



UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

DISEÑO E IMPLEMENTACIÓN DEL SOFTWARE DE VUELO PARA UN
NANO-SATÉLITE TIPO CUBESAT

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO

CARLOS EDUARDO GONZÁLEZ CORTÉS

PROFESOR GUÍA:
MARCOS DÍAZ QUEZADA

MIEMBROS DE LA COMISIÓN:
CLAUDIO ESTÉVEZ MONTERO
ALEX BECERRA SAAVEDRA

SANTIAGO DE CHILE
SEPTIEMBRE 2013

A mis padres Carlos González Espeleta y Paula González Cortés por que les debo todo lo que soy, porque gracias a su gran esfuerzo y apoyo incondicional he podido llegar a estas instancias en la vida, los amo. A mis hermanas, siempre estaré con ustedes.

*A Marcia, mi amor, por su invaluable apoyo, comprensión y dedicación. Junto a ti han transcurrido estos maravillosos años y has sido parte fundamental para lograr mis metas.
Te amo.*

Agradecimientos

Nosotros no hubiesemos sido nada sin ustedes, sino por toda la gente que estuvo a nuestro alrededor desde el comienzo.

Algunos siguen hasta hoy.

Gracias... Totales.

Índice general

1. Introducción	1
1.1. Objetivos generales	2
1.2. Objetivos específicos	2
1.3. Estructura	3
2. Marco teórico	4
2.1. Sistemas embebidos	4
2.1.1. Microcontroladores PIC	5
2.2. Sistemas operativos	8
2.2.1. Sistemas operativos de tiempo real	9
2.3. Ingeniería de Software	12
2.3.1. Calidad de software	12
2.3.2. Patrones de diseño	15
2.3.3. Arquitecturas de <i>software</i> basadas en patrones	18
2.4. Pequeños Satélites	21
2.4.1. Satélites tipo Cubesat	21
2.5. Proyecto SUCHAI	23
3. Diseño del <i>software</i>	25
3.1. Requerimientos	25
3.1.1. Requerimientos operacionales	25
3.1.2. Requerimientos no operacionales	28
3.1.3. Requerimientos mínimos	29
3.2. Plataforma	31
3.2.1. Computador a bordo	31
3.3. Arquitectura de <i>software</i>	33
3.3.1. Arquitectura Global	33
3.3.2. Controladores de hardware	34
3.3.3. Sistema operativo	37
3.3.4. Aplicación	38
3.4. Diseño de la solución	42
4. Implementación	48
4.1. Ambiente de desarrollo	48
4.2. Organización del proyecto	51
4.2.1. Directorios	51

4.2.2. IDE	52
4.2.3. Documentación	52
4.3. Controladores de hardware	54
4.3.1. Microcontrolador	55
4.3.2. Periféricos	55
4.4. Sistema operativo	56
4.5. Aplicación	59
4.5.1. Implementación del patrón de diseño	60
4.5.2. Comandos	61
4.5.3. Repositorio de comandos	65
4.5.4. Repositorios de estados	66
4.5.5. Dispatcher	70
4.5.6. Executer	72
4.5.7. Listeners	74
4.6. Específico al proyecto SUCHAI	80
4.6.1. Consola serial	80
4.6.2. Plan de vuelo	81
4.6.3. Comunicaciones	82
4.6.4. Inicialización del sistema	83
5. Pruebas y resultados	86
5.1. Pruebas de rendimiento	86
5.1.1. Estadísticas del programa	86
5.1.2. Estadísticas de uso de memoria	86
5.1.3. Estadísticas de uso del procesador	87
5.2. Pruebas de integración	89
5.2.1. Configuración	89
5.2.2. Resultados	91
6. Conclusiones	109
6.1. Conclusiones generales	109
6.2. Limitaciones	111
6.3. Trabajo futuro	112

Índice de tablas

2.1. Guía de microcontroladores PIC	5
3.1. Requerimientos no funcionales	30
3.2. Requerimientos mínimos	30
3.3. Comparación de sistemas operativos para sistemas embebidos	38
3.4. Análisis de la arquitectura, según requerimientos operacionales, para el área de comunicaciones	43
3.5. Análisis de la arquitectura, según requerimientos operacionales, para el área de control central	44
3.6. Análisis de la arquitectura, según requerimientos operacionales, para el área de tolerancia a fallos	44
3.7. Análisis de la arquitectura, según requerimientos operacionales, para el área de energía órbita y payloads	45
4.1. Requerimientos de <i>hardware</i> recomendados para desarrollo[?].	49
4.2. Organización de directorios del proyecto	52
4.3. Configuración del compilador XC16	53
4.4. Drivers para periféricos del microcontrolador	55
4.5. Configuración del compilador XC16 para FreeRTOS	56
4.6. Estándar para identificar comandos	66
5.1. Estadísticas del código	86
5.2. Uso de memoria de la aplicación	87
5.3. Uso de memoria del sistema operativo	87
5.4. <i>Beacons</i> generados por el <i>software</i> de vuelo	94
5.5. Plan de vuelo	98
5.6. Resultados para test SOC	107

Índice de figuras

2.1. Arquitectura de la CPU del PIC24F	6
2.2. Arquitectura del PIC24F	7
2.3. Sistema operativo como capa de abstracción de <i>hardware</i>	8
2.4. <i>Real time scheduling</i>	10
2.5. Tareas de FreeRTOS, diagrama de estados	11
2.6. <i>Scheduling</i> de tareas	12
2.7. ISO/IEC 25010, categorías y subcategorías.	15
2.8. Diagrama de clases del patrón de diseño MVC.	17
2.9. Esquema de colaboración del patrón de diseño MVC.	18
2.10. Diagrama de clases del patrón de diseño procesador de comandos.	20
2.11. Esquema de colaboración del patrón de diseño procesador de comandos.	20
2.12. Satélite tipo Cubsat	21
2.13. Especificaciones de diseño y dimensiones del estándar Cubesat de una unidad [?].	22
2.14. Logo del proyecto SUCHAI	23
2.15. Satélite SUCHAI junto a una <i>langmuir probe</i>	24
3.1. CubesatKit de Pumpkin Inc.	31
3.2. Dos módulos que componen el computador a bordo del satélite	32
3.3. Arquitectura de tres capas para un sistema embebido.	34
3.4. Arquitectura para controladores síncronos	35
3.5. Arquitectura para controladores asíncronos	36
3.6. Arquitectura de un controlador de entrada serial asíncrono	37
3.7. Arquitectura de <i>software</i> para el control del satélite	43
3.8. Arquitectura de <i>software</i> para el control del satélite	47
4.1. Entorno de desarrollo integrado MPLABX	50
4.2. Tarjeta de desarrollo para Cubesat Kit de Pumpkins	51
4.3. Diálogo de configuración del compilador XC16 en MPLABX	53
4.4. Documento HTML generado por Doxygen	54
4.5. Problema del productor-consumidor	70
4.6. FreeRTOS Queue	71
4.7. Diagrama de flujo para <i>dispatcher</i>	72
4.8. Diagrama de flujo para <i>executer</i>	74
4.9. Diagrama de flujo para <i>listeners</i>	76
4.10. Diagrama de flujo para <i>housekeeping</i>	78

4.11. Diagrama de flujo para <i>taskConsole</i>	81
4.12. Diagrama de flujo para <i>taskFlightPlan</i>	82
4.13. Diagrama de flujo para <i>taskCommunications</i>	84
4.14. Diagrama de flujo para <i>taskDeployment</i>	85
5.1. Registro de uso del procesador	88
5.2. Porcentaje de uso del procesador	88
5.3. Montaje de la prueba de integración	90
5.4. Despliegue de antenas	92
5.5. Beacon recibido en la estación terrena	94
5.6. Recepción de telemetría en la estación terrena	96
5.7. Gráfico de funcionamiento del plan de vuelo	99
5.8. Gráfico con el valor de las variables de estado en el tiempo	101
5.9. Variables de estado ante un reinicio	102
5.10. Gráfico con variables de EPS y estimación de SOC	105

Capítulo 1

Introducción

Durante mucho tiempo el desarrollo de proyectos aeroespaciales ha estado limitado a gobiernos de países con gran poderío económico o grandes corporaciones con objetivos tanto militares como civiles para la observación de la tierra, comunicaciones a distancia o posicionamiento global. Con el reciente desarrollo de estándares para pequeños satélites, como los Cubesat, se abre la posibilidad a gobiernos e instituciones educacionales en todo el mundo, de desarrollar proyectos aeroespaciales de menor riesgo y bajos costos.

El presente trabajo se enmarca en el desarrollo del proyecto SUCHAI, el cual consistente en la implementación, lanzamiento y operación de un nano-satélite tipo Cubesat de 1U, siendo la primera aproximación en la materia para la universidad y el país. Los fines de este proyecto son básicamente científicos y educacionales buscando abrir las posibilidades de desarrollar futura tecnología aeroespacial así como formar profesionales con la experiencia necesaria en este ámbito.

El satélite del proyecto SUCHAI consta de tres subsistemas principales: computador a bordo, energía y comunicaciones. Además se debe considerar la posibilidad de integrar diferentes *payloads* que desarrollan misiones específicas una vez que el satélite se encuentra operando en el espacio.

El computador a bordo del satélite debe ejecutar un programa que permita desarrollar todas las acciones que requiere la misión. Esta aplicación se denomina *software* de vuelo y su objetivo es controlar el satélite, ejecutar operaciones específicas durante el vuelo y permitir la comunicación con la estación terrena para la descarga de datos y subida de telecomandos, entre otras tareas fundamentales. Su desarrollo se realiza siguiendo una metodología ordenada que consta de tres etapas: diseño, implementación y pruebas.

La etapa de diseño consiste en explicitar los requerimientos del sistema y generar una arquitectura de *software* que representa de manera conceptual la solución a implementar, en esta etapa se hace un uso extensivo del concepto de patrones de diseño para conseguir la solución al problema planteado. Durante la implementación se toma la solución conceptual y se programa el *software* utilizando las herramientas adecuadas a la plataforma consiguiendo una plataforma base que consta de controladores de *hardware*, sistema operativo y la apli-

cación específica a la misión. Las pruebas se desarrollan sobre el sistema integrado en sus tres subsistemas básicos mediante la puesta en funcionamiento del satélite por un tiempo prolongado usando la técnica denominada *hardware on the loop simulation* adecuada para probar sistemas embebidos complejos.

El resultado de este trabajo es la base para obtener un vehículo espacial con las funciones fundamentales para su operación y así dejar establecido el flujo a seguir para agregar más subsistemas.

1.1. Objetivos generales

Uno de los componentes fundamentales de un satélite es su computador abordo, sistema encargado de dar inteligencia y operatividad durante todo su tiempo de vida útil en el espacio. En el caso de un nano-satélite se tiene el desafío de dotar con las funcionalidades estándar de un satélite de gran envergadura a un sistema con recursos de energía, cómputo y espacio extremadamente limitados como por ejemplo, microcontroladores PIC18, PIC24 o PIC32.

El objetivo general de este trabajo es el diseño, desarrollo e implementación del *software* que controla el computador a bordo del satélite. Se requiere diseñar una arquitectura de *software* que abarque desde controladores de *hardware* hasta la aplicación final para la operación durante la misión. Esta arquitectura debe cumplir, además, con requerimientos de calidad de *software* como modularidad, modificabilidad y facilidad de mantenimiento en general, adaptándose en específico a sistemas embebidos.

La implementación realizará específicamente para el satélite SUCHAI, por lo que se busca proveer la base del sistema que lo controla, así como integrar los sistemas de energía y comunicaciones en un *software* que se graba y ejecuta en el computador a bordo. El resultado de este trabajo permite la posterior integración física del vehículo satelital así como la adición de los *payloads* específicos de la misión, lo cual es abordado en detalle en el trabajo de Tomás Opazo titulado Requerimientos, Implementación y Verificación del Cubesat SUCHAI[?].

1.2. Objetivos específicos

Los objetivos específicos del proyecto definen las tareas que permiten cumplir el objetivo general del proyecto y se enumeran a continuación:

- Diseñar una arquitectura de *software* de vuelo del satélite.
- Implementar controladores de *hardware* para el microcontrolador y sus periféricos.
- Implementar controladores de los subsistemas principales: *transceiver* y EPS.
- Integrar un sistema operativo de tiempo real multitarea en el sistema embebido.
- Implementar el flujo principal de la arquitectura del *software* de vuelo del satélite.
- Integrar sistema de comunicaciones y energía al *software* de vuelo.

- Pruebas del sistema integrado verificando requerimientos funcionales.

El trabajo se puede considerar terminado cuando se ha probado la implementación e integración del *software* con los módulos de comunicaciones y energía obteniendo un sistema satelital con las funcionalidades básicas.

1.3. Estructura

El presente trabajo se estructura según un proceso de diseño, implementación y pruebas propio de un proyecto de desarrollo de software, en sus diferentes capítulos se presentará la siguiente información:

- **Capítulo 1. Introducción:** Descripción del trabajo realizado, objetivos generales, específicos y alcances del proyecto.
- **Capítulo 2. Marco teórico:** Revisión bibliográfica. Se explican los principales conceptos necesarios para contextualizar el trabajo desarrollado.
- **Capítulo 3. Diseño del software:** Se definen los requerimientos operacionales y no operacionales del proyecto que guiarán el diseño. Se plantea una arquitectura de *software* a diferentes niveles de abstracción basada en patrones de diseño para luego generar el diseño final de la solución a implementar.
- **Capítulo 4. Implementación:** Se implementa la solución propuesta guiando paso a paso la generación de la plataforma base de la aplicación sobre la plataforma objetivo. Las particularidades del proyecto SUCHAI se detallan aparte como flujos que explican su implementación.
- **Capítulo 5. Pruebas:** Se realizan pruebas de integración sobre el sistema implementado para verificar cada uno de los requerimientos planteados en el capítulo 3.
- **Capítulo 6. Conclusiones:** Se concluye sobre los resultados obtenidos y el proceso de desarrollo del proyecto en sí. Se analizan las limitaciones de la solución y se plantean posibles mejoras como trabajos futuros.

Capítulo 2

Marco teórico

2.1. Sistemas embebidos

Los sistemas embebidos, a diferencia de un computador personal que es usado con fines generales para una amplia variedad de tareas, son sistemas computacionales normalmente utilizados para atender una cantidad limitada de procesos; realizar tareas específicas o dotar de determinada inteligencia a un sistema más complejo. Está compuesto por uno o más microcontroladores pequeños , los que cuentan con periféricos para manejar diferentes protocolos de comunicación, conversores análogo-digital, *timers*, puertos de entrada y salida digitales, todos integrados en un mismo chip para ahorrar espacio y energía. Parte fundamental de un sistema embebido es el *software* que provee la funcionalidad final. Usualmente se usa el término *firmware* para referirse a este código con que se programa el microcontrolador, el que por lo general es específico para la plataforma de *hardware* y se relacionan a muy bajo nivel. A diferencia de un computador de propósito general, donde el usuario puede cargar una serie de programas para un amplio rango de usos, acá no se tiene la capacidad de re-programarlo fuera de las posibilidades que el desarrollador ha brindado al sistema [?].

Para el diseño de sistemas embebidos se deben considerar ciertos aspectos que los diferencian de otros tipos de sistemas de computacionales, tales como [?]:

- Se mantiene siempre funcionando y debe proveer respuesta a entradas y órdenes externas en tiempo real. Se debe diseñar considerando una operación continua y una posible reconfiguración del sistema estando ya en marcha.
- Las interacciones con el sistema pueden ser impredecibles y no se tiene control sobre ellas. Existen sistemas que son controlados por el usuario mediante una interfaz preparada para ello, mientras que otros deben atender eventos imprevistos sin dejar de realizar tareas rutinarias.
- Existen limitaciones físicas. Normalmente estos sistemas poseen limitadas características de poder de cómputo, memoria de datos y de programa, espacio físico y disponibilidad de energía.

- El diseño de *software* para sistemas embebidos requiere una interacción de bajo nivel. Existe una amplia gama de plataformas de *hardware* para desarrollar sistemas embebidos y se requiere interactuar también con una variedad de dispositivos externos. Por esto, se requiere desarrollar capas de controladores de periféricos que oculten las diferencias de *hardware* a la aplicación final del sistema.
- Es importante considerar aspectos de seguridad y confiabilidad del sistema durante todo su desarrollo debido a que la mayoría de los sistemas embebidos son usados para controlar otros sistemas críticos en diversos procesos.

2.1.1. Microcontroladores PIC

Todo sistema embebido está formado fundamentalmente por un microcontrolador, que brinda la capacidad de cómputo y el control de diferentes periféricos que normalmente están integrados en el mismo chip. Entre los principales fabricantes de microcontroladores se encuentran: Microchip, Texas Instrument, ARM, Motorola, NVidia. Este trabajo se concentra en los microcontroladores PIC desarrollados por la compañía Microchip. La familia de microcontroladores PIC es bastante amplia, adaptándose a un gran rango de necesidades. La tabla 2.1 resume las principales características de los diferentes modelos y puede ser utilizada como una guía para determinar el dispositivo adecuado según la aplicación:

Tabla 2.1: Guía de microcontroladores PIC

Familia	Instrucciones	Datos	Memoria Programa	Memoria RAM	Velocidad	Periféricos	Usos
PIC10	12 bit	8 bit	512 Words	64 Bytes	16MHz	IO, ADC	Espacio reducido, bajo costo. Lógica digital, control IO
PIC12	12 bit	8 bit	4 Kwords	256 Bytes	32MHz	IO, ADC, TIMER, USART	Bajo costo. Logica digital, control IO, sensores
PIC16	14 bit	8 bit	16 Kwords	2 Kbytes	48MHz	IO, ADC, TIMER, USART, PWM	Control, sensores, recolección de datos, display, interfaz serial.
PIC18	16 bit	8 bit	64 Kwords	4 Kbytes	64MHz	IO, ADC, TIMER, USART, PWM, USB, CAN, ETHERNET, LCD	Control, sensores, datos, interfaz serial, ethernet, display.
PIC24	24 bit	16 bit	512 Kbytes	96 Kbytes	70 MIPS	IO, ADC, TIMER, USART, PWM, USB, CAN, ETHERNET, RTCC, DMA, CRC, PPS	Uso general
dsPIC	24 bit	16 bit	512 Kbytes	54 Kbytes	70 MIPS	IO, ADC, DAC, TIMER, USART, PWM, USB, CAN, ETHERNET, RTCC, DMA, CRC, PPS, CODEC, QEI	Uso general. Procesamiento señales. Control de motores.
PIC32	32 bit	32 bit	512 Kbytes	128 Kbytes	80 MHz	IO, ADC, TIMER, USART, PWM, USB, CAN, ETHERNET, RTCC, DMA, CRC, PPS, CODEC, CTMU	Uso general

Los valores representan el tope de linea para cada familia

A continuación, se describen las características específicas de los microcontroladores PIC24 que corresponde al dispositivo utilizado en este trabajo.

Arquitectura.

Poseen un juego de instrucciones *Reduced Instruction Set Computing* (RISC) (80 instrucciones) de ancho fijo en 24 bits que en su mayoría se ejecutan en un solo ciclo, excepto:

divisiones, cambios de contexto y acceso por tabla a memoria de programa [?]. Se basa en una arquitectura Harvard modificada de 16 bits de datos [?], lo que significa que el dispositivo posee una memoria de datos tipo *Random Access Memory* (RAM), separada de la memoria de datos (FLASH) pudiendo acceder de manera independiente e incluso simultánea a las instrucciones del programa y a los datos de este alojados en RAM. La arquitectura de la CPU completa una *Arithmetic Logic Unit* (ALU), con *hardware* dedicado para realizar multiplicaciones y divisiones. El detalle de la arquitectura del microcontrolador PIC24 se muestra en la **figura 2.1**. También posee un vector de hasta 128 interrupciones, con capacidad para atender hasta 8 de ellas de manera simultánea, lo que permite liberar al procesador de la espera de sucesos asíncronos, ya que son notificados y atendidos de manera específica en una rutina de atención de la interrupción.

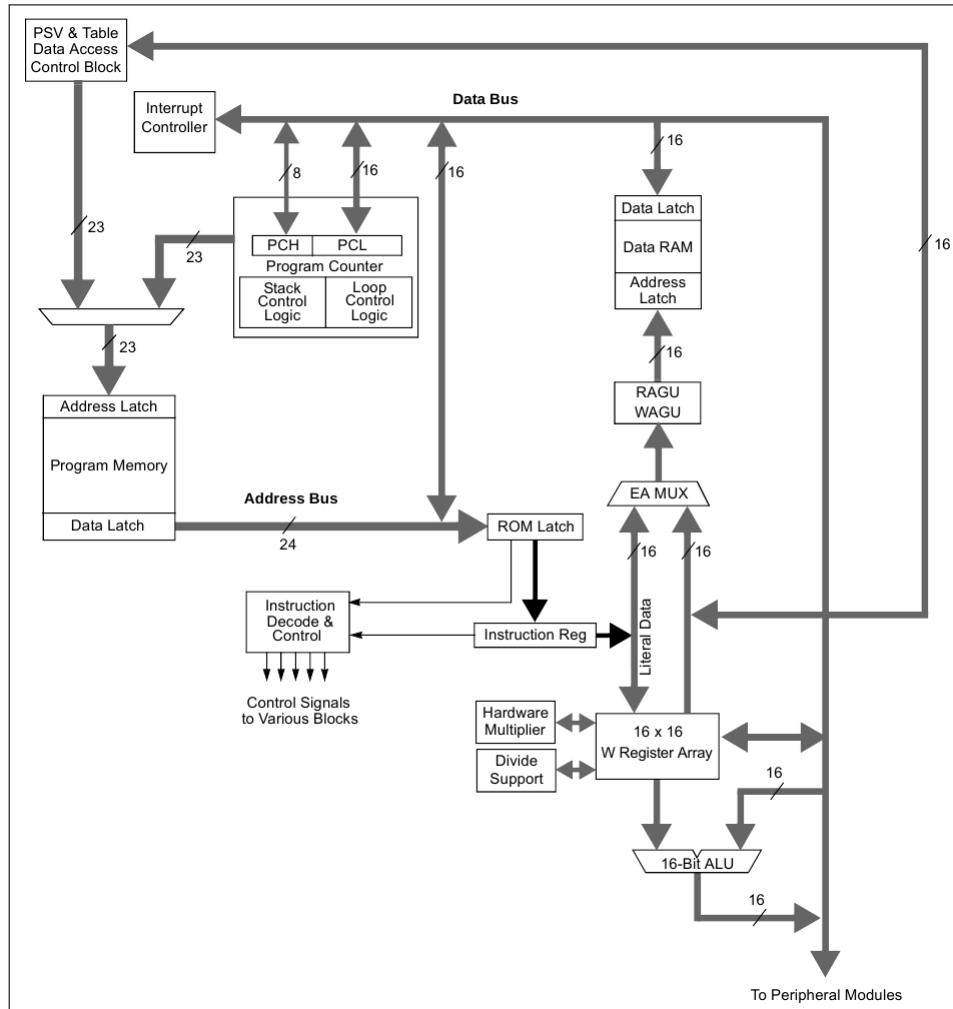


Figura 2.1: Arquitectura de la CPU del PIC24F

Periféricos.

La familia de microcontroladores PIC24F, integra en el mismo chip una serie de periféricos que permiten realizar funciones específicas a través de *hardware*. Esto hace que el PIC24F se convierta en un sistema embebido capaz de ser utilizado en aplicaciones que requieran:

conversores análogo-digital, temporizadores, comunicación síncrona y asíncrona como RS232, SPI o I2C, USB o Ethernet, manteniendo acotados los costos del sistema. Una lista de los periféricos disponibles para estos microcontroladores se detalla en la **figura 2.2**. Los periféricos son controlados a través de registros mapeados en la memoria del microcontrolador, por lo que su acceso y configuración se realiza mediante instrucciones típicas de lecturas y escritura en memoria. Los compiladores para lenguaje C de Microchip proveen librerías de más alto nivel así como una completa documentación sobre el funcionamiento de cada periférico.

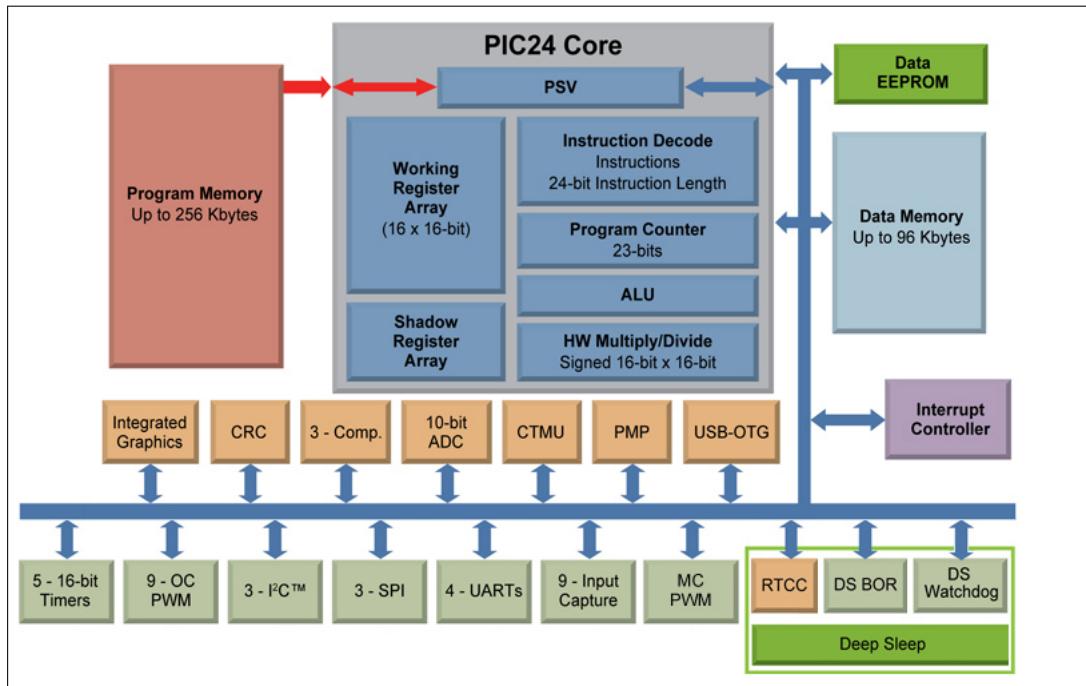


Figura 2.2: Arquitectura del PIC24F

Desarrollo.

Para el desarrollo de aplicaciones sobre este tipo de microcontroladores se requiere, al menos, de las siguientes herramientas:

1. Compilador
2. Entorno de desarrollo
3. Programador

Por lo general, es el fabricante del dispositivo el que provee la mayoría de ellas. En este caso el compilador y entorno de desarrollo se pueden obtener a través de su página web: <http://www.microchip.com/developmenttools/>. Se dispone de diferentes compiladores para dispositivos de 8, 16 y 32 bits, correspondiendo para la familia PIC24 el compilador *xc16* para lenguaje C que es gratuito en su versión *lite*. El entorno de desarrollo integrado que se denomina MPLABX, un programa multiplataforma basado en NetBeans, que integra las funcionalidades del compilador y programador sumado a un editor de texto avanzado para proveer un entorno de desarrollo completo.

Para grabar el *software* desarrollado para estos dispositivos se debe utilizar una herramienta externa denominada programador. Existe un serie de programadores disponibles los cuales deben ser adquiridos por separado. Entre los más populares se encuentran:

- PICKIT3: Programador y depurador de bajo costo adecuado para entornos de desarrollo.
- ICD3: Programador y depurador adecuado para entornos de producción.
- REAL ICE: Emulador y programador de gama alta adecuado para entornos de producción.

2.2. Sistemas operativos

Un sistema operativo es la aplicación base de un sistema computacional, pues brinda servicios básicos al resto de las aplicaciones de uso general ejecutadas en el computador. El sistema operativo es la capa entre el *hardware* y las aplicaciones, el cual puede variar considerablemente entre un sistema y otro. Por lo tanto se necesita una capa de abstracción que haga a la aplicación independiente de la plataforma en que se ejecuta. Para esto, el sistema operativo provee servicios que usan interfaces de bajo nivel con el *hardware*, las cuales no están disponibles para la aplicación como se ilustra en la **figura 2.3**. Ejemplos de sistemas operativos los son UNIX, GNU/Linux, FreeRTOS, entre otros.

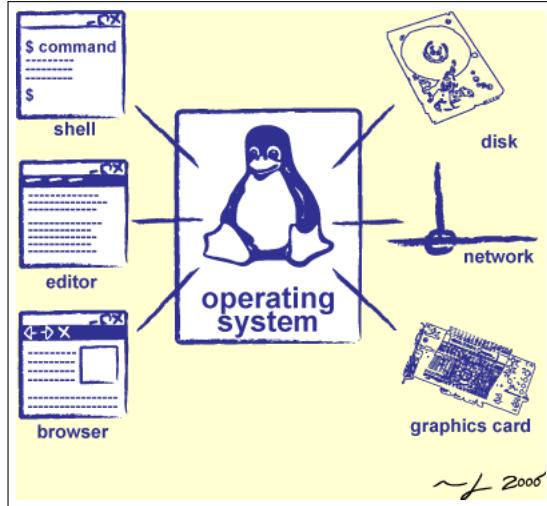


Figura 2.3: Sistema operativo como capa de abstracción de *hardware*

Los servicios básicos que provee a través de su kernel son:

- **Gestión de tareas:** También denominada gestión de procesos o *scheduler*. El sistema operativo ve a la aplicación como una serie de tareas o procesos, a las cuales se deben entregar los servicios de creación, ejecución y asignación de prioridades. Cada tarea cumple objetivos específicos en la aplicación, posee sus propios recursos y limitaciones de tiempo. Pueden existir varios procesos funcionando a la vez, y el sistema operativo se encarga de proveer tiempo de procesamiento a cada una de ellos. Existen diferen-

tes alternativas para gestionar la ejecución de las tareas entre las que sobresalen dos: *preemptive* y cooperativo. La implementación de la aplicación depende mucho del tipo de *scheduler* disponible por lo cual se describe cada uno a continuación:

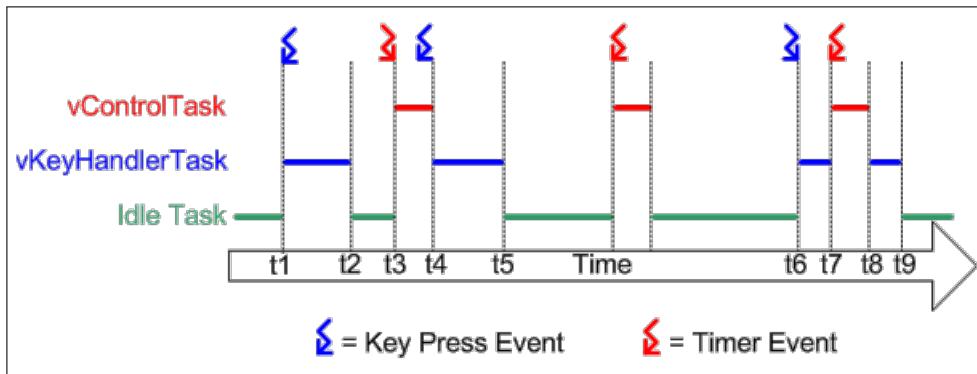
- **Modo *preemptive*:** El sistema operativo puede interrumpir la actual tarea en cualquier momento de su ejecución, para realizar el cambio de contexto y ceder el procesador a una diferente. Por lo general se realiza a intervalos fijos, denominados *ticks*. Esto puede ir acompañado de un sistema de prioridades, así el *scheduler* se asegura que siempre se esté ejecutando la tarea de mayor prioridad disponible.
- **Modo cooperativo:** En el modo cooperativo, el sistema operativo nunca inicia un cambio de contexto, sino que cada tarea en ejecución cede voluntariamente el procesador a una nueva. El sistema operativo es quien decide la siguiente tarea a ejecutar ya, sea por el algoritmo de *round robin*, un sistema de prioridades o ambos. Se denomina cooperativo porque todas las tareas deben estar correctamente programadas, y cooperar con la ejecución del resto iniciado el proceso de cambio de contexto.
- **Comunicación y sincronización entre tareas:** Si bien cada tarea posee su propio contexto de ejecución y pueden existir varias funcionando de manera concurrente, la verdadera utilidad nace de la posibilidad de comunicación entre ellas, para compartir estados y mensajes, que permitan cambiar el flujo de ejecución de las operaciones en el sistema embebido. Así, las tareas compartirán los mismos recursos de *hardware* o requerirán de memoria compartida, en esquemas tipo productor-consumidor, donde el sistema operativo es quien provee las estructuras de sincronización adecuadas. De este modo los mecanismos de comunicación aseguran que la información compartida no se corrompa, y no existan interferencias entre tareas al acceder simultáneamente a recursos compartidos.
- **Temporización:** Dado los requerimientos estrictos de tiempo propios de un sistema de tiempo real, el sistema operativo debe proveer servicios de tiempo como *delays* y *time-outs*, para controlar la periodicidad o los límites de tiempo de ejecución de cada tarea.
- **Gestión de memoria:** Las diferentes tareas y procesos que se ejecutan en el sistema requieren reservar, usar y liberar memoria de datos para su ejecución de manera segura, es decir, sin corromper la memoria utilizada por el resto de los procesos o el propio sistema operativo. Por esto, debe proveer servicios de gestión de memoria, como por ejemplo, la reserva dinámica de memoria de datos.
- **Gestión de dispositivos de entrada y salida:** Los dispositivos de entrada y salida son compartidos por todas las tareas que se ejecutan, por lo tanto, el sistema operativo provee el servicio de gestionar el acceso a estos dispositivos de manera uniforme y organizada.

2.2.1. Sistemas operativos de tiempo real

Un sistema operativo de tiempo real, posee un *scheduler* diseñado para proveer un flujo de ejecución determinista, pues solo sabiendo con exactitud la tarea que el sistema ejecutará, en un determinado momento, se pueden cumplir los requerimientos estrictos de *timing* [?].

Esto es un aspecto de especial interés en sistemas embebidos que, normalmente, requieren respuesta en tiempo real ante eventos no predecibles como las interrupciones.

La figura 2.4 muestra la forma de conseguir un sistema de tiempo real, mediante el uso de prioridades para las diferentes tareas. En este ejemplo, la mayor parte del tiempo el sistema está en estado *idle*, sin código que ejecutar. Sin embargo, ante la presencia de ciertos eventos, el sistema debe responder de manera instantánea cambiando de contexto a la tarea correspondiente. Ciertas tareas pueden requerir un estricto *timing* ejecutándose de manera periódica. En tal caso se le asigna una alta prioridad para asegurar que el sistema operativo siempre ejecute esta tarea cuando corresponda.



FreeRTOS

FreeRTOS es un tipo de *Real Time Operating System* (RTOS), que está diseñado para ser lo suficientemente pequeño, en términos de consumo de memoria como, para ser utilizado en un microcontrolador [?]. Como estos sistemas son realmente limitados, normalmente no existe la posibilidad de ejecutar un sistema operativo completo como GNU-Linux, sin embargo FreeRTOS provee un kernel capaz de manejar múltiples tareas con prioridades, comunicación entre tareas, *timing* y primitivas de sincronización. Por su reducido tamaño no entrega funcionalidades de alto nivel como una consola de comandos, así como tampoco funcionalidades de bajo nivel, como controladores para el *hardware* o periféricos.

Entre sus principales características se encuentran [?]:

- *Scheduler pre-emptive* o cooperativo.
- Sincronización y comunicación entre tareas a través de colas, semáforos, semáforos binarios y mutexes.
- Mutexes con herencia de prioridades.
- *Software timers*.
- Bajo consumo de memoria (Entre 6K y 10K en ROM).
- Altamente configurable.
- Detección de *stack overflow*

- Soporte oficial a 33 arquitecturas de sistemas embebidos.
- Estructura de código portable, escrito en C.
- Licenciado bajo *General Public License* (GPL) modificada que permite su uso comercial sin publicar código fuente.
- Gratuito
- Amplia documentación, foros y asistencia técnica.

Funcionamiento Los conceptos fundamentales detrás del funcionamiento de FreeRTOS son las tareas y el *scheduler*. Una tarea es un hilo de procesamiento, normalmente una función que se ejecuta de manera continua. Se puede encontrar en dos estados fundamentales: “ejecutándose” y “no ejecutándose”. Cuando se está ejecutando, tiene el control del procesador y el código dentro de la función que la representa es procesado. El estado “no ejecutándose” en realidad consta de tres sub-estados ,como se observa en la **figura 2.5**: se inicia en un estado “listo”, lo que indica que la tarea está en condiciones de ser seleccionada por el *scheduler* y pasar a estar “ejecutándose”. Estar “bloqueado” significa que la tarea no está disponible para ser procesada pues está en espera de algún evento, por ejemplo, la liberación de un *mutex*. Por último el estado “suspensionado” donde tampoco se puede ejecutar y el proceso debe explícitamente reanudarse para quedar en condiciones de ser ejecutada. La creación de tareas y el control de sus estados se realiza a través de la API de FreeRTOS que documenta claramente todas las posibles operaciones que se pueden realizar ellas.

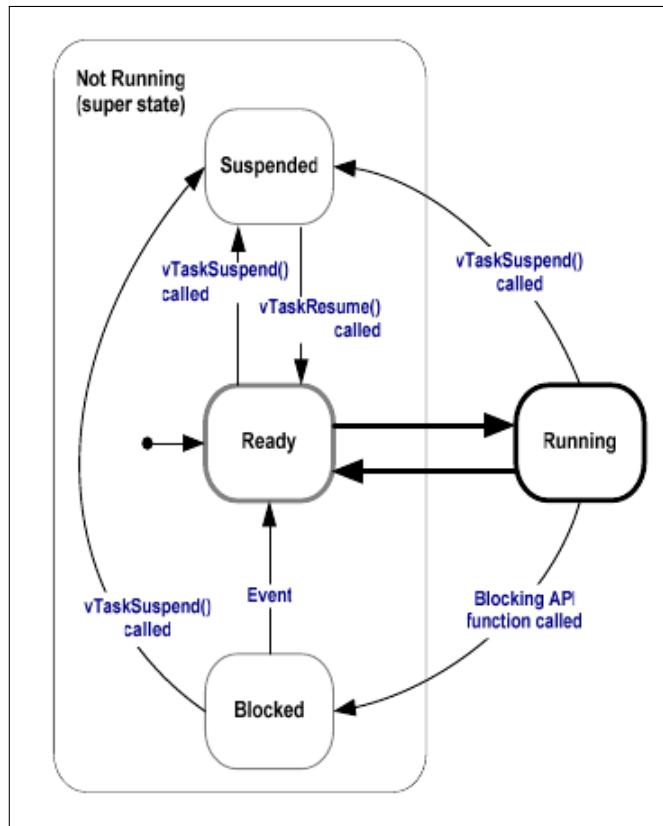


Figura 2.5: Tareas de FreeRTOS, diagrama de estados

El *scheduler* es la parte fundamental del kernel, que controla la ejecución de las diferentes

tareas disponibles. Su objetivo es generar la sensación de estar en un ambiente multi-proceso, cuando en realidad solo una función puede ejecutarse a la vez, ya que se cuenta solo con un procesador. Como se detalla en la **figura 2.6**, la función del *scheduler* es entregar una porción de tiempo de ejecución fijo a una tarea, y una vez que se agota se debe guardar su estado y se procede a ejecutar otra. Así cada una de las tareas se procesa durante un breve momento, de manera cíclica, hasta que completa su trabajo; si el tiempo de proceso asignado a cada una es lo suficientemente pequeño, parecería que muchas cosas ocurrieron simultáneamente. Una sola operación tomaría, en términos absolutos, un lapso menor en completarse si no fuera interrumpida, pero se gana un sistema más fluido cuando se deben ejecutar, en conjunto, tareas que toman mucho tiempo de proceso y otras relativamente cortas.

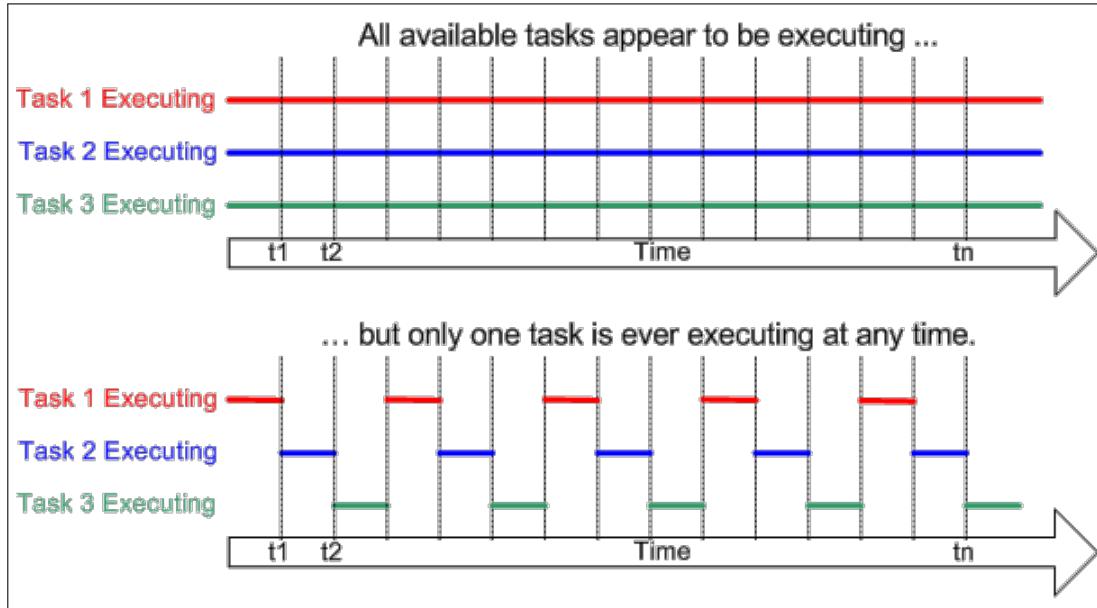


Figura 2.6: *Scheduling* de tareas

El algoritmo de *scheduling* se basa en un sistema de prioridades donde la tarea, en condiciones de ejecutarse, que tenga la mayor prioridad siempre debe ser procesada. Si varias tareas en estado “listo” comparten la misma prioridad, se aplica un algoritmo de *round-robin*. Las que están en estado “suspendido” y “bloqueado” nunca son seleccionadas por el *scheduler* y por lo tanto no consumen recursos. Haciendo un correcto uso de las prioridades y los diferentes estados se consigue un sistema que se ejecuta de manera fluida, haciendo un uso óptimo del procesador.

2.3. Ingeniería de Software

2.3.1. Calidad de software

Los requerimientos no funcionales de un proyecto de *software* se pueden definir como los parámetros de calidad buscados en el producto, entre los que se encuentran una serie de calificativos muy subjetivos, y tal vez difícilmente medibles como rapidez, seguridad,

escalabilidad y modularidad. Algunos conceptos pueden ser utilizados de manera poco clara o equivalentes como lo extensible o escalable que puede ser un *software*, sin dejar claro las diferencias o acotaciones entre ellos. Es por esto, que se han creado normas en torno a las metodologías para desarrollar y utilizar modelos de calidad de *software*, que permitan establecer de manera clara los parámetros a medir.

En especial se tratará la norma ISO/IEC 25010, una actualización a la antigua ISO/IEC 9126, que plantea un modelo de calidad *software* que consta de ocho características con sus respectivos sub-atributos, y puede ser utilizado para evaluación o especificación de una aplicación durante las etapas de: identificación de requerimientos; validación de la integralidad de la lista de requerimientos; definición de los objetivos del diseño del *software*; determinar de los objetivos de las pruebas; identificación de los criterios de calidad; o la definición de criterios para determinar si un producto está completo [?].

A continuación, se definen los parámetros de calidad fijados en la norma ISO/IEC 25010 para productos de *software*, así como un resumen a través de la **figura 2.7**.

- **Idoneidad Funcional:** Grado en que un producto provee la funciones requeridas.
 - **Compleitud funcional:** Grado en que las funciones cubren todas las tareas especificadas u objetivos.
 - **Correctitud funcional:** Grado en que el producto provee resultados correctos según el grado de precisión.
 - **Adecuación Funcional:** Grado en que las funciones facilitan el cumplimiento de las tareas requeridas.
- **Eficiencia del desempeño:** Desempeño relativo a la cantidad de recursos usados bajo ciertas condiciones.
 - **Tiempo:** El grado en que el sistema cumple con los requerimientos de tiempo de respuesta y tasa de rendimiento.
 - **Utilización de recursos:** Grado en que la cantidad y tipos de recursos usados por el sistema cumplen los requerimientos.
 - **Capacidad:** Grado en que los límites máximos de un sistema cumplen los requerimientos
- **Compatibilidad:** Grado en que el producto puede compartir información con otros productos o sistemas, y realizar sus funciones mientras se comparte el mismo entorno de *hardware* o *software*.
 - **Coexistencia:** Cómo un producto puede llevar a cabo sus funciones mientras comparte un entorno y recursos comunes con otros, sin afectarlos.
 - **Interoperación:** Cómo un producto puede compartir y usar información con otro.
- **Usabilidad:** Cómo el producto puede ser usado para sus fines determinados de manera efectiva, eficiente y satisfactoria.
 - **Reconocible como apropiado:** Grado en que los usuarios pueden reconocer que el producto es apropiado para sus necesidades.
 - **Aprendizaje.** Grado en que el producto se puede aprender a usar de manera efectiva, sin riesgos y satisfactoria.

- **Operatividad.** Grado en que el producto tiene atributos que lo hacen fácil de operar
 - **Protección de cometer errores:** Grado en que el sistema previene al usuario de cometer errores.
 - **Estética de la interfaz de usuario:** Grado en que la interfaz de usuario permite una interacción placentera y satisfactoria.
 - **Accesibilidad:** Cómo el producto puede ser usado por personas con variedad de características y capacidades.
- **Fiabilidad:** Grado en que el producto o sus componentes cumplen las funciones específicas por un determinado periodo de tiempo.
 - **Madurez:** Grado en que el sistema cumple las necesidades de fiabilidad bajo una operación normal.
 - **Disponibilidad:** Grado en que el sistema es operacional y accesible cuando se requiere su uso.
 - **Tolerancia a fallas:** Grado en que el sistema o sus componentes operan como es debido a pesar de la ocurrencia de fallos de *software* o *hardware*.
 - **Capacidad de recuperación:** La capacidad del sistema de recuperar los datos afectados y restablecer el estado deseado ante una interrupción o falla.
- **Seguridad.** Cómo un sistema protege la información y los datos, de modo que las personas o productos tengan el grado de acceso adecuado a sus tipos y niveles de autorización.
 - **Confidencialidad:** Grado en que el sistema asegura que los datos sólo son accesibles por las personas autorizadas.
 - **Integridad:** Grado en que el sistema previene el acceso y modificación de los datos o programas.
 - **No rechazo:** Grado en que es posible demostrar que las acciones han tenido lugar, para no poder ser negadas más tarde.
 - **Responsabilidad:** Grado en que las acciones de una entidad pueden ser asociadas de manera inequívoca a ella.
 - **Autenticidad:** Grado en que la identidad de un sujeto o recurso puede ser comprobada.
- **Mantenimiento:** Grado de la eficiencia y eficacia con la que un producto o sistema puede ser modificado por los mantenedores.
 - **Modularidad:** Grado en que un sistema o *software* está compuesto por elementos discretos, de modo que el cambio en un componente tiene el mínimo impacto en el resto del sistema.
 - **Reusabilidad:** Grado en que un activo puede ser usado en más de un sistema o en la construcción de otro.
 - **Analizable:** Grado de eficiencia y eficacia con que es posible identificar el impacto de un cambio en una parte del sistema, o diagnosticar deficiencias o fallas en alguna de sus partes, o identificar aquellas que deben ser modificadas.
 - **Modificable:** Grado en el sistema que puede ser modificado de manera efectiva y eficiente, sin introducir defectos o degradar la calidad existente.

- **Testable:** Grado en que es posible establecer un criterio para probar el sistema y las pruebas que pueden ser desarrolladas, para determinar que el criterio se ha cumplido.
- **Portabilidad.** Grado de la eficiencia y eficacia con la que un producto o sistema puede ser transferido de un *hardware*, *software* o ambiente de uso a otro diferente.
 - **Adaptabilidad:** Grado en que el producto puede ser adaptado a un *hardware* o *software* diferente de manera eficiente y efectiva.
 - **Instalación:** Grado de efectividad y eficiencia con que el sistema puede ser instalado o desinstalado.
 - **Reemplazo:** Grado en que el producto puede reemplazar a otro para el mismo propósito en el mismo ambiente.

Estos parámetros de calidad se definen también como requerimientos del proyecto, definiendo la importancia de cada apartado en el contexto de la aplicación final.

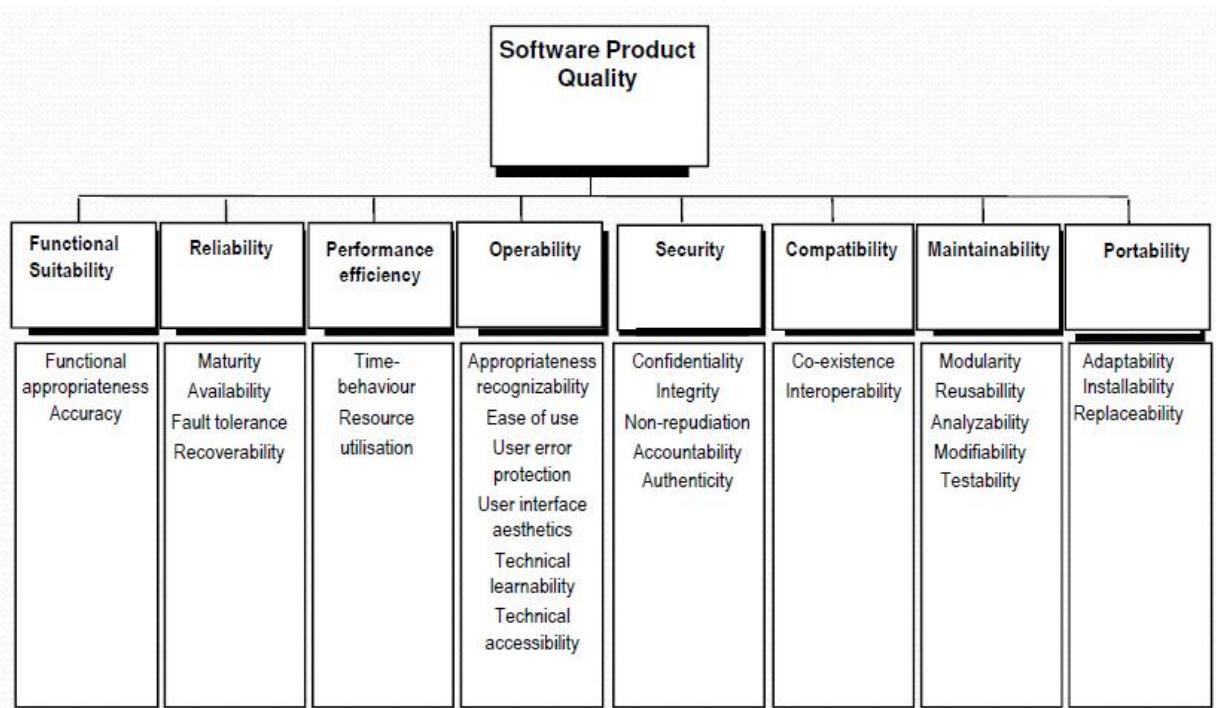


Figura 2.7: ISO/IEC 25010, categorías y subcategorías..

2.3.2. Patrones de diseño

En computación, especialmente en la programación orientada a objetos, se utilizan patrones para sortear la dificultad de generar buenos diseños, seleccionando los objetos pertinentes y sus correspondientes relaciones dentro de la aplicación. La reutilización de soluciones previas es una técnica muy utilizada en desarrollo de nuevo *software*, las que al irse adaptando a nuevos problemas, generan una metodología para resolver cierta clase de ellos. Estas metodologías son llamadas patrones de diseño, y se encuentran bien documentadas para su aplicación sucesiva en cada nueva aplicación [?].

Un patrón de diseño, por lo tanto, describe un problema particular y recurrente dentro de cierto contexto, presentando un esquema genérico y bien probado que resuelve este problema. Describe los componentes que constituyen la solución, sus responsabilidades, relaciones y la forma en que colaboran entre sí [?].

Los patrones de diseño se encuentran documentados de manera bien homogénea a través de una plantilla, que permite comprender rápidamente los siguientes aspectos fundamentales del diseño: cuál es el problema que se resuelve, cuál es la solución propuesta, y las consecuencias de su aplicación tales como ventajas y desventajas. A continuación se detallará uno de los ejemplos clásicos de patrones de diseño, denominado Modelo-Vista-Controlador [?], con el objetivo de exemplificar el concepto.

Modelo-Vista-Controlador.

Este patrón de diseño divide una aplicación interactiva en tres componentes: el modelo, que contiene la funcionalidad base y los datos; la vista, que despliega la información al usuario; y el controlador, que maneja la entrada del usuario para cambiar el estado de la aplicación. Así, la interfaz gráfica se refleja en el controlador y la vista, de modo que cada acción se propaga a través del controlador hacia el modelo, actualizando a la vez la vista [?][?].

- **Contexto:** aplicaciones interactivas con una interfaz de usuario flexible.
- **Problema:** el diseño de una interfaz gráfica de usuario que esté desacoplada del programa principal. La interfaz gráfica por lo general debe ser flexible, permitiendo cambios en su distribución, o la forma de llamar a las funcionalidades, o varios métodos de entrada que llaman a la misma funcionalidad como botones, linea de comandos o menús. También se puede requerir portar la interfaz gráfica a diferentes ambientes o estándares, manteniendo la funcionalidad de la aplicación. Otro caso típico corresponde a la visualización de la misma información de diferentes maneras o en diferentes ventanas, como pueden ser vistas de gráficos o tablas para los mismos datos. Por lo tanto, se requiere desacoplar los datos de la vista para evitar la duplicación o corrupción de la información.
- **Solución:** dividir la aplicación en tres áreas: el modelo, la vista y el controlador. Donde el modelo controla la lógica y funcionalidades internas de la aplicación, la vista gestiona la representación de la aplicación hacia el usuario y el controlador maneja las entradas del usuario.
El usuario sólo interactúa con el controlador, y los cambios realizados en el modelo deben ser propagados hacia la vista. Como cada vista comparte el modelo, los cambios se verán actualizados en cada una de ellas.
- **Estructura:** los componentes que completan la estructura del patrón MVC se detallan a continuación:
 - **Modelo:** encapsula la información y todas las funcionalidades de la aplicación, de manera independiente de su representación gráfica. Provee los métodos para realizar los procesos específicos de la aplicación así como para acceder a los datos de esta. A la vez implementa el mecanismo de propagación de los cambios hacia las diferentes vistas suscritas a este modelo.

- **Vista:** despliega la información al usuario a partir de los datos del modelo y puede permitir múltiples formas de visualizarlo. Provee un método de actualización que es activado por el mecanismo de propagación de cambios del modelo suscrito.
- **Controlador:** cada vista tiene asociada un controlador, que recibe la entrada del usuario, como presionar un botón o seleccionar una entrada de menú. El controlador solicita el servicio correspondiente al modelo, causando a la vez una actualización de la vista si es necesario.

Para clarificar las relaciones entre los componentes del patrón se cuenta con el diagrama de la **figura 2.8**. La esencia dinámica del esquema en ejecución se observa a través del diagrama de colaboración detallado en la **figura 2.9**.

- **Consecuencias:** alguno de los beneficios que provee son: múltiples vistas para el mismo modelo, las cuales pueden estar sincronizadas dado que los cambios en el modelo se propagan a aquellas que lo comparten; vista y controlador intercambiable de manera independiente, permitiendo portar la aplicación a diferentes plataformas sin cambiar la funcionalidad. Por otro lado, los inconvenientes que genera son: aumento de complejidad, al necesitar adaptar cada módulo al controlador y sus funcionalidades; excesivo número de actualizaciones al propagar cambios del modelo a todas las vistas; alto acoplamiento entre la vista y el controlador -aún como elementos separados su diseño está muy relacionado-; acoplamiento entre el modelo y el conjunto vista-controlador, ya que cualquier cambio en la interfaz que se provee, tiene un efecto directo en el funcionamiento de este conjunto, que accede directamente a sus servicios.

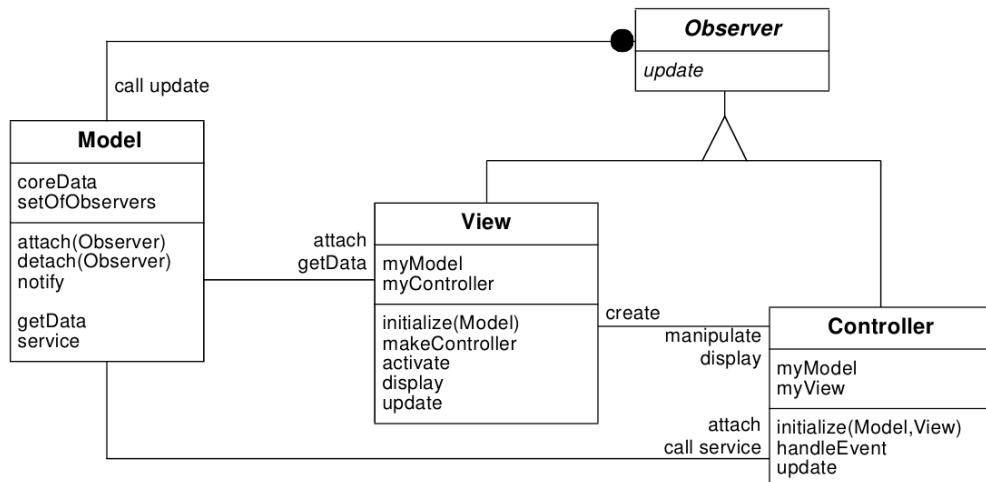


Figura 2.8: Diagrama de clases del patrón de diseño MVC..

A partir del ejemplo, se observa que el proceso de creación de una aplicación parte por la búsqueda de patrones de diseño bien documentados y que solucionen un tipo de problema similar al que se tiene por objetivo. Comparar patrones de diseño cuando se tienen varios candidatos a solucionar el problema, también se hace sencillo dado que la forma de presentar cada patrón permite obtener sus principales características y posible implementación. Además una aplicación puede hacer uso de varios patrones de diseño, en diferentes niveles de su arquitectura para cumplir la totalidad de las especificaciones.

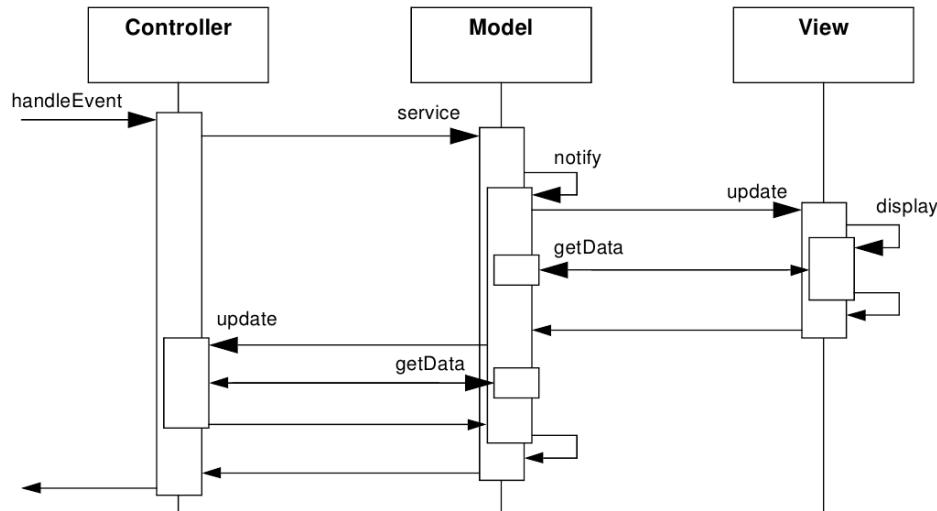


Figura 2.9: Esquema de colaboración del patrón de diseño MVC..

No obstante, se debe considerar que el patrón de diseño en sí, no entregará la solución completa a un problema concreto, en cuanto solo presenta una visión general, sin ahondar en los detalles de implementación o funcionalidades específicas de la aplicación en cuestión. Los patrones de diseño resuelven una clase de problema relacionado, si el que se busca solucionar corresponde con los alcances de algún patrón de diseño, se tiene el punto de partida, pero se debe adaptar y completar el esquema original con los requerimientos y funcionalidades específicas del nuevo desarrollo.

2.3.3. Arquitecturas de *software* basadas en patrones

Si bien el diseño de aplicaciones, basadas en patrones, está muy enfocada en la programación orientada a objetos y se beneficia de sus posibilidades, esta técnica se puede extender más allá y ser aplicada en cualquier paradigma o lenguaje de programación. En efecto, a nivel de diseño de arquitectura de *software*, la mayoría de los patrones sólo requieren cierta capacidad de abstracción del lenguaje como módulos o estructuras de datos [?], lo cual abre las posibilidades de utilizar esta técnica sobre lenguajes procedurales, como los utilizados en el desarrollo de sistemas embebidos.

Command pattern

Dentro de los patrones clasificados como de comportamiento, se encuentra el denominado procesador de comandos, o mejor conocido por su nombre en inglés *command pattern* [?] o *command processor pattern* [?], el cual es de especial interés en este trabajo. Lo esencial de este patrón es separar el requerimiento de un servicio de su ejecución, encapsulándolo en la forma de un comando.

- **Contexto:** aplicaciones que requieren flexibilidad al añadir funcionalidades. Aquellas

orientadas a la ejecución de funciones por parte del usuario, soportando características como llevar un registro de ejecución o deshacer acciones. Aplicaciones donde el instante de generación de un requerimiento es independiente del momento en que se ejecuta.

- **Problema:** una aplicación que requiere dar soporte a una gran cantidad de funcionalidades, se puede ver beneficiada de aislar la forma genérica en que se realiza el llamado a cada función de la implementación misma. También cuando se requiere agregar capacidades como registro de eventos, deshacer, programación de macros o suspender cierta ejecución de un proceso, se ve la necesidad de separar las funciones específicas del núcleo que las gestiona. Un desarrollo que requiere ser altamente escalable, sin afectar el código existente, debe encontrar una manera homogénea de agregar estas nuevas características.
- **Solución:** encapsular los requerimientos en objetos denominados comandos, así cada llamada a una función específica crea uno nuevo, también específico a esa llamada, que implementa la funcionalidad. El patrón describe la estructura de gestión para la generación y ejecución de los comandos la cual es homogénea para cada uno. La adaptabilidad y extensión de la aplicación se realiza mediante la implementación de nuevos comandos sobre el sistema de gestión base.
- **Estructura:** la arquitectura está compuesta por los siguientes módulos.
 - **Comando:** todos los comandos poseen una estructura básica, implementando un método para ejecutarlo. Para cada función requerida en la aplicación, se crea una derivación de la estructura base del comando que implementa la lógica necesaria.
 - **Controlador o cliente:** representa la interfaz de la aplicación y para cada requerimiento crea el comando adecuado, el que es transferido al *invoker*.
 - **Procesador de comandos o invoker:** recibe los comandos, agenda e inicia su ejecución. Este módulo es independiente de cada orden, en cuanto sólo utiliza la interfaz genérica que cada comando debe respetar.
 - **Proveedor o receptor:** provee la funcionalidad del comando en sí, ya que su lógica de ejecución incluye el llamado de uno o varios de los servicios entregados por este módulo, que puede ser representado, por ejemplo, como una librería.
- **Consecuencias:** La aplicación de este patrón de diseño implica los siguientes beneficios:
 - Los comandos pueden ser requeridos de manera flexible, es decir, desde diferentes fuentes según se implemente el controlador correspondiente. De hecho, podría, existir más de un controlador o cliente generando el mismo comando desde diferentes lugares.
 - La aplicación es fácil de modificar y extender a través de la implementación de nuevos comandos, dado que el procesador de comandos sólo trabaja con la interfaz genérica de ellos. Es posible agregar algunos más complejos combinando los ya existentes en una macro.
 - El procesador de comandos, al centralizar la gestión de éstos, es el lugar adecuado para implementar servicios como: registro de comandos ejecutados, filtrar, pospo-

ner, repetir o deshacer la ejecución, o incluso gestionar un sistema de prioridades entre comandos.

Entre las desventajas y limitaciones de este diseño se debe mencionar:

- La cantidad de comandos puede crecer considerablemente en cuanto aumenta la complejidad de la aplicación.
- El cliente que genera un comando tiene poca o nula información sobre el momento en que, efectivamente, este se ejecuta o sobre el resultado de su ejecución.
- Se dificulta la creación de aplicaciones fuertemente basadas en eventos, ya que cuando el comando obtiene los parámetros para su ejecución, puede ser diferente del momento en que se crea.

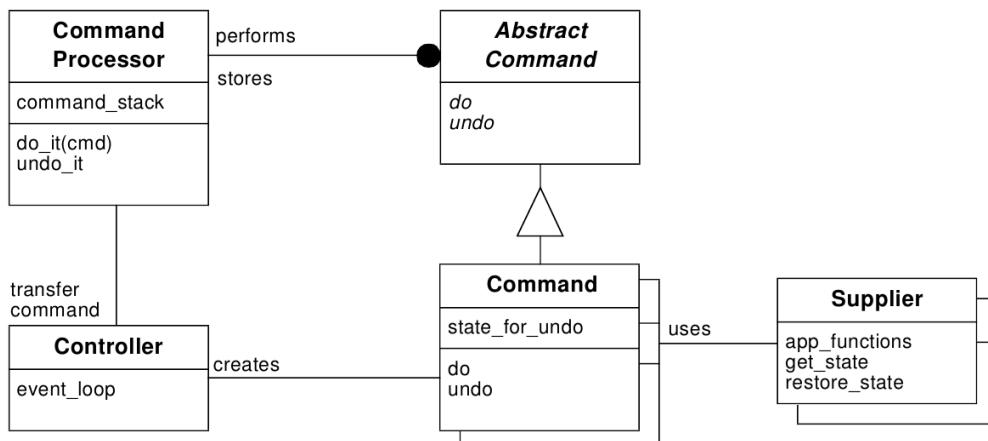


Figura 2.10: Diagrama de clases del patrón de diseño procesador de comandos..

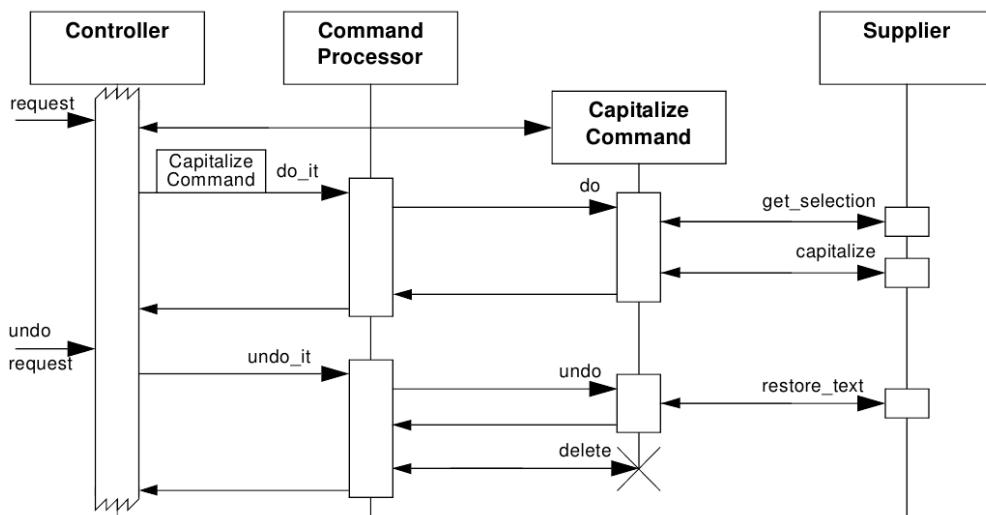


Figura 2.11: Esquema de colaboración del patrón de diseño procesador de comandos..

2.4. Pequeños Satélites

2.4.1. Satélites tipo Cubesat

A partir del año 1999, se comienza a desarrollar el proyecto Cubesat como una iniciativa entre *California Polytechnic State University, San Luis Obispo* y *Stanford University*. Con el objetivo de facilitar el acceso al espacio a pequeños *payloads*, en un corto periodo de desarrollo y a bajo costo. Para esto se crea un nuevo estándar de vehículo espacial, que considera satélites de pequeñas dimensiones, acotados a un cubo de 10 [cm] de arista y un peso máximo de 1.3 kg. como el mostrado en la **figura 2.12 (a)**. Cuando se combinan se pueden conseguir Cubesat de 2 unidades (2U) o tres unidades (3U); un satélite compuesto por un solo cubo se denomina una unidad (1U). El proyecto contempla tanto las especificaciones generales del satélite, así como una plataforma estándar para ser desplegados desde el vehículo de lanzamiento. Este dispositivo se denominada P-POD y consiste en un compartimiento capaz de albergar hasta tres Cubesat de 1U y desplegarlos mediante un sistema de resortes cuando se requiera. La **figura 2.12 (b)** ilustra la estructura de un P-POD.

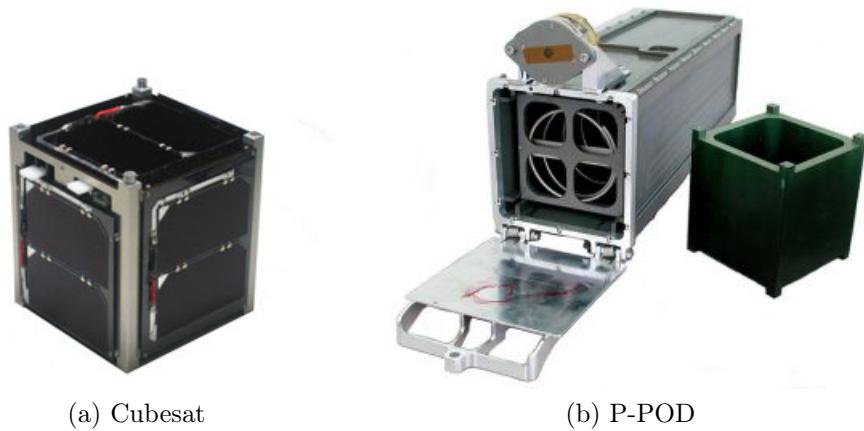


Figura 2.12: Satélite tipo Cubsat

Estándar. Las restricciones que fija el estándar Cubesat se detallan formalmente en las especificaciones de diseño, desarrolladas por *California Polytechnic State University* [?]. A continuación se resumen las principales consideraciones:

- Requerimientos generales

- Todos los componentes deben estar fijos al satélite durante el lanzamiento, despliegue y operación. No se permite liberar elementos extras al espacio.
 - No se permite ningún tipo de elemento explosivo o pirotécnico.
 - La energía química almacenada no debe superar los 100 Wh.

• Requerimientos Mecánicos

- La configuración y dimensiones del satélite deben estar de acuerdo a la **figura 2.13**

- El cubo debe tener dimensiones de 100x100x113 mm. en los ejes x, y, z respectivamente según la **figura 2.13**.
- Los componentes no deben sobresalir más de 6.5 mm. en dirección normal a cada cara.
- Sólo los rieles exteriores de la estructura pueden tener contacto con el P-POD.
- El satélite no debe superar los 1.33 Kg. de masa.
- La estructura externa debe estar construida en aluminio 7075 o 6061, anodizado en los rieles.

• Requerimientos Eléctricos

- Ningún componente electrónico debe estar activo durante el lanzamiento, esto incluye desactivar completamente o descargar las baterías.
- El Cubesat debe poseer un interruptor en la base de uno de sus rieles que permita apagar completamente el satélite cuando está presionado.
- Adicionalmente debe contar con un conector tipo *Remove Before Flight* que debe cortar toda la energía del satélite y será removido una vez se integre en el P-POD.

• Requerimientos de operación

- Cubesat con baterías deben ser capaces de recibir un comando para apagar transmisiones según las regulaciones de la FCC.
- Todos los mecanismos de despliegue del satélite no se deben activar antes de 30 minutos luego del lanzamiento desde el P-POD.
- Transmisores de radio de potencia mayor a 1 mW. no deben funcionar antes de cumplirse 30 minutos luego del despliegue desde el P-POD.
- Se debe contar con la licencia de uso de frecuencia de radio. Para frecuencias *amateur* la coordinación se realiza a través de la *International Amateur Radio Union* (IARU).

El estándar mencionado corresponde a un Cubesat de una unidad (1U). Opcionalmente, se pueden combinar hasta tres unidades para obtener satélites de 2U o 3U. Esto permite contar con mayor volumen, para posicionar los componentes físicos del satélite y una mayor superficie para situar paneles solares y así brindar mayor autonomía energética.

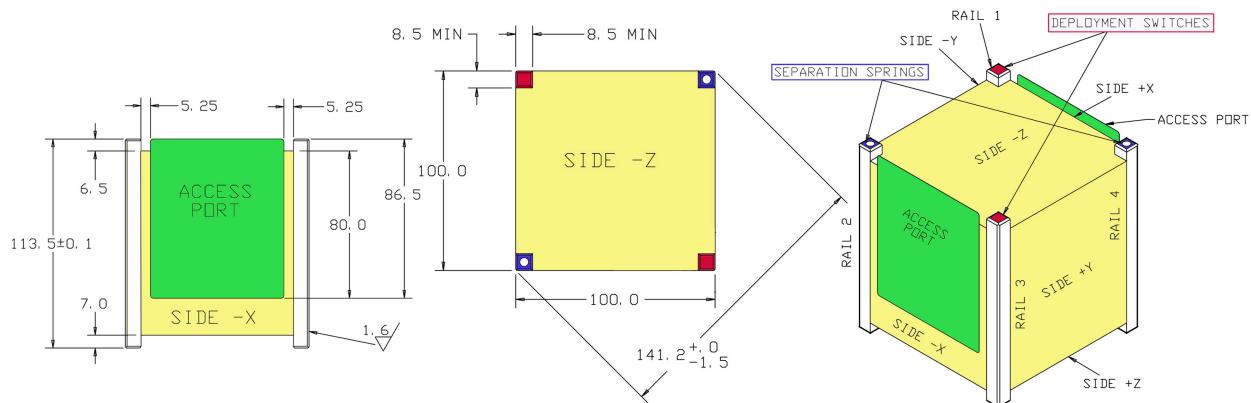


Figura 2.13: Especificaciones de diseño y dimensiones del estándar Cubesat de una unidad [?].

Aplicaciones. En la actualidad cerca de un centenar de satélites tipo Cubesat han sido lanzados de manera exitosa. Las aplicaciones con posibilidades de ser desarrolladas a través de este tipo de tecnologías incluyen proyectos con fines educacionales, pruebas de nuevas tecnologías espaciales, proyectos de investigación científica y desarrollos privados.

Desde el punto de vista educacional, dadas las características de rápida construcción a un costo comparativamente bajo, este tipo de tecnologías abre las posibilidades de investigación y desarrollo de proyectos en materia aeroespacial a instituciones educacionales y gobiernos de todo el mundo. A esto se suma como antecedente que la mayoría de los proyectos de satélites Cubesat, han sido ejecutados por estudiantes universitarios, utilizando componentes comerciales o desarrollos propios. El valor educacional de un proyecto satelital de estas características incluye: el desarrollo de nueva tecnología aeroespacial, que puede ser probada a pequeña escala en ambientes realistas; la formación de profesionales entrenados en el área, mediante la experiencia práctica que brinda la ejecución de este tipo de proyectos; la aplicación de metodologías en la formación académica que incluyan el trabajo en equipos multidisciplinarios, grupos de alto desempeño, gestión, entre otros.

Contando con una plataforma estándar de vehículo espacial, se abre la posibilidad al desarrollo de diferentes *payloads* de tipo científico, capaces de tomar diferentes datos en el espacio para su posterior análisis o uso en proyectos de investigación. Entre las aplicaciones más destacadas, se encuentra la observación remota de la tierra mediante el uso de sensores para tomar datos sobre la ionosfera y termosfera, o el uso de cámaras fotográficas que otorguen imágenes para estudios posteriores. Este tipo de misiones y sus datos, pueden llegar incluso a poner a prueba sistemas de predicción o alerta temprana de desastres [?].

Iniciativas privadas o de propósito general, tales como redes de comunicaciones de apoyo en caso de catástrofe, redes de comunicaciones privadas, monitoreo remoto de plantaciones y proyectos de exploración minera, son algunas de las posibles aplicaciones capaces de ser desarrolladas como un proyecto aeroespacial tipo Cubesat.

2.5. Proyecto SUCHAI

El proyecto satelital del Departamento de Ingeniería Eléctrica de la Universidad de Chile, con la participación de académicos, ingenieros y estudiantes, se denomina *Satellite of University of Chile for Aerospace Investigation* (SUCHAI) y consiste en el desarrollo, puesta en órbita y operación del primer satélite creado netamente en el país.



Figura 2.14: Logo del proyecto SUCHAI.

Se trata de un satélite tipo Cubesat de una unidad, con fines educacionales y científicos. El objetivo del proyecto es poner en órbita un satélite, desarrollado por estudiantes de la carrera de ingeniería, que sea capaz de enviar un *beacon* para ser escuchado y decodificado por la estación terrena, también ejecutar experimentos asociados a *payloads* y enviar como telemetría la información de funcionamiento del satélite y los datos recolectados. Por otro lado, su ejecución permitirá adquirir el conocimiento necesario para desarrollar proyectos satelitales de manera local, que será la base para futuras misiones relacionadas. Además posibilitará la integración de alumnos de pregrado en equipos multidisciplinarios, que requieren proponer soluciones no triviales a un problema abierto.

El satélite se compone de tres sistemas básicos: el computador a bordo, que consiste en un microcontrolador de gama media, para ejecutar el *software* que controla su operación en órbita; el sistema de comunicaciones, compuesto por un transmisor y receptor de radio UHF, así como antenas para desplegar durante la operación; y un sistema de control de energía o EPS que incluye baterías y paneles solares, para proveer la energía eléctrica que permite operar al satélite así como también, cargar baterías. Como carga útil se considera la integración: una cámara digital que pueda realizar tomas de la tierra; sensores de temperatura; giróscopos; un sensor para medir la densidad y temperatura de partículas de la ionosfera con una *langmuir probe*; y un dispositivo para realizar un estudio estadístico de la transferencia de potencia en ambientes de microgravedad y la disipación de calor de la electrónica en ambientes escaso de aire. En la **figura 2.15** se observa el satélite SUCHAI en una etapa temprana de integración junto a uno de los *payloads* de la misión.

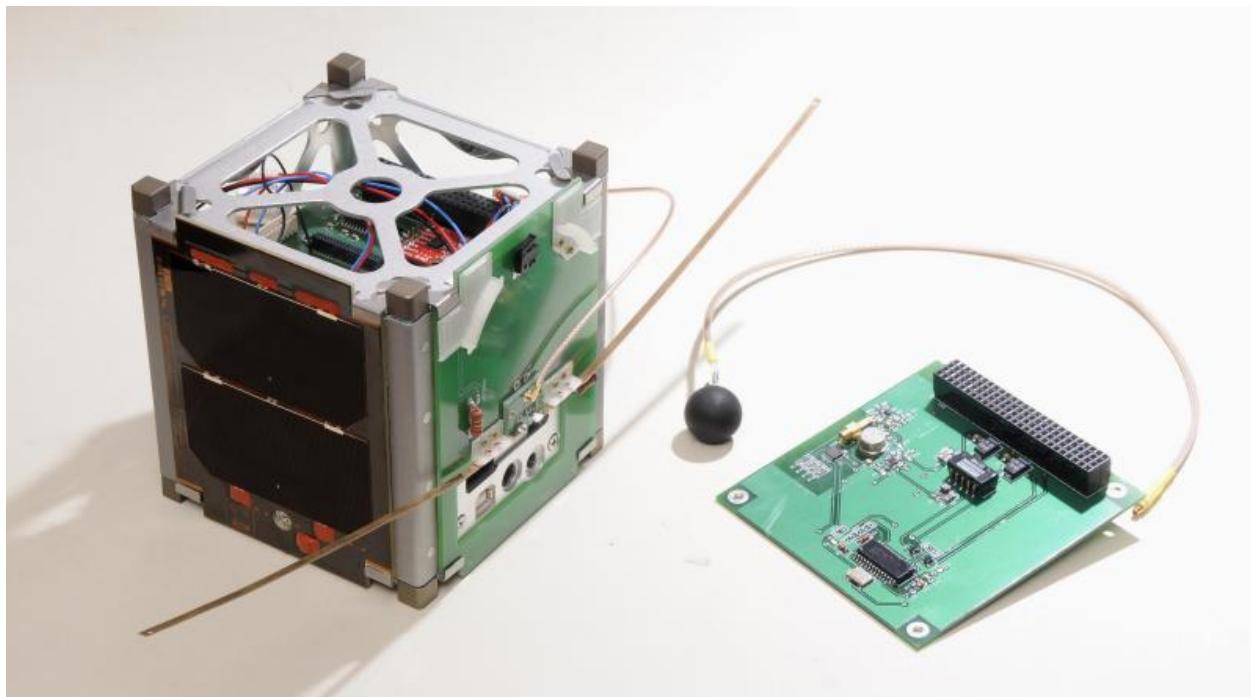


Figura 2.15: Satélite SUCHAI junto a una *langmuir probe*.

Capítulo 3

Diseño del *software*

En este capítulo se describe el proceso de diseño del *software* de vuelo para el satélite. El proceso considera, en primera instancia, los requerimientos operacionales de la misión; requerimientos no operacionales, relacionados con la calidad del *software*; y la plataforma de *hardware* objetivo, para tener claro las limitaciones y alcances de la solución. El diseño de la aplicación se detalla en diferentes niveles, incluyendo una visión global sobre los componentes principales de esta y se centra, en específico, en la propuesta de una arquitectura de alto nivel basada en un patrón de diseño. Finalmente, se realiza un análisis de la solución, esto con la finalidad de plantear una arquitectura final que permita implementar las funcionalidades detalladas en los requerimientos operacionales.

3.1. Requerimientos

3.1.1. Requerimientos operacionales

Se refieren a las funcionalidades que se espera que el computador a bordo del satélite SUCHAI deba realizar. Son los requisitos básicos que el sistema debe cumplir para considerar que se cuenta con un satélite capaz de llevar a cabo la misión. La lista de requerimientos proviene de una serie de reuniones sostenidas con los integrantes de los diferentes grupos de trabajo, todo de acuerdo con los lineamientos del jefe de proyecto.

Área de comunicaciones

Configuración inicial del *transcevier*: el satélite debe ser capaz de fijar, si lo permite, las configuraciones iniciales del sistema de comunicaciones, por ejemplo: encender el *transcevier*, silenciar las comunicaciones durante determinado tiempo luego del lanzamiento, configurar la frecuencia de transmisión a la asignada por la IARU y la potencia si el equipo lo permite, configurar y encender la baliza o *beacon*, entre otros.

Se debe almacenar de manera permanente estas configuraciones iniciales, para así poder reconfigurar el *transcevier* en caso de reinicio o falla, también para permitir un ajuste de parámetros durante el desarrollo de la misión.

Despliegue de antenas: el satélite, luego de transcurrido el tiempo de silencio radial obligatorio, debe desplegar las antenas de comunicaciones. Esto se realiza mediante la activación de un sistema eléctrico, que cuenta con retroalimentación para comprobar que las antenas fueron efectivamente desplegadas. Esta operación puede requerir sucesivos intentos hasta que las antenas sean desplegadas.

Procesamiento de telecomandos: el sistema de comunicaciones debe ser capaz de recibir telecomandos desde la estación terrena, y el *software* de control deberá recogerlos para ejecutar las acciones requeridas. Esto incluye: la definición de un formato de telecomandos, la capacidad del sistema de analizar los comandos y sus argumentos dentro de un paquete de comunicaciones, y la posterior ejecución del comando recibido.

Protocolo de enlace: el satélite debe ser capaz de recibir y enviar datos a la estación terrena. Para esto se requiere, en primer lugar, que se pueda rastrear mediante la transmisión de una señal de baliza o *beacon*. Segundo, el satélite debe establecer comunicación con la estación terrena para determinar si se recibirán órdenes o si se descargará información. Y tercero, proceder con las operaciones de descarga y subida de datos.

Envío de telemetría: el satélite recolectará los datos requeridos por la misión, los que incluyen información general sobre el estado de funcionamiento de todos los subsistemas, así como los datos generados por los *payloads* abordo. El envío de telemetría puede ser automático cada vez que el satélite se enlace con la estación terrena, o bien bajo demanda a través de telecomandos que indiquen el tipo de información que es requerida.

Control central

Organizar telemetría: durante la misión se generarán datos que provienen de diferentes subsistemas o *payloads*. Se debe contar con un medio de almacenamiento con la capacidad adecuada para guardarlos, con un sistema de organización de los diferentes datos con el objetivo de ser requeridos de manera selectiva, para así poder ser enviados como telemetría a la estación terrena.

Plan de vuelo: se debe ser capaz de recibir y ejecutar un plan de vuelo, consistente en una serie de acciones a realizar en momentos determinados de tiempo. El plan de vuelo puede ser precargado en el satélite antes de ser lanzado, así como también debe ser actualizado mediante telecomandos. Esto provee flexibilidad en las tareas que se ejecutarán durante la misión, permitiendo controlar el uso de los diferentes recursos del satélite.

Obtener el estado del sistema: de manera autónoma, el *software* de control debe recolectar información básica sobre el estado del satélite en general. Esta información será usada para determinar la salud del sistema y tomar las acciones necesarias en órbita, o bien será descargada como telemetría para ser posteriormente analizada. Ejemplos de variables asociadas al estado del sistema son: el nivel de carga de las baterías, la hora del reloj interno, el estado del dispositivo de comunicaciones, el estado del computador abordo, entre otras.

Inicialización del sistema: el *software* control debe poseer un algoritmo de inicio del sistema en general, que considera la configuración del *software* con los parámetros adecuados, la inicialización de otros módulos o subsistemas, así como las obligaciones de silencio radial durante el lanzamiento, y el despliegue de antenas. Debe tener la capacidad de reiniciarse de manera segura manteniendo variables fundamentales y la consistencia con el estado anterior al reinicio

Área de energía

Estimación de la carga de la batería: el *software* de control debe considerar un método de estimación de la carga de las baterías. Esta información debe estar a disposición como una variable, para poder ser utilizada por el subsistema de control de energía utilizado por el satélite.

Presupuesto de energía: se debe considerar la cantidad de energía disponible en el satélite para ejecutar las acciones requeridas. Asimismo se debe tener claro el presupuesto energético de cada acción que se realiza en el satélite. El *software* de control debe plantear una estrategia para evitar que se ejecuten acciones que estén fuera de la capacidad energética disponible así como una manera de planificar el consumo de energía de la misión.

Órbita

Actualizar parámetros de órbita: se debe contar con una estrategia de estimación y actualización de los parámetros de órbita del satélite. Esto con el objetivo de contar con la información necesaria para realizar acciones dependientes de la posición real de satélite. Ejemplos de este tipo de acciones son la ejecución de un experimento, algún *payload* o el enlace con una estación terrena para descargar datos de telemetría.

Payloads

Ejecución de comandos de *payloads*: el sistema debe tener la capacidad de controlar diferentes módulos asociados a *payloads* del satélite. Se debe considerar que los *payloads* abordo pueden variar de misión en misión, e incluso pueden ser descartados o agregados en etapas tardías del proyecto. Por eso el *software* debe ser flexible en la capacidad de agregar

o eliminar módulos que se relacionen con el control de *payloads* sin afectar al resto de los sistemas.

Tolerancia a fallos

El sistema debe tener cierto grado de tolerancia a fallos de *hardware* y *software* que permitan mantener la misión operativa. Debido a las adversas condiciones del medio espacial y la incapacidad de acceder directamente al dispositivo, este debe ser capaz de:

- Restablecer su funcionamiento normal luego de un reinicio, evitando la pérdida de información.
- Restablecer, en la medida de lo posible, el funcionamiento del sistema ante fallas causadas por radiación.
- Recuperarse ante fallas de *software*.
- Establecer modos de funcionamiento a prueba de fallos.
- Detectar y solucionar problemas al desplegar antenas.
- Capacidad de *debug* durante el desarrollo y previo al lanzamiento.

3.1.2. Requerimientos no operacionales

Por requerimientos no funcionales, se entienden aquellos atributos asociados a la calidad del *software*, o bien criterios que permiten determinar cómo debería ser la aplicación que se está diseñando. A diferencia de los requerimientos funcionales que explican lo que el *software* debería hacer, los no funcionales explican las cualidades y restricciones que guiarán el proceso de diseño.

El eje principal para el diseño del *software* de control del satélite, responde a contar con una aplicación que sea altamente mantenable debido a dos factores básicos: primero, el desarrollo será incremental, estará a cargo de más de una persona, por lo tanto, debe ser flexible en la adición de funcionalidades, desacoplando módulos para que las intervenciones en el código sean lo más acotadas posible; segundo, el sistema debe ser la base para futuras misiones aeroespaciales, por lo tanto, debe ser fácilmente adaptable a nuevos requerimientos funcionales que incluyen la adición de nuevos *payloads* y sus módulos de control. En el futuro, la arquitectura diseñada debe proveer la capacidad de análisis para los nuevos desarrolladores, con el objetivo de determinar claramente qué módulos se deben intervenir, para agregar nuevas funcionalidades o corregir posibles errores. Especial mención requiere la necesidad de expandir la aplicación, pues se considera como filosofía de trabajo, el contar siempre con un sistema funcional ante la eventual posibilidad de lanzar el satélite. Así, se partirá con una implementación que realice las operaciones básicas o requerimientos mínimos, para así agregar complejidad y funciones al *software* de manera incremental. Lo anterior, conduce a poner especial énfasis en el diseño de una arquitectura que genere un *software* modular, reusable, analizable y modificable, elementos agrupados en la característica denominada “mantenible” en la norma ISO/IEC 25010 [?].

La fiabilidad del sistema es un elemento importante en cualquier misión espacial, debido al ambiente extremo en el cual se desarrolla la misión, lo que incluye grandes cambios de temperatura y efectos de la radiación solar sobre los componentes electrónicos. Esto significa que la aplicación debe tener un nivel adecuado de tolerancia a fallos, y ser capaz de recuperarse ante una interrupción o reinicio. En lo que a *software* respecta, la característica de tolerancia a fallos será considerada en su nivel básico, debido a que, por lo general, esto significa diseñar sistemas altamente redundantes que elevan la complejidad del diseño y el costo de implementación. Lo anterior se contradice con otros requerimientos no operacionales y con la filosofía propia de los satélites tipo Cubesat que, por lo general, utilizan componentes de tipo comercial no diseñados, necesariamente, para su uso en el espacio.

La idoneidad funcional es un requisito importante desde el siguiente punto de vista: si bien, en la etapa de diseño, no se busca obtener una solución detallada de la implementación de cada uno de los requerimientos operacionales, la arquitectura seleccionada debe tener una respuesta a cada función necesaria de implementar, para ser considerada como válida. En las primeras iteraciones se buscará cumplir con los requisitos mínimos de la misión, pero la solución debe ser la idónea para delinejar la forma de agregar todas las funcionalidades requeridas y que las tareas sean llevadas a cabo de manera correcta.

Existen algunas características de calidad que no serán relevantes en este diseño, ya que al ser la primera aproximación en la materia para el equipo de desarrollo, se requiere mantener cierto nivel de simplicidad en la solución, luego probarla y así ganar la experiencia necesaria en proyectos aeroespaciales. Por ejemplo, el desempeño, referido a términos computacionales, no es una restricción importante por la siguientes razones: se cuenta con una plataforma de baja capacidad de cómputo y limitados recursos energéticos; el sistema requiere realizar una cantidad baja de tareas y aquellas de alta demanda computacional pueden ser ejecutadas en tierra; y no se consideran acciones que requieran una gran precisión de tiempo. La seguridad tampoco es una componente fundamental, si bien, se podría tratar de evitar un uso mal intencionado de la plataforma por parte de otros operadores satelitales, salvo por errores, esto es poco probable; por el contrario se busca obtener la mayor cooperación posible de otros operadores como la comunidad de radio-aficionados. Por último en el caso de la portabilidad, lo único importante es determinar claramente los diferentes niveles de abstracción de la arquitectura y su intercomunicación, debido a que las plataformas objetivo cuentan con muy bajo nivel de estandarización en los niveles más cercanos al *hardware*.

En definitiva, la **tabla 3.1** resume los requerimientos no funcionales del proyecto, asignándole una determinada prioridad o importancia a considerar en el diseño.

3.1.3. Requerimientos mínimos

Para que el sistema desarrollado en el proyecto SUCHAI pueda considerarse como un satélite, el vehículo debe satisfacer al menos los requerimientos mínimos dispuestos en la tabla 3.2.

Dada la naturaleza iterativa de la metodología de trabajo, es importante considerar los requerimientos mínimos a la hora de la implementación de la arquitectura pues dará al

Tabla 3.1: Requerimientos no funcionales

Características		Importancia			Observaciones
Categoría	Subcategoría	Poca	Media	Alta	
Idoneidad Funcional	Compleitud funcional Correctitud funcional Adecuación Funcional		X X X		El software debe entregar solución a todos los requerimientos operacionales
Eficiencia del desempeño	Tiempo Utilización de recursos Capacidad	X	X X		Pocas restricciones de tiempo Debe caber en el sistema embebido No se pretende sobrecargar el sistema
Compatibilidad	Co-existencia Interoperabilidad	X	X		El software controla todo el sistema Comunicación con subsistemas
Usabilidad	Reconocible como apropiado Aprendizaje Operabilidad Protección de cometer errores Estética de la interfaz de usuario Accesibilidad		X X X X X		El sistema funciona principalmente de manera utópica. Su operación se lleva a cabo por expertos
Fiabilidad	Madurez Disponibilidad Tolerancia a fallas Capacidad de recuperación		X X X X		Se busca llegar a un nivel aceptable de fiabilidad. Alta incertidumbre debido a la inexistencia de experiencia previa
Seguridad	Confidencialidad Integridad No rechazo Responsabilidad Autenticidad	X	X X X X		No se consideran estos aspectos en este primer diseño. Se prefiere simplicidad.
Mantenimiento	Modularidad Reusabilidad Analizable Modificable Testeable		X X X X		Sistema altamente modular Base para futuras misiones Arquitectura clara Flexibilidad en agregar funcionalidades Pruebas de funcionalidad
Portabilidad	Adaptabilidad Instalación Reemplazo		X X	X	Capacidad de agregar nuevos modulos Instalación experta No aplica

Tabla 3.2: Requerimientos mínimos

Área	Requerimiento
Energía	Proveer energía
	Monitorizar el estado de carga de las baterías
Control central	Recoger telemetría periódicamente
	Ejecutar comandos
Comunicaciones	Despliegue de antenas
	Emitir beacon
	Enviar telemetría
	Recibir telecomandos

sistema la simplicidad necesaria asegurando funcionalidad. La totalidad de los requerimientos operacionales se obtendrá a través de mejoras incrementales sobre los requerimientos mínimos ya mencionados.

3.2. Plataforma

La plataforma para la cual se diseña el *software*, corresponde a un kit Cubesat de una unidad (1U) de la compañía Pumpkins Inc. similar al mostrado en la figura 3.1 (a). El kit está compuesto por los siguientes elementos:

- Chasis de 1U tipo esqueleto (ver figura 3.1 (b)).
- Placa madre rev. D.
- Módulo de procesador D1 con PIC24FJ256GA110.
- Placa de desarrollo.
- Programador y *debugger* Microchip ICD3.

Adicionalmente, se cuenta con los siguientes dispositivos para EPS y comunicaciones respectivamente, que completan la base del vehículo satelital:

- EPS Clyde Space CS-1UEPS2-NB
- *Transceiver* Allspace AS-COM01

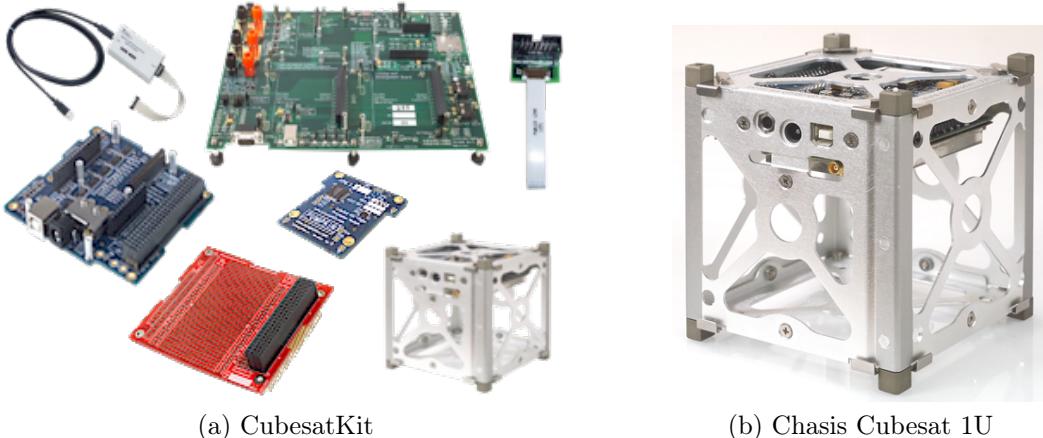


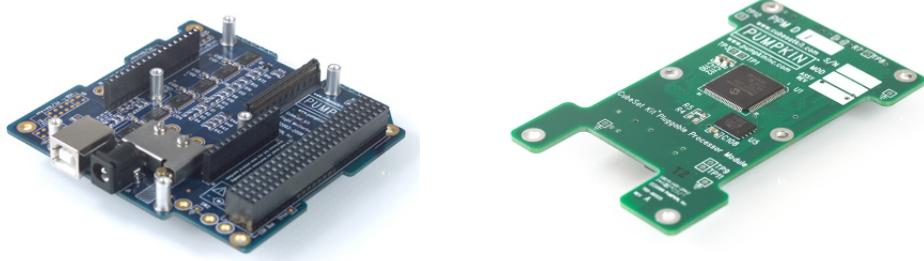
Figura 3.1: CubesatKit de Pumpkin Inc.

Por lo tanto, se cuenta con todo el *hardware* básico para el funcionamiento del satélite y se considera fijo para el proceso de diseño del *software* de vuelo.

3.2.1. Computador a bordo

El computador a bordo del satélite, está compuesto por una placa madre que contiene una interfaz PC104, a través de la cual se conectan el resto de los subsistemas, y por el módulo para el procesador que se integra directamente a la placa madre.

La placa, cuenta además con un reloj de tiempo real que se comunica por un bus I2C, el cual puede ser usado como reloj, alarma y/o *watchdog* externo. Cuenta con un *slot* de memoria SD, para albergar un medio de almacenamiento masivo de hasta 2 GB.; una interfaz



(a) Placa madre

(b) Módulo del procesador

Figura 3.2: Dos módulos que componen el computador a bordo del satélite

USB 2.0 para comunicaciones previas al lanzamiento; así como la electrónica para proveer alimentación al bus PC104 y al módulo del procesador [?].

El módulo del procesador cuenta con un microcontrolador PIC24FJ256GA110; una memoria flash de 64 Mbit con interfaz SPI; osciladores; circuitos de alimentación, protección de sobre corriente y *reset*. El microcontrolador posee las siguientes características [?]:

- Arquitectura de 16 bit.
- Memoria de programa flash de 256 kB.
- 16KB de memoria SRAM.
- Hasta 16MIPS @ 32 MHz.
- 4 UARTs, 3 SPIs, 3 I2Cs.
- ADC de 10bit, 16 canales, 500 ksps.
- 5 *timers* de 16 bit.
- RTCC, WDT
- Selector de pines de periféricos.

La capacidad de cómputo de la plataforma es muy limitada, así como la cantidad de memoria de programa y de datos disponibles. Esto supone fuertes restricciones tanto en el diseño como en la implementación del *software*, por lo que no se puede recurrir a alternativas como un sistema operativo Linux o la utilización de lenguajes de programación interpretado tipo Java o Python. Las implicancias de las restricciones que impone la plataforma de *hardware* derivan en los siguientes puntos:

- Se deberá utilizar las herramientas de desarrollo que provee el fabricante del microcontrolador, es decir, lenguaje C para el compilador XC16 de Microchip.
- Se debe efectuar un trabajo de bajo nivel, implementando *drivers* para los periféricos del microcontrolador y para cada dispositivo que se agregue al sistema.
- Solo existen algunos sistemas operativos básicos para este tipo de dispositivos, y en general solo permiten organizar el *software* en módulos, procesos o tareas que se ejecuten de manera concurrente.
- Existe una cantidad muy limitada de código previo que se pueda reutilizar para implementar el diseño de la aplicación, descartando de plano la posibilidad de utilizar

librerías para bases de datos, protocolos TCP, UDP o POSIX, como en otros proyectos de similares características [?].

Con todo el diseño de la aplicación final será a medida y no deriva de un trabajo previo, con el objetivo de ajustarse a los requerimientos y restricciones de la mejor manera posible.

3.3. Arquitectura de *software*

La arquitectura del *software* de vuelo consiste en diseño conceptual, con un alto grado de abstracción, de los principales módulos que conformarán la aplicación y la manera de relacionarse entre sí.

Se analizará esta arquitectura a diferentes niveles, partiendo por una visión general de la aplicación, donde se identifican las principales áreas en que se divide. Cada una de estas áreas o capas de abstracción presenta sus propias particularidades, por lo que se requiere ahondar en el diseño de la arquitectura de cada uno de ellos.

3.3.1. Arquitectura Global

A nivel global, el *software* de vuelo, se concibe como una serie de capas que agrupan funcionalidades similares, e interactúan según una dinámica en que la capa inferior es una prestadora de servicios para la superior [?]. Este diseño se denomina arquitectura de capas y es una buena forma de generar sistemas portables entre diferentes plataformas de *hardware*. Esto porque, siempre que se mantengan las interfaces entre capas, cualquiera de ellas puede ser reemplazada por una implementación diferente.

La figura 3.3 detalla una arquitectura de tres capas apropiada, en general,s para el diseño de *software* en sistemas embebidos donde se pueden distinguir al menos los siguientes niveles: capa de bajo nivel relacionada con los controladores de *hardware*, también llamada capa de abstracción de *hardware* o HAL por sus siglas en inglés; la intermedia que corresponde al nivel del sistema operativo o gestor de tareas; y una superior que corresponde a la aplicación final del proyecto en cuestión.

La capa de más bajo nivel corresponde a los controladores de *hardware*, que pueden ser varios módulos específicos para cada componente físico o subsistema presente, y permite a las capas superiores acceder a ellos sin conocer en detalle sus particularidades. Por ejemplo, provee los servicios para acceder al periférico de comunicaciones o subsistemas externos, como pueden ser los diferentes *payloads*. Por lo general, existen varias alternativas de *hardware* disponibles para un mismo objetivo, o bien se requiere que el sistema se adapte a la presencia de diferentes *payloads*, por lo que la integración de varios controladores en este nivel permite mantener el resto del sistema intacto, siempre y cuando se respeten las interfaces entre capas.

La capa intermedia corresponde al sistema operativo, que provee soporte para la gestión de tareas en el sistema embebido, permitiendo la ejecución concurrente de procesos. A este

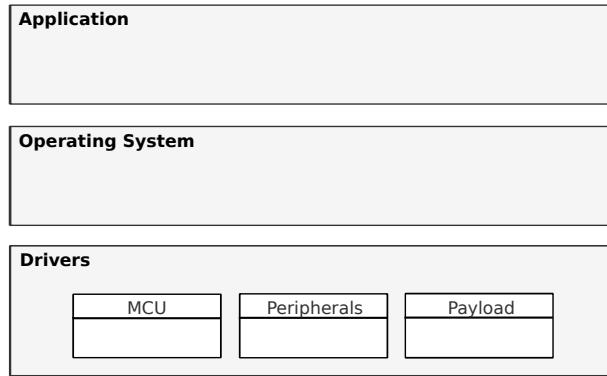


Figura 3.3: Arquitectura de tres capas para un sistema embebido..

nivel en el diseño, la decisión de qué sistema operativo utilizar no es relevante, pues toda la lógica de gestión de tareas está concentrada en esta capa que utilizará las funcionalidades dadas por los niveles inferiores y que provee la base para implementar la aplicación.

La capa superior es específica a la aplicación e implementa la funcionalidad final del sistema embebido que se diseña. Nuevamente se observa la ventaja de utilizar un diseño de capas en la arquitectura global, en cuanto la capa inferior e intermedia será común en cualquier sistema embebido y el diseño de la aplicación final puede contar con estos servicios. Esto significa que al implementar ambos niveles, se tiene un porcentaje considerable de un desarrollo ya disponible.

Una de las principales desventajas de esta arquitectura, sobre todo en sistemas embebidos, es que puede existir la necesidad de una comunicación entre capas no adyacentes, rompiendo la arquitectura [?] propuesta. Esto se debe tratar de evitar, o al menos realizar de manera controlada.

3.3.2. Controladores de hardware

En todo sistema computacional se requiere de una capa de bajo nivel que realice la interfaz entre el *hardware* y el *software*, para entregar las funcionalidades de configurar, controlar y deshabilitar adecuadamente las diferentes dispositivos. Se llaman controladores o *drivers* a aquellas librerías que permiten inicializar y manejar el acceso a este *hardware* a las capas de sistema operativo y de aplicación [?].

Se requiere al menos un controlador para cada componente electrónico existente en el sistema embebido, y se diseñan bajo la idea de agrupar aquellos de similares características o funcionalidades bajo una interfaz común, que permita abstraer las particularidades de la implementación de cada uno. Hacia las capas superiores se maneja el concepto de “qué hace” el dispositivo, entregando funciones básicas que persisten entre diferentes arquitecturas de *hardware*, tales como: la inicialización, encendido o habilitación del dispositivo; su configuración; la lectura y escritura de datos; así como deshabilitar o apagarlo. A esto se suman todas

aquellas funcionalidades específicas del componente.

Hacia la capa de *hardware* el controlador maneja el “cómo funciona”, lo que incluye, el control de las interfaces de entrada y salida de datos, manejo de las interrupciones, o de la memoria del dispositivo. Si bien la implementación de cada *driver* es específica al dispositivo, se pueden diferenciar al menos tres tipos de arquitecturas comunes en el diseño de estas piezas de *software*: controladores de entrada y salida síncronos; controladores de entrada y salida asíncronos; y colas de entrada de datos seriales.

Controladores de entrada y salida síncronos

Se considera el uso de una arquitectura de controladores síncronos, cuando las tareas que lo invocan necesariamente deben esperar la respuesta del controlador. Si se provee la adecuada sincronización, entonces el resto del sistema puede seguir funcionando mientras la tarea en cuestión se encuentra esperando. Cuando termina el proceso de entrada o salida de datos, la tarea retoma su funcionamiento.

La arquitectura correspondiente se detalla en la figura 3.4, donde se observa que el *driver* se provee como un módulo, o función, que es llamado por alguna capa superior del *software*. Una vez que el controlador tiene acceso al dispositivo, a través de alguna estructura de sincronización, realiza las operaciones de entrada y salida. Estas operaciones se pueden realizar a través de rutinas de atención de interrupciones, o bien mediante un *polling* al estado del dispositivo. Si se usa la rutina de atención de interrupciones, esta también se considera parte del controlador.

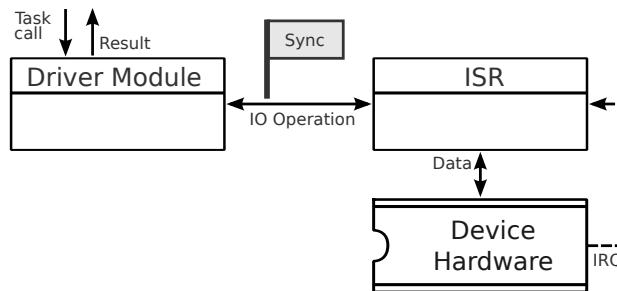


Figura 3.4: Arquitectura para controladores síncronos.

El módulo del controlador realiza las siguientes acciones:

- Iniciar las operaciones de entrada y salida.
- Esperar por una estructura de sincronización.
- Obtener el estado o la información desde el dispositivo.
- Retornar la información requerida.

Cuando se utiliza una rutina de atención de interrupciones, ésta se encarga de las siguientes operaciones:

- Atender la interrupción y restablecer el estado del dispositivo.
- Obtener los datos a enviar o escribir nuevos datos en el dispositivo
- Liberar la estructura de sincronización cuando se terminen las operaciones.

Controladores de entrada y salida asíncronos

En ocasiones, la tarea que solicita las acciones al controlador, puede continuar su ejecución sin esperar el resultado. En este caso, se habla de un controlador asíncrono. Este tipo de *driver* no es común de encontrar, y por lo general se pueden evitar, cuando no tiene sentido avanzar en la ejecución sin que se terminen las operaciones de entrada y salida. Un caso especial es cuando se realizan operaciones en varias etapas, aquí, la tarea puede procesar los datos de la primera etapa, mientras el controlador obtiene los datos de la siguiente.

Este tipo de arquitectura se detalla en la figura 3.5. Se observa que el *driver* lo componen el módulo o función que ejecuta la operaciones, la rutina de atención de interrupciones, así como una cola de mensajes que almacena los resultados parciales de la operación de entrada o salida. La sincronización, a través de esta cola, permite a la tarea procesar en paralelo la información que se le entrega.

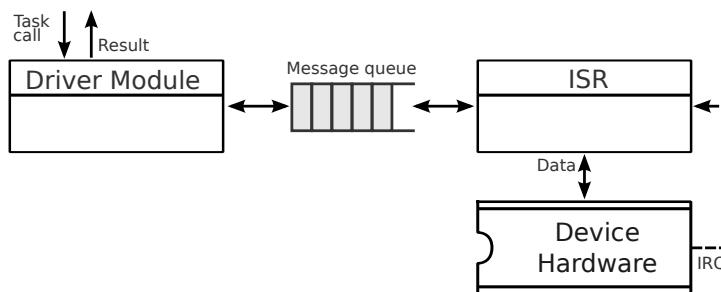


Figura 3.5: Arquitectura para controladores asíncronos.

Las operaciones que realiza el controlador son las siguientes:

- Esperar a que lleguen mensajes a la cola.
- Obtener el mensaje y entregar la información a la tarea.
- Continuar con nuevas operaciones de entrada o salida.

Mientras que en la rutina de atención de interrupciones se realiza lo siguiente:

- Atender la interrupción y restablecer el estado del dispositivo.
- Obtener la información desde (o enviar hacia) el dispositivo de hardware.
- Empaquetar la información en un mensaje.
- Agregar el mensaje a la cola.

Cola de entrada de datos seriales

Un caso particular de *driver* asíncrono, que se observa de manera común en sistemas embebidos, corresponde a la entrada de datos seriales asíncronos. En este tipo de controladores se cuenta con la llegada de una gran cantidad de datos de manera serial, y el término de la operación está dado por la cantidad máxima a recibir o por algún indicador, en los mismos datos, del término de la secuencia.

La arquitectura que responde a esta situación se detalla en la figura 3.6. En este caso, en adición a los módulos mencionados anteriormente, se agrega un *buffer* de memoria que es creado al inicio de las operaciones de entrada de datos, y es accedido por referencias para evitar el uso adicional de memoria.

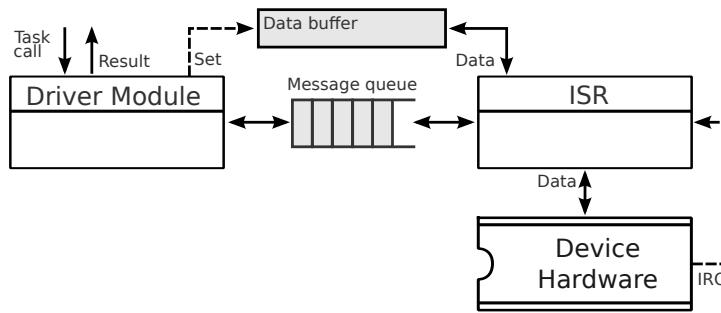


Figura 3.6: Arquitectura de un controlador de entrada serial asíncrono.

Las operaciones que corresponden al módulo del controlador son:

- Inicializar un *buffer* de datos.
- Si se implementa como driver asíncrono, entonces se espera por la llegada de un nuevo mensaje.
- Extraer los datos desde el *buffer*
- Retornar los datos hacia la tarea

Corresponden a la rutina de atención de interrupciones las siguientes operaciones:

- Atender la interrupción y restablecer el estado del dispositivo.
- Obtener un nuevo dato desde el dispositivo.
- Agregar un nuevo dato al *buffer*.
- Si se detecta el final de la operación, señalizar o agregar un mensaje a la cola.

3.3.3. Sistema operativo

El sistema operativo es la capa de abstracción entre la aplicación y el *hardware* en la arquitectura de un sistema embebido. Permite diseñar la aplicación sin considerar las par-

ticularidades de la plataforma en que se ejecuta, así como estructurarlo en una serie de procesos o tareas cuya gestión la proveen los servicios del sistema operativo los que, como se ha detallado en la sección 2.2, son los siguientes:

- Gestión de tareas.
- Sincronización entre tareas.
- Temporización.
- Gestión de memoria.
- Gestión de dispositivos de entrada y salida.

El diseño de un sistema operativo está fuera del alcance del proyecto, en cuanto existen una serie de alternativas ya disponibles para su uso. A continuación se analiza la arquitectura y servicios que proveen algunos de los sistemas operativos para sistemas embebidos recomendados por el fabricante del microcontrolador utilizado.

Tabla 3.3: Comparación de sistemas operativos para sistemas embebidos

Sistema Operativo	Kernel	Gestión de tareas	Gestión de memoria	Sincronización
FreeRTOS	Micro kernel	Preemptive o cooperativo, ambos con prioridades.	Fija, best fit o dinámica	Semáforos, semáforos binarios, mutex, colas.
Salvo	Micro kernel	Cooperativo con prioridades	No tiene	Semáforos, semáforos binarios, mensajes, colas de mensajes.
AVIX	Monolítico	Preemptive con prioridades	Dinámica	Semáforos, mutex, grupos de eventos, pipes, mensajes.
uC/OS-II	Micro kernel	Preemptive con prioridades	Por bloques fijos	Semáforos, eventos, mutex, colas.
Q-Kernel	Monolítico	Preemptive o cooperativo, ambos con prioridades	Dinámica	Semáforos, mutex, eventos, mensajes, pipes.
RoweBots DSPnano	Micro kernel	Preemptive con prioridades	Dinámica	Semáforos, mutex, variables condicionales, barreras, eventos.
embOS	Micro kernel	Preemptive con prioridades	No tiene	Semáforos, mensajes

En particular, nos interesan dos de ellos: Salvo RTOS, por ser parte del kit adquirido y es el recomendado por el fabricante del computador a bordo; y FreeRTOS por ser una solución ampliamente utilizada comercialmente en sistemas embebidos y además posee una licencia libre y es gratuito.

3.3.4. Aplicación

Para el diseño de la aplicación, se ha estudiado la adaptación de un patrón utilizado en programación orientada a objetos, llamado *command pattern*. Como se detalla en la sección 2.3.3 este soluciona la necesidad de procesar las peticiones de diferentes objetos sin necesidad

de saber, en particular, cómo se ejecutarán estas acciones. De este modo la petición es encapsulada como un comando que se entrega al objeto adecuado para ser ejecutado. Este patrón entrega algunas ventajas como: separar los módulos que generan los comandos de aquellos que los ejecutan; la posibilidad de gestionar colas de comandos, registros u deshacer acciones; y ofrecer un flujo uniforme para ejecutar las acciones, permitiendo extender el *software* al agregar comandos nuevos.

Como las restricciones que se han impuesto al diseño de la aplicación indican que no se utilizará un lenguaje de programación orientado a objetos, sino que uno procedural, se realizará una adaptación de la idea de este patrón, utilizando los recursos y servicios que proveen las capas inferiores de la arquitectura. En el esquema de la figura 2.10, se observan dos tipos de módulos: activos y no activos.

Entre los módulos activos encontramos a los controladores o *listeners*, y el procesador de comandos. Además, siguiendo la convención de E. Gamma [?], se requiere un receptor o *executer*. La característica común de estos, es que son procesos que siempre se están ejecutando en la aplicación. Realizan tareas específicas, toman decisiones en base a las reglas programadas y forman la ruta crítica por donde fluye la información dentro del *software*. Para el sistema operativo, estos módulos se representan como tareas que tendrán su espacio de tiempo en el procesador, su espacio de memoria independiente y se comunican entre sí a través de las estructuras de sincronización disponibles.

Los módulos no activos son aquellos que sólo prestan servicios a las tareas que están funcionando dentro del sistema operativo, como librerías, estructuras de datos, controladores y repositorios de datos. Entre ellos se encuentran: la estructura de comandos genérica o *abstract command*, los comandos, el proveedor de servicios o *supplier*, además de otras librerías y repositorios presentes. No se justifica considerar estos elementos de *software* como tareas debido a que pasarían la mayor parte del tiempo sin realizar acciones, salvo cuando son requeridos por módulos activos. Si estuvieran al mismo nivel que estos, se debería agregar una completa batería de sincronización, que permita activarlos y desactivarlos, lo cual a la larga se hace complejo de mantener. La forma correcta de visualizarlos es como estructuras de datos, repositorios y librerías, implementados según las posibilidades de la capa de sistema operativo, *drivers* o características propias del lenguaje de programación.

A continuación, se describen en detalle los módulos que conforman la arquitectura del *software* en la capa de aplicación:

Listeners.

Los *listeners* o escuchadores son los módulos en la capa superior de la arquitectura, y emulan lo que serían los clientes o controladores dentro de *command pattern* (ver figura 2.3.3). Son los únicos encargados de generar los comandos en la aplicación. Pueden existir varios *listeners*, dado que implementan la inteligencia del sistema para generar las acciones deseadas ante diferentes circunstancias. Su denominación proviene de la siguiente filosofía adoptada en el diseño: la única justificación para que un *listener* exista es que se mantenga “escuchando” alguna variable del sistema o el estado de un subsistema, para tomar decisiones sobre los

comandos que se deben ejecutar en cada momento. Se pueden ver como las “aplicaciones” o “procesos” de otras arquitecturas de *software* para pequeños satélites [?][?], en cuanto acá se realizan procesos de manera periódica y se mantienen activos durante todo el funcionamiento del sistema. Estos módulos permiten extender las funcionalidades del satélite, en cuanto se requiera una aplicación que, dado ciertos parámetros, tome decisiones en tiempo real y ejecute las acciones necesarias, por ejemplo, conceptualmente se pueden considerar como un *listener* los siguientes procesos:

- Dado un temporizador, realizar de manera periódica una revisión del estado del sistema.
- Dada la posición actual en la órbita, ejecutar un plan de vuelo.
- Dado que llegan telecomandos desde la estación terrena, procesarlos y ejecutar las solicitudes.
- Dado que se reciben datos por el puerto serial, procesar y ejecutar las acciones solicitadas.

Se hace hincapié en que la existencia de cada *listener* requiere de un subsistema o variable al cual prestar atención, ya sea un temporizador, la posición actual o el arribo de un telemando, por mencionar algunos ejemplos. Una vez definido esto, el programador debe determinar bajo qué condiciones se toma la decisión de generar un comando que realice las acciones requeridas.

Los *listeners* no realizarán acciones que involucren directamente el acceso a bajo nivel hacia otros subsistemas, evitando así el uso simultáneo de recursos de *hardware* compartido -como módulos de comunicaciones- que puedan causar un estado de *data race*. El patrón de diseño indica que en este nivel tampoco se llevan a cabo, directamente, las acciones de los comandos, sino que son generados y encolados para su posterior ejecución. Esto plantea la limitante de que quien envía el comando, no puede saber si efectivamente fue ejecutado o cuál fue el resultado. No obstante se puede lograr cerrar el lazo de control a través de la lectura del repositorio de estados y mediante la modificación de una variable en este repositorio por parte del comando generado. Por lo tanto se permite el acceso de solo lectura a los repositorios de estados, datos y de comandos para obtener la realimentación necesaria de las acciones que han sido requeridas.

Dispatcher.

El siguiente nivel en la arquitectura es el módulo *dispatcher*. Dentro del patrón de diseño original tiene su símil con el objeto denominado *invoker*, en cuanto es el encargado de pedir la ejecución de un comando generado por un *listener*. Todos los comandos que son generados por los múltiples *listeners* llegan a este módulo y son agregados a una cola. Aquí se realiza un control sobre ellos y se pueden establecer políticas de rechazo a su ejecución. Si el comando es aceptado el, *dispatcher* encargará su ejecución al siguiente nivel de la arquitectura. Entre las responsabilidades que pueden ser asignadas se encuentran:

- Recibir todos los comandos generados y decidir si serán enviados para su ejecución.
- Ordenar la ejecución de comandos según prioridades.

- Filtrar comandos según el estado de salud del sistema. Por ejemplo, evitar la ejecución de aquellos que usan mucha energía cuando el nivel de carga de las baterías sea crítico.
- Filtrar los comandos provenientes o hacia un determinado sub-sistema que pueda estar causando fallas.
- Llevar un registro de los comandos que se han generado.
- Llevar un registro del resultado de la ejecución de comandos.

Sólo una instancia de este módulo existe en el sistema, permitiendo centrar las estrategias de control de las operaciones que se realizan, sin afectar el funcionamiento de otras áreas. La información que dispone el *dispatcher* para establecer el control son dos: el estado del sistema obtenido desde el repositorio correspondiente y la meta información disponible en los comandos que son encolados.

Entre las ventajas que presenta este esquema, se encuentra la capacidad de encapsular el proceso de auditoría de las operaciones que realiza el software, así como la forma de ejecutar los comandos de cara a los clientes o *listeners* que solicitan estas acciones. Este es el punto adecuado para implementar las funcionalidades de deshacer acciones, llevar un registro de sucesos, postergar la ejecución de comandos, entre otras.

La principal desventaja es que por ser el único módulo que recoge las acciones solicitadas por todos los *listeners*, representa el principal cuello de botella, retrasando la ejecución de comandos a un punto en que no se logra cumplir con los requisitos de tiempo del resto de las tareas del software de vuelo. Por ello, se debe mantener acotada la cantidad de acciones que se realizan en el *dispatcher* y establecer un tamaño adecuado a la cola de llegada para no bloquear el resto de las tareas.

Executer.

Corresponde al módulo final en el flujo de comandos del sistema, donde estos son ejecutados. Equivale al objeto *receiver* dentro del patrón de diseño original, en cuanto es quien finalmente realiza las acciones solicitadas. El *executer* recibe un comando desde el *dispatcher* y obtiene la función que se debe ejecutar desde el repositorio de comandos. Luego se ejecuta la función, que realiza todas las acciones implementadas dicho comando, tales como: leer datos, acceder a dispositivos, cálculos y almacenamiento resultados. Al término su código de retorno es notificado al *dispatcher*, indicando la disponibilidad del *executer* para una nueva operación.

Sus responsabilidades incluyen recibir el comando, identificar y obtener la función a ejecutar, los parámetros, realizar el llamado a la función, recibir el código de retorno y notificar a al *dispatcher* el resultado. En cuánto a la cantidad de *executers* que pueden existir en el sistema se pueden tomar dos aproximaciones:

- **Executer único:** Tener un sólo *executer* que funciona con máxima prioridad respecto al resto de las tareas. Esto permite brindar acceso exclusivo del sistema al comando en ejecución. Así se ahorran los problemas que surgen de sincronizar el uso compartido de recursos o sub-sistemas. Sin embargo, se debe cuidar que el comando en ejecución no

cause una falla que deje al sistema congelado.

- **Múltiples executers:** Se puede implementar un patrón *thread pool* para permitir la ejecución de varios comandos de manera concurrente. Esto implica tener varios *executers* esperando a recibir ordenes desde una cola. Cuando uno de ellos esté disponible toma un comando y lo ejecuta, encolando también su resultado. Se puede obtener un sistema que funcione de manera más fluida cuando se cuenta con una alta demanda de comandos, sin embargo, se requiere una cuidada sincronización de todos los recursos compartidos del sistema para evitar situaciones de *data race*.

Repositorios.

Los módulos en ejecución requieren del acceso a los datos básicos del sistema, que indican el estado de funcionamiento y permiten tomar decisiones según determinadas variables de control; del mismo modo los comandos ejecutados requieren informar de cambios en el estado de funcionamiento, reconfiguración de parámetros y el almacenamiento de datos provenientes de experimentos. Ambas necesidades de acceder a la información disponible en el sistema se abstrae en el concepto de repositorios de datos que son módulos encargados de organizar toda la memoria disponible en el sistema y proveer métodos de acceso que transparenten la lectura y escritura de datos. Los repositorios de datos por lo general no son módulos activos en su ejecución, es decir, se tratan como librerías que proveen funciones para manejar el acceso a los datos, organizar el lugar físico en que se conservará la información y supervisar la integridad de éstos, sobre todo ante casos de falla o reinicio del sistema.

Arquitectura de la aplicación

En definitiva la arquitectura del *software* en la capa de aplicación queda descrita por el diagrama de la figura 3.7 que muestra el flujo de información entre los diferentes módulos que lo componen, así como sus dependencias, principales operaciones y su correspondencia con el patrón de diseño que inspira la arquitectura propuesta.

3.4. Diseño de la solución

Dado los requerimientos operacionales y no operacionales, así como la arquitectura de *software* en todos sus niveles, se pasa a validar el diseño propuesto para poder asegurar que el *software* requerido y sus funcionalidades son posibles de implementar con esta solución. Para esto se vuelven a revisar los requerimientos operacionales y se propone una solución conceptual a través de la arquitectura de *software* propuesta. Los resultados de este proceso para cada área se detallan en las tablas 3.4, 3.5, 3.4 y 3.6.

El resultado de este ejercicio revela los módulos que son necesarios en el diseño de la

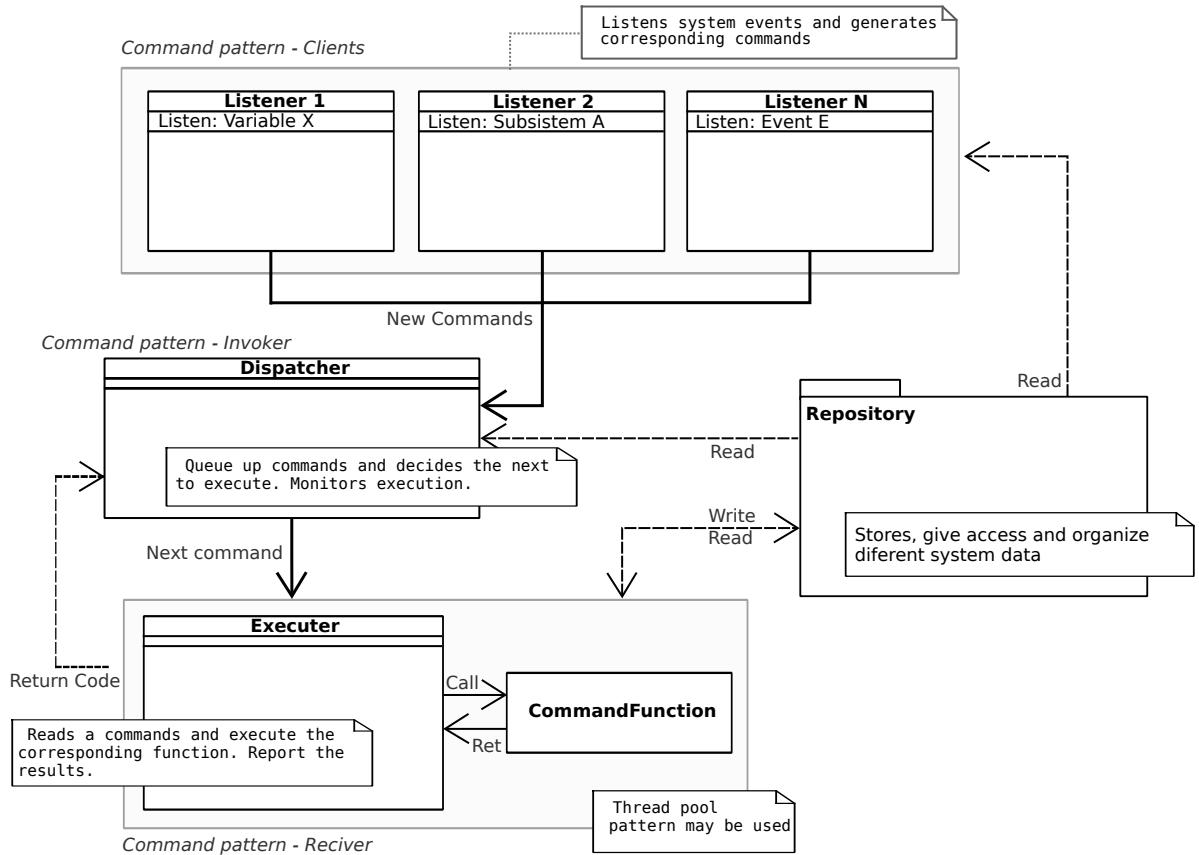


Figura 3.7: Arquitectura de *software* para el control del satélite.

arquitectura de *software* a nivel de aplicación, específicamente los *listeners* que se deben

Tabla 3.4: Análisis de la arquitectura, según requerimientos operacionales, para el área de comunicaciones

Área de comunicaciones		
Función	Módulo	Implementación
Configuración inicial del transceiver	Listener (Deployment)	Al inicio del sistema se ejecutan comandos que configuran los parámetros adecuados. El sistema se duerme en su totalidad para respetar el tiempo de silencio radial.
Procesamiento de telecomandos	Listener (Communications)	Se consulta periódicamente el estado del <i>transceiver</i> y cuando llega un telemando se decodifica para generar los comandos del sistema correspondientes
Protocolo de enlace	Listener (Communications)	Cuando se recibe un telemando o señal que inicia la sesión de comunicaciones con tierra se generan comandos que responden según el protocolo y se lleva a cabo la subida y descarga de datos.
Envío de telemetría	NA	Se reciben telecomandos para envío de telemetría bajo demanda. Se genera el comando adecuado que lee el repositorio de datos y envía la información al subsistema de comunicaciones.
Despliegue de antenas	Sistema de inicio	Durante el inicio se activa el sistema de despliegue. Se revisa el estado del sensor externo (switch) y se reintenta hasta conseguir el despliegue.

Tabla 3.5: Análisis de la arquitectura, según requerimientos operacionales, para el área de control central

Área de control central		
Función	Módulo	Implementación
Organizar Telemetría	Repositorio de datos	Se cuenta con un repositorio de datos que brinda acceso de forma ordenada a los diferentes tipos de datos. Existen comandos específicos que recogen la información y usan el repositorio de datos para guardarla.
Plan de vuelo	Listener (FlightPlan)	El plan de vuelo consta de un itinerario con los comandos a ejecutar según la ubicación del satélite. Se puede configurar a través de comandos que modifiquen entradas específicas del itinerario.
Recoger información del estado del sistema	Listener (HouseKeeping)	De forma periódica se ejecutan comandos que revisan parámetros del sistema y actualizan variables en el repositorio de estados
Tolerancia a fallos de software	Dispatcher	Se lleva un registro de los comandos ejecutados y sus códigos de retorno. Se puede evitar la ejecución de comandos (o grupos de comandos) que están generando problemas en el sistema. (Lista negra). Se puede filtrar comandos desde ciertos <i>listeners</i> .
Capacidad de debug	Listener (DebugConsole)	Se cuenta con una consola serial capaz de interpretar órdenes como comandos internos del sistema. Comandos se pueden ejecutar en modo <i>debug</i> para tener una salida con información relevante sobre la ejecución.
Inicialización del sistema	Listener (Deployment)	Inicialmente sólo existe un <i>listener</i> , que genera los comandos para toda la secuencia de inicio, que implica configurar parámetros, repositorios, subsistemas, silencio radial, despliegue de antenas y la activación del resto de los <i>listeners</i> .

Tabla 3.6: Análisis de la arquitectura, según requerimientos operacionales, para el área de tolerancia a fallos

Área de tolerancia a fallos		
Función	Módulo	Implementación
Estado de salud del sistema	Listener (HouseKeeping)	De forma periódica se generan comandos que revisen el estado del sistema y actualicen el repositorio de estados. Se pueden ejecutar comandos de re-configuración, pasar a modos de fallo, desactivar módulos o sub-sistemas.
Mucho tiempo sin conexión con tierra	Listener (Communications)	Si no se establece comunicación con tierra luego de N días el sistema pasa a modo de fallo de comunicaciones, permitiendo descargar telemetría básica de manera automática.
Problemas con despliegue de antenas	Listener (Deployment)	Se chequean sensores que indican si las antenas se han desplegado. Se generan comandos que intenten desplegar las antenas.
Watchdog	Sistema Operativo	Se cuenta con un <i>watchdog</i> en el microcontrolador cuyo contador se reinicia periódicamente. Reinicia el sistema si la aplicación no responde.
Fallos de hardware externo	NA	Se pueden generar comandos para que la EPS apague los buses de energía de los <i>payloads</i> y <i>hardware</i> externo
Watchdog Externo	Listener (HouseKeeping)	Generar comandos periódicamente que reinicien el <i>watchdog</i> externo, reiniciando el sistema ante fallas generales en el microcontrolador.

Tabla 3.7: Análisis de la arquitectura, según requerimientos operacionales, para el área de energía órbita y payloads

Área de energía, órbita y payloads		
Función	Módulo	Implementación
Estimación de la carga de la batería	Listener (HouseKeeping)	De forma periódica se ejecuta comando que lee datos desde EPS y actualiza variables en el repositorio de estado del sistema.
PowerBudget	Dispatcher	Antes de ejecutar un comando se chequea el estado de energía del sistema. Los comandos tienen niveles de energía aceptables y sólo se ejecuta si su nivel requerido es menor o igual al estado de carga actual.
Actualizar parámetros de órbita	NA	(1) La órbita se calcula en tierra y se actualiza el itinerario según las predicciones de órbita. (2) Se cuenta con un GPS y se ejecuta el itinerario según localización.
Ejecución de comandos de payloads	Listener (FlightPlan)	Comandos se generan según el itinerario del plan de vuelo

implementar son:

- **Communications:** Este *listener* se encarga de controlar los eventos relacionados con el subsistema de comunicaciones, específicamente presta atención a la llegada de telecomandos que implican la ejecución de comandos en el sistema.
- **FlightPlan:** Este *listener* tiene a cargo el control del plan de vuelo del satélite, concebido como un itinerario de comandos a ejecutar en un determinado momento. Se presta atención al reloj del sistema o bien a la información entregada por un subsistema de posicionamiento, dependiendo de la implementación.
- **HouseKeeping:** Este *listener* tiene por función la ejecución de comandos relacionados con el control del estado del sistema mismo. Son acciones que se ejecutan periódicamente durante todo el funcionamiento del sistema, a diferentes intervalos según se requiera, por lo tanto la variable de interés para este *listener* es el conteo de *ticks* interno del sistema operativo.
- **DebugConsole:** Este *listener* tiene por función atender las órdenes entregadas a través de la consola serial y generar una salida de depuración con información útil sobre su ejecución. Si bien este módulo no tiene utilidad durante la misión, es de vital importancia para la etapa de desarrollo y previo al lanzamiento del satélite.

El módulo *dispatcher* contará con las siguientes características, que permiten cumplir con los requerimientos de tolerancia a fallos y control del estado del sistema:

- Recibir todos los comandos generados y decidir si serán enviados para su ejecución.
- Filtrar comandos que requieren mayor energía que la disponible en determinado momento.
- Llevar un registro de los comandos que se han generado
- Llevar un registro del resultado de la ejecución de comandos.

Respecto al módulo *executer*, en el diseño actual se considera la utilización de sólo un *executer* dado que el sistema no será utilizado bajo una alta demanda de ejecución de co-

mandos según lo expresado en los requerimientos no operacionales y aprovechando la ventaja que implica no requerir una gran cantidad de elementos de sincronización entre procesos concurrentes.

La arquitectura se completa con los repositorios de datos, que según lo analizado deben ser tres:

- **Repositorio de estados:** Provee acceso a todas las variables de estado del sistema, por ejemplo, información sobre el funcionamiento, estado de salud y parámetros de configuración actuales. La información en este repositorio suele estar presente en forma de *flags*, contadores o registros de configuración. Dependiendo de la aplicación, algunos de estos datos pueden requerir almacenamiento persistente de modo de mantener el estado de funcionamiento entre reinicios.
- **Repositorio de datos:** Provee funcionalidades para almacenar y recuperar datos generales, como resultados de experimentos, registro de sucesos o telemetría general. Por lo general se requerirá de almacenamiento persistente y de gran capacidad.
- **Repositorio de comandos:** Este repositorio provee el acceso a todos los comandos disponibles en el sistema. Es usado tanto para construir el comando que se desea generar por parte de los *listeners*, así como para determinar la función asociada al código que es recibido por el *executer*.

Con esto, el resultado del diseño de la arquitectura de *software* queda detallado en la figura 3.8 que muestra los módulos participantes y el flujo de información en la aplicación:

Las capas inferiores en la arquitectura global no se ven afectados por los requerimientos operacionales, en cuanto su diseño e implementación es menos flexible y siempre necesaria para sostener la arquitectura de nivel de aplicación. En la capa de *drivers* lo importante es contar con las librerías de acceso a todos los dispositivos requeridos. La capa de sistema operativo actúa como caja negra en cuanto se utiliza una solución de terceros lo cual es ventajoso dado que es un apartado crítico y de altamente complejo y sus detalles escapan al alcance de este proyecto.

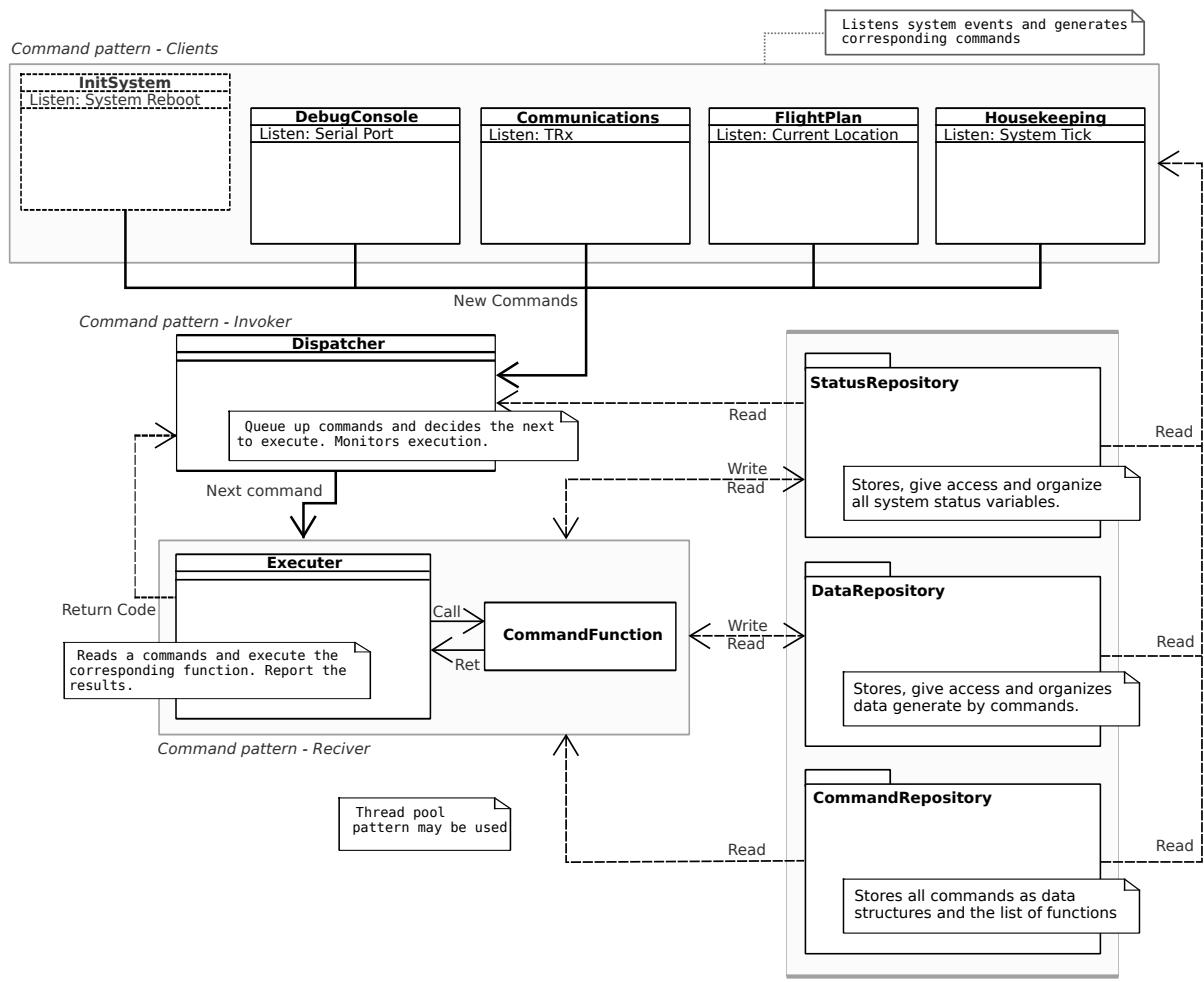


Figura 3.8: Arquitectura de *software* para el control del satélite.

Capítulo 4

Implementación

En este capítulo se describe el proceso de implementación la arquitectura para el *software* de vuelo de un nano-satélite diseñada en el capítulo 3. El proceso de implementación se concentrará en completar los requerimientos mínimos del sistema para alcanzar cierta generalidad que convierta al presente trabajo en la base de futuras misiones satelitales. En este sentido se evitará la discusión de detalles de implementación específicos de la misión del proyecto SUCHAI, para lo cual se puede tomar como referencia el trabajo desarrollado en torno al diseño e integración del proyecto[?]. En la misma línea, al presentar en primer lugar la implementación del núcleo de la aplicación como base de la característica de reusabilidad, se demuestra la capacidad de modificabilidad del sistema al agregar de manera incremental nuevas funcionalidades que completen los requerimientos esperados.

El proceso de implementación consta de tres etapas principales, que se alinean con la visión global del *software* en forma de arquitectura de tres capas. En primer lugar se detallan los *drivers* que se requieren implementar para esta plataforma de *hardware* específica, definiendo el tipo de arquitectura utilizada para guiar la implementación. En segundo lugar se detalla la forma de integrar el sistema operativo para luego montar la aplicación final la que se implementará módulo a módulo hasta lograr un sistema funcional.

4.1. Ambiente de desarrollo

La implementación del proyecto comienza por la definición del entorno y herramientas de desarrollo disponibles, pues son elementos esenciales que definen las posibilidades, limitaciones y marco de trabajo durante todo el proceso. El ambiente de desarrollo incluye los siguientes elementos: computadores, sistema operativo, control de versiones, el ambiente de desarrollo integrado o IDE, compilador, programador del microcontrolador y tarjeta de desarrollo.

Computadores. En general para el desarrollo de *software* no se tienen requerimientos de *hardware* elevados, considerando que bastaría con ejecutar un procesador de texto, una

aplicación de linea de comandos para compilar y la disponibilidad de al menos un puerto USB para utilizar el programador del microcontrolador. La aplicación más demandante en recursos es el IDE cuyos requerimientos recomendados de *hardware* se encuentran en la tabla 4.1.

Tabla 4.1: Requerimientos de *hardware* recomendados para desarrollo[?].

Procesador	Intel Pentium 4 @ 2.6 GHz o equivalente
Memoria RAM	2 GB
Espacio en disco	1 GB

Sistema Operativo. En principio no existen restricciones sobre el sistema operativo a utilizar dado que las principales herramientas como el IDE y el compilador son multiplataforma. Sin embargo el presente trabajo se ha desarrollado sobre plataformas GNU/Linux por su flexibilidad, libertad de distribución, disponibilidad de herramientas y estabilidad. Las principales distribuciones de Linux utilizadas fueron Kubuntu 12.04 LTS amd64 y LinuxMint 14 amd64.

Control de versiones. Un sistema adecuado de control de cambios es fundamental en el desarrollo de un proyecto de *software*, incluyendo el desarrollo de una aplicación para sistemas embebidos. Un sistema de control de versiones permite no sólo mantener un registro de los cambios incrementales del código, si no también revertir estos cambios, abrir ramas paralelas de desarrollo y el trabajo colaborativo entre un equipo de programadores. En este proyecto se utilizó el *software* Subversion con un servidor propio dedicado netamente a proveer el servicio de almacenamiento remoto del código junto al control de versiones. Subversion permite mantener un repositorio remoto y hacer copias de trabajo locales en el equipo de cada desarrollador (*checkout*). Los programadores realizan cambios sobre el código y suben las modificaciones al servidor como una nueva versión (*commit*), estos cambios se ven reflejados cuando el resto del equipo sincroniza sus copias de trabajo con el servidor remoto (*update*). Existe una amplia gama de *software* de control de versiones, si bien Subversion es una herramienta adecuada, existen herramientas mas avanzadas y flexibles como Git que posee como principal característica ser un sistema distribuido donde cada copia local actúa como un repositorio en si mismo haciéndolo más robusto. Si no se cuenta con servidores propios, se pueden utilizar servicio de almacenamiento de repositorios en línea, como GitHub, que se integra con Git y es gratuito para repositorios públicos.

IDE. El ambiente de desarrollo integrado o IDE es la aplicación fundamental del proceso, un buen ambiente de desarrollo proveerá las herramientas adecuadas para el desarrollo organizado y consistente del *software* integrando el editor de texto, servicio de control de versiones, integración con el compilador, integración con el programador, sistema de *debug*, sistema de documentación entre otros. En el caso de este proyecto se utiliza el entorno de desarrollo integrado de Microchip MPLAB X que se caracteriza por ser un entorno multiplataforma, basado en el proyecto de código libre NetBeans. Este IDE integra un avanzado editor de texto, con funcionalidades de control de cambios locales, múltiples configuraciones

para un mismo proyecto, integración con múltiples compiladores, acceso directo a la programación y depuración del dispositivo, todo desde la misma aplicación centralizando el proceso de desarrollo en un ambiente adecuado.

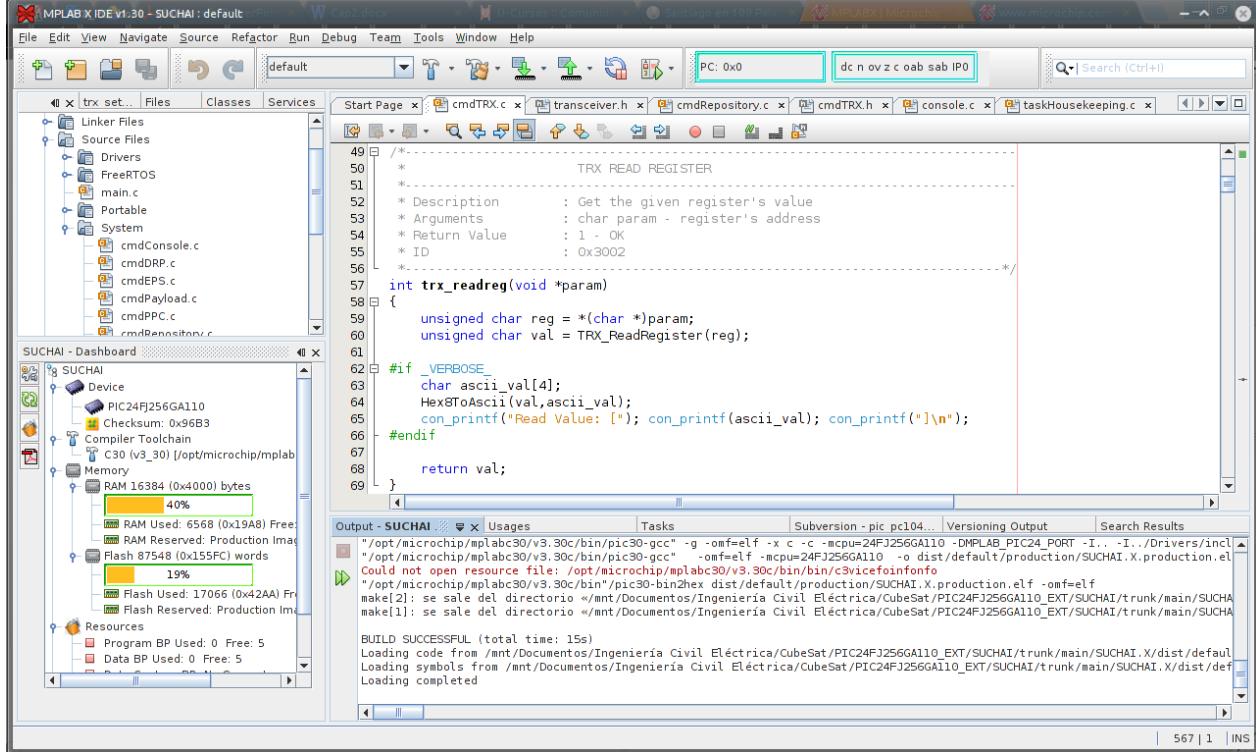


Figura 4.1: Entorno de desarrollo integrado MPLABX.

Compilador. El compilador es específico a cada microcontrolador para el cual se desea programar. En este caso corresponde al compilador Microchip XC16 en su versión 1.1, adecuado para la familia de microcontroladores PIC24.

Programador. El programador utilizado en este caso corresponde al Microchip ICD3, adecuado para ambientes de producción.

Tarjeta de desarrollo. Permite realizar las pruebas sobre el sistema embebido funcionando y es fundamental para el desarrollo de la aplicación del sistema embebido debido a que la aplicación final no se puede ejecutar en el mismo computador, sino que en la plataforma objetivo que corresponda. En este caso se utiliza la plataforma de desarrollo que provee el Cubesat Kit de Pumpkins[?]. Esta tarjeta permite montar un módulo de procesador con un PIC24F256GA110[?] y un bus PC104 al cual se conectan todos los componentes del satélite. Cuenta además con un *slot* de memoria SD, un reloj de tiempo real y un conversor RS232 a USB para fines de *debug* (ver figura 4.2). Es eléctricamente idéntica al la placa madre que se utilizará en el satélite por lo tanto es la herramienta adecuada para realizar todo el trabajo de programación y pruebas del *software*. Se debe hacer hincapié en lo fundamental de esta herramienta en el proceso de desarrollo de un sistema embebido debido a que: la aplicación

compilada es específica para el dispositivo objetivo; las herramientas de simulación de microcontroladores no son suficientes para probar las reales condiciones de ejecución del *software*; y porque el ciclo de desarrollo se completa con la solución y ajuste de problemas observados durante la ejecución de la aplicación en su sistema objetivo y de manera dinámica como resultado de respuestas a entradas no deterministas.

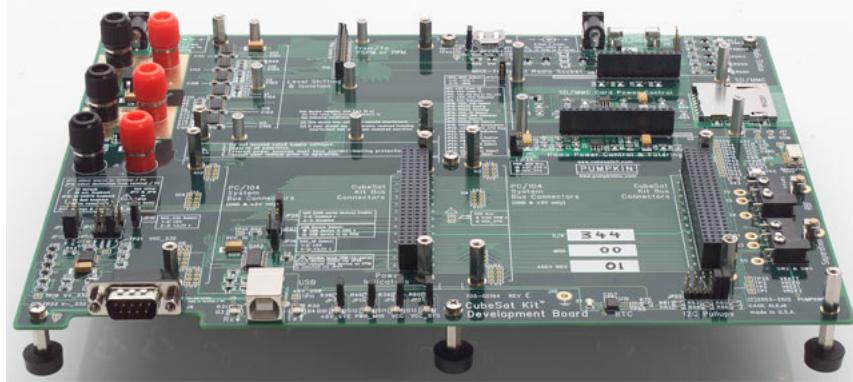


Figura 4.2: Tarjeta de desarrollo para Cubesat Kit de Pumpkins.

4.2. Organización del proyecto

4.2.1. Directorios

Con el objetivo de mantener un orden lógico a lo largo del desarrollo del *software* se debe dar una estructura lógica a los diferentes archivos fuentes que lo componen. Así se organiza un árbol de directorio que permite encontrar de manera sencilla cada archivo fuente según su función en el sistema. La organización de los directorios sigue la arquitectura de capas a nivel global de la aplicación quedando de la siguiente manera:

```

main/
├── Drivers/
│   └── include/
├── FreeRTOS/
├── Payloads/
└── Cmd/
    ├── include/
    └── Drivers/
        └── include/
SUCHAI.X/
System/
└── include/
main.c

```

Los desarrollos deben seguir esta estructura al momento de agregar archivo con código

fuente al sistema. En la tabla 4.2 se detalla la funcionalidad de cada directorio.

Tabla 4.2: Organización de directorios del proyecto

Directorio	Descripción
main	Directorio principal, contiene el archivo main.c y archivos de configuración globales
include	Dentro de cada directorio de fuentes, se agrega un directorio <i>include</i> que contiene las cabeceras de cada archivo fuente en el nivel superior.
Drivers	Contiene las fuentes para los <i>drivers</i> del sistema como el computador a bordo, el sistema de comunicaciones y el sistema de energía.
FreeRTOS	Carpeta con el nombre del sistema operativo. Contiene los archivos fuentes, cabeceras y librerías del sistema operativo según su organización particular.
Payloads	Comandos y <i>drivers</i> relacionados con <i>payloads</i> . Se encuentra en un directorio aparte pues acá se concentrarán la mayoría del <i>software</i> específico de la misión.
Payloads/Cmd	Implementación de comandos del sistema relacionados con <i>payloads</i> .
Payloads/Drivers	Implementación de <i>drivers</i> relacionados con <i>payloads</i> .
SUCHAI.X	Directorio con la configuración del proyecto generado por el IDE MPLABX.
System	Archivos con las fuentes del sistema base, incluye implementación de comandos, repositorios y tareas.

4.2.2. IDE

Esta estructura de directorios creada es la base para configurar adecuadamente los archivos con las fuentes del proyecto en el IDE, en este caso MPLAB X. Para la correcta construcción de *software* en el IDE se deben ajustar las configuraciones del compilador según las indicaciones de la tabla 4.3 donde los parámetros no mencionados mantienen su configuración por defecto. Ejemplos del dialogo para configurar las opciones del compilador en MPLAB X se detallan en la figura 4.3.

4.2.3. Documentación

Parte fundamental para cumplir con los requerimientos no operaciones es la correcta documentación del código desarrollado. Documentar tiene dos propósitos principales: por un lado guiar la implementación de la arquitectura para futuras modificaciones; y por otro documentar el objetivo, parámetros y uso de cada comando implementado.

La principal problemática consiste en mantener varios documentos a la vez separando el proceso de programación de la documentación. Para evitar este problema, la documentación se realiza en el mismo código y es en el mismo momento en que se programa una nueva

Tabla 4.3: Configuración del compilador XC16

XC16		
xc16-gcc		
Categoría: Memory Model		
Opción	Valor	Observaciones
Code model	Large	El tamaño de la aplicación supera el espacio de memoria cercano.
Data model	Large	El tamaño de la aplicación supera el espacio de memoria cercano.
Scalar model	Large	El tamaño de la aplicación supera el espacio de memoria cercano.
Location of constant model	Code space	
Categoría: Optimizations		
Opción	Valor	Observaciones
Optimization level	0	Las optimizaciones pueden introducir cambios en la forma de ejecución del código, por ejemplo, evitar ciclos <code>for</code> o <code>while</code> que realizan <i>busy waitings</i> .
Categoría: Preprocessing and messages		
Opción	Valor	Observaciones
C include dirs	<code>..; .../Drivers/include;</code> <code>.../System/include;</code> <code>.../<RTOS>/<include>;</code> <code>.../Payloads/Cmd/include;</code> <code>.../Payloads/Drivers/include</code>	Configura los directorios donde el IDE busca las cabeceras para poder incluirlas solo por su nombre y activar el auto completado.
Additional warnings	Seleccionada	Permite un nivel mayor de advertencias en tiempo de compilación

función, se crea un nueva definición o una nueva estructura de datos, en que se documenta el propósito del código.

Con el objetivo de extraer esta información desde el código a un formato más adecuado

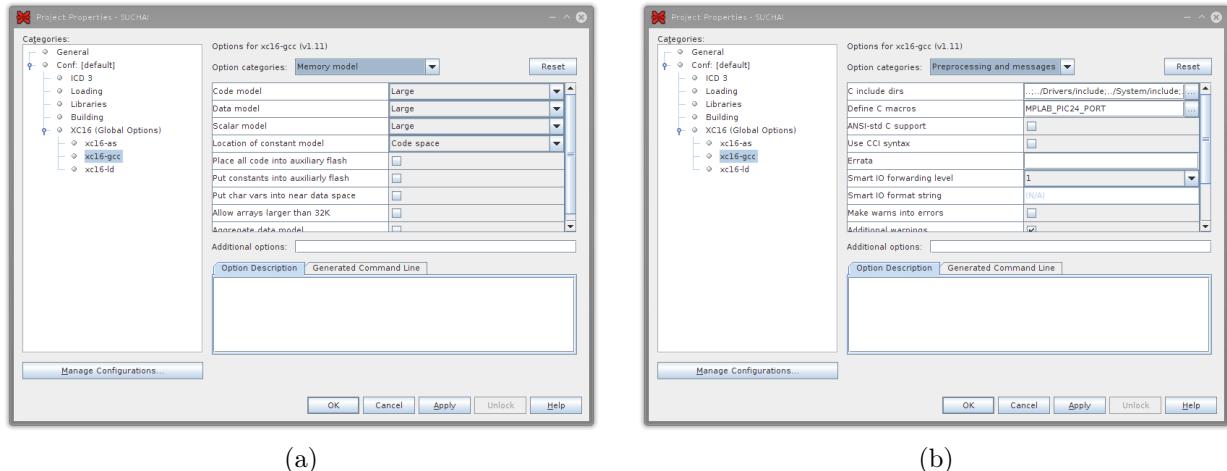


Figura 4.3: Diálogo de configuración del compilador XC16 en MPLABX

Listing 4.1: Prueba de Doxygen

```
1 /**
2  * Suma dos valores ingresados como parametros
3  * @note Solo acepta valores enteros sin signo
4  * @param a Primer elemento a sumar
5  * @param b Segundo elemento a sumar
6  * @return Suma de los parametros
7 */
8 int suma(unsigned int a, unsigned int b)
9 {
10     return a+b;
11 }
```

para su presentación y para establecer una convención en la forma de documentar se utiliza el *software* Doxygen. Al seguir un determinado formato en los comentarios agregados al código Doxygen es capaz de extraer la documentación y generar documentos bien acabados en diferentes formatos como: documentos HTML, L^AT_EX, RTF, entre otros.

El formato definido por Doxygen se detalla en su página web (www.doxygen.org) y consiste en comentarios más una serie de directivas que guían el formato de salida. Por ejemplo, para documentar una función en C con dos parámetros se utiliza la sintaxis del código 4.1 y al ejecutar Doxygen para generar la salida como página web se obtendría el resultado de la figura 4.4

Documentación de las funciones

The screenshot shows a generated HTML page for the 'suma' function. At the top, there is a code block showing the C function definition:

```
int suma ( unsigned int a,  
          unsigned int b  
        )
```

Below the code, the function's purpose is described:

Suma dos valores ingresados como parametros

Under the heading "Nota", there is a note:

Solo acepta valores enteros sin signo

Under the heading "Parámetros", there are two parameters listed:

- a Primer elemento a sumar
- b Segundo elemento a sumar

Under the heading "Devuelve", the return value is described:

Suma de los parametros

Figura 4.4: Documento HTML generado por Doxygen.

4.3. Controladores de hardware

La primera capa de la aplicación corresponde a una serie de módulos que implementan los controladores de *hardware*. En este caso un módulo significa una librería que consiste en

una serie de funciones escritas en lenguaje C las cuales acceden a funcionalidades específicas de cierto *hardware* respetando sus protocolos o API. La librería consta entonces de su correspondiente archivo fuente con extensión .c y un archivo de cabecera con extensión .h que contiene la declaración de las funciones.

La hoja de datos de cada dispositivo de *hardware* entrega la información necesaria para poder configurarlo y acceder a sus funciones. Se puede requerir de diferentes niveles de abstracción a la hora de implementar el controlador:

1. Programar en lenguaje ensamblador (*assembler*). Se maneja directamente el juego de instrucciones del procesador (también conocido como lenguaje máquina) para configurar sus registros, manejar periféricos o ejecutar un programa general. Requiere de un compilador de *assembler* para generar el ejecutable binario.
2. Utilizar un compilador en C. El compilador provee un nivel mayor de abstracción al permitir programar en un lenguaje de alto nivel y portable como C. Además provee librerías básicas para las funciones específicas de cada dispositivo.
3. Utilizar una librería externa. Un controlador de *hardware* puede requerir los servicios de otro controlador para su funcionamiento. Es el caso típico de dispositivos que utilizan algún protocolo de comunicación (RS232, SPI, I2C) y su controlador consiste en implementar una API de llamadas sobre este protocolo.

En este caso se hará un uso extensivo de las librerías escritas en C que provee el compilador XC16 para construir *drivers* más complejos a través de sus funciones base. Lo principal es implementar los *drivers* de cada periférico disponible en el microcontrolador pues serán los recursos que utilizarán los dispositivos externos que completan el sistema del satélite.

4.3.1. Microcontrolador

4.3.2. Periféricos

En la sección 2.1.1 se revisaron los periféricos disponibles en el PIC24F256GA110, de ellos al menos los siguientes drivers para soportar el resto de las funcionalidades de la aplicación: Timers, I2C, SPI y UART. En la tabla 4.4 se detalla la arquitectura que se utilizará en cada uno de ellos.

Tabla 4.4: Drivers para periféricos del microcontrolador

Periférico	Arquitectura
Timers	Síncrono
I2C	Síncrono
SPI	Síncrono
UART Escritura	Síncrono
UART Lectura	Cola de entrada de datos seriales

4.4. Sistema operativo

En la capa de sistema operativo se ha optado por utilizar FreeRTOS. Este sistema operativo está diseñado específicamente para sistemas embebidos y provee la capacidad de implementar tareas que son módulos de *software* que funcionan de manera concurrente y pueden compartir información a través de diferentes estructuras de sincronización. Soporta una gran variedad de microcontroladores a través de *ports* y aplicaciones demo que se obtienen al descargar el *software* desde la página web del proyecto (www.freertos.org). La estructura de software de este sistema operativo consta de cinco archivos fuentes en lenguaje C (sólo tres son necesarios para la utilidad básica), once archivos de cabecera y una capa portable dependiente del dispositivo sobre el cual se trabaja. Fuera de los demos y diferentes ports incluidos con la descarga el siguiente árbol de directorio se agrega a la carpeta del proyecto y en la configuración del proyecto en MPLABX:

```
FreeRTOS/Source/tasks.c  
FreeRTOS/Source/queue.c  
FreeRTOS/Source/list.c  
FreeRTOS/Source/portable/[compiler]/[architecture]/port.c  
FreeRTOS/Source/portable/[compiler]/[architecture]/portasm_[architecture].S  
FreeRTOS/Source/portable/MemMang/heap_2.c  
FreeRTOS/Source/include  
FreeRTOS/Source/portable/[compiler]/[architecture]
```

Se requieren algunas configuraciones extra en el compilador para el correcto funcionamiento de FreeRTOS, como se detalla en la tabla 4.5

Tabla 4.5: Configuración del compilador XC16 para FreeRTOS

XC16		
xc16-gcc		
Categoría: Optimizations		
Opción	Valor	Observaciones
Omit frame pointer	Seleccionada	Ver documentación de FreeRTOS[?].
Categoría: Preprocessing and messages		
Opción	Valor	Observaciones
Define C macros	MPLAB_PIC24_PORT	Ver documentación de FreeRTOS[?].
xc16-as		
Categoría: General Options		
Opción	Valor	Observaciones
ASM include dirs	.../FreeRTOS/Source/portable/ MPLAB/PIC24_dsPIC/	Funciones <i>assembler</i> específicas de la arquitectura.

El primer paso para integrar el sistema operativo es crear el archivo de cabecera con su configuración llamado `FreeRTOSConfig.h`. El archivo se compone de una serie de *defines* que cambian el comportamiento del sistema operativo y lo ajustan a las necesidades de la aplicación. Una plantilla con los posibles valores a configurar se encuentra en la página web de FreeRTOS: <http://www.freertos.org/a00110.html>. Un ejemplo de este archivo se detalla en el código 4.2

Listing 4.2: FreeRTOSConfig.h

```
1 #ifndef FREERTOS_CONFIG_H
2 #define FREERTOS_CONFIG_H
3
4 #include <xc16.h>
5
6 #define configUSE_PREEMPTION           1
7 #define configUSE_IDLE_HOOK            1
8 #define configUSE_TICK_HOOK             0
9 #define configTICK_RATE_HZ              250
10 #define configCPU_CLOCK_HZ             16000000
11 #define configMAX_PRIORITIES          4
12 #define configMINIMAL_STACK_SIZE       115
13 #define configTOTAL_HEAP_SIZE          5120
14 #define configMAX_TASK_NAME_LEN        8
15 #define configUSE_TRACE_FACILITY       0
16 #define configUSE_16_BIT TICKS          1
17 #define configIDLE_SHOULD_YIELD        1
18
19 #define INCLUDE_vTaskPrioritySet         1
20 #define INCLUDE_uxTaskPriorityGet        0
21 #define INCLUDE_vTaskDelete              0
22 #define INCLUDE_vTaskSuspend             1
23 #define INCLUDE_vTaskDelayUntil          1
24 #define INCLUDE_vTaskDelay               1
25
26 #define configKERNEL_INTERRUPT_PRIORITY 0x01
27
28 #endif /* FREERTOS_CONFIG_H */
```

Listing 4.3: taskTest.c

```
1 #include taskTest.h
2
3 void taskTest(void *param)
4 {
5     const unsigned long Delayms = 500 / portTICK_RATE_MS;
6     char *msg = (char *)param;
7
8     while(1)
9     {
10         vTaskDelay(Delayms);
11         printf("[taskTest] %s\n", msg);
12     }
13 }
```

En segundo lugar se deben crear las tareas que ejecutará el sistema. En FreeRTOS, una tarea es una función con una firma específica y que por lo general entrará en un ciclo infinito para mantener su ejecución en el tiempo. Toda tarea debe poseer la siguiente firma: `void taskName(void *)`, donde el nombre de la función varía de tarea en tarea, pero siempre debe retornar `void` y recibir un puntero `void` como parámetro. Un simple prototipo de tarea en FreeRTOS se detalla en el código 4.3. La tarea se denomina `taskTest` y recibe a través de su parámetro una cadena de texto. La tarea entra en un ciclo e imprime de manera periódica el *string* entregado como parámetro. Aunque la tarea está en un ciclo infinito, no se encuentra en una situación de *busy waiting* dado que utiliza la función `vTaskDelay` que detiene la ejecución de la tarea durante un periodo de tiempo liberando los recursos del procesador; en este caso se dice que la tarea se encuentra dormida.

Una vez programada la tarea, se puede configurar el sistema para que la ejecute. Esto se realiza en el archivo `main.c` donde se realizan las configuraciones del *hardware*, se cargan las configuraciones del sistema operativo contenida en el archivo `FreeRTOSConfig.h`, se crean las tareas y se inicia el sistema operativo. Para crear tareas se utiliza la función `xTaskCreate` indicando la función a ejecutar, un nombre, prioridad, memoria asignada y parámetros; para detalles referirse a la documentación de FreeRTOS[?]. El sistema operativo se inicia con la llamada a `vTaskStartScheduler()` y una vez alcanzado este punto, el control de los procesos a ejecutar queda en manos de FreeRTOS quien según su algoritmo de `scheduling` seleccionará la tarea que debe ejecutarse en cada instante. El llamado a la función `vTaskStartScheduler` no retorna a menos que se produzca un error en la ejecución del sistema operativo. El código 4.4 se ilustra la forma de iniciar FreeRTOS con dos tareas ejecutándose de manera simultanea, para mantener la generalidad no se incluyen configuraciones de *hardware* específicas.

El resultado de ejecutar el programa anterior se puede visualizar a través de la consola serial conectada al sistema objetivo:

```
>>>Starting FreeRTOS scheduler [->]
[taskTest] T2 Running...
[taskTest] T1 Running...
[taskTest] T2 Running...
```

Listing 4.4: main.c

```
1 /* RTOS Includes */
2 #include "FreeRTOSConfig.h"
3 #include "FreeRTOS.h"
4 #include "task.h"
5
6 /* Task includes */
7 #include "taskTest.h"
8
9 int main(void)
10 {
11     /* Creating all task */
12     xTaskCreate(taskTest, (signed char*)"taskTest",
13                 configMINIMAL_STACK_SIZE, (void *)"T1 Running...", 1, NULL);
14     xTaskCreate(taskTest, (signed char*)"taskTest",
15                 configMINIMAL_STACK_SIZE, (void *)"T2 Running...", 2, NULL);
16
17     /* Start the scheduler. Should never return */
18     printf(">>Starting FreeRTOS\n");
19     vTaskStartScheduler();
20
21 }
```

```
[taskTest] T1 Running...
[taskTest] T2 Running...
```

Se observa que la tarea de mayor prioridad es la primera en ejecutarse y por lo tanto imprime su mensaje en pantalla: [taskTest] T2 Running.... Esta tarea ahora pasa a estado suspendido y por lo tanto la siguiente tarea en orden de prioridad tiene acceso al procesador para ejecutarse, imprimir su mensaje: [taskTest] T1 Running... y pasar a estado suspendido. Cuando ninguna tarea está disponible para ejecutarse FreeRTOS ejecuta la tarea `Idle` iniciada por defecto por el sistema operativo. Luego de 500[ms] la tarea de mayor prioridad despierta y toma el control del procesador repitiendo el ciclo anterior.

Con esto concluye la integración del sistema operativo en el *software* de vuelo y se demuestra el correcto funcionamiento de FreeRTOS. Ahora se cuenta con una nivel mayor de abstracción en la aplicación donde el sistema operativo tiene el control sobre los procesos que se ejecutan en el microcontrolador y la aplicación final se implementa en las diferentes tareas que se crean.

4.5. Aplicación

En la capa de aplicación se debe implementar la arquitectura de *software* detallada en la figura 3.8 que permite el cumplimiento de todos los requerimientos de operacionales del

satélite. El proceso incluye interpretar el patrón de diseño que inspira la arquitectura base para ser implementado en un lenguaje *procedural* como C. Se parte por implementar la arquitectura base del patrón de ejecutor de comandos para contar con un sistema base que sea general y bien probado sobre el cual agregar las funcionalidades específicas del proyecto. Las funciones propias de este proyecto satelital estarán implementadas en la capa de *listeners* y la lista de comandos. Finalmente se detallará la implementación de los distintos módulos que competan los requerimientos del proyecto SUCHAI en específico.

4.5.1. Implementación del patrón de diseño

Cómo se describe en la sección 2.3.2 se debe implementar un patrón de diseño, específicamente *Command Pattern* en un lenguaje procedural como C. El problema radica en la programación basada en patrones de diseño es una técnica utilizada principalmente en lenguajes de programación orientados a objetos y los patrones se describen según diagramas de colaboración entre objetos incluyendo técnicas como herencia o polimorfismo. No obstante, el diseño de una arquitectura de *software* debe ser independiente del lenguaje de programación a utilizar, lo mismo se cumple en el diseño de una arquitectura basándose en patrones[?]. Para sortear esta situación se procede a definir cómo se traduce cada elemento del patrón de diseño deseado a la plataforma objetivo.

- **Controladores o clientes:** Corresponden a tareas del sistema operativo. Pueden ser tareas que se ejecuten de manera periódica o basadas en eventos. Los clientes estarán ejecutándose constantemente y según la inteligencia que tengan programada pueden responder a algún evento periódico o un estímulo externo para generar los comandos del sistema. Cada tarea posee su propio **stack** de memoria y se pueden comunicar con otras o utilizar recursos compartidos del sistema a través de estructuras de sincronización.
- **Procesador de comandos o despachador:** Corresponde a una tarea del sistema operativo cuyo funcionamiento está basado en un evento, en específico, el arribo de un comando.
- **Ejecutador:** o *executer* también es una tarea del sistema operativo que se encarga de realizar la llamada del comando, es decir, ejecutar la función. Su funcionamiento es basado en eventos según la llegada de un comando.
- **Transferencia de comandos:** La transferencia de comandos se realiza a través de una cola de FreeRTOS, que es una estructura de sincronización que permite el intercambios de mensajes en un esquema productor-consumidor.
- **Comandos:** Los comandos están representados por un nuevo tipo definido en C. Este tipo hace referencia a una función con una firma específica. Siempre que una función cumpla la firma específica en el tipo definido podrá ser considerada un comando y ser ejecutada como tal. Para encapsular parámetros importantes de un comando, que en el caso original serían variables de estado de un objeto, se crea una estructura en C.
- **Repositorios:** Los repositorios y proveedores de servicios se implementarán como bibliotecas que acceden a funciones a las capas inferiores del sistema como *drivers* a submódulos o dispositivos de entrada y salida.

Listing 4.5: Prototipo de un comando

```
1  /**
2   * Defines the prototype that every command must conform
3   */
4  typedef int (*cmdFunction)( void * );
```

Listing 4.6: Estructura de comandos para *executer*

```
1  /**
2   * Structure that represents a command passed to executer. Contains a
3   * pointer of type cmdFunction with the function to execute and one
4   * parameter for that function
5   */
6  typedef struct exec_command{
7      int param;                                ///< Command parameter
8      cmdFunction fnct;                      ///< Command function
9  }ExeCmd;
```

4.5.2. Comandos

Los comandos se han implementado como funciones que poseen una firma específica que se convierte en un nuevo tipo de dato. En este caso la función debe retornar un entero y recibir un parámetro, como se detalla en el código 4.5, lo que define dos aspectos fundamentales del *software*:

- Todo comando debe retornar un valor que puede tomar un significado dentro de la aplicación, en este caso, se define que el comando ha terminado su ejecución de manera insatisfactoria si retorna un valor cero o bien un valor no cero para indicar un éxito en la operación; con esa convención se hace natural utilizar el llamado a las funciones dentro de una sentencia condicional que utilice directamente el valor retornado como condición.
- Por otro lado los comandos reciben un sólo parámetro, en principio de cualquier tipo, a través de un puntero de tipo *void*. Esto tiene una serie de consecuencias: por un lado se hace flexible la llamada a la función ya que diferentes comandos podrían recibir diferentes tipos de datos siempre que se dereferencien adecuadamente; también hace eficiente la llamada a la función pues no se requiere realizar una copia del parámetro en la llamada ya que pasan por referencia; la desventaja es natural al uso de punteros, pues este debe seguir siendo válidos en el momento la ejecución del comando para evitar dereferenciar un puntero nulo o corromper algún sector de memoria.

Para encapsular la información asociada a los comandos que llegan hasta el módulo *executer* se crea una nueva estructura de datos definida como se muestra en el código 4.6. La estructura contiene dos campos: el puntero a la función que implementa el comando según la definición del código 4.5 y su parámetro.

Listing 4.7: Estructura de comandos para *dispatcher*

```

1 /**
2  * Structure that represent a command passed to dispatcher. Contains
3  * only a code that represent the function to call, a parameter and
4  * other command's metadata
5 */
6 typedef struct ctrl_command{
7     int cmdId;           ///< Command id, represent the desired command
8     int param;           ///< Command parameter
9     int idOrig;          ///< Metadata: Id of sender subsystem
10    int sysReq;          ///< Metadata: Level of energy the command requires
11 }DispCmd;

```

Con el objetivo de implementar una estrategia simple y flexible que permita tanto generar comandos así como determinar la función que se asocia a cada uno se ha definido una nueva estructura de datos que representa a los comandos que son generados por los *listeners* cuya definición se detalla en el código 4.7. Si se utiliza la estructura definida en el código 4.6 significa que cada *listener* debe conocer todos los comandos disponibles en el sistema y cada vez que se requiere generar uno se debe agregar de manera manual la función que corresponde en la estructura. Esto genera inconvenientes ya que al separar la definición de cada comando en diferentes archivos se debe realizar una serie de *includes*, hace poco automatizado la generación de comandos en serie y no es una forma práctica de generarlos de manera remota.

Por esta razón serán representados a través de un código numérico único, delegando al repositorio de comandos la responsabilidad de mapear entre los códigos y la función asociada. La estructura de datos que representa a los comandos que son enviados desde los *listeners* hacia el *dispatcher* contiene además ciertos meta datos que le permiten tomar decisiones sobre la ejecución de un comando basado en esta información extra. Dos campos de información son especialmente relevantes:

- **Origen del comando:** este campo contiene un código numérico que identifica el módulo que ha generado el comando, esto permite implementar el filtrado de comandos desde cierto módulo cuando, por ejemplo, no esté funcionando correctamente.
- **Requerimiento de energía:** este campo indica el nivel de energía que requiere el comando para ejecutarse, así se pueden filtrar aquellos que requieran más de la energía estimada en el sistema.

Todas estas definiciones son incluidas en un archivo de cabecera denominado `cmdIncludes.h` que homogeneiza el tratamiento de los comandos a lo largo de la aplicación.

La implementación de un comando en específico requiere la definición de una función que cumpla con la firma del código 4.5, un ejemplo es el código 4.8 donde se implementa el comando `get_rtos_memory`; notar que este comando forma parte del archivo `cmdOBc.c`, por lo tanto cada función y definición incluye el prefijo `obc_` como parte del estándar de nombres utilizado en el proyecto. Este comando que será utilizado como ejemplo es, sin embargo, una importante funcionalidad dentro del *software* de vuelo que nos permitirá obtener información sobre el uso de memoria del sistema operativo. El objetivo es utilizar la función

Listing 4.8: Ejemplo de comando

```
1  /**
2   * Performs debug tasks over current RTOS. Get rtos memory usage in bytes
3   *
4   * @param param Not used
5   * @return Available heap memory in bytes
6   */
7  int obc_get_rtos_memory(void *param)
8  {
9      size_t mem_heap = xPortGetFreeHeapSize();
10     printf("Free RTOS memory: %d", mem_heap);
11
12     return mem_heap;
13 }
```

`xPortGetFreeHeapSize` de FreeRTOS que entrega la cantidad de memoria en *bytes* disponible en el *heap[?]* para desplegarla en consola. La convención indica que retornar un valor cero desde el comando significa error lo cual demuestra su conveniencia en este caso pues el comando puede retornar directamente la cantidad de memoria disponible.

Además de la implementación de la función se debe generar un método de registro del comando para que esté disponible en el sistema y sea reconocido por el repositorio correspondiente. Este proceso es fundamental para permitir la modificabilidad del sistema de vuelo extendiéndolo a través de nuevos comandos, por lo cual se exploraron una serie de alternativas en pos de lograr el método más directo, transparente y de menor impacto en el código del proyecto:

1. Un arreglo único de comandos centralizado en el repositorio. Se llena manualmente un arreglo de punteros a funciones tipo *cmdFunction* y para registrar un nuevo comando se agrega al arreglo.
 - **Ventajas:** Se puede ahorrar memoria RAM creando un arreglo de tipo *const* que se almacena en memoria de programa.
 - **Desventajas:** Complejo seguir cambios en el orden en que se registran los comandos y por lo tanto su código asociado. Ofrece pocas posibilidades de agrupar comandos según funciones. Alto impacto al agregar un nuevo comando pues implica la modificación de múltiples archivos fuente, incluyendo el repositorio de comandos.
2. Varios arreglos con comandos centralizados en el repositorio. Se agrupan comandos con funcionalidades relacionadas en varios arreglos, así cuando se agrega uno nuevo sólo se modifica el arreglo relacionado con su grupo.
 - **V:** Mejor agrupación de los comandos, localizando cambios. Los códigos numéricos se pueden diferenciar por grupo.
 - **D:** Agregar un comando requiere modificar varios archivos, incluyendo el archivo donde se implementa y el repositorio de comandos.
3. Descentralizar cada arreglo de comandos en el archivo correspondiente. Cada archivo que implementa un grupo de comandos cuenta con un arreglo con las funciones. El

Listing 4.9: Lista de comandos disponibles

```
1  /**
2   * List of available commands
3   */
4  typedef enum{
5      obc_id_reset = 0x1000,    ///< @cmd_first
6      obc_id_get_rtos_memory, ///< @cmd
7
8      ppc_id_last_one      // Dummy element
9 }OBC_CmdIdx;
```

repositorio de comandos utiliza este arreglo como una variable externa.

- **V:** Mejor agrupación de comandos. Bajo impacto en el código al agregar uno nuevo, pues sólo se modifica el archivo dónde se encuentra sin intervenir el repositorio.
 - **D:** Se dificulta el seguimiento de cambios en los códigos que identifican cada comando. Agregar un nuevo grupo de comandos, en un nuevo archivo, requiere modificaciones en el repositorio.
4. Descentralizar comandos y utilizar **enums** para asignar códigos identificadores. Similar a la alternativa anterior pero el identificador de cada comando y la forma de llenar los arreglos se basan en una estructura de enumeración de C que abstrae el uso de códigos numéricos por sentencias textuales.
- **V:** Agregar un comando sólo requiere cambios locales al archivo que agrupa un determinado tipo. Se facilita el seguimiento de cambios en los comandos disponibles pues la definición textual de la enumeración no cambia.
 - **D:** Aún se requiere modificar el repositorio de comando cuando se agrega un archivo con un nuevo grupo de funciones. Al completar los arreglos de funciones a través de las enumeraciones no se puede crear un arreglo de manera manual en memoria de programa y se utiliza más memoria RAM, se requiere una función que inicialice los arreglos cada vez que se inicia el sistema.

Todas las estrategias mencionadas fueron implementadas en algún momento del desarrollo pero debido a las desventajas mencionadas se fueron desecharlo hasta derivar en la cuarta alternativa que ha demostrado ser la opción más flexible y conveniente. De este modo el proceso de registro del comando creado sólo requiere modificaciones en los archivos relacionados a su implementación, en este caso **cmdOBC.c** y **cmdOBC.h**.

Cada archivo con un grupo de comandos debe tener una estructura de enumeración (**enum** en C) que representan los códigos de cada comando. El primer valor de la enumeración debe ser diferente para cada grupo de comandos para no generar ambigüedades y para agruparlos también por códigos. El último valor de la enumeración es un valor *dummy* que sólo es utilizado para controlar el tamaño del arreglo. Luego, por cada función que implemente un comando se agrega un valor en la enumeración, como en el código 4.9.

En este caso se registran dos comandos, **obc_reset** y **obc_get_rtos_memory**, la convención de sintaxis utilizada indica que se debe utilizar el prefijo del grupo **obc_** y un prefijo que

Listing 4.10: Registro de comandos

```
1 cmdFunction obc_Function[OBC_NCMD];
2
3 /**
4  * This function registers the list of command in the system,
5  * initializing the functions array. This function must be called
6  * at every system start up.
7 */
8 void obc_onResetCmdOBC(void)
9 {
10     obc_Function[(unsigned char)obc_id_reset] = obc_reset;
11     obc_Function[(unsigned char)obc_id_get_rtos_memory] =
12         obc_get_rtos_memory;
13 }
```

indique que se trata de un código identificador `id_`. El código 4.9 también muestra como se utiliza Doxygen para mantener en la documentación del proyecto una lista actualizada con el valor de cada enumeración que permita de manera directa asociar el código de un comando cuando estos se generen de manera remota.

También en el archivo `cmdOBC.c` se debe crear el arreglo que contiene la lista de funciones, en este caso denominado `obc_Function` cuyo tamaño se determina a través del último elemento de la enumeración. Esta lista debe ser inicializada a través de una función que se ejecutará al inicio del sistema, en este caso se denomina `obc_onResetCmdOBC`. El código 4.10 detalla este proceso.

En la sección 4.5.3 se describe la implementación del repositorio de comandos y las funciones disponibles para acceder a los comandos registrados en el sistema.

4.5.3. Repositorio de comandos

El repositorio de comandos corresponde a una librería con funciones que brindan acceso a los comandos registrados en el sistema. Sus dos responsabilidades principales son:

- Inicializar el repositorio de comandos, es decir, inicializar los arreglos con los comandos disponibles.
- Mapear cada código de comando con su función asociada.

Inicializar el repositorio de comandos significa llamar a la función de inicialización presente en cada archivo donde se implementan las funciones, esto se realiza en la función `repo_onResetCmdRepo` disponible en el código 4.11.

Los códigos de los comandos se representan como un entero sin signo de 16 bit en formato hexadecimal, los 8 bits más significativos identifican el grupo al que pertenece el comando dejando los 8 bit restantes para la posición dentro de su arreglo, limitando la cantidad de comandos por grupo a un total de 256 como se detalla en la tabla 4.6

Tabla 4.6: Estándar para identificar comandos

Comando	Grupo	Numero
0xAABB	0xAA	0xBB

Por esta razón la función `repo_getCmd`, encargada de retornar la función asociada a un código de comando, divide este número para identificar el arreglo desde el cual obtenerlo y la posición dentro del arreglo en que se ubica el puntero a la función buscada. Esto se implementa en el código 4.11.

Por diseño, una vez inicializado, el repositorio tiene acceso de sólo lectura, de modo que no hace necesario la implementación de sincronización en su acceso. La única acción que puede generar una condición de *data race* es la lectura de un elemento del arreglo de comandos, sin embargo los arreglos son almacenados en memoria interna por lo que esta operación es atómica y no genera problemas de acceso concurrente.

4.5.4. Repositorios de estados

Este repositorio almacena aquellas variables que contienen información sobre el estado de operación del sistema satelital. Las variables de estado son consultadas por los *listeners* para tomar decisiones sobre los comandos que se generarán. También cumple la función de cerrar el lazo de control en el sistema, puesto que los *listeners* sólo generan el comando sin tener información sobre la ejecución o su resultado. El *dispatcher* utiliza esta información para comparar los requerimientos de cada comando con los recursos disponibles en el sistema y decidir sobre su ejecución. Los comandos tienen acceso de escritura y lectura sobre el repositorio de estados, pues dentro de lo que se espera de ellos es que ajusten variables de funcionamiento o cambien el modo de operación del satélite o sus subsistemas.

Las operaciones básicas que provee el repositorio son: leer una variable de estado, lo cual se realiza en la función `dat_getCubesatVar`; escribir el valor de una variable de estado, lo cual se realiza en la función `dat_setCubesatVar`; e inicializar el repositorio de estados ante un reinicio del sistema, a través de la función `dat_onResetCubesatVar`. Estas funciones se encuentran implementadas en el código 4.12.

Este repositorio es leído y escrito de manera concurrente por una serie de tareas por lo cual se puede generar una condición de *data race*. Esto se hace más evidente cuando se implementa en una memoria externa lo que puede resultar operaciones de lectura o escritura no atómicas, donde además se requiere utilizar recursos compartidos del sistema como periféricos de entrada y salida. Por esta razón se debe implementar una estructura de sincronización que provea la exclusión mutua entre las diferentes tareas que acceden a este repositorio. Esta situación se observa en el código 4.12 cuando se usan las funciones `xSemaphoreTake` y `xSemaphoreGive` de FreeRTOS[?].

Durante el desarrollo del proyecto se estudiaron varias alternativas de implementación del repositorio de estados en lo referente al lugar de almacenamiento, modo de acceso y sistemas de tolerancia a fallos. Entre los métodos explorados se encuentra:

Listing 4.11: cmdRepository.c

```
1  /* Add external cmd arrays */
2  extern cmdFunction obc_Function;
3
4  /**
5   * Returns a pointer with the function associated to each cmdID.
6   * @param cmdID Command id
7   * @return Pointer to function of type cmdFunction
8   */
9  cmdFunction repo_getCmd(int cmdID)
10 {
11     int cmdOwn, cmdNum;
12     cmdFunction result;
13
14     cmdNum = (unsigned char)cmdID;
15     cmdOwn = (unsigned char)(cmdID>>8);
16
17     switch (cmdOwn)
18     {
19         case CMD_OBC:
20             if (cmdNum >= OBC_NCMD)
21                 result=cmdNULL;
22             else
23                 result = obc_Function[cmdNum];
24             break;
25
26         default:
27             result = cmdNULL;
28             break;
29     }
30
31     return result;
32 }
33
34 /**
35  * Initializes all cmd arrays
36  * @return 1, always successful
37  */
38 int repo_onResetCmdRepo(void)
39 {
40     obc_onResetCmdOBC();
41     return 1;
42 }
43
44 /**
45  * Null command, just print to stdout
46  * @param param Not used
47  * @return 1, always successful
48  */
49 int cmdNULL(void *param)
50 {
51     int arg=*( (int *)param );
52     printf("cmdNULL was used with param %d\n", arg);
53     return 1;
54 }
```

Listing 4.12: dataRepository.c

```
1 #include "dataRepository.h"
2
3 extern xSemaphoreHandle dataRepositorySem; // Mutex for status repository
4 int DAT_CUBESAT_VAR_BUFF[dat_cubesatVar_last_one]; // Internal buffer
5
6 /**
7  * Sets a status variable's value
8  * @param idxVar Variable to set @sa DAT_CubesatVar
9  * @param value Value to set
10 */
11 void dat_setCubesatVar(DAT_CubesatVar idxVar, int value)
12 {
13     xSemaphoreTake(dataRepositorySem, portMAX_DELAY);
14     DAT_CUBESAT_VAR_BUFF[idxVar] = value;
15     xSemaphoreGive(dataRepositorySem);
16 }
17
18 /**
19  * Returns a status variable's value
20  * @param idxVar Variable to set @sa DAT_CubesatVar
21  * @return Variable value
22 */
23 int dat_getCubesatVar(DAT_CubesatVar idxVar)
24 {
25     int value = 0;
26     xSemaphoreTake(dataRepositorySem, portMAX_DELAY);
27     value = DAT_CUBESAT_VAR_BUFF[idxVar];
28     xSemaphoreGive(dataRepositorySem);
29     return value;
30 }
31
32 /**
33  * Initializes status repository
34 */
35 void dat_onResetCubesatVar(void)
36 {
37     int i;
38     for(i=0; i<dat_cubesatVar_last_one; i++)
39     {
40         dat_setCubesatVar(i,0xFFFF);
41     }
42 }
```

1. Almacenamiento interno en RAM. Un arreglo de enteros en memoria RAM donde la posición de cada variable se maneja a través de una estructura de enumeración.
 - **V:** Rápido acceso a los datos, puede no requerir sincronización si la lectura y escritura se implementan como operaciones atómicas. Siempre puede ser utilizado como método de respaldo.
 - **D:** Memoria volátil, los datos no se mantienen entre reinicios del sistema. No ofrece mecanismos de tolerancia a fallos.
2. Almacenamiento en memoria EEPROM externa. Se utiliza una memoria EEPROM a través de un bus I2C para guardar los datos de manera permanente, este tipo de dispositivos pueden usarse como memorias de acceso aleatorio no volátiles.
 - **V:** Memoria no volátil, los datos persisten entre reinicios del sistema. Acceso aleatorio a los datos.
 - **D:** Escritura y lectura no atómica, requiere sincronización. No ofrece mecanismos de tolerancia a fallos. Acceso a datos es más lento.
3. Almacenamiento redundante en dos memorias EEPROM, se utiliza una como respaldo en caso de falla, escribiendo siempre una copia de los datos en ambas. Cuando se detectan problemas en la operación de una memoria, el sistema activa la lectura desde el dispositivo de respaldo.
 - **V:** Almacenamiento no volátil, acceso aleatorio a los datos. Provee un mecanismo de protección de fallos.
 - **D:** Requiere sincronización de la lectura y escritura. Requiere método de detección de fallos. Acceso a datos es más lento.
4. Almacenamiento redundante en dos memorias EEPROM más memoria interna. La escritura y lectura de variables se realiza sobre una copia en memoria interna de los datos. De manera periódica las variables se respaldan en dos memorias EEPROM con sumas de verificación. Al inicio del sistema se cargan los datos desde la memoria que contenga la suma de verificación correcta.
 - **V:** Almacenamiento no volátil de los datos. Acceso aleatorio a los datos. Rápida escritura y lectura. Puede no requerir sincronización si se implementa con operaciones atómicas. Menor deterioro de las memorias EEPROM. Provee tolerancia a fallos y detección de fallos.
 - **D:** Dependiendo del periodo de respaldo, las copias pueden no contar con toda la información actualizada de las variables. El cálculo de sumas de verificación puede ser caro computacionalmente.

En general cualquier tipo de memoria no volátil se puede utilizar para estos fines, aunque si se utilizan los métodos 2 o 3, es conveniente utilizar una memoria de acceso aleatorio en lugar de dispositivos que leen o escriben por bloques ya que el uso de este repositorio es bastante intensivo dentro del diseño del sistema.

El primer método es el más simple en cuanto no requiere de controladores externos para su funcionamiento y la sincronización no es crítica. Por esto y para mantener la generalidad en el código del ejemplo 4.12 se implementa este diseño en el repositorio de estados. Sin embargo, el no contar con un almacenamiento persistente de los datos es una limitación crítica ya que este módulo permite que el sistema de vuelo sea tolerante a reinicio inesperados volviendo

a funcionar según sus últimos estados almacenados. Este método es factible de utilizar para pruebas o bien como un sistema de respaldo en caso de que una falla fuerce al sistema a trabajar con funcionalidades mínimas.

El diseño implementado en el sistema de vuelo utilizado en el proyecto SUCHAI es el tercero de la lista, y cuenta con dos memorias EEPROM de 512 bytes comunicadas a través de un bus I2C exclusivo. Esto provee la suficiente funcionalidad para que el sistema mantenga consistencia en su funcionamiento ante reinicios y además entrega redundancia como medida de tolerancia a fallos[?].

4.5.5. Dispatcher

El *dispatcher* o procesador de comandos es el módulo encargado de tomar un comando, procesar sus meta datos y entregar este comando el ejecutor. En este módulo se concentra toda la inteligencia asociada al control de la ejecución de un comando y es el punto adecuado para establecer un sistema de registro de los acciones que realiza el sistema.

Una parte fundamental de este módulo es la cola de comandos. Los comandos generados por cada *listener* llegan a esta cola en espera de ser procesado. Al ser un recurso compartido presenta el esquema típico del productor-consumidor con múltiples productores y un único consumidor, por lo que el acceso concurrente debe estar sincronizado (ver figura 4.5). La solución a este tipo de esquemas es bien conocido y se logra mediante la utilización de semáforos o mutexes.

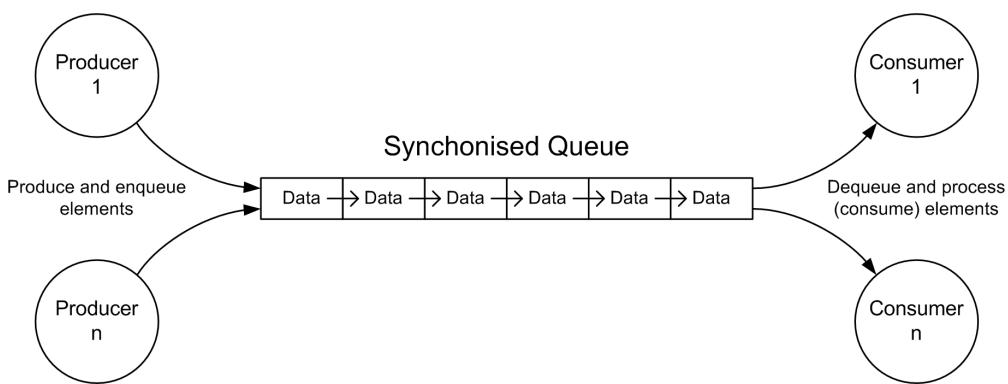


Figura 4.5: Problema del productor-consumidor.

Sin embargo FreeRTOS provee una estructura de datos adecuada para esta tarea denominadas *queues* que poseen las siguientes características[?][?]:

- Las colas son creadas con un tamaño de elementos fijo.
- El tamaño de cada elemento también se fija en su creación.
- Escribir y leer un elemento en la cola implica una copia byte a byte de los datos.
- Permiten operaciones de lectura y escritura a múltiples tareas.
- La lectura de una cola es bloqueante si la cola está vacía. La tarea que espera por un elemento en la cola despierta de manera automática cuando hay elementos disponibles.

- La escritura en una cola es bloqueante si la cola está llena. La tarea que espera por un espacio en la cola despierta de manera automática cuando hay espacios disponibles.

El flujo de trabajo cuando se utilizan colas en FreeRTOS se ilustra en la figura 4.6.

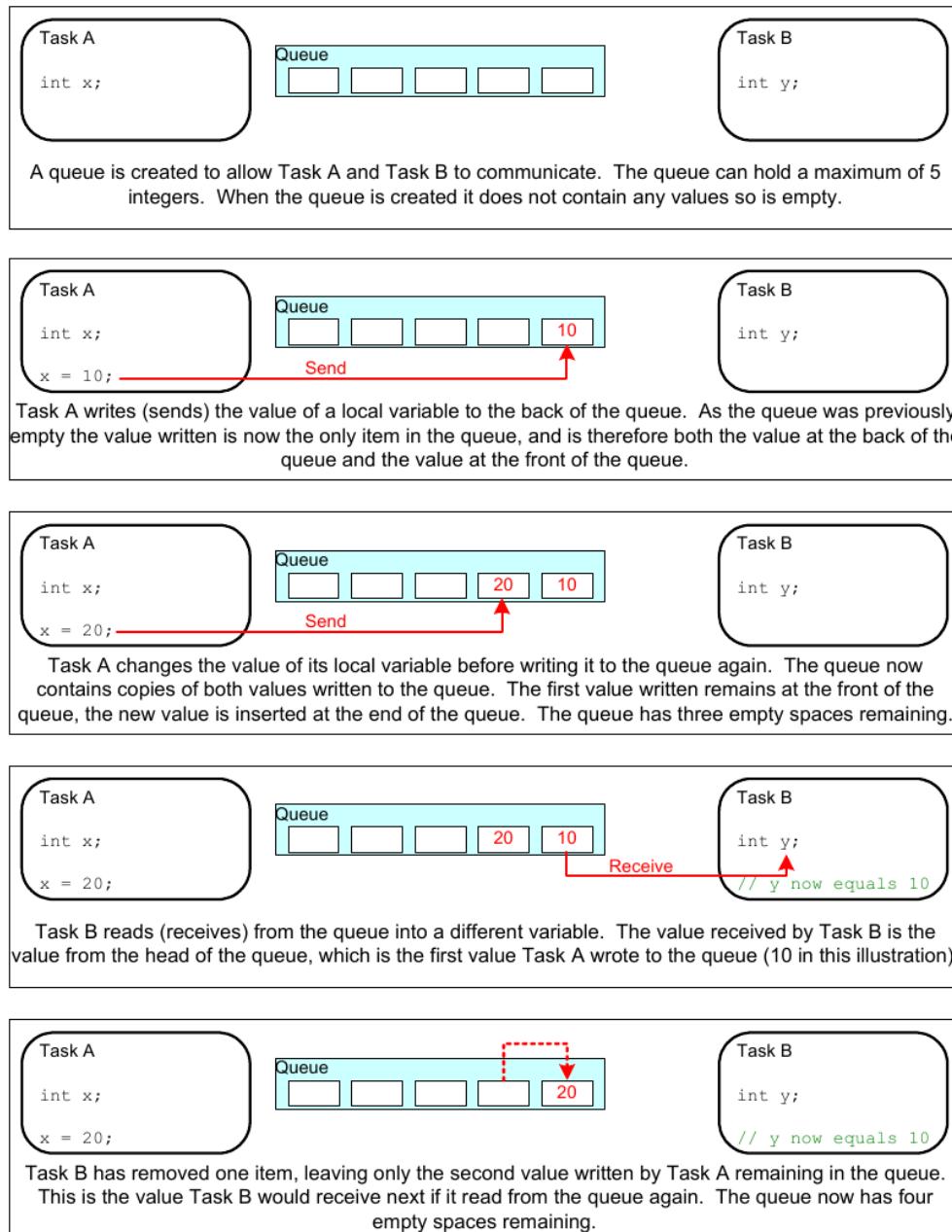


Figura 4.6: FreeRTOS Queue.

La implementación de este módulo corresponde al diagrama de flujo de la figura 4.7. Las tres funcionalidades básicas del procesador de comandos son: recibir un comando y obtener sus meta datos; determinar si su ejecución es posible; construir la estructura que se entregará al ejecutor. En el código 4.13 se implementan este diagrama de flujo. Se tiene entonces una tarea de FreeRTOS basada en eventos, pues despierta solamente cuándo existe algún comando disponible en la cola. La función `check_if_executable` implementa la lógica que determina

si el comando se puede ejecutar o no, en base a los meta datos de comandos y el estado actual del sistema. Finalmente se consulta al repositorio de comandos por la función asociada al código entregado y se construye la estructura de datos que representa un comando para el ejecutor que contiene el puntero a la función y su parámetro. Como sólo existe un proceso que ejecuta los comandos el *dispatcher* debe esperar a que termine su ejecución antes recuperar uno nuevo. Esta comunicación también se logra a través de una cola, de un sólo elemento, donde el ejecutor envía el resultado que retornó la función que acaba de ejecutar.

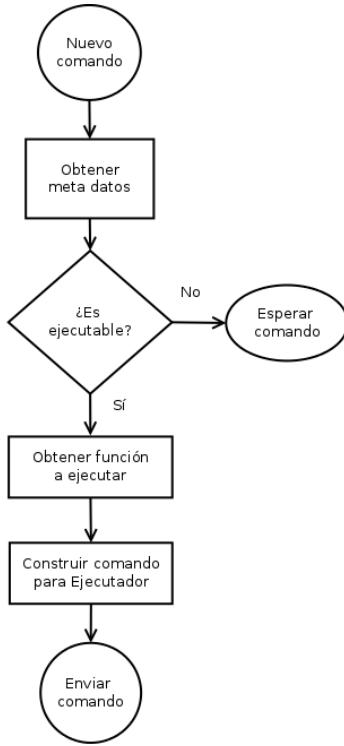


Figura 4.7: Diagrama de flujo para *dispatcher*.

4.5.6. Executer

Este módulo es el último eslabón en la cadena de acciones que involucran la ejecución de un comando y sus responsabilidades incluyen: recibir en una estructura la función que se requiere ejecutar junto a su parámetro; ejecutar la función y obtener su resultado; notificar el término y resultado de la operación al *dispatcher*. Esta tarea es activada mediante eventos, en específico la llegada de un comando. El *Executer* se debe comunicar con el *Dispatcher* de dos maneras:

- El *dispatcher* prepara un comando para el ejecutor y debe notificar la disponibilidad de este nuevo comando.
- El ejecutor notifica el término de la operación y su resultado al *dispatcher*.

Para la comunicación entre tareas involucrando el intercambio de mensajes se utiliza nuevamente la estructura *Queue* de FreeRTOS. En este caso se tienen dos mensajes que entregar para sincronizar dos estados del proceso total, por lo tanto se implementan dos colas de largo

Listing 4.13: taskDispatcher.c

```
1 #include "taskDispatcher.h"
2
3 extern xQueueHandle cmdQueue; /* Commands queue */
4 extern xQueueHandle executerCmdQueue; /* Executer commands queue */
5 extern xQueueHandle executerStatQueue; /* Executer result queue */
6
7 void taskDispatcher(void *param)
8 {
9     printf(">>[Dispatcher] Started\n");
10    portBASE_TYPE status; /* Status of cmd reading operation */
11
12    DispCmd newCmd; /* The new cmd readed */
13    ExeCmd exeCmd; /* Strucutre to executer */
14    int cmdId, idOrig, sysReq, cmdParam, cmdResult; /* Cmd metadata */
15
16    while(1)
17    {
18        /* Read newCmd from Queue - Blocking */
19        status = xQueueReceive(cmdQueue, &newCmd, portMAX_DELAY);
20
21        if(status == pdPASS)
22        {
23            /* Gets command metadata*/
24            cmdId = newCmd.cmdId;
25            idOrig = newCmd.idOrig;
26            sysReq = newCmd.sysReq;
27            cmdParam = newCmd.param;
28
29            /* Check if command is eecutable */
30            if(check_if_executable(&newCmd))
31            {
32                printf("[Dispatcher] Cmd: %X, Param: %d, Orig: %X\n",
33                      cmdId, cmdParam, idOrig);
34
35                /* Fill the executer command */
36                exeCmd.fnct = repo_getCmd(newCmd.cmdId);
37                exeCmd.param = param;
38
39                /* Send the command to executer Queue - BLOCKING */
40                xQueueSend(executerCmdQueue,&exeCmd, portMAX_DELAY);
41
42                /* Get the result from Executer Stat Queue - BLOCKING */
43                xQueueReceive(executerStatQueue ,&cmdResult, portMAX_DELAY);
44            }
45        }
46    }
```

igual a un elemento. Al ser las colas de largo uno, estas actúan como *mutexes* bloqueando un proceso hasta que el otro haya completado sus operaciones.

El flujo de operación del *executer* o ejecutor se detalla en la figura 4.8 y es bastante simple en cuánto cumple sus tres obligaciones de manera secuencial. Esta simplicidad, sin embargo, no revela la verdadera utilidad de este módulo: proveer un ambiente de ejecución exclusivo al comando. Por diseño sólo un módulo ejecutor existe en el sistema, por lo tanto, una vez aquí el comando toma el control del procesador, de todos los recursos de *hardware* y *software* para realizar su tarea. Además aprovecha la generalidad definida por la interfaz de los comandos, para permitir a este módulo ejecutar cualquier función que implemente la interfaz requerida.

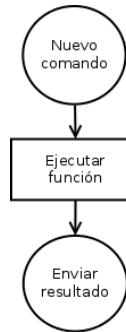


Figura 4.8: Diagrama de flujo para *executer*.

El código 4.14 detalla la implementación de este módulo. Al momento de crear esta tarea se debe tener en cuenta las siguientes consideraciones:

- Entregar la mayor cantidad de memoria de *stack* posible. Esta tarea realiza todas las funciones importantes para el sistema y cada función que se ejecuta en esta tarea utilizará la memoria de *stack* que se le asigna. A diferencia del resto de las tareas, donde su uso de memoria es parejo en el tiempo, el *executer* llama a diferentes funciones y algunas de ellas pueden requerir más memoria que otras. Como el objetivo es proveer el entorno de ejecución para un comando, se debe contar con la memoria suficiente de modo que ningún comando cause un *stack overflow* lo cual se debe determinar de manera experimental utilizando las herramientas de depuración disponibles en el entorno de desarrollo.
- Asignar la prioridad más alta posible. De esta manera se consigue que el comando en ejecución no sea interrumpido por ningún otro proceso con lo cual se aseguran dos puntos: primero el comando se ejecuta en el menor tiempo posible; y segundo, se elimina la necesidad de brindar una sincronización excesiva de los recursos compartidos de *hardware* asegurando el acceso exclusivo cada vez que se ejecuta un comando.

4.5.7. Listeners

Los *listeners* son los módulos encargados de implementar la lógica de generación de comandos dentro del sistema de vuelo. Existen una serie de *listeners* bajo la lógica de que a cada uno le corresponde controlar un determinado subsistema del satélite. Estos módulos se implementan como tareas de FreeRTOS que se activan de manera periódica, ejecutando

Listing 4.14: taskExecuter.c

```
1 #include "taskExecuter.h"
2
3 extern xQueueHandle executerCmdQueue; /* Comands queue*/
4 extern xQueueHandle executerStatQueue; /* Comands queue*/
5
6 void taskExecuter(void *param)
7 {
8     printf(">>[Executer] Started\n");
9     ExeCmd RunCmd;
10    int cmdStat, queueStat, cmdParam;
11
12    while(1)
13    {
14        /* Read the CMD that Dispatcher sent - BLOCKING */
15        queueStat = xQueueReceive(executerCmdQueue ,&RunCmd ,portMAX_DELAY);
16        if(queueStat == pdPASS)
17        {
18            printf("[Executer] Running a command... \n");
19            ClrWdt();
20
21            /* Execute the command */
22            cmdParam = RunCmd.param;
23            cmdStat = RunCmd.fnct((void *)&cmdParam);
24
25            ClrWdt();
26            printf("[Executer] Command result: %d\n", cmdStat);
27
28            /* Send the result to Dispatcher - BLOCKING */
29            xQueueSend(executerStatQueue , &cmdStat , portMAX_DELAY);
30        }
31    }
32 }
```

operaciones de control que tienen como salida comandos que son agregados a la cola del *dispatcher*. La lógica de funcionamiento se detalla en el diagrama de la figura 4.9. FreeRTOS ofrece dos funciones que implementan la periodicidad en las ejecución de las tareas, mediante la técnica de suspender su funcionamiento por una cantidad determinada de *ticks*: `vTaskDelay` y `vTaskDelayUntil`. Cada una de estas funciones cambia el estado de la tarea a suspendida, un estado donde no utiliza ningún recurso del procesador y luego del tiempo especificado retoma su ejecución. La diferencia entre ambas funciones es que la primera siempre suspende la tarea durante un tiempo fijo, mientras que la segunda mantiene fijo el tiempo que transcurre entre ambas llamadas de `delay[?]`. `vTaskDelayUntil` es más adecuada para tareas de tipo *hard real-time* y es la utilizada en los *listeners* para proveer una resolución de tiempo más precisa.

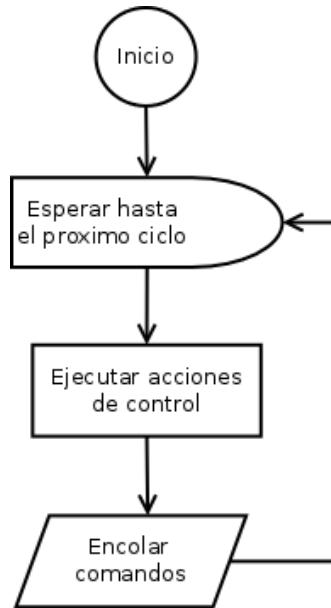


Figura 4.9: Diagrama de flujo para *listeners*.

Los *listeners* y su implementación pueden ser bastante específicos a cada sistema. La regla general es que se crea uno por cada subsistema que se debe controlar, pero la complejidad de cada tarea se esconde bajo la lógica llamada “operaciones de control” del diagrama 4.9.

Una de las funcionalidades principales que se puede esperar del sistema es la ejecución de acciones de manera periódica, acciones que incluyen tareas de control interno del propio subsistema al que corresponde el computador a bordo. Este *listener* es denominado *housekeeping* y su lógica de control es: ejecutar comandos de control interno a periodos fijos de tiempo. Según las necesidades del sistema, se puede implementar varios periodos fijos, en este caso ejecutarán comandos cada 1, 10, y 30 segundos. Para guiar la implementación de este módulo, su lógica se detalla en la figura 4.10

Los comandos que se ejecutan en cada intervalo de tiempo, dependerán de la aplicación, en este caso se ejecutarán una serie de ellos cuya función es actualizar las variables de estado del sistema. La implementación de esta tarea se detalla en el código 4.15.

Con la implementación del primer *listener* y la correcta inicialización del sistema opera-

Listing 4.15: taskHouskeeping.c

```
1 #include "taskHouskeeping.h"
2 extern xQueueHandle dispatcherQueue; /* Commands queue */
3
4 void taskHouskeeping(void *param)
5 {
6     printf(">>[Houskeeping] Started\r\n");
7     portTickType delay_ms      = 1000;      //Task period in [ms]
8     portTickType delay_ticks = delay_ms / portTICK_RATE_MS; //Task period
9
10    unsigned int elapsed_sec = 0;          // Seconds count
11    unsigned int _10sec_check = 10;        //10[s] condition
12    unsigned int _10min_check = 10*60;     //10[m] condition
13    unsigned int _1hour_check = 60*60;     //1[h] condition
14
15    DispCmd NewCmd;
16    NewCmd.idOrig = CMD_IDORIG_THOUSEKEEPING; //Housekeeping
17
18    portTickType xLastWakeTime = xTaskGetTickCount();
19    while(1)
20    {
21        vTaskDelayUntil(&xLastWakeTime, delay_ticks); //Suspend task
22        elapsed_sec += delay_ms/1000; //Update seconds counts
23
24        /* 10 seconds actions */
25        if((elapsed_sec % _10sec_check) == 0)
26        {
27            printf("[Houskeeping] _10sec_check\n");
28            NewCmd.cmdId = obc_id_get_rtos_memory;
29            NewCmd.param = 0;
30            xQueueSend(dispatcherQueue, &NewCmd, portMAX_DELAY);
31        }
32
33        /* 10 minutes actions */
34        if((elapsed_sec % _10min_check) == 0)
35        {
36            printf("[Houskeeping] _10min_check\n");
37            NewCmd.cmdId = drp_id_print_CubesatVar;
38            NewCmd.param = 0;
39            xQueueSend(dispatcherQueue, &NewCmd, portMAX_DELAY);
40        }
41
42        /* 1 hours actions */
43        if((elapsed_sec % _1hour_check) == 0)
44        {
45            printf("[Houskeeping] _1hour_check\n");
46            NewCmd.cmdId = drp_id_update_dat_CubesatVar_hoursWithoutReset;
47            NewCmd.param = 1; //Add 1 hour
48            xQueueSend(dispatcherQueue, &NewCmd, portMAX_DELAY);
49        }
50    }
51 }
```

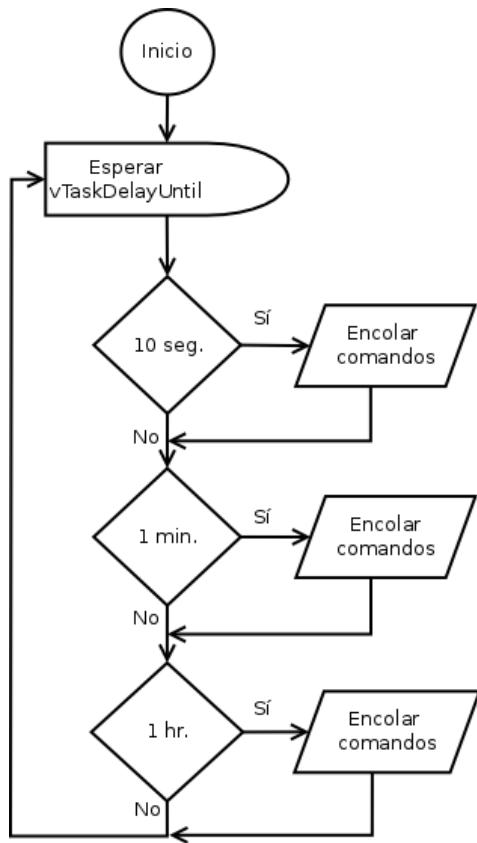


Figura 4.10: Diagrama de flujo para *housekeeping*.

tivo, las tareas y los repositorios, se tiene la primera versión funcional del sistema de vuelo. Si bien no se realiza ninguna tarea fundamental para lo que significa el proyecto satelital, la generalidad de la implementación responde principalmente a los requerimientos no operacionales del proyecto y se convierte en la base para cualquier proyecto derivado.

Este punto del desarrollo está marcado como la versión v0.1-base dentro del sistema control de versiones. El resultado de la ejecución del *software* a este punto se detalla a continuación.

```

>>Starting FreeRTOS [->]
>>[Executer] Started
>>[Dispatcher] Started
>>[Houskeeping] Started
[Houskeeping] _10sec_check
[Dispatcher] Cmd: 1001, Param: 0, Orig: 1001
[Executer] Running a command...
Free RTOS memory: 2282
[Executer] Command result: 2282
[Houskeeping] _10sec_check
[Dispatcher] Cmd: 1001, Param: 0, Orig: 1001
[Executer] Running a command...
Free RTOS memory: 2282

```

```
[Executer] Command result: 2282
(...)
[Houskeeping] _10min_check
[Dispatcher] Cmd: 5002, Param: 0, Orig: 1001
[Executer] Running a command...
=====
          Status repository
=====
0, -1
1, -1
2, -1
3, -1
4, -1
5, -1
6, -1
7, -1
8, -1
9, -1
10, -1
(...)

[Executer] Command result: 1
(...)
[Houskeeping] _1hour_check
[Dispatcher] Cmd: 5000, Param: 1, Orig: 1001
[Executer] Running a command...
[Executer] Command result: 1
(...)
[Houskeeping] _1hour_check
[Dispatcher] Cmd: 5000, Param: 1, Orig: 1001
[Executer] Running a command...
[Executer] Command result: 1
(...)
[Houskeeping] _10min_check
[Dispatcher] Cmd: 5002, Param: 0, Orig: 1001
[Executer] Running a command...
=====
          Status repository
=====
0, -1
1, 2
2, 2
3, -1
4, -1
5, -1
6, -1
7, -1
8, -1
```

```
9, -1  
10, -1  
(...)
```

De la salida se observa la inicialización de todas las tareas del sistema. *Houskeeping* se ejecuta de manera periódica enviado comandos al *Dispatcher*, lo que provoca la activación de este módulo que registra el nuevo comando recibido, su parámetro y su origen. El efecto en cadena implica la activación del *Executer* quien registra el inicio de la ejecución del comando. Si el comando considera salida de datos por la consola serial, se ve reflejado en este punto, sino de todas maneras el *Executer* muestra el resultado de la operación.

4.6. Específico al proyecto SUCHAI

Las secciones anteriores implementan la base del sistema de vuelo con mínimas funcionalidades, básicamente completando la arquitectura de *software* propuesta. Sobre esta base se completa el resto de los requerimientos operacionales del satélite lo que implica extender el sistema mediante dos métodos: agregar *listeners* que controlan subsistemas específicos del satélite; y agregando comandos para cada función específica.

4.6.1. Consola serial

La consola serial es una herramienta fundamental para el proceso de depuración y pruebas del sistema en desarrollo. Se implementa el protocolo de transferencia de datos seriales RS232 a través de los periféricos disponibles en la plataforma de *hardware*:

- Módulos UART en el microcontrolador.
- Puerto serial DB9 con conversor de voltaje en la placa de desarrollo.
- Puerto USB con conversor USB-Serial (FT232R) en placa de desarrollo.

La salida de datos hacia el puerto serial es bastante directa haciendo uso del controlador implementado para los módulos UART, además de la adaptación de funciones comunes como *printf*. Como la salida de *debug* está bastante extendida a lo largo del programa, las tareas que se ejecutan de manera concurrente puede requerir el acceso simultaneo a este recurso compartido, por esto la principal adaptación consiste en convertir a *printf* en una función *thread safe* mediante la utilización de herramientas de sincronización para proveer exclusión mutua sobre el recurso.

La entrada de datos requiere mayor detalle en la implementación en cuanto debe permitir el reconocimiento de las cadenas de caracteres que se ingresan como órdenes al sistema. Para esto se implementa un *listener* con dedicación exclusiva tomar los caracteres de la consola serial, formar las cadenas de texto, separar parámetros e interpretar la orden como comandos del sistema de vuelo que serán encolados para su ejecución. El detalle de la implementación de esta tarea se ilustra a través del diagrama de flujo de la figura 4.11.

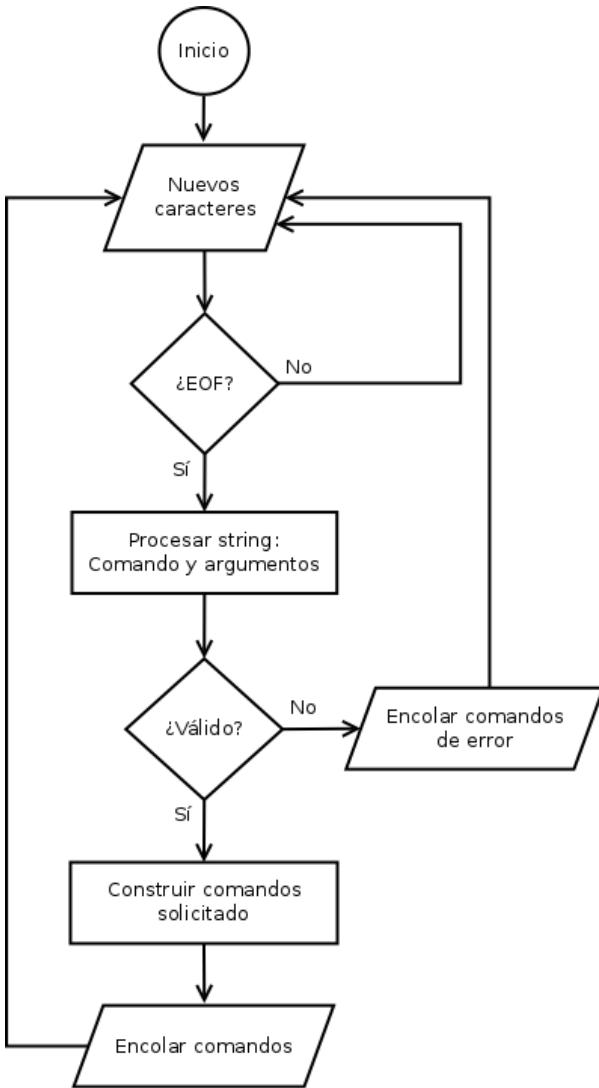


Figura 4.11: Diagrama de flujo para *taskConsole*.

4.6.2. Plan de vuelo

El plan de vuelo es la solución a parte importante de los requerimientos operacionales del satélite y consiste en la capacidad de realizar tareas programadas para cierto instante de tiempo o ubicación en la órbita.

La solución consiste en implementar un *listener* que monitorice la hora y fecha del sistema para obtener desde el plan de vuelo el comando adecuado a ser encolado para su ejecución. El plan de vuelo en sí, consiste en una lista ordenada temporalmente con códigos de comandos y sus parámetros implementando como arreglos o archivos en un memoria externa no volátil. El flujo de operación de la tarea corresponde al diagrama de la figura 4.12.

El plan de vuelo implica también la implementación de comandos que realicen las siguientes operaciones:

- Leer una determinada entrada del plan de vuelo

- Modificar una entrada del plan de vuelo
- Borrar completamente el plan de vuelo

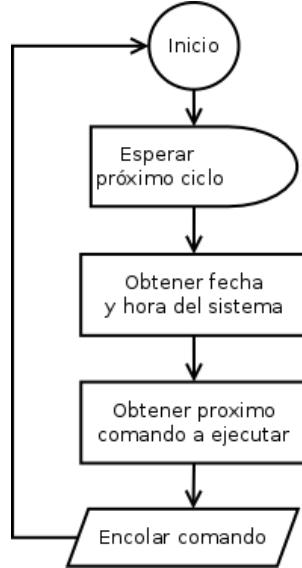


Figura 4.12: Diagrama de flujo para *taskFlightPlan*.

4.6.3. Comunicaciones

Uno de los módulos fundamentales del sistema satelital corresponde a las comunicaciones. El sistema debe ser capaz de recibir y procesar telecomandos enviados desde la estación terrena; enviar la telemetría generada hacia la estación terrena; activar la transmisión de *beacon* configurables de manera periódica; y realizar todas la configuraciones de *hardware* para su correcto funcionamiento.

Por diseño se tiene nuevamente dos formas de completar estos requerimientos: agregar comandos y programar un *listener*. Las funciones que se deben implementar a través de nuevos comandos en el sistema incluyen:

- Configurar *hardware* de comunicaciones.
- Leer configuraciones de *hardware* de comunicaciones.
- Leer telecomandos recibidos por el subsistema de comunicaciones.
- Escribir telemetría para ser transmitida por el subsistema de comunicaciones.
- Configurar un *beacon* y transmitirlo.

Al igual que el caso de la consola serial, se requiere la implementación de un *listener* para controlar las operaciones de lectura y procesamiento de las órdenes enviadas de manera remota al satélite. El estado de funcionamiento del sistema de comunicaciones está disponible como variables de estado a través del repositorio de estados, por lo tanto, las operaciones del *listener* de comunicaciones incluyen monitorear estas variables para realizar las operaciones correspondientes. Estas incluyen monitorear la llegada de nuevos *frames* de telecomandos para proceder su lectura; procesar los telecomandos leídos generando los comandos del sistema

solicitados; y generar a intervalos fijos de tiempo el *beacon* del satélite con la información adecuada. La implementación de este *listener* responde al diagrama de flujo de la figura 4.13

4.6.4. Inicialización del sistema

El software de vuelo debe ser capaz de recuperarse adecuadamente a un reinicio del sistema inesperado, el cual se puede provocar por diferentes motivos como: falla en la ejecución de una instrucción en el microcontrolador; un reinicio provocado por el *watchdog* si la aplicación se congela; la falta de energía en los buses de alimentación por falta de carga en las baterías; sobre corrientes; o por algún factor externo como radiación.

Parte de esta capacidad se implementa en el repositorio de estados que provee un almacenamiento no voltátil de las variables del sistema, pero además al inicio se debe asegurar que todos los sub-sistemas de *hardware* y *software* estén correctamente inicializados lo que incluye las operaciones de despliegue del satélite desde su vehículo lanzador. Estas operaciones incluyen:

- Silencio radial al momento del lanzamiento.
- Despliegue de antenas.
- Inicializar y configurar todos los subsistemas de *hardware*.
- Inicializar todos los repositorios: datos, comandos y estados.
- Crear las tareas que corresponden a los *listeners*.

Para realizar estas funciones se implementa un *listener* cuya función sea ejecutar las acciones de inicialización del sistema. Este *listener* es el primero en entrar en funcionamiento y tiene prioridad por sobre el resto ya que sus operaciones son fundamentales para la correcta operación de todo el satélite. Esta tarea es no periódica, por el contrario, sólo se ejecuta una vez y luego es eliminada. Antes de terminar su ejecución inicia el resto de los *listeners* que toman el control del sistema. La implementación se basa en el diagrama de flujo de las figura 4.14

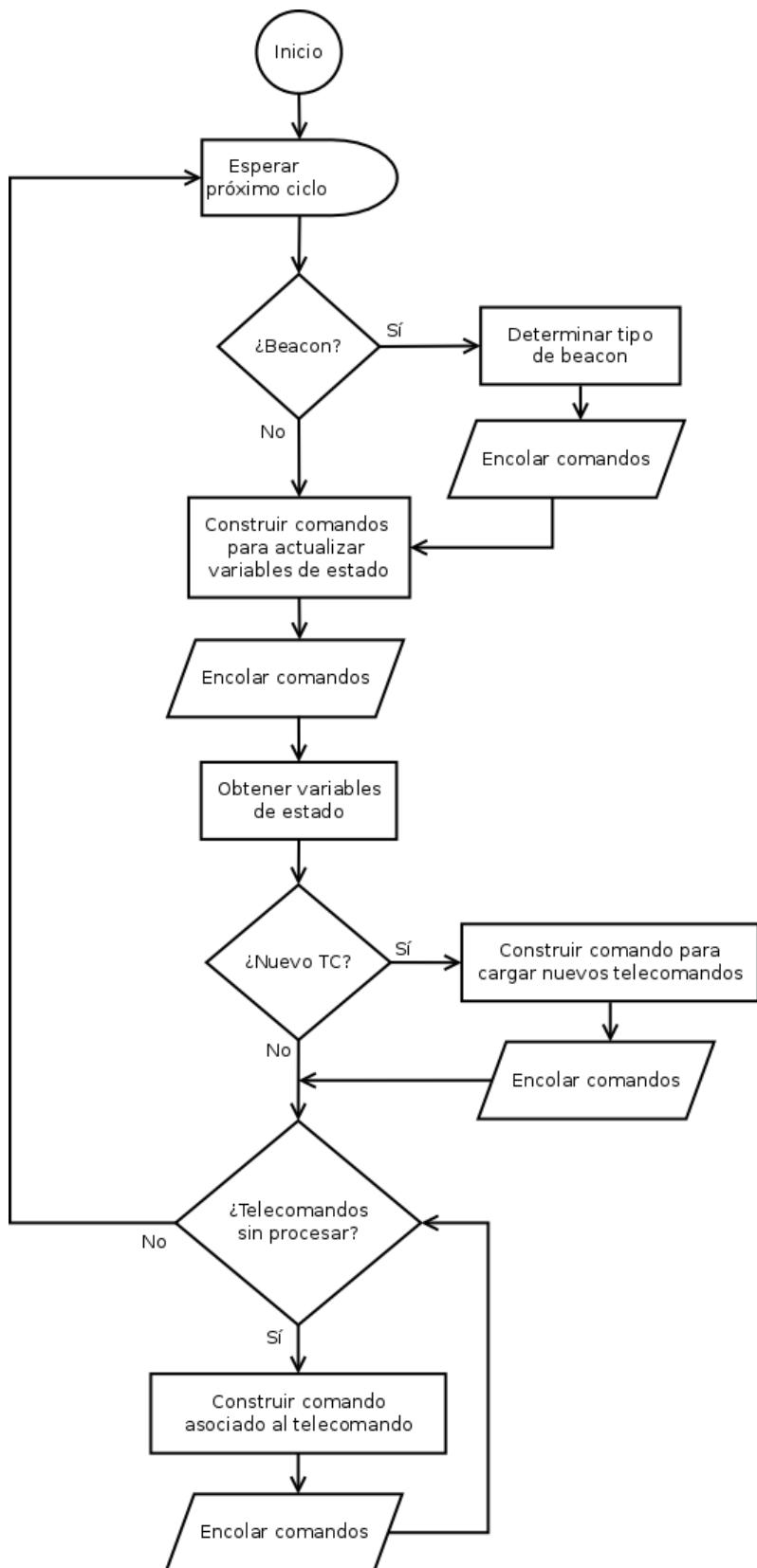


Figura 4.13: Diagrama de flujo para *taskCommunications*.

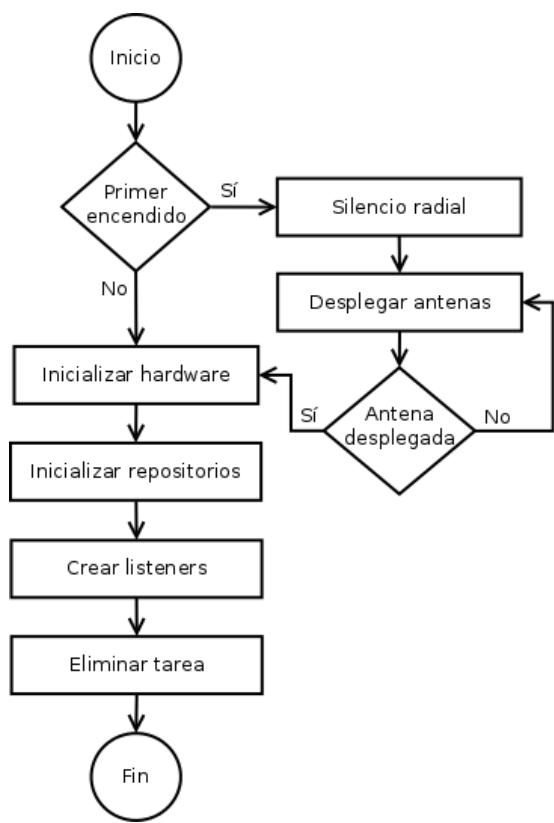


Figura 4.14: Diagrama de flujo para *taskDeployment*.

Capítulo 5

Pruebas y resultados

En este capítulo se detallan y discuten los resultados obtenidos luego de la implementación del sistema. Se analiza el funcionamiento del satélite, con sus tres subsistemas básicos integrados, y se revisa el cumplimiento de las funcionalidades esperadas por el equipo del proyecto SUCHAI. Se realizan pruebas de rendimiento y se obtienen estadísticas del uso de los recursos de *hardware*, para así determinar si la solución es adecuada para la plataforma objetivo.

5.1. Pruebas de rendimiento

5.1.1. Estadísticas del programa

Mediante las herramientas que provee el IDE MPLABX, se obtienen diferentes estadísticas sobre el código que se ha programado, para generar una medida básica de la envergadura del proyecto. Lo anterior se muestra en la tabla 5.1.

Tabla 5.1: Estadísticas del código

Lineas de código	17475
<i>Commits</i>	986
<i>Tags</i>	2
Contribuyentes	4
Comandos	107

5.1.2. Estadísticas de uso de memoria

A continuación, se detalla el uso de memoria de programa y datos que utiliza el *software*, luego de ser compilado y programado en el microcontrolador. Los resultados se detallan en la tabla 5.2.

Tabla 5.2: Uso de memoria de la aplicación

Memoria	Usada [bytes]	Total [bytes]	Porcentaje de uso
RAM	9688	16384	59 %
FLASH	37566	87548	43 %

Lo anterior, demuestra que la solución programada se ajusta a la cantidad de memoria de datos y de programa disponibles. A pesar de la reducida cantidad de memoria de programa disponible, aún queda más del 50 % libre para programar más comandos y nuevas funcionalidades. Lo más crítico es la memoria RAM, que se encuentra utilizada en cerca del 60 %, lo cual podría ver reducida la capacidad de agregar nuevos *listeners*. Esto porque cada tarea del sistema operativo requiere reservar una cantidad de memoria para su funcionamiento.

Otro punto importante a considerar, es la cantidad de memoria asignada al sistema operativo, y la configuración de memoria de *stack* que se ha asignado a las diferentes tareas. Si la memoria no es suficiente, se pueden generar condiciones de *stack overflow* que dejarían al sistema inestable o no funcional. La asignación de memoria para las tareas se realiza de manera experimental, según los requerimientos de la aplicación [?]. Para comprobar que no existen problemas, se realiza la tabla 5.3 a partir de la salida de depuración que provee el entorno de desarrollo.

Tabla 5.3: Uso de memoria del sistema operativo

Tarea	Prioridad	Memoria [bytes]		
		Asignada	Usada	Porcentaje de uso
idle	0	115	79	69 %
flightplan	2	230	145	63 %
communications	2	230	125	54 %
console	2	230	91	39 %
housekeeping	2	230	97	42 %
dispatcher	3	230	109	47 %
executer	4	575	85	14 %

La tabla muestra que la asignación de memoria de las tareas es la correcta, en cuanto ninguna está utilizando la totalidad del valor asignado. La cantidad de memoria utilizada por una tarea es una variable dinámica, que depende de las acciones concretas que esta ejecutando la aplicación, por lo que siempre se debe dejar un margen de seguridad en la asignación. El caso mas crítico es el *executer*, debido a que ejecuta una variedad de comandos, por lo tanto, el dato mostrado en la tabla 5.3 no es confiable y se debe asignar la mayor cantidad de memoria de *stack* posible para evitar que algún comando cause *stack overflow*.

5.1.3. Estadísticas de uso del procesador

Para evaluar si la solución implementada es adecuada, se debe considerar el porcentaje de uso del procesador, bajo condiciones de operación nominales. Este parámetro indica si la

CPU se ve sobre cargada con la ejecución del *software* o bien, si quedan recursos disponibles para agregar más funcionalidades.

Para obtener estos resultados la técnica habitual es contabilizar el porcentaje de tiempo en que el sistema operativo no tiene tareas por ejecutar y, por lo tanto, se encuentra en estado *idle*. Esto se logra generando una señal cada vez que se entra en este estado, en este caso, se imprime por consola serial el valor de una variable, como se detalla en la figura ??

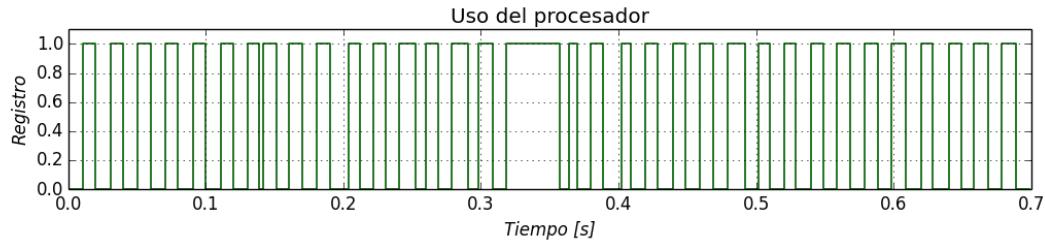


Figura 5.1: Registro de uso del procesador. El registro cambia de 0 a 1 cada vez que se entra en estado idle. Si no hay tareas que ejecutar esto ocurre cada 100 ms, de otro modo se observan períodos más largos.

Con estos datos, se procede a determinar la cantidad de veces en que la transición de la variable toma 100 ms. (el período de los *ticks* del sistema), lo que indica que el sistema operativo estuvo un ciclo completo sin tareas que ejecutar. Como se detalla en la figura ?? se obtiene un 91.8 % de disponibilidad. Esto significa que el procesador puede ejecutar sin problemas la solución implementada, dejando espacio para agregar más funcionalidades.



Figura 5.2: Porcentaje de uso del procesador. La solución no sobrecarga la CPU, dejando un 91.8 % de capacidad de cómputo para nuevas funcionalidades

5.2. Pruebas de integración

Con las pruebas de integración se busca determinar el cumplimiento de los requerimientos operacionales del proyecto, especialmente aquellas capacidades que dependen directamente de la implementación del *software* de vuelo. Para esto, se han integrado los tres sub-sistemas básicos del satélite (computador a bordo, energía y comunicaciones) y el sistema se hace funcionar durante un tiempo prolongado simulando condiciones de operación nominales siguiendo la técnica denominada *hardware in the loop simulation* común en las pruebas de sistemas embebidos complejos.

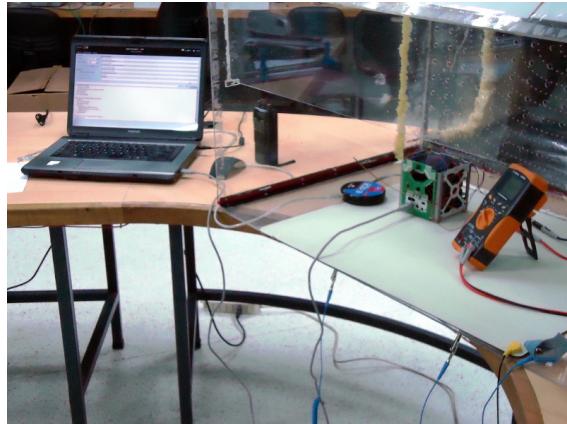
5.2.1. Configuración

Las herramientas utilizadas para el desarrollo de la prueba consisten en:

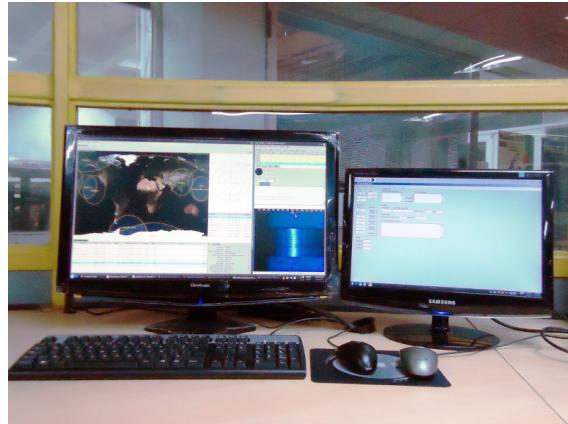
- Entorno de prueba:
 - Caja limpia para albergar el satélite.
 - Computador para registro de datos.
 - Programador Microchip ICD3.
 - Cable USB para comunicación serial.
 - Equipo de radio portátil UHF.
 - Cámara de video para registro de la prueba.
 - Multímetro.
- Estación terrena:
 - Antena monopolo, 50Ω , de baja ganancia.
 - Equipo de radio ICOM 910H.
 - Computador para capturar y procesar audio.
- Satélite:
 - Computador a bordo con placa madre CubesatKit rev. D y módulo de procesador CubesatKit D1 con PIC24F256GA110.
 - EPS CS-1UEPS2-NB ClydeSpace
 - Transceptor AllSpace ASCOM-01 rev. 3
 - Paneles solares.

El montaje de la prueba, como se muestra en la figura 5.3, consiste en mantener el satélite dentro de la cámara limpia conectado, a través de un cable USB, al computador que registrará los resultados de la prueba y que envía comandos al *software* de vuelo. Este cable también provee de alimentación al sistema de energía, para la carga de las baterías. Se utilizará un multímetro para monitorear su voltaje, además de los registros generados por el *software*. La radio portátil se utiliza para una rápida verificación del funcionamiento del sistema de comunicaciones, pues permite detectar si se produce alguna transmisión durante el periodo de inactividad obligatorio. La estación terrena está configurada para escuchar el sistema de comunicaciones del satélite, con un programa para de decodificar su baliza y telemetría. El

satélite, a través del *software* de vuelo, funciona de manera autónoma y la interfaz de *debug* sólo se utilizará para registrar información útil para la prueba.



(a) Montaje del satélite



(b) Estación terrena

Figura 5.3: El satélite se encuentra en una caja limpia, conectado a un computador que registra la prueba (a). La estación terrena se controla remotamente, a través del *software* GPredict y el proveído por el fabricante del *transceiver*(b)

El desarrollo de la prueba consiste en los siguientes pasos.

- Configurar el *software* de vuelo y programar el computador a bordo.
- Llevar al *software* de vuelo al estado de 'primer encendido'.
- Comenzar el registro de la consola serial.
- Monitorear el cumplimiento del periodo de inactividad.
- Monitorear el correcto despliegue de antenas.
- Monitorear la correcta inicialización del *software* de vuelo.
- Registrar los comandos ejecutados en el *software* de vuelo.
- Registrar periódicamente el valor de las variables de estado.
- Registrar periódicamente los registros de la EPS.
- Recibir el *beacon* emitido en la estación terrena.
- Enviar una señal de radio para activar el envío de telemetría.
- Recibir la telemetría enviada en la estación terrena.
- Ejecutar test de reinicio.
- Ejecutar test de falla de *software*.
- Ejecutar test de energía baja.

Los resultados de la prueba se obtendrán en el siguiente formato:

- Archivo de texto con el registro de la consola serial.
- Texto y audio del *beacon* recibido en la estación terrena.
- Datos de la telemetría decodificada en la estación terrena.
- Imágenes del proceso de despliegue de antena.

5.2.2. Resultados

Los resultados obtenidos son utilizados para verificar los requerimientos operacionales de la misión, en específico, aquellos que dependen directamente del *software* de vuelo.

Área de comunicaciones

Configuración inicial del *transcevier*: la configuración inicial del *transcevier* se lleva a cabo durante el proceso de inicio del software de vuelo. Esto se observa en el registro de la consola que indica la ejecución del comando `trx_initialize`, el que incluye la configuración del *beacon* por defecto mediante el comando `trx_setbeacon`:

Listing 5.1: Inicialización del *transcevier*

```
* Setting TRX
Setting beacon: SUCHAIATINGDOTUCHILEDOTCL -
>> TRX Setting BEACON:
    Setting Offset...
    Setting Content...
    Setting Beacon Length... 26
```

La correcta ejecución de estos comandos, como se muestra en el registro 5.1, implica una comunicación activa con el dispositivo, así como una confirmación de la correcta configuración de las variables escritas. Con esto se comprueba que el dispositivo, en lo referido a su interfaz con el *software*, se encuentra operativo.

El silencio radial se lleva a cabo configurando el *transcevier* en modo silencioso al inicio del sistema. Como medida adicional, se evita la generación de cualquier transmisión, inhibiendo la ejecución de comandos durante el periodo de inactividad. Durante la prueba, se mantiene cerca la radio portátil para detectar cualquier transmisión, así como para generar intentos de comunicación con el satélite que no deben tener respuesta. El resultado se detalla en el registro 5.2 que revela que, en efecto, ningún comando de transmisión de datos se efectúa durante el periodo de inactividad radial. También se comprueba el tiempo de inactividad toma 33 minutos, de acuerdo a lo requerido.

Listing 5.2: Registro del periodo de inactividad

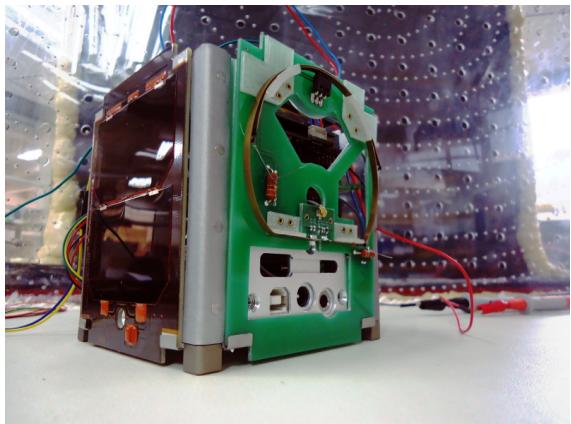
```
[16:35:23.046699] [dep_silent_time] Mandatory inactivity time...
[16:35:23.081650]      First time on. Antenna NOT deployed
[16:35:23.095963]      STARTING 30MIN SILENT TIME
[16:35:23.120451]      * Turning off TX
[16:35:23.131351]      * System halted at >>[29/8/13 - 16:35:31]
[17:07:04.564344]      * 65[s] remaining ...
[17:08:10.107138]      * System resumed at >>[29/8/13 - 17:8:17]
[17:08:10.143954]      FINISHING SILENT TIME
```

Despliegue de antenas: este proceso está a cargo de la secuencia de inicio del *software* de vuelo, que activa y controla el sistema de despliegue. El proceso se muestra en la consola, como se observa en el registro 5.3. En este caso luego del séptimo intento de despliegue se detecta que el *switch* indicador del despliegue se ha liberado continuando con el inicio del sistema.

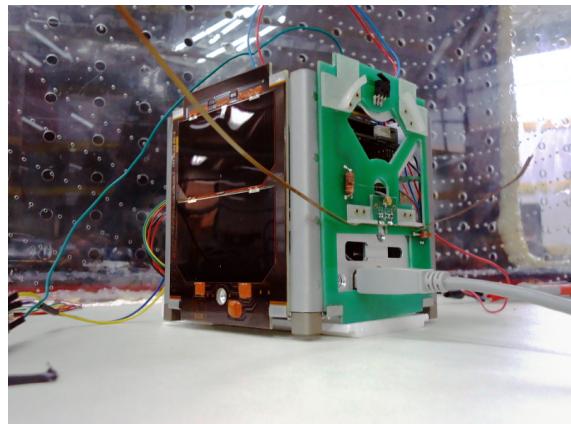
Listing 5.3: Registro del despliegue de antenas

```
[17:08:10.163688] [dep_deploy_antenna] Deploying TRX Antenna...
[17:08:10.201290]      [Deploying] Attempt #1 //Intenta desplegar cada
      antena
[17:11:21.425436]      [Deploying] Attempt #2
[17:14:32.682874]      [Deploying] Attempt #3
[17:17:43.940662]      [Deploying] Attempt #4
[17:20:55.198422]      [Deploying] Attempt #5
[17:24:06.456182]      [Deploying] Attempt #6
[17:27:17.713377]      [Deploying] Attempt #7
[17:30:30.970832]      ANTENNA DEPLOYED SUCCESSFULLY [7 TRIES]
```

En la figura 5.4 se observan las antenas antes y después del proceso de despliegue.



(a) Previo al despliegue



(b) Antenas desplegadas

Figura 5.4: Despliegue de antenas

Procesamiento de telecomandos: utilizando la aplicación de control de la estación terrena para el *transceiver* del satélite, se generaron secuencias de comandos que el *software* de vuelo debe decodificar y ejecutar. La secuencia de comandos generados es: imprimir la hora del sistema, imprimir las variables de estado y enviar las variables de estado como telemetría. Esto se traduce en el siguiente telecomando:

```
<cmd1> <para> <cmd2> <para> <stop>
0x5008 0x0000 0x8003 0x0000 0xFFFF
```

El resultado de esta prueba no es satisfactorio debido a que el satélite no es capaz de recibir los telecomandos enviados, producto de una falla del equipo de comunicaciones a bordo. Esta

falla, a lo largo del proceso de desarrollo, ha hecho imposible la comunicación confiable entre la estación terrena y el satélite. Los detalles de su detección y posibles soluciones escapan al desarrollo de este trabajo.

Protocolo de enlace: el *software* de vuelo se encarga de generar *beacons* de manera periódica, los que permiten identificar al satélite cuando está en órbita y contienen información valiosa sobre su funcionamiento. Cada cuatro minutos, y de manera intercalada, se generan dos tipos de *beacon*: uno simple que contiene sólo un identificador; y luego otro más largo, que agrega el valor de algunas variables de estado de interés. Esto se observa en el registros de la consola 5.4:

Listing 5.4: Beacons generados por el *software de vuelo*

```
//El primer beacon solo contiene un identificador
[17:34:36.705630] >>Beacon:SUCHAIATINGDOTUCHILEDOTCL -0
[17:34:36.722492] >> TRX Setting BEACON:
[17:34:36.736283]           Setting Offset...
[17:34:36.873040]           Setting Content...
[17:34:40.890843]           Setting Beacon Length... 27

//Luego de cuatro minutos, se genera un beacon con variables de estado
[17:38:36.710971] >>Beacon:SUCHAIATINGDOTUCHILEDOTCL -11000017
H30761940780001
[17:38:36.741475] >> TRX Setting BEACON:
[17:38:36.761285]           Setting Offset...
[17:38:36.910411]           Setting Content...
[17:38:44.190745]           Setting Beacon Length... 49
```

A los cuatro minutos de funcionamiento se genera el primer *beacon*, cuyo contenido es: SUCHAIATINGDOTUCHILEDOTCL-0. Esto es el identificador seguido de un guión y un número que indica que este mensaje no tiene más información. El siguiente *beacon* corresponde al texto: SUCHAIATINGDOTUCHILEDOTCL-11000017H30761940780001 e incluye variables de estado en su contenido, según lo especificado en la tabla 5.4.

Para comprobar el correcto funcionamiento de la lógica que genera la información de cada *beacon*, se realiza la tabla 5.4, donde se compara el contenido generado y el valor de las variables de estado en ese momento.

Se debe notar que, en el texto del *beacon*, la hora del despliegue se codifica de 1 a N, por lo tanto, cuando se lee una H se interpreta como un 17, lo cual es consistente con el valor de la variable en memoria. Con esto, se concluye que la lógica que permite rastrear al satélite y recuperar información esencial de su funcionamiento, actúa según lo esperado.

En la estación terrena se reciben estas señales y son procesadas por el *software* que permite decodificar el código Morse. Un ejemplo de la señal recibida en la estación terrena se muestra en la figura ??.

Los textos obtenidos para una secuencia de tres *beacons* transmitidos fueron:

Tabla 5.4: Beacons generados por el *software* de vuelo

Parámetro	Estado	Beacon	Estado	Beacon	Estado	Beacon
Hora del registro		18:34:36		19:38:36		20:11:37
Modo de operación	1	1	1	1	1	1
Horas sin reset	1	1	2	2	0	0
Contador de reset	0	0	0	0	7	7
Antena desplegada	1	1	1	1	1	1
Intentos de despliegue	7	7	7	7	7	7
Hora de despliegue	17	H	17	H	17	H
Minuto de despliegue	30	30	30	30	30	30
Voltaje baterías [V]	7,70642	7,7	7,80971	7,8	7,64069	7,6
Temperatura baterías [C]	19,873	19	19,873	19	18,406	18
Nivel de carga [SOC]	5	5	6	6	4	4
Baterías cargando	1	1	1	1	0	0
Promedio RSSI [dBm]	-51	-78	-54	-78	-52	-78
Contador de TM	0	0	0	0	0	0
Contador de TC	0	0	0	0	0	0
Estado memoria SD	1	1	1	1	1	1

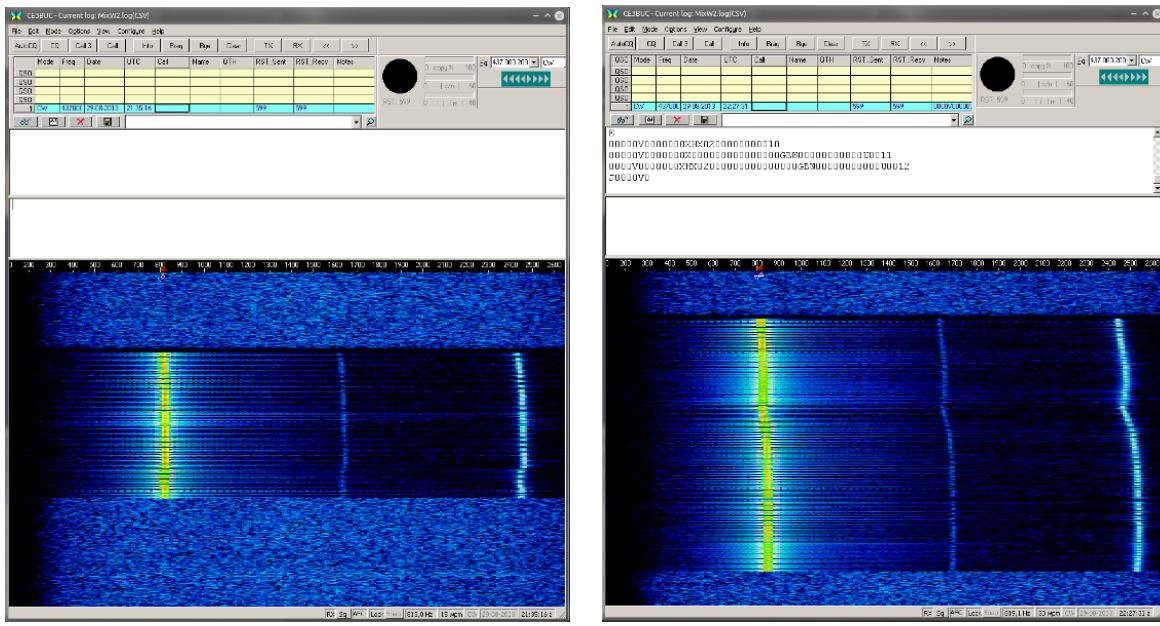


Figura 5.5: Se utiliza el *software* MixW para decodificar una transmisión de radio demodulada por la estación terrena.

00000V0000000XHX02000000000000000GBW00000000DK000024,
 00000V0000000XHX02000000000000000GBW00000000DK000025,
 00000V0000000XHX020000000000026.

Lo anterior claramente significa una falla, debido a que no se recibe el texto esperado, y se debe a un error del equipo de comunicaciones al configurar el nuevo texto del *beacon*. Se descartan errores en la estación terrena debido a que la secuencia decodificada tiene sentido: al final de cada linea recibida se observa un contador, que es agregado automáticamente por el *transcevier*, y que aumenta de manera secuencial en cada transmisión. El cero al inicio de la secuencia, también es parte del formato agregado por el dispositivo transmisor, lo cual permite concluir que es este quien no configura de manera correcta el mensaje.

El protocolo de enlace, una vez identificado el satélite, requiere enviarle una señal iniciar la descarga de datos o ejecutar telecomandos. Debido a las fallas del equipo de comunicaciones, que no permiten la recepción de telecomandos, el satélite funciona un modo de respaldo. En este modo, ante la presencia de un nivel fuerte de señal en la banda asignada, durante al menos 10 segundos, se inicia automáticamente la descarga de los datos recolectados.

El sistema reacciona ante la emisión de la señal portadora, en la frecuencia asignada, por parte del equipo de radio portátil, así comienza una cuenta con los segundos que la señal está presente, como se muestra en el registro 5.6. Cuando se superan los 10 segundos y se detecta la ausencia de señal, se produce la inmediata descarga de la telemetría almacenada en el satélite.

Listing 5.5: Descarga de telemetría en modo de respaldo

```
//Al detectar la señal portadora comienza el conteo
[19:37:55.076357] RSSI_CNT=2
[19:37:57.480979] RSSI_CNT=4
[19:37:59.055074] RSSI_CNT=6
[19:38:01.055657] RSSI_CNT=8
[19:38:03.063135] RSSI_CNT=10
[19:38:05.055308] RSSI_CNT=12

//Al superar los 10 segundos, se genera el comando de descarga
[19:38:07.107302] com_doOnRSSI..
[19:38:07.116383] [Dispatcher] Orig: 0x1102 | Cmd: 0x8003 | Param: 2

//Primer frame de telemetría
[19:38:07.191023] >> Sending START_WRITE_TM command... [OK]
[19:38:08.990042] >> Sending START_SEND_TM command...
[19:38:09.769988] >> Telemetry transmitted [OK]

//Segundo frame de telemetría
[19:38:09.790260] >> Sending START_WRITE_TM command... [OK]
[19:38:12.316259] >> Sending START_SEND_TM command...
[19:38:13.104915] >> Telemetry transmitted [OK]
```

Asimismo, en la estación terrena se recibe la señal proveniente del satélite, y el *software* decodifica los datos, mostrando el resultado en pantalla, como en la figura 5.6.

Envío de telemetría: el envío de telemetría es activado desde la estación terrena, por alguno de los siguientes métodos, dependiendo del modo de funcionamiento del *software* de vuelo:

- **Modo normal:** la telemetría se descarga bajo demanda, como resultado de la ejecución de un telecomando, lo que permite seleccionar el tipo de datos que se desea obtener.
- **Modo de respaldo:** si no se reciben telecomandos durante un periodo de tiempo determinado, por ejemplo 24 horas, se asume una falla en el sistema de comunicaciones. En este modo, para ordenar una transmisión de telemetría, basta con enviar la señal de la portadora durante una cantidad de tiempo determinada, por ejemplo, 10 segundos. No se puede seleccionar el tipo de telemetría a descargar, sino que se envían todos los datos recolectados hasta el momento.

Durante la prueba sólo estuvo disponible el modo de respaldo, por lo que se transmite la portadora durante el tiempo especificado en la configuración de la aplicación, en este caso, 10 segundos. El satélite detecta la señal y el *software* determina si estuvo presente el tiempo necesario, como se detalla en el registro ??.

Listing 5.6: Descarga de telemetría en modo de respaldo

```
//Se detecta la portadora y se inicia el conteo
[19:37:55.076357] RSSI_CNT=2
[19:37:57.480979] RSSI_CNT=4
[19:37:59.055074] RSSI_CNT=6
[19:38:01.055657] RSSI_CNT=8
[19:38:03.063135] RSSI_CNT=10
```

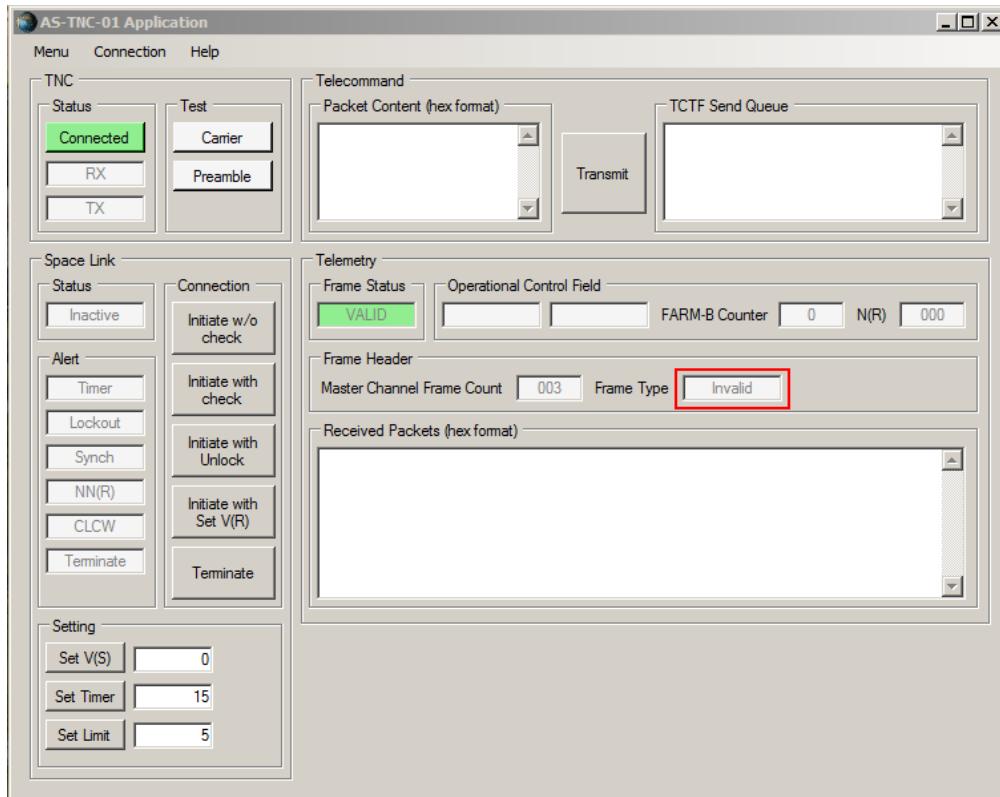


Figura 5.6: Recepción de telemetría en la estación terrena. El *software* debería mostrar el frame descargado en formato hexadecimal, sin embargo, lo identifica como inválido debido a serias fallas en el equipo

```

[19:38:05.055308] RSSI_CNT=12

//Luego 12 segundos, se genera un comando para enviar TM
[19:38:07.107302] com_doOnRSSI..
[19:38:07.116383] [Dispatcher] Orig: 0x1102 | Cmd: 0x8003 | Param: 2

//Primer frame de telemetría
[19:38:07.191023] >> Sending START_WRITE_TM command... [OK]
[19:38:08.990042] >> Sending START_SEND_TM command...
[19:38:09.769988] >> Telemetry transmitted [OK]

//Segundo frame de telemetría
[19:38:09.790260] >> Sending START_WRITE_TM command... [OK]
[19:38:12.316259] >> Sending START_SEND_TM command...
[19:38:13.104915] >> Telemetry transmitted [OK]

```

De esta manera se induce el envío de toda la telemetría almacenada, que corresponde al valor de las variables de estado, registradas cada 20 minutos. En la estación terrena se reciben los datos decodificados por el TNC y los resultados se detallan en la figura 5.6.

Se observa que, a pesar de las fallas en el sistema de comunicaciones, el *software* de vuelo puede efectuar la descarga de telemetría operando en modo de respaldo. Al igual que el *beacon*, la señal es recibida, pero los datos no son decodificados correctamente y el programa indica este error.

Control central

Organizar telemetría: durante la misión, se generarán datos que provienen de diferentes subsistemas o *payloads*. Se debe contar con un medio de almacenamiento con la capacidad adecuada para mantenerlos y un sistema de organización de los diferentes datos guardados, con el objetivo de ser requeridos de manera selectiva para ser enviados como telemetría a la estación terrena. En esta etapa del desarrollo el repositorio de datos cuenta con una serie de *buffers* para este propósito.

Plan de vuelo: la ejecución del plan de vuelo está a cargo del *listener* denominado `taskFlightPlan` el cual, de manera periódica, consulta la lista de comandos manejada por el repositorio de datos y determina el siguiente que se debe ejecutar.

El plan de vuelo, consiste en una lista de comandos a realizar en un periodo de 24 horas, con espacio de 10 minutos entre cada uno (configurable). El módulo `taskFlightPlan` realiza una correspondencia entre la hora actual del sistema y el índice a leer desde el plan de vuelo. La lista de comandos se configura al inicio del sistema, según sea necesario, y para efectos de esta prueba se genera el itinerario mostrado en la tabla 5.5. Este consiste en ejecutar dos comandos de manera intercalada: `rtc_print`, que muestra en consola la hora del sistema, para hacer evidente el funcionamiento de este módulo; y el comando `pay_FSM_default` cuyo objetivo es recolectar las variables de estado, a modo de telemetría, cada 20 minutos.

Tabla 5.5: Plan de vuelo

Hora	Comando	Código	Parámetro
00:00	rtc_print	0x7007	0
00:10	pay_FSM_default	0x6001	dat_pay_tmEstado
00:20	rtc_print	0x7007	0
00:30	pay_FSM_default	0x6001	dat_pay_tmEstado
...

Para la verificación de este requerimiento, se revisan los registros de la consola, que cuentan con una estampa de tiempo, para determinar la frecuencia y momento de ejecución de los comandos producidos por el plan de vuelo. En el registro 5.7 se observa el resultado de ejecutar el comando `rtc_print` desde el plan de vuelo.

Listing 5.7: Comandos generados por el plan de vuelo

```
//El origen del comando es 0x1103, que indentifica al plan de vuelo
//Como resultado del comando rtc_print se muestra la hora del sistema
[17:40:06.811852] [Dispatcher] Orig: 0x1103 | Cmd: 0x7007 | Param: 0
[17:40:06.837689] >>[29/8/13 - 17:40:14]

//El siguiente comando se genera 10 minutos después: pay_FSM_default,
//pero se ha omitido su salida de debug. Exactamente a los 20 minutos
//se vuelve a ejecutar el comando rtc_print desde el plan de vuelo.
[18:00:06.914741] [Dispatcher] Orig: 0x1103 | Cmd: 0x7007 | Param: 0
[18:00:06.969861] >>[29/8/13 - 18:00:14]

//Luego de cuarenta minutos, se vuelve a ejecutar el plan de vuelo
[18:20:06.919318] [Dispatcher] Orig: 0x1103 | Cmd: 0x7007 | Param: 0
[18:20:06.952946] >>[29/8/13 - 18:20:14]

[18:40:07.004340] [Dispatcher] Orig: 0x1103 | Cmd: 0x7007 | Param: 0
[18:40:07.036649] >>[29/8/13 - 18:40:14]
```

El registro obtenido en la consola muestra cómo, de manera exacta, cada 20 minutos se ejecuta el comando 0x7007 equivalente a `rtc_print`, proveniente del módulo 0x1103 que identifica a `taskFlightPlan`, y como resultado se muestra en pantalla la hora leída desde el RTC del satélite. Lo anterior corresponde con el comportamiento esperado.

En la figura 5.7, se ha graficado toda la serie de comandos ejecutados durante la prueba y que provienen de este módulo con lo cual se comprueba la ejecución precisa y periódica del plan de vuelo configurado.

De este modo se comprueba el correcto funcionamiento de este módulo, el cual ofrece una manera flexible y configurable de programar la ejecución de comandos en momentos específicos de la órbita.

Obtener el estado del sistema: en el módulo `taskHousekeeping`, las variables de estado son actualizadas cada 20 segundos, a través del comando `drp_updateAll_dat_CubesatVar`.

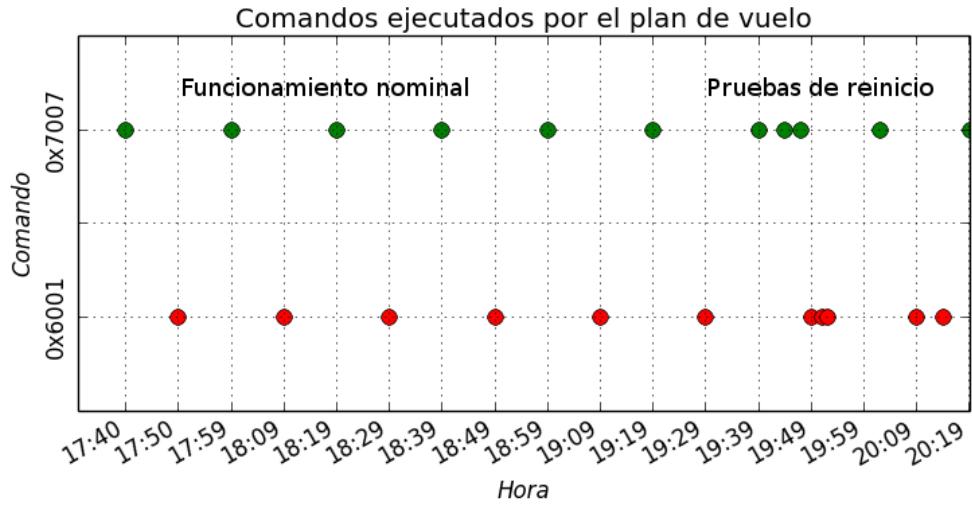


Figura 5.7: Gráfico de funcionamiento del plan de vuelo. Se observan los dos comandos programados ejecutándose cada 10 minutos. Cerca de las 19:39 hrs. se produce un reinicio del sistema y como respuesta a la falla, el plan de vuelo ejecuta el comando pendiente.

Asimismo, su valor se despliega a través de la consola serial a cada minuto para obtener un registro de las variables. Esta información es de vital importancia para los desarrolladores y operadores del satélite, debido a que contiene todos los datos relevantes sobre el estado de funcionamiento del sistema. Un ejemplo de esta información se muestra en el registro 5.8. Para comprobar la correcta actualización de las variables de estado durante el funcionamiento del satélite, se ha tomado un set de ellas y se ha graficado su valor en el tiempo, lo cual se muestra en la figura 5.8.

El set de variables consideradas en el gráfico 5.8, tiene por objetivo mostrar el funcionamiento del repositorio de estados, más que un análisis exhaustivo de sus valores instantáneos. Los resultados muestran cómo este repositorio permite mantener actualizado el estado de funcionamiento del satélite, retroalimentando al resto de los módulos de *software* con esta información. Acá se almacenan variables con información instantánea, como el caso de los voltajes y corrientes en el sistema de energía, o el nivel RSSI presente el sistema de comunicaciones. Otras son modificadas de manera periódica como la hora actual del sistema o el tiempo total de operación. Algunas variables representan configuraciones del satélite y no varían, a menos que se instruya una modificación manual de ellas, como el caso de las potencias de transmisión de telemetría y *beacon*.

Es muy importante que ciertas variables mantengan su valor entre cada reinicio del sistema, para que el software de vuelo retome su estado de operación anterior a una falla o *reset*. Esto se verifica en las pruebas de reinicio, revisando el valor de las siguientes variables: `ppc_lastResetSource`, `ppc_hoursAlive`, `ppc_hoursWithoutReset`, `ppc_resetCounter` y `dep_ant_deployed`. La información se resume en el gráfico de la figura 5.9. Se observa que ciertas variables se van actualizando con el tiempo, como el caso de las horas de funcionamiento y las horas sin reinicio. Las pruebas de *reset* comienzan a las 19:44 hrs. lo cual se refleja en el valor de las variables `ppc_hoursWithoutReset` y `ppc_resetCounter`, que indican la cantidad de horas que el sistema ha funcionado sin reiniciarse y el contador de estos

Listing 5.8: Comandos generados por el plan de vuelo

```
=====
Contenido de CubestatVar:
=====

ppc_opMode= 1           //Modo de operación
ppc_lastResetSource= 4  //Origen de último reinicio
ppc_hoursAlive= 0       //Total de horas de funcionamiento
ppc_hoursWithoutReset= 0 //Horas sin reinicio
ppc_resetCounter= 1     //Contador de reinicios
ppc_enwdt= 1            //Estado del watchdog
ppc_osc= 3              //Estado del oscilador
ppc_MB_nOE_USB_nINT_stat= 0 //Estado del selector del USB
ppc_MB_nOE_MHX_stat= 1   //Estado del selector del TRX
ppc_MB_nON_MHX_stat= 0   //Estado del selector de voltaje del TRX
ppc_MB_nON_SD_stat= 0    //Estado del selector de la tarjeta SD
-----
dep_ant_deployed= 1     //Estado de despliegue de las antenas
dep_ant_tries= 3        //Intentos de despliegue de las antenas
-----
rtc_year= 13             //Fecha del sistema, año
rtc_month= 8              //Fecha del sistema, mes
rtc_week_day= 6            //Fecha del sistema, dia
rtc_day_number= 23         //Fecha del sistema, dia de la semana
rtc_hours= 20              //Hora del sistema, horas
rtc_minutes= 35             //Hora del sistema, minutos
rtc_seconds= 19             //Hora del sistema, segundos
-----
eps_bat0_voltage= 172     //Voltaje de la batería (ADC)
eps_bat0_current= 902      //Corriente de la batería (ADC)
eps_bus5V_current= 1023    //Corriente en bus 5V (ADC)
eps_bus3V_current= 1023    //Corriente en bus 3V (ADC)
eps_bat0_temp= 559          //Temperatura de la batería (ADC)
eps_panel_pwr= 0            //Potencia del panel solar (ADC)
eps_status= -16384          //Estado de la EPS (ADC)
eps_soc= 8                 //Nivel de carga de la batería
eps_socss= 8                //Estimador de la carga de la batería
eps_state_flag= 2            //Estado del estimador de carga
eps_charging= 1              //Estado de carga de la batería
-----
trx_frec_tx= -14510        //Frecuencia de TX (ADC)
trx_frec_rx= -16448        //Frecuencia de RX (ADC)
trx_opmode= 16               //Modo de operación del TRX (ADC)
trx_temp_hpa= 2164          //Temperatura del HPA (ADC)
trx_temp_mcu= 1150           //Temperatura MCU del TRX (ADC)
trx_rssi= -53 dBm            //Nivel de señal (dBm)
trx_bacon_pwr= 1              //Potencia del bacon (ADC)
trx_telemetry_pwr= 1          //Potencia de la telemetría (ADC)
trx_status_tc= 0              //Estado de recepción de comandos
trx_count_tm= 0                //Contador de telemetría
trx_count_tc= 0                //Contador de telecomandos
trx_lastcmd_day= -1             //Días desde el último telecomando
-----
```

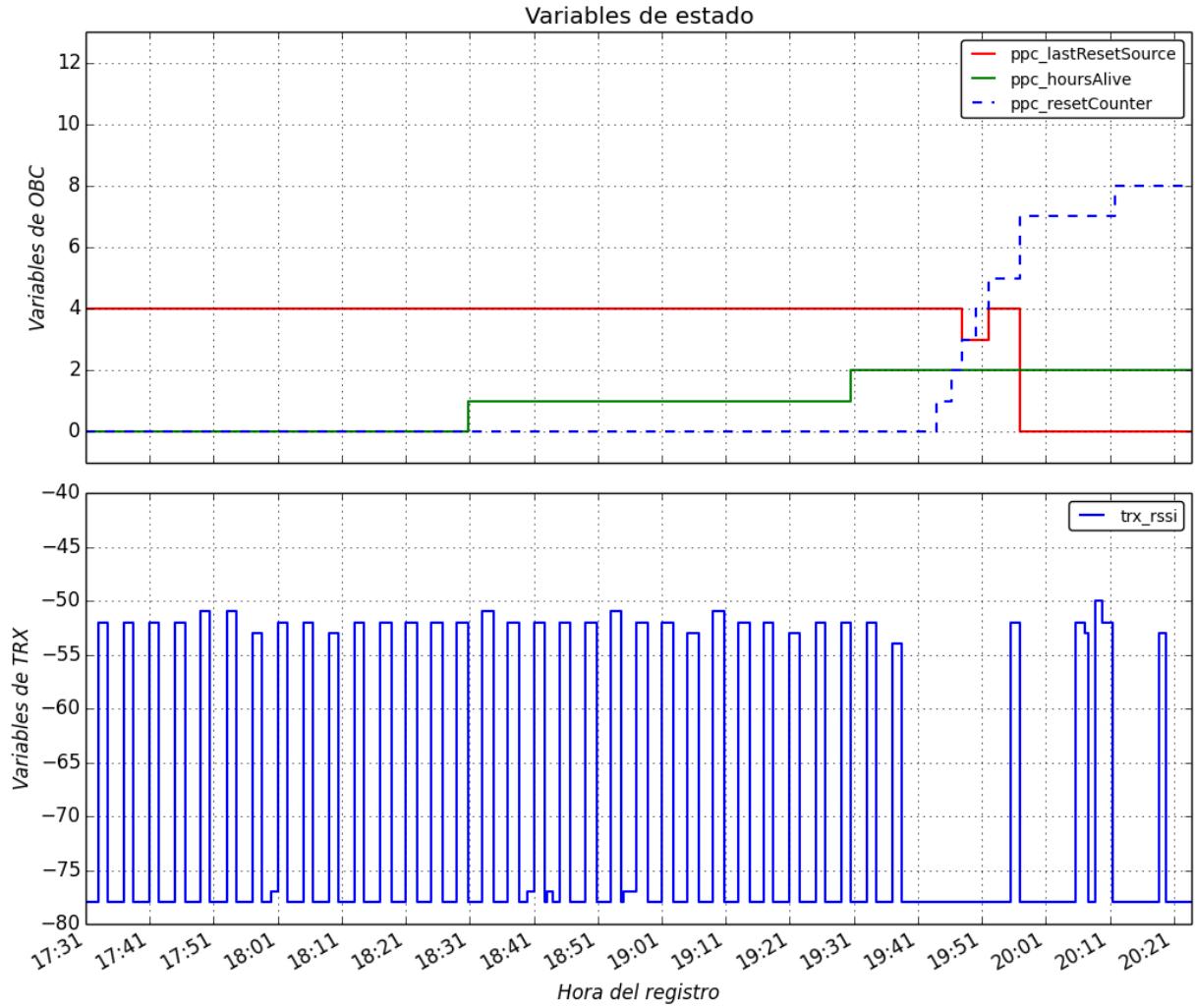


Figura 5.8: Gráfico con el valor de las variables de estado en el tiempo. Arriba, variables relacionadas con el computador a bordo, las horas de funcionamiento aumentan con el tiempo y durante pruebas de reinicio aumenta el contador y el indicador del tipo de reinicio. Abajo, el valor de la intensidad de señal (RSSI) del *transceiver*, los máximos coinciden con las transmisiones de *beacon*.

eventos, respectivamente. La prueba incluye reinicios generados por diferentes fuentes, como *software*, *hardware* y el *watchdog*, las cuales se observan en la variable `ppc_lastResetSource`. La prueba del buen funcionamiento del sistema, ante estos eventos, es que los valores de las variables de estado siguen siendo coherentes cada vez que el sistema se vuelve a iniciar, como se ve en el caso del indicador de antenas desplegadas (`dep_ant_deployed`), el contador de reinicios y la cantidad total de horas de funcionamiento.

Inicio del sistema: el inicio del sistema cuenta de dos etapas. En la primera, se inician funciones de bajo nivel como configuraciones de periféricos del procesador, puertos de entrada-salida, instanciar variables del sistema operativo e iniciararlo. De la correcta ejecución de esta etapa depende la posibilidad de contar con la salida para la consola serial, por lo que se comprueba su funcionamiento a través de la salida de depuración que se encuentra en el

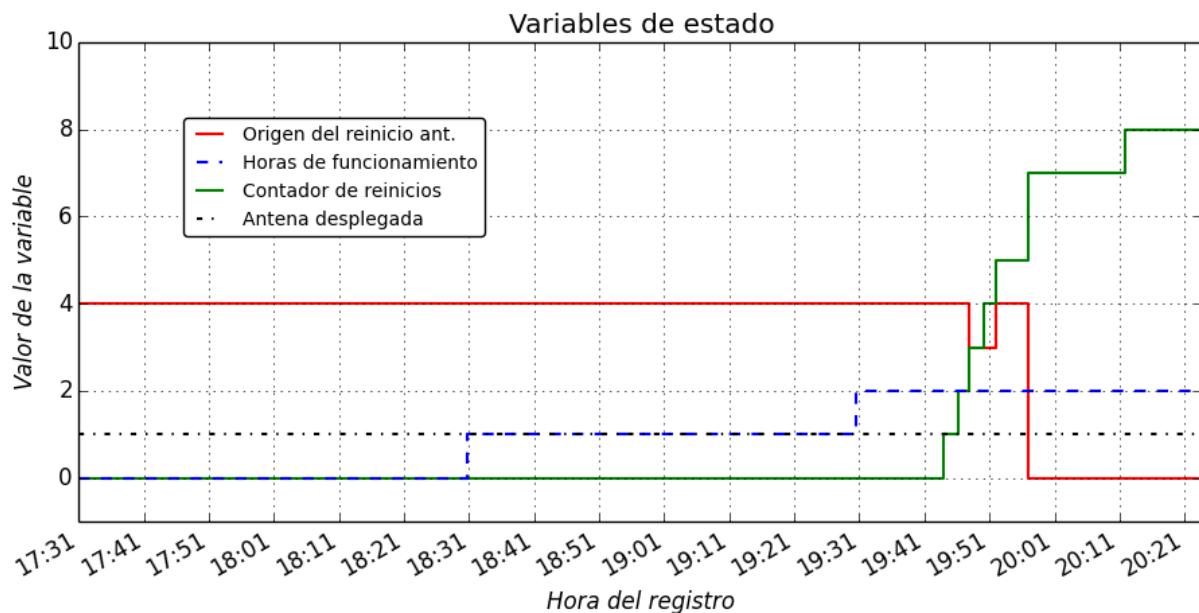


Figura 5.9: Variables de estado ante un reinicio. El repositorio de estados mantiene el valor de las variables consistentes ante fallas o reinicios, como se observa en el indicador de antena desplegada y el contador de reinicios.

registro 5.9.

Listing 5.9: Secuencia de inicio de bajo nivel

```
//La salida serial es la primera en configurarse
----- SUCHAI BOOT SEQUENCE -----

//Configuraciones de perifericos
Initializing satelite bus
    * Microcontroller IO pins
    * CubesatKit MB
    * PC104 Bus

//Configuraciones propias del microcontrolador
Initilizing Microcontroller
    * Reset status
    * Setting oscillator
    * mPWRMGNT_Regulator_ActiveInSleep
    * Enabling WDT

//Se inicia el scheduler de FreeRTOS. A partir ahora el
//sistema opeartivo toma el control del procesador
Starting FreeRTOS [->]

//Se inician las tareas principales
>>[Executer] Started
>>[Dispatcher] Started
>>[Deployment] Started
```

Desde este momento comienza la segunda etapa de inicialización, implementada en `taskDeployment`. En este punto ya se cuenta con todas las funcionalidades de bajo nivel y el sistema operativo se encuentra funcionando. Así, se procede al inicio de los servicios de la capa de aplicación, incluyendo: periféricos externos, repositorios, estructuras de datos y el resto de los *listeners*. En este momento se ejecuta, si corresponde, el periodo de inactividad, así como el despliegue de antenas.

La salida por consola, que se muestra en el registro 5.10, es incremental a cada función ejecutada y demuestra la correcta operación de cada etapa de la secuencia de inicio. También se demuestra la capacidad del *software* para recordar el estado anterior a su reinicio, siendo capaz de determinar si es la primera vez que se inicia y en tal caso, ejecutar el periodo de inactividad junto al despliegue de antenas.

Listing 5.10: Secuencia de inicio de alto nivel

```
//Cada linea representa la correcta ejecución de la acción,
//en caso de error se indicaría en la misma consola.

[Deployment] INITIALIZING SUCHAI FLIGHT SOFTWARE

//Se inicializan perifericos externos
[dep_init_Peripherals] Initializig external pheripherals...
    * External RTC
    * EEPROM Memories
    * SD Memory [delay]
    * SD Memory [init OK]
    * SD Memory [onReset] dat_onReset_memSD()..
    Static structures..
    DAT_Payload()..
    DAT_GnrlPurpBuff().. 

//Se inicializan todos los repositorios de datos y comandos
[dep_init_Repos] Initializing data repositories...
    * FlightPlan
    * Telecommands buffer
    * Payloads status rep.
    * Status rep. //Se aumenta contador de reset
        - LastResetSouce: SOFTWARE_Reset
        - NOT the First time on! resetCounter++
        - resetCounter = 6
    * Commands rep.

[dep_init_GnrlStruct] Initializing other structures...
    * init EPS structs

//Comienza el periodo de inactividad
[dep_silent_time] Mandatory inactivity time...
    Satellite launched. Antenna deployed
    //Como no es el primer inicio, no se realiza inactividad.
    FINISHING SILENT TIME

//Se configura el sistema de comunicaciones
[dep_deploy_antenna] Deploying TRX Antenna...
    * Antenna is already deployed
    * Setting TRX
```

```

Setting beacon: SUCHAIATINGDOTUCHILEDOTCL -
>> TRX Setting BEACON:
    Setting Offset...
    Setting Content...
    Setting Beacon Length... 26

//Antes de terminar se deben iniciar todos los demás listeners
[dep_launch_tasks] Starting all tasks...
    * Creating taskConsole
    * Creating taskHousekeeping
    * Creating taskCommunications
    * Creating taskFlightPlan
[Deployment] ENDS
[Deployment] Deleting task

```

Al final de esta secuencia, se inician el resto de los *listeners* que comienzan su ejecución en segundo plano. El éxito de la operación se logra al visualizar el *prompt* de la consola serial, indicando que el satélite está preparado recibir comandos y registrar los sucesos del *software* de vuelo, lo cual se muestra en el registro 5.11

Listing 5.11: Pantalla de bienvenida de la consola serial del satélite

```

>>[Console] Started

//La pantalla de bienvenida indica el fin de la secuencia de inicio
-----
   /---| | | | /---| ||| /_ \ | _ |
  \__ \|_| | | (---| _ _ | / - \ | | |
   |___/\---/ \---|_| |/_/ \_\---|
-----
//Comienzan a funcionar los listeners
>>[Communications] Started
>>[FlightPlan] Started
>>[Houskeeping] Started

>>

```

Área de energía

Estimación de la carga de la batería: este función es parte del controlador de la EPS. La estimación se actualiza de manera periódica, como parte de las tareas del *listener taskHousekeeping*, cada 20 segundos. El parámetro denominado *state of charge* (SOC), es una variable de estado del sistema y se calcula a partir de los datos de consumo de energía, temperatura y voltaje que reporta la EPS. Esta variable es utilizada para determinar si un comando se puede ejecutar o debe ser rechazado. Durante la prueba se registraron los valores de corriente, voltaje y temperatura de la EPS así como el nivel de SOC calculado. La figura 5.10 muestra los resultados obtenidos, así como la estimación del SOC a *posteriori*, para comprobar la operación realizada en el computador a bordo.

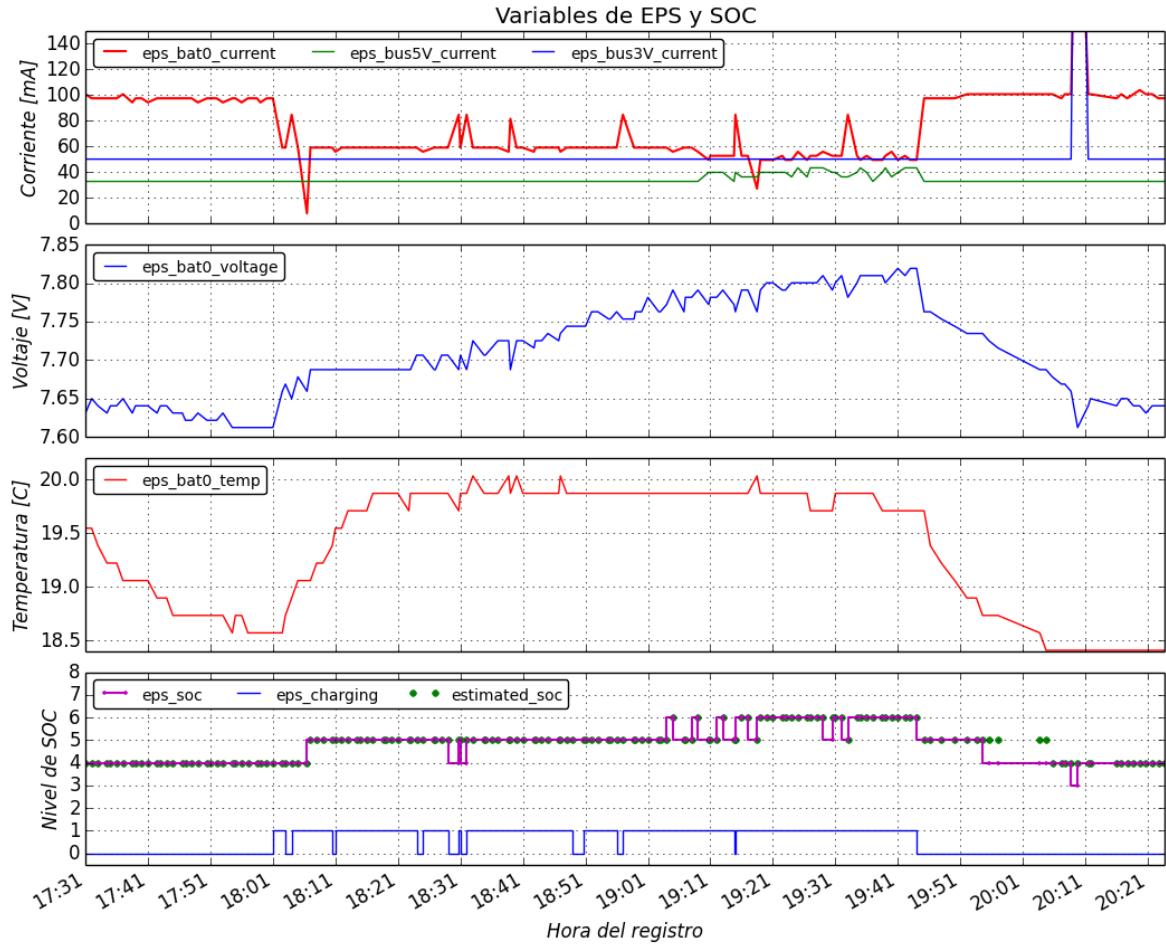


Figura 5.10: Gráfico con variables de EPS y estimación de SOC. El sistema de energía se comporta según lo esperado, según lo indicado por las variables de corriente y voltaje. Se observa una correspondencia entre la variable *eps_charging* y el voltaje de las baterías. El nivel de SOC estimado por el *software* de vuelo corresponde a los calculados a *posteriori*.

A partir de los datos del gráfico 5.10, se verifica el correcto funcionamiento del sistema de energía. Los valores de corriente leídos en la EPS son normales: al rededor de 30mA para el bus de 3.3V, al que se conecta el computador a bordo, y 50mA para el bus de 5.0V, donde se encuentra el *transceiver*. Las baterías se cargan en cuanto se conecta el puerto USB, lo que es indicado por la variable *eps_charging* y se refleja en el voltaje leído. También se verifica el requerimiento de estimar la carga presente en las baterías, a través de la variable *eps_soc* y contrastando estos datos con valores calculados a *posteriori*, según los datos de voltaje y corriente. Cabe destacar que la diferencia en la estimación del SOC, observada en los registros de las 20:01 hrs. es normal, producto de una corrección según los datos de temperatura, realizada por el *software* de vuelo.

Presupuesto de energía: el nivel de SOC, como variable de estado, indicará la energía que se dispone para ejecutar comandos en el sistema. El encargado de velar por el cumplimiento del presupuesto energético es el módulo *Dispatcher*, quien realiza una comparación entre la energía requerida por un comando y la disponible en ese momento.

Para verificar el requerimiento de contar con un presupuesto de energía, que guía las operaciones del satélite, se lleva a cabo el test de SOC. Para esto, de manera simulada, se cambian los requerimientos energéticos de una serie de comandos, a niveles mayores que la energía disponible, para luego intentar su ejecución. El comando correspondiente es el 0x8006 o `tcm_set_sysreq`, que cambia los requerimientos energéticos de todas las funciones del grupo TCM, encargadas de generar telemetría y *beacons*. En el registro 5.12 se observa la ejecución de este comando, cambiando los requerimientos a nivel 10, cuando se dispone de un SOC de 4. Así cuando se intenta ejecutar un comando del grupo 0x8000 estos son rechazados por el *Dispatcher* evitando su ejecución.

Listing 5.12: Registro de la prueba de nivel SOC bajo

```
//Se elevan los requerimientos del grupo TCM a un SOC de 10
[20:17:59.701281] SOC= 4
[20:18:10.788012] exe_cmd 0x8006 10
[20:18:10.823338] [Console] Se genera comando: 0x8006
[20:18:10.860289] [Dispatcher] Orig: 0x1101 | Cmd: 0x8006 | Param: 10
[20:18:10.890811] [Dispatcher] CMD Result: 1

//Se intenta ejecutar un comando del grupo TCM
[20:19:10.667124] exe_cmd 0x8000 0
[20:19:10.825339] [Console] Se genera comando: 0x8000

//El dispatcher rechaza el comando, por no cumplir los requerimientos de
//energía
[20:19:10.864141] [Dispatcher] Command: 0x8000 | From: 0x1101 Refused
because of SOC

//Otros comandos, con requerimientos de SOC bajos, se ejecutan normalmente
[20:19:11.197873] [Dispatcher] Orig: 0x1102 | Cmd: 0x3007 | Param: 0
[20:19:11.231369] [Dispatcher] CMD Result: 1
```

Esta situación no solo ocurre bajo condiciones de prueba, puede ocurrir que tras algunas operaciones que demandan grandes cantidades de energía, el nivel de SOC disminuya, quedando bajo el umbral que requieren los comandos. Este es el caso de las transmisiones de telemetría y *beacons*, como se observa en el registro 5.13, donde luego de dos envíos de telemetría el nivel de SOC baja a 3, causando que una posterior solicitud de descarga de datos sea rechazada.

Listing 5.13: Comando rechazado debido a un nivel de SOC bajo

```
//Se producen la transmisión de dos frames de telemetría sucesivos.
[20:08:56.149539] com_doOnRSSI..
[20:08:56.171706] [Dispatcher] Orig: 0x1102 | Cmd: 0x8003 | Param: 2
[20:08:56.221945] >> Sending START_WRITE_TM command...
[20:08:58.031927] >> Sending START_SEND_TM command...
[20:08:58.798705] >> Telemetry transmitted [OK]

[20:08:58.854076] >> Sending START_WRITE_TM command...
[20:09:01.352976] >> Sending START_SEND_TM command...
[20:09:02.133597] >> Telemetry transmitted [OK]

//Las baterías se descargan y el nivel de SOC baja a 3.
```

```

[20:09:02.414715] [Dispatcher] Orig: 0x1104 | Cmd: 0x5000 | Param: 0
[20:09:02.429875] drp_updateAll_dat_CubesatVar()..
[20:09:02.710113] SOC= 3

//Una posterior solicitud de telemetría es rechazada, como es de esperar,
//debido a que no hay energía suficiente para realizar la transmisión.
[20:09:06.121742] RSSI_CNT=2
[20:09:08.138337] RSSI_CNT=4
[20:09:10.146584] RSSI_CNT=6
[20:09:12.197035] RSSI_CNT=8
[20:09:14.152838] RSSI_CNT=10
[20:09:16.140583] RSSI_CNT=12
[20:09:18.140335] com_doOnRSSI..
[20:09:18.158957] [Dispatcher] Command: 0x8003 | From: 0x1102 Refused
because of SOC

```

La tabla 5.6 resume los resultados de la prueba de SOC. Se observa que el control realizado por el módulo Dispatcher aplica efectivamente las indicaciones del presupuesto de energía planificado, el cual se expresa en el parámetro SOC y la cantidad de energía que requiere cada comando. Durante la prueba todos los comandos que requieren al menos el nivel de energía disponible fueron ejecutados mientras que aquellos con requerimientos superiores fueron rechazados, validando el requerimiento.

Tabla 5.6: Resultados para test SOC

Hora	Comando	Nombre	SysReq	SOC	Resultado
20:18:10	0x8006	tcm_set_sysreq	1	4	Ejecutado
20:18:19	0x5000	drp_updateAll_dat_CubesatVar	1	4	Ejecutado
20:18:33	0x300C	trx_tm_trxstatus	4	4	Ejecutado
20:19:10	0x8000	tcm_testframe	10	4	Rechazado
20:19:13	0x8000	tcm_testframe	10	4	Rechazado
20:19:19	0x8002	tcm_send_beacon	10	4	Rechazado
20:19:19	0x5000	drp_updateAll_dat_CubesatVar	1	4	Ejecutado
20:19:45	0x8003	tcm_sendTM_cubesatVar	10	4	Rechazado
20:20:03	0x8003	tcm_sendTM_cubesatVar	10	4	Rechazado

El *power budget* se aplica a diferentes niveles dentro del sistema que representa el satélite. La parte que corresponde al *software* de vuelo, tiene relación con la estimación de la cantidad de energía disponible en base a los parámetros que entrega la EPS, y con evitar la ejecución de comandos que requieren más recursos energéticos que los disponibles. El resto de la planificación energética queda por parte del equipo de operación del sistema, en lo que refiere a determinar la cantidad de energía correcta para cada comando, la frecuencia de ejecución de aquellos de alto consumo, como el envío de *beacon*, o la correcta programación del plan de vuelo.

Órbita

Actualizar parámetros de órbita: en esta versión del *software* de vuelo no se contemplan métodos de estimación de órbita y se relega este requerimiento al operador del satélite. En tierra, se puede realizar la propagación de órbita, para así obtener una relación entre el plan de vuelo del satélite y su posición orbital, programando las operaciones requeridas en determinados momentos.

Payloads

Ejecución de comandos de *payloads* : en esta versión del *software* de vuelo, dada las características de los *payloads* que van a bordo, la capacidad de ejecutar determinadas acciones de *payloads* puede ser completamente programada a través del plan de vuelo, si se proveen los comandos pertinentes a cada uno. No obstante, la integración a nivel de *payloads* del satélite y sus resultados, se puede encontrar en la memoria *Proceso de diseño, arquitectura y testeo del nano-satélite SUCHAI [?]*.

Capítulo 6

Conclusiones

6.1. Conclusiones generales

Al definir globalmente la aplicación como una arquitectura de tres capas se ha logrado separar efectivamente los diferentes niveles de abstracción del sistema así como una forma de separar el trabajo desarrollado en etapas claramente definidas. Así durante el proceso de implementación el equipo pudo de manera temprana comenzar el trabajo de implementación de controladores y a probar gradualmente la plataforma mientras se iban definiendo las soluciones adecuadas para las capas superiores. En la misma linea se observa que esta separación ha permitido integrar de manera clara una solución de terceros como lo es el sistema operativo con los desarrollos específicos del proyecto lo que también es prueba de las características de modularidad, modificabilidad y reusabilidad presentes en la solución y que se han considerado con una importancia alta en el diseño.

En la capa de sistema operativo se ha integrado con éxito una solución externa como lo es FreeRTOS que ha resultado ser una excelente opción para el proyecto al proveer una base sólida, estable y bien probada en uno de los aspectos críticos del sistema embebido. Lo anterior se suma a una gran simplicidad encontrada al momento de su implementación permitiendo obtener rápidamente un sistema funcional y de la misma manera extenderlo. Durante el desarrollo se encontró que FreeRTOS fue el módulo con la menor cantidad de fallas o problemas asociados y siempre que se presentaron dificultades se contó con la documentación adecuada para solucionarlos.

En lo que se refiere a la capa de aplicación del software de vuelo el gran valor del proyecto es haber analizado y adaptado el patrón de diseño *command pattern*. Según lo analizado en el proceso de diseño este patrón permite implementar todos los requerimientos operaciones lo cual se ha demostrado en las pruebas realizadas. En linea con la filosofía del uso de patrones de diseño se concluye que esta metodología de trabajo ha disminuido considerablemente los riesgos asociados al desarrollo de una aplicación por los siguientes motivos: los patrones de diseño son esquemas bien probados para resolver cierta clase de problemas; están claramente documentados; proveen un lenguaje común para entender la forma de funcionamiento del software facilitando los procesos de pruebas, verificación y modificación; y permiten de forma

directa cumplir con los requerimientos no operacionales más importantes.

Esta capa ha demostrado ser lo suficientemente general y flexible para que a partir de lo expuesto en la sección de implementación del sistema base, se implementaran la totalidad de las funcionalidades requeridas a través de los dos métodos propuestos: agregar *listeners* y agregar comandos.

La base de la aplicación como un ejecutor de comandos brinda una forma homogénea de extenderla mediante la programación de nuevos comandos que implementan las funcionalidades deseadas. Esta es una forma segura de programar funcionalidades al satélite en cuanto el desarrollo de un nuevo comando es un proceso aislado del resto de la aplicación y no interfiere con el resto de las funcionalidades.

Al agregar *listeners* se ha incorporado inteligencia al *software* de vuelo para permitir la generación de comandos de manera autónoma según diferentes estrategias de control así como bajo demanda bajo dos principales interfaces de comunicación: la consola de depuración por interfaz serial y el equipo de telecomunicación para operación desde tierra. Los *listeners* considerados en este trabajo permiten implementar los requerimientos operacionales del proyecto en cuestión, sin embargo agregar nuevos también es una manera segura y práctica para cambiar el comportamiento del *software* según los requisitos de la misión porque se puede agregar a la base actual sin modificarla, funcionando en paralelo, y se pueden desactivar fácilmente.

Se ha seguido la linea planteada por los requerimientos operacionales y no operacionales a lo largo del proceso de diseño y de pruebas produciendo una apropiada validación de la solución implementada. Esto enfatiza en la correcta decisión de definir adecuadamente los requerimientos del proyecto, como un trabajo conjunto del grupo de trabajo, antes de proveer cualquier idea de solución a priori

La arquitectura base del sistema ha sido licenciada según los términos de la GPL y su código puesto a disposición en repositorios públicos. De esta manera el trabajo desarrollado se transforma en un aporte a la comunidad de software libre esperando ser un aporte para futuros proyectos aeroespaciales o sistemas embebidos en general. El uso de una licencia GPL, a diferencia de otras licencias más permisivas, asegura que el código desarrollado se mantenga abierto en miras de la posibilidad de colaboración en la mejora y extensión de este trabajo.

Se concluye entonces que los objetivos del trabajo se han cumplido satisfactoriamente en cuanto:

1. Se ha diseñado una arquitectura de *software* clara que soluciona el problema planteado, la cual es independiente de la implementación y se basa en patrones de diseño para cumplir los parámetros de calidad buscados.
2. Se han implementado controladores de *hardware* para obtener la interfaz hacia los diferentes módulos que componen el satélite, siendo la base para implementar las funcionalidades de alto nivel de la aplicación. En específico se han realizado pruebas de funcionamiento que implican el uso de periféricos de comunicación UART, I2C y SPI propios del microcontrolador así como el control de periféricos externos como lo son el

transceiver, RTC y EPS.

3. Se ha integrado un sistema operativo de tiempo real adecuado para su uso en sistemas embebidos como lo es FreeRTOS con excelentes resultados en su integración y funcionamiento.
4. Se implementa y prueba la arquitectura básica del *software* de vuelo a nivel de aplicación como una adaptación al patrón de diseño *command pattern* lo que ha sido la base para extender al sistema con funcionalidades específicas de la misión.
5. Se integran los sub-sistemas de comunicación y energía a través de la programación de *listeners* y comandos para su operación. Se comprueba su funcionamiento a través de las pruebas de integración.
6. Con el sistema integrado se realizan pruebas de funcionamiento según los lineamientos planteados por los requerimientos operacionales donde se ha puesto a prueba y verificado cada uno de ellos con resultados satisfactorios salvo por los relacionados con el sub-sistema de comunicaciones que ha presentado fallas a lo largo de todo el desarrollo del proyecto.

6.2. Limitaciones

No obstante el cumplimiento de los objetivos generales y específicos planteados, la solución desarrollada presenta algunas limitaciones:

- Al arquitectura de procesador de comandos provee una reducida re-alimentación de las acciones generadas por los *listeners* limitando las estrategias de control que se pueden implementar en estos módulos. Generar esta re-alimentación ha requerido agregar una serie de variables de estado que permiten a los *listeners* determinar si las acciones comandadas han surtido efecto o no. La desventaja radica en que las variables de estado representan en general la salud del sistema satelital y no necesariamente indicadores específicos del punto de ejecución de las operaciones de control que llevan a cabo los *listeners*.
- El sistema de comunicaciones, incluyendo a nivel de *software* de vuelo, no ha sido probado exhaustivamente debido a graves fallas en el *transceiver* que hasta la fecha cuenta el proyecto.
- La implementación ha limitado la cantidad de argumentos que reciben los comandos a sólo uno. Como no pueden recibir muchos argumentos que cambien su funcionamiento estos debe ser bastante específicos aumentando la cantidad de comandos disponibles en el sistema con el consiguiente aumento de complejidad en el desarrollo y operación de la aplicación. Esta limitante se ha hecho patente sólo en una cantidad limitada de comandos relacionados con la escritura de datos en posiciones específicas de alguna memoria y se puede sortear mediante la inclusión de una variable de estado que represente el primer parámetro y dividiendo el comando en dos: uno que setea el parámetro a través del repositorio de estados y otro que lee la variable y ejecuta la funcionalidad requerida.

6.3. Trabajo futuro

El trabajo desarrollado es una de las primeras aproximaciones en la materia para el país, por lo tanto se desprenden una serie de trabajos futuros:

- Agregar más funcionalidades en el área de tolerancia a fallos en el software del sistema, específicamente en torno al módulo *dispatcher*, entre las que se incluyen el registro de actividades, registro de comandos fallidos y lista negra de comandos que producen error en el sistema.
- Implementar la arquitectura de *thread pool* en el *executer* para permitir la ejecución de múltiples comandos mejorando el fluidez del sistema.
- Portar el *software* de vuelo, en especial la capa de controladores, a múltiples plataformas soportando una variedad de microcontroladores para hacer al proyecto más directo de aplicar en futuras misiones.
- Integrar un nuevo *transceiver* al software de vuelo dado que la solución actual ha resultado totalmente insatisfactoria por su funcionamiento anómalo. Además se recomienda integrar el protocolo CSP (*Cubesat Space Protocol*) en la capa de transporte entre tierra y el satélite e incluso entre sub-sistemas del mismo para avanzar en la unificación de las interfaces, protocolos y estándares en los proyectos aeroespaciales.