UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Desarrollo de un sistema supervisor para incrementar la fiabilidad de una computadora de a bordo que utilice Core Flight System

TESIS

Que para obtener el título de

Ingeniero Eléctrico Electrónico

P R E S E N T A

David Zavala Pérez

DIRECTOR DE TESIS

M.I. Christo Aldair Lara Tenorio

Ciudad Universitaria, Cd. Mx., 2025

PROTESTA UNIVERSITARIA DE INTEGRIDAD Y HONESTIDAD ACADÉMICA Y PROFESIONAL
(Titulación con trabajo escrito)

De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a la institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado DESARROLLO DE UN SISTEMA SUPERVISOR PARA INCREMENTAR LA FIABILIDAD DE UNA COMPUTADORA DE A BORDO QUE UTILICE CORE FLIGHT SYSTEM. que presenté para obtener el titulo de INGENIERO ELÉCTRICO ELECTRÓNICO es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Entidad Académica, citando las fuentes de ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia, acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad de los actos de carácter académico administrativo del proceso de titulación.

DAVID ZAVALA PEREZ
Número de cuenta: 420053655

Agradecimientos

A mi familia, Berenice, Martín y Marisol. Por su amor, paciencia y apoyo incondicional en cada etapa de mi vida, por ser mi refugio y motivación para seguir adelante. Porque incluso en la distancia, siempre estuvieron presentes. Ustedes son mi base y fuerza para salir adelante.

A Aurora Rábago, por su compañía, apoyo y cariño durante este proyecto.

A las personas que me acompañaron en mi tiempo en la facultad: Enrique Guerra, Óscar Rendón, Germán Alday, Daniela Maldonado, Kalid González, Fernando Plata , Dennisse Viveros, Diana Hernández, Leorando Nicolás y Rodrigo García. Que con su apoyo, cariño y compañía hicieron de la universidad un lugar más agradable. Gracias por estar presentes en mis buenos y malos momentos, por ayudarme a ser mejor persona cada día y por motivarme cuando yo más dudaba.

A mi asesor, M.I. Aldair Lara, por su guía, dedicación y consejo durante gran parte de mi carrera. Gracias por compartir tus conocimiento e impulsarme a mejorar personal y profesionalmente. Al Dr. Saúl de la Rosa, quien siempre me brindó su apoyo y experiencia, en este y otros proyectos.

A mis profesores, por haberme formado con pasión, dedicación y conocimiento, en especial a: M.I. Lauro Santiago, Dr. Jorge Rodríguez, M.I. Larry Escobar, Dr. José María, Dr. Francisco Martínez, M.I. Ricardo Mota y M.I. Vicente Flores.

A todos ustedes, mi sincero agradecimiento.

-David Z.

Contenido

Resumen

La NASA desarrolló *Core Flight System* (cFS), un *framework* de código abierto que permite reducir costos y tiempos en el diseño de software de a bordo (FSW) para satélites, al proporcionar una base común para la mayoría de proyectos espaciales, además de ser reutilizable y modular. Este *framework* cuenta con un excelente sistema de detección de errores, sin embargo, la mitigación de estos requiere en su mayoría de intervención humana o de múltiples aplicaciones y algoritmos que hacen el proceso ineficiente y propenso a fallar. En este trabajo de tesis se desarrolló un sistema supervisor como esquema de tolerancia a fallas para complementar el FSW basado en cFS, que permite la reconfiguración autónoma de aplicaciones dañadas, además de integrar un *watchdog* como redundancia de hardware. El sistema supervisor presenta un impacto mínimo en el sistema principal, es de bajo costo y es portable a cualquier plataforma que ejecute cFS.

Parte I

PLANTEAMIENTO DEL PROYECTO

1 Introducción

La exploración espacial ha creado enormes beneficios para la humanidad, incluyendo la observación de nuestro planeta con aplicaciones en el monitoreo de la formación de huracanes y el seguimiento de su trayectoria, permitiendo detectar y alertar rápidamente a poblaciones en riesgo; el monitoreo del cambio climático, detección de incendios forestales y de inundaciones, seguimiento de tsunamis, entre otros. También existen aplicaciones de comunicaciones, como internet de alta velocidad en cualquier parte del mundo, transmisión de señales de televisión y telefonía satelital; aplicaciones de navegación por satélites como: GPS, GLONASS y Galileo, que facilitan navegar de forma precisa y segura por aire, mar y tierra. Además, se desarrollan aplicaciones de monitoreo del Sol y el espacio profundo, con el objetivo de entender la formación del sistema solar, buscar indicios de vida en otros planetas, entender cómo se ha formado el universo, y buscar amenazas que puedan comprometer la vida en nuestro planeta como asteroides o partículas de alta energía [1].

En los sistemas espaciales existe una gran variedad de problemas como producto de las condiciones extremas en las que operan. Para órbita terrestre baja (LEO, por sus silgas en inglés), los eventos de efecto único (SEE, por sus siglas en inglés) pueden provocar errores transitorios o permanentes en los dispositivos electrónicos de todos los subsistemas del satélite. En este trabajo se tiene como objetivo proteger a la computadora de a bordo (OBC, por sus siglas en inglés) de un, en donde es imprescindible detectar y mitigar fallas para evitar la pérdida de la misión [2]. A la capacidad de un sistema para continuar realizando sus funciones a pesar de la presencia de fallas o errores se le conoce como tolerancia a fallas [3], el cual es un parámetro fundamental durante todas las fases del desarrollo de OBC, desde la planeación hasta la implementación de hardware y de software [4].

Durante las últimas décadas el desarrollo de nanosatélites [1] ha aumentado significativamente cambiando el paradigma de la industria espacial, pasando de grandes satélites insignia a pequeñas naves espaciales mucho más económicas, principalmente para LEO, con las capacidades para ser usados con fines científicos, comerciales y sociales [6]. Sin embargo, este tipo de satélites añaden como limitantes en el desarrollo de sus subsistemas el tamaño, la masa, el consumo de potencia y los costos.

El desarrollo de sistemas espaciales es costoso debido a la complejidad de las tareas necesarias para garantizar el correcto funcionamiento del sistema en el espacio. Para lograr esto, se trabaja con equipos multidisciplinarios encargados de resolver problemas específicos. Por ejemplo, el desarrollo del software de a bordo (FSW, por sus siglas en inglés) requiere de un gran personal y el uso de múltiples herramientas, además, cuando se trabaja en múltiples proyectos, cada plataforma que se realice necesita del desarrollo de un software especializado que muchas veces son incompatibles con las misiones anteriores, obligando al personal a iniciar el desarrollado desde etapas tempranas. Para contrarrestar estos problemas, la NASA desarrolló un *framework* de código abierto, denominado *Core Flight System (cFS)*, que proporciona una plataforma de código reutilizable y compatible con

[1] Todo satélite con una masa localizada entre 1kg y 10kg, además de *CubeSats* de 1 a 27U con masa de hasta 40kg [5].

todo tipo de arquitectura, siempre que esta cuente con las capacidades suficientes para ejecutar un sistema operativo (OS, por sus siglas en inglés) compatible como Linux o RTEMS [7].

El uso de cFS se hace muy atractivo como FSW para nanosatélites, ya que permite reducir costos de forma significativa, sin embargo, el software resultante es demandante para un sistema embebido (como un microcontrolador), además de que el uso de una computadora con garantía de resistencia a la radiación (RHA, por sus siglas en inglés), también llamadas de grado espacial, con las características suficientes para ejecutar cFS presenta costos muy elevados, por lo que, es común el uso de computadoras embebidas[2] por su capacidad de procesamiento, precio y su tamaño compacto, sin embargo, estas computadoras son construidas con componentes comerciales, también llamados productos comerciales disponibles en el mercado (COTS, por sus siglas en inglés), los cuales no cumplen con las certificaciones suficientes para operar en el entorno espacial. Debido a esto, existe una alta probabilidad de que la OBC presente fallas de tipo *soft*[3] inducidas por el medio ambiente de radiación espacial lo que hace sensible el sistema a pesar de contar con cFS, lo que resulta en la necesidad de implementar técnicas de tolerancia a fallas.

[2]Las computadoras embebidas o mono placa son sistemas completos con microprocesador, memorias, puertos y demás funciones necesarias para tener una computadora funcional, con la capacidad de ejecutar sistemas operativos y grandes cargas de procesamiento a un bajo costo energético [8].

[3]Falla que permite seguir operando, pero con capacidades parciales [9]

2 Metodología

Como método de diseño se implementó un flujo en espiral (véase la figura 2.1), en donde cada etapa, a partir del planteamiento del proyecto, podrá iterarse cuantas veces sea necesaria hasta conseguir un resultado satisfactorio. Entre cada etapa se encuentra un punto de revisión para confirmar que la etapa se concluyó con éxito y determinar si es adecuado pasar a la siguiente fase.

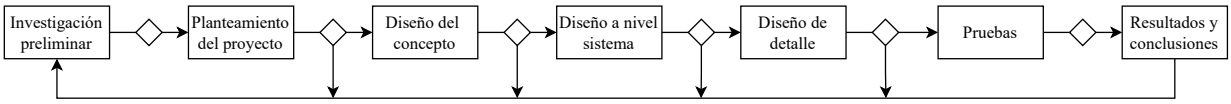


Figura 2.1: Metodología de diseño [10].

Esta es una metodología de desarrollo de proyectos planteada por Ulrich [10], que se compone por 6 etapas que permiten pasar de una necesidad a un producto:

1. **Investigación previa:** para identificar una necesidad que no ha sido explotada en el mercado.
2. **Planteamiento**: en donde se definen los objetivos y los alcances del proyecto, esta sección es la base para un desarrollo ordenado, eficiente y enfocado del producto y, por lo tanto, debe prestarse especial atención para alcanzar el éxito del proyecto.
3. **Diseño del concepto:** se identifican las necesidades y se genera una arquitectura general con divisiones por subsistemas para facilitar su desarrollo.
4. **Diseño a nivel sistema:** se define la arquitectura del sistema, los subsistemas y la interacción entre estos.
5. **Diseño de detalle**, se definen con precisión las especificaciones del producto.
6. **Pruebas:** el producto generado en las etapas anteriores se somete a evaluaciones para determinar el desempeño del diseño e identificar si cumple o no con los objetivos iniciales.
7. **Resultados y conclusiones:** resume el desarrollo realizado, analizando con detalle los resultados de las pruebas para determinar las áreas de mejora.

3 Planteamiento

En este capítulo se definen los objetivos y límites del proyecto. Es la base para un desarrollo ordenado, eficiente y, por lo tanto, debe prestarse especial atención para alcanzar el éxito del proyecto.

3.1 Definición del problema

Es importante no subestimar los efectos de la radiación espacial en semiconductores, ya que puede causar degradación de los componentes reduciendo su vida útil, o pueden inducir fallas que podrían comprometer la exactitud de sus resultados [2].

La NASA desarrolló un *framework* de código abierto, llamado cFS, que permite desarrollar software de vuelo para satélites en un corto periodo de tiempo. Además, proporciona diversas aplicaciones que permiten detectar fallas en el sistema, como la corrupción de datos en memoria. Sin embargo, la mitigación de fallas depende en su mayoría de intervención humana (a través de la estación terrena) o de la implementación de grandes cantidades de código, haciendo ineficiente el proceso e incrementando la probabilidad de fallas, por lo que en este trabajo de tesis se desarrolló un sistema supervisor como técnica de tolerancia a fallas que complemente el FSW de una OBC para nanosatélites que ejecute cFS, con la capacidad de realizar acciones correctivas autónomas en órbita con el fin de incrementar el nivel de fiabilidad del sistema, sin perjudicar la portabilidad característica de cFS.

3.2 Hipótesis

El uso de un supervisor externo que complemente a las técnicas de tolerancia a fallas propias del *framework* cFS para corregir errores por corrupción de información en las librería compartidas (aplicaciones de cFS), es una opción viable para incrementar el nivel de fiabilidad de una Computadora de a bordo (OBC) para nanosatélites *CubeSat*, conservando la portabilidad de cFS.

3.3 Objetivos

☐ General

- Incrementar el nivel de fiabilidad de una OBC que ejecute software de a bordo para nanosatélites basado en el *framework* cFS, a través de un sistema supervisor externo como esquema de tolerancia a fallas.

☐ Específicos

- Desarrollar un sistema supervisor utilizando principalmente componentes COTS, como técnica de redundancia por hardware para una OBC con software basado en cFS.
- Analizar los puntos únicos de falla (SPF, por sus siglas en inglés) de una OBC con FSW basado en cFS.
- Conservar la portabilidad de cFS por medio de una interfaz en la capa de aplicaciones de usuario para establecer comunicación con el supervisor.

3.4 Alcances

Este trabajo contempla el desarrollo de un sistema supervisor externo en conjunto con una aplicación desarrollada con cFS para su comunicación con la OBC, compatible con toda plataforma que cumpla con las especificaciones necesarias para ejecutar un OS compatible, implementando técnicas de tolerancia a fallas por hardware y software para detectar y corregir errores de corrupción en las librerías compartidas de aplicaciones de cFS, con el objetivo de garantizar un nivel de fiabilidad suficiente para desempeñar sus funciones en LEO, empleando principalmente componentes COTS.

3.5 Justificación

De acuerdo con el portal de la NASA [7], el *framework* cFS es una herramienta que reduce los tiempos de desarrollo en sistemas espaciales debido a su alta compatibilidad entre misiones, lo que facilita la reutilización de software. Además, es de código abierto y puede ejecutarse en cualquier arquitectura compatible con la norma de interfaz de sistema operativo portátil para Unix (POSIX, por sus siglas en inglés) establecida por IEEE, permitiendo el uso de computadoras construidas con componentes COTS reduciendo significativamente los costos de desarrollo, a costa de una reducción en el nivel de fiabilidad debido a una alta probabilidad de falla como efecto de la radiación en LEO.

El *framework* de la NASA cuenta con un excelente sistema de detección y reporte de errores, sin embargo, es responsabilidad del usuario la mitigación de éstos. Por tanto, cFS cuenta con múltiples aplicaciones que en conjunto permiten realizar la corrección de algunos errores, pero estos sistemas son altamente propensos a fallar debido a la dependencia de múltiples aplicaciones, en donde si una falla puede hacer imposible la recuperación del sistema, además de requerir muchas veces de la intervención humana que hace ineficiente el proceso. Esto crea la necesidad de implementar un sistema que, mediante técnicas de tolerancia a fallas, utilice los procesos de diagnóstico propios de cFS para identificar errores y realice tareas de recuperación autónomas del FSW, aumentando su nivel de fiabilidad con un bajo impacto en el sistema y a un bajo costo económico.

Parte II

INVESTIGACIÓN PRELIMINAR

4 Medio ambiente espacial

Desde tiempos prehistóricos el ser humano ha observado el cielo nocturno con admiración buscando patrones en las estrellas para darles un significado. Antiguas civilizaciones han dejado vestigios arqueológicos qué lo demuestran, como lo son Stonehenge (3000 a.C), un monumento alineado con el Sol y posiblemente usado para la observación de la Luna y el Sol [11]; o los mayas en Chichen Itzá y el famoso observatorio de Caracol (906 d.C), con el cual llegaron a utilizar los objetos del cielo nocturno para medir el tiempo y calcular con gran precisión fenómenos como eclipses solares y lunares [12].

Hoy en día, la curiosidad del ser humano no ha cambiado, ya que continúa estudiando y explorando el espacio exterior para el cual existen distintas interpretaciones sobre su frontera con la atmosfera terrestre, sin embargo, para efectos de este trabajo se tomará en cuenta la línea de Kárman, a 100km sobre el nivel del mar, planteada por Theodore Von Kárman y reconocida por la Federación Aeronáutica Internacional (FAI) siendo un estándar extensamente aceptado por la comunidad científica.

El espacio exterior, muchas veces pensado como un vacío absoluto entre planetas y estrellas, realmente mantiene un flujo constante de partículas cargadas conformadas principalmente por protones, núcleos de helio e iones pesados, pero también pueden incluir electrones, positrones y otras partículas subatómicas de alta energía. A este flujo de partículas se le conoce como rayos cósmicos (CR, por sus siglas en inglés) [13].

4.1 Fuentes de radiación

Los CR viajan por todo el espacio, incluyendo el interior y exterior de nuestro sistema solar, por lo que se establece una clasificación según su origen, lo que a su vez determina su nivel de energía (tabla 4.1).

Cuando los CR provienen del exterior de la galaxia, Vía Láctea, se les conoce como rayos cósmicos extra-galácticos, que incluyen la emisión de partículas altamente energéticas de hasta $10^{21}eV$, y son causadas por el agujero negro masivo que forma parte del núcleo de alguna galaxia, conocidos como cuásares que, al consumir materia, liberan grandes cantidades de energía. Por otra parte, están los CR galácticos, que se originan dentro de nuestra galaxia como producto de explosiones de supernovas o sus remanentes; y estrellas, con niveles de energía de hasta los $10^{16}eV$. Estos rayos están conformados casi por completo de protones, de los cuales un 90 % son del núcleo de átomos de hidrógeno, 9 % de helio y 1 % de elementos pesados [13].

Dentro del sistema solar se generan partículas cargadas (también conocidas como plasma) como consecuencia del proceso de fusión del Sol. En concreto, existen dos fuentes que emiten estas partículas: la atmósfera superior del Sol, que genera un flujo constante de partículas cargadas (principalmente protones y electrones) con un intervalo de energía entre 1.5 y 10keV, los cuales son eyectados a velocidades supersónicas abarcando todo lo que se conoce como heliosfera. Por otra parte, están las erupciones solares y eyecciones de masa coronal, que son fenómenos asociados a

un incremento espontáneo de energía magnética en la atmósfera del Sol, ocasionando la eyección de grandes cantidades de electrones, protones e iones con niveles de energía que alcanzan algunas decenas de GeV [13].

Origen	Niveles de energía	Fuentes
Extra-galáctico	10^{21} eV	Radiogalaxias, cuásares.
Galáctico	10^{15}–10^{16} eV	Explosiones de supernova, magnetosferas de pulsares y estrellas dobles, ondas de choque en el espacio interestelar.
Sol	15–30 GeV	Intensas erupciones solares de la corona.
Interplanetario	10–100 MeV	Ondas de choque terminal en el límite de la heliosfera.
Magnetosfera (Tierra)	3 keV	Generados dentro de la magnetosfera.

Tabla 4.1: Fuentes de radiación (adaptado de [13]).

Los planetas que mantienen un campo magnético fuerte generan una región en el espacio llamado magnetosfera, que se forma por la interacción del viento solar y el campo magnético del planeta, actuando como una barrera protectora contra las partículas solares y cósmicas. La altura de la magnetosfera terrestre es dinámica, pero de manera general se extiende a una distancia de 38,000km a 63,000km de la cara que ve al Sol mientras que del lado opuesto se extiende como una cola llegando a medir más de 600,000km como producto de la compresión generada por el constante bombardeo del viento solar [14].

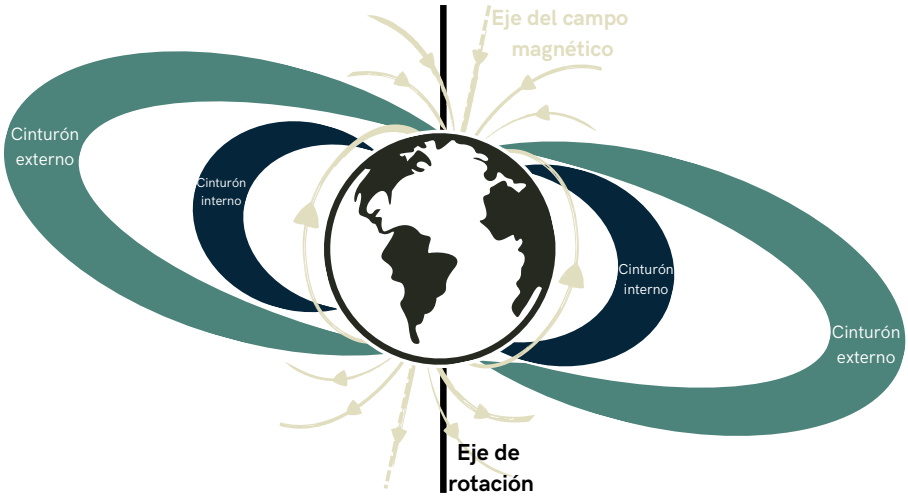


Figura 4.1: Cinturones de Van Allen (adaptado de [13]).

Algunas partículas cargadas que interactúan con el campo magnético de la Tierra quedan atrapadas y se mueven en espiral a lo largo de las líneas del campo magnético hasta quedar confinadas en zonas toroidales alrededor del planeta. A estas regiones se les conoce como cinturones de Van Allen (figura 4.1) y representan un riesgo para los satélites que orbiten a alturas cercanas a ellos. Los cinturones son simétricos alrededor del eje magnético de la Tierra, el cual está desplazado aproximadamente 11° del eje de rotación, encontrándose el cinturón interno entre 1000 y 12,000km sobre la superficie terrestre y estando conformado por protones y electrones, mientras que el cinturón exterior se encuentra entre 13,000 y 60,000km y contiene principalmente electrones [13].

4.2 Efectos de la radiación en los componentes electrónicos

Los errores inducidos por la radiación son un problema reciente en la historia, siendo en 1957 los primeros reportes de fallas y errores aleatorios e inexplicables en circuitos digitales tras realizarse pruebas nucleares superficiales. Los efectos de la radiación en los componentes electrónicos pueden clasificarse en dos tipos de acuerdo con su naturaleza: efectos acumulativos y eventos de efecto único (SEE, por sus siglas en inglés) [13].

Los efectos acumulativos, suelen estar asociado a la dosis total ionizante (TID, por sus siglas en inglés) que se define como la radiación total que recibe un componente electrónico durante su vida útil. Esto tiene un impacto en el funcionamiento de semiconductores, ya que las partículas cargadas pueden ionizar el material generando pares electrón-hueco. Por ejemplo, en la tecnología MOS puede cambiar el umbral del voltaje de encendido, generar corrientes de fuga y ruido. De forma general causan la degradación progresiva y eventualmente la pérdida del componente[2].

Los SEE son causados por la interacción instantánea de partículas cargadas con cientos de MeV. En la tecnología MOS, las partículas causan una transferencia de carga al atravesar el óxido, seguida de una recolección de carga por el nodo de salida causando pulsos de corrientes inesperadas en el dispositivo. De acuerdo con la cantidad de energía inyectada por la partícula pueden ocurrir 3 tipos de fallas: transitorias, *upset* y *latchup* [15].

Los transitorios de evento único (SET, por sus siglas en inglés), generan pequeños picos de corriente o voltaje en el dispositivo electrónico, sin embargo, esto sucede por un breve periodo de tiempo, por lo que, si ningún elemento lo registra al instante, la falla no se propaga [16].

Ocurre un *upset* de evento único (SEU, por sus siglas en inglés) cuando una partícula cargada genera un cambio del estado lógico en una celda de memoria o *flipflop*, cambiando un valor alto (1) por bajo (0), o un valor bajo por uno alto. Si la partícula tiene la suficiente energía, pueden afectarse de la misma forma las celdas de memoria adyacentes [16].

El *latchup* de evento único (SEL, por sus siglas en inglés), sucede cuando la partícula incidente lleva mucha energía, causando altas corrientes dentro del dispositivo, que de no llevar una protección contra altas corrientes, el dispositivo puede quemarse y por lo tanto quedar inservible [16].

5 Tolerancia a fallas

Las **fallas** son un defecto o condición fuera de lo esperado que ocurre en un componente de hardware como un corto circuito o de software como un *bug*; mientras que un **error** es la manifestación de la falla, definido como una desviación del resultado esperado, lo que puede resultar en un **malfuncionamiento** parcial o crítico del sistema. Las fallas son una amenaza para los sistemas, ocasionando una desviación en su funcionamiento que puede generar un malfuncionamiento o un servicio incorrecto, y como consecuencia el incumplimiento de los objetivos de la misión. El periodo de tiempo durante el cual el sistema opera de forma incorrecta se le conoce como tiempo de inactividad o interrupción de servicio. Por otro lado, cuando el sistema vuelve a la normalidad se dice que ocurrió una recuperación o reparación del servicio o sistema. Al proceso de generación de la falla y su propagación por el sistema se le conoce como ciclo de vida de un malfuncionamiento, representado gráficamente en la figura 5.1, donde en su primera etapa se origina una falla, sin embargo, el sistema mantiene un servicio correcto hasta que la falla se activa, convirtiéndose en un error y por tanto, ocurre un malfuncionamiento [17].

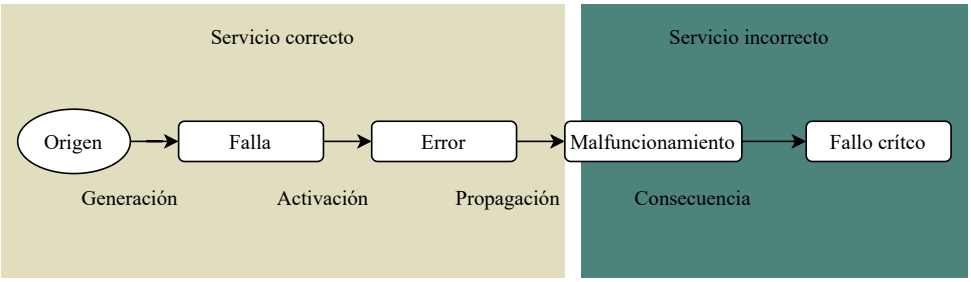


Figura 5.1: Ciclo de vida de un fallo (adaptado de [17]).

Las fallas pueden clasificarse de acuerdo con su duración: permanentes, en el cual un componente queda deshabilitado de forma indefinida si no se toman acciones correctivas, como lo puede ser un transistor que se ha quemado; transitorias, en donde un componente falla momentáneamente y vuelve a operar correctamente después de un tiempo definido por el origen de la falla, como el ruido en una conversación telefónica; intermitente, la falla nunca desaparece por completo, pero opera correctamente por intervalos, como un falso contacto en una conexión eléctrica. Otra clasificación en las fallas es de acuerdo con su impacto: benignas, en donde ocurre una falla, pero no genera resultados incorrectos y por lo tanto no existe riesgo de perder el sistema, como un led de estado que se funde; y las fallas malignas, que producen un resultado incorrecto, lo que puede generar una falla catastrófica, como lo podría ser la falla en el sensor de orientación de un satélite generando un valor erróneo en su ángulo de orientación que resulta en un satélite apuntando indefinidamente a la posición incorrecta, comprometiendo los objetivos de la misión [18].

No hay nada mejor que tener un sistema que funcione correctamente todo el tiempo sin una sola falla, sin embargo, por motivos fuera de nuestro control, como defectos en la fabricación, errores en el ensamblaje o diseño, errores generados por el entorno, entre otros, los sistemas se encuentran en un riesgo constante de presentar una falla, por lo que si se trata de un sistema crítico, como un satélite que se colocará en un entorno hostil con un acceso al mismo, el sistema debe ser capaz

de prevenir un malfuncionamiento a pesar de la existencia de fallas ocurridas durante el tiempo de ejecución, evitando tanto la activación como la propagación de la falla en el sistema, evitando un error, a esto se le conoce como tolerancia a fallas. También es posible prevenir fallas desde su origen, empleando componentes blindados contra la radiación desde la etapa de diseño del sistema. Sin embargo, esto presenta las desventajas de incrementar significativamente los costos y tiempos de desarrollo [17].

La base para tener una tolerancia a fallas es la redundancia, definida como la adición de recursos de software, hardware, tiempo o información. Se le conoce como redundantes porque en un sistema ideal, no son componentes necesarios para su funcionamiento [17].

5.1 Redundancia por hardware

La redundancia por hardware consiste en añadir componentes extras como procesadores, sensores, memorias, entre otros. Los métodos de redundancia por hardware se pueden clasificar en dos tipos: pasiva y activa. La **redundancia pasiva**, en la cual los componentes redundantes están desactivados hasta que una falla es detectada, reduciendo la probabilidad de fallas y el envejecimiento de los componentes en reposo, a costa de tener una lenta velocidad de respuesta al tener que esperar la activación de los componentes redundantes. Mientras que, la **redundancia activa** mantiene todos los componentes energizados y en operación, realizando de forma simultánea el mismo proceso con el fin de comparar sus resultados, lo que permite detectar y mitigar fallas con mayor velocidad que la redundancia pasiva, a costa de un mayor consumo de potencia, el envejecimiento de los módulos redundantes y un mayor costo económico [19].

5.1.1 Redundancia modular triple (TMR)

Redundancia de hardware activa que consiste en tener tres módulos idénticos que trabajan en paralelo, generando una salida independiente para cada uno conectadas a un sistema de votación aplicando una lógica de mayoría en donde, si al menos dos salidas coinciden entre sí, se toma como resultado correcto, si un módulo cualquiera difiere con el resultado se asume que ha fallado y se enmascara su resultado para evitar su propagación. Tiene como desventajas el costo económico ya que el número de módulos se triplica, en comparación con un sistema no redundante, además de consumir mayor potencia energética y solo puede detectar y enmascarar una falla si solo un módulo presenta error, ya que si 2 o 3 módulos fallan, el sistema ya no puede determinar el resultado correcto de forma inmediata, por tanto, es necesario repetir la operación [17].

5.1.2 Redundancia modular doble (DMR)

Redundancia de hardware activa en donde se tienen dos módulos idénticos con un comparador conectado a sus salidas para compararlas y determinar una condición de error al tener salidas diferentes, es entonces responsabilidad del diseñador o del usuario manejar esta falla, por ejemplo, realizando de nuevo la ejecución que presentó el error. En comparación con la TMR, el consumo de potencia y los costos económicos son menores, sin embargo, se requiere de una mayor atención para el manejo de la falla detectada ya que, este método no determina por sí solo qué componente fue el que presentó la falla [17].

5.1.3 Redundancia de reserva

Es una técnica de redundancia pasiva de hardware en donde se tiene uno o más módulos inactivos de reserva que, al momento de detectarse una falla en el módulo principal, el módulo de reserva

se activa y puede tomar su lugar. De esta manera, es posible tener múltiples módulos de reserva optimizando el consumo energético, sin embargo, requiere de más tiempo ya que el módulo de respaldo debe de iniciarse al momento. Otro punto necesario para considerar, es la necesidad de implementar algoritmos de detección de errores o de diagnóstico para identificar una falla en el módulo principal o el uso de sistemas externos que lo monitoreen de forma periódica, además, esta técnica tiene limitaciones de fiabilidad, ya que los algoritmos de software no pueden detectar todos los posibles tipos de fallas [17].

5.1.4 Watchdog timer

El *watchdog timer* (WD) funciona como redundancia por hardware activa, ya que es un sistema externo permanentemente encendido que puede garantizar un estado libre de fallas del módulo principal, siempre y cuando este sea capaz de realizar una tarea de forma periódica, como reiniciar el temporizador del sistema redundante, a esta señal de reinicio se le conoce como señal de alivio. La frecuencia de reinicio del temporizador depende de su aplicación. Este sistema se utiliza para corregir fallas de tipo *halt*[4] y *crash*[5], por ejemplo, si durante la ejecución de un código ocurre una falla que coloque al procesador en un bucle infinito, el WD termina su cuenta y reinicia al sistema, sacándolo del bucle [18].

El WD debe configurarse con cuidado, ya que de presentarse una falla permanente entre la conexión del módulo principal y el *watchdog*, la señal de alivio no puede reiniciar el temporizador, aunque el módulo principal no presente una falla se estaría reiniciando de forma indefinida, generando un fallo crítico. Otro caso puede ser que, durante la ejecución del software, este salte a una instrucción cercana a la señal de reinicio del WD, entonces, este no reiniciará el sistema y por lo tanto la falla pasa desapercibida, a este fenómeno se le conoce como *fast watchdog* [20].

5.2 Redundancia por software

Usada para contrarrestar errores en el software, principalmente *bugs*, ya que al diseñar software largo y complejo como lo es FSW es altamente probable la presencia de errores que pueden comprometer la misión [19]. Existen diversas técnicas de redundancia por software, entre las cuales se encuentran: n versiones, verificación de capacidades y manejo de excepciones.

5.2.1 N versiones

Redundancia por software consiste en el desarrollo de las mismas funciones por distintos e independientes equipos de trabajo dedicados al diseño de software, o con diferentes algoritmos que producen el mismo resultado, lo que reduce las posibilidades de cometer el mismo error. De esta manera, se pueden ejecutar diferentes programas de forma concurrente y establecer un sistema de votaciones para determinar errores y evitar su propagación a fallos críticos. Por otro lado, esto incrementa el tiempo y costos de desarrollo de software, y hace más complicado el mantenimiento, además de ser necesario incorporar más memoria y un procesador más rápido para ejecutar dos hilos al mismo tiempo o de manera secuencial y compensar con una mayor velocidad del reloj[19].

[4] *Halt:* falla que ocasiona una pausa no esperada en un programa computacional [9].
[5] *Crash:* falla que detiene y causa un malfuncionamiento total de un programa computacional [9].

5.2.2 Verificación de capacidades

Algoritmos que permiten garantizar que el sistema conserve sus capacidades esperadas, por lo tanto, pueden variar de acuerdo con las necesidades de cada misión. Por ejemplo, en sistemas distribuidos multiprocesador se puede tener un algoritmo que de manera periódica verifique que se mantiene comunicación entre ambos núcleos, de no confirmarse se declara un estado de fallo. Otro ejemplo es la verificación del funcionamiento de una ALU, en este caso se requiere de un proceso que periódicamente realice operaciones con resultados definidos y los compare con resultados almacenados en la memoria persistente [19].

5.2.3 Excepciones

Las excepciones son indicadores de que ha ocurrido un evento inesperado durante la ejecución de software en el cual se ha detectado un error. Este debe de ser manejado para evitar que su propagación acabe en un malfuncionamiento del sistema. Un buen manejo de excepciones puede contribuir a tener un sistema tolerante a fallos y por esto se le debe prestar especial atención. Es responsabilidad del programador generar las excepciones ante cualquier evento inesperado que se presente [19]. Por ejemplo, al intentar leer un archivo pero este no abrió, ocurre una excepción que puede manejarse intentando repetir la operación un par de veces y de no funcionar, reiniciar todo el sistema.

5.3 Redundancia de información

Consiste en agregar información adicional a los datos obtenidos para detectar o enmascarar fallas, esto se realiza comúnmente a través de la codificación y decodificación de información. Existen diferentes tipos de codificación que se pueden implementar según la cantidad y naturaleza de los errores que desean evitarse, algunos de los códigos más comunes se detallan a continuación [21].

5.3.1 Paridad

La forma más sencilla y antigua de codificación para la detección de errores es la del bit de paridad, que consiste en añadir un bit adicional en la transmisión de un dato entre dos dispositivos, por ejemplo, de una memoria al procesador o de un procesador a otro. De manera general funciona al determinar el número de bits con un estado lógico alto (1) y completar la cadena según sea la variante par o impar: **par**, en donde si el número de bits en alto es impar se añade un bit adicional en condición lógica alta (1) y si es par se añade un bit en bajo (0); e **impar**, en donde si el número de bits en alto es impar se añade un bit adicional con un valor bajo (0), mientras que, de ser par se añade un bit en alto (1). Entonces es posible detectar una condición de error al comprobar la cantidad de bits en alto en la cadena completa (incluyendo el bit adicional) y determinar si coincide la condición par/impar según la variante. El bit de paridad es un esquema simple que funciona cuando ha fallado una cantidad impar de bits, ya que en el caso contrario, la falla de un bit puede enmascarar la falla de otro. Hay que tener en cuenta que su implementación no garantiza un sistema tolerante a fallas, ya que solo detecta el error y es necesario añadir un sistema o algoritmo que lo corrija, además, esta codificación solo detecta un estado de error y no el bit que falló, por lo que para la corrección de una cadena es necesario descartarla en su totalidad o solicitar nuevamente su envío [21].

5.3.2 Código de Redundancia Cíclica

El código de redundancia cíclica (CRC, por sus siglas en inglés) es una codificación altamente usada en la verificación de la integridad de la información de comunicaciones o del almacenamiento de un sistema, ya que permite detectar múltiples errores en cadenas de bits. Forma parte de una categoría de codificación llamada códigos cíclicos separables, basados en aritmética de polinomios sobre un campo binario. Esta codificación consiste en representar los datos de un mensaje como un polinomio *d(x)* en donde cada bit es un coeficiente, seleccionado un polinomio generador *g(x)* de grado r predefinido por estándares como el CRC-12 $\left(1 + x + x^2 + x^3 + x^{11} + x^{12}\right)$, CRC-16 $\left(1+x^2+x^{15}+x^{16}\right)$, o CRC-32 $\left(1+x+x^2+x^4+x^7+x^8+x^{10}+x^{11}+x^{12}+x^{16}+x^{22}+x^{23}+x^{26}+x^{32}\right)$. Posteriormente se recorre l posiciones el polinomio *d(x)* añadiendo ceros a la derecha obteniendo el polinomio *d'(x)* que se divide entre *g(x)* obteniendo un residuo *r(x)*, el cual se suma al polinomio desplazado *d'(x)* obteniendo el mensaje codificado *c(x)*, este mensaje será el que se transmitirá o guardará para detectar errores al dividir dicho polinomio entre *g(x)*. Hay un error si el residuo de la operación es diferente de cero, ya que *c(x)* es divisible exactamente por *g(x)*, por lo que su residuo debería ser cero y cualquier cambio en el mensaje afecta el resultado de la división, permitiendo detectar todos los errores en ráfaga en una longitud menor o igual a l [21].

5.3.3 Duplicidad de información

Otra forma de redundancia de información es la duplicación de la misma, evitando la necesidad de implementar algún tipo de codificación, y comparando la información antes de ser usada para detectar cambios en ésta, sin embargo, no suele ser una opción viable ya que requiere de un aumento significativo de la capacidad de almacenamiento, que en sistemas embebidos es poco viable por las limitantes de tamaño [18].

5.4 Redundancia en tiempo

Cuando las restricciones de un sistema impiden agregar hardware adicional o redundancia de software (que muchas veces requiere también de más hardware), se puede depender del tiempo de ejecución para detectar y corregir fallas, lo que tiene un impacto en el desempeño pero que es posible contrarrestar con procesadores más veloces, que los usados en sistemas no redundantes, o cuando el sistema no tiene que cumplir con un esquema de tiempo estricto o de tiempo real [21].

5.4.1 Repetición de operaciones

Consiste en realizar la misma operación o transmisión de datos más de una vez, almacenando los resultados en memoria y comparándolos entre sí para determinar si coinciden. Este tipo de redundancia es muy efectiva para detectar fallas transitorias, ya que al existir una, los resultados serán distintos. En caso de querer detectar cuál fue el resultado correcto, es posible realizar la operación una tercera vez. [17].

6 Nanosatélites

Desde los inicios de la exploración espacial, una de las principales metas era colocar un objeto en órbita con el objetivo de mantener instrumentación como cámaras, sensores, radares y antenas de comunicación en el espacio exterior, creando una infraestructura espacial. Esto se consiguió el 4 de octubre de 1957 cuando un cohete R-7 despegó del corazón de la unión soviética con una esfera de 58 cm de diámetro y una masa de 83.6kg, el Sputnik I, siendo el primer satélite artificial puesto en órbita que marcaría el inicio de la carrera espacial [22].

Se entiende por satélite a cualquier cuerpo que orbite alrededor de un cuerpo celeste en el espacio exterior. Pueden ser satélites naturales como la Luna orbitando la Tierra; o artificiales, que son máquinas construidas y lanzadas al espacio por el ser humano como el Sputnik I o la Estación Espacial Internacional (ISS, por sus siglas en inglés). Los satélites artificiales pueden ser de distintas formas y tamaños, con diferente tipo de instrumentación a bordo para cumplir con los objetivos de su misión. El desarrollo de un satélite artificial puede durar meses o incluso años según su complejidad, además, deben pasar por estrictas pruebas para garantizar que sobrevivan el agresivo lanzamiento y las duras condiciones del espacio exterior [23].

En el inicio de la exploración espacial, los primeros satélites como el Sputkink I y el Vanguard I, estaban limitados en tamaño y masa, y por lo tanto en funcionalidad, debido a las restricciones establecidas por los vehículos lanzadores de esa época. Sin embargo, a medida que mejoró la tecnología de los lanzadores, con el objetivo de cumplir con las especificaciones de cada misión los satélites se hicieron cada vez más grandes, pesados y por lo tanto costosos [24].

6.1 Pequeños satélites

Debido a que el desarrollo, operación y mantenimiento de satélites de gran tamaño presentan un reto significativo, las organizaciones que pueden adquirirlos se limitan a: gobiernos, instituciones militares y grandes empresas de telecomunicaciones [25]. En la actualidad, gracias al desarrollo de nuevas tecnologías que permiten reducción del tamaño de la electrónica, es posible integrar grandes capacidades computacionales con un bajo costo y consumo de potencia a los satélites pequeños (especialmente para LEO), sin comprometer sus funciones [24].

El uso de los componentes COTS ha favorecido la popularidad de los pequeños satélites, siendo en 1981 su primer uso en satélites con el UoSat-1 de la Universidad de Surrey [26], ya que la electrónica COTS es muy económica y de alta disponibilidad en el mercado, y aunque sean más vulnerables a la radiación que sus contrapartes de grado espacial, pueden ser fiables si se complementan con componentes blindados a la radiación o implementando técnicas de tolerancia a fallas [2], haciendo posible que en la actualidad se puedan comprar cada uno de los componentes necesarios, tanto de software como de hardware para construir un satélite pequeño desde sitios web, abriendo la posibilidad a que cualquier institución, con los recursos suficientes, pueda integrar y lanzar un satélite pequeño propio [6].

La clasificación de pequeños satélites difiere un poco entre instituciones y países, para este trabajo se considera como pequeño satélite a todo aquel que tenga una masa menor a 500kg, y se

tomará en cuenta la clasificación propuesta por Kulu E. [5] para nanosatélites, con todo satélite con una masa localizada entre 1kg y 10kg, además de *CubeSats* de 1 a 27U con masa de hasta 40kg, *PocketQubes* de entre 100 y 250g, *TubeSats* de hasta 750g, *SunCubes* de 35 g por unidad y *ThinSats* con una masa aproximada de 280g.

6.2 CubeSat

El estándar *CubeSat* surgió en 1999 con un proyecto colaborativo entre la Universidad Estatal Politécnica de California y la Universidad de Standford, con el objetivo de reducir los costos y los tiempos de desarrollo, aumentar la accesibilidad al espacio y mantener lanzamientos frecuentes. El estándar establece el tamaño y la forma del satélite como un cubo de 10 cm de arista y una masa no mayor a 2kg, conocido como unidad (U), las cuales son apilables para adaptarse a los objetivos de una misión [27].

Además, el estándar define las especificaciones para su integración y lanzamiento, incluyendo la forma y mecanismos de despliegue, conocidos como dispensadores o lanzadores. Esto permite a diferentes instituciones lanzar sus satélites de forma sencilla y económica a través de los *piggybag launch*, un método de lanzamiento en el cual la carga útil principal de la misión suele ser un sistema de gran tamaño absorbiendo la mayoría de los costos, pero abriendo la posibilidad de montar lanzadores pequeños [26].

6.3 Computadora de a bordo

Para que un satélite cumpla con sus objetivos debe ser capaz de administrar su carga útil (como por ejemplo cámaras, sensores y antenas), procesar la información obtenida durante la misión y administrar la recepción y envío de telemetría a la Tierra. Todo lo anterior es gestionado por la computadora de a bordo (OBC, por sus siglas en inglés), también llamado sistema de comando y manejo de datos [2].

De manera general las OBC suelen mantener una arquitectura común (véase la figura 6.1), exceptuando la de vehículos lanzadores y misiones humanas, cumpliendo con las funciones descritas a continuación [4]:

- ☐ Procesamiento: controladores de hardware y un sistema operativo para ejecutar el FSW.
- ☐ Almacenamiento: almacena telemetría, datos operativos críticos y de la carga útil.
- ☐ Gestión de tiempo: debe proporcionar una referencia de tiempo global para todos los subsistemas del satélite y las cargas útiles.
- ☐ Comunicación con submódulos: debe interconectar todos los subsistemas de la nave incluyendo la carga útil.
- ☐ Comunicación con Tierra: debe gestionar un enlace bidireccional de comunicación con la estación terrena, envía la telemetría y procesa los comandos recibidos.
- ☐ Interfaces de comunicación: debe proporcionar interfaces de comunicación para sensores y actuadores.

Existe una gran variedad de estándares establecidos por agencias espaciales e instituciones privadas con el objetivo de garantizar la compatibilidad entre módulos comerciales, software de vuelo y la comunicación con estaciones terrenas, entre otros. De esta manera, se reducen los costos y el tiempo de desarrollo, posibilitando la colaboración entre distintas instituciones. Por ejemplo, los estándares desarrollados por el Comité Consultivo para Sistemas de Datos Espaciales (CCSDS, por

sus siglas en inglés), que define los protocolos para la comunicación con satélites, estableciendo la estructura de tramas para enviar comandos desde la estación terrena, así como el protocolo de transmisión de la telemetría del satélite [4].

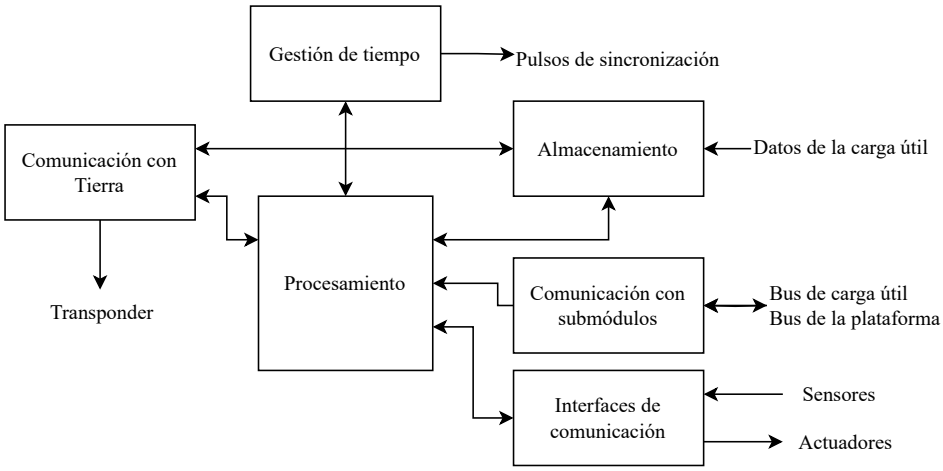


Figura 6.1: Arquitectura común de una OBC (adaptado de [4]).

Existen diversas opciones al momento de seleccionar una unidad de procesamiento que para satélites pequeños a menudo se utilizan componentes COTS con el objetivo de reducir los costos de desarrollo, utilizando microcontroladores, arreglos de compuertas programables en campo (FPGA, por sus siglas en inglés), y recientemente sistemas en chip (SoC, por sus siglas en inglés) [2].

De acuerdo con la misión, en ocasiones las exigencias computacionales podrían sobrecargar un microcontrolador, provocando cuellos de botella, latencia e incluso la inestabilidad del sistema, por lo que se usan SoC integrados con múltiples núcleos, unidades de procesamiento gráfico (GPU, por sus siglas en inglés) y unidades de gestión de memoria (MMU, por sus silgas en inglés), entre otros componentes [2]. Un ejemplo de estos sistemas son las llamadas *single board computer* o computadoras mono placa o embebidas, que son sistemas completos con microprocesador, memorias, puertos y demás funciones necesarias para tener una computadora funcional, con la capacidad de ejecutar sistemas operativos y grandes cargas de procesamiento a un bajo costo energético y económico, además de tener un tamaño reducido en comparación con computadoras de escritorio o portátiles [8].

7 Software de vuelo para nanosatélites

Con el paso del tiempo, los satélites se han hecho más dependientes de software para cumplir con sus funciones que, en un principio, eran implementadas con electrónica analógica y lógica cableada, permitiendo realizar tareas más complejas en arquitecturas flexibles y reconfigurables [28].

Se le conoce como software de vuelo (FSW, por sus siglas en inglés) al programa que se ejecuta en la OBC del satélite, el cual es responsable de administrar todos los recursos de hardware e interfaces de la OBC para lograr los objetivos establecidos para la misión, como el control de orientación de la nave espacial y el procesamiento de datos y control de cargas útiles, además de gestionar la comunicación con la estación terrena, permitiendo operar y supervisar el sistema durante la misión [28].

Debido a que el FSW condiciona las pruebas del funcionamiento del hardware es fundamental que sea desarrollado en el menor tiempo posible [28]. Esto supone un reto para los programadores, al tener que desarrollar un software fiable que cumpla con todas las características necesarias para satisfacer las especificaciones de una misión, en un tiempo corto. Para darle solución a estos problemas, se usan herramientas como los *framework* [29].

Se entiende por *framework* al conjunto herramientas que facilitan el diseño reutilizable de software. El *framework* establece la arquitectura y estructura general que seguirá una aplicación, así como sus clases, objetos, flujo de control y de datos, por lo que facilitan la reutilización de código. Como consecuencia se agiliza el desarrollo de aplicaciones, su mantenimiento y entendimiento para el usuario, cumpliendo con una estructura estándar [30].

Agencias espaciales, universidades y empresas del sector espacial han creado *frameworks* de código abierto con el objetivo de fomentar la reutilización de código, así como definir diseños de referencia para el desarrollo de software de vuelo en sus proyectos, resultando en una disminución en los tiempos y los costos de desarrollo, además de una calidad general superior. Uno de los *framework* para misiones espaciales con herencia de vuelo, constantemente actualizado y mantenido, que cumple con estándares internacionales para sistemas espaciales, como CCSDS, es el cFS desarrollado por la NASA, el cual se ejecuta desde un sistema operativo compatible [29].

Un sistema operativo (OS, por sus siglas en inglés) es un software que controla la ejecución de programas. Además, proporciona herramientas o servicios que facilita la interacción entre el usuario y la computadora [31].

El procesamiento en tiempo real tiene como concepto clave el tiempo. El sistema debe de responder a eventos o tareas dentro de un intervalo de tiempo estricto, de no cumplir, ya sea con una acción temprana o con retardo, se compromete el resultado [32].

Los sistemas operativos en tiempo real (RTOS, por sus siglas en inglés) presentan cuatro características importantes: deben tener una **latencia mínima** en las interrupciones, es decir, debe transcurrir la menor cantidad de tiempo posible entre la llegada de una interrupción al sistema y su procesamiento. Tienen **regiones críticas cortas**, los OS dependen de regiones críticas para

proteger datos compartidos y poder sincronizar procesos, para un RTOS, estas regiones deben ser lo más pequeñas posibles. Necesitan contar con una **capacidad de desalojo** que permita a una tarea de mayor prioridad interrumpir a una de menor prioridad en cualquier instante. una total libertad de controlar con precisión la prioridad de las tareas. Por último, necesitan contar con un excelente **algoritmo de planificación de tareas** para evitar que las tareas de mayor prioridad se vean retrasadas por las de menor prioridad [33].

8 Core Flight System

Core Flight System (cFS) es un *framework* modular y multiplataforma de código abierto desarrollado por la NASA para el desarrollo de software de a bordo para misiones espaciales. Proporciona una arquitectura que se ejecuta sobre RTOS u otros OS convencionales (como Linux) mediante capas de abstracción. Tiene sus orígenes en 2004 cuando ingenieros de software del centro espacial *Goddard* trabajaban en dos misiones de forma simultánea, sin tiempo ni personal suficiente decidieron encontrar puntos en común entre ambas misiones desarrollando un *framework* reutilizable, implementando el *Core Flight Executive* (cFE) como un punto de partida. Desde entonces, aquella división continúa usando el *framework* para cada misión, debido a su buen rendimiento y a las ventajas que ofrece, tales como una alta facilidad de uso y gran calidad. Con el paso del tiempo, el software ha recibido mejoras constantes y ha sido liberado como código abierto, lo que le ha hecho ganar popularidad no solo en toda la NASA, sino en todo el mundo al ser usado por distintas agencias espaciales e instituciones públicas y privadas [34].

Una de las mayores virtudes de cFS es su portabilidad, la cual le permite ejecutarse en cualquier arquitectura siempre que esta tenga los recursos de hardware suficientes[6] para ejecutar un OS compatible, que incluyen: sistemas operativos bajo el estándar de interfaz de sistemas operativos portátil para Unix (POSIX, por sus siglas en inglés) como Linux y algunos RTOS como RTEMS y VxWorks. Esto gracias a que cFS está diseñado como un sistema multicapas 8.1, en donde cada una se comunica únicamente con la capa adyacente. Estas capas incluyen el hardware, el OS, la capa de abstracción del sistema operativo (OSAL, por sus siglas en inglés), Core Flight Executive (cFE) y las aplicaciones de usuario [7].

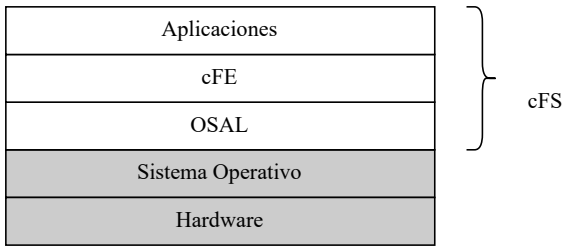


Figura 8.1: Capas de cFS (adaptado de [7]).

8.1 OSAL

La capa de abstracción del sistema operativo (OSAL, por sus siglas en inglés) es una biblioteca que proporciona funciones para manejar la memoria, gestionar hilos, sincronizar tareas (semáforos, mutexes), manejo de archivos, entre otras. Esto permite abstraer las funciones nativas del sistema operativo a funciones genéricas de OSAL. Por ejemplo, para crear un hilo en Linux se usa utilizan las funciones *pthread_create* y *task_spawn* para VxWorks. Mientras que en cFS basta con usar la interfaz de programación de aplicaciones (API, por sus siglas en inglés) de la OSAL con la función ***OS_TaskCreate***, para ser compatible con ambos OS [35].

[6] Al ser muy flexible cFS, no existen especificaciones mínimas de hardware concretas. Sin embargo, se ha normalizado a 16MB y 500MHz para garantizar un buen desempeño, pero dependerá en su mayoría de la misión.

Esta capa hace posible que cFS sea portable a toda plataforma que sea compatible con cualquier sistema operativo soportado, permitiendo trabajar en sistemas embebidos o computadoras de escritorio sin necesidad de modificar el software [35].

El repositorio oficial de la NASA incluye el OSAL que permite ejecutar cFS en OS bajo el estándar POSIX (como Linux o macOS) y para los RTOS RTEMS y VxWorks. Algunas instituciones han complementado el OSAL para ejecutar cFS sobre FreeRTOS, sin embargo, al ser desarrollos por organizaciones independientes pueden tener bastantes errores [34].

8.2 Core Flight Executive (cFE)

El cFE es el núcleo del *framework* de cFS. Crea un entorno de ejecución para aplicaciones, solicitando a la capa OSAL la asignación de un hilo para cada tarea. Además, ofrece distintas funciones y servicios que suelen ser comunes para la mayoría de FSW, sentando las bases para su desarrollo [36].

El cFE está conformado por cinco servicios básicos: gestión de aplicaciones (*Executive Service*), comunicación entre procesos (*Software Bus*), registro de eventos (*Event Service*), manejo de tablas de configuración (*Table Service*) y sincronización temporal (*Time Service*). Cada servicio cuenta con un proceso de ejecución específico y con un conjunto de interfaces de programación de aplicaciones API disponibles para todas las aplicaciones como biblioteca externa. Su arquitectura modular facilita el desarrollo de FSW confiable y mantenible, promoviendo la reutilización de código entre diferentes misiones [36].

8.2.1 Executive service (ES)

El *Executive Service* es el servicio encargado de la gestión de aplicaciones y recursos del sistema. Su objetivo principal es proporcionar un conjunto organizado de reglas, interfaces y procedimientos para garantizar el funcionamiento coordinado y confiable de las aplicaciones, la administración de memoria y la supervisión del estado del sistema. De forma más específica tiene tareas como inicializar el cFE (o reiniciarlo de ser necesario) cargar, iniciar, detener o reiniciar todas las aplicaciones. Proporciona servicios de memoria dinámica y gestiona el almacenamiento de datos críticos (CDS, por sus siglas en inglés), permitiendo el monitorear y diagnosticar el sistema proporcionando información sobre el estado del CPU y uso de memoria. Registra eventos a través de una bitácora (*System log*) y proporciona herramientas de sincronización como lo son semáforos, colas y mutexes, lo que permite implementar sistemas de procesamiento concurrente eficiente y seguro [36].

8.2.2 Software Bus (SB)

El *Software Bus* forma parte de los servicios básicos de cFE y es el medio de comunicación entre aplicaciones basado en un modelo de mensajería de *Publisher-Suscriber*. Este es un modelo de comunicación asíncrona que permite una comunicación escalable y fiable entre módulos o bloques independientes. Consta de distintos componentes [37].

- ☐ Mensajes: conjunto de datos transmitidos entre aplicaciones.
- ☐ Canales: son colas de recepción que cada bloque crea para recibir mensajes.
- ☐ Temas: identificador que agrupa todos los mensajes de un mismo tipo.
- ☐ Suscriptores: aplicaciones interesadas en recibir el mensaje de algún tema.
- ☐ Emisores: cualquier aplicación que envíe un mensaje.

El emisor no necesita saber a quién le envía la información que transmite, mientras que los suscriptores no necesitan conocer el origen del mensaje, lo que permite un desacoplo entre aplicaciones o módulos. Para cFS, una app crea un canal o pipe especificando la máxima profundidad, después, la app indica qué *msgID* o tema quiere recibir y de esta forma queda configurado el suscriptor, mientras que el emisor llama a la API de SB para enviar un mensaje, por lo que no es necesario indicar la app de destino. Posteriormente la app de SB realiza copias del mensaje en cada canal suscrita al tema [36].

8.2.3 Event Service (EVS)

La aplicación de *Event Service* forma parte de los servicios del cFE que gestiona eventos, como errores, detectados por distintas aplicaciones, los eventos pueden clasificarse en cuatro tipos de acuerdo con su nivel de impacto en el sistema [36]:

1. *Debug*: usados solo en etapa de desarrollo y depuración.
2. *Information*: identifican un cambio de estado o acción que no es un error.
3. *Error*: identifican un error, posible de corregir.
4. *Critical*: identifican un error catastrófico, incorregible sin intervención.

EVS permite controlar qué aplicaciones pueden registrar eventos y qué tipo de eventos de cada aplicación son los que generan el mensaje. Todos los eventos generan mensajes estandarizados que son enviados al sistema de telemetría y registros locales en la bitácora de eventos (*Event Log*) [36].

Los eventos llegan en forma de mensaje enviados a través de SB a cualquier aplicación que esté suscrita al tema (o *message ID*), incluyendo el siguiente contenido [36]:

1. Marca de tiempo.
2. Tipo de evento.
3. ID de la nave espacial.
4. ID del procesador.
5. Nombre de la aplicación.
6. ID del evento.
7. Mensaje.

8.2.4 Time service (TIME)

Proporciona servicios de tiempo (como API) para conservar una sincronización en el tiempo para todas las aplicaciones y servicios, mantiene un registro del tiempo transcurrido de la misión, permite sincronizar el tiempo de la nave con el tiempo de la estación terrena, distribuye de forma periódica paquetes a 1Hz con la hora correcta a toda aplicación que se suscriba [36].

8.2.5 Table service (TBL)

Algunas aplicaciones requieren de parámetros como ganancias para el control de orientación de la nave espacial o factores de conversión, entre otros. TBL proporciona una serie de comandos que le permiten al usuario cargar y descargar tablas desde archivos. Además, proporciona los servicios necesarios para su validación y garantizar que la tabla a ser cargada no se encuentra dañada antes de utilizarla [36].

8.3 *Capa de aplicaciones*

Esta es la capa del nivel superior de cFS en donde existen las bibliotecas y aplicaciones que, con apoyo de los servicios de cFE y OSAL, permiten que el sistema cumpla con los objetivos de la misión.

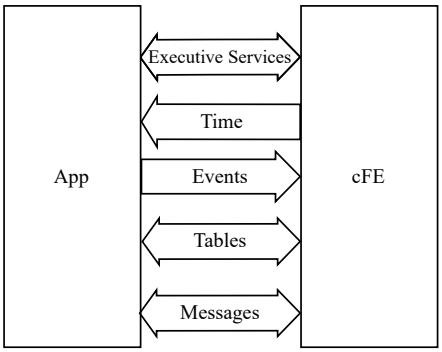


Figura 8.2: Interfaz app y cFE

Las bibliotecas son un conjunto de funciones reutilizables que operan dentro de la aplicación que las llama, es decir, no cuentan con un proceso propio [36].

Las aplicaciones de cFS son módulos de software compilados como librerías compartidas u objetos dinámicos (`.so` para Linux), que implementan funciones como el envío de telemetría, recepción de comandos, manejo de la carga útil, entre otros, mediante tareas gestionadas por OSAL. Aunque su formato es el de librerías, funcionan como ejecutables independientes gracias a cFE, que carga sus puntos de entrada en el espacio de memoria del proceso principal y les asigna recursos propios. Este modelo permite la recarga dinámica en vuelo, aislamiento de fallas y la portabilidad entre plataformas [36].

Las aplicaciones de cFS mantienen una arquitectura similar a la mostrada en la figura 8.3; que incluye la inicialización de la app, en donde se crean los canales de comunicación de SB, y posteriormente se realizan los procesos de acuerdo con sus necesidades, ya sea durante el proceso de un comando o la ejecución de un proceso periódico definido por la aplicación de *scheduler* [36].

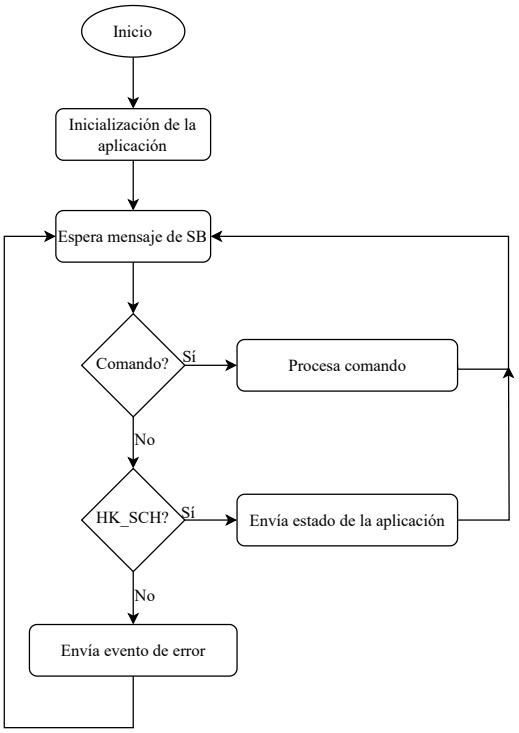


Figura 8.3: Estructura de aplicaciones de cFS.

Con el fin de mantener la portabilidad, las aplicaciones deben apegarse al estándar establecido por el *framework* de cFS usando las API de cFE y OSAL (figura 8.2), evitando depender de un sistema operativo o hardware específico. El modelo de mensajería entre aplicaciones es estandarizado por SB. Cada aplicación es cargada durante el arranque del sistema siguiendo un orden establecido por el archivo de configuración `cfe_es_startup.scr` y permitiendo la posibilidad de recargar o actualizar cada aplicación de forma individual sin tener que reiniciar todo el sistema [36].

8.4 *cFS Bundle Pack*

El *cFS Bundle Pack* es la distribución oficial de cFS desarrollado y mantenido por la NASA (disponible en el repositorio de GitHub [34]), la última versión oficial es la 6.7.0a (Aquila), publicada en julio de 2020, sin embargo, la NASA ha anunciado el lanzamiento de una nueva versión **Draco** planeada para 2025. Actualmente existe una versión preliminar llamada draco-rc5. Estas distribuciones contienen el núcleo del *framework* (cFE), la capa de OSAL (para OS del estándar POSIX, RTEMS y VxWorks), herramientas como *Ground System*, que es una interfaz gráfica para enviar comandos y recibir telemetría, los archivos necesarios para compilar y ejecutar el sistema, que incluye el CMake y archivos de ejemplo de arranque (`.scr`) y aplicaciones comunes que incluyen: CI, TO y SCH [7].

- **Scheduler (SCH)**: permite generar mensajes (a través de SB), en intervalos de tiempo predeterminados, con el objetivo de que el FSW opere de forma multiplexada por división de tiempo (TDM, por sus siglas en inglés) con un comportamiento determinista, es decir, para las mismas entradas y condiciones iniciales se obtienen resultados idénticos en el mismo orden e instante de tiempo. Para esto se tienen dos marcos: el mayor y el menor. En el marco mayor se tiene una señal de sincronización gestionada por el servicio *TIME* de cFE que es por lo regular de 1Hz. Mientras que, el tiempo del marco menor es totalmente configurable, eligiendo el número de ranuras ejecutadas en cada marco mayor para ajustar la granularidad de la planificación [38].
- **Command ingest (CI):** tiene el objetivo de recibir comandos bajo el estándar CCSDS desde una fuente externa a través de un puerto UDP/IP, como lo puede ser la estación terrena, y los reenvía a la aplicación correspondiente a través del SB [7].
- **Telemetry output (TO):** se encarga de enviar los paquetes de telemetría, como eventos, o mensajes de *Housekeeping app (HK)*, hacia un sistema externo por medio de un puerto UDP/IP, como lo es la estación terrena [7].

Además de SCH, CI y TO, existe una gran variedad de aplicaciones *open source* disponibles en el repositorio de la NASA, mostradas en la tabla del anexo B.1 [36].

8.5 *Tolerancia a fallas en cFS*

Implementar cFS en un software de vuelo no garantiza obtener un sistema tolerante a fallas, sin embargo, el *framework* proporciona una gran cantidad de herramientas que permiten tener cierta confiabilidad, siendo responsabilidad del diseñador plantear una arquitectura según las necesidades de su misión para alcanzar los niveles de fiabilidad deseados. Algunas de las técnicas de tolerancia a fallas que se pueden implementar con cFS incluyen la redundancia de hardware, el monitoreo de aplicaciones y el manejo de excepciones, así como la detección y el registro de errores [36].

cFS cuenta con funciones que permiten tener más de un procesador trabajando, ya sea de forma activa para implementar un sistema de votación y enmascaramiento de errores, o bien de forma pasiva, de modo que en el momento de declararse que el procesador principal presenta fallas permanentes, se cambie a un procesador de reserva. Casi todas las funciones de cFE devuelven el estado de la operación, indicando que se realizó correctamente o que existió un error. En el caso de presentarse una condición de error, se recibe un código que indica la gravedad o posible causa, a partir del cual el programador puede desarrollar algoritmos para corregirlo y recuperar la funcionalidad del sistema [36].

En caso de que alguna aplicación no responda, se genera una excepción que atiende cFE, con la cual se reiniciará la aplicación o el procesador, corrigiendo las fallas transitorias. Sin embargo, en ocasiones pueden ocurrir fallas permanentes como los SEU que podrían comprometer la integridad de la información, lo que resulta en la necesidad de una intervención para corregir la falla [36].

cFS proporciona aplicaciones (véase la figura B.1) que permiten reemplazar los archivos dañados por respaldos a bordo o transmitidos desde la estación terrena. Para esto se sigue el proceso mostrado en la figura 8.4, en donde una aplicación presenta una falla que resulta en el intento de reinicio de la app, que de estar corrupto el archivo ejecutable, se presentará una falla que será reportada como un evento de tipo error, que al ser recibido en la estación terrena podrá enviarse un respaldo para reemplazar el archivo dañado y permitir la ejecución de la aplicación [36].

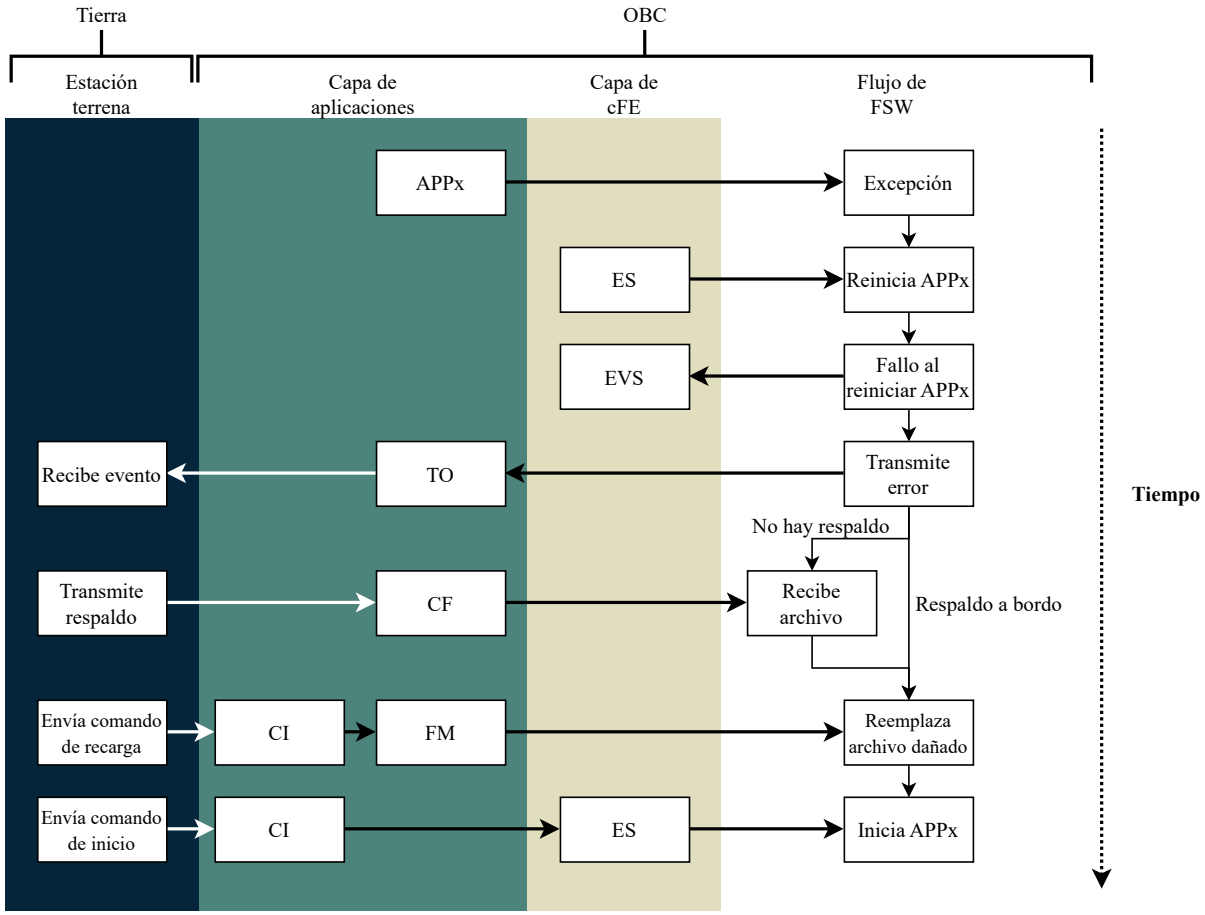


Figura 8.4: Proceso de reconfiguración de aplicaciones en cFS.

Las técnicas de tolerancia a fallas en cFS son una parte fundamental para el diseño de cualquier FSW. Sin embargo, antes de corregir cualquier error, es necesario detectarlo. Par esto cFS cuenta con diversos mecánismos de monitoreo de fallas, como la implementación de algoritmos para la validación de memoria y archivos por medio de CRC [36].

9 Estado del arte

El estado del arte presenta una revisión concreta y crítica del contexto más reciente y relevante para comprender la situación actual de un tema en específico, además de ser el punto de partida en el proceso de diseño, ya que establece una referencia para determinar los requisitos del sistema.

La primera sección examina la tendencia global de los satélites pequeños y sus aplicaciones, mientras que la segunda sección reporta las OBC con componentes COTS utilizadas en sistemas espaciales, y la tercera sección revisa las misiones o proyectos que han usado el *framework* cFS como base para su FSW.

9.1 Tendencia global de satélites pequeños

De acuerdo con un estudio realizado en 2017 [26], de 2500 satélites lanzados en los últimos 20 años cerca de un tercio fueron satélites pequeños con una masa menor a 500kg. Los satélites pequeños lanzados en los últimos 5 años representan una cantidad similar a los acumulados en los 15 años anteriores, lo que demuestra una tendencia en la reducción del tamaño de los satélites. Una de las muchas razones de esta tendencia es el aumento general del desempeño de las computadoras electrónicas, con la capacidad de realizar una mayor cantidad de operaciones por segundo, en una menor área de construcción, un menor costo monetario y un menor consumo energético.

Los satélites pequeños han democratizado el acceso al espacio exterior, ya que gracias a sus bajos costos y cortos tiempos de desarrollo permiten a universidades, *startups* y países en vías de desarrollo participar en misiones espaciales, resultando en un aumento de inversión pública y privada a nivel mundial de 4,900 millones de dólares en 2022, esperando una tasa de crecimiento anual compuesta de 14.6 %, equivalente a 18,500 millones de dólares para 2032, abriendo oportunidades en toda la industria dedicada al diseño, producción y lanzamiento de satélites pequeños [6].

Para el 31 de diciembre 2024, de acuerdo con Kulu E. [5] (véase figura 9.1), se han lanzado 2806 nanosatélites, comprendiendo a todos los *CubeSat* de 0.25 a 27U, *PocketQubes*, *TubeSats*, *SunCubes*, *ThinSats* y pico-satélites no estándar, con una predicción de más de 1700 nanosatélites para los próximos 5 años. Se puede apreciar en la figura 9.2 cómo el estándar *CubeSat* es el dominante en nanosatélites con un porcentaje acumulado del 93 % entre 0.24 y 27U, en donde tan solo los *CubeSat* 3U representan 42.7 % del total lanzado.

En la figura 9.3 se muestra el tipo de organización que ha lanzado nanosatélites, siendo el 56.4 % la industria privada y en segundo lugar con 28.8 % las universidades, dando un total acumulado del 85.2 %, lo que representa 2,373 nanosatélites lanzados.

9.2 OBC con componentes COTS

Periódicamente, la NASA publica un informe técnico que de forma general reporta un panorama del estado actual de las nuevas tecnologías y avances para el desarrollo de satélites de tamaño pequeño. La edición 2023 del *State of the Art Small Spacecraft Technology* [39], reporta que para OBC de satélites pequeños, como *CubeSats*, el costo y la disponibilidad de componentes en el mercado

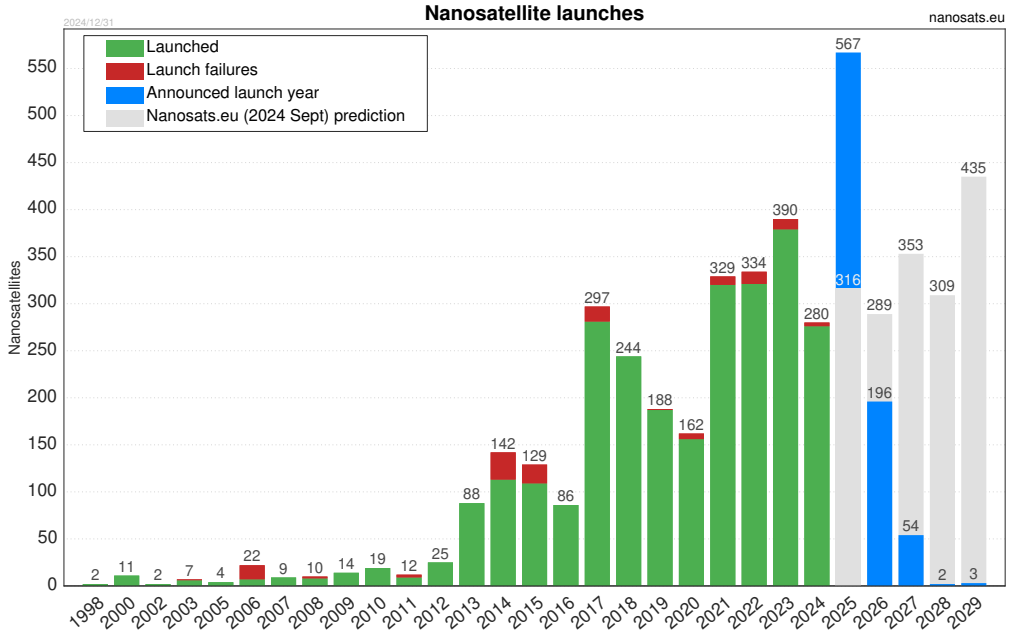


Figura 9.1: Lanzamientos de nanosatélites hasta el 2024, con predicciones hasta el 2029 [5].

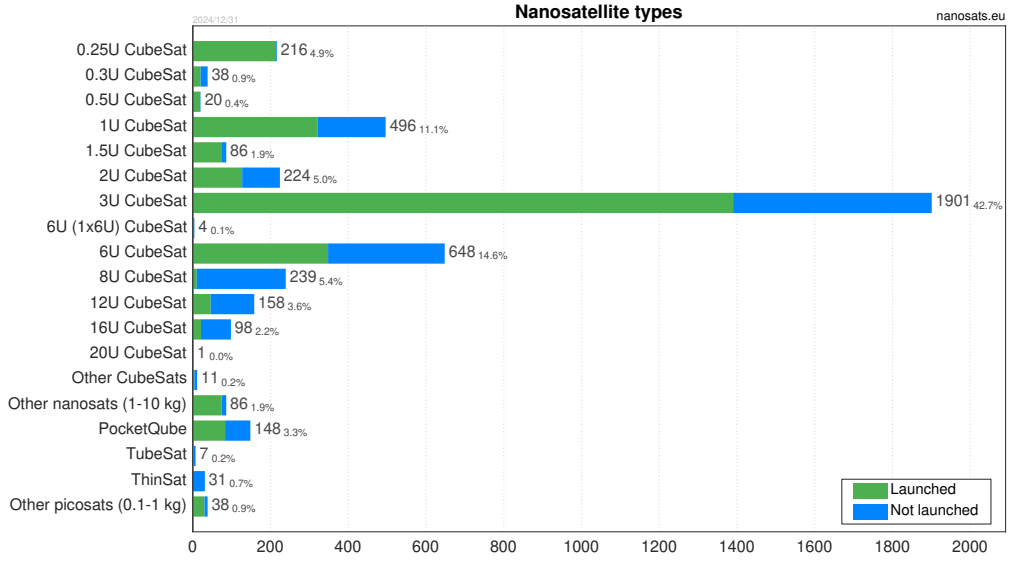


Figura 9.2: Tipos de nanosatélites lanzados hasta 2024 [5].

son parámetros esenciales al momento de seleccionar una computadora, además de puntualizar que los componentes COTS se encuentran varias generaciones por delante de sus contrapartes de grado espacial, a costa de ser más susceptibles a presentar fallos como consecuencia de su exposición a la radiación espacial. En el informe se menciona a cFS como uno de los *framework* más usados para el desarrollo de FSW, en conjunto con OS como VxWorks, RTEMS, FreeRTOS y Linux. Además, se puede observar en la gráfica de la figura 9.4 que de un total de 61 OBC, el 12 % emplean componentes de grado espacial o con garantía de resistencia a la radiación (RHA, por sus siglas en inglés), el 21 % utilizan componentes COTS y el 44 % componentes de grado industrial y automotriz, con un cierto nivel de tolerancia a la radiación.

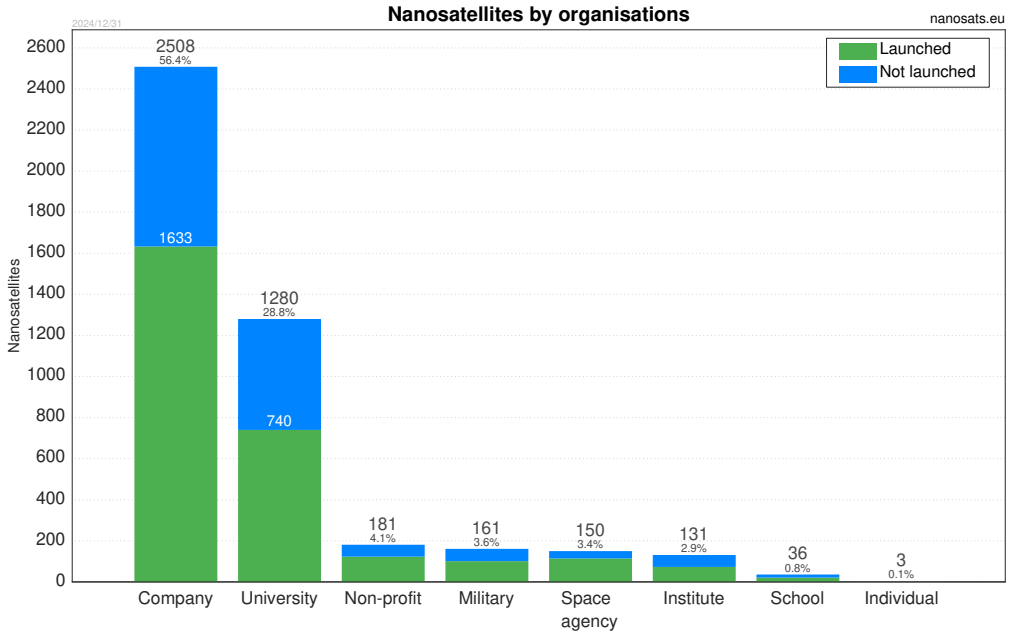


Figura 9.3: Tipos de organizaciones que lanzaron satélites hasta 2024 [5].

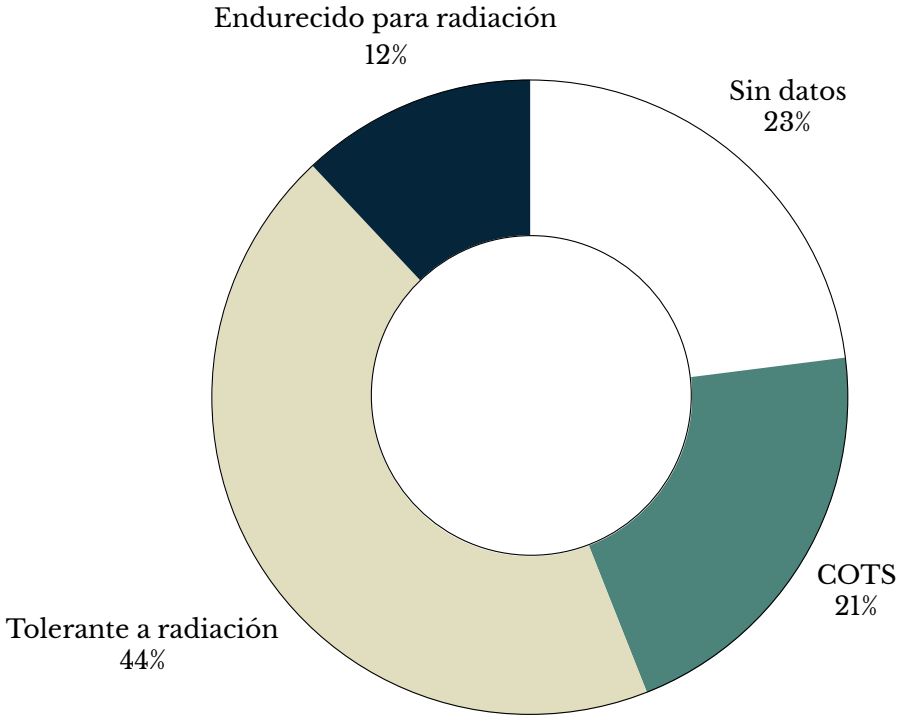


Figura 9.4: Clasificación de las OBC, de acuerdo con el grado de sus componentes.

Un equipo de la Universidad Técnica de Viena (*TU Wien Space Team*), como parte de la misión *SpaceTeamSat 1* (STS-1), desarrollaron un *CubeSat* de 1U para operar en LEO. Para su OBC usaron una Raspberry Pi 4B. Sin embargo, externaron una preocupación por la vulnerabilidad de la memoria *flash* integrada de tipo tarjeta multimedia (eMMC, por sus siglas en inglés), concluyendo que es necesario implementar un *reset* y reconfiguración externa como trabajo a futuro [40].

9.3 Core Flight System (cFS) en misiones espaciales

En 2016 la NASA publicó un reporte de fiabilidad tras un análisis de nueve versiones del *framework* cFS, encontrando que las versiones que presentan un mayor decaimiento tienen como causa del problema las plataformas de prueba y el uso de componentes COTS [41].

En 2019, un equipo liderado por la empresa Visiona, en conjunto con otras empresas e institutos de ciencia y tecnología de Brasil, realizaron la comparación de seis *frameworks* tomando en cuenta siete criterios: documentación, herencia de vuelo, huella de recursos, calidad, mantenimiento a largo plazo, comunidad de usuarios y estandarización CCSDS, con el fin de seleccionar el más adecuado para la misión VCUB1 que consta de un *CubeSat* 6U para LEO, de bajo costo con una OBC COTS, para la cual cFS de la NASA obtuvo la mayor puntuación y por lo tanto fue el *framework* seleccionado [29]. De acuerdo con el sitio web oficial de Visiona, el 15 de abril de 2023 fue lanzado con éxito al espacio y después de un año de maduración y desarrollo de sus sistemas, las primeras imágenes tomadas por el satélite fueron recibidas confirmando el éxito de la misión.

La *Ebry-Riddle Aeronautical University* diseñó e implementó una cámara desplegable en 2019 llamada *EagleCam* con el objetivo de capturar el aterrizaje en la superficie lunar de Nova-C Lander desde otro punto de vista por medio de una estructura *CubeSat* de 1.5 U; este sistema usa el cFS 6.5 como FSW para su OBC, usando las aplicaciones proporcionadas por el *cFS Bundlepack*, que incluye el núcleo cFE y las aplicaciones: SCH, TO y CI. Además, desarrollaron 4 aplicaciones propias de uso específico para la misión. El cFS se ejecuta sobre un OS Ubuntu Linux 18.04 en una NVIDIA Jetson TX2 [42]. En febrero de 2024, se llevó a cabo el lanzamiento de la misión, que de acuerdo con la página oficial del equipo reportaron que debido a complicaciones técnicas el sistema fue incapaz de capturar imágenes [43].

Un Equipo de Corea utilizó en 2017 cFS como software de control para la observación del eclipse solar visible en Estados Unidos de América, lo que permitió medir el brillo polarizado de la corona solar en cuatro longitudes de onda. Para la misión, se implementaron cuatro aplicaciones propias en conjunto con SCH, CI y TO como aplicaciones reutilizables de cFS en un sistema embebido, controlado por medio de la estación terrena proporcionada como herramienta del *framework*, conectado por medio de un bus *Ethernet* [44].

En un artículo de la NASA publicado en 2012, se detalla el concepto y prueba de un vehículo lanzador como parte del proyecto *Morhpeous* capaz de aterrizar y despegar de forma vertical, dicho vehículo utiliza como FSW una de las primeras versiones de cFS, se eligió este*framework* por sus flexibilidad y adaptabilidad, añadiendo únicamente pequeñas porciones de software específicas para la misión [45].

Iniciada en 2005 y lanzada en 2009, la misión *Lunar Reconnaissance Orbiter (LRO)* desempeñó con éxito sus objetivos: orbitar y observar a 50km de altitud la superficie lunar por un año con posibilidad de incrementar la duración de la misión a 3 años. Para esto, usó el cFS como principal FSW ejecutado desde un RTOS VxWorks con un procesador RAD750 RHA, aprovechando las

características de detección de errores con las aplicaciones de CS y HK, complementadas con el uso de un *watchdog*, que en conjunto con electrónica de grado espacial proporcionan una plataforma tolerante a fallas, capaz de desempeñar sus funciones de forma correcta en entornos espaciales. Sin embargo, el costo del procesador RHA es significativamente alto, superando los 200,000 dólares [46].

En 2015, el proyecto *Neutron Star Interior Composition Explorer (NICER)* de la NASA concluyó con su fase de diseño y desarrollo, el cual es un instrumento de rayos X para la Estación Espacial Internacional (ISS, por sus siglas en inglés). Este sistema utiliza el cFS como FSW para una OBC que se ejecuta desde un procesador MEB con VxWorks, con la función de administrar las comunicaciones con la ISS y el control de orientación, entre otras tareas [47].

A continuación, en la tabla 9.1 se muestra una comparación entre las distintas misiones que usan cFS del estado del arte, resaltando el tipo de OBC y el objetivo principal de la misión.

Misión	Tipo	Año	Procesador	Tipo de componentes	FSW	Referencias
EagleCam	*CubeSat*	2019	NVIDIA Jetson TX2	COTS	cFS con SCH, TO, CI, y 4 apps propias.	[43]
Control de telescopio	Terrena	2017	-	COTS	-	
LRO	Satélite	2009	RAD750	Grado espacial	14 apps de cFS + 7 apps propias.	[46]
Morpheous	Lanzador	2012	-	-	-	[45]
NICER	Carga útil	2015	MEB	Grado espacial	11 app de cFS + 10 app propias.	[47]
Observatorio Dinámico Solar (SDO, por sus siglas en inglés)	Satélite	2010	-	-	-	[44]
Observatorio Swift	Satélite	2004	-	-	-	[47]
Medidor Global de la Precipitación (GPM, por sus siglas en inglés)	Satélite	2014	-	-	-	[44]

Tabla 9.1: Misiones que declaran el uso de cFS. (-) No especifica.

De querer implementar cFS en un nanosatélite *CubeSat* para LEO, el uso de procesadores como el RAD750 implicaría costos muy elevados, limitando el tipo de institución que podría desarrollarlo, por lo que a continuación se presentan algunas opciones en la tabla 9.2 de computadoras embebidas

con los recursos suficientes para ejecutar un FSW basado en cFS en concreto que sean capaces de ejecutar un sistema operativo bajo el estándar POSIX, además como el objetivo son *CubeSat*, se acotó la búsqueda a computadoras con dimensiones menores a los 10 x 10 cm. Se puede notar como el consumo promedio de este tipo de computadoras es de 2.95W y el procesador más usado son los Cortex de la familia A, algunos con más de un núcleo lo cual permite establecer políticas de cómputo redundante.

Nombre	Herencia de vuelo	Consumo aproximado de potencia (W)	Tamaño (mm)	Procesador	Referencias
Raspberry Pi 4	AstroPi	3.62	85 x 56	Quad core Cortex A72.	[48], [49]
Raspberry Pi Zero	GASPACS CubeSat	0.59	65 x 30	ARM 11.	[50], [51]
Beagle Bone Black	-	3	86.36 x 53.34	Sitara AM3358 (ARM Cortex A8).	[52]
PocketBeagle	OreSat	0.925	56 x 35	ARM Cortex-A8.	[53], [54]
NVIDIA Jetson Tx2	EagleCam	7.5	69.6 x 45 o 92 x 60	Hexa core (2x Denver2 + 4x Cortex A57) + GPU de 256 núcleos.	[42], [55]
Colibri iMX6 (*Viola Carrier Board*)	-	1.45	74 x 74	Dual core Cortex A9.	[56]
Odroid C4	-	3.57	85 x 56	Quad core Cortex A55.	[57]

Tabla 9.2: Lista de computadoras embebidas. (-) No especifica.

9.4 Conclusiones del estado del arte

Los satélites pequeños son una tendencia en la exploración espacial, consolidándose como una herramienta accesible para todo tipo de organizaciones (privadas y públicas) con un presupuesto menor comparado con el necesario para desarrollar y lanzar satélites insignia de mayor tamaño, impulsando el desarrollo de áreas como las telecomunicaciones, el monitoreo ambiental o el manejo de desastres naturales. Esto gracias a la miniaturización de tecnología que permite desarrollar pequeños satélites con las capacidades de para ejecutar funciones similares a las de satélites insignia de gran tamaño, pero en LEO por una fracción de su costo.

El uso de estándares como el *CubeSat* permite tener diseños modulares, de fácil integración y con una gran variedad de opciones al momento de elegir los componentes, con computadoras resistentes a la radiación o COTS dependiendo del presupuesto y de las características de la misión.

Esto ha impulsado la adopción de arquitecturas de software más abiertas, escalables, reutilizables y robustas, siendo el *framework* de la NASA (cFS) una de las opciones más destacadas [29].

La arquitectura de cFS favorece el desarrollo colaborativo y la reutilización de software y mantenimiento eficiente, además de que al establecer una estructura definida, facilita la integración de módulos desarrollados por distintos equipos. Esto resulta valioso para misiones con restricciones de tiempo y recursos, comunes para instituciones educativas como universidades y pequeñas empresas privadas.

No hay que perder de vista que la implementación de cFS no garantiza un sistema tolerante a fallas, pero sí tiene el potencial para serlo, dependiendo de las restricciones de diseño impuestas por la misión. Para misiones de satélites insignia y de instituciones gubernamentales se prefiere el uso de componentes resistentes para la radiación, acompañados de buenas prácticas aplicadas durante el desarrollo del software, como el manejo de errores. Pero para pequeños satélites, cFS suele usarse en computadoras embebidas que proporcionan una capacidad de procesamiento elevada a un costo reducido, sin embargo, esto presenta una mayor probabilidad de falla y, por lo tanto, es necesario implementar algún tipo de redundancia para asegurar un nivel de fiabilidad suficiente para desempeñar su misión.

El uso de cFS facilita la implementación de técnicas de tolerancia a fallas, comúnmente mediante algún tipo de redundancia, ya que cuenta con un buen sistema de detección de errores, pero es responsabilidad del programador seleccionar el método o técnica para solucionarlo, lo que incrementa el tiempo de desarrollo y por tanto los costos. El *framework* contiene varios módulos para la implementación de tolerancia a fallas, pero dependen de la interacción de múltiples aplicaciones para realizar sus funciones, lo que aumenta el riesgo de errores ya que de fallar una aplicación, se compromete todo el sistema.

De manera general, para la corrupción de información se tienen dos soluciones posibles con las aplicaciones existentes de cFS, que incluyen: enviar un respaldo desde la estación terrena en caso de una falla, que de comprometerse el enlace satélite-Tierra el sistema se perdería; y la opción de mantener un respaldo a bordo, que de tratarse de una OBC COTS, el respaldo tiene altas probabilidades de dañarse. En conclusión, es necesario desarrollar un sistema de bajo costo y de rápida integración, que aproveche las características de cFS para incrementar el nivel de fiabilidad en computadoras de a bordo de nanosatélites, con la capacidad de corregir fallos de forma autónoma y eficiente.

Parte III

DISEÑO E IMPLEMENTACIÓN

10 Diseño del concepto

En este capítulo se identifican las necesidades y se genera una arquitectura general con divisiones por subsistemas para facilitar su desarrollo.

10.1 Necesidades

Como primera etapa, previa a la definición de requisitos y especificaciones, hay que establecer una lista de necesidades que deberá satisfacer este proyecto, enfocándose en responder la pregunta qué se necesita y no cómo se realizará [4].

Tomando en cuenta una metodología descendente, se contemplaron dos sistemas generales como un máximo nivel de abstracción, cada uno con diferentes necesidades, el Software de vuelo (FSW) de la OBC basado en el *framework* de cFS como sistema para proteger, y el supervisor como el sistema protector. Se agruparon todas las necesidades en la tabla 10.1 en donde, a cada expresión se le asignó una clave única, con el objetivo de distinguirlas de forma individual y hacer referencia a ellas en futuros capítulos. Además, se definieron pesos o prioridades en donde cero es el de mayor importancia y a medida que incrementa la cuenta (hasta un límite de 5), decrece su importancia.

Para el supervisor se necesita de un sistema compatible con cualquier plataforma que ejecute el *framework* de cFS (NA-005), con el objetivo de responder a los errores generados por la corrupción de aplicaciones en la memoria (NA-001) de la OBC mediante información de respaldo a bordo (NA-004), de la cual, se debe de garantizar su integridad ante la radiación y estar disponible siempre que sea necesario (NA-003). Con un consumo energético que sea despreciable (NA-002) comparado con la OBC con el fin de reducir el impacto de su integración, además, se busca que sea de bajo costo (NA-006), por lo que se priorizará el uso de componentes COTS.

10.2 Requisitos

Los requisitos del sistema son un punto medio entre lo cualitativo y lo cuantitativo, se establecen como una aproximación para cumplir con los objetivos establecidos, respondiendo a la pregunta qué debe hacer el sistema, sin especificar el cómo lo hará. Existen tres tipos básicos: **funcionales**, que definen qué debe realizar un sistema y cómo debe desempeñar; **operacionales**, que definen cómo se usará un sistema; y las **restricciones**, que definen las limitantes del sistema [58].

Se definieron requisitos para ambos sistemas (tabla 10.2), para responder a las necesidades previamente planteadas. Para el sistema supervisor, se requiere que este cumpla con sus funciones en LEO sin interrupciones, es decir, que sea fiable (RA-001). Esto considerando las condiciones extremas en las que va a operar (radiación y temperatura) (RA-004). Además, el supervisor debe responder en tiempo real (RA-005) a la solicitud de la transmisión del respaldo de la OBC (RA-002), por lo tanto, el supervisor debe mantener una comunicación bidireccional en todo momento (RA-003). Con el fin de reducir el impacto del supervisor en la OBC, este deberá consumir menos del 1 % (o 100 veces menos) de la potencia activa (RA-007) de una computadora embebida promedio. Por último, para mantener el costo económico bajo, se debe priorizar el uso de componentes COTS con al menos el 50 % (RA-006).

#	Clave	Peso	Necesidad	Justificación
			Sistema supervisor	
1	NA-001	1	Funcionamiento	El supervisor debe solucionar los errores de corrupción de memoria de las aplicaciones que reporte cFS.
2	NA-002	4	Consumo energético	El consumo de potencia del supervisor debe ser despreciable en comparación a la OBC.
3	NA-003	0	Fiabilidad	El supervisor debe desempeñar sus funciones sin interrupciones durante el tiempo de vida de la misión.
4	NA-004	3	Almacenamiento	El supervisor necesita contar con la capacidad suficiente para almacenar un respaldo de las aplicaciones críticas del FSW.
5	NA-005	2	Portabilidad	El supervisor deberá ser compatible con cualquier plataforma que esté ejecutando cFS.
6	NA-006	5	Costos económicos	El sistema deberá mantener bajos costos de construcción.
			Sistema a proteger	
7	NB-001	1	Diagnóstico	El FSW debe reportar y clasificar los errores críticos en el sistema de forma inmediata.
8	NB-002	0	Recuperación	El sistema debe procesar los archivos de recuperación enviados desde el supervisor.

Tabla 10.1: Necesidades del sistema

Por su parte, el FSW de la OBC debe diagnosticar, clasificar y reportar las fallas o errores críticos del sistema de forma inmediata (NB-001), con el fin de que el sistema supervisor pueda enviar los archivos de recuperación, para que el FSW los pueda procesar (NB-002). Definiendo los requisitos, el sistema debe ser capaz de detectar cuando una aplicación crítica de cFS falle y reportarla en caso de que el origen de la falla sea una corrupción de datos en memoria (RB-002), además, deberá realizar la petición de datos de recuperación a través de un canal de comunicación exclusiva (RB-005) con el supervisor y estar disponible (RB-004) al momento de recibir la respuesta para procesar la información en tiempo real (RB-003). Para que todo funcione, el OS y los servicios básicos de cFE deben operar de forma correcta durante el tiempo de vida útil de la misión (RB-001).

#	Clave	Peso	Requisito	Justificación
			Sistema supervisor	
Funcionales				
1	RA-001	0	Fiabilidad	Que el sistema funcione correctamente durante el tiempo de vida útil de la misión.
2	RA-002	1	Desempeño	Debe enviar los respaldos solicitados al detectarse una corrupción de datos en alguna de las aplicaciones críticas de cFS.
3	RA-003	4	Portabilidad	Debe mantener una comunicación bidireccional con cFS, independientemente de la plataforma que esté ejecutando el FSW.
Operacionales				
4	RA-004	6	Temperatura	Debe operar adecuadamente en un rango de temperatura de -40°C y 85°C [59].
5	RA-005	2	Respuesta	Debe responder a las peticiones de cFS en tiempo real.
Restricciones				
6	RA-006	8	Bajos costos	Debe utilizar al menos un 50 % de componentes COTS.
7	RA-007	7	Consumo energético	Debe consumir al menos 100 veces menos potencia que la promedio de una OBC para nanosatélites.
			Sistema a proteger	
Funcionales				
8	RB-001	0	Fiabilidad	El OS y cFE funcionen correctamente durante el tiempo de vida útil de la misión.
9	RB-002	1	Desempeño	El FSW debe detectar errores en las aplicaciones críticas, reportar al supervisor la aplicación que tuvo la falla y procesar los archivos de recuperación.
Operacionales				
10	RB-003	3	Respuesta	El FSW debe reportar los errores y procesar los archivos de recuperación en tiempo real.
11	RB-004	2	Disponibilidad	El sistema debe estar disponible siempre que el supervisor envíe los archivos de recuperación.
Restricciones				
12	RB-005	4	Comunicación	Debe contener al menos un medio de comunicación exclusivo con el supervisor.

Tabla 10.2: Requisitos del sistema clasificados por categorías

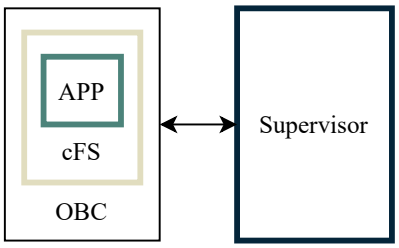


Figura 10.1: Diseño de concepto del sistema supervisor

Con los dos subsistemas identificados, el FSW de la OBC y el supervisor, se realizó un diagrama a bloques (figura 10.1), en donde se resalta el bloque de la OBC el cual, de forma general tiene las tareas de control y procesamiento y se supone que usará el *framework* cFS como FSW. Conociendo las aplicaciones incluidas en el *cFS bundle-pack* (véase capítulo 8.4), así como las aplicaciones disponibles en el repositorio oficial de la NASA (véase tabla B.1), será necesaria una app de usuario que envíe las peticiones de restauración y las procese, actuando como interfaz con el sistema supervisor.

11 Diseño a nivel sistema

Esta es la etapa intermedia entre el concepto y el diseño de detalle, en donde se define la arquitectura general del sistema, los subsistemas que lo conforman y cómo interactúan entre sí.

Con un poco más de detalle en cada uno de los bloques, la OBC incluye tanto el hardware como el FSW, que al ser cFS y recordando su sistema de capas planteadas en la figura 8.1, el software resulta ser portable en cualquier plataforma siempre que pueda ejecutar alguno de los OS permitidos, dando como resultado una gran cantidad de arquitecturas de hardware distintas con las que se pueden trabajar. Esto dificulta el diseño de un supervisor específico para cada arquitectura, por lo tanto, el bloque de aplicación de usuario es indispensable para mantener su portabilidad, conteniendo la información suficiente para garantizar la transferencia bidireccional de datos, así como las líneas de control necesarias para realizar las tareas de recuperación.

Analizando el supervisor un poco a más detalle, es necesario considerar un bloque de almacenamiento RHA, en donde se mantiene una copia de las aplicaciones críticas de cFS para la misión que podrán ser cargadas a la OBC desde el supervisor. Por lo tanto, es necesario coordinar cuando se envía y se recibe información, además de establecer la lógica que indique el tiempo y la acción necesaria para recuperar la funcionalidad del sistema, haciendo evidente la necesidad de un bloque de procesamiento, o un microcontrolador que actúe como cerebro del supervisor, encargado de recibir información del estado de cFS y enviar la información necesaria para su recuperación. De esta manera, se requiere de un bus de datos bidireccional y las líneas de control para la OBC, además de las conexiones necesarias que permitan actualizar al supervisor de forma remota. Se muestra la conexión en la figura 11.1.

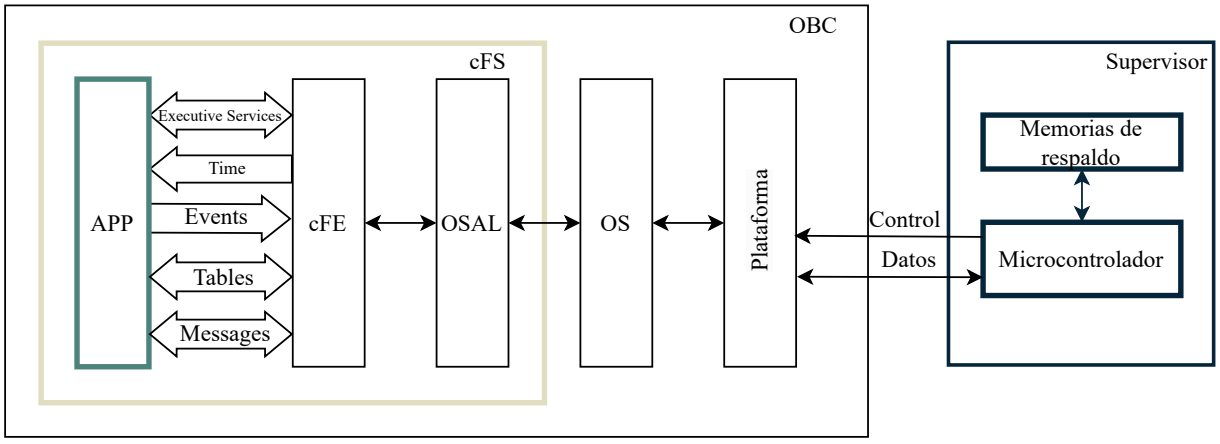


Figura 11.1: Diseño a nivel sistema.

Para explicar la conexión entre la aplicación y el supervisor es necesario recordar la estructura de cFS planteada en la figura 8.1, en donde se puede notar que las aplicaciones no tienen control directo sobre el hardware, ya que usan las API de cFE como interfaz (figura 8.2), que a su vez, utilizan el sistema operativo que se encarga de gestionar los recursos del hardware que finalmente se conecta por medio de puertos de comunicación serial y entradas o salidas de propósito general (GPIO, por

sus siglas en inglés) con el supervisor. Este sistema por capas garantiza una conexión adecuada con el supervisor independientemente de la plataforma sobre la que se esté ejecutando cFS.

Recordando el funcionamiento de cFS, sabemos que al momento del arranque, el FSW copia a la memoria RAM todas las aplicaciones necesarias en conjunto con cFE y empieza su proceso nominal. De presentarse una falla que impida el correcto funcionamiento de una aplicación se lanza una excepción que, según la gravedad, puede tener como respuesta el reinicio de la aplicación dañada, volviendo a cargar la librería compartida (`.so` para Linux) de la aplicación o puede reiniciar todo el sistema. Sin embargo, de tener una corrupción en la memoria persistente y que el archivo `.so` de la aplicación se encuentre dañado, cFS rechazará la carga resultando en la pérdida de la aplicación, dando como resultado un evento de tipo error gestionado por *Event Service*, pero enviado por *Executive Service* a través del SB.

Tomando en cuenta las capacidades de reconfiguración de cFS (véase la figura 8.4), se sabe que el *framework* cuenta con diferentes aplicaciones que, en conjunto, pueden reempalzar archivos dañados. Estas aplicaciones incluyen a: *Command Ingest* para recibir y procesar comandos, *File Manager* para manejar archivos, *Telemetry Output* para notificar a la estación terrena que ha ocurrido un error, y la app de CFDP que permite enviar datos a través del protocolo CCSDS. Sin embargo, si alguna de estas aplicaciones presenta fallas, no se podrá completar el proceso de recuperación. Por ejemplo, de fallar la aplicación de *Command Ingest*, aún sería posible transmitir un nuevo archivo, sin embargo, se necesita de los comandos para que la aplicación de *File Manager* sea capaz de reemplazar el archivo dañado por la copia de respaldo recibida, además, para iniciar la aplicación que ha sido actualizada se requiere de un comando de inicio, haciendo imposible la reparación del sistema.

Entonces, conociendo el funcionamiento de cada componente de cFS y su detección de fallas se pueden definir tareas concretas para cada subsistema, empezando con el FSW basado en el *framework* cFS que debe ser capaz de detectar y reportar las fallas encontradas entre su variedad de aplicaciones, además de que debe de gestionar el reinicio, recarga e inicio de todas las aplicaciones de cFS, gestionar la frecuencia de envío de la señal de alivio y procesar cualquier comando que reciba desde la estación terrena, dichas tareas se resumen en la tabla 11.1.

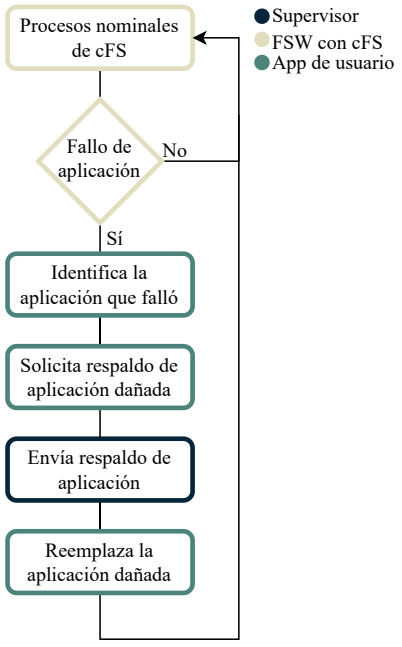


Figura 11.2: Proceso propuesto para la recarga de una aplicación dañada.

La aplicación planteada como interfaz entre el supervisor y el FSW tiene como tareas (tabla 11.1) el monitoreo periódico de los eventos críticos reportados a EVS a través del SB. Al llegar la notificación de un evento crítico la app de usuario se encarga de identificar la aplicación que ha causado la falla y transmitir por un puerto serie al supervisor la petición de envío del respaldo de la aplicación dañada, que al ser recibida deberá realizar la gestión de la reparación e inicio de la aplicación recuperada. Este proceso puede apreciarse en el diagrama de flujo presentado en la figura 11.2.

#	Clave	Tarea
		Tareas FSW/cFS
1	TA-001	SB: Encargado de transmitir datos entre aplicaciones
2	TA-002	EVS: Registrar y clasificar eventos y errores
3	TA-003	CI: Procesar comandos de estación terrena o Apps internas
4	TA-004	ES: Iniciar, detener o reiniciar aplicaciones de cFS
5	TA-005	SCH: Gestionar la frecuencia de la señal de alivio
		Tareas APP
1	TB-006	Proporcionar interfaz con el supervisor externo
2	TB-007	Recibir y procesar la información transmitida por el supervisor
3	TB-008	Enviar señal de alivio para *watchdog*
4	TB-009	Monitorear EVS en busca de errores
		Tareas Supervisor
1	TC-010	Respaldar aplicaciones críticas
2	TC-011	Monitorea la señal de alivio del *watchdog*
3	TC-012	Transmisión de respaldos

Tabla 11.1: Tareas por subsistema.

El bloque del sistema supervisor (tabla 11.1) consta de un sistema externo a la OBC conectado de forma física por puerto serie y GPIO para el *watchdog*, con una interfaz de software proporcionada por el bloque de aplicación. Este bloque deberá estar en reposo hasta recibir la petición de envío del respaldo de la aplicación que deberá responder de forma inmediata, además, debe monitorear la señal de alivio de la OBC y en caso de no recibirla en un tiempo predeterminado debe reiniciar el procesador (figura 11.3).

En resumen, en este capítulo se ha planteado un diseño a nivel sistema que incluye la arquitectura general del sistema, con 3 subsistemas clave interconectados: el FSW de la OBC, la aplicación de usuario y el supervisor. Se definió cómo interactúan entre sí y qué tareas deben realizar para alcanzar los objetivos del proyecto.

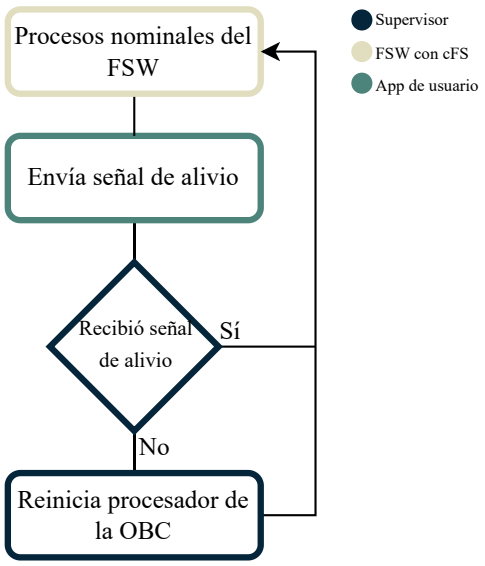


Figura 11.3: Proceso de funcionamiento del *watchdog*.

12 Diseño de detalle

Una vez planteado cada subsistema, la interconexión de sus componentes y sus funciones, se reporta cómo se implementó cada parte del sistema para cumplir con los objetivos establecidos en el planteamiento de este proyecto.

12.1 Aplicación de usuario

Siguiendo las recomendaciones para el desarrollo de aplicación de cFS [36], se utiliza la aplicación muestra (*Sample_app*) que forma parte del *cFS Bundlepack* en su versión draco-rc5, con el fin de mantener la estructura genérica de las aplicaciones de usuario (véase la figura 8.2) que incluye: la creación de un canal para recibir paquetes del *Software Bus* (TA-001), el procesamiento de comandos terrestres, reporte de *Housekeeping*, reporte de eventos y manejo de excepciones. Esta estructura proporciona el esqueleto para la aplicación desarrollada, reduciendo así el tiempo de programación y facilitando la compatibilidad con cFS. Siguiendo la nomenclatura de cFS se utilizaron tres letras para nombrar a la nueva aplicación: **ERS app**, acrónimo de *External Recovery Supervisor application* o aplicación de recuperación del supervisor.

Del diseño de concepto conocemos las tareas con las que debe cumplir el software de la aplicación de usuario. Empezando con el monitoreo de *Event Service* (TB-009) para detectar el momento en el que ocurra una corrupción de datos (TA-002) en alguna aplicación. Se sabe que *Executive Service* es el encargado de emitir el evento al fallar en intentar iniciar, reiniciar o recargar una aplicación. ES cuenta con 53 eventos de tipo `error` (para la versión draco-rc5) de los cuales, solamente cinco (véase tabla 12.1) eventos se pueden lanzar al momento que de rechazar un archivo ejecutable de una aplicación por corrupción de sus datos. Para poder responder cuando ocurran estos errores, es necesario registrar el *msg-ID* de eventos (`CFE_EVS_LONG_EVENT_MSG_MID`) al canal de la app de usuario, sin embargo, a este ID llegan todos los tipos de eventos por lo que será necesario filtrar para dejar pasar únicamente los cinco eventos de interés.

Eventos de Executive Service		
ID	Nombre cFS	Descripción
26	CFE_ES_START_ERR_EID	Fallo al crear aplicación al usar *Start Application*
38	CFE_ES_RESTART_APP_ERR1_EID	Fallo al solicitar *Restart Application*
40	CFE_ES_RESTART_APP_ERR3_EID	Fallo al iniciar aplicación en *Restart Application*
42	CFE_ES_RELOAD_APP_ERR1_EID	Fallo al solicitar *Reload Application*
44	CFE_ES_RELOAD_APP_ERR3_EID	Fallo al iniciar aplicación en *Reload Application*

Tabla 12.1: Eventos de tipo error asociados al fallo en el inicio, reinicio y recarga de aplicaciones para cFS Draco-rc5.

Una vez que se levanta un evento de tipo error (`CFE_EVS_EventType_ERROR`) se copia el mensaje en el canal de la aplicación de usuario y se determina si es uno de los cinco errores seleccionados.

De ser el caso, se procesa el evento (figura 12.2) para determinar cuál es el archivo que corresponde a la aplicación que presenta fallas y, una vez identificado el origen, solicitar al supervisor que envíe el respaldo correspondiente (TB-007). Una vez completada la recepción del archivo (TB-007), ERS app reemplaza la aplicación dañada por el respaldo recibido y, por último, solicita su activación. Existen distintos métodos para activar una aplicación, la más sencilla es usando la API de *Executive Service* (TA-004) pero ésta sólo permite reiniciar aplicaciones, por lo que únicamente es útil en el caso de los eventos `CFE_ES_START_ERR_EID` y `CFE_ES_RESTART_APP_ERR1_EID`, de no ser el caso ERS app optará por usar el comando de inicio de *Executive Service* enviado a través de *Command Ingest* (TA-003). Hay que tener en cuenta la posibilidad de que la aplicación que falle sea *Command Ingest*, haciendo imposible la activación de la aplicación, por lo que ERS app se diseñó con la opción de escoger si el método de activación de la aplicación restaurada es por aplicación o con la opción de reiniciar todo el sistema, eliminando de esta forma el problema con *Command Ingest*.

Para la comunicación entre ERS app y el supervisor (TB-006) se escogió UART por ser un protocolo de comunicación sencillo y popular, por lo que es compatible con bastantes computadoras embebidas. Además, UART proporciona una codificación con bit de paridad, lo que aporta redundancia de información y fiabilidad al sistema.

Para configurar el *watchdog*, se programó la aplicación de *Scheduler* (TA-005) un *msg-ID* (`ERS_WD_MID`) que distribuye un mensaje de sincronización por el *Software Bus* de forma periódica a una frecuencia programable. Basta con que dentro de la aplicación se asocie el ID al canal para que al llegar la notificación, ERS app envié la señal de alivio al supervisor a través del puerto UART (TB-008).

Todas las tareas de ERS app descritas anteriormente se encuentran representadas de forma gráfica en dos diagramas de flujo. En el primero, se muestra la estructura general de ERS app (figura 12.1), en donde se puede notar cómo mantiene el diseño genérico de las aplicaciones de cFS, añadiendo la capacidad de procesar eventos y atender la señal periódica del *watchdog*. En el segundo diagrama de flujo (figura 12.2) se encuentra el proceso detallado de la reconfiguración de una aplicación dañada.

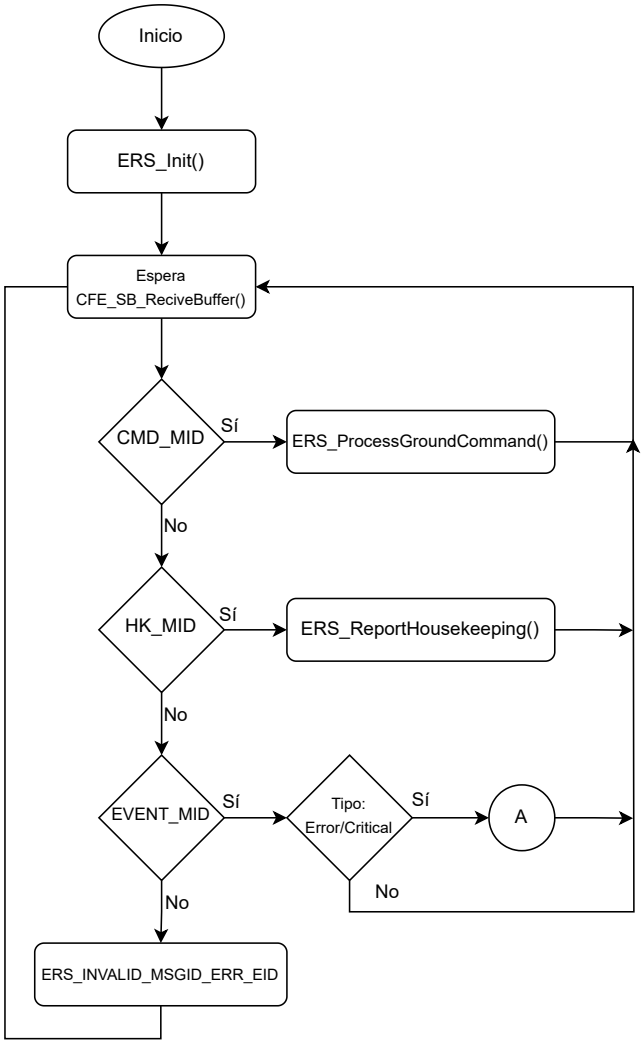


Figura 12.1: Diagrama de flujo de la función principal de la aplicación de recuperación.

A continuación, se muestra la descripción a detalle de las funciones presentes en el diagrama de flujo de la figura 12.2:

- ☐ `ERS_TxUARt()`: transmite al supervisor la petición de envío de una aplicación, debe recibir como parámetro el identificador de aplicación a ser enviada.
- ☐ `ERS_RxUART()`: llena el buffer de datos que será usado para restablecer la aplicación dañada.
- ☐ `ERS_ReloadFile()`: escribe en el archivo correspondiente según la app indicada como parámetro.
- ☐ `CFE_ES_ResetCFE()`: forma parte de las API de ES, la cual se encarga de reiniciar el cFE, con la opción de un reinicio exclusivo del software cFE, o un *power-on reset*, el cual equivale al reinicio de todo el procesador.
- ☐ `ERS_StartAppCmd()`: configura el comando `ES_START_APP_CMD`, para posteriormente llenar el mensaje que incluye el nombre de CFS de la App, el nombre del archivo ejecutable, el punto de entrada (la función `main()`), el tamaño de pila, la prioridad, y la excepción en caso de fallo.
- ☐ `CFE_ES_RestartApp()`: es parte de las API de ES, y necesita como parámetros, el AppID, que es un número asignado en tiempo de ejecución a cada aplicación, este número cambia cada que se reinicia Core Flight Executive (cFE) o al reiniciar la aplicación.
- ☐ `ERS_WatchDog()`: envía la señal de alivio por UART al recibir la señal de sincronización de SCH.

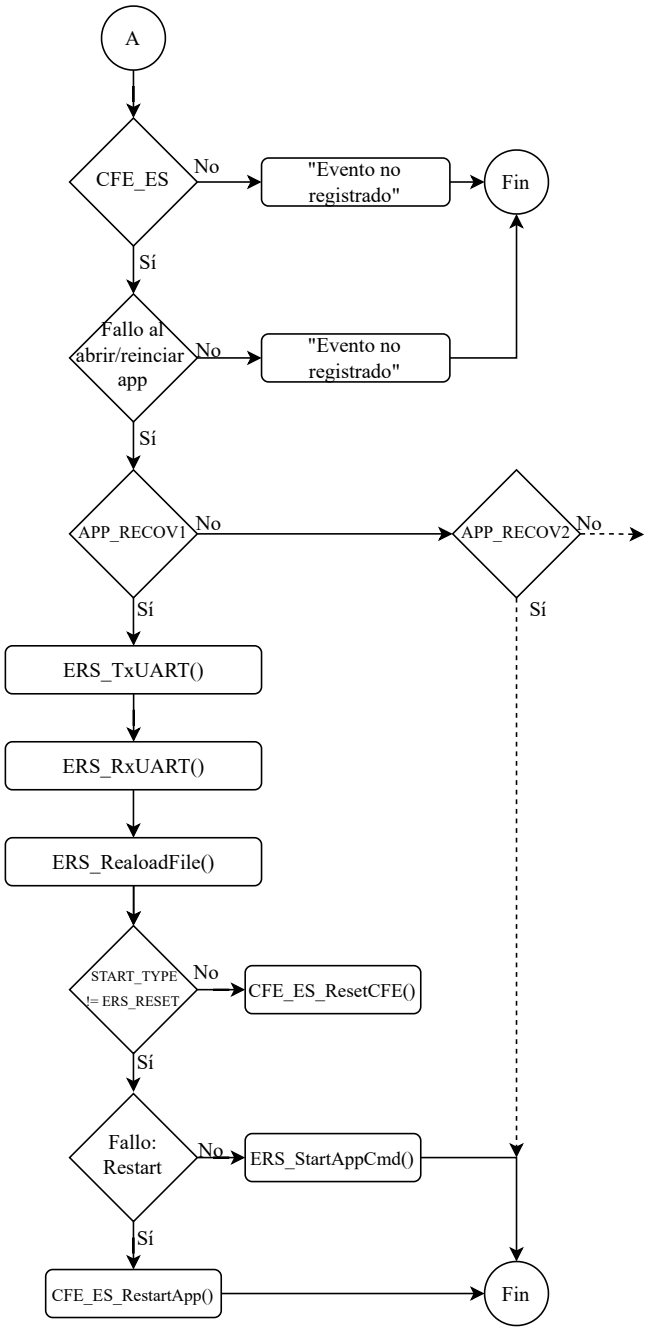


Figura 12.2: Manejo de eventos en la aplicación de recuperación.

12.1.1 Manejo de errores dentro de ERS App

Dado que ERS app es una aplicación que reside en la misma memoria que el resto de cFS se puede esperar que también falle. De ser éste el caso, la comunicación con el supervisor para la recepción de los respaldos del resto de aplicaciones se vería comprometida, sin embargo, el fallo de ERS app no interfiere con el resto del funcionamiento del sistema por lo que la misión puede continuar su curso. Además, ERS app puede estar monitoreada a su vez por el cFS ya que, si falla y no envía su estado cuando HK lo solicite, el sistema detecta un fallo y lo transmite como parte de la telemetría a Tierra, lo que permite una intervención manual para corregir el fallo. Además, el supervisor externo al monitorear de forma periódica una señal de alivio emitida por ERS app, conserva la capacidad de actuar como un *watchdog* y de reiniciar el sistema en caso de una falla para intentar restablecer el sistema.

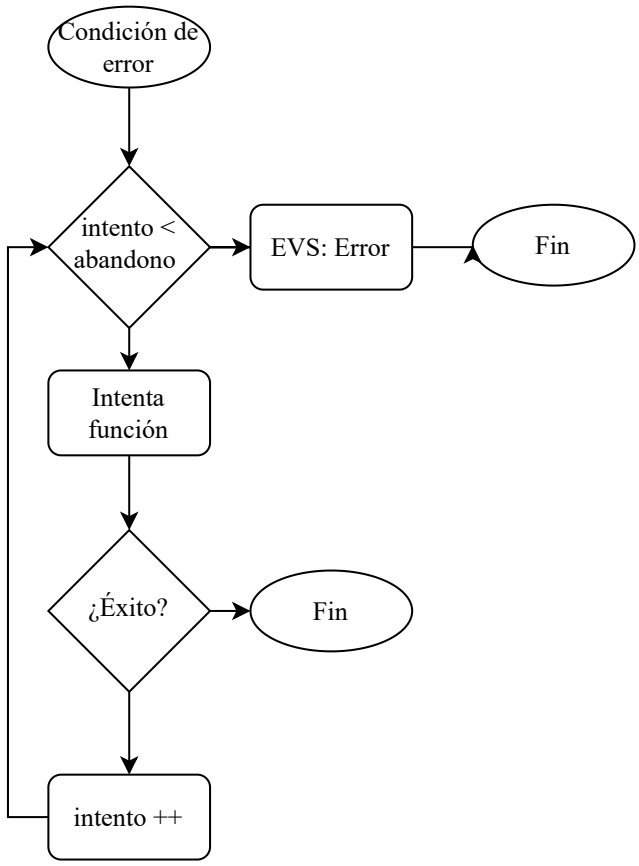


Figura 12.3: Diagrama de flujo del manejo de errores.

Con el objetivo de mantener un nivel de fiabilidad adecuado para sistemas espaciales, todas las funciones de la aplicación ERS app incluyen un manejo de excepciones. Para esto, todas las funciones devuelven un estado de operación con 0 para indicar el éxito y con un valor diferente para indicar una condición de error.

En el caso de presentarse un error al intentar ejecutar una función, el programa entra en un ciclo (véase la figura 12.3) que intentará completar la función, esto se itera un número de veces establecido por el usuario (para evitar la saturación del procesador). Si el ciclo termina sin completar con éxito la función, se emite un evento de tipo `error` por medio de *Event Service*.

12.2 Supervisor

A continuación, se presenta el diseño a detalle del supervisor, en donde se definen las especificaciones de hardware, así como los algoritmos de software que debe implementar para cumplir con sus objetivos.

12.2.1 Hardware

Para definir la capacidad de almacenamiento mínima para respaldar las aplicaciones necesarias de una misión con cFS (TC-010), se registró el tamaño de almacenamiento usado por cada una de las aplicaciones *open source* disponibles en el github de cFS [34] para la versión Draco-rc5 (véase

la tabla 12.2). Se puede notar que la capacidad varía bastante entre cada aplicación, con algunas llegando a ser 10 veces más que otra, por lo que se tomó en cuenta el promedio que es de 0.09MB por aplicación. De asumir que queremos respaldar todas las aplicaciones, se requiere poco más de 1.3MB. Además, si se considera que se almacenarán aplicaciones propias del usuario la capacidad debe ser mayor, por lo que se tomará en cuenta al menos el doble de almacenamiento necesario, dando un total de **3MB** después de ajustar el valor a un número entero.

ID cFS	Nombre de aplicación	Tamaño (Bytes)	Tamaño (MB)
CI	Command Ingest Lab	26,776	0.0267
CS	Checksum application	157,840	0.1578
DS	Data Store application	107,984	0.1079
FM	File Manager application	128,744	0.1287
HK	Housekeeping application	53,656	0.0536
HS	Health & Safety application	87,904	0.0879
LC	Limit Checker application	97,640	0.0976
MD	Memory Dwell application	68,616	0.0681
MM	Memory Manager application	88,640	0.0886
SBN	Software Bus Network application	107,968	0.1079
SC	Stored Commands application	132,592	0.1325
SCH	Scheduler Lab	24,392	0.0243
TO	Telemetry Output	37,912	0.0379
Suma Aplicaciones		1,120,664	1.12066
Ejecutable de cFS		1,270,320	1.27030
Ejecutable + aplicaciones		2,390,984	2.39098

Tabla 12.2: Tamaño de aplicaciones de cFS.

Si se toma en cuenta el consumo promedio de potencia aproximado (2.95W) de la tabla 9.2 y recordando que el consumo energético del supervisor debe ser despreciable en comparación con la OBC, se consideró una potencia menor a 29mW[7] (RA-007). Esto resulta en la necesidad de un microcontrolador de bajo consumo, además de que debe tener puertos serie para la comunicación con ERS app y contar con GPIO y temporizadores para la conexión del *watchdog*. Por otro lado, se requiere mantener bajos costos (priorizando el uso de componentes COTS) (RA-006) que puedan desempeñar sus funciones en LEO, por lo que debe tener cierto grado de tolerancia a la radiación (RA-001) y soportar temperaturas de operación esperadas dentro de un nanosatélite en el medio ambiente espacial de entre -40°C y 85°C (RA-004).

Entonces, es posible determinar un conjunto de especificaciones (tabla 12.3) con las cuales debe cumplir el hardware del supervisor para poder satisfacer con las necesidades y requisitos planteados anteriormente en este capítulo.

[7]Para aplicaciones de ingeniería, una variable 100 veces menor a otra puede ser considerada despreciable [60]

Clave	Peso	Especificación	Valor mínimo	Valor máximo	Unidades
ES-001	0	Almacenamiento	2.6	–	MB
ES-006	1	Tolerancia a la radiación	Almacenamiento RHA	–	–
ES-005	2	Interfaces	UART/SPI/I2C/GPIO	–	–
ES-004	3	Temperatura de operación	-40	+85	°C
ES-003	4	Bajo consumo energético	–	29	mW
ES-002	5	Bajos costos	Microcontrolador COTS	–	–

Tabla 12.3: Especificaciones de hardware del supervisor. *(-) No especifica.*

Por lo tanto, para el supervisor se escogió el microcontrolador MSP430FR5969 de Texas Instruments [61], el cual es un dispositivo COTS de ultra bajo consumo energético, basado en la arquitectura de 16 bits RISC, que integra una memoria FRAM de 64kB, 2kB de SRAM, con 40 GPIO, 2 UART, 3 SPI, 1 I2C, 5 *timer* de 16 bits con una temperatura de operación de -40 a 85°C. Gracias a su memoria con tecnología FRAM, que almacena información mediante la polarización de dipolos eléctricos dentro del material (a diferencia de la tecnología CMOS que almacena cargas eléctricas), le permite ser más tolerante a la radiación. Además, este microcontrolador cuenta con una amplia herencia de vuelo[8] y tiene una versión idéntica de grado espacial (MSP430FR5969-SP) que se puede sustituir en aplicaciones críticas. Dado que su capacidad de almacenamiento interno no cumple con las especificaciones mínimas (ES-001), será complementado con un banco de tres memorias CY15B108QN de *Infineon Technologies* [63] que aportan 1MB de almacenamiento cada una, con tecnología FRAM, con SPI como protocolo de comunicación con una frecuencia máxima de 40MHz y con un rango de temperatura de operación entre –40°C a +85°C.

12.2.2 Software

Se sabe que el supervisor estará conectado a la OBC y en comunicación con ERS app, por lo que se configuró un puerto UART con interrupción para esperar la llegada de la petición de un respaldo. Una vez que llega la solicitud, el supervisor extrae del banco de memorias la información correspondiente y la transmite por puerto serie. Además, el puerto recibe la señal de alivio de forma periódica.

Al recibir la señal de alivio, un temporizador de periodo configurable se reinicia para evitar que termine. De no recibir la señal de alivio, el supervisor supone que la OBC se encuentra en un estado de malfuncionamiento, por lo que por medio de un GPIO enviará la señal correspondiente para reiniciar el procesador. El supervisor mantiene un contador que registra el número de veces que el *watchdog* reinicia la OBC en un período de tiempo, si el número de reinicios alcanza un límite establecido se considera que existe una falla permanente y el supervisor se rinde.

[8]Para sistemas espaciales, herencia de vuelo hacer referencia a un diseño que ha volado con éxito, o a un componente de software o de hardware que ha sido validado en misiones anteriores [62]

Es importante aclarar que la interrupción del UART debe tener mayor prioridad sobre la del *timer*, ya que de lo contrario, podría darse el caso de que estando dentro de la rutina de recepción de UART y antes de terminar de procesar el caracter recibido, se reinicie el procesador al no poder procesar la señal de alivio y por lo tanto, nunca concretar el reinicio del temporizador.

El proceso de todo el software del supervisor se encuentra detallado en el diagrama de flujo de la figura 12.4.

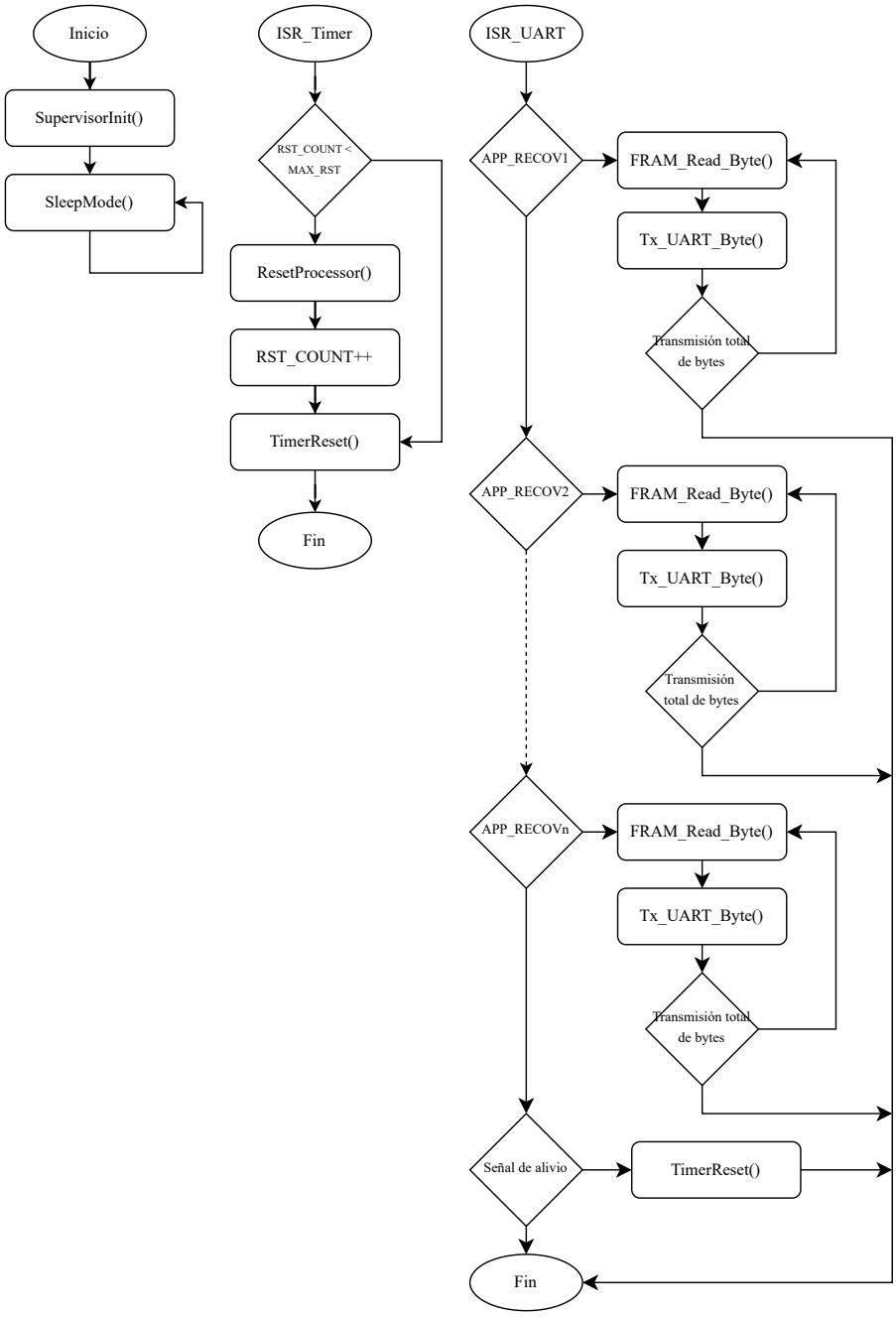


Figura 12.4: Diagrama de flujo del supervisor.

A continuación se describen las características de cada una de las funciones del diagrama de flujo 12.4:

- `SupervisorInit()`: inicializa todos los periféricos (SPI, UART, GPIO y *Timer*).
- `SleepMode()`: mantiene en modo de bajo consumo al microcontrolador y en espera de una interrupción.
- `ResetProcessor()`: envía la señal de reinicio para la OBC a través de un GPIO.
- `TimerReset()`: reinicia el contador del *timer* para el *watchdog*.
- `FRAM_Read_Byte()`: lee un byte del banco de memorias a través de SPI. Esta función fue desarrollada por el equipo de software del LIESE.
- `Tx_UART_Byte()`: utiliza UART para transmitir un byte a la OBC.

En este capítulo se presentó el diseño a detalle del bloque supervisor y de la aplicación de usuario. Se definieron las especificaciones de hardware y software que debe cumplir cada uno de los bloques, tomando en cuenta los objetivos de este trabajo y los requisitos de diseño. En el siguiente capítulo se presentará la implementación de los bloques diseñados.

13 Implementación

Una vez concluido el diseño a detalle de cada uno de los subsistemas, se presenta la implementación del proyecto, en donde los algoritmos y especificaciones previas se materializan en un sistema físico y funcional.

13.1 Aplicación de usuario

Aprovechando las características de cFS, se implementó el diseño en una máquina virtual (*Oracle VM*) con un sistema operativo Xubuntu 22.04.4, se le instaló un editor de código, y de acuerdo con el repositorio de la NASA [34], se instalaron las herramientas necesarias que incluyen: Make, CMake, GCC, Git, PyQt5 y PyZMQ. Una vez terminada la implementación en la computadora portátil, es posible migrar ese diseño a cualquier plataforma embebida con un OS compatible con cFS sin realizar cambios significativos en el código.

Se utilizó el *framework* cFS Draco-rc5, esta no es una versión oficial ya que sigue en desarrollo, pero se seleccionó ya que la NASA anunció el lanzamiento de una nueva versión basada en la Draco-rc5 antes de que acabe el año, la cual es distinta a la versión oficial Aquila en definición de cabeceras y compatibilidad con aplicaciones nuevas, lo cual dificulta la actualización a la futura versión.

Para la implementación del código se siguieron las buenas prácticas recomendadas en la guía de desarrollo de aplicaciones para cFS [36], en donde se recomienda el uso estricto de la capa OSAL, en conjunto con las API de los servicios de cFE, buscando evitar accesos directos al hardware o sistema operativo, con el objetivo final de lograr la reutilización y portabilidad del código.

De la OSAL se utilizaron principalmente funciones para el manejo de archivos, que permiten realizar la recarga de los archivos ejecutables de las aplicaciones de cFS como `OS_open, OS_close, OS_write, OS_read`. Por otra parte, se usaron las API de CFE para suscribirse a los buses de eventos de EVS y para poder enviar comandos a ES en conjunto con las API proporcionadas por *Command Ingest*.

La configuración del UART, en conjunto con las funciones de transmisión y recepción de datos, quedaron como las únicas funciones dependientes de la plataforma sobre la cual se vaya a ejecutar y, por lo tanto, queda como responsabilidad del usuario diseñarlas según la plataforma. Para la implementación de este trabajo, se realizaron las pruebas en una Raspberry Pi4, por lo que se proporcionan las funciones para comunicación exclusivas para esta plataforma con las características mostradas en la tabla 13.1.

Ya que se usó el código base de *Sample_App* como esqueleto para la aplicación, minimizando los cambios en el trabajo principal, añadiendo dos suscripciones adicionales al canal del SB: al bus de eventos y a la señal de sincronización de SCH para el *watchdog*. Además de las funciones para darle servicio a dichas suscripciones. El directorio de carpetas (véase la figura 13.1) se mantuvo similar a las demás aplicaciones, con una carpeta principal de **fsw** en donde se encuentran los archivos .c y de cabecera de las diferentes funciones a ejecutarse, y dos carpetas auxiliares: **mission_inc**, que contiene el archivo de cabecera con los identificadores de mensajes (*message ID´s*) para poder usar

#	Parámetro	Valor
1	Velocidad	115,200 bps
2	Bits de datos	8 bits
3	Bit de paridad	Paridad par
4	Bits de parada	1
5	Control de flujo	Sin control de flujo

Tabla 13.1: Parámetros de configuración del UART para Raspberry Pi 4

el Software Bus Service (SB); y `plataform_inc`, que incluye el archivo de cabecera con los códigos que permiten a cFS medir el desempeño de la aplicación.

Dentro de la carpeta principal, se encuentra el archivo `ers_app.c` que contiene la implementación general del diagrama 12.1, en conjunto con todas las funciones que procesan comandos, que incluyen:

- ☐ `ERS_NOOP_CC:` comando de no operación. Como su nombre lo indica no realiza ninguna operación dentro de la app, pero genera un evento de tipo información lo que permite al usuario conocer el estado de la aplicación desde la base terrena.
- ☐ `ERS_RESET_COUNTERS_CC:` comando de reinicio de contadores. Todas las aplicaciones de cFS cuentan con la posibilidad de reiniciar contadores.

En la carpeta *fsw*, el código `ers_recovery.c`, se encuentran las funciones de apoyo, de transmisión y recepción de UART y de recarga de aplicaciones. Además, dentro del archivo `ers_recovery.h` (anexo C), se encuentran las configuraciones del usuario en donde podrá seleccionar las aplicaciones para proteger, incluyendo el tamaño en bytes del archivo, el nombre de la aplicación, el nombre del archivo ejecutable, el punto de entrada (función `main`), la prioridad, tamaño de pila, y el tipo de lanzamiento de la aplicación tras su recuperación cuyas opciones son reiniciar únicamente la aplicación o reiniciar todo el sistema. La segunda opción es importante ya que las formas de iniciar la aplicación incluyen el uso del comando de *Executive Service* `ES_START_APP`, en caso de que el fallo fuera ocasionado en el inicio de la aplicación o en el caso de un fallo de reinicio de aplicación, se usa la API de ES de `CFE_ES_Restart_App`.

Dentro del archivo de cabecera `ers_events.h` se encuentran las definiciones de eventos de la aplicación, que incluyen los predeterminados para el manejo general de errores para cFS y procesos genéricos, estos son:

- ☐ `ERS_RESERVED_EID:` evento reservado para uso futuro.
- ☐ `ERS_STARTUP_INF_EID:` indica que ERS App se inicializó correctamente.
- ☐ `ERS_COMMAND_ERR_EID:` reporta un error relacionado con un comando inválido o mal configurado recibido por la aplicación.
- ☐ `ERS_COMMANDNOP_INF_EID:` informa que se recibió un comando de no operación.
- ☐ `ERS_COMMANDRST_INF_EID:` informa que se recibió un comando de *reset*.
- ☐ `ERS_INVALID_MSGID_ERR_EID:` reporta la recepción de un mensaje con un ID no válido o no reconocido.
- ☐ `ERS_LEN_ERR_EID:` reporta el error causado por una longitud incorrecta en un mensaje o paquete recibido.
- ☐ `ERS_PIPE_ERR_EID:` reporta un error relacionado con el canal de comunicación, como fallos al leer o escribir en una cola de mensaje.

Se crearon eventos propios para informar al usuario el estado de las funciones principales dentro de ERS app, que incluyen:

- `ERS_DATACORRUPT_DET_INF_EID:` informa al usuario que el error de EVS se asoció con la corrupción de información en alguna de las aplicaciones registradas dentro de ERS app.
- `ERS_FILELOADSUCCESS_INF_EID:` informa que el proceso de petición, recepción y recarga del archivo de recuperación concluyó con éxito.
- `ERS_FILELOADFAIL_ERR_EID:` reporta un error causado por la falla en la petición, recepción o recarga del archivo de recuperación.
- `ERS_WD_SUCCESS_INF_EID:` notifica que la señal de alivio del *watchdog* se envió con éxito.
- `ERS_WD_FAIL_ERR_EID:` reporta que la señal de alivio del *watchdog* no pudo ser enviada.

Con la finalidad de facilitar la programación de las memorias FRAM del supervisor, se crearon herramientas con *Python*, almacenadas dentro de la carpeta *tools*. El archivo llamado `ERS_converter.py` recibe como parámetro el nombre de una librería compartida (.so para Linux) correspondiente a una aplicación compilada de cFS. El programa abre el archivo, lo lee y separa en bytes, almacenándolo en otro archivo de tipo `.txt`, en donde guarda cada byte en un arreglo de nombre `DATA[]`. Sin embargo, en caso de que la cantidad de bytes sea mayor 2,000 elementos, el supervisor no podrá llenar las memorias de respaldo adecuadamente. Para esto, se creó un algoritmo en python `ERS_FileDivider.py` que recibe como parámetro el nombre del archivo `.txt` y el número máximo de bytes por archivo, entonces, el programa lo divide en múltiples `.txt` de menor tamaño lo que permite al supervisor programar los respaldos de las aplicaciones de forma correcta.

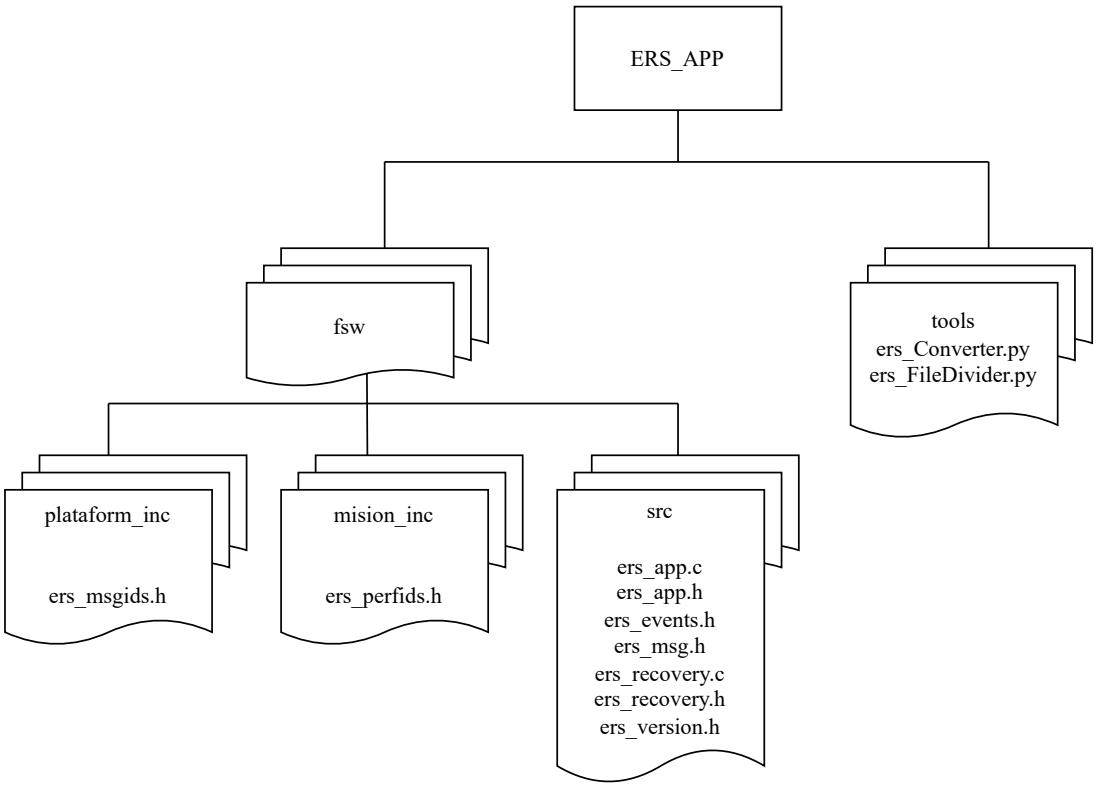


Figura 13.1: Directorio de archivos de ERS_APP.

El software está preparado con las definiciones y ciclos para respaldar cuatro aplicaciones de cFS configurables por el usuario, sin embargo, es posible aumentar este número solo copiando y pegando,

ya que se programó solo con cuatro para probar el concepto. Para esto, a cada aplicación y señal de alivio del *watchdog* se le asignó un identificador (tabla 13.2) el cual es enviado al supervisor, lo que le permite saber de qué aplicación debe de enviar la información.

Aplicación	ID en el supervisor
APP_RECOV1	A
APP_RECOV2	B
APP_RECOV3	C
APP_RECOV4	D
Señal de alivio	#

Tabla 13.2: Identificador de señales en el supervisor.

13.2 Supervisor

Para manejar la memoria CY15B108QN se configuró uno de los puertos SPI del microcontrolador para enviar y recibir bytes, dado que las memorias permiten leer y escribir múltiples bytes en una sola operación sin desactivar el *chip select*, este pin se configuró con un GPIO ya que el predeterminado solo permite leer hasta 16 bits. Primero, es necesario conocer su hoja de datos [63], en donde se indica que la memoria trabaja por medio de operaciones definidas por comandos *opcode* que son instrucciones en formato hexadecimal que definen las acciones que la memoria debe ejecutar, los usados para este proyecto son:

- ☐ WREN (0x06): habilita la escritura de la memoria
- ☐ WRITE (0x02): inicia escritura de datos en la memoria
- ☐ READ (0x03): lee los datos de la memoria

En la escritura, primero se debe habilitar con el *opcode* **WREN**, después, se envía el comando **WRITE**, seguido de 3 bytes de dirección y por último los datos a grabar. Para la lectura se utiliza el comando **READ**, segsuido de los 3 bytes de dirección.

En el software realizado, al momento de recibir un caracter en el buffer de datos del UART, se activa la interrupción **UCRXIE**, se lee el byte recibido y se compara. En caso de ser la `APP_RECOV1` asociada al identificador **A**, se comienza a leer la FRAM desde la dirección 0x00; cada byte leído, es transmitido de inmediato por UART y una vez concretada la transmisión se lee el siguiente byte. Este proceso se realiza n veces, dependiendo del tamaño de la app.

Parte IV

PRUEBAS Y RESULTADOS

14 Pruebas

Las pruebas se realizaron de forma modular (la OBC y el supervisor). Una vez validado el correcto funcionamiento modular, se integraron en único sistema y se realizó la prueba en conjunto.

14.1 Pruebas modulares

14.1.1 Prueba de OBC

Para probar la OBC se utilizó una Raspberry Pi 4B (figura 14.1) con Raspberry Pi OS Legacy 2025 de 64-bit, el cual es una distribución de Linux basada en Debian pero optimizada para Raspberry Pi, por lo que es totalmente compatible con cFS, además de que, como se puede ver en el estado del arte las Raspberry Pi son usadas como OBC de nanosatélites.

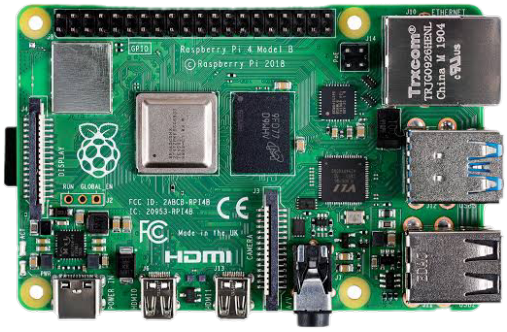
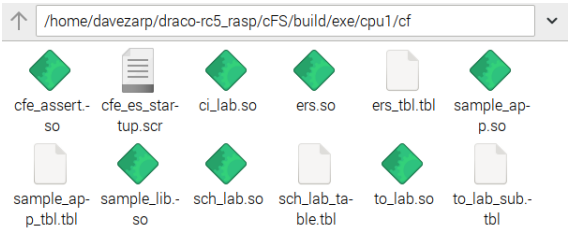
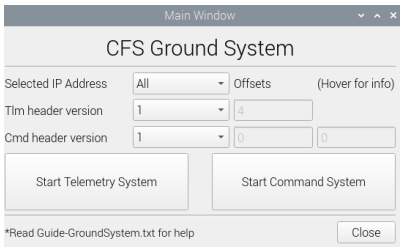


Figura 14.1: Raspberry Pi 4B, computadora portátil.

Se descargó e instaló la versión Draco-rc5 de cFS en una Raspberry Pi4B y posteriormente se incluyó la aplicación ERS (véase anexo A). Se probó el sistema con la aplicación de `Sample_app` y después se cumplió con cFS, se copió el archivo de `sample_app.so` ubicado en la dirección `cFS/build/exe/cpu1/cf` a la carpeta *tools* dentro de ERS (`cFS/apps/ers/tools`) con el objetivo de obtener todos los bytes del archivo ejecutable dentro de un arreglo de datos, para esto se ejecutó el algoritmo de python `ERS_Converter.py` para llenar el arreglo de *DATA*, el cual que se copió dentro de la función `ERS_FillSupport()`, que se encuentra comentada por defecto en `ers_recovery.h` y `ers_recovery.c` y la cual deberá ser activada, mientras que se tiene que desactivar la función de `ERS_FillData()`, ya que dicha función es la que se comunica con el supervisor y se usará hasta la prueba integral. Para simular la corrupción de datos en el archivo ejecutable de la aplicación `Sample_App` se editaron bits de forma aleatoria del archivo ejecutable dentro la carpeta *cf* (véase la figura 14.2a).



(a) Librerías compartidas de cFS.



(b) Herramienta de *Ground-System* de cFS.

Figura 14.2

Es necesario tener control sobre el encendido y apagado de la aplicación para forzar su actualización, para esto se hace uso de la herramienta *cFS Ground System* (figura 14.2b) proporcionada por

la NASA, como parte del *cFS bundle pack*, con ella es posible enviar comandos a las aplicaciones y servicios básicos de cFS, además permite recibir telemetría. La conexión por defecto entre el FSW y *Ground System* (o estación terrena) se realiza por red, ya sea inalámbrica (WiFi), o física (ethernet), basta con introducir la correcta dirección IP para una conexión exitosa. Cuando se trabaja de forma local, es decir, la estación terrena y el FSW en un mismo dispositivo, la dirección IP predeterminada es **127.0.0.1**. En concreto, se usaron los comandos: `CFE_ES_START_APP_CC` y `CFE_ES_STOP_APP_CC` (figura 14.3) para detener la aplicación y posteriormente intentar iniciarla con los datos corruptos, además, de probar reiniciar la aplicación de forma directa con el comando `CFE_ES_RESTART_APP_CC` (figura 14.4).

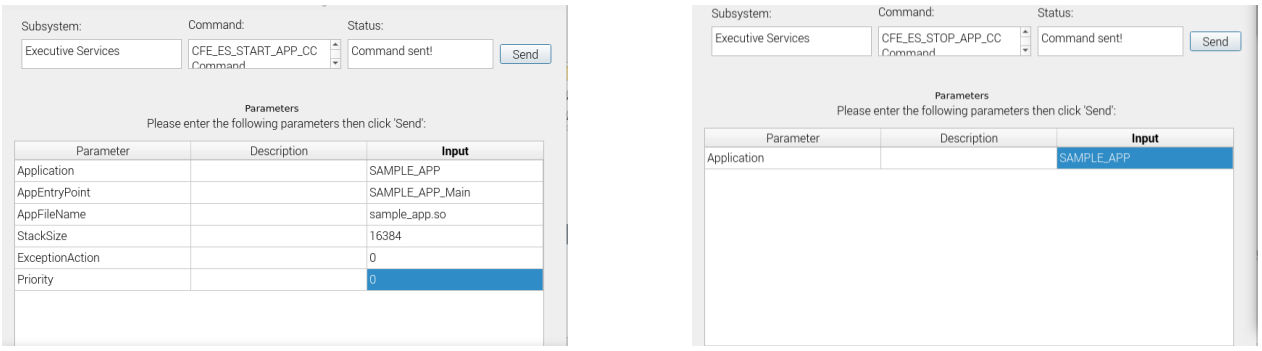


Figura 14.3: Configuración de comandos para inicio y apagado.

Figura 14.4: Configuración de comandos de reinicio de aplicación.

Ya que se requieren probar todas las funcionalidades del software después de la recuperación de la aplicación dañada, existen dos formas integradas para iniciar la aplicación: el reinicio de todo cFE o únicamente el lanzamiento de la aplicación recién recuperada. Esto se configura en el archivo de cabecera de `ers_recovery.h` en la definición de `START_RECOVx`, que de tomar un valor de cero se reinicia o inicia únicamente la aplicación y con un valor de 1, se reinicia todo el sistema. Para probar la segunda funcionalidad es necesario configurar la Raspberry para que el FSW se reinice automáticamente, para esto se creó un servicio, *cfs.service* de *systemd*, a través del comando en la terminal:

Terminal

```
sudo nano /etc/systemd/system/cfs.service
```

El cual incluyó la siguiente información:

cfs.service

```
[Unit]
Description=Core Flight System
After=network.target

[Service]
ExecStart=/home/davezarp/draco-rc5_rasp/cFS/build/exe/cpu1/
core-cpu1
workingDirectory=/home/davezarp/draco-rc5_rasp/cFS/build/
exe/cpu1
Restart=always

[Install]
WantedBy=multi-user.target
```

Una vez creado el servicio, la Raspberry se encarga de mantener abierto en todo momento cFS, esto se puede probar con el comando de reinicio de la estación terrena `CFE_ES_RESTART_CC` (figura 14.5).

Una vez ha sido configurada adecuadamente la Raspberry Pi, se puede corroborar el correcto funcionamiento de la aplicación ante la detección de errores con sus distintos modos de recuperación configurables en `ERS_recovery.h` (véase anexo C), por inicio exclusivo de la aplicación o por medio del reinicio total del FSW.

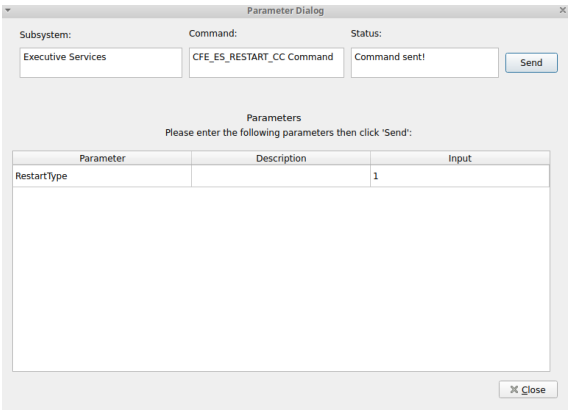


Figura 14.5: Configuración de comando para reinicio total de FSW.

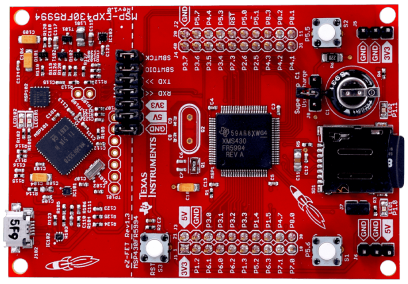
14.1.2 Prueba del supervisor

A pesar de especificarse la MSP430FR5969 como microcontrolador para el sistema supervisor, para las pruebas se utilizó una tarjeta de desarrollo con el integrado MSP430FR5994 (14.6a) debido a su disponibilidad, sin embargo, el código desarrollado y la funcionalidad es totalmente compatible con la versión especificada.

Para la FRAM, se utilizó el módulo *FRAM 2 click* (figura 14.6b) de la marca *Mikroe*, que usa el integrado CY15B104Q el cual es de la misma familia que la CY15B108Q, pero de menor capacidad y obsoleta por lo que es de un precio aún más accesible, sin embargo, su programación es la misma y es totalmente compatible incluso en empaquetado con su contraparte especificada.

Para la prueba se escribió en la memoria haciendo uso de la función `FRAM_Write_Byte()` proporcionada por el equipo de software del LIESE, con el arreglo *DATA* de `sample_app.so`, obtenido en la prueba de la OBC. Sin embargo, surgió el problema al momento de llenar la memoria, ya que *Sample_app* consta de 26144 bytes y al grabar la memoria desde un microcontrolador MSP430FR5994, el compilador indica que solo se tienen 4096 bytes para este tipo de datos. Por lo tanto, el archivo

original de 26144 bytes se partió en 15 archivos más pequeños con ayuda de un programa de python (anexo D) y se llenó uno a uno tomando en cuenta sus direcciones de memoria de inicio y final (tabla 14.1) para no sobrescribir los datos.



(a) Tarjeta de desarrollo MSP430FR5994 [64].

(b) FRAM CY15B104-Q [65].

Figura 14.6: Elementos del hardware para pruebas del supervisor.

Bloque	# bytes	Dirección inicial	Dirección final
1	0–1800	0x0000	0x0708
2	1801–3601	0x0709	0x0E11
3	3602–5402	0x0E12	0x151A
4	5403–7203	0x151B	0x1C23
5	7204–9004	0x1C24	0x232C
6	9005–10805	0x232D	0x2A35
7	10806–12606	0x2A36	0x313E
8	12607–14407	0x313F	0x3847
9	14408–16208	0x3848	0x3F50
10	16209–18009	0x3F51	0x4659
11	18010–19810	0x465A	0x4D62
12	19811–21611	0x4D63	0x546B
13	21612–23412	0x546C	0x5B74
14	23413–25213	0x5B75	0x627D
15	25214–26143	0x627E	0x662F

Tabla 14.1: Direcciones en memoria FRAM de *SampleApp*.

Se probó la lectura de la memoria con la función `FRAM_Read_Byte()` para comprobar que la memoria fue grabada de forma correcta, poniendo atención a los bytes que se encuentran en las fronteras de cada bloque para garantizar que no ocurrió un error en la escritura que pudiera causar un empalme entre bloques.

14.2 Prueba integral

Una vez comprobado que el FSW se comportaba de la forma esperada, y la tarjeta de desarrollo de MSP430 en conjunto con la memoria de respaldo operan de forma esperada, se integraron en un

mismo sistema, añadiendo además, la computadora portátil usada en el desarrollo de software para actuar como estación terrena.

Para realizar la conexión de la estación terrena con la OBC se conectaron a la misma red WiFi con direcciones IP 192.168.0.237 y 192.168.0.218 respectivamente, para visualizar la telemetría también se usó la herramienta de cFS (figura 14.2b), la cual debe configurarse con el comando `TO_LAB_OUTPUT_ENABLE_CC`, enviando como parámetro la dirección IP de la computadora en donde se recibirá la telemetría (figura 14.7). Para visualizar los eventos enviados desde la OBC se usa el *Start Telemetry System* y *Dsiplay Page* de *Event Messages* (figura 14.8).

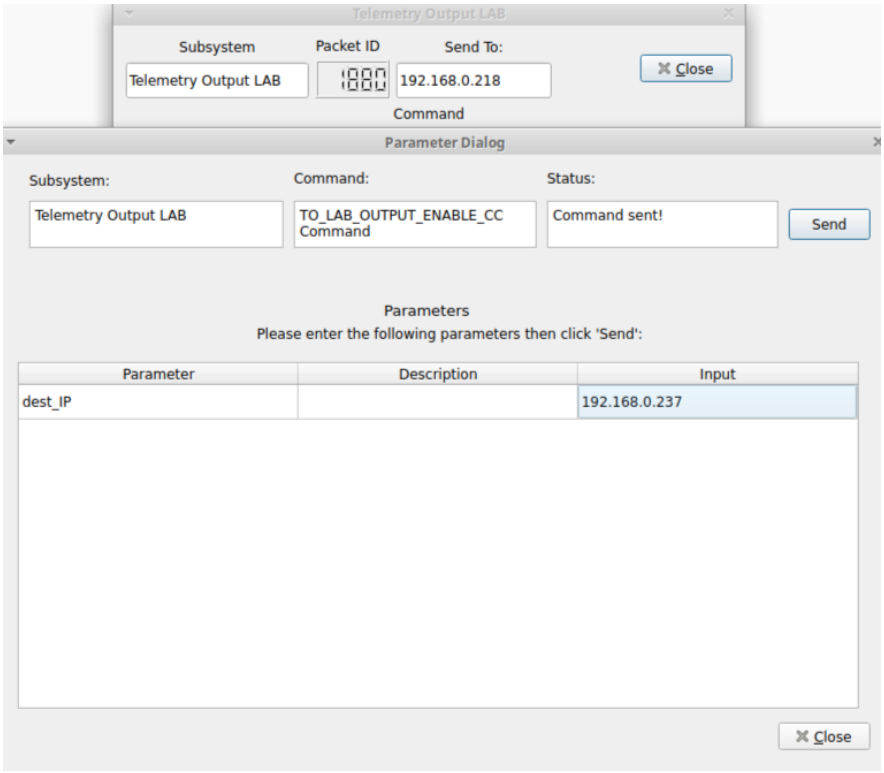


Figura 14.7: Configuración para el envío de telemetría.

Una vez configurada la telemetría es posible ver todos los eventos de tipo error y crítico desde la estación terrena, sin embargo, si se quisieran ver los de tipo información es necesario retirar el filtro a través del comando `CFE_EVS_FILTER_CC`.

Una vez preparada la estación terrena y la OBC, se conectan la Raspberry Pi y la tarjeta de desarrollo MSP430FR5994 por UART, junto con la memoria FRAM al supervisor por SPI de USCB1, el diagrama de conexiones se muestra en la figura 14.9. Como se trata de una prueba en un ambiente no hostil, no se esperan fallas naturales, por lo que no se requiere simularlas modificando la librería compartida de la aplicación de forma manual.

La prueba integral se realizó también con la aplicación *Sample_app* ya que la memoria FRAM ya se encuentra programada con la información necesaria. Para esta prueba, el código proveniente de `recovery.h` y `recovery.c` se usó sin modificaciones desde el repositorio de ERS app, con la función `ERS_FillSupport()` comentada y `ERS_FillData()` activada, configurada con *Sample_app*

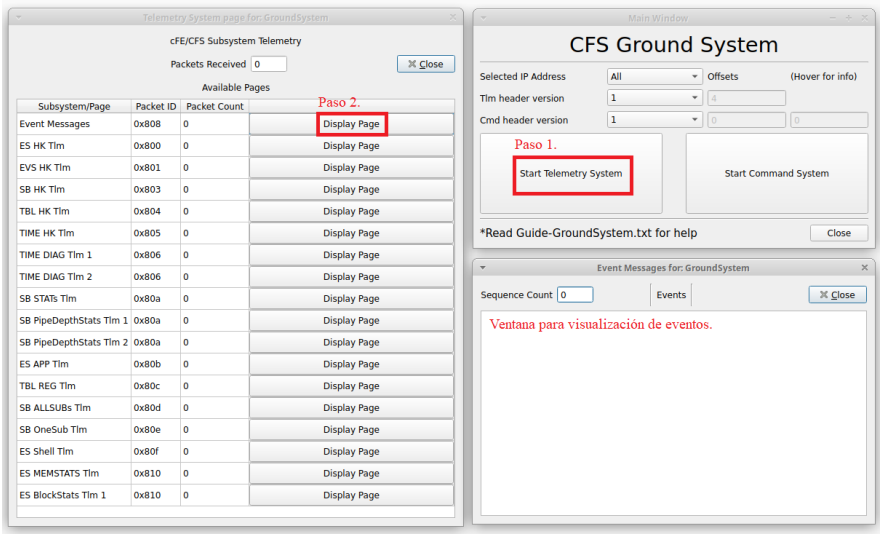


Figura 14.8: Configuración para la recepción de telemetría.

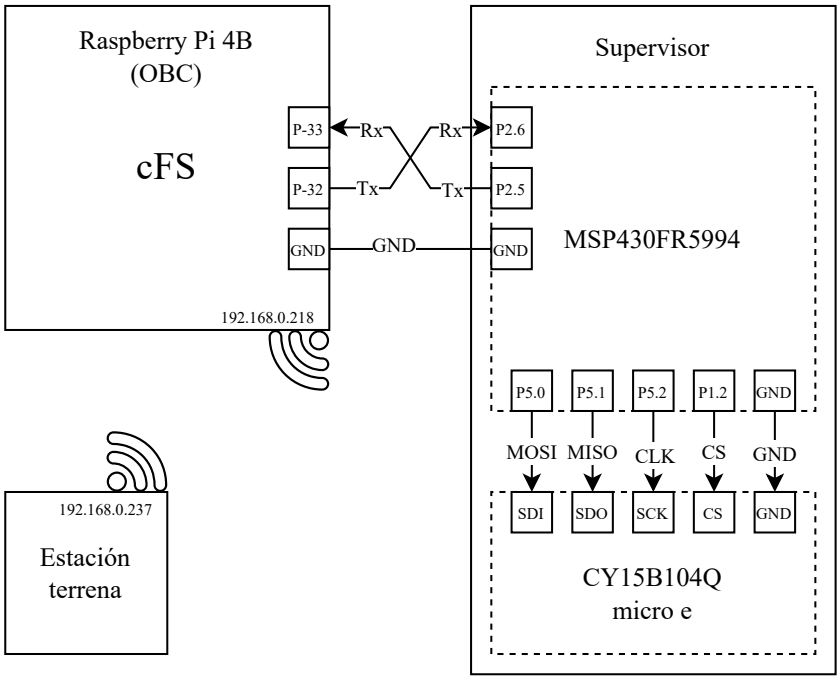


Figura 14.9: Diagrama de conexiones de prueba integral.

como la aplicación de recuperación 1 y por lo tanto con los parámetros de la tabla 14.2. Se guardan todos los cambios y se compila como se indica en el anexo A.

Antes de probar cFS y para garantizar una correcta conexión de UART entre el supervisor y la OBC, se usó la herramienta de *CuteCom* (figura 14.10) desde la cual se envió el caracter **A**, que de acuerdo con el código ASCII, se representa con el número hexadecimal **0x41**, caracter que corresponde a la primera APP de recuperación (tabla 13.2), el cual, al momento de ser enviado

Parámetro	Sample App
APP_RECOV1	SAMPLE_APP
FILENAME_RECOV1	/cf/sample_app.so
MAIN_RECOV1	SAMPLE_APP_Main
STACK_RECOV1	16384
PRIORITY_RECOV1	50

Tabla 14.2: Parámetros para Sample_app en `ers_recovery.h`.

comienza a recibir la respuesta de inmediato, en este caso los 26,314 bytes correspondientes al respaldo de la aplicación *Sample_app*.

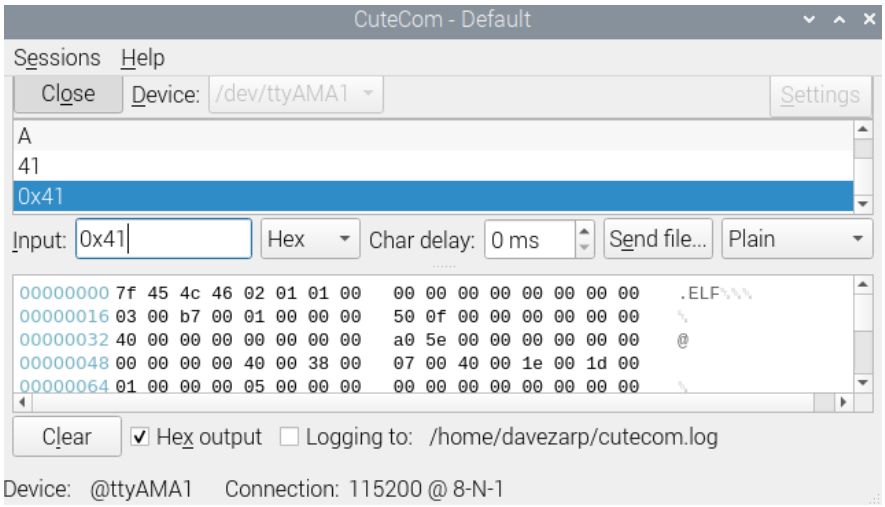


Figura 14.10: Prueba *CuteCom*.

14.2.1 Prueba básica de recuperación de aplicación

Una vez concluida la preparación del sistema y validar las conexiones de puerto serie, se envía un comando de no operación del servicio ES y otro NOOP para *Sample_app* para corroborar que el servicio y la aplicación están operando de forma nominal. Posteriormente se cambia el archivo por el corrupto y se reinicia por medio de comandos, enviando de inmediato el evento de error a la estación terrena y el evento propio de ERS app que indica que el proceso de recuperación ha iniciado. Después de 3 minutos y 18 segundos se completa la transmisión y por lo tanto se restablece la aplicación dañada de forma automática tal y como se muestra en la tabla 14.3, que resume la información proporcionada por el registro de eventos en la bitácora de cFS.

14.2.2 Prueba de recuperación de aplicación con otro proceso en ejecución

Dado de que cFS es un programa que opera de forma concurrente, se realizó la misma prueba de reconfiguración de una aplicación dañada mientras se ejecuta otro proceso. Para esto, se utilizó la estación terrena para enviar un comando de no operación de *Executive Service*, el cual se procesó de forma normal al mismo tiempo que se realizaba la recepción de datos del supervisor (tabla 14.4).

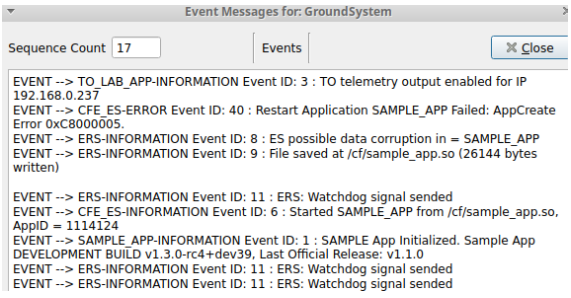
t_x	$t_x - t_0$	Emisor	ID	Contenido
t_0	0	CFE_ES	40	Restart Application SAMPLE_APP Failed
t_1	150 μs	ERS App	8	Possible data corruption in = SAMPLE_APP
t_2	3min 18.05s	ERS App	9	File saved at /cf/sample_app.so (26144 bytes written)
t_3	3min 18.10s	SAMPLE_APP	1	SAMPLE App Initialized

Tabla 14.3: Registro de la bitácora de cFS durante la prueba básica.

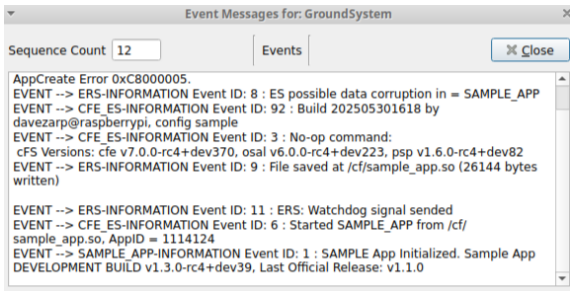
t_x	$t_x - t_0$	Emisor	ID	Contenido
t_0	0	CFE_ES	40	Restart Application SAMPLE_APP Failed
t_1	140 μs	ERS App	8	Possible data corruption in = SAMPLE_APP
t_2	19.75s	CFE_ES	3	No-op command
t_3	3min 18.19s	ERS App	9	File saved at /cf/sample_app.so (26144 bytes written)
t_4	3min 18.24s	SAMPLE_APP	1	SAMPLE App Initialized

Tabla 14.4: Registro de la bitácora de cFS durante la prueba con otro proceso en ejecución.

Los resultados de la prueba básica y de la prueba con otro proceso en ejecución también se pueden observar desde el registro de eventos de la estación terrena, en donde se puede ver que el proceso de recuperación se realizó de forma correcta a través de la recepción de telemetría, como se muestra en la figura 14.11.



(a) Telemetría de la prueba básica.



(b) Telemetría de la prueba comando noop.

Figura 14.11: Telemetría de las pruebas de recuperación recibida en la estación terrena.

Dado que la Raspberry Pi 4-B no cuenta con un pin exclusivo para realizar un reinicio del sistema, no se probó de forma directa, sin embargo, en la figura 14.12 se muestra cómo la señal

de alivio se envía de forma periódica cada 4 minutos, un valor seleccionado de forma aleatoria, únicamente con el motivo de probar que la señal sea enviada. Dependerá del usuario la selección de la frecuencia de envío de esta señal según sus necesidades.

```
May 30 19:23:20 raspberrypi core-cpu1[5922]: EVS Port1 1980-012-20:37:11.70033 66/1/ERS 11: ERS: Watchdog signal sended
May 30 19:25:09 raspberrypi core-cpu1[5922]: EVS Port1 1980-012-20:39:00.90016 66/1/CFE_ES 92: Build 202505301618 by dav
ezarp@raspberrypi, config sample
May 30 19:25:09 raspberrypi core-cpu1[5922]: EVS Port1 1980-012-20:39:00.90026 66/1/CFE_ES 3: No-op command:
May 30 19:25:09 raspberrypi core-cpu1[5922]:  cFS Versions: cfe v7.0.0-rc4+dev370, osal v6.0.0-rc4+dev223, psp v1.6.0-rc
4+dev82
May 30 19:27:20 raspberrypi core-cpu1[5922]: EVS Port1 1980-012-20:41:11.70075 66/1/ERS 11: ERS: Watchdog signal sended
May 30 19:31:20 raspberrypi core-cpu1[5922]: EVS Port1 1980-012-20:45:11.70053 66/1/ERS 11: ERS: Watchdog signal sended
May 30 19:35:20 raspberrypi core-cpu1[5922]: EVS Port1 1980-012-20:49:11.70059 66/1/ERS 11: ERS: Watchdog signal sended
```

Figura 14.12: Prueba del *watchdog*.

En las pruebas realizadas, se observó que la aplicación de recuperación *ERS_App* cumple su función, entrando en acción de forma automática al detectar un error en la aplicación en tan solo 150μs. Y aunque el proceso de recuperación puede tardar hasta 3 minutos y 20 segundos, la aplicación de recuperación no interfiere con el funcionamiento de cFS, ya que permite ejecutar otras tareas de forma concurrente. Hay que considerar que el tiempo de recuperación, aunque parezca largo, es aceptable ya que el tiempo que tardaría en corregir el mismo error de forma manual, sería mucho mayor, al tener que esperar que el satélite se comunique con la estación terrena.

15 Resultados

En la tabla 12.2 se muestra que el tamaño de almacenamiento requerido para cada aplicación en cFS Draco-rc5 compilado para una Raspberry Pi-4, es de 2.39MB si se consideran todas las aplicaciones y 1.83 MB si solo se utilizaran las aplicaciones incluidas en el *cFS bundle pack*. Dado que la aplicación desarrollada necesita de 0.0352MB, aumentar el nivel de fiabilidad de cFS requiere de incrementar en 1.92 % el almacenamiento si se usan todas las apps o 2.59 % si se considera únicamente el *cFS bundle pack*.

Considerando un voltaje de alimentación de 3.3V y que el MSP430FR5969 en estado activo consume 0.8mA, además de que las memorias consumen hasta 2.6mA en estado activo, se estima un consumo activo de potencia de 11.2mW. Recordando que el consumo promedio de una computadora de embebida está en 2.95 W (véase la tabla 9.2), el impacto del sistema supervisor está en 0.38 %, cumpliendo con los requisitos de diseño.

De manera general, se resume el proceso de reconfiguración diseñado e implementado en la figura 15.1, en donde se puede apreciar que el proceso involucra únicamente ERS app, con el supervisor y los servicios básicos de cFE. En comparación con el sistema existente (figura 8.4), que necesita de 4 aplicaciones, la estación terrena y los servicios básicos de cFE, ERS app no solo utiliza menos pasos haciendo más eficiente el proceso, también es más segura al ser autónoma ya que no requiere de la intervención de la estación terrena.

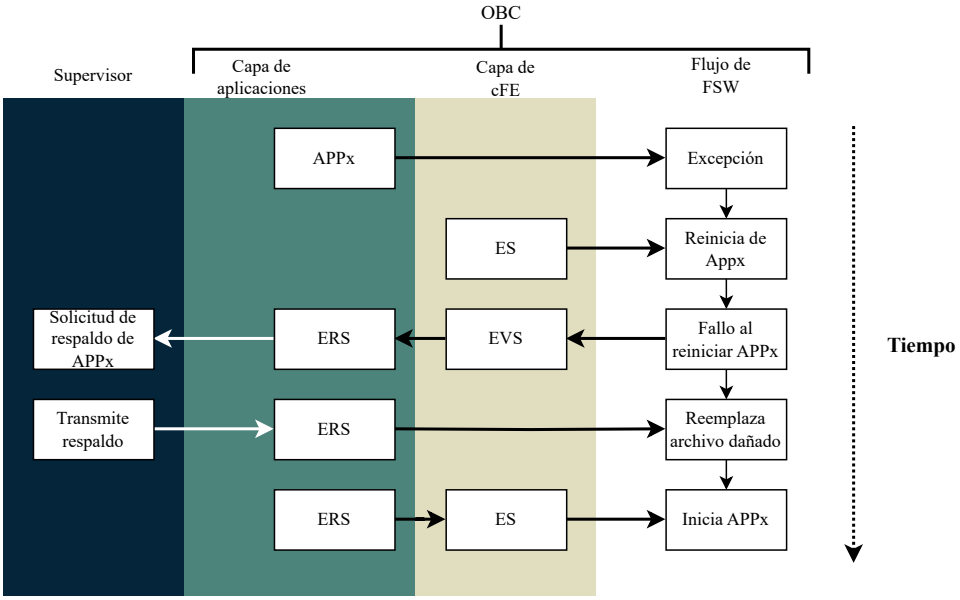


Figura 15.1: Proceso de reconfiguración de aplicaciones de cFS gestionado por ERS app.

De manera general, se puede concluir que los resultados obtenidos muestran que el sistema de recuperación de aplicaciones de cFS, diseñado e implementado en este trabajo, cumple con los requisitos de diseño establecidos, permitiendo la recuperación de aplicaciones dañadas de forma autónoma y segura, sin interferir con el funcionamiento nominal de cFS. Además, al ser un sistema no invasivo (potencia y almacenamiento) permite una fácil integración en cualquier OBC para nanosatélites, que use cFS como su software de a bordo.

Parte V

CONCLUSIONES

16 Conclusiones

Se ha visto cómo la exploración espacial fue, es y seguirá siendo de gran interés para el ser humano. Hoy más que nunca, gracias a las nuevas tecnologías, más organizaciones (universidades y pequeñas empresas) pueden acceder a sus beneficios con los nanosatélites gracias a su bajo costo de desarrollo, sin embargo, la radiación espacial representa un reto al utilizar componentes COTS en estos dispositivos, por lo que se requiere de técnicas de tolerancia a fallas para garantizar una misión exitosa.

El *framework* cFS de la NASA permite a todo tipo de organizaciones realizar FSW de gran calidad en un menor tiempo, sin embargo, para misiones de clase A o B, no es recomendable por sí mismo, ya que no garantiza ser un sistema tolerante a fallas, pero proporciona las herramientas para implementarse, gracias a su capacidad para la detección de fallas.

En este trabajo se planteó el diseño de un sistema supervisor que complementa el software de vuelo desarrollado con cFS con una app compatible con este *framework*. Esta app aprovecha la detección de errores de cFS para determinar cuando la información del ejecutable de una aplicación almacenada en la OBC ha sido dañada, para restablecerla de forma autónoma. El sistema diseñado para abordar la problemática de fiabilidad del cFS tiene un impacto mínimo de procesamiento, de almacenamiento (con 2.59 % adicional para una pequeña misión) y un consumo de potencia activa adicional de 11.2mW. Además, resulta ser un sistema redundante en software, hardware e información.

El sistema diseñado tiene la capacidad de funcionar como un sistema *watchdog*, el cual se considera como un sistema de **redundancia activa de hardware** que detecta y corrige algunas fallas, además de contar con memorias FRAM endurecidas a la radiación proporcionando hardware adicional para el almacenamiento de las aplicaciones críticas. Por otro lado, el sistema añade **redundancia en software**, ya que la aplicación ERS cumple con funciones similares a *File Manager app*, *CFDP app*, *Housekeeping app*, que en conjunto, permiten la reconfiguración de módulos de cFS. Sin embargo, ERS app sigue un flujo de trabajo distinto, además depende únicamente de los servicios de cFE para restaurar aplicaciones dañadas reduciendo su probabilidad de falla. Además, el software desarrollado contiene un manejo de excepciones en el caso de fallas. Por último, el diseño agrega **redundancia en información**, ya que duplica la información de los archivos ejecutables de las aplicaciones críticas de cFS, llevando a bordo en la OBC en la memoria persistente del sistema y además el sistema supervisor mantiene una copia en el banco de FRAM con una mayor tolerancia a la radiación que las memorias *flash*, asegurando la integridad de la información.

En conclusión, en este trabajo de tesis se puede apreciar el desarrollo de un sistema autónomo que es poco invasivo, de fácil integración, con un impacto mínimo (en almacenamiento, procesamiento y potencia energética) y de bajo costo económico, que permite a una OBC con FSW basado en cFS incrementar su nivel de fiabilidad considerablemente, independientemente de la plataforma que se utilice.

17 Trabajo futuro

Como trabajo futuro se necesita realizar un análisis de fiabilidad para obtener, de forma cuantitativa, el impacto en el nivel de fiabilidad de una OBC COTS que ejecute cFS al añadir el sistema supervisor. Además, se realizará un análisis del medio ambiente de radiación espacial para determinar la frecuencia en la ocurrencia de fallas esperadas y, con esa información, establecer el tiempo óptimo para enviar la señal de alivio del *watchdog* y cambiar su modelo de funcionamiento para evitar el fenómeno de *fast watchdog*.

En este trabajo se le da respuesta a 5 errores que están asociados a la corrupción de datos en las aplicaciones de cFS, sin embargo, existen 187 errores reportados por los servicios de cFE, entre los cuáles es posible detectar errores de sincronización temporal, corrupción de datos en tablas de configuración, corrupción de datos en la transmisión de mensajes del *Software Bus* y desbordamientos de colas de mensajes, para los cuales pueden implementarse estrategias con un supervisor externo para mitigar dichos errores. Por lo tanto, es necesario identificar los problemas en el funcionamiento de cFS cuando suceden todos los eventos de falla y, de esta manera, implementar métodos que permitan corregirlos.

Dado que cFS busca la facilidad para el usuario de diseñar software, se plantea una herramienta (*tool*) dentro de ERS APP, que permita la configuración de las memorias FRAM de respaldo y la configuración del supervisor, que incluye cuántas aplicaciones se van a respaldar, el tamaño y la dirección de memoria de cada una de ellas, así como la duración del temporizador y el número máximo de reinicios que este realizará.

Referencias

[1] *Why Go to Space - NASA*. (visitado 01-05-2025).

[2] A. Cratere, L. Gagliardi, G. A. Sanca, F. Golmar y F. Dell'Olio, «On-Board Computer for CubeSats: State-of-the-Art and Future Trends,» *IEEE Access*, vol. 12, págs. 99 537-99 569, 2024, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2024.3428388. (visitado 16-01-2025).

[3] *ISO/IEC 2382-14:1997(En), Information Technology— Vocabulary— Part 14: Reliability, Maintainability and Availability*, https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:-14:ed-2:v1:en. (visitado 31-05-2025).

[4] M. Macdonald y V. Badescu, eds., *The International Handbook of Space Technology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ISBN: 978-3-642-41100-7 978-3-642-41101-4. DOI: 10.1007/978-3-642-41101-4. (visitado 20-01-2025).

[5] E. Kulu, *Nanosats Database*, https://www.nanosats.eu/index.html. (visitado 31-03-2025).

[6] I. M. Siddique, «Emerging Trends in Small Satellite Technology: Challenges and Opportunities,» feb. de 2024, ISSN: 2394-658X. DOI: 10.5281/ZENODO.12748476. (visitado 16-01-2025).

[7] D. McComas, «NASA/GSFC's Flight Software Core Flight System,» en *Flight Software Workshop-Flight Workshop*, 2012.

[8] S. J. Johnston, P. J. Basford, C. S. Perkins et al., «Commodity single board computer clusters and their applications,» *Future Generation Computer Systems*, vol. 89, págs. 201-212, dic. de 2018, ISSN: 0167739X. DOI: 10.1016/j.future.2018.06.048. (visitado 04-05-2025).

[9] *ISO/IEC 2382-14:1997(En), Information Technology— Vocabulary— Part 14: Reliability, Maintainability and Availability*, https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:-14:ed-2:v1:en. (visitado 02-06-2025).

[10] K. T. Ulrich, S. D. Eppinger y M. C. Yang, *Product Design and Development*, Seventh edition, International student edition. New York, NY: McGraw-Hill, 2020, ISBN: 978-1-260-04365-5 978-1-260-56954-4.

[11] *Stonehenge — History, Location, Map, Meaning, & Facts — Britannica*, https://www.britannica.com/topic/Stonehenge, mar. de 2025. (visitado 02-05-2025).

[12] *Chichen Itza — Description, Buildings, History, & Facts — Britannica*, https://www.britannica.com/place/Chichen-Itza, mar. de 2025. (visitado 02-05-2025).

[13] J.-L. Autran y D. Munteanu, *Soft Errors: From Particles to Circuits* (Devices, Circuits, and Systems Ser v.39), K. Iniewski, ed. Boca Raton London New York: CRC Press, an imprint of the Taylor & Francis Group, 2015, ISBN: 978-1-4665-9084-7.

[14] *Magnetospheres - NASA Science*, jun. de 2007. (visitado 04-05-2025).

[15] S. Duzellier, «Radiation effects on electronic devices in space,» *Aerospace Science and Technology*, vol. 9, n.º 1, págs. 93-99, ene. de 2005, ISSN: 12709638. DOI: 10.1016/j.ast.2004.08.006. (visitado 09-05-2025).

[16] T. Kuwahara, Y. Tomioka, K. Fukuda, N. Sugimura e Y. Sakamoto, «Radiation Effect Mitigation Methods for Electronic Systems,» en *2012 IEEE/SICE International Symposium on System Integration (SII)*, Fukuoka, Japan: IEEE, dic. de 2012, págs. 307-312, ISBN: 978-1-4673-1497-8 978-1-4673-1496-1 978-1-4673-1495-4. DOI: 10.1109/SII.2012.6427324. (visitado 09-05-2025).

[17] V. Castano e I. Schagaev, *Resilient Computer System Design*. Cham: Springer International Publishing, 2015, ISBN: 978-3-319-15068-0 978-3-319-15069-7. DOI: 10.1007/978-3-319-15069-7. (visitado 21-04-2025).

[18] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems* (Addison-Wesley Series in Electrical and Computer Engineering). Reading, Mass: Addison-Wesley Pub. Co, 1989, ISBN: 978-0-201-07570-0.

[19] I. Koren y C. M. Krishna, *Fault-Tolerant Systems*, Second edition. Cambridge, MA: Morgan Kaufmann Publishers, 2021, ISBN: 978-0-12-818105-8.

[20] A. M. El-Attar y G. Fahmy, «An Improved Watchdog Timer to Enhance Imaging System Reliability In The Presence Of Soft Errors,» en *2007 IEEE International Symposium on Signal Processing and Information Technology*, Giza, Egypt: IEEE, dic. de 2007, págs. 1100-1104, ISBN: 978-1-4244-1834-3 978-1-4244-1835-0. DOI: 10.1109/ISSPIT.2007.4458184. (visitado 19-04-2025).

[21] E. Dubrova, *Fault-Tolerant Design*. New York, NY: Springer New York, 2013, ISBN: 978-1-4614-2112-2 978-1-4614-2113-9. DOI: 10.1007/978-1-4614-2113-9. (visitado 02-05-2025).

[22] H. R. Slotten, *Beyond Sputnik and the Space Race: The Origins of Global Satellite Communications*. Baltimore, Maryland: Johns Hopkins university press, 2022, ISBN: 978-1-4214-4122-1.

[23] NASA, *What Is a Satellite?* Sep. de 2018. (visitado 04-02-2025).

[24] H. J. Kramer y A. P. Cracknell, «An overview of small satellites in remote sensing,» *International Journal of Remote Sensing*, vol. 29, n.º 15, págs. 4285-4337, ago. de 2008, ISSN: 0143-1161, 1366-5901. DOI: 10.1080/01431160801914952. (visitado 04-02-2025).

[25] J. Rendleman, «Why SmallSats?» En *AIAA SPACE 2009 Conference & Exposition*, Pasadena, California: American Institute of Aeronautics and Astronautics, sep. de 2009, ISBN: 978-1-60086-980-8. DOI: 10.2514/6.2009-6416. (visitado 05-02-2025).

[26] T. Wekerle, J. B. Pessoa Filho, L. E. V. L. D. Costa y L. G. Trabasso, «Status and Trends of Smallsats and Their Launch Vehicles — An Up-to-date Review,» *Journal of Aerospace Technology and Management*, vol. 9, n.º 3, págs. 269-286, ago. de 2017, ISSN: 2175-9146. DOI: 10.5028/jatm.v9i3.853. (visitado 16-01-2025).

[27] Cal Poly, *CubeSat Design Specification*, 2022.

[28] M. Tipaldi, C. Legendre, O. Koopmann, M. Ferraguto, R. Wenker y G. D'Angelo, «Development strategies for the satellite flight software on-board Meteosat Third Generation,» *Acta Astronautica*, vol. 145, págs. 482-491, abr. de 2018, ISSN: 00945765. DOI: 10.1016/j.actaastro.2018.02.020. (visitado 18-03-2025).

[29] D. José Franzim Miranda, M. Ferreira, F. Kucinskis y D. McComas, «A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions,» *Journal of Aerospace Technology and Management*, e4619, oct. de 2019, ISSN: 2175-9146. DOI: 10.5028/jatm.v11.1081. (visitado 18-12-2024).

[30] E. Gamma, ed., *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley Professional Computing Series). Reading, Mass: Addison-Wesley, 1995, ISBN: 978-0-201-63361-0.

[31] *ISO/IEC 2382-1:1993(En), Information Technology— Vocabulary— Part 1: Fundamental Terms*, https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:-1:ed-3:v1:en:en%2001.04.08. (visitado 03-06-2025).

[32] A. S. Tanenbaum y H. Bos, *Modern Operating Systems*, 4. ed. Boston: Prentice Hall, 2015, ISBN: 978-0-13-359162-0 978-1-292-06142-9.

[33] K. C. Wang, *Embedded and Real-Time Operating Systems*. Cham: Springer International Publishing, 2023, ISBN: 978-3-031-28700-8 978-3-031-28701-5. DOI: 10.1007/978-3-031-28701-5. (visitado 03-06-2025).

[34] NASA's Goddard Space Flight Center, *Core Flight System (cFS)*, abr. de 2025. (visitado 01-04-2025).

[35] *cFS/Osal-Apiguide.Pdf at Gh-Pages · Nasa/cFS*, https://github.com/nasa/cFS/blob/gh-pages/osal-apiguide.pdf. (visitado 07-05-2025).

[36] *cFE/docs/cFE Application Developers Guide.md at main · nasa/cFE*, https://github.com/nasa/cFE/blob/main/docs/cFE%20Application%20Developers%20Guide.md.

[37] *¿En qué consisten los mensajes de publicación/suscripción? - Explicación sobre los mensajes de publicación/suscripción - AWS*, https://aws.amazon.com/es/what-is/pub-sub-messaging/. (visitado 16-04-2025).

[38] *Nasa/SCH*, NASA, ene. de 2025. (visitado 18-04-2025).

[39] NASA, «State-of-the-Art Small Spacecraft Technology,» Ames Research Center, inf. téc. TP—20240001462, feb. de 2024.

[40] SpaceTeamSat1, «CubeSat: SpaceTeamSat1 - Preliminary Design I: System Architecture.,» TU Wien Space Team., Vienna, Austria., inf. téc. 3.1. Feb. de 2021.

[41] H. Sukhwani, J. Alonso, K. S. Trivedi e I. Mcginnis, «Software Reliability Analysis of NASA Space Flight Software: A Practical Experience,» en *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Vienna, Austria: IEEE, ago. de 2016, págs. 386-397, ISBN: 978-1-5090-4127-5. DOI: 10.1109/QRS.2016.50. (visitado 18-10-2024).

[42] M. Eleffendi, D. Posada, M. I. Akbas y T. Henderson, *NASA/GSFC's Flight Software Core Flight System Implementation For A Lunar Surface Imaging Mission*, abr. de 2022. DOI: 10.48550/arXiv.2204.07015. arXiv: 2204.07015 [cs]. (visitado 05-12-2024).

[43] Mike Cavaliere, *EagleCam Updates: Embry-Riddle Device Lands on Moon.* Feb. de 2024. (visitado 05-12-2024).

[44] J. Park, J.-O. Lee, J. Kim et al., «Application of NASA Core Flight System to Telescope Control Software for 2017 Total Solar Eclipse Observation,» *Publications of the Astronomical Society of the Pacific*, vol. 134, n.º 1033, pág. 034 504, mar. de 2022, ISSN: 0004-6280, 1538-3873. DOI: 10.1088/1538-3873/ac5848. (visitado 18-10-2024).

[45] J. B. Olansen, S. R. Munday, J. D. Mitchell y M. Baine, «Morpheus: Advancing Technologies for Human Exploration.,» inf. téc. GLEX-2012.05.2.4 x12761, mayo de 2012.

[46] C. R. Tooley, M. B. Houghton, R. S. Saylor et al., «Lunar Reconnaissance Orbiter Mission and Spacecraft Design,» *Space Science Reviews*, vol. 150, n.º 1-4, págs. 23-62, ene. de 2010, ISSN: 0038-6308, 1572-9672. DOI: 10.1007/s11214-009-9624-4. (visitado 11-10-2024).

[47] K. C. Gendreau, Z. Arzoumanian, P. W. Adkins et al., «The Neutron Star Interior Composition Explorer (NICER): Design and Development,» en *SPIE Astronomical Telescopes + Instrumentation*, J.-W. A. Den Herder, T. Takahashi y M. Bautz, eds., Edinburgh, United Kingdom, jul. de 2016, 99051H. DOI: 10.1117/12.2231304. (visitado 18-10-2024).

[48] R. P. Ltd, *Buy a Raspberry Pi 4 Model B*, https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. (visitado 14-05-2025).

[49] *Astro Pi — A free challenge where kids run their code in space*, https://astro-pi.org/. (visitado 14-05-2025).

[50] R. P. Ltd, *Buy a Raspberry Pi Zero*, https://www.raspberrypi.com/products/raspberry-pi-zero/. (visitado 14-05-2025).

[51] U. S. University, *GASPACS CubeSat — Projects — GAS — Physics*, https://www.usu.edu/physics/gas/projects/gaspacs. (visitado 14-05-2025).

[52] *BeagleBone® Black*. (visitado 14-05-2025).

[53] *PocketBeagle® - BeagleBoard*, https://www.beagleboard.org/boards/pocketbeagle-original. (visitado 14-05-2025).

[54] C. Wicks, *OreSat Uses PocketBeagle® to Provide STEM Motivation With Giant "Selfie-Stick" from Space*, ago. de 2019. (visitado 14-05-2025).

[55] *NVIDIA Jetson TX2: IA de Alto Rendimiento en el Edge*, https://www.nvidia.com/es-la/autonomous-machines/embedded-systems/jetson-tx2/. (visitado 14-05-2025).

[56] *Colibri iMX6 — Toradex Developer Center*, https://developer.toradex.com/hardware/colibri-som-family/modules/colibri-imx6/. (visitado 14-05-2025).

[57] *ODROID-C4 – ODROID*, https://www.hardkernel.com/shop/odroid-c4/. (visitado 14-05-2025).

[58] J. R. Wertz, D. F. Everett y J. J. Puschell, eds., *Space Mission Engineering: The New SMAD* (Space Technology Library 28). Torrance: Microcosm Press, 2011, ISBN: 978-1-881883-15-9 978-1-881883-16-6.

[59] N. S. Bennett, A. Hawchar y A. Cowley, «Thermal Control of CubeSat Electronics Using Thermoelectrics,» *Applied Sciences*, vol. 13, n.º 11, pág. 6480, mayo de 2023, ISSN: 2076-3417. DOI: 10.3390/app13116480. (visitado 10-05-2025).

[60] E. Kreyszig, H. Kreyszig y E. J. Norminton, *Advanced Engineering Mathematics*, 10th ed. Hoboken, NJ: John Wiley, 2011, ISBN: 978-0-470-45836-5.

[61] *MSP430FR5969 Data Sheet, Product Information and Support — TI.Com*, https://www.ti.com/product/MSP430FR5969?utm_source=google&utm_medium=cpc&utm_campaign=epd-msp-null-44700045336317338_prodfolderdynamic-cpc-pf-google-ww_en_int&utm_content=prodfolddynamic&ds_k=DYNAMIC+SEARCH+ADS&DCM=yes&gad_source=1&gclid=EAIaIQobChMIu9bB19HZjAMVeCRECB1B-ziIEAAYASAAEgJp9_D_BwE&gclsrc=aw.ds. (visitado 15-04-2025).

[62] O. Gonzalez, Y. Chen, R. L. Ladbury y D. R. Morgan, «Guidelines for Verification Strategies to Minimize RISK Based On Mission Environment, -Application and –Lifetime (MEAL),» 2018.

[63] «CY15B108QN, CY15V108QN, 8Mb EXCELON™ LP Ferroelectric RAM (F-RAM) Serial (SPI), 1024K × 8, 40 MHz, industrial,»

[64] *MSP-EXP430FR5994 Development Kit — TI.Com*, https://www.ti.com/tool/MSP-EXP430FR5994?utm_source=google&utm_medium=cpc&utm_campaign=epd-msp-null-44700045336317329_prodfolderdynamic-cpc-pf-google-ww_en_int&utm_content=prodfolddynamic&ds_k=DYNAMIC+SEARCH+ADS&DCM=yes&gad_source=1&gad_campaignid=7213436380&gclid=EAIaIQobChMIppHY8pmkjQMV6iNECB1IGwPlEAAYASAAEgKb_fD_BwE&gclsrc=aw.ds. (visitado 15-05-2025).

[65] *FRAM 2 click - carries the CY15B104Q 4-Mbit (512K x 8) serial SPI F-RAM — MikroElektronika*, http://www.mikroe.com/fram-2-click?srsltid=AfmBOooUxdGcCBrMkYaRXxMkgSrOkeXzmpROve-5TGE-tfGBfIRa1O2L. (visitado 15-05-2025).

Parte VI

ANEXOS

A Instalación de cFS

A continuación se muestran los pasos para descargar e instalar cFS (draco-rc5) en un sistema operativo Linux, haciendo uso de la terminal o consola con privilegios de superusuario.

Clona cFS draco-rc5

```
git clone https://github.com/nasa/cFS.git
cd cFS
git checkout drac-rc5
git submodule init
git submodule update
```

Copia archivos de makefile y definiciones

```
cp cfe/cmake/Makefile.sample Makefile
cpr cfe/cmake/sample_defs sample_defs
```

Una vez instaldo cFS, se muestran los pasos para la descarga e instalación de ERS app, aplicación desarrollada en este trabajo de tesis.

Terminal

```
cd cFS/apps
```

Terminal

```
git clone https://github.com/DaveZarp/cFS_APP_ERS
```

Terminal

```
cd ../sample_defs
```

Abrir `cpu1_cfe_es_startupt.scr` y añade la declaración de inicio como se muestra a continuación:

cpu1_cfe_es_startupt.scr

```
CFE_APP, ers, ERS_APP_Main, ERS, 90, 64000, 0x0, 0;
```

Añadir en *targets.cmake.txt* el nombre de la nueva aplicación **ers** al final de la lista existente.

targets.cmake.txt

SET(cpu1_APPLIST ci_lab to_lab sch_lab **ers**)

Para compilar y ejecutar cFS de forma nativa:

Terminal

```
cd ..
make prep
make
make install
```

Terminal

```
cd build/exe/cpu1/
./corecpu1
```

B Aplicaciones de cFS

Aplicación	Función
CFDP	Transfiere y recibe archivos entre la estación terrena y el satélite.
Checksum	Realiza verificación de integridad de memoria, tablas y archivos.
Data Storage	Registra datos de mantenimiento, ingeniería y científicos a bordo para su descarga.
File Manager	Interfaz para la gestión de archivos del satélite desde la estación terrena.
Housekeeping	Recolecta y organiza telemetría de otras aplicaciones.
Health and Safety	Asegura la supervisión de tareas críticas, activa el *watchdog* de servicios, calcula el uso del CPU y detecta sobrecargas.
Limit Checker	Monitorea la telemetría y responde a violaciones de límites.
Memory Dwell	Permite a tierra recibir contenidos de ubicaciones de memoria. Útil para depuración.
Memory Manager	Proporciona la capacidad de cargar y volcar memoria.
Software Bus Network	Transfiere mensajes del bus de software a través de varios protocolos de red.
Scheduler	Programador de actividades simple.
Stored Command	Secuenciador de comandos de a bordo.
Stored Command Absolute	Permite el procesamiento concurrente de hasta 5 secuencias de comandos.

Tabla B.1: Aplicaciones de cFS y sus funciones.

C Porción de código ERS_APP

El siguiente código muestra el archivo de cabecera *ers_recovery.h*, en donde se muestran las configuraciones que puede personalizar el usuario según los objetivos de su misión. En el código se observa una configuración de ejemplo para *Command Ingest* y *Sample_app*.

Fragmento de ers_recovery.h

```
1
2  /*App max bytes*/
3  #define MAX_SO_LENGTH 27352
4  /*Start type: 0 for CI start, 1 for processor restart*/
5  #define START_RECOV1 0
6  #define START_RECOV2 0
7
8  /* CFE APP NAME, declared at cpu1_cfe_es_startupt.scr*/
9  #define APP_RECOV1 "SAMPLE_APP"
10 #define APP_RECOV2 "CI_LAB_APP"
11 /* Path/Filename (.so for linux), declared at
       cpu1_cfe_es_startupt.scr*/
12 #define FILENAME_RECVO1 "/cf/sample_app.so"
13 #define FILENAME_RECVO2 "/cf/ci_lab.so"
14 /* Entry Point (main function for Apps), declared at
       cpu1_cfe_es_startupt.scr*/
15 #define MAIN_RECOV1 "SAMPLE_APP_Main"
16 #define MAIN_RECOV2 "CI_Lab_AppMain"
17 /* Stack size for the App, declared at
       cpu1_cfe_es_startupt.scr*/
18 #define STACK_RECOV1 16384
19 #define STACK_RECOV2 16384
20 /* Priority for the App, declared at cpu1_cfe_es_startupt
       .scr*/
21 #define PRIORITY_RECOV1 50
22 #define PRIORITY_RECOV2 60
```

D Herramientas

Como parte de las herramientas proporcionadas al descargar el repositorio ed ERS app, se incluyen algoritmos de python para facilitar al usuario la programación de las memorias de respaldo con los archivos ejecutables de su elección.

ERS_Converter.py

```
################################################
#########################  File names ##########
# Object/Executable cFS file (.so for linux)
FileInput = "ci_lab.so"
# Output file (.txt)
FileOutput = "FileOut_CI.txt"
################################################
###################### Function definition ######
def FileGenerator(FInput, FOutput):
    try:
        #Open input file read mode:binary
        with open(FInput, "rb") as f:
            info = f.read()
        #Open output file write mode
        with open(FOutput, "w") as out:
        #Save byte as : DATA[n]=info;
            for i, byte in enumerate(info):
                line = f"DATA[{i}]={byte};\n"
                out.write(line)
        print(f"Process completed. File saved at {FOutput}")
        #Error handler
    except FileNotFoundError:
        print(f"Error:Input file doesnt exist {FInput}")
    except Exception as e:
        print(f"Failure: {e}")
###################### Main execution ##########
FileGenerator(FileInput, FileOutput)
```

ERS_FileDivider.py

```
##################################################
#########################  File names ############
# Input text file from ERS_Converter.py
FileInput = "FileOut_CI.txt"
# Number of data items per output file
FileSize = 1801
# Total number of output files to generate
NoFiles = 15
##################################################
###################### Function definition ######
def FileSeparator(FInput, FSize, NFile):
    try:
        # Open input file in read mode
        with open(FInput, 'r') as f:
            lines = [line.strip() for line in f if line.strip()]

        total_elements = FSize * NFile
        if len(lines) < total_elements:
            print(f"Warning: Input file only contains {len(lines)
                } data entries.")

        for i in range(NFile):
            begin = i * FSize
            end = begin + FSize
            block = lines[begin:end]

            if not block:
                break  # No more data available

            # Renumber from 0
            block_renumbered = []
            for j, line in enumerate(block):
                value = line.split('=')[1].replace(';', '')
                block_renumbered.append(f"DATA[{j}]={value};")

            with open(f"block_{i+1}.txt", 'w') as out_file:
                out_file.write('\n'.join(block_renumbered))

        print("Process completed. Files successfully generated.")
    # Error handlers
    except FileNotFoundError:
        print(f"Error: Input file does not exist - {FInput}")
    except Exception as e:
        print(f"Failure: {e}")

################## Main execution ###############
FileSeparator(FileInput, FileSize, NoFiles)
##################################################
```

Lista de acrónimos

API	Interfaz de programación de aplicaciones
CCSDS	Consultative Committee for Space Data Systems
CDS	Almacenamiento de datos críticos
CI	Command Ingest
COTS	Producto comercial disponible en el mercado
CR	Rayos cósmicos
CRC	Código de redundancia cíclica
cFE	Core Flight Executive
cFS	Core Flight System
eMMC	Tarjeta multimedia integrada
ERS	External Recovery Supervisor app
ES	Executive Service
EVS	Event Service
FAI	Federación Aeronáutica Internacional
FPGA	Arreglo de compuertas programables en campo
FSW	Software de vuelo
GPIO	Entrada/salida de propósito general
GPM	Medidor Global de la Precipitación
GPU	Unidad de Procesamiento Gráfico
HK	Housekeeping Application
ISS	Estación Espacial Internacional
LEO	Órbita terrestre baja
LRO	Lunar Reconnaissance Orbiter
MMU	Unidad de Gestión de Memoria
NASA	National Aeronautics and Space Administration
NICER	Neutron Star Interior Composition Explorer
OBC	Computadora de a bordo
OS	Sistema operativo
OSAL	Operating system abstraction layer
POSIX	Interfaz de sistema operativo portátil para unix
RHA	Garantía de resistencia a la radiación
RTOS	Sistema operativo en tiempo real
SB	Software Bus Service
SCH	Scheduler Application
SDO	Observatorio Dinámico Solar
SEE	Efecto de evento único
SEL	Latchup de evento único
SET	Transitorio de evento único
SEU	Upset de evento único
SoC	Sistema en chip
SPF	Punto único de falla
TBL	Table Service
TDM	Multiplexación por división de tiempo
TID	Dosis total ionizante
TIME	Time Service
TMR	Redundancia modular triple
TO	Telemetry Output
WD	Watchdog timer

Índice de Figuras

Índice de Tablas