite que usa cFS como software de vuelo.

## 1.2 Objetivos

El objetivo del proyecto es el diseño e implementación de un banco de pruebas a través de la plataforma NOS<sup>3</sup> para el desarrollo de software para nanosatélites en el entorno de cFE, ambos desarrollados por NASA. A su vez, este objetivo se puede desglosar en diferentes metas o subobjetivos.

A continuación, se enumeran los objetivos que se pretenden conseguir con el desarrollo de este proyecto:

- Crear el entorno de simulación a través de la infraestructura que proporciona NOS<sup>3</sup> y ejecutar el cFS en él.
- Implementación de 2 simuladores de componentes nuevos en la infraestructura NOS<sup>3</sup>.
- Desarrollo de una aplicación cFS que interactúe con ambos simuladores.
- Comprobar que el software de vuelo se ejecuta correctamente en el entorno de simulación

## 1.3 Estructura del documento

En esta sección se detallan los diferentes capítulos que componen la estructura de este documento. Cada capítulo se enumera junto a una breve descripción para facilitar su comprensión. El documento se compone de los siguientes apartados:

**Estado del arte.** En esta sección se exponen el contexto necesario para facilitar la lectura del documento a cualquier tipo de lector.

**Análisis y diseño.** Se detallan los requisitos de usuario y los requisitos de sistema; así como los casos de uso. También se expone el diseño de la solución del proyecto y las matrices de trazabilidad.

**Implementación.** En este apartado se exponen los detalles técnicos relevantes de la implementación de la solución.

**Pruebas y evaluación.** Se describen las pruebas realizadas para validar la implementación del sistema. Se muestra la trazabilidad con los requisitos de software y se evalúa el resultado de las pruebas.

**Plan de proyecto.** Se divide el proyecto en un conjunto de tareas y se muestra la planificación temporal de éstas. En esta sección se añade también el computo del presupuesto del proyecto.

**Marco regulador y entorno socio-económico.** Se expone por un lado el marco legal que se ha tenido en cuenta para la realización del proyecto. Por otro el contexto social y económico en el que se enmarca el proyecto.

**Conclusiones y trabajos futuros.** Se muestran las conclusiones del trabajo tanto generales como personales. Y, por último, se exponen futuros pasos que se pueden llevar a cabo para profundizar sobre este proyecto.





## Capítulo 2.

# Estado del arte

### 2.1 Historia

Los satélites artificiales nacieron durante la guerra fría entre Estados Unidos y La Unión Soviética. Por aquel entonces, el cohete ya era una poderosa arma bélica. Pero su potente propulsión hizo que su uso se extendiera más allá de la batalla.

Antes de 1946, los científicos americanos mandaban globos conectados por radio a 30km de altitud para estudiar la capas más bajas de la atmósfera. A partir de ese año comienzan a usar cohetes V-2, misiles balísticos capturados a los Alemanes, para su investigación. Éstos suponen un gran avance, ya que les permiten llegar a los 200km de altitud y estudiar las capas superiores de la atmósfera.

En el mismo año, el entonces jefe del organismo de investigación aéreo (“Air Staff for Research and Development”) encarga a la organización de investigación y desarrollo americana “Proyecto RAND” un informe sobre la operaciones espaciales. El Proyecto RAND publica “Preliminary Design of an Experimental World-Circling Spaceship” en pleno inicio de la era espacial. Éste se centra entorno a la viabilidad de un vehículo espacial desde el punto de vista de la ingeniería así como en las visiones futuristas como la de Louis Ridenour sobre la importancia de los satélites para ampliar el conocimiento humano o la de Francis Clauser sobre la posibilidad de mandar al hombre al espacio. El interés militar se unió al de los científicos y Estados Unidos anuncia su intención de mandar satélites a principios de 1958.

Sin embargo, en 1957 los soviéticos adelantan a los americanos en la carrera espacial logrando un hito que cambiaría el mundo: el primer lanzamiento exitoso de un satélite artificial, el Sputnik 1.

En un primer momento los científicos de la Unión Soviética tenían pensado enviar un satélite en forma de cono de 1.5 toneladas capaz de realizar múltiples mediciones físicas. Pero, tras conocer la iniciativa estadounidense de mandar 2 satélites en 1958, uno pequeño primero y otro más complejo para enviarlo después, deciden seguir la misma estrategia y desarrollar el Sputnik 1: un satélite compuesto de una esfera de aluminio de 58 centímetros y cuatro finas y largas antenas capaces de transmitir señales de radio. Con una masa de 83,6kg el Sputnik 1 fue lanzado el 4 de octubre desde una pequeña ciudad al

sur de Kazajistán, perteneciente por entonces a la Unión Soviética. La victoria soviética se vio hecha realidad cuando el satélite, tras entrar en órbita, emitió satisfactoriamente una serie de señales en forma de pitido.



Figura 2.1: Sputnik-1

## 2.2 Sistemas Empotrados

Un sistema de control es un sistema que controla el comportamiento de un entorno físico determinado. Se componen de una placa base con un microprocesador incrustado, un conjunto de sensores y otro de actuadores. Los sensores sirven exclusivamente para recibir información relevante del entorno. Los actuadores, por otro lado, sirven para alterar el entorno.

Su funcionamiento es muy simple: ejecutan repetidamente un bucle desde su inicio hasta que se detienen. Dentro del bucle se ejecuta un algoritmo que realiza las siguientes tareas:

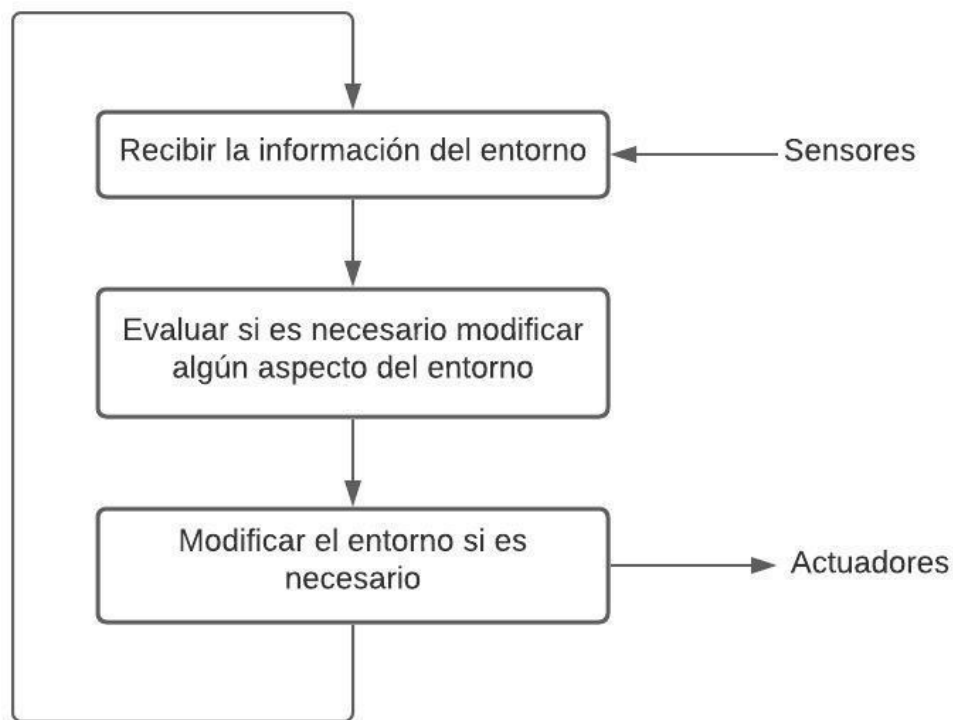


Figura 2.2: Funcionamiento de un sistema empotrado

Los sistemas empotrados (o embebidos) son un tipo de sistemas de control diseñados para realizar una tarea concreta, a diferencia de los computadores tradicionales que son de propósito general. Entre otras características encontramos:

- Están diseñados para cubrir necesidades específicas.
- No suelen ser dispositivos separados. Suelen estar acoplados al dispositivo que controlan.
- Los recursos hardware se suelen ajustar a las necesidades del sistema. Esto logra disminuir el coste económico y el consumo de energía.
- A menudo son también sistemas de tiempo real.
- En general se programan directamente en el lenguaje ensamblador del microprocesador incorporado en el sistema para conseguir una ejecución rápida.
- El software y el hardware se crean a la vez.
- El software del sistema empotrado o también conocido como firmware no se suele cambiar y siempre se sustituye entero.

## Hardware

Un sistema empotrado suele componerse de un módulo electrónico acoplado a otro sistema más grande llamado “host” o anfitrión. Este módulo tiene una arquitectura de hardware similar a la de un ordenador convencional.

El componente principal es el microprocesador o CPU. Es el cerebro del sistema y el que se encarga de realizar los cálculos y de ejecutar el código para el funcionamiento del sistema embebido. Lo podemos encontrar en diferentes formatos: un microprocesador, un microcontrolador de 4, 8, 16 o 32 bits, un procesador de señales digitales (también llamado DSP) o un procesador diseñado a medida (como los FPGA).

Otro componente importante en cualquier sistema electrónico es el de la memoria. En ella se almacena los datos sobre los que la CPU opera. La memoria es volátil: no hay necesidad de que almacene información de forma permanente. Por lo tanto, no es necesario que esté alimentada constantemente. Como se ha mencionado anteriormente, el tiempo de ejecución de estos sistemas es crucial por lo que la memoria debe tener un tiempo de acceso a lectura y escritura lo más breve posible. En la misma dirección encontramos otro tipo de memoria: la memoria caché, basada en el principio de localidad.

El principio de localidad es una propiedad que consiste en que los programas informáticos tienden a reutilizar los datos e instrucciones que usaron anteriormente. Una regla empírica revela que gastan el 90% del tiempo en el 10% del código. Esto implica que se puede predecir con cierta precisión las direcciones de memoria a las que el programa necesitará acceder en un futuro. En los sistemas embebidos este comportamiento se ve acentuado ya que se ejecutan un número pequeño de tareas en un bucle constante.

Las memorias caché son memorias pequeñas y rápidas que se sitúan entre el microprocesador y la memoria principal. De esta forma le ahorra tiempo al procesador, ya que si un dato se encuentra en la memoria caché no hace falta que acceda a la memoria principal. Suele haber más de un nivel de memoria caché nombrados como L1, L2, L3.

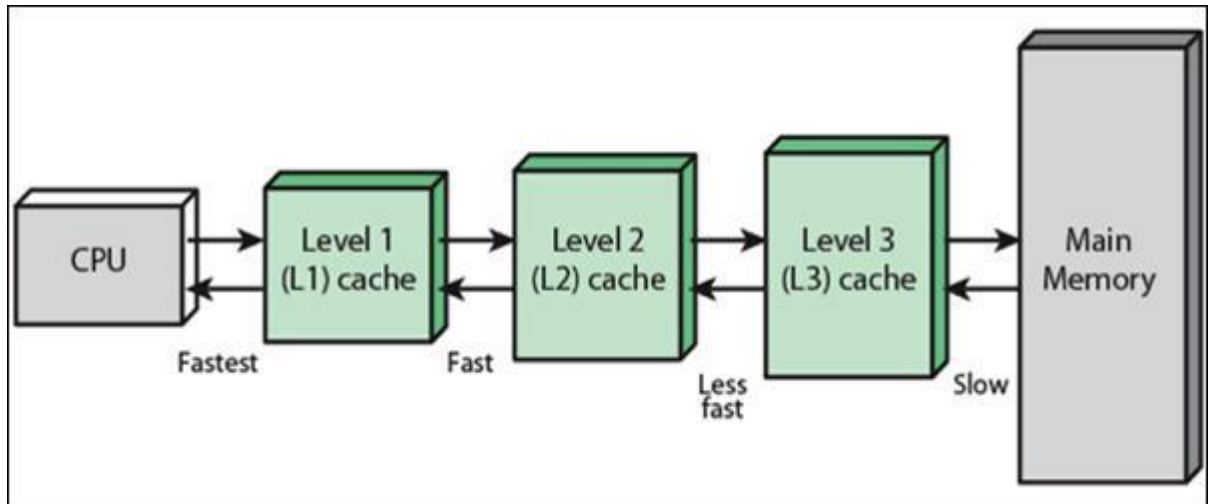


Figura 2.3: Arquitectura de la memoria caché

Otro tipo de memoria presente en los sistemas es la BIOS-ROM. Es una memoria muy pequeña de solo lectura (*Read Only Memory*) que sirve para almacenar el código necesario para iniciar el sistema. Este código, también llamado BIOS, se graba en el proceso de creación del chip y nunca más se modifica.

En ocasiones un sistema empujado también necesita almacenar información de manera permanente. Por ejemplo, el software de una máquina expendedora necesita saber cuántas unidades quedan disponibles de un producto concreto. Para ello se usan memorias no volátiles. Las tradicionales están basadas en discos duros magnéticos. Pero éstos suelen ser grandes y pesados para un sistema embebido. En su lugar, se usan unidades de estado sólido más rápidas y ligeras (aunque más caras también).

Por último, tenemos los elementos de entrada y salida. Los elementos de entrada reciben información del exterior del sistema y varían desde un teclado o ratón hasta entradas de comunicación en serie o paralelo o receptores inalámbricos. Los de salida nos permiten extraer información del sistema para una correcta monitorización. Encontramos pantallas LCD, monitores, salidas de comunicación en serie o paralelo...

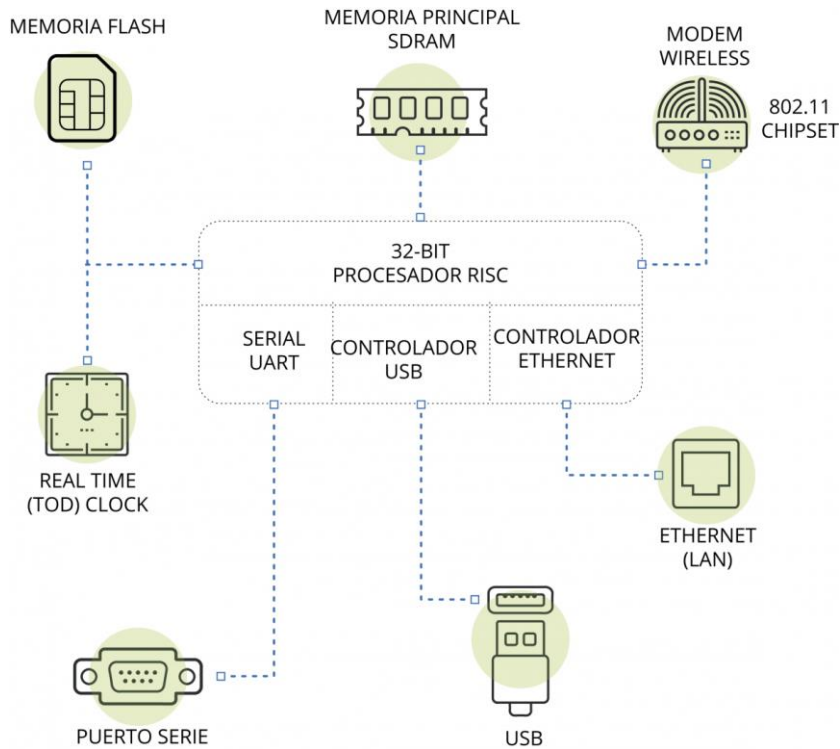


Figura 2.4: Ejemplo de arquitectura de un sistema empujado

En la imagen anterior tenemos un ejemplo de arquitectura de un sistema empujado que cuenta con un procesador de 32 bits, una memoria principal SDRAM, una memoria de almacenamiento FLASH y entradas y salidas UART, USB y ETHERNET.

## Aplicaciones

La versatilidad y gran personalización de los sistemas empujados hacen que su uso llegue a infinidad de sectores y áreas de la industria.

En las fábricas son imprescindibles para tener un control de la temperatura en un horno, el volumen de mezcla que ha entrado en un recipiente o el número de artículos por minuto que pasan por una cinta transportadora.

Además, nos encontramos en la Cuarta Revolución Industrial, concepto creado oficialmente en el Foro Económico Mundial 2016. En el siglo XVIII, la primera revolución industrial comenzó con la mecanización de las tareas gracias a la máquina de vapor. Un siglo más tarde, la segunda revolución aportó la energía eléctrica para la producción en masa. La tercera comenzó en los 80 y sigue en curso. Se basa en la aparición del Internet y de la digitalización de los dispositivos. Y la más actual, la Cuarta Revolución Industrial aparece con los avances en robótica, Internet de las Cosas (IoT), inteligencia artificial, etc. Su objetivo es tener los dispositivos conectados entre sí intercambiándose información para que la automatización sea mucho más avanzada y eficiente. Los dispositivos empujados juegan un papel importante en la tercera y cuarta

etapas y su uso en la industria cada vez es más común y a su vez más avanzado y sofisticado.

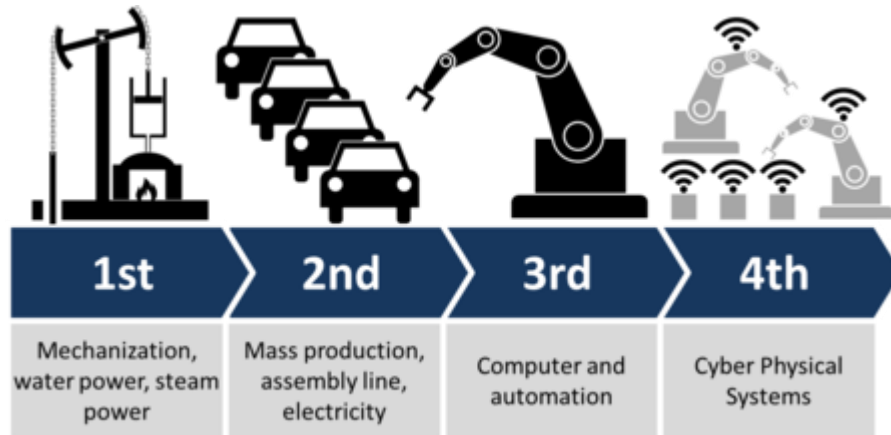


Figura 2.5: Gráfico de las revoluciones industriales

En el mundo de la medicina juegan un papel vital. Lo podemos ver en los marcapasos, los monitores de latidos y presión arterial, los escáneres o los dispositivos usados en cirugía.

En el ámbito doméstico cada vez son más los que se apuntan a la moda de tener una “casa inteligente”. Hay una amplia gama de dispositivos para añadirle a tu hogar: sensores de movimiento, termostatos inteligentes, sistemas para cerrar puertas y ventanas, asistentes de voz, etc.

El sector aeroespacial es uno de los sectores en los que el rendimiento y la seguridad son más importantes. En él se pueden encontrar los sistemas embebidos más sofisticados ya que deben ser pequeños, ligeros y soportar condiciones extremas de presión, temperatura. Por ejemplo: sistemas de control de vuelo, sistemas de gestión del aire y de la presión, dispositivos GPS.

Y un largo etcétera de posibilidades aún por descubrir.

## 2.3 Sistemas de tiempo real

A menudo estos sistemas de control son también sistemas de tiempo real. Un sistema de tiempo real o STR es aquel en el que el tiempo que tarda en interactuar con el entorno



es crucial para su correcto funcionamiento. Si el tiempo de respuesta excede el límite impuesto, se produce una degradación del funcionamiento o un funcionamiento erróneo.

## Tipos

- **Sistema de tiempo real crítico o duro:** es absolutamente necesario que se cumplan los plazos de respuesta. Una respuesta fuera de plazo puede ocasionar graves consecuencias. Un ejemplo de este tipo de sistemas es un control de vuelo de un avión.
- **Sistema de tiempo real acrítico o suave:** Se permite respuestas fuera de plazo de forma ocasional. Aunque por regla general los plazos deben cumplirse. Un ejemplo de sistema acrítico es un sensor de temperatura y humedad.

Un sistema de tiempo real está definido por una lista de eventos a los que debe atender y una serie de requisitos temporales para esas respuestas (sus deadlines). Para cumplir dichos requisitos, a veces hay que evitar usar técnicas que implican tiempos largos y/o poco predecibles. Algunos ejemplos son los siguientes: programación orientada a objetos, Garbage collecting (como el de Java), malloc() y free() comunes de C [4].

Un sistema de tiempo real puede diseñarse e implementarse en código ensamblador directamente. De esta forma las instrucciones se ejecutan de forma muy rápida ya que no tienen que pasar por un compilador. No obstante, cuando la complejidad del problema aumenta, es complicado asegurar siempre que los requisitos temporales se cumplan. Por ello, es conveniente utilizar un sistema operativo de tiempo real que nos ofrezca seguridad en el cumplimiento de dichos requisitos.

## Sistema operativo de tiempo real

Un sistema operativo de tiempo real (RTOS o *Real Time Operating System*) es un software que ha sido diseñado para abarcar problemas de tiempo real de forma simplificada. Al sistema se le imponen una serie de restricciones o requisitos temporales. Si no se cumple alguna de ellas, se dice que el sistema ha fallado.

Estos sistemas permiten al programador diseñar la aplicación como una serie de tareas que se ejecutan de manera concurrente o al mismo tiempo. Aunque esto no es del todo cierto. El sistema reparte el tiempo entre las tareas y las va alternando de manera muy rápida creando un efecto de concurrencia. Este mecanismo se denomina como *multitasking* o multitarea.

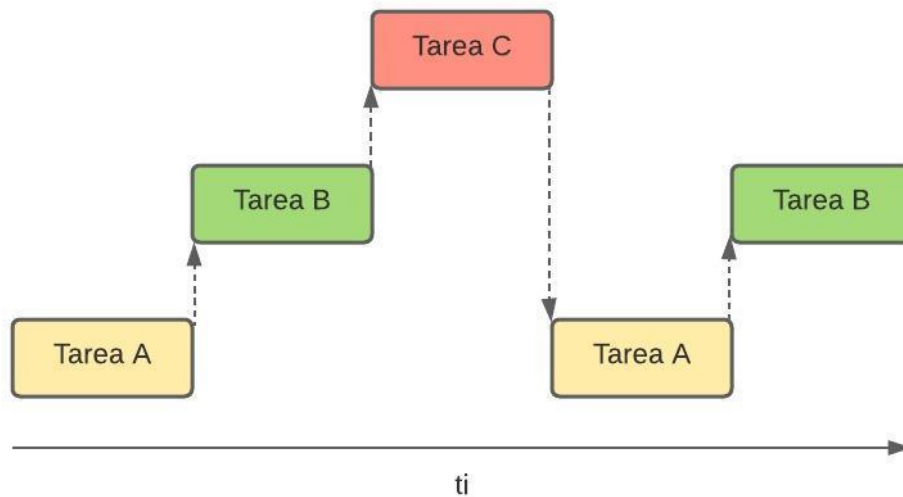


Figura 2.6: Multitask

En la Figura 2.6 se puede observar un ejemplo de multitarea. Durante un intervalo de tiempo  $ti$  efectivamente se ejecutan 3 tareas a la vez: tarea A, tarea B y tarea C. Pero en realidad en ningún momento determinado se ejecutan 2 tareas a la vez, sino que se van intercalando de manera muy rápida.

Los RTOS tienen 5 características fundamentales:

**Determinismo.** Este consiste en que las operaciones que realiza el sistema deben realizarse en intervalos de tiempo predeterminados y determinables con alta probabilidad.

**Responsividad.** La responsividad, por otra lado, refiere al tiempo entre que se acepta la interrupción hasta la ejecución de la tarea correspondiente. El determinismo y la responsividad son aspectos fundamentales ya que de no saber con certeza cuánto tarda en reaccionar el RTOS a un evento sería imposible garantizar un requisito temporal.

**Gestión del sistema.** En un RTOS el programador es capaz de asignar prioridades a las tareas así como de alterar sus requisitos temporales.

**Fiabilidad.** Como hemos mencionado anteriormente, los RTOS son sistemas en los que el rendimiento y la seguridad son fundamentales. Por lo tanto, deben ser fiables y su fiabilidad no debe degradarse en el tiempo.

**Diseño a prueba de fallos.** Aún así, ningún sistema es perfecto. Los fallos ocurren y seguirán ocurriendo; son inevitables. Lo que se debe hacer es intentar minimizar los daños y las pérdidas ocasionadas cuando ocurre uno. Los RTOS están diseñados para afrontar cualquier fallo e intentar que el sistema se recupere lo antes posible para reanudar su ejecución.

Un RTOS suele tener la siguiente arquitectura:

- Programador o scheduler: establece el orden en el que se ejecutarán las tareas.
- Ejecutor o dispatcher: se encarga del inicio y fin de ejecución de tareas así como el cambio de contexto para cambiar de tarea.
- Servicios: drivers necesarios para acceder al hardware, administrador de interrupciones, etc.
- Gestor de configuración o configuration manager: encargado de modificar las configuraciones sin interrumpir la ejecución.
- Gestor de fallas o fault manager: detecta los fallos y los intenta remediar minimizando los daños ocasionados.

Algunos ejemplos de RTOS son:

**QNX.** Es un sistema operativo de tiempo real que asegura una combinación de rendimiento, seguridad y fiabilidad en misiones críticas [5]. Fue desarrollado por QNX Software Systems, una empresa canadiense que fue adquirida por BlackBerry en abril de 2010, convirtiéndose así en subsidiaria de esta última. Usa una arquitectura de microkernel que aísla cada aplicación, controlador y sistema de archivos en sus propias direcciones de espacio reservadas. Esto evita que el fallo de un componente haga que otros componentes o el kernel fallen también.

**VxWorks.** Desarrollado por Wind River Systems, es el RTOS más confiable y ampliamente implementado de la industria para sistemas integrados de misión crítica que deben ser seguros y protegidos [6]. Independientemente de la industria o el tipo de dispositivo, los sistemas empotrados que necesitan una respuesta del orden de microsegundos confían en la seguridad, protección, alto rendimiento y confiabilidad de VxWorks.

**MaRTE OS.** Es un RTOS español. Está desarrollado por el departamento de Ingeniería Software y Electrónica de la Universidad de Cantabria []. Ofrece los servicios definidos en POSIX.13 y soporta aplicaciones mixtas en Ada y C.

**FreeRTOS.** Distribuido bajo la licencia de uso abierto de MIT, FreeRTOS es el líder en el mercado de microcontroladores y microprocesadores pequeños []. Se enfoca en la fiabilidad y la facilidad de uso. El núcleo consta de solo tres archivos implementados

en C para que sea un sistema pequeño, simple y fácil de portar y de mantener. Ha sido portado a un total de 35 plataformas de microcontrolador.

**RTEMS.** Es un RTOS caracterizado por ser de software libre. Consta de un micro-kernel, un entorno o shell y varias aplicaciones diseñadas para hacer funcionar el dispositivo electrónico correctamente. Fue utilizado por ejemplo dentro del proyecto de radio UHF llamado Electra, como parte de la misión Mars Reconnaissance Orbiter de la NASA de 2005. Y es el RTOS seleccionado por SENER para el desarrollo del proyecto.

## 2.4 cFS

El *core Flight System* o cFS es una plataforma y un marco de software reutilizable independiente del proyecto y un conjunto de aplicaciones de software reutilizables [7]. 3 aspectos clave caracterizan la arquitectura cFS: un entorno de ejecución dinámico, un software estructurado en capas y un diseño basado en componentes. La combinación de estas tres características hace que esta arquitectura sea reutilizable en varios proyectos, ya sean proyectos de vuelo de la NASA o proyectos de sistemas embebidos.

Esta plataforma se utiliza junto al simulador NOS<sup>3</sup> para llevar a cabo este proyecto.

## Orígenes

Cada misión de vuelo es un mundo diferente. Y antiguamente el software utilizado también lo era.

Antes de la aparición de cFS, la portabilidad y adaptabilidad del software entre los sistemas operativos y el hardware eran mínimas. Las interfaces estándar entre aplicaciones, casi inexistentes. Entonces para desarrollar un nuevo software de vuelo se aplicaba el enfoque de “clone and own”. Esta práctica consistía en partir de un software ya usado en una misión anterior y adaptarlo para el nuevo proyecto. Los requisitos, el código y la documentación se copiaba y se modificaba acorde a las nuevas necesidades. Pero este proceso conlleva una problemática importante: el proceso es largo, costoso y propenso a errores.

En la rama de software de vuelo de la NASA se percataron de estos problemas y en 2005 comenzaron a trabajar para lograr una solución. Un grupo de ingenieros de software senior realizaron un análisis exhaustivo en el código de las diferentes misiones. Concluyeron con la existencia de elementos comunes al software de todas las misiones y determinaron que el software de vuelo podría rediseñarse en una línea de programas

reutilizable. No solo el código se podía reutilizar, sino que también el diseño, los requisitos, los procedimientos y resultados y la documentación, ahorrando el coste y esfuerzo de rediseñar cada uno de estos elementos una y otra vez para cada misión de vuelo. El resultado final es el cFS: una arquitectura flexible, adaptable y personalizable.

En general se ha demostrado que la arquitectura cFS:

- Reduce el tiempo para desarrollar software de vuelo de alto rendimiento.
- Reduce la incertidumbre del coste del proyecto y el tiempo que tarda en completarse.
- Facilita la reutilización de software.
- Habilita la colaboración entre organizaciones.
- Simplifica el proceso de mantenimiento de software.
- Proporciona una plataforma para la creación de prototipos y conceptos avanzados.
- Proporciona estándares y herramientas comunes de la NASA.

## Estructura

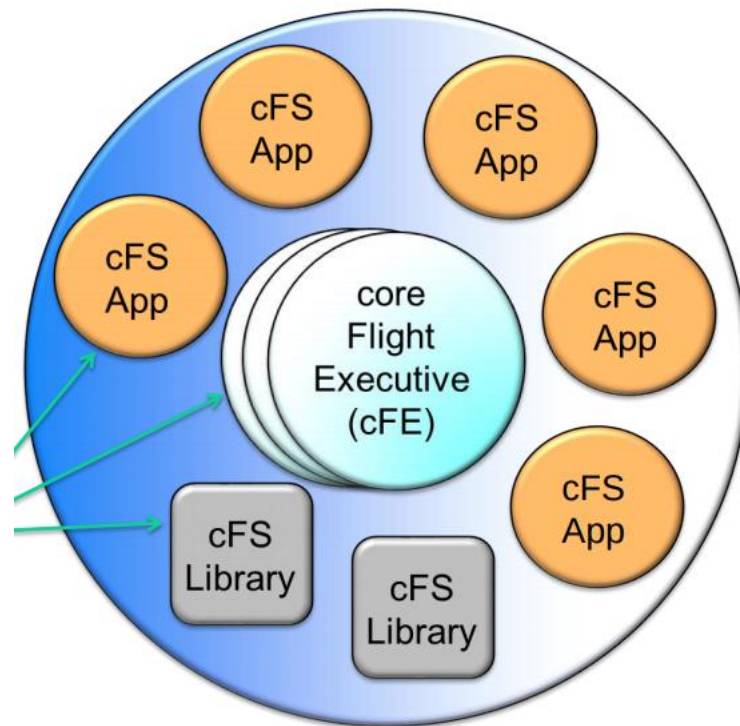


Figura 2.7: Arquitectura del cFS

El cFS se compone de:

**OS Abstraction Layer (OSAL).** Pequeña librería enfocada a aislar el software de vuelo del sistema operativo para que se pueda ejecutar en diferentes RTOS. Desde simuladores de sistemas de vuelo hasta ordenadores de escritorio. La versión actual soporta RTEMS, VxWorks y Linux/x86.

**Platform Support Package (PSP).** Es todo el código necesario para adaptar el cFE a una tarjeta de procesador en particular. Cada misión debe personalizar su PSP. Incluye funciones tal que:

- Código de inicio.
- Lectura, escritura, copia y funciones de protección para la memoria y EEPROM.
- Funciones de reinicio de la tarjeta del procesador.
- Funciones de manejo de errores.

- Funciones del tiempo.

**cFE Core:** entorno de desarrollo y ejecución de aplicaciones. Proporciona un conjunto de servicios básicos que sirven de base para el desarrollo. Define una interfaz API para cada uno. Éste será explicado en detalle en el siguiente apartado.

**Librerías cFS.** Librerías de código usadas por el cFS.

**Aplicaciones cFS.** Por defecto vienen una serie de aplicaciones en el directorio cFS/apps necesarias para el correcto funcionamiento del sistema. A continuación se nombran algunas de ellas con la función que ejercen:

- *CFDP*: envía y recibe datos al *ground system*.
- *Checksum*: realiza la verificación de la integridad de los datos de la memoria, los archivos, etc.
- *Command Ingest Lab*:
- *File Manager*: almacena los datos del vuelo y de *housekeeping*.

Como hemos mencionado anteriormente, una de las características importantes del cFS es que está estructurado a capas. Esta estructura se puede observar en la siguiente imagen:

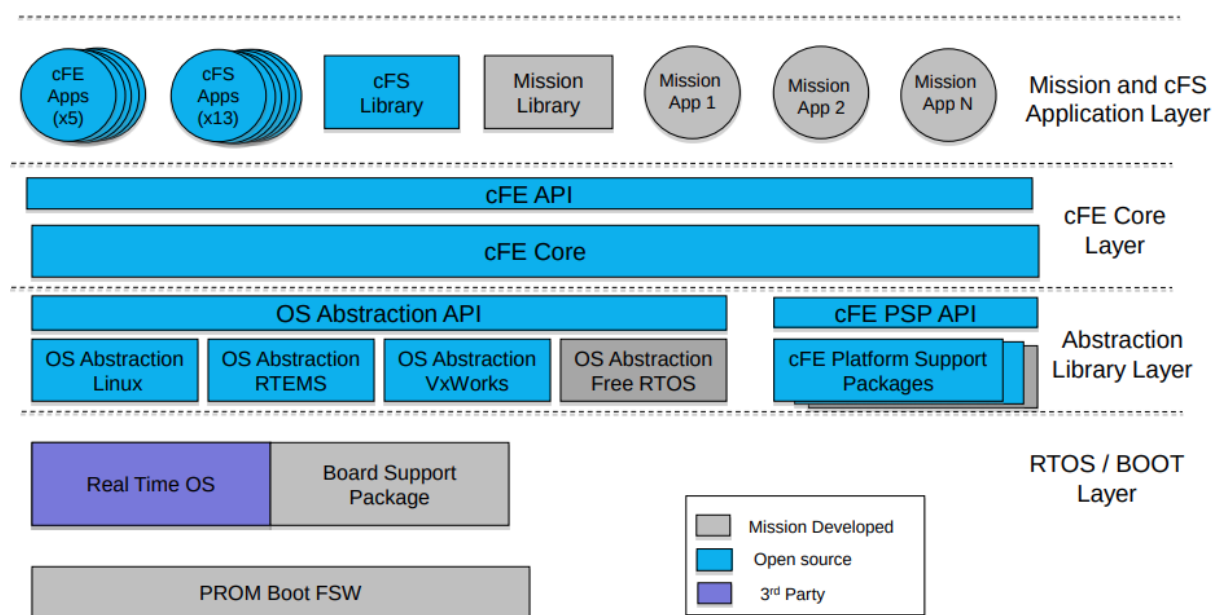


Figura 2.8: Arquitectura del cFS II

Se diferencian 4 capas, numeradas de abajo a arriba:

- Capa 0: es la capa más cercana al hardware. Se encuentra el software encargado de iniciar el RTOS (también llamado *boot*) y el propio RTOS.
- Capa 1: se encuentran el OSAL y el PSP.
- Capa 2: en esta capa se encuentra el cFE core y las API's de las aplicaciones del cFE.
- Capa 3: la última capa es la capa relativa a las aplicaciones y la misión. Contiene las aplicaciones del cFE y cFS, las librerías del cFS, la librería de la misión y una o varias misiones.

## cFE

El cFE es uno de los componentes del cFS. Es un entorno de desarrollo y ejecución de aplicaciones y proporciona un conjunto de servicios básicos que sirven de base para el desarrollo. Exactamente proporciona 5 servicios: servicios ejecutivos, servicios de eventos, bus de software, servicios de tablas y servicios de tiempo. Y define una API para cada uno.

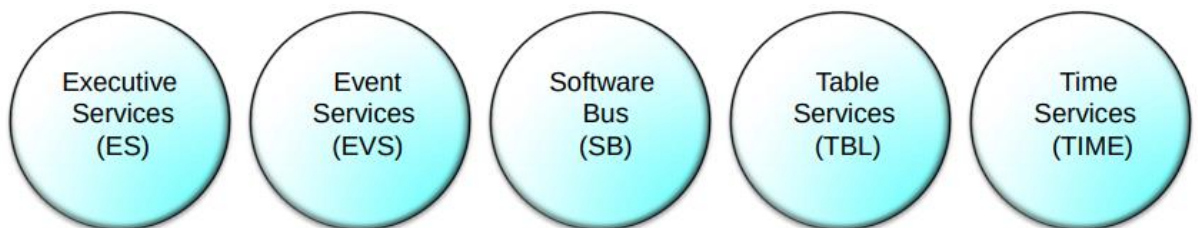


Figura 2.9: Servicios ejecutivos del cFE

Cada servicio lleva integrada una aplicación que permite al usuario interactuar de una manera sencilla con el servicio. El cFE también tiene un conjunto de requerimientos y código para fomentar la reutilización. Los parámetros configurables permiten la personalización del cFE a cada entorno, desde simuladores de sistemas de vuelo hasta ordenadores de escritorio. Además contiene las siguientes herramientas de desarrollo:

- Unit Test Framework (UTF) para realizar tests en las aplicaciones desarrolladas.
- Software de análisis de tiempo para lograr un alto rendimiento en sistemas empotrados de tiempo real.
- Desarrollador de tablas.



- Herramientas de comunicación y telemetría.

En la siguiente tabla se enumera cada uno de los servicios y las funciones que proporcionan.

Servicio básico	Funciones
Executive services	Gestiona el inicio del cFE. Es capaz de iniciar, reiniciar y eliminar aplicaciones cFS y cargar bibliotecas compartidas. Proporciona soporte para controladores de dispositivos. Registra información relativa a excepciones y <i>resets</i> . Administra un registro del sistema para guardar información y errores.
Event services	Proporciona una interfaz para registrar de forma asíncrona mensajes de tipo <i>debug</i> , <i>information</i> o <i>error</i> . Permite enviar mensajes a través de puertos hardware. Permite enviar mensajes en formato corto o largo. Proporciona una interfaz para filtrar mensajes de eventos.
Software bus	Proporciona un servicio de mensajería entre aplicaciones. Envía mensajes a todas las aplicaciones suscritas. Reporta los errores producidos en la transferencia de mensajes. También permite mostrar estadísticas e información sobre el enrutamiento.
Table services	Gestiona las imágenes de las tablas del cFS. Proporciona una interfaz API para que las aplicaciones gestionen las tablas de una manera sencilla. Una tabla se puede compartir entre aplicaciones. Por ello, todas las actualizaciones de las tablas se realizan de forma síncrona con la aplicación para preservar la integridad de los datos de la tabla.
Time services	Proporciona una interfaz API a las aplicaciones para la gestión del tiempo de la nave espacial. Calcula el tiempo de la nave derivado de la misión, el tiempo transcurrido (MET) y un factor de correlación del tiempo (STCF). Distribuye un paquete de comando “1Hz wakeup” y un paquete “time at the tone” que contiene el tiempo correcto en el instante de la señal de 1Hz.

Tabla 2.1: Servicios ejecutivos de cFE

---

## 2.5 Bancos de prueba

El *testing* o el proceso de realizar pruebas a una aplicación es una de las actividades más importantes en el desarrollo de un proyecto ya que contiene las herramientas necesarias para garantizar la calidad de cualquier producto. Además, el *testing* debe estar presente en todas las fases del proyecto (desde la toma de requerimientos hasta la entrega final del producto) para detectar a tiempo cualquier tipo de error y garantizar que cumple con los requisitos definidos.

En la industria aeroespacial este proceso cobra más importancia dada la alta inversión económica que tienen los proyectos. Y es que tan solo un pequeño error puede echar a perder años de esfuerzo y trabajo y millones de euros invertidos en un proyecto. Además, en otros sectores es más factible hacer pruebas en el entorno real del dispositivo. Por ejemplo, en una aplicación software para coches. En el caso de dispositivos para satélites, el entorno real se complica mucho más.

### Hardware-in-the-loop

Una herramienta que se suele emplear para realizar pruebas a dispositivos embebidos es un simulador de hardware en bucle o HILS (*hardware-in-the-loop simulator*). El HILS es un sistema que engaña al sistema embebido haciéndole creer que está operando con entradas y salidas del mundo real, en tiempo real. Esto lo realiza mediante el uso de modelos matemáticos de los sistemas dinámicos que actúan sobre el dispositivo a probar (en el caso del satélite, usaría las leyes físicas presentes en el espacio). De este modo, forma lo que se denomina como simulación de planta. El sistema empujado interactúa directamente con esta simulación de planta. Además, el HILS permite desarrollar y probar software con hardware real o simulado. Es decir que el software se puede probar sin tener todo el hardware real disponible.

En la industria aeroespacial se suelen emplear ciclos de desarrollo cortos: se desarrolla el software mientras que en paralelo se prueba con simuladores HIL para mejorar el diseño y la integración. Este proyecto se centra en el entorno de simulación NOS<sup>3</sup>. Antes de indagar directamente sobre el NOS<sup>3</sup>, se describen los principales proveedores de simuladores de hardware en bucle: Opal-RT, dSPACE y ETAS.

### Opal-RT

Con sede en Montreal (Canada) [], actualmente es la compañía líder en simulación en tiempo real y equipos HIL. Ofrecen soluciones de tiempo real a más de 800 clientes

en 40 países de diferentes industrias: industria aeroespacial, automovilística, electrónica y de energías.

Aunque ofrecen soluciones de *testing* para todo el proceso de desarrollo, en la siguiente tabla se muestran los principales productos hardware que ofrecen y una pequeña comparativa de sus características.





				
Modelo	OP4510	OP5660XG	OP5707XG	OP5031XG
Familia CPU	Intel Xeon E3	2nd gen Intel Xeon Scalable Processors	2nd gen Intel Xeon Scalable Processors	2nd gen Intel Xeon Scalable Processors
Núcleos	4	4, 8 o 16	4, 8 o 16	4, 8, 16 o 44
Frecuencia	2.60 GHz	2.60, 3.80 o 3.30 GHz	3.80 o 3.30 GHz	2.60, 3.80, 3.30 o 2.10 GHz
Módulos de entrada/salida	4	8	8	n/a
Número máximo de canales de entrada/salida	140	256	256	n/a

Tabla 2.2: Comparativa de modelos OPAL-RT

## d-SPACE

Otro gran proveedor de soluciones para la simulación de sistemas complejos es d-SPACE [8]. Emplea a más de 1800 personas en todo el mundo y tiene su sede en Alemania. Se enfoca principalmente en el cambio dinámico de la industria automotriz: conducción autónoma, electromovilidad, digitalización, etc. Aunque también está presente en el sector aeroespacial y de automatización industrial.

Una de sus gamas para la simulación se llama SCALEXIO. Es una línea de producto modular que cubre una amplia gama de áreas de uso como la creación de prototipos de control rápido (RCP), la adquisición y monitoreo de datos (DAQ), las pruebas de hardware-in-the-loop (HIL) y los bancos de pruebas. En la siguiente tabla se muestra una comparativa de los sistemas que componen esta línea de productos.




			
Modelo	LabBox	AutoBox	Rack Version
Área de uso	Desarrollo y prueba de funciones	Desarrollo de funciones	Todos los dominios de <i>testing</i>
CPU	Procesador DS6001 / SCALEXIO Processor Unit	Procesador DS6001	SCALEXIO Processor Unit
Módulos de entrada/salida	Hasta 18	Hasta 7	Hasta 20
Conectores	Sub-D		Hypertac
Ambiente de uso	Laboratorio	Vehículo	Laboratorio

Tabla 2.3: Comparativa de modelos d-SPACE

ETAS es una empresa que provee soluciones para el desarrollo de sistemas empotrados [9]. Fue fundada en 1994 y está presente en más de 10 países en Europa, Asia y América. Sus principales clientes son fabricantes de automóviles, proveedores de componentes automotrices y proveedores de servicios de ingeniería.

Las pruebas HIL tienen gran importancia para garantizar la seguridad de los automóviles ya que son capaces de reproducir con gran detalle las condiciones de simulación. Ofrece 2 productos de simulación de fallos en tiempo real y simulación de hardware-in-the-loop exclusivos para automóviles: ETAS ES4440 y ETAS ES4441.



Figura 2.10: ETAS ES4440

El ETAS ES4440 es un módulo compacto de simulación que permite probar el sistema con el vehículo parado o con el vehículo en un dinamómetro. Se puede utilizar independientemente o en combinación con otros sistemas HIL.



Figura 2.11: ETAS ES4441

El otro módulo de simulación es el ES4441. Es más grande y está integrado en una caja de conexiones. Ofrece, entre otras cosas, un adaptador para integrar fácilmente un voltímetro o amperímetro. La posibilidad de cambiar la toma de tierra por una resistencia programable. Así como 255 canales de hasta 30 V.

## 2.6 NOS3

El simulador *NASA Operational Simulator for Small Satellites* (NOS<sup>3</sup>) es un banco de pruebas software de código abierto para nanosatélites. Los bancos de prueba descritos en la anterior sección eran componentes físicos (hardware). Sin embargo, el NOS<sup>3</sup> es una colección de ejecutables y bibliotecas de Linux. Se puede ejecutar sin tener ningún hardware físico conectado.

### Background

El *Simulation-to-Flight 1* (STF-1) es un nanosatélite desarrollado por el equipo de *Katherine Johnson Independent Verification and Validation Facility* (IV&V) en Virginia Occidental en colaboración con Consorcio de Becas Espaciales de Virginia Occidental y Universidad de Virginia Occidental. Tiene un tamaño de 3U (unidades del estándar Cubesat) y fue lanzado por la NASA el 16 de diciembre de 2018. El objetivo principal de este satélite era demostrar que se puede llevar a cabo un lanzamiento de este tipo usando bancos de prueba únicamente software. El simulador usado es el NOS<sup>3</sup>. Es más, las

simulaciones que trae por defecto están basadas en el hardware que se usó en el cubesat STF-1.

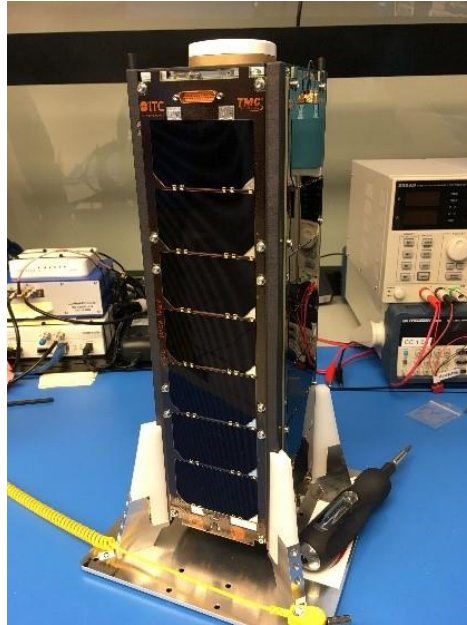


Figura 2.12: Cubesat STF-1

## Componentes

El NOS<sup>3</sup> se ejecuta en una máquina virtual de Linux. Esta máquina virtual juega un papel de un contenedor software para garantizar que todas las dependencias se cumplen antes de ejecutar el programa. Estos componentes se listan en la siguiente tabla.

Componente	Descripción
Oracle VirtualBox	Oracle VirtualBox es un software de virtualización de máquinas virtuales.
Vagrant	Vagrant es una herramienta para la creación y configuración de máquinas virtuales de VirtualBox. Entre sus opciones se incluye la instalación de paquetes, creación de usuarios, manipulación de archivos y directorios, etc.

NOS Engine	El NOS <i>Engine</i> o motor de NOS es un componente software que se encarga de simular buses hardware. Permite que se comuniquen el software de vuelo y los componentes de hardware simulados.
Componentes de hardware simulados	Una colección de componentes de hardware simulados que se conectan al motor de NOS y proporciona la entrada y salida de hardware al software de vuelo.
42	Algunos de los componentes hardware necesitan datos del entorno ambiental. 42 es una herramienta de código abierto de visualización y simulación de la altitud y la órbita de la nave espacial desarrollada por la NASA. Se utiliza para proporcionar datos ambientales a los componentes hardware.
cFS	El cFS está incluido en el NOS <sup>3</sup> . Se utiliza como base para desarrollar el software de las misiones.
COSMOS	COSMOS es un sistema de tierra ( <i>ground system</i> ) de código abierto desarrollado por la empresa Ball Aerospace. Proporciona la comunicación y control remoto del software de vuelo.
CFC	El CFC (COSMOS File Creator) es una herramienta que permite la generación de archivos de control remoto y telemetría.

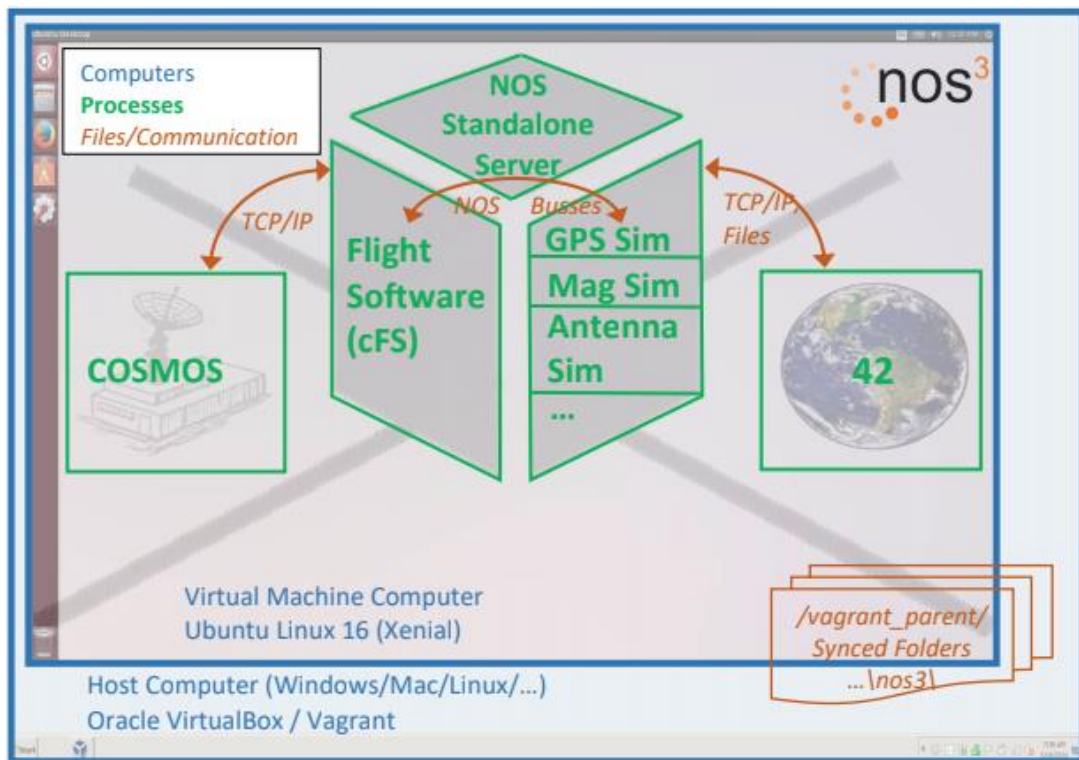
Tabla 2.4: Componentes que forman el NOS<sup>3</sup>

## Arquitectura

Como se ha descrito en la sección anterior, el NOS<sup>3</sup> proporciona las herramientas necesarias para crear y configurar una máquina virtual. Esta máquina actúa como un contenedor: contiene las dependencias necesarias para el correcto funcionamiento del simulador. Aunque si se desea evitar el uso de la VM, se pueden ejecutar directamente los ejecutables del NOS<sup>3</sup> en un host Ubuntu 16.04, habiendo instalado los paquetes necesarios antes.

En este proyecto se ha implementado el NOS<sup>3</sup> a través de la máquina virtual y las herramientas que se proporcionan para ello. En la Figura 2.13 se ilustran los componentes y cómo se conectan entre ellos.



Figura 2.13: Arquitectura NOS<sup>3</sup>

El host y la máquina virtual comparten la carpeta *nos3*. En ella se encuentra el código y las bibliotecas. El cFS y los simuladores de hardware se comunican entre sí a través de los buses que proporciona el *NOS Engine*. Por otro lado, el sistema del satélite recibe datos ambientales del 42 a través del protocolo TCP/IP. Y el COSMOS interactúa con el satélite usando el mismo protocolo.

## NOS Engine

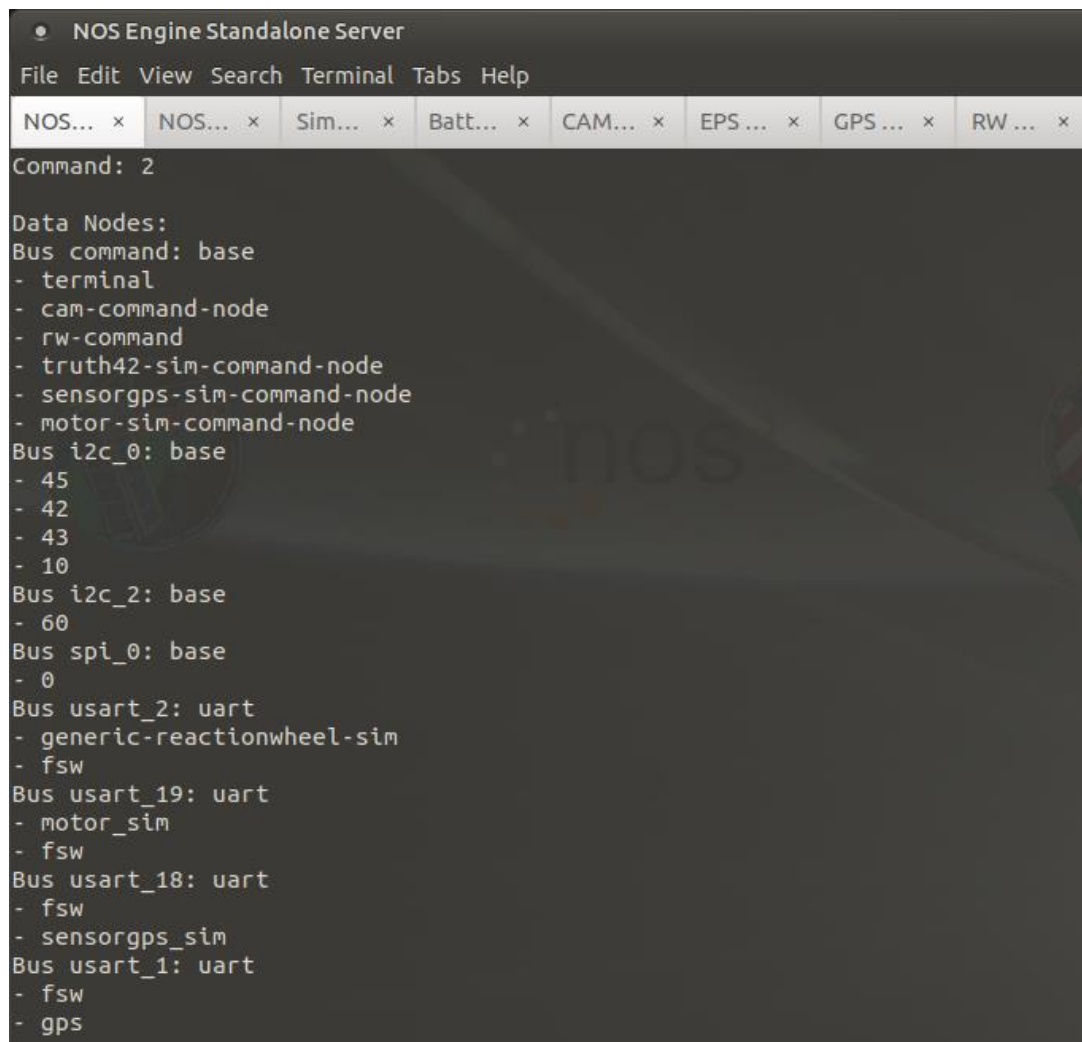
El NOS Engine o el motor de NOS es un componente cuya función es simular los buses hardware para que el software de vuelo pueda comunicarse con los simuladores hardware de una manera rápida, sencilla y reutilizable. Con un diseño modular, la biblioteca del NOS Engine proporciona un core que se puede ampliar para simular protocolos de comunicación específicos. Los protocolos que soporta son: MIL-STD-1553, SpaceWire, I2C, SPI y UART.

El motor de NOS se basa en un modelo compuesto por 2 tipos de objetos fundamentales:

- **Nodos.** Un nodo es cualquier tipo de punto final en el sistema, capaz de enviar y recibir mensajes.

- **Buses.** Una serie de nodos pertenecen a un grupo, denominado bus. Un nodo puede pertenecer a más de un bus y un bus puede contener un número arbitrario de nodos. Sin embargo, dentro de un mismo bus cada nodo debe tener un nombre diferente. Entre nodos del mismo bus pueden intercambiar mensajes, pero no pueden comunicarse con nodos de otros buses.

Cada componente de hardware se simula como un nodo en el sistema. Cuando el motor de NOS está en ejecución, se pueden consultar los buses y nodos activos mediante su consola. En la siguiente imagen se muestra un ejemplo de la salida de esta consulta.



```
NOS Engine Standalone Server
File Edit View Search Terminal Tabs Help
NOS... x NOS... x Sim... x Batt... x CAM... x EPS ... x GPS ... x RW ... x
Command: 2
Data Nodes:
Bus command: base
- terminal
- cam-command-node
- rw-command
- truth42-sim-command-node
- sensorgps-sim-command-node
- motor-sim-command-node
Bus i2c_0: base
- 45
- 42
- 43
- 10
Bus i2c_2: base
- 60
Bus spi_0: base
- 0
Bus usart_2: uart
- generic-reactionwheel-sim
- fsw
Bus usart_19: uart
- motor_sim
- fsw
Bus usart_18: uart
- fsw
- sensorgps_sim
Bus usart_1: uart
- fsw
- gps
```

Figura 2.14 Consola de NOS Engine

## Capítulo 3.

# Análisis y diseño.

En este capítulo se describe el análisis del proyecto realizado. Se detallan los requisitos de usuario y los requisitos de sistema; así como los casos de uso. En la sección 3.5 se describe el diseño de la solución. Por último, se realizan las matrices de trazabilidad para hacer un seguimiento de los requisitos.

### 3.1 Requisitos de usuario

Estos requisitos describen la funcionalidad y restricciones del sistema a desarrollar.

Existen 2 tipos de requisitos de usuario:

**Requisitos de capacidad.** Describen la funcionalidad que el sistema debe implementar.

**Requisitos de restricción.** Imponen restricciones sobre los requisitos de capacidad.

Los requisitos se especifican a través de la plantilla de la Figura X.X. Ésta contiene los siguientes atributos:

**Identificador.** Define el formato en el cual el requisito será identificado. Será de la forma XX-UR-YY donde XX puede ser CA (de capacidad) o RE (de restricción), UR identifica que es un requisito de usuario, YY es un número de secuencia que tiene comienzo en 01.

**Descripción.** Una breve especificación del requisito haciendo uso de un lenguaje no ambiguo.

**Necesidad.** La prioridad del requisito para el cliente. Esta tomará el valor esencial, conveniente u opcional.

**Prioridad.** La prioridad para el desarrollador. Esta tomará el valor alta, media o baja.

Estabilidad. Indica si varía durante el proceso de desarrollo. Esta tomará el valor no cambia, cambiante o muy inestable.

**Verificabilidad** del requisito. Toma el valor alta, media o baja.

**XX-UR-YY**

---

Descripción:

Necesidad:

Prioridad:

Estabilidad:

Verificabili-  
dad:

Figura 3.1: Plantilla de requisito de usuario

## Requisitos de capacidad

### CA-UR-01

Descripción: El banco de pruebas contiene al menos 2 simuladores de componentes: un sensor y un actuador

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

### CA-UR-02

Descripción: El banco de pruebas contiene al menos 1 aplicación de cFE

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

### CA-UR-03

Descripción: El sensor y la aplicación de cFE deben estar conectados mediante un hardware bus simulado.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

### CA-UR-04

Descripción: El actuador y la aplicación de cFE deben estar conectados mediante un hardware bus simulado.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

### CA-UR-05

Descripción: El sensor simulado debe enviar periódicamente mensaje a la aplicación con la información relativa del sensor.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

### CA-UR-06

Descripción: El sensor simulado debe ser capaz de leer las coordenadas del satélite.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**CA-UR-07**

Descripción: Los mensajes enviados por el sensor deben contener las coordenadas del satélite.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**CA-UR-10**

Descripción: Los mensajes enviados al actuador deben contener el sentido y la dirección en los que el actuador debe actuar.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**CA-UR-08**

Descripción: Los mensajes enviados por el sensor deben contener un identificador del mensaje.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabilidad: Alta

**CA-UR-11**

Descripción: Los mensajes enviados al actuador deben contener un identificador de mensaje.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabilidad: Alta

**CA-UR-09**

Descripción: La aplicación debe procesar los mensajes recibidos y, en caso de ser necesario, comunicarse con el actuador.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: Cambiante

Verificabilidad: Alta

**CA-UR-12**

Descripción: Cuando el actuador recibe un mensaje debe imprimir en consola qué acción va a ejecutar.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**Requisitos de restricción****RE-UR-01**

Descripción: El banco de pruebas debe estar implementado mediante la librería software de NOS<sup>3</sup>.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**RE-UR-02**

Descripción: La comunicación entre los componentes se debe realizar a través de los buses que proporciona el NOS Engine.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**RE-UR-03**

Descripción: Los buses utilizados deben usar un protocolo de transmisión soportado por las librerías de NOS<sup>3</sup>.

Necesidad: Conveniente

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**RE-UR-04**

Descripción: Ambos simuladores deben estar conectados al bus de tiempo que proporciona el NOS Engine.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**RE-UR-05**

Descripción: La aplicación de cFE debe estar conectada al bus de tiempo que proporciona el NOS Engine.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

**RE-UR-06**

Descripción: Se debe utilizar el servicio de eventos para notificar acciones y errores al sistema.

Necesidad: Conveniente

Prioridad: Baja

Estabilidad: No cambia

Verificabilidad: Alta

**RE-UR-07**

Descripción: El sensor y el actuador deben estar conectados a dos buses software diferentes.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabili-

dad:

**RE-UR-08**

Descripción: El sensor debe enviar las 3 coordenadas (x,y,z) en formato de punto flotante con hasta x decimales.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabili-

dad:

**RE-UR-09**

Descripción: La ejecución de la solución completa debe ser en tiempo real.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabili-

dad:

**RE-UR-10**

Descripción: La aplicación debe comprobar que los mensajes recibidos provienen del sensor.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabili-

dad:

**RE-UR-11**

Descripción: La aplicación debe gestionar posibles errores ocasionados durante la comunicación.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabili-

dad:

**RE-UR-12**

Descripción: El actuador debe comprobar que los mensajes recibidos provienen de la aplicación.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabili-

dad:



**RE-UR-13**

---

Descripción: El actuador debe gestionar posibles errores ocasionados durante la comunicación.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabilidad: Alta

### 3.2 Casos de uso

En esta sección se especifican los casos de uso del banco de pruebas. Permiten visualizar las funcionalidades que ofrece el sistema. En la Figura 3.2 se pueden ver descritos mediante un diagrama UML.

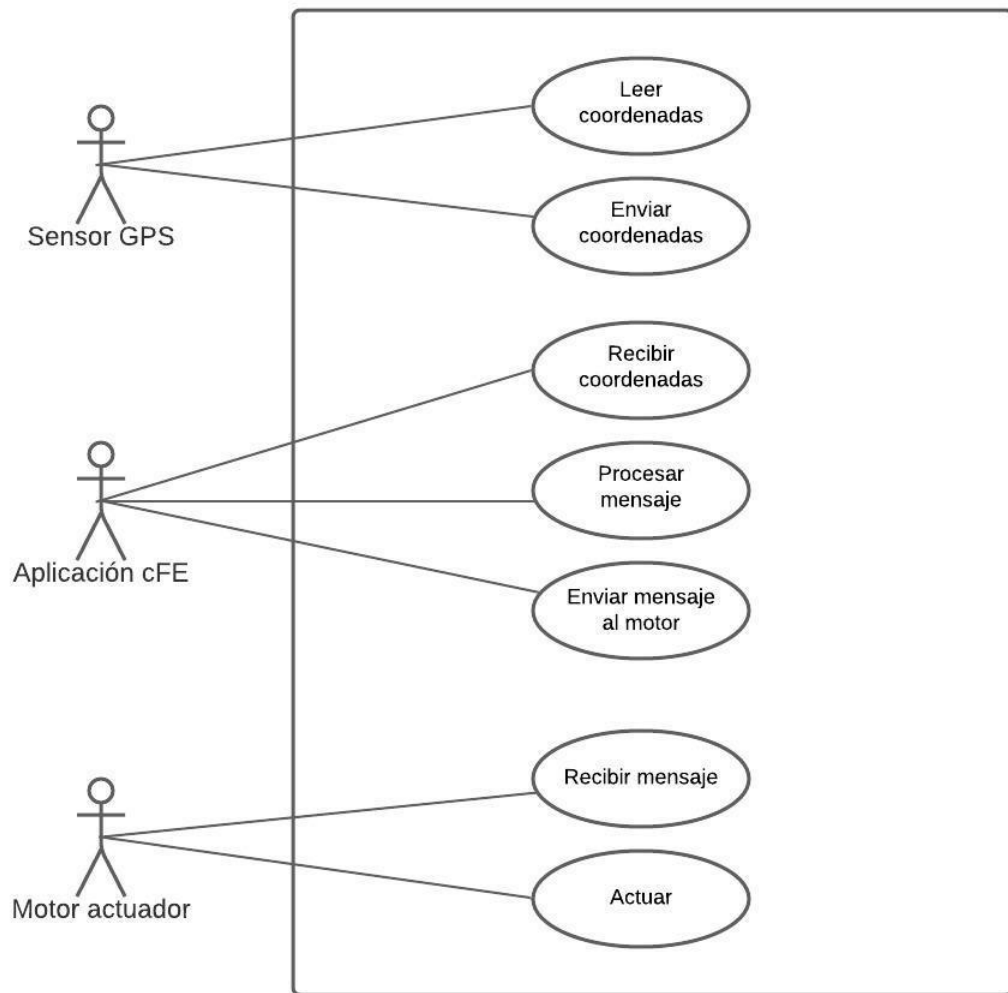


Figura 3.2: Diagrama UML

Una plantilla como la de la figura X.X se usa para detallar los casos de uso. Los atributos que componen esta plantilla son los siguientes:

**Identificador.** Identifica unívocamente al caso de uso. Será UC-XX, donde XX indica el número de secuencia comenzando en 01.

**Nombre.** Descripción breve del caso de uso.

**Actores.** Agente externo que ejecuta el caso de uso.

**Objetivo.** Propósito del caso de uso.

**Descripción.** Pasos que debe seguir el actor para ejecutar el caso de uso.

**Pre-condición.** Condiciones previas que se deben cumplir para ejecutar el caso de uso.

**Post-condición.** Condiciones que se deben cumplir después de ejecutar el caso de uso.

**UC-01**

---

Nombre:

Actores:

Objetivo:

Descripción:

Pre-  
condición:

Post-  
condición:

Figura 3.3: Plantilla de caso de uso

**UC-01**


---

Nombre:	Leer coordenadas
Actores:	Simulador de sensor GPS
Objetivo:	Leer las coordenadas en el espacio del nanosatélite.
Descripción:	El sensor GPS simula que lee las coordenadas del satélite para procesarlas y/o enviarlas.
Pre-condición:	
Post-condición:	El sensor ha leído correctamente las 3 coordenadas (x,y,z) del satélite

**UC-02**


---

Nombre:	Enviar coordenadas
Actores:	Simulador de sensor GPS
Objetivo:	Enviar un mensaje que contenga las coordenadas del satélite
Descripción:	El simulador del sensor GPS envía un mensaje a la aplicación de cFE que contiene las coordenadas del satélite y un identificador del mensaje.
Pre-condición:	<ul style="list-style-type: none"> <li>▪ Estar suscrito al mismo software bus que la aplicación</li> <li>▪ Que haya transcurrido el periodo de tiempo establecido para enviar el mensaje</li> </ul>
Post-condición:	El paquete se envía al software bus que comparten el sensor y la aplicación de cFE

**UC-03**


---

Nombre:	Recibir coordenadas
Actores:	Aplicación de cFE
Objetivo:	Recibir el paquete que contiene las coordenadas del sensor GPS
Descripción:	La aplicación de cFE recibe del sensor GPS un mensaje que contiene un las 3 coordenadas (x,y,z).
Pre-condición:	<ul style="list-style-type: none"> <li>▪ Estar suscrito al mismo software bus que el sensor GPS</li> <li>▪ Comprobar que se ha recibido un mensaje</li> <li>▪ El <i>header</i> del mensaje debe ser el correcto</li> </ul>
Post-condición:	El mensaje se ha recibido correctamente

**UC-04**

---

Nombre:	Procesar mensaje
Actores:	Aplicación de cFE
Objetivo:	Procesar el mensaje recibido del sensor GPS
Descripción:	La aplicación de cFE comprueba si las coordenadas recibidas siguen la trayectoria preestablecida. En caso de que no la sigan, procede a enviar un paquete al actuador. En el caso contrario, sigue su ejecución
Pre-condición:	Haber recibido correctamente un mensaje del sensor GPS
Post-condición:	La aplicación ha procesado el mensaje y ha decidido si debe comunicarse con el actuador o no

**UC-05**

---

Nombre:	Enviar mensaje al actuador
Actores:	Aplicación de cFE
Objetivo:	Enviar un mensaje que contiene cómo debe actuar el motor
Descripción:	La aplicación de cFE envía un mensaje al motor que contiene en qué dirección y sentido debe actuar.
Pre-condición:	<ul style="list-style-type: none"><li>▪ Estar suscrito al mismo software bus que el actuador</li><li>▪ Haber procesado un mensaje del sensor GPS en el que las coordenadas no sigan la trayectoria establecida</li></ul>
Post-condición:	Se ha enviado el mensaje al actuador

**UC-06**

---

Nombre:	Recibir mensaje
Actores:	Simulador del actuador
Objetivo:	Recibir un mensaje de la aplicación de cFE
Descripción:	El actuador recibe un mensaje de la aplicación de cFE que contiene en qué dirección y sentido debe actuar.
Pre-condición:	<ul style="list-style-type: none"><li>▪ Estar suscrito al mismo software bus que la aplicación</li><li>▪ Comprobar que se ha recibido un mensaje</li><li>▪ Comprobar que el header del mensaje es el correcto</li></ul>
Post-condición:	El mensaje se ha recibido correctamente

**UC-07**


---

Nombre:	Actuar
Actores:	Simulador del actuador
Objetivo:	Actuar para corregir la trayectoria del satélite
Descripción:	El motor simula que actúa en la dirección y el sentido recibido para corregir la trayectoria del satélite. Para ello imprime un mensaje en la consola del simulador.
Pre-condición:	Haber recibido un mensaje de la aplicación de cFE
Post-condición:	El satélite ha corregido su trayectoria. Se ha impreso en la consola del motor en qué dirección y sentido ha actuado

### 3.3 Requisitos de software

En esta sección se especifican los requisitos de software. Estos requisitos se han obtenido a partir de los requisitos de usuario de la Sección 3.1 y describen las funcionalidades y limitaciones que el analista ha interpretado del proceso.

Los requisitos de software se dividen en 2 tipos:

**Requisitos funcionales.** Especifican las características funcionales del software.

**Requisitos no funcionales.** Especificaciones adicionales que no aportan una nueva funcionalidad.

El formato de los requisitos de software se define a tal y como se muestra en la plantilla de la Figura 3.6, que contiene los siguientes atributos:

**Identificador.** Define el formato en el cual el requisito será identificado. Tiene el formato XX-YY-ZZ donde:

- XX puede ser RF (si es requisito funcional) o NF (si es un requisito no funcional).
- YY describirá los tipos de requisitos, SR (para requisitos funcionales y para no funcionales). ZZ es un número de secuencia que tiene comienzo en 01.

**Necesidad.** La prioridad del requisito para el cliente. Esta tomará el valor esencial, conveniente o opcional.

**Prioridad.** La prioridad para el desarrollador. Esta tomará el valor alta, media o baja.

**Estabilidad.** Indica si varía durante el proceso de desarrollo. Esta tomará el valor no cambia, cambiante o muy inestable.

**Verificabilidad** del requisito. Toma el valor alta, media o baja.

**Origen.** Hace referencia a que requisitos de usuario originaron este requisito.

**XX-YY-ZZ**

---

Descripción:

Necesidad:

Prioridad:

Estabilidad:

Verificabili-  
dad:

Origen:

Figura 3.4: Plantilla de requisito de sistema

## Requisitos funcionales

### RF-SR-01

Descripción: Ambos simuladores de componentes deben estar registrados en el archivo nos3-simulator.xml.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-01

### RF-SR-02

Descripción: El sensor se conecta al bus mediante las llamadas a la librería *hwlib* del NOS Engine.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-03

### RF-SR-03

Descripción: El sensor GPS debe enviar un mensaje cada 3 s.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-05

### RF-SR-04

Descripción: El sensor debe incluir en los mensajes los campos:

- Uint8 id
- Float x
- Float y
- Float z.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-06, CA-UR-07, CA-UR-08

### RF-SR-05

Descripción: El sensor debe incluir un *header* y un *trailer* en el mensaje. Cada uno debe ocupar 1 byte de tamaño.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-08



**RF-SR-06**

Descripción: La aplicación de cFE debe estar incluida en los *Makefiles* necesarios y en el script de ejecución de inicio (*cfe\_es\_startup.scr*).

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabili-

dad:

Origen: CA-UR-02

**RF-SR-07**

Descripción: La aplicación se conecta al bus mediante las llamadas a la librería *hwlib* del NOS Engine.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabili-

dad:

Origen: CA-UR-04

**RF-SR-08**

Descripción: La aplicación debe comprobar la autenticidad del paquete mediante la inspección de su header.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabili-

dad:

Origen: CA-UR-09

**RF-SR-09**

Descripción: La aplicación decide si debe comunicarse con el actuador o no en función de si la trayectoria sigue los valores preestablecidos.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: Cambiante

Verificabili-

dad:

Origen: CA-UR-09

**RF-SR-10**

Descripción: La aplicación debe incluir en los mensajes el campo:

- Uint8 id
- Uint8 dirección
- Uint8 sentido

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabili-

dad:

Origen: CA-UR-10, CA-UR-11

**RF-SR-11**

Descripción: La aplicación de cFE incluye un header de 1 byte en el mensaje.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabili-

dad:

Origen: CA-UR-11

**RF-SR-12**

Descripción: El actuador se conecta al bus mediante las llamadas a la librería *hwlib* del NOS Engine..

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-04

**RF-SR-13**

Descripción: El actuador debe comprobar la autenticidad del paquete mediante la inspección de su header.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-12

**RF-SR-14**

Descripción: El actuador cada vez que se ejecuta debe imprimir en consola el mensaje:

“Motor Activated on Axis X/Y/Z POSITIVE/NEGATIVE”.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-12

## Requisitos no funcionales

### NF-SR-01

Descripción: El protocolo utilizado en los buses de software será UART.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-01, CA-UR-02, CA-UR-03

### NF-SR-02

Descripción: Ambos simuladores deben estar conectado al software bus “command”. Éste se encarga de distribuir el tiempo.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-04

### NF-SR-03

Descripción: La aplicación de cFE debe estar conectado al software bus “command”. Éste se encarga de distribuir el tiempo.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-05

### NF-SR-04

Descripción: Se notificará al sistema del inicio de cada simulador así como de la aplicación.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-06

### NF-SR-05

Descripción: Se notificará al sistema del envío y recepción de cualquier mensaje.

Necesidad: Conveniente

Prioridad: Media

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-06

**NF-SR-06**

Descripción: La aplicación de cFE debe gestionar los posibles errores en los mensajes recibidos y reportarlos al sistema mediante el servicio de eventos.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-06, CA-UR-11

**NF-SR-07**

Descripción: El sistema debe desarrollar en el lenguaje C++ utilizando la infraestructura que proporciona NOS<sup>3</sup>.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-01

**NF-SR-08**

Descripción: La ejecución del banco de pruebas se lleva a cabo mediante el uso de la máquina virtual que proporciona la infraestructura de NOS<sup>3</sup>.

Necesidad: Esencial

Prioridad: Alta

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-01

**NF-SR-09**

Descripción: El simulador del sensor y el del actuador se ejecutarán en consolas diferentes para que su visualización sea más clara.

Necesidad: Conveniente

Prioridad: Baja

Estabilidad: No cambia

Verificabilidad: Alta

Origen: CA-UR-01

### 3.4 Diseño de la solución

En este apartado se estudia el diseño de la solución propuesta para la realización de pruebas. La solución se puede dividir en diversos componentes. En la Figura 3.6 se puede observar la arquitectura de la solución esquematizada por sus componentes. Después, se hará uso de la plantilla de la Figura 3.5 para describir cada uno de éstos. Los atributos presentes en la plantilla son:

**Identificador.** Nombre del componente.

**Rol.** Papel del componente en el sistema.

**Dependencias.** Componentes que dependen de este.

**Descripción.** Explicación del componente.

**Datos.** Valores de entrada y salida del componente.

**Recursos.** Recursos que utiliza el componente.

**Origen.** Requisitos software que dieron lugar al componente.

**Identificador**

Rol:

Dependencias:

Descripción:

Datos:

Recursos:

Origen:

Figura 3.5: Plantilla de componente

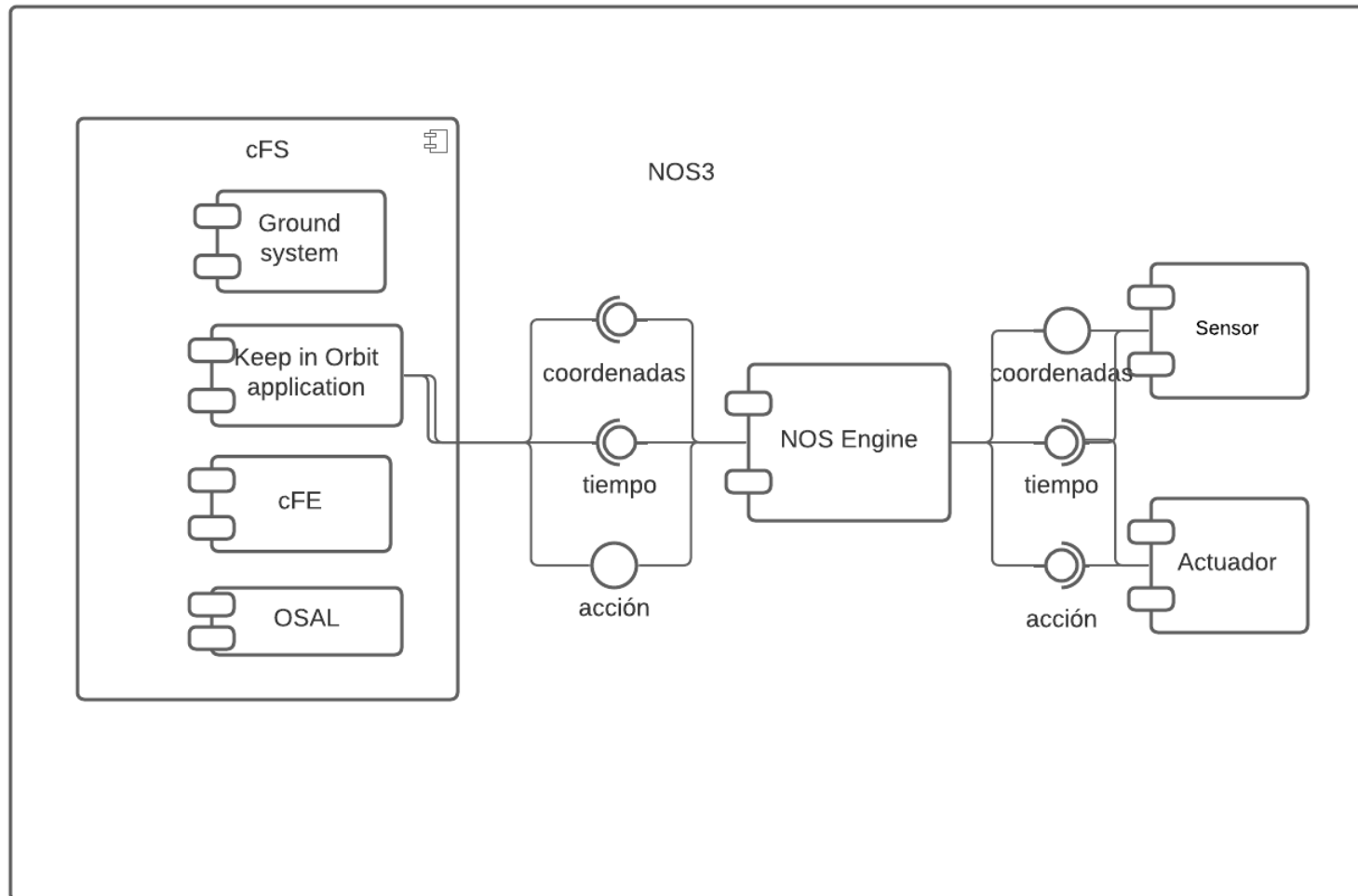


Figura 3.6: Diagrama de componentes

**NOS Engine**

---

Rol: Permite la comunicación entre las aplicaciones del cFS y los simuladores hardware

Dependencias:

Descripción: Simula 30 buses hardware de cada protocolo soportado. Soporta 4 protocolos de transmisión: uart, i2c, spi y can. Las aplicaciones y los simuladores se suscriben a los buses como nodos. Cada nodo debe tener un identificador único en el bus. 2 o más nodos suscritos al mismo bus pueden comunicarse a través de llamadas a la librería del Nos Engine.

Datos: Las coordenadas del satélite, el tiempo, y la acción al actuador

Recursos:

Origen:

**Sensor**

---

Rol: Provee las coordenadas del satélite

Dependencias: NOS Engine

Descripción: Simula un componente hardware de un sensor GPS que lee las coordenadas x, y, z del satélite y las envía de forma periódica a un bus del Nos Engine..

Datos: Coordenadas x, y, z del satélite

Recursos:

Origen:

**Actuador**

---

Rol: Corrige la trayectoria cuando el satélite se desvía

Dependencias: NOS Engine

Descripción: Simula un motor en el satélite. Si el satélite se desvía de la trayectoria preestablecida, el actuador recibe órdenes de un bus del Nos Engine con la dirección y sentido en el que debe actuar para corregir al satélite.

Datos: Dirección y sentido

Recursos:

Origen:

**cFS**

Rol: Software de vuelo que se ejecuta en sistemas empotrados de la nave y contiene un *payload*.

Dependencias:

Descripción: Software de ejecución en tiempo real (crítico). Consiste en una arquitectura por capas que contiene una serie de servicios, aplicaciones y un entorno operativo para ejecutar un RTOS en una nave.

Datos:

Recursos:

Origen:

**Ground system**

Rol: Comunicarse con el satélite desde la Tierra.

Dependencias: cFS

Descripción: Software independiente del cFE. Su función es comunicarse con el satélite desde la Tierra. El cFS actualmente soporta COSMOS y AIT como software de tierra.

Datos:

Recursos:

Origen:

**Aplicación “Keep in Orbit”**

Rol: Comprueba si el satélite se ha desviado de su trayectoria y, en tal caso, envía un mensaje al actuador

Dependencias: cFS

Descripción: Se trata de una de las aplicaciones que se ejecutan en el RTOS. Recibe las coordenadas de un bus del Nos Engine. Las procesa y evalúa si el satélite está en la órbita correcta. Si se ha desviado, envía un mensaje a otro bus del Nos Engine con la dirección y el sentido en el que el motor debe actuar.

Datos:

Recursos:

Origen:



**cFE**

---

Rol:	Proporciona un conjunto de servicios básicos y aplicaciones que sirven de base para el desarrollo. Define una interfaz API para cada uno.
Dependencias:	cFS
Descripción:	Entorno de desarrollo y ejecución de aplicaciones. Provee 5 aplicaciones y una interfaz (API) para cada una. Las 5 aplicaciones son: <i>Executive Services</i> , <i>Event Services</i> , <i>Software Bus</i> , <i>Table Services</i> y <i>Time Services</i> .
Datos:	
Recursos:	
Origen:	

**OSAL**

---

Rol:	Librería software que cuya función es aislar el software de vuelo del sistema operativo en el que se ejecuta
Dependencias:	cFS
Descripción:	Se trata de un proyecto a parte del cFE disponible en la web de NASA. Permite que el software de vuelo (cFS) se ejecute sobre distintos sistemas operativos. Actualmente soporta RTEMS, VxWorks y Linux / x86.
Datos:	Dirección y sentido
Recursos:	
Origen:	

### 3.5 Matrices de trazabilidad

Uno de los retos importantes de cualquier proyecto es garantizar que los requisitos definidos en la fase de análisis están presentes en el diseño y por tanto, se llevan a cabo en la implementación. La matriz de trazabilidad es una herramienta gráfica muy útil para esta labor.

En este capítulo se incluyen 3 matrices de trazabilidad. La primera de ellas agrupa los requisitos de capacidad y los requisitos funcionales. La segunda es entre los requisitos de restricción y los no funcionales. Y la última muestra la trazabilidad entre los componentes y los requisitos funcionales.

	CA-UR-01	CA-UR-02	CA-UR-03	CA-UR-04	CA-UR-05	CA-UR-06	CA-UR-07	CA-UR-08	CA-UR-09	CA-UR-10	CA-UR-11	CA-UR-12
RF-SR-01	•											
RF-SR-02		•										
RF-SR-03			•									
RF-SR-04				•	•	•						
RF-SR-05						•						
RF-SR-06	•											
RF-SR-07			•									
RF-SR-08							•					
RF-SR-09							•					
RF-SR-10								•	•			
RF-SR-11									•			
RF-SR-12			•									
RF-SR-13											•	
RF-SR-14											•	

Tabla 3.1: Trazabilidad entre requisitos funcionales y requisitos de capacidad

	RE-UR-01	RE-UR-02	RE-UR-03	RE-UR-04	RE-UR-05	RE-UR-06	RE-UR-07	RE-UR-08	RE-UR-09	RE-UR-10	RE-UR-11	RE-UR-12	RE-UR-13
NF-SR-01	•	•	•										
NF-SR-02				•									
NF-SR-03					•								
NF-SR-04						•							
NF-SR-05						•							
NF-SR-06						•				•			
NF-SR-07	•												
NF-SR-08	•												
NF-SR-09	•												

Tabla 3.2: Trazabilidad entre requisitos de restricción y requisitos no funcionales

	RF-SR-01	RF-SR-02	RF-SR-03	RF-SR-04	RF-SR-05	RF-SR-06	RF-SR-07	RF-SR-08	RF-SR-09	RF-SR-10	RF-SR-11	RF-SR-12	RF-SR-13	RF-SR-14
cFS														
Ground system														
Keep in Orbit App						•		•	•	•	•			
cFE														
OSAL														
NOS Engine		•					•					•		
Sensor	•		•	•	•									
Actuador	•												•	•

Tabla 3.3: Trazabilidad entre componentes y requisitos funcionales

## Capítulo 4.

# Implementación.

En este capítulo se describen los aspectos más relevantes de la implementación del banco de pruebas. Sin embargo, no se trata de un manual de uso. Para referirse al manual de instalación y uso de software NOS<sup>3</sup>, véase el Apéndice C.

Como se ha mencionado anteriormente, el NOS<sup>3</sup> provee una máquina virtual que cumple una función de contenedor software: contiene las dependencias necesarias para que el software se ejecute sin problemas. La máquina virtual consta de una imagen Ubuntu 16.04 más unos paquetes de software que incluyen AIT, COSMOS, 42 y las librerías del NOS Engine. Este proyecto se ha desarrollado mediante esta máquina virtual.

### 4.1 Simuladores

La implementación de los simuladores del banco de pruebas se encuentra en la ruta /nos3/sims. Un simulador se compone de una clase principal que contiene el método main y una clase de hardware model en el que se encuentran las funciones que simulan el componente hardware.

Los parámetros de configuración más importantes de cada simulador se encuentran en el archivo /nos3/sims/cfg/nos3-simulator.xml. El formato XML es un lenguaje de marcado basado en texto utilizado para estructurar datos de forma lógica. Este archivo contiene una entrada con el tag “<simulator>” por cada simulador de componentes en el sistema. Dentro de cada entrada se especifican parámetros como: el nombre del simulador, la librería que usa para el compilado, las conexiones que utiliza (nombre, tipo y puerto de cada bus) o los parámetros relativos al envío periódico de datos. Este archivo de configuración facilita la labor del programador ya que desde un mismo archivo se pueden editar los parámetros más relevantes de cada simulador.

El banco de pruebas trae por defecto una serie de simuladores. Entre ellos, cabe destacar el simulador “NOS Time Driver”. Éste es el componente que se encarga de suministrar el

tiempo al NOS Engine. El NOS Engine después distribuye el tiempo a todos los componentes a través del bus “command”.

Se han implementado 2 simuladores de componentes hardware: un sensor GPS y un motor.

## Sensor GPS

El sensor GPS se encarga de leer las coordenadas del satélite y enviarlas al NOS Engine. La entrada del sensor en el archivo de configuración se puede observar en la siguiente figura.

```
<simulator>
  <name>sensorgps_sim</name>
  <active>true</active>
  <library>libsensorgps_sim.so</library>
  <hardware-model>
    <type>SENSORGPS</type>
    <connections>
      <connection><type>command</type>
        <bus-name>command</bus-name>
        <node-name>sensorgps-sim-command-node</node-name>
      </connection>
      <connection><type>usart</type>
        <bus-name>usart_18</bus-name>
        <node-port>18</node-port>
      </connection>
    </connections>
    <default-streams>
      <stream>
        <name>sensorgps_stream</name>
        <initial-stream-time>1.0</initial-stream-time>
        <stream-period-ms>3000</stream-period-ms>
      </stream>
    </default-streams>
    <data-provider>
      <type>SAMPLE_42_PROVIDER</type>
    </data-provider>
  </hardware-model>
</simulator>
```

Figura 4.1: Configuración XML del sensor

Se pueden distinguir 2 conexiones diferentes. La primera es al bus “command” para recibir el tiempo mientras que en la segunda conexión se ha elegido el bus de protocolo UART número 18 para enviar los datos al NOS Engine. Los parámetros del envío de datos se encuentran dentro de la etiqueta <stream>: el periodo es de 3 segundos y el envío comienza en el segundo 1.

La estructura del mensaje se muestra en la siguiente tabla.

H	ID	ID	X	X	Y	Y	Z	Z	T
---	----	----	---	---	---	---	---	---	---

Tabla 4.1: Estructura del mensaje enviado por el sensor

Cada celda de la tabla corresponde a 1 byte de tamaño. El tamaño total del mensaje es de 10 bytes. A continuación, se detalla cada campo del mensaje.

Abreviatura	Campo	Tamaño en bits	Combinaciones	Tipo de valor
H	Header	8 bits	256	uint
ID	Identificador de mensaje	16 bits	65536	uint
X	Coordenada X	16 bits	65536	double*
Y	Coordenada Y	16 bits	65536	double*
Z	Coordenada Z	16 bits	65536	double*
T	Trailer	8 bits	256	uint

Tabla 4.2: Descripción de los campos del mensaje enviado por el sensor

El header y el trailer tienen valores fijos (se muestran en la Tabla ) mientras que los demás son variables. Las coordenadas inicialmente son de punto flotante. Una variable de tipo double ocupa 8 bytes de memoria. Para reducir el peso de los mensajes y simplificar el problema se ha acotado el valor de las coordenadas a un intervalo cerrado de 0 a 1. Los valores en punto flotante de las coordenadas se transforman a enteros sin signo a través de una transformación lineal:

$$\text{uint16\_t resultado} = (\text{uint16\_t}) (\text{coordenada} * 32767.0 + 32768)$$

Esta transformación garantiza una precisión de hasta 4 decimales; suficiente para cumplir el requisito. Por lo tanto los valores de las coordenadas soportados por el sistema están incluidos en el intervalo cerrado de 0.0000 a 1.0000.

Campo	Valor (HEX)
Header	0xAA
Trailer	0xFF

Tabla 4.3: Valores del header y el trailer

Para implementar el envío periódico se han seguido los siguientes pasos:

- Se asigna la llamada a una función `send_streaming_data` cada vez que se recibe un mensaje por el bus `command`. Es decir, cada vez que se recibe una actualización de tiempo.
- Dentro de la función `send_streaming_data` se almacena el valor temporal del último envío y, por lo tanto, también del siguiente. Cada vez que se le llama, comprueba si ha transcurrido el periodo necesario para volver a enviar.
- En caso afirmativo, se envía un mensaje por el bus de datos y se actualiza el valor temporal del último envío.

## Motor

La función del motor es corregir las desviaciones de la trayectoria del satélite. Su configuración en el archivo `nos3-simulator.xml` es la siguiente

```
<simulator>
  <name>motor_sim</name>
  <active>true</active>
  <library>libmotor_sim.so</library>
  <hardware-model>
    <type>MOTOR</type>
    <connections>
      <connection><type>command</type>
        <bus-name>command</bus-name>
        <node-name>motor-sim-command-node</node-name>
      </connection>
      <connection><type>usart</type>
        <bus-name>usart_19</bus-name>
        <node-port>19</node-port>
      </connection>
    </connections>
  </hardware-model>
</simulator>
```

Figura 4.2: Configuración XML del motor

Es más simple que la del sensor. Contiene 2 conexiones: una al bus “command” para recibir el tiempo y otra al bus de USART 19 por la que recibirá los mensajes de la aplicación de cFE. Para simplificar el problema, se ha supuesto que el motor solo actúa en los 3 ejes de coordenadas y en ambos sentidos (un total de 6 direcciones diferentes). Aunque se han asignado más bits de los necesarios para proporcionar escalabilidad a más direcciones. La estructura del mensaje que recibe es la siguiente:

H	ID	ID	P	T
---	----	----	---	---

Tabla 4.4: Estructura del mensaje recibido por el motor

Cada celda representa 1 byte de tamaño. El tamaño total del mensaje es de 5 bytes. Los campos del mensaje se describen en la siguiente tabla.

Abreviatura	Campo	Tamaño en bits	Combinaciones	Tipo de valor
H	Header	8 bits	256	uint
ID	Identificador de mensaje	16 bits	65536	uint
P	Payload	8 bits	256	uint
T	Trailer	8 bits	256	uint

Tabla 4.5: Descripción de los campos del mensaje recibido por el motor

El header y trailer tienen valores fijos establecidos por la aplicación de cFE (presentes en la Tabla X) y se comprueban cada vez que se recibe un mensaje. En el payload se codifica la dirección y el sentido en los que el motor debe actuar. Los primeros 4 bits se usan para codificar si el movimiento debe ser positivo o negativo mientras que los 4 bits restantes contienen el eje de dirección. En la tabla X se muestran los posibles valores del mensaje. Tras recibir y decodificar el mensaje, el motor simula su ejecución imprimiendo en su consola el mensaje “Motor Activated on Axis X/Y/Z POSITIVE/NEGATIVE”.

Valor de payload (HEX)	Eje de dirección	Sentido
0x00	X	POSITIVO
0x10	X	NEGATIVO
0x01	Y	POSITIVO
0x11	Y	NEGATIVO
0x02	Z	POSITIVO
0x12	Z	NEGATIVO

Tabla 4.6: Codificación del campo "payload"



<b>Campo</b>	<b>Valor (HEX)</b>
Header	0xBB
Trailer	0xFF

Tabla 4.7: Valores del header y del trailer

## 4.2 Aplicación “Keep in Orbit”

Para el procesamiento de los datos del sensor y la comunicación con los simuladores se ha desarrollado una aplicación en el cFE. El código del software de vuelo (cFE, PSP, OSAL) se encuentra en el directorio /nos3/fsw. Dentro, las aplicaciones se encuentran divididas en 2 carpetas:

- /fsw/apps contiene las aplicaciones que trae el cFE por defecto y
- /fsw/components incluye las aplicaciones específicas para interactuar con la arquitectura NOS. También se encuentra la librería usada para conectarse a los buses simulados en la carpeta hwlib.

La aplicación “Keep in Orbit” se encuentra en /fsw/components

La aplicación consta principalmente de 2 archivos: keep\_in\_orbit\_app.c y keep\_in\_orbit\_device.c. El primero contiene las funciones básicas de una aplicación: la inicialización de la app, la comunicación con el “Ground” y la comunicación con el software bus del cFE (no se confunda con los buses simulados del NOS Engine). A parte de éstas, incluye la inicialización de la clase device.

La clase device de la app incluye las librerías necesarias para comunicarse con el NOS Engine (hwlib) y se ejecuta como un proceso hijo del proceso de la app. En el método de inicialización de la clase, se ha implementado la conexión a 2 buses software: uart 18 para recibir los datos del sensor y uart 19 para enviar datos al motor.

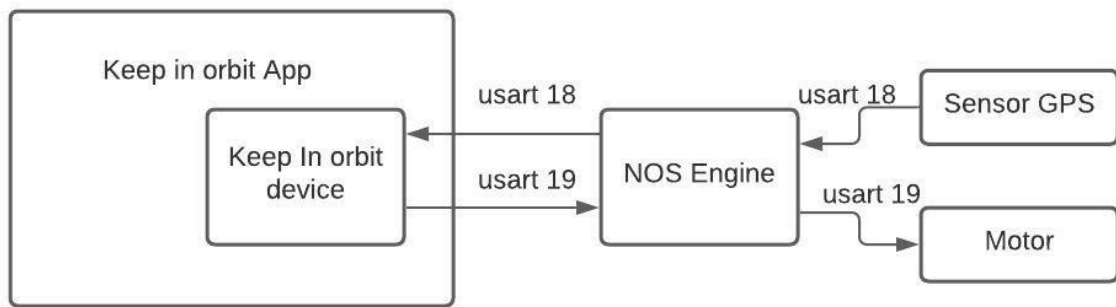


Figura 4.3: Diagrama de comunicación

Dentro de la clase device se encuentra la implementación del procesamiento de datos del sensor y la toma de decisiones. Las coordenadas del sensor GPS son recibidas en formato `uint16_t` (entero sin signo de 2 bytes). Para convertirlas otra vez a formato de punto flotante es necesario calcular la inversa de la transformación que ha realizado el sensor.

$$\text{double coordenada} = (\text{double}) (\text{resultado} - 32768) / 32767.0$$

Tras haber preprocesado las coordenadas, la aplicación comprueba si las coordenadas siguen la trayectoria establecida. Para simplificar el problema, se ha definido una trayectoria del satélite muy sencilla: las 3 coordenadas deben tener el mismo valor ( $x = y = z$ ). La app comprueba si alguna coordenada es distinta, y entonces prepara un mensaje para enviarlo por el bus `usart 19`. El algoritmo de la app se muestra en la Figura 4.4.

**Data:** valores de coordenadas x, y, z

**Result:** mensaje enviado al motor

**while** mensajeRecibido **do**:

```
    if mensajeLibreDeErrores y headerCorrecto:
        preprocesarCoordenadas();
        for coordenada:
            if coordenada != otras 2 coordenadas:
                if coordenada < otra coordenada:
                    mensaje = [coordenada, positivo]
                else:
                    mensaje = [coordenada, negativo]
                end
            enviarMensaje;
        end
    end
    else:
        enviarErrorAlSistema;
    end
end
```

Figura 4.4: Algoritmo de la aplicación

## Capítulo 5.

# Pruebas y evaluación

En este capítulo se exponen las pruebas realizadas sobre el sistema para verificar que su funcionamiento es correcto. En la sección 5.2 se evalúa el rendimiento del sistema en la realización de las pruebas.

## 5.1 Pruebas

### Entorno de prueba

El entorno utilizado en este proceso puede condicionar los resultados. Por ello, en la siguiente lista se describe los componentes tanto hardware como software en los que esta serie de pruebas se han llevado a cabo.

- GNU/Linux - Ubuntu 18.04 LTS
- Intel i7-6700K 4.0GHz
- 2 x 8GB RAM DDR4
- SSD Samsung 850 Evo 500GB

### Definición de pruebas

Las pruebas se han definido con la ayuda de una plantilla expuesta en la Figura 5.1. La plantilla se compone de los siguientes atributos:

**Identificador.** Será CP-XX donde XX es el número que identifica la prueba. El identificador comienza en CP-01.

**Descripción.** Descripción del caso de prueba.

**Entrada.** Datos necesarios para la realización de la prueba.

**Salida.** Resultado esperado de la prueba.

**Origen.** Requisitos software que se verifican con la prueba.

**Identificador**

---

Descripción:

Entrada:

Salida:

Origen:

Entorno:            n/a

Figura 5.1: Plantilla de casos de pruebas

**CP-01**

**Descripción:** El sensor lee las coordenadas del satélite y las envía al software bus. El sensor realiza un envío periódico cada 3s.

**Entrada:** Coordenadas x, y, z del nanosatélite

**Salida:** Se ha enviado un mensaje al software bus

**Origen:** F-SR-01, F-SR-02, F-SR-03, F-SR-04, F-SR-05

**Entorno:** n/a

**CP-03**

**Descripción:** La aplicación recibe un mensaje incorrecto del software bus. El mensaje contiene un header diferente al enviado por el sensor.

**Entrada:** Mensaje en el software bus con un header incorrecto

**Salida:** La aplicación ha identificado el error, lo ha reportado al sistema mediante el servicio de eventos y no ha procesado el mensaje

**Origen:** F-SR-06, F-SR-07, F-SR-08

**Entorno:** n/a

**CP-02**

**Descripción:** La aplicación recibe el mensaje del software bus y extrae los datos (coordenadas e identificador)..

**Entrada:** Mensaje en el software bus

**Salida:** La aplicación ha recibido correctamente el mensaje y ha extraído los datos

**Origen:** F-SR-06, F-SR-07

**Entorno:** n/a

**CP-04**

**Descripción:** La aplicación envía un mensaje al software bus. El motor lee el mensaje del bus y simula que actúa.

**Entrada:** La aplicación envía un mensaje que contiene una dirección y un sentido.

**Salida:** El motor ha recibido correctamente el mensaje, ha extraído los valores de dirección y sentido y ha simulado que actúa.

**Origen:** F-SR-01, F-SR-06, F-SR-07, F-SR-10, F-SR-11, F-SR-12, F-SR-14

Entorno: n/a

**CP-05**

**Descripción:** El motor lee un mensaje con fallos del software bus.

**Entrada:** Un mensaje con header erróneo en el software bus

**Salida:** El motor ha identificado el error, lo ha reportado al sistema mediante el servicio de eventos y no ha procesado el mensaje

**Origen:** F-SR-01, F-SR-12, F-SR-13

**Entorno:** n/a

**CP-06**

**Descripción:** El sensor lee las coordenadas del satélite y envía un mensaje cada 3 segundos. La aplicación lee las coordenadas del software bus. Identifica qué coordenada no sigue la trayectoria establecida. Y envía un mensaje al motor para que corrija dicha coordenada.

**Entrada:** Coordenadas del satélite desviadas

Mensaje en el software bus con una coordenada desviada

**Salida:** El sensor ha enviado el mensaje al bus. La aplicación ha leído y procesado el mensaje. Ha identificado la coordenada desviada y ha enviado un mensaje al software bus. El motor ha leído dicho mensaje y ha actuado con el fin de corregir la trayectoria

**Origen:** F-SR-01, F-SR-02, F-SR-03, F-SR-04, F-SR-05, F-SR-06, F-SR-07, F-SR-09, F-SR-10, F-SR-11, F-SR-12, F-SR-14

**Entorno:** n/a

## 5.2 Matriz de trazabilidad

	RF-SR-01	RF-SR-02	RF-SR-03	RF-SR-04	RF-SR-05	RF-SR-06	RF-SR-07	RF-SR-08	RF-SR-09	RF-SR-10	RF-SR-11	RF-SR-12	RF-SR-13	RF-SR-14
CP-01	•	•	•	•	•									
CP-02						•	•							
CP-03						•	•	•				•		
CP-04	•					•	•			•	•	•		•
CP-05	•		•	•	•				•			•	•	
CP-06	•	•				•	•			•	•			•

Tabla 5.1: Trazabilidad entre casos de prueba y requisitos funcionales

## 5.3 Evaluación

La verificación de las pruebas realizadas ha consistido en recrear las condiciones descritas en cada prueba, configurar los parámetros de entrada, ejecutar el banco de pruebas y observar el resultado de la ejecución. Si el resultado coincide con la salida esperada la prueba se da por validada. Los 6 casos de prueba han sido validados satisfactoriamente.

La evaluación del rendimiento no aplica al problema ya que las tareas realizadas no contienen una carga de computación alta. Sin embargo, al tratarse de un problema de ejecución en tiempo real crítico, el tiempo de ejecución sí resulta una métrica importante. Se ha realizado un total de 10 pruebas. En cada una se ha tomado 2 muestras: el tiempo en el que el sensor lee las coordenadas del satélite y el tiempo en el que el actuador se ejecuta. El tiempo de respuesta del sistema completo es la diferencia entre estos dos.

Los resultados (expuestos en la tabla 5.2) realmente no son aplicables porque las pruebas se han realizado con un hardware bastante superior al del satélite.



Prueba	Tiempo sensor (s)	Tiempo actuador (s)	Tiempo respuesta (s)
1	46.743513	46.736476	0.007037
2	37.119745	37.118783	0.000962
3	00.339932	00.338873	0.001059
4	06.612991	06.611974	0.001017
5	54.060415	54.059480	0.000935
6	14.297428	14.296374	0.001054
7	24.202627	24.202028	0.000599
8	34.080619	34.079877	0.000742
9	45.780461	45.774946	0.005515
10	19.917324	19.912992	0.004332
<b>Tiempo de respuesta medio</b>			<b>0.0023252 s</b>

Tabla 5.2: Evaluación del tiempo de respuesta del sistema

## Capítulo 6.

# Plan de proyecto

En el presente capítulo se muestran las tareas que han formado el desarrollo del proyecto y su planificación en el tiempo a través de un gráfico GANTT. En la sección 5.3 se calcula y se desglosa el presupuesto requerido para llevar a cabo este proyecto.

### 6.1. Tareas del proyecto

El proyecto se divide en las siguientes tareas:

**Presentación del proyecto.** Descripción de la Cátedra Uc3m-SENER por Javier Fernández Muñoz. Se explica el trabajo hecho en años anteriores y los objetivos de este año.

**Inicio.** Se asignan los 3 Trabajos de Fin de Grado entre los 3 alumnos dentro de la cátedra.

**Planificación.** Consiste en la lectura de la documentación proporcionada por el tutor y la planificación de las tareas necesarias para llevar a cabo el proyecto. Se divide en las siguientes subtareas:

**Lectura de documentación.** Primera lectura de documentación para obtener un contexto teórico. La documentación proporcionada por el tutor trata de la cátedra misma, del cFE y del banco de pruebas NOS<sup>3</sup>.

**Planificación del proyecto.** Definición de tareas a realizar y primera planificación temporal.

**Análisis y diseño.** Análisis de viabilidad y de requisitos. Consta de las siguientes subtareas:

**Definición de requisitos.** Especificación de los requisitos de usuario. Extracción de los requisitos de sistema a partir de los anteriores. Y descripción de los casos de uso.

**Diseño del sistema.** Diseño por componentes de la arquitectura del sistema a desarrollar.

**Implementación.** Desarrollo del sistema diseñado siguiendo los requisitos ya definidos. Se divide en:

**NOS3.** Desarrollo de la infraestructura que trae el banco de pruebas por defecto.

**Sensor.** Desarrollo de un simulador de un sensor. Implementación de la función relativa al envío periódico de datos obtenidos del sensor.

**Actuador.** Desarrollo de un simulador de un componente actuador. Implementación del comportamiento del actuador tras haber recibido un mensaje.

**Aplicación de cFE.** Desarrollo de una aplicación de cFE capaz de interactuar con ambos simuladores: recibir, procesar y enviar mensajes.

**Pruebas y evaluación.** Definición e implementación de pruebas para comprobar el correcto funcionamiento del software. Contiene:

**Pruebas.** Conjunto de pruebas que se llevan a cabo sobre el software implementado.

**Evaluación.** Evaluación de los resultados obtenidos en las pruebas.

**Documentación.** Desarrollo de la memoria y de la presentación del proyecto.

**Memoria.** Redacción del presente documento.

**Presentación.** Creación de diapositivas para la presentación final del proyecto ante el tribunal.

## 6.2 Planificación

A partir de la definición de tareas de la sección anterior se planifica el desarrollo del proyecto. La planificación se ha llevado a cabo a través de un diagrama de Gantt. Los diagramas de Gantt permiten visualizar de una manera gráfica el orden cronológico y el tiempo de dedicación previsto para cada tarea del proyecto.

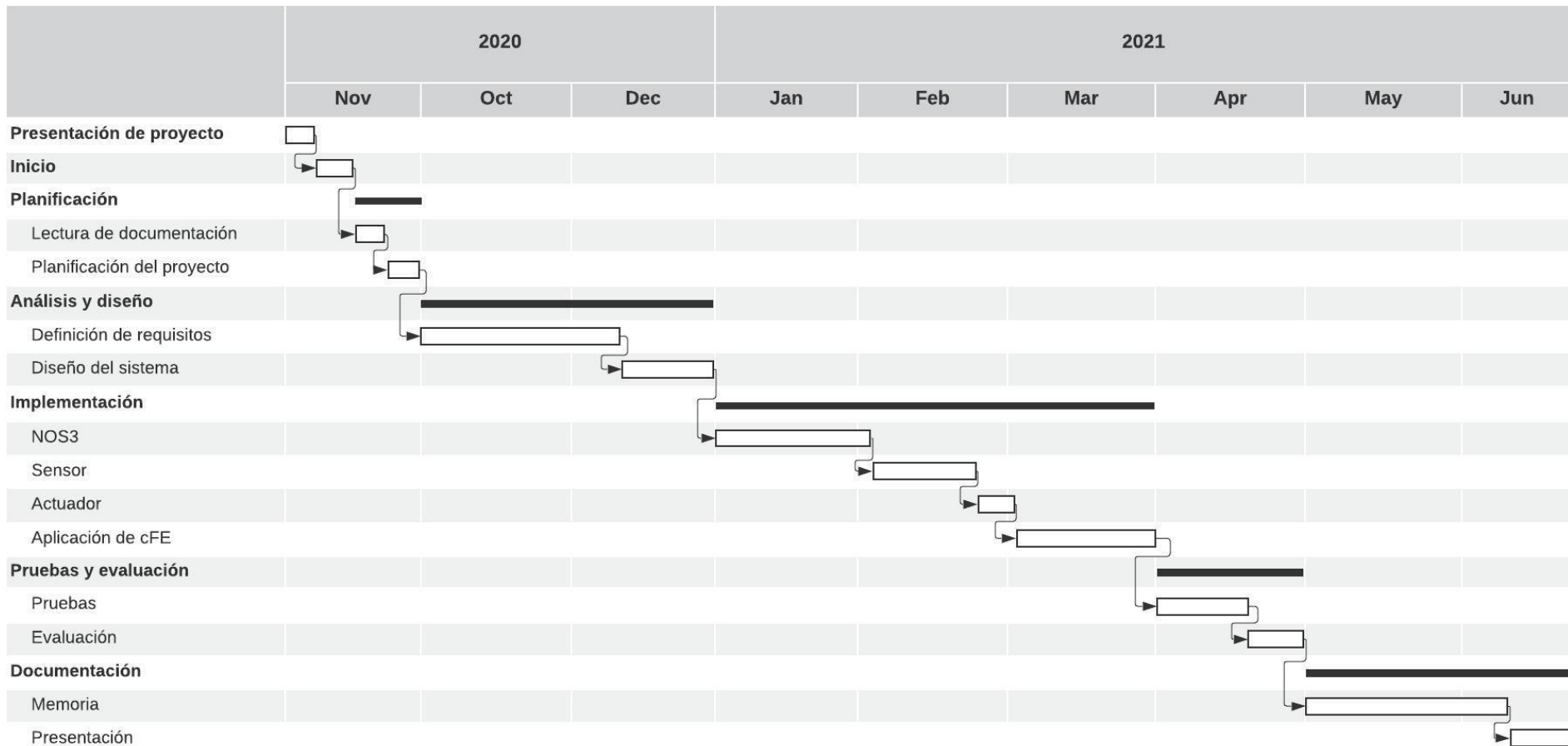


Figura 6.1: Diagrama Gantt

## 6.3 Presupuesto del proyecto

En esta sección se cuantifica el coste total del proyecto en base a su duración y a los recursos usados. El coste total se desglosa en: coste de personal, coste de material, y costes indirectos. Los costes subtotales no incluyen su I.V.A correspondiente. En el coste final se añadirá el 21% de Impuesto de Valor Añadido.

### Costes de personal

En la tabla siguiente se expone el número de horas empleadas en cada fase.

Tarea	Horas
Planificación	15 h
Análisis y diseño	85 h
Implementación	130 h
Pruebas y evaluación	20 h
Documentación	100 h
Total	350 h

Tabla 6.1: Distribución de las horas del proyecto

A continuación, se calculará los costes de personal en función al tipo de profesional empleado y al número de horas que ha trabajado.

Cargo	Horas	Coste por hora	Total
Analista	40 h	45.00 €	1800.00 €
Analista/Programador	125 h	28.00 €	3500.00 €
Programador	185 h	17.00 €	3145.00 €
Total	350 h		8445.00 €

Tabla 6.2: Costes de personal

### Costes de material

Este proyecto se ha desarrollado usando software de código abierto y gratuito, la mayoría proporcionado por la empresa NASA. Sin embargo, el hardware que se ha usado para llevar a cabo el proyecto sí tiene un coste asociado. Aunque no se cargará la totalidad del coste del hardware, sino una parte proporcional de su vida útil (lo que se denomina como amortización).

En la siguiente tabla se exponen los componentes hardware, su coste total y su amortización para este proyecto.

Producto	Coste total	% Uso	Tiempo usado	Vida útil	Amortización
Procesador Intel i7-6700K	339.00 €	100 %	9 meses	60 meses	50.85 €
Placa base Gigabyte GA-Z170-HD3P	126.00 €	100 %	9 meses	60 meses	18.90 €
Memoria RAM G.Skill Ripjaws V Red DDR4 2x8GB	128.00 €	100 %	9 meses	60 meses	19.20 €
SSD Samsung 850 Evo SSD Series 500GB	164.00 €	100 %	9 meses	60 meses	24.60 €
Fuente de alimentación Tacens Mars Gaming 800W	53.50 €	100 %	9 meses	60 meses	8.03 €
Tarjeta gráfica Asus Dual GTX 1060 OC 6GB	309.00 €	100 %	9 meses	60 meses	46.35 €
Monitor LG 29UM68-P 29"	289.00 €	100 %	9 meses	60 meses	43.35 €
Teclado Tacens Mars MK10	12.95 €	100 %	9 meses	60 meses	1.94 €
Ratón Logitech M170	9.99 €	100 %	9 meses	60 meses	1.49 €
Total					214.71 €

Tabla 6.3: Costes de material

### Costes indirectos

Los costes indirectos no se pueden asignar directamente a la producción del sistema, aunque son imprescindibles: sin ellos no se podría haber llevado a cabo este proyecto. Se encuentran en esta categoría gastos como el del internet, que consiste en una tarifa plana mensual de 38.00 €; dividida a partes iguales entre los 3 inquilinos del inmueble daría un coste mensual de 12.67 €. Para el computo del gasto energético se ha supuesto un coste de 0.45 € por KWh y un consumo medio de 400W por hora de la fuente de alimentación. Y en el material académico se ha incluido la impresión de un manual oficial de NOS<sup>3</sup>, bolígrafos y folios.

Recurso	Coste
Electricidad	25.00 €
Internet	114.00 €
Material académico	45.00 €
<b>Total</b>	<b>184.00 €</b>

Tabla 6.4: Costes indirectos

### Coste total

Tras haber considerado todos los costes obtenidos en las secciones anteriores, se puede hacer un cómputo total del coste del proyecto. Es necesario incluir un margen del 10% para subsanar posibles imprevistos (fallos de hardware, bajas médicas) así como un margen de rentabilidad de un 25% de donde se obtiene el beneficio.

Como se ha mencionado en secciones anteriores, el coste computado hasta el momento es libre de impuestos. Según la ley 37/1992 publicada en el “BOE” número 312 sobre el Impuesto de Valor Añadido, un porcentaje del 21% debe ser añadido como impuesto para obtener el precio final del cliente.

Tipo de coste	Valor
Costes de personal	8445.00 €
Costes de material	214.71 €
Costes indirectos	184.00 €
Margen de imprevistos (10%)	884.37 €
Margen de beneficio (25%)	2432.02 €
Impuesto de Valor Añadido (21%)	2553.62 €
<b>Total</b>	<b>14713.72 €</b>

Tabla 6.5: Coste total

Por lo tanto, el precio final de la solución con el impuesto del 21% incluido asciende a 14713.72 €, (**CATORCEMIL SETECIENTOS TRECE EUROS Y 72 CÉNTIMOS**).

## Capítulo 7.

# Marco regulador y entorno socio-económico

En este capítulo se describe el entorno socio-económico que en el que se sitúa el proyecto y se exponen las leyes y estándares que se han tenido en cuenta para la realización del mismo.

## 7.1 Marco regulador

Hoy en día, la presencia humana en el espacio es cada vez más frecuente. Que los conflictos entre humanos aparezcan allí no sería de extrañar, tal y como pasa en la Tierra. Es por ello que existe la necesidad de un marco legal que regule los derechos en el espacio extraterrestre.

### **Derecho espacial**

La Oficina para los Asuntos del Espacio Exterior (UNOOSA en adelante) es una organización de las Naciones Unidas que se encarga de implementar políticas relacionadas con el espacio y de promover la cooperatividad a nivel internacional en el uso del espacio exterior. La UNOOSA define el derecho espacial como “el cuerpo normativo que rige las actividades relacionadas con el Espacio. Comprende una variedad de acuerdos internacionales, tratados, convenciones y resoluciones de la Asamblea General de las Naciones Unidas, así como normas y reglamentos de las organizaciones internacionales”

En definitiva, se entiende por derecho espacial a una serie de normas y leyes internacionales, así como de la legislación de cada Estado, que se aplican en el espacio ultraterrestre.

### **Normativa general por las Naciones Unidas**



Tras en inicio de la carrera aeroespacial en los años 60, apareció la necesidad de racionalizar la actividad espacial. Con este fin, en 1959 la Asamblea General de las Naciones Unidas creó la COPUOS o Comisión sobre la Utilización del Espacio Ultraterrestre con Fines Pacíficos. Este organismo ha desarrollado cinco tratados generales sobre los principios jurídicos que deben regir las actividades de los Estados en la exploración y utilización del espacio [10]. Entre los 5, los más relevantes para este proyecto son los siguientes:

El *Tratado sobre los principios que deben regir las actividades de los Estados en la exploración y utilización del espacio ultraterrestre, incluso la Luna y otros cuerpos celestes* [11]. Puede considerarse la base jurídica general para la utilización del espacio ultraterrestre con fines pacíficos. Se puede decir que los otros cuatro tratados tratan específicamente de ciertos conceptos incluidos en el Tratado de 1967 [10]. Los puntos más importantes del tratado son:

- La exploración espacial está abierta a todos los países en igualdad de condiciones y deberá hacerse en provecho e interés de todos los países.
- Los estados no pueden apropiarse del espacio extraterrestre, la Luna y otros cuerpos celestes.
- Los estados deberán realizar su exploración espacial asegurando el mantenimiento de la paz y la seguridad y fomentando la cooperación y la comprensión.
- Queda prohibido realizar ensayos con cualquier tipo de armas en los cuerpos celestes.
- Los estados se declaran responsables de las actividades que realicen en el espacio ultraterrestre y de los daños producidos por el lanzamiento de un objeto al espacio.

El *Convenio sobre la responsabilidad internacional por daños causados por objetos espaciales* [12]. En el se desarrollan aspectos ya introducidos en el primer tratado con todo tipo de detalles. Se enfoca las responsabilidades que asumen los Estados al lanzar un objeto al espacio: daños a personas físicas o jurídicas y daños a otros Estados.

El *Convenio sobre el registro de objetos lanzados al espacio ultraterrestre* [13]. Este convenio impone a los Estados que deseen lanzar un objeto al espacio ultraterrestre la obligación de registrarlo en un registro estatal propio. En el caso de España, tenemos el Registro Español de Objetos Espaciales Lanzados al Espacio Ultraterrestre, creado por el Real Decreto 278/1995 de 24 de febrero [14]. También establece un registro internacional de libre acceso de objetos lanzados al espacio. Cada vez que un Estado desee lanzar un satélite u otro objeto debe comunicarse con las NNUU para inscribir el objeto en el registro. Cada entrada debe contener una serie de atributos (véase la Tabla 7.1).

Nombre del Estado de lanzamiento
Designación del objeto espacial o número de registro
Fecha y lugar de lanzamiento
Parámetros orbitales básicos
Función general del objeto

Tabla 7.1: Atributos de registros

En los años posteriores, las NNUU aprobaron 5 resoluciones de la Asamblea General. La más relevante para este proyecto es la segunda: *Principios que han de regir la utilización por los Estados de satélites artificiales de la Tierra para las transmisiones internacionales directas por televisión*. En ella se exponen los que deben seguir los Estados y las responsabilidades que asumen en las transmisiones por televisión: cooperación internacional, derecho y deber de consulta, arreglo pacífico de controversias, deber de notificación a las NNUU.

## Estándares europeos

En 1973, la fusión de 2 organizaciones dio lugar a la fundación de la Agencia Espacial Europea (ESA). Pronto los ingenieros de la ESA se percataron de la necesidad de estándares y requisitos formales para gestionar de una manera eficiente los proyectos espaciales.

La Cooperación Europea para la Normalización del Espacio (ECSS) es una colaboración establecida en 1993 entre la ESA, la industria espacial europea y varias agencias espaciales para mantener un único conjunto de estándares. Posee un total de 121 estándares activos divididos en 4 ramas: gestión de proyecto, aseguramiento de la calidad, ingeniería de sistemas y desarrollo de software sostenible.

El estándar ECSS-E-ST-10-03C – “Testing” [15] está relacionado directamente a este proyecto. Éste contiene, entre otros:

- Requerimientos de un programa de pruebas.
- Requerimientos de una gestión correcta de las pruebas.
- Requerimientos para pruebas de redundancia.
- Requerimientos para pruebas ambientales.
- Requerimientos para pruebas funcionales y de rendimiento.
- Requerimientos de las condiciones, tolerancias y precisión de las pruebas.
- Requerimientos de las pruebas antes del lanzamiento (*pre-launch*).

- Lista de pruebas de todo tipo y sus aplicaciones.

## 7.2 Entorno socio-económico.

Desde sus inicios hasta la actualidad, el sector aeroespacial ha crecido considerablemente. Cada año son más los países que aumentan su presupuesto para este sector y que desean tener una mayor presencia en el espacio. En los años 60 fueron EEUU y la URSS fueron los que más apostaron para convertirse en la potencia mundial. En la actualidad no se trata tanto sobre una carrera entre países, sino entre agencias espaciales. Las más importantes son la NASA y la ESA. La segunda integra la mayoría de estados europeos.

Cabe destacar la aparición de empresas privadas en el sector y en concreto la de Elon Musk: SpaceX [16]. Fundada en 2002 con el objetivo de reducir los costes del acceso al espacio y de hacer que los viajes espaciales sean accesibles a “casi cualquiera” [17], no ha dejado de lograr hitos. Por ejemplo:

- El primer cohete de financiación privada puesto en órbita (2008).
- Primera empresa privada en poner en órbita y recuperar una nave espacial (2010).
- Primera empresa privada en poner en órbita una cápsula apta para humanos. (2019)
- Primera empresa privada que ha llevado a humanos a la Estación Espacial Internacional (2020).

Esta empresa ha logrado reducir significativamente los precios de lanzamiento de satélites geoestacionarios y, en consecuencia, ha ejercido presión sobre el mercado para que la competencia reduzca sus precios. Antes de 2013, la empresa francesa *Arianespace* era líder en el mercado con un precio de 56 millones de dólares estadounidenses por lanzamiento a órbitas bajas. SpaceX indicó en 2014 que, si logran alcanzar sus objetivos, ofrecerían un precio de lanzamiento de un modelo Falcon 9 reusable de entre 5 y 7 millones de dólares [18], lo que supone una diferencia con el precio de *Arianespace* de más del 90%.

### Impacto económico

La aparición de los nanosatélites ha conseguido reducir el precio de un satélite del orden de millones de dólares hasta una cifra menor incluso que 500.000 dólares. Aún así, tan sólo un pequeño error de diseño o implementación puede echar a perder el presupuesto y todo el tiempo por los ingenieros. Para evitarlo es indispensable planificar e implementar un proceso de pruebas al software durante todas las fases de implementación.

Por este motivo, el impacto económico que tiene este proyecto es muy grande. Todos los requisitos del sistema deben ser probados y verificados más de una vez antes del lanzamiento. Este proyecto implementa un banco de pruebas para probar el software de vuelo usado en el satélite de la cátedra: el cFS. La arquitectura del banco de pruebas (NOS<sup>3</sup>) provee una

simulación de los componentes hardware del satélite y de las condiciones en las que se debe ejecutar para recrear el entorno en el que se ejecutará el software de vuelo del satélite.

## Capítulo 8.

# Conclusiones y trabajos futuros

Este es el último capítulo del cuerpo del documento. En él se exponen las conclusiones obtenidas a partir de la realización del Proyecto de Fin de Grado. Este capítulo está dividido en 3 partes. En primer lugar, se comprueba que los objetivos establecidos de la Sección 1.2 se han cumplido. Seguidamente, se exponen las conclusiones obtenidas, tanto generales como personales. Por último, en la Sección 8.3 se muestra un conjunto de tareas futuras que se pueden llevar a cabo para profundizar sobre este trabajo.

## 8.1 Objetivos

Se verifica que los 4 objetivos definidos en la Sección 1.2 se cumplen.

- Se ha creado un entorno de simulación con la arquitectura del NOS<sup>3</sup>. El NOS<sup>3</sup> trae por defecto una versión del cFS como software de vuelo. Se ha conseguido hacer una ejecución inicial del NOS<sup>3</sup> junto a la versión del cFS que trae.
- Se han implementado satisfactoriamente 2 simuladores de componentes hardware en el NOS<sup>3</sup>. El primero simula el comportamiento de un sensor GPS mientras que el segundo simula un motor. Este proceso ha permitido verificar que los simuladores funcionan correctamente.
- Se ha implementado una nueva aplicación en el cFS que hace uso de las librerías de NOS<sup>3</sup> para lograr intercambiar mensajes con los simuladores de componentes.
- Por último, se ha llevado a cabo una fase de verificación del sistema completo. Se ha diseñado un escenario en el que la aplicación tiene que procesar los mensajes recibidos del sensor y comunicarse con el actuador si es necesario. Las pruebas han sido satisfactorias y todos los requisitos han sido verificados.

## 8.2 Conclusiones del proyecto

### Generales

Este Trabajo de Fin de Grado ha significado un gran reto. Se ha requerido de una planificación para distribuir la carga en 8 meses que ha durado y una dedicación y constancia durante todo el proceso. Ha habido una primera parte de investigación entre todos los participantes de la beca. Hemos realizado varias reuniones en las que nuestro tutor, Javier Fernández Muñoz, nos ha puesto al tanto del proyecto general de la cátedra y de las tecnologías que usa (principalmente cFE y RTEMS). Después se ha asignado un proyecto a cada uno y el trabajo ha pasado a ser individual.

Este proyecto me ha permitido aprender cómo funciona un banco de pruebas y en general una arquitectura de software compleja (una estructura de ficheros grande, muchas librerías, muchos *Makefile*, y aplicaciones escritas en diferentes lenguajes...). Gracias a él he aprendido a desarrollar software de una manera sistemática. Es decir, en vez de ponerse a programar directamente, seguir el proceso de desarrollo de software:

- Extraer una serie de requisitos de usuario que se deben cumplir.
- Hacer un análisis de los requisitos y obtener requisitos del sistema: menos generales y más concisos.
- Diseñar la solución con ayuda de diagramas.
- Implementar la solución.
- Y final realizar una serie de pruebas para verificar que el sistema funciona correctamente y los requisitos se cumplen.

Han surgido varios problemas y obstáculos durante todo el proyecto. La mayor parte durante la implementación de los simuladores porque la información online sobre el NOS<sup>3</sup> es escasa. Hay una guía para desarrolladores publicada por la NASA que ofrece una idea general de cómo funciona la arquitectura. Pero no ofrece casi ejemplos y ningún desarrollador ha publicado en Internet un tutorial o ejemplo de cómo usar el banco de pruebas. La parte de implementación de la aplicación del cFE ha sido más sencilla ya que sí hay más material publicado.

### Personales

En general la experiencia de realizar este Trabajo de Fin de Grado ha sido muy grata. Nunca antes había afrontado un proyecto de este tamaño y he aprendido varias lecciones de él.

He aprendido a poner en práctica los contenidos teóricos aprendidos durante la asignatura de *Sistemas en Tiempo Real*. Han sido necesarios para implementar tanto la parte de los simuladores como la de la aplicación del software de vuelo y ha sido gratificante ver el

resultado final: cómo las piezas funcionan correctamente, se comunican entre sí y forman un sistema mayor.

Han surgido varios obstáculos y adversidades durante el proyecto que he aprendido a afrontar con esfuerzo personal y enfocando el problema de diferentes ángulos. Varios problemas me han sobrepasado y he recurrido a mi tutor Javier Fernández Muñoz que me ha ayudado de manera incondicional y entre los dos hemos conseguido sobrepasarlos.

He aprendido a documentar de una manera completa un proyecto de desarrollo de software. Para lograrlo, he utilizado el programa Microsoft Word de manera profesional y pienso que es una aptitud que me va a servir en un futuro.

Por último, este proyecto me ha permitido conocer un poco de un sector que siempre me había llamado la atención pero nunca había hecho nada relacionado con él: el sector aeroespacial. Sin duda me alegro de haber participado en la cátedra y me llevo una gran satisfacción de haber aportado, aunque haya sido una diminuta parte, a un proyecto tan increíble como es el de poner en órbita un nanosatélite.

### 8.3 Trabajos futuros

Este proyecto se puede ampliar de varias formas. Algunas de ellas se exponen a continuación:

**Simular el comportamiento de los componentes reales del satélite.** A través del NOS<sup>3</sup> es muy sencillo implementar más simuladores. Se podría implementar todos los que componen al nanosatélite para conseguir una simulación realista.

**Juntar hardware real y simulado para la ejecución.** El NOS<sup>3</sup> permite conectar hardware real. El hardware real y los componentes simulados hacen uso de un cliente NOS para comunicarse con el NOS<sup>3</sup> a través del protocolo TCP/IP. Esto permite alternar de una manera sencilla hardware real y simulado.

## Apéndice A.

# Summary

## A.1 Introduction

This end-of-degree project has been possible thanks to the collaboration between the SENER company and the University Carlos III of Madrid. After a long and fruitful link of collaboration in research and teaching tasks, in 2018 the UC3M-SENER Aerospace Chair was created, a new framework in where to continue and strengthen the relationship between both companies. This chair brings together students from multiple engineering disciplines: aerospace, computer science, telecommunications, etc. And it aims to establish the bases of collaboration in research tasks related to the development of a nanosatellite.

Since the launch of the first satellite into space, the trend has been to build and launch increasingly complex, large and expensive satellites. The space was reserved for large corporations dependent on the government and military. It is what is called the *Old Space*. But the current landscape has changed a lot. The new wave of colonization is being carried out by small, cheap and fast systems that favor those companies that need the space to achieve their objectives and expand their services. This philosophy is what we call *New Space*.

Nanosatellites are the latest trend at New Space. Small, with a mass of between 1 and 10kg, they offer great precision. In addition, thanks to advances in microelectronics and the CubeSat standard, its manufacture allows mass production and low cost.

Although these advances have considerably reduced the price of building and putting a satellite into orbit, the difficulty of such an operation or the possible risks (such as not entering the correct orbit or colliding with another star) are still present. Multiple tests and simulation tools need to be applied to ensure that everything goes as expected..



## A.2 Objectives

The objective of the project is the design and implementation of a test bed through the NOS<sup>3</sup> platform for the development of software for nanosatellites in the cFE environment, both developed by NASA. In turn, this objective can be broken down into different goals or subgoals.

The objectives to be achieved with the development of this project are listed below:

- Create the simulation environment through the infrastructure provided by NOS<sup>3</sup> and run cFS on it.
- Implementation of 2 new component simulators in the NOS<sup>3</sup> infrastructure.
- Development of a cFS application that interacts with both simulators..
- Check that the flight software runs correctly in the simulation environment

## A.3 NOS<sup>3</sup>

The NASA Operational Simulator for Small Satellites (NOS<sup>3</sup>) simulator is an open source software benchmark for nanosatellites. NOS<sup>3</sup> is a collection of Linux executables and libraries. Can be run with or without having any physical hardware attached.

The NOS<sup>3</sup> runs on a Linux virtual machine. This virtual machine plays a role of a software container to ensure that all dependencies are met before running the program. These components are listed in the following table.

Component	Description
Oracle VirtualBox	Oracle VirtualBox is virtual machine virtualization software.
Vagrant	Vagrant is a tool for creating and configuring VirtualBox virtual machines. Your options include installing packages, creating users, manipulating files and directories, and more.
NOS Engine	The NOS Engine is a software component that is responsible for simulating hardware buses. Allows flight software and simulated hardware components to communicate each other.
Simulated hardware components	A collection of simulated hardware components that connect to the NOS engine and provide hardware input and output to the flight software.
42	Some of the hardware components require data from the environmental environment. 42 is an open source spacecraft altitude and orbit visualization and simulation tool developed by

	NASA. It is used to provide environmental data to hardware components.
cFS	The cFS is included in the NOS <sup>3</sup> . It is used as the basis for developing the mission software.
COSMOS	COSMOS is an open source ground system developed by Ball Aerospace. Provides communication and remote control of flight software.
CFC	The CFC (COSMOS File Creator) is a tool that allows the generation of remote control and telemetry files.

Table A.1: NOS<sup>3</sup> dependency list

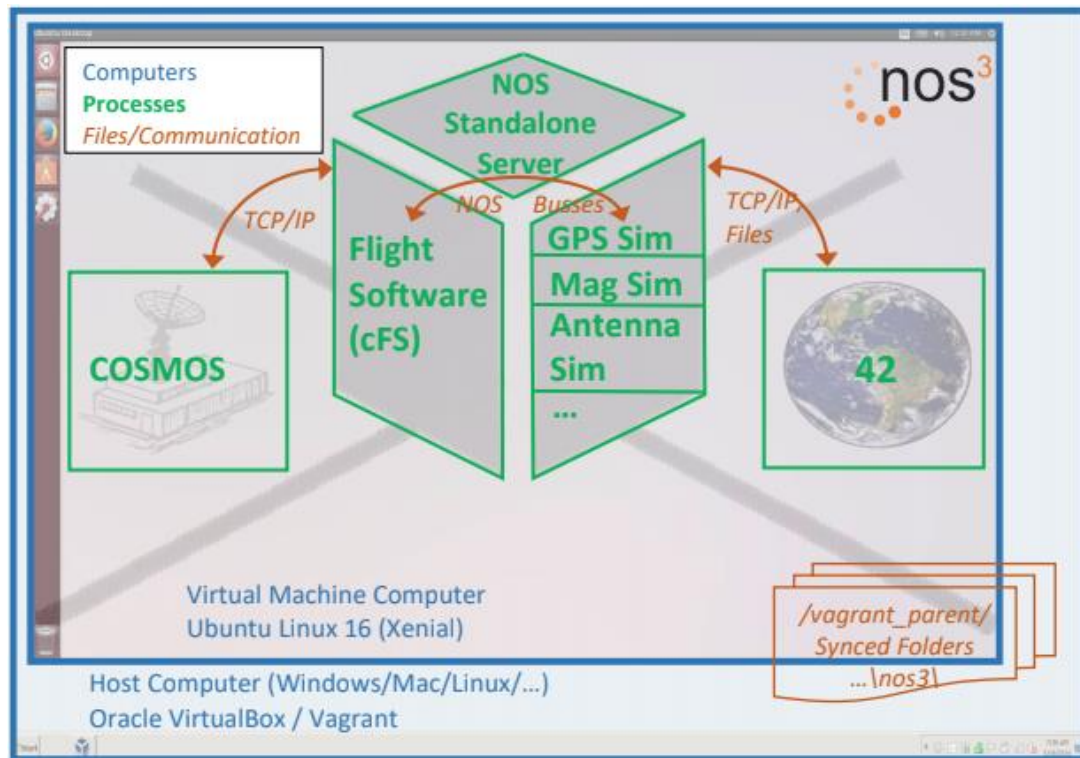
The NOS Engine is a component whose function is to simulate the hardware buses so that the flight software can communicate with the hardware simulators in a fast, simple and reusable way. With a modular design, the NOS Engine library provides a core that can be expanded to simulate specific communication protocols. The protocols it supports are: MIL-STD-1553, SpaceWire, I2C, SPI and UART.

The NOS engine is based on a model composed of 2 types of fundamental objects:

- **Nodes.** A node is any type of endpoint in the system, capable of sending and receiving messages
- **Buses.** A series of nodes belong to a group, called a bus. A node can belong to more than one bus and a bus can contain an arbitrary number of nodes. However, within the same bus, each node must have a different name. Between nodes on the same bus they can exchange messages, but they cannot communicate with nodes on other buses.

As described in the previous section, the NOS<sup>3</sup> provides the necessary tools to create and configure a virtual machine. This machine acts as a software container: it contains the necessary dependencies for the correct functioning of the simulator. Although if you want to avoid using the VM, you can run the NOS<sup>3</sup> executables directly on an Ubuntu 16.04 host, having installed the necessary packages before.

In this project, the NOS<sup>3</sup> has been implemented through the virtual machine and the tools that are provided for it. The components and how they are connected to each other are illustrated in Figure A.1.

Figure A.1: NOS<sup>3</sup> architecture

The host and the virtual machine share the nos3 folder. In it the code and libraries can be found. The cFS and the hardware simulators communicate with each other through the buses provided by the NOS Engine. On the other hand, the satellite system receives environmental data from 42 through the TCP / IP protocol. COSMOS interacts with the satellite using the same protocol.

## A.4 System architecture

The architecture of the system, see Figure A.2, has been defined through a component diagram. The list of components has been extracted from the set of user requirements. Among all of the componets of the diagram, the following have been implemented:

- Sensor simulator.
- Actuator simulator.
- “Keep in Orbit” application.

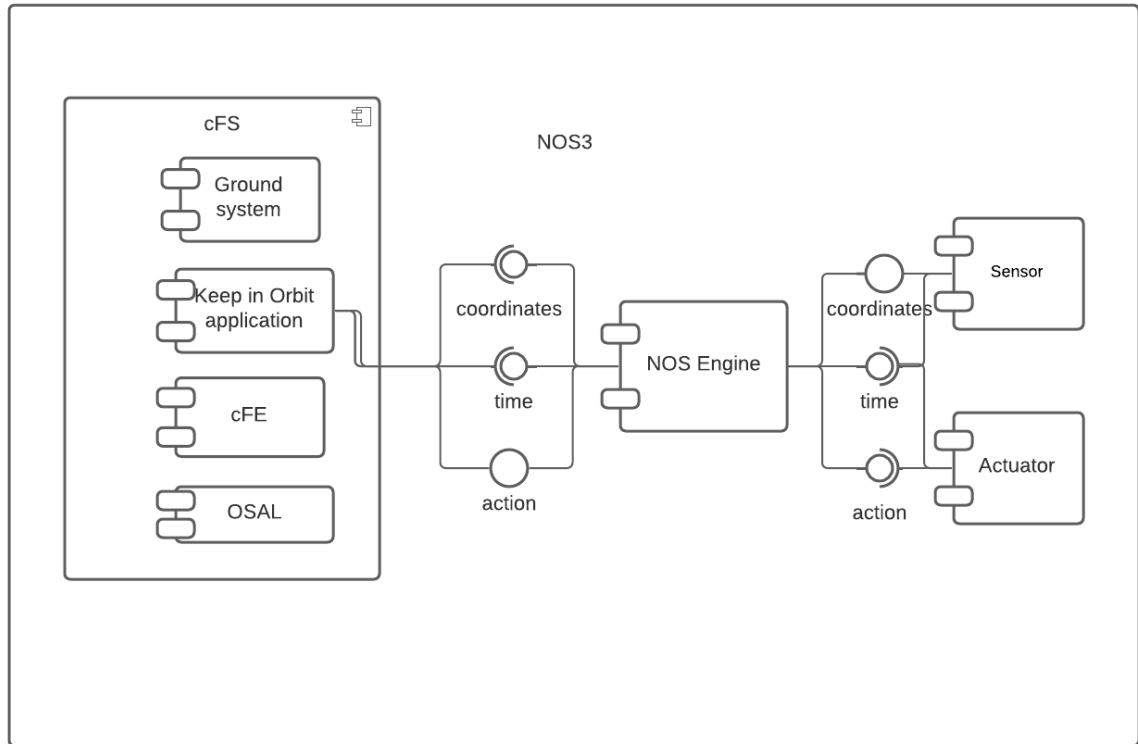


Figure A.2: Component diagram of the solution

## A.5 Implementation

The most important configuration parameters for each simulator can be found in the `/nos3/sims/cfg/nos3-simulator.xml` file. The XML format is a text-based markup language used to structure data logically. This file contains an entry with the tag “<simulator>” for each component simulator in the system. Within each entry, parameters such as: the name of the simulator, the library used for compiling, the connections it uses (name, type and port of each bus) or the parameters related to the periodic sending of data are specified. This configuration file facilitates the programmer's work since the most relevant parameters of each simulator can be edited from the same file.

### Sensor GPS

The GPS sensor is responsible for reading the satellite coordinates and sending them to the NOS Engine. The sensor has 2 different connections. The first is to the "command" bus to receive the time while the second connection is to the UART protocol bus number 18 to send the data to the NOS Engine. The period is 3 seconds and the sending begins in second 1.

The structure of the message is shown in the following table.

H	ID	ID	X	X	Y	Y	Z	Z	T
---	----	----	---	---	---	---	---	---	---

Table A.2: Message structure sent by the sensor

Each cell in the table is 1 byte in size. The total size of the message is 10 bytes. Each field in the message is detailed below.

Abbreviation	Data field	Size in bits	Combinations	Value type
H	Header	8 bits	256	uint
ID	Message identifier	16 bits	65536	uint
X	Coordinate X	16 bits	65536	double*
Y	Coordinate Y	16 bits	65536	double*
Z	Coordinate Z	16 bits	65536	double*
T	Trailer	8 bits	256	uint

Table A.3: Description of the fields of the message sent by the sensor

The header and trailer have fixed values while the others are variable. The coordinates are initially floating point. A variable of type double occupies 8 bytes of memory. To reduce the weight of the messages and simplify the problem, the value of the coordinates has been limited to a closed interval from 0 to 1. The floating point values of the coordinates are transformed to unsigned integers through a linear transformation:

$$\text{uint16\_t result} = (\text{uint16\_t}) (\text{coordinate} * 32767.0 + 32768)$$

This transformation guarantees a precision of up to 4 decimal places; enough to meet the requirement. Therefore the coordinate values supported by the system are included in the closed interval from 0.0000 to 1.0000.

## Engine

The function of the engine is to correct deviations from the satellite's path.

It is simpler than the sensor. It contains 2 connections: one to the “command” bus to receive the time and another to the USART 19 bus through which it will receive the messages from the cFE application. To simplify the problem, it has been assumed that the motor only acts in the 3 coordinate axes and in both directions (a total of 6 different directions). Although more bits have been allocated than necessary to provide scalability to more addresses. The structure of the message you receive is as follows:

H	ID	ID	P	T
---	----	----	---	---

Table A.4: Structure of the message received by the engine

Each cell represents 1 byte in size. The total size of the message is 5 bytes. The message fields are described in the following table.

Abbreviation	Data field	Size in bits	Combinations	Value type
H	Header	8 bits	256	uint
ID	Message identifier	16 bits	65536	uint
P	Payload	8 bits	256	uint
T	Trailer	8 bits	256	uint

Table A.5: Description of the fields of the message received by the engine

In the payload, the direction and the sense in which the motor must act are encoded. The first 4 bits are used to encode the axis if the movement should be positive or negative while the remaining 4 bits contain direction. Table X shows the possible values of the message. After receiving and decoding the message, the engine simulates its execution by printing the message on its console “Motor Activated on Axis X/Y/Z POSITIVE/NEGATIVE”.

Value of payload (HEX)	Axle direction	Sense
0x00	X	POSITIVE
0x10	X	NEGATIVE
0x01	Y	POSITIVE
0x11	Y	NEGATIVE
0x02	Z	POSITIVE
0x12	Z	NEGATIVE

Table A.6: Coding of the payload

## “Keep in Orbit” Application

The “Keep in Orbit” can be found in /fsw/components

The application consists of 2 files: *keep\_in\_orbit\_app.c* and *keep\_in\_orbit\_device.c*. The first contains the basic functions of an application: the initialization of the app, the communication with the “Ground” and the communication with the cFE software bus (do not be confused with the simulated buses of the NOS Engine). Apart from these, it includes the initialization of the device class.

The device class of the app includes the necessary libraries to communicate with the NOS Engine (hwlib) and is executed as a child process of the app process. In the class initialization method, the connection to 2 software buses has been implemented: uart 18 to receive the sensor data and uart 19 to send data to the motor.

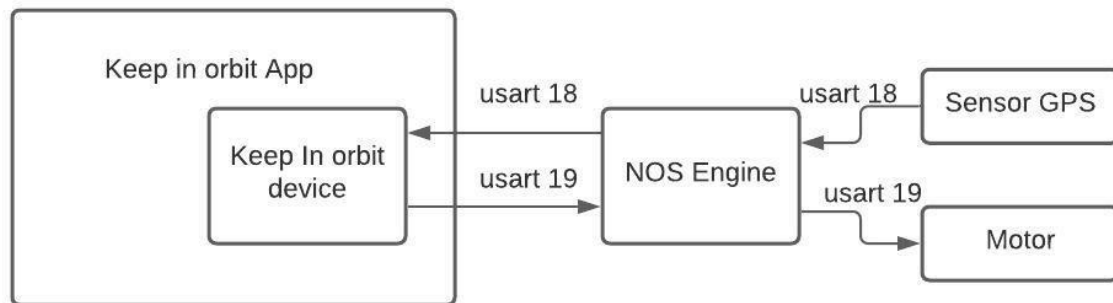


Figure A.3: Communication diagram

Within the device class is the implementation of sensor data processing and decision making. GPS sensor coordinates are received in `uint16_t` (2-byte unsigned integer) format. To convert them back to floating point format it is necessary to calculate the inverse of the transformation that the sensor has carried out.

$$\text{double coordinate} = (\text{double}) (\text{result} - 32768) / 32767.0$$

After having preprocessed the coordinates, the application checks if the coordinates follow the established path. To simplify the problem, a very simple satellite path has been defined: the 3 coordinates must have the same value ( $x = y = z$ ). The app checks if any coordinate is different, and then prepares a message to send it on the `uart 19` bus.

## A.6 Project plan

This section quantifies the total cost of the project based on its duration and the resources used. The total cost is broken down into: personnel cost, material cost, and indirect costs. Subtotal costs do not include the corresponding “IVA” 21% tax. Taxes will be added to the final cost.

### Personnel costs

The Table A.7 shows the number of hours used in each phase.

Task	Hours
Planification	15 h
Analysis y design	85 h
Implementation	130 h
Testing y evaluación	20 h
Documentation	100 h
Total	350 h

Table A.7: Distribution of project hours

Next, the personnel costs will be calculated according to the type of professional employed and the number of hours they have worked.

Job title	Hours	Cost per hour	Total
Analyst	40 h	45.00 €	1800.00 €
Analyst/Programer	125 h	28.00 €	3500.00 €
Programer	185 h	17.00 €	3145.00 €
Total	350 h		8445.00 €

Table A.8: Personnel costs

## Material costs

This project has been developed using free and open source software, most of it provided by NASA. However, the hardware that has been used to carry out the project does have an associated cost. Although the entire cost of the hardware will not be charged, but a proportional part of its useful life (which is called as amortization). The following table shows the hardware components, their total cost and their amortization for this project.

The material costs including the amortization of all of the hardware used go up to 214.71 €.

## Indirect costs

Indirect costs cannot be directly assigned to the production of the system, although they are essential: without them this project could not have been carried out. In this category are expenses such as internet, which consists of a flat monthly rate of € 38.00; divided equally between the 3 tenants of the property would give a monthly cost of € 12.67. For the calculation of energy expenditure, a cost of € 0.45 per KWh and an average consumption of 400W per hour from the power supply has been assumed. And the academic material has included the printing of an official NOS<sup>3</sup> manual, pens and sheets.



Resource	Cost
Electricity	25.00 €
Internet	114.00 €
Academic material	45.00 €
Total	184.00 €

Table A.9: Indirect costs

## Total cost

After having considered all the costs obtained in the previous sections, a total calculation of the cost of the project can be made. It is necessary to include a 10% margin to correct possible unforeseen events (hardware failures, medical leave) as well as a profit margin of 25% from where the profit is obtained.

As mentioned in previous sections, the cost computed so far is tax-free. According to law 37/1992 published in the "BOE" number 312 on "Impuesto de Valor Añadido", a percentage of 21% must be added as a tax to obtain the final price of the client.

Description	Valor
Personal costs	8445.00 €
Material costs	214.71 €
Indirect costs	184.00 €
Contingency margin (10%)	884.37 €
Profit margin (25%)	2432.02 €
Taxes (21%)	2553.62 €
<b>Total</b>	<b>14713.72 €</b>

Table A.10: Total costs

Therefore, the final price of the solution with the 21% tax included amounts to 14713.72 €, (**FOURTEEN THOUSAND SEVEN HUNDRED THIRTEEN EUROS AND 72 CENTS**)

## A.7 Conclusions

This end-of-degree Final Project has been a great challenge. It has required planning to distribute the load in 8 months that has lasted and dedication and perseverance throughout the process. There has been a first part of research among all the participants of the scholarship.

We have held several meetings in which our tutor, Javier Fernández Muñoz, has informed us of the general project of the chair and the technologies he uses (mainly cFE and RTEMS). Then a project has been assigned to each one and the work has become individual.

This project has allowed me to learn how a test bench works and in general a complex software architecture (a large file structure, many libraries, many Makefiles, and applications written in different languages...). Thanks to it I have learned to develop software in a systematic way. That is, instead of directly programming, following the software development process:

- Extract a set of user requirements that must be met.
- Do a requirements analysis and obtain system requirements: less general and more concise.
- Design the solution with the help of diagrams.
- Implement the solution.
- And finally, carry out a series of tests to verify that the system works correctly and the requirements are met.

Various problems and obstacles have arisen throughout the project. Most of it during the implementation of the simulators because there is not much online information about the NOS<sup>3</sup>. There is a developer guide published by NASA that provides a general idea of how the architecture works. But it offers almost no examples and no developer has posted a tutorial or example of how to use the test bench on the internet. The implementation part of the cFE application has been easier since there is more material published.

## Future work

This project can be expanded in a number of ways. Some of them are listed below:

**Simulate the behavior of the real components of the satellite.** Through the NOS<sup>3</sup> it is very easy to implement more simulators. All those that make up the nanosatellite could be implemented to achieve a realistic simulation.

**Bring together real and simulated hardware for execution.** The NOS<sup>3</sup> allows to connect real hardware. The real hardware and simulated components make use of a NOS client to communicate with the NOS<sup>3</sup> through the TCP / IP protocol. This allows to easily switch between real and simulated hardware.



## Apéndice B.

# Glosario

**API:** *Application Programming Interface*. Conjunto de funciones que permiten la comunicación entre componentes.

**cFS:** *Core Flight Software*. Software de vuelo desarrollado por la NASA.

**cFE:** *Core Flight Executive*. Colección de software desarrollada por la NASA que sirve entorno para desarrollar y ejecutar aplicaciones de vuelo

**EEUU:** Estados Unidos

**ESA:** European Space Agency. Traducción al español: Agencia Espacial Europea

**NASA:** National Aeronautics and Space Administration.

**NNUU:** Naciones Unidas

**RTOS:** Real Time Operating System. Traducción al español: Sistema Operativo de Tiempo Real.

**UNOOSA:** United Nations Office for Outer Space Affairs Traducción al español: Oficina de Naciones Unidas para Asuntos del Espacio Exterior

**URSS:** Unión de Repúblicas Socialistas Soviéticas

## Apéndice C

# Manual de usuario

Este capítulo describe la guía de instalación y uso del software NASA Operational Simulator for Small Satellites (NOS<sup>3</sup>).



Figura C.1: Entorno de ejecución de NOS<sup>3</sup>

Instalar y poner NOS<sup>3</sup> a funcionar es un proceso sencillo. De todos modos, será descrito a continuación de una forma detallada.

### Paso 1:

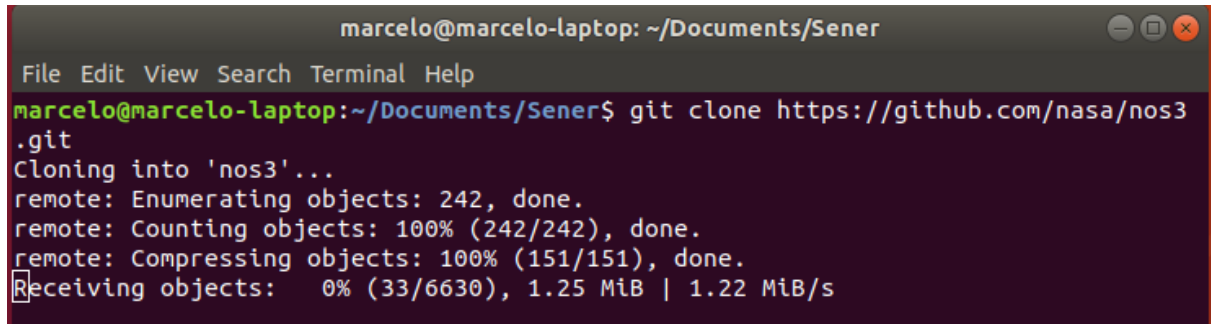
El primer paso consiste en instalar las dependencias requeridas. En la Tabla C.1 se exponen las dependencias y el enlace para descargarlas. En una distribución de linux debian, también se pueden descargar los paquetes con el comando apt.

Software	Version	Enlace de descarga
Oracle VirtualBox	5.1 o superior	<a href="https://www.virtualbox.org/wiki/Downloads">https://www.virtualbox.org/wiki/Downloads</a>
Hashicorp Vagrant	1.9 o superior	<a href="https://www.vagrantup.com/downloads">https://www.vagrantup.com/downloads</a>
Git	1.8 o superior	<a href="https://git-scm.com/downloads">https://git-scm.com/downloads</a>

Tabla C.1: Dependencias de NOS<sup>3</sup>**Paso 2:**

El segundo paso trata de descargar el directorio de archivos de NOS<sup>3</sup> de github. La forma más sencilla para el usuario hacerlo es a través de la terminal:

- Abrir una terminal.
- Navegar al directorio deseado donde los archivos se alojarán.
- Clonar el repositorio con git. Use `git clone https://github.com/nasa/nos3.git`
- Clonar (descargar) los submódulos de nos3. Use `cd nos3` y entonces `git submodule init` y `git submodule update`



```

marcelo@marcelo-laptop: ~/Documents/Sener
File Edit View Search Terminal Help
marcelo@marcelo-laptop:~/Documents/Sener$ git clone https://github.com/nasa/nos3
.git
Cloning into 'nos3'...
remote: Enumerating objects: 242, done.
remote: Counting objects: 100% (242/242), done.
remote: Compressing objects: 100% (151/151), done.
Receiving objects: 0% (33/6630), 1.25 MiB | 1.22 MiB/s

```

Figura C.2: Uso de git para clonar el repositorio NOS3

**Paso 3:**

Una vez descargado el repositorio de nos3, se ejecuta vagrant para que cree y configure la máquina virtual de NOS<sup>3</sup>. Antes de ello, se pueden editar algunas opciones de configuración en el archivo *CONFIG*:

- Sistema operativo: Ubuntu o CentOS (aún no soportado en esta versión de NOS<sup>3</sup>)
- Software de Tierra: AIT o COSMOS.

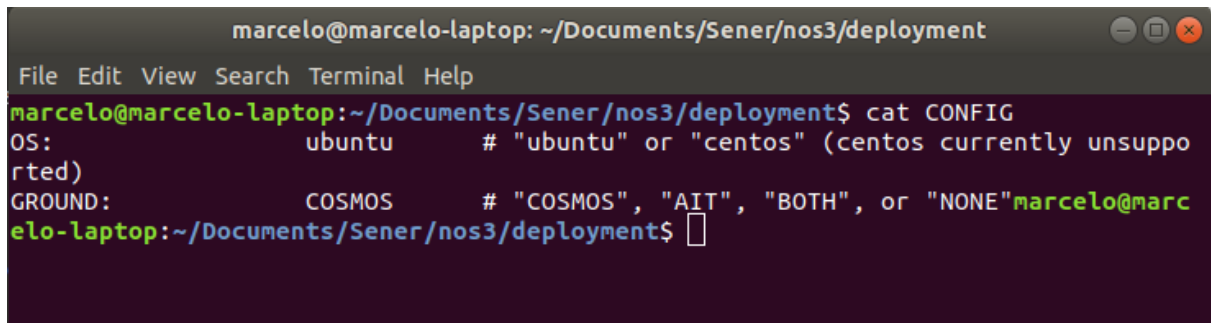
A terminal window titled 'marcelo@marcelo-laptop: ~/Documents/Sener/nos3/deployment'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command 'cat CONFIG' and its output: 'OS: ubuntu # "ubuntu" or "centos" (centos currently unsupported)' and 'GROUND: COSMOS # "COSMOS", "AIT", "BOTH", or "NONE"'. The prompt is 'marcelo@marcelo-laptop:~/Documents/Sener/nos3/deployment\$'.

Figura C.3: Opciones de configuración

Para ejecutar vagrant, se debe ejecutar el comando `vagrant up` desde el directorio `/nos3/deployment`. Una vez ejecutado, vagrant se encargará del resto del trabajo. Este comando puede durar desde minutos hasta varias horas en función de la potencia de computación y de los recursos de la red.

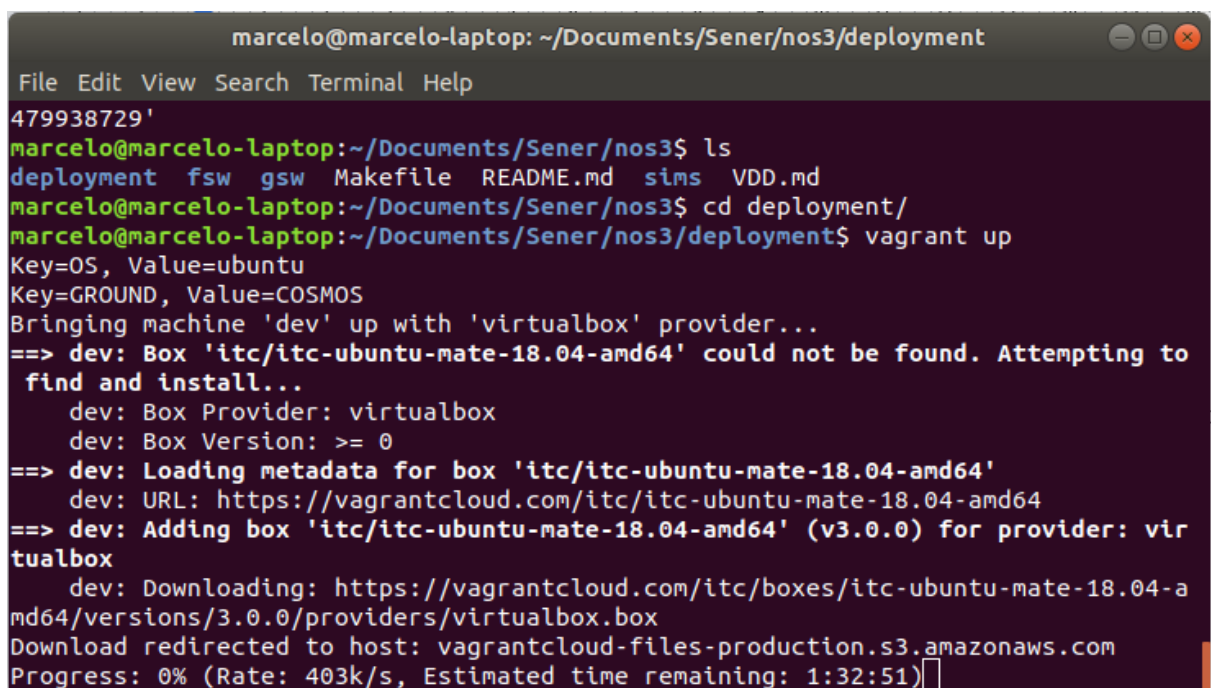
A terminal window titled 'marcelo@marcelo-laptop: ~/Documents/Sener/nos3/deployment'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command 'ls' and its output: 'deployment fsw gsw Makefile README.md sims VDD.md'. Then, the command 'cd deployment/' is executed. Finally, the command 'vagrant up' is executed, showing progress: 'Key=OS, Value=ubuntu', 'Key=GROUND, Value=COSMOS', 'Bringing machine 'dev' up with 'virtualbox' provider...', '==> dev: Box 'itc/itc-ubuntu-mate-18.04-amd64' could not be found. Attempting to find and install...', 'dev: Box Provider: virtualbox', 'dev: Box Version: >= 0', '==> dev: Loading metadata for box 'itc/itc-ubuntu-mate-18.04-amd64'', 'dev: URL: https://vagrantcloud.com/itc/itc-ubuntu-mate-18.04-amd64', '==> dev: Adding box 'itc/itc-ubuntu-mate-18.04-amd64' (v3.0.0) for provider: virtualbox', 'dev: Downloading: https://vagrantcloud.com/itc/boxes/itc-ubuntu-mate-18.04-amd64/versions/3.0.0/providers/virtualbox.box', 'Download redirected to host: vagrantcloud-files-production.s3.amazonaws.com', 'Progress: 0% (Rate: 403k/s, Estimated time remaining: 1:32:51)'. The prompt is 'marcelo@marcelo-laptop:~/Documents/Sener/nos3/deployment\$'.

Figura C.4: Ejecución de vagrant

Vagrant descargará la máquina virtual ubuntu-18.04 en la ruta `~/.vagrant.d/boxes` y la usará como base en VirtualBox. Después aplicará diversos cambios a la máquina:

- Editar el nombre de la máquina.
- Editar el tamaño de la memoria RAM.
- Editar la configuración de red.
- Activar el portapapeles compartido.
- Configurar las carpetas compartidas.

Una vez la máquina haya reiniciado varias veces, el usuario deberá introducir la contraseña *nos3123!* en el usuario *nos3*.



Figura C.5: Escritorio inicial del NOS3

## Ejecutar NOS3

Para ejecutar el banco de primero se debe crear una carpeta compartida para que la VM tenga acceso al nos3 (alojado en el host).



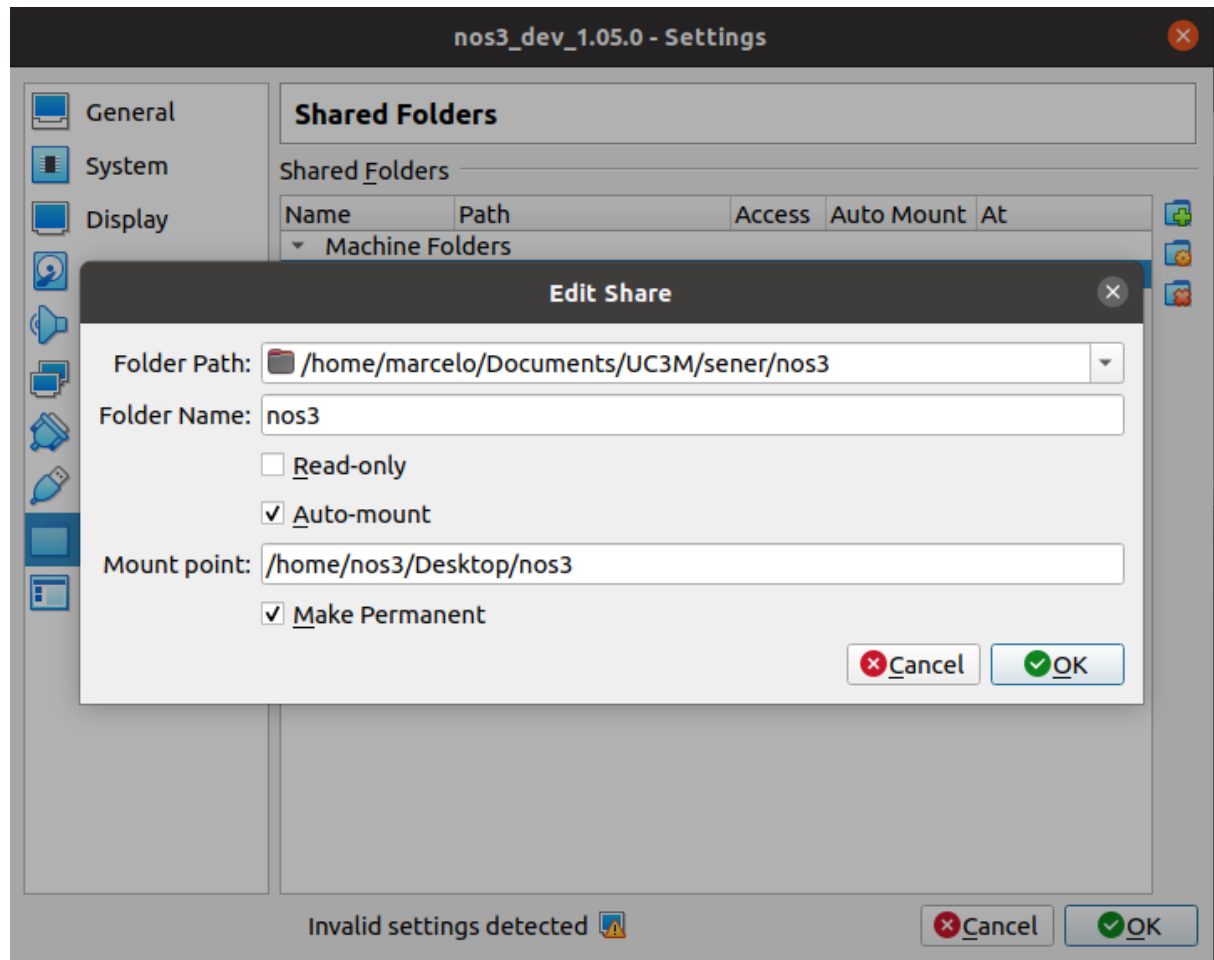


Figura C.6: Configuración de carpeta compartida

Una vez creada, se ejecuta el comando `make` desde el repositorio de `/nos3` para compilar los ejecutables. Acto seguido, se ejecuta `make launch` para correr la simulación y `make stop` para detenerla.

Para finalizar, es conveniente detener la vm usando `vagrant suspend`.

## Apéndice C

# Bibliografía

- [1] «Wiki,» [En línea]. Available: <https://es.wikipedia.org/wiki/SENER>. [Último acceso: 20 Mayo 2021].
- [2] «Alen SPace,» [En línea]. Available: <https://alen.space/es/space-business-es/>. [Último acceso: 27 Mayo 2021].
- [3] «CubeSat,» [En línea]. Available: <https://www.cubesat.org/about>. [Último acceso: 27 Mayo 2021].
- [4] UNED, «ieec.uned.es,» [En línea]. Available: [http://www.ieec.uned.es/investigacion/Dipseil/PAC/archivos/Informacion\\_de\\_referencia\\_ISE4\\_2\\_2.pdf](http://www.ieec.uned.es/investigacion/Dipseil/PAC/archivos/Informacion_de_referencia_ISE4_2_2.pdf). [Último acceso: 5 Mayo 2021].
- [5] «Blackberry QNX,» [En línea]. Available: <https://blackberry.qnx.com/en/software-solutions/embedded-software/qnx-neutrino-rtos>. [Último acceso: 20 Mayo 2021].
- [6] «VxWorks,» [En línea]. Available: <https://resources.windriver.com/vxworks/vxworks-product-overview>. [Último acceso: 20 Mayo 2021].
- [7] NASA, «cFS website,» [En línea]. Available: <https://cfs.gsfc.nasa.gov/>. [Último acceso: 5 Junio 2021].
- [8] «dSPACE,» [En línea]. Available: <https://www.dspace.com/en/pub/home.cfm>. [Último acceso: 20 Mayo 2021].
- [9] «ETAS,» [En línea]. Available: <https://www.etas.com/en/>. [Último acceso: 11 Mayo 2021].
- [10] N. Unidas, «unoosa.org,» [En línea]. Available: <https://www.unoosa.org/pdf/publications/STSPACE11S.pdf>. [Último acceso: 5 Junio 2021].

- [11] UNOOSA, «Treaty on principles governing the activities of states,» [En línea]. Available: <https://www.unoosa.org/oosa/en/ourwork/spacelaw/treaties/travaux-preparatoires/outerspacetreaty.html>. [Último acceso: 5 Junio 2021].
- [12] UNOOSA, «Convention on international liability for damage caused by space objects,» [En línea]. Available: <https://www.unoosa.org/oosa/en/ourwork/spacelaw/treaties/travaux-preparatoires/liability-convention.html>. [Último acceso: 5 Junio 2021].
- [13] UNOOSA, «Convention on registration of objects launched into outer space,» [En línea]. Available: <https://www.unoosa.org/oosa/en/ourwork/spacelaw/treaties/registration-convention.html>. [Último acceso: 5 Junio 2021].
- [14] H. R. Gil, «Universidad Autónoma de Barcelona,» [En línea]. Available: [https://ddd.uab.cat/pub/tfg/2020/225082/TFG\\_hruizgil.pdf](https://ddd.uab.cat/pub/tfg/2020/225082/TFG_hruizgil.pdf). [Último acceso: 5 Junio 2021].
- [15] ECSS, «ECSS-E-ST-10-03C – Testing,» [En línea]. Available: <https://ecss.nl/standard/ecss-e-st-10-03c-testing/>. [Último acceso: 5 Junio 2021].
- [16] «SpaceX,» [En línea]. Available: <https://www.spacex.com/>. [Último acceso: 11 Junio 2021].
- [17] E. Musk, «Elon Musk Motivational Video».
- [18] Parabolicarc, «Reusable Falcon 9 Would Cost \$5 to \$7 Million Per Launch,» [En línea]. Available: <http://www.parabolicarc.com/2014/01/14/shotwell/>. [Último acceso: 11 Junio 2021].
- [19] «Wiki,» [En línea]. Available: [https://es.wikipedia.org/wiki/Sat%C3%A9lite\\_artificial](https://es.wikipedia.org/wiki/Sat%C3%A9lite_artificial). [Último acceso: 2 Mayo 2021].
- [20] «RAE,» [En línea]. Available: <https://dle.rae.es/sat%C3%A9lite#MW8PII5>. [Último acceso: 5 Mayo 2021].
- [21] U. D. Algo, «MarteOS,» [En línea]. Available: <https://marte.unican.es/>. [Último acceso: Mayo 20 2021].
- [22] «FreeRTOS,» [En línea]. Available: <https://www.freertos.org/>. [Último acceso: 20 Mayo 2021].

- [23] «OPAL-RT,» [En línea]. Available: <https://www.opal-rt.com/>. [Último acceso: 20 Mayo 2021].
- [24] N. Unidas, «UNOOSA,» [En línea]. Available: <https://www.unoosa.org/oosa/en/ourwork/spacelaw/index.html>. [Último acceso: 5 Junio 2021].
- [25] J. F. Muñoz, *Sistemas de Tiempo Real 2: Sistemas empotrados*, UC3M.