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

DISEÑO E IMPLEMENTACIÓN DEL SOFTWARE DE VUELO PARA SATÉLITES TIPO CUBESAT

TESIS PARA OPTAR AL TÍTULO DE TÍTULO DE INGENIERO CIVIL ELECTRICISTA

CARLOS EDUARDO GONZÁLEZ CORTÉS

PROFESOR GUÍA:
MARCOS DÍAZ QUEZADA

MIEMBROS DE LA COMISIÓN:
CLAUDIO ESTÉVEZ MONTERO
ALEX BECERRA SAAVEDRA

SANTIAGO DE CHILE
MAYO 2013

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1. Fundamentación y objetivos generales

Esta memoria se enmarca en el desarrollo del proyecto SUCHAI que consiste en la implementación, lanzamiento y operación de un pico-satélite Cubesat, siendo esta la primera aproximación en esta materia para la universidad y el país. Uno de los componentes fundamentales de un satélite es su computador abordo, sistema encargado de dar inteligencia y operatividad al satélite durante todo su tiempo de vida útil en el espacio. En el caso de un pico-satélite se tiene el desafío de dotar de todas la funcionalidades estándar de un satélite en un sistema computacional de recursos extremadamente limitados, estamos hablando de sistemas embebidos que utilizan microcontroladores de baja potencia y capacidad de cómputo como microcontroladores PIC24 o PIC18.

El objetivo de este trabajo es el diseño, desarrollo e implementación del *software* que gobierna el computador a bordo del satélite. Se requiere diseñar una arquitectura de *software* que abarque desde controladores de hardware hasta la aplicación final para el control de satélite. Esta arquitectura debe cumplir con requerimientos de calidad de *software* como modularidad, expansibilidad y facilidad de mantenimiento estando adaptada en específico a sistemas embebidos que emplean microcontroladores de gama media.

La implementación se llevará a cabo en específico para el satélite SUCHAI y busca proveer la funcionalidad básica de este sistema que incluye la interacción de un computador a bordo, un sistema de control de energía y un sistema de comunicaciones. De esta manera el *software* de control se cuenta como un recurso más que será considerado y adaptado a la necesidades específicas del proyecto en la etapa de integración general de sistemas del satélite.

1.2. Objetivos específicos

Los objetivos específicos del proyecto se enumeran a continuación

- Diseñar una arquitectura de *software* para el sistema de control del satélite
- Implementar controladores de hardware para el microcontrolador
- Implementar controladores de periféricos principales (Transceiver, EPS y RTC)
- Integrar un sistema operativo de tiempo real multitarea como sistema embebido
- Implementar el flujo principal de la arquitectura del *software* de control del satélite
- Integrar sistema de comunicaciones al *software* de control
- Integrar sistema de energía al *software* de control
- Pruebas del sistema integrado

El listado de objetivos presenta un orden temporal en la ejecución de estas tareas puesto que objetivo es dependiente del cumplimiento de los objetivos anteriores. 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 mínimas funcionalidades.

Capítulo 2

Contextualización

2.1. Marco Teórico

Se deben revisar los siguientes temas que son parte del contexto del proyecto realizado.

2.1.1. 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. Un sistema embebido está compuesto por uno o más microcontroladores pequeños que cuentan con periféricos para manejar diferentes protocolos de comunicación; conversores ADC; timers; puertos de entrada y salida digitales, todo en integrado en un mismo chip para guardar espacio y ahorrar 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 cual por lo general es específico para la plataforma de hardware y se relaciona a muy bajo nivel. A diferencia de un computador de propósito general donde el usuario puede cargar una serie de programas en él para un amplio rango de usos, el usuario de un sistema embebido no tiene la capacidad de reprogramarlo fuera de las posibilidades que el desarrollador ha brindado al sistema[**?**].

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

- Un sistema embebido se mantiene siempre funcionando y debe proveer respuesta 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 so-

bre ellas. Existen sistemas que son controlados por el usuario mediante una interfaz preparada para ellos, mientras que otros sistemas 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 amplia gama de dispositivos externos. Por esto se requiere desarrollar capas de drivers 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.

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 amplio 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 simultáneamente a las instrucciones del programa y a los datos de este alojados en RAM. La arquitectura de la CPU la completan una *Arithmetic Logic Unit* (ALU) con *hardware* dedicado para realizar multiplicaciones y divisiones. El detalle de la arquitectura del microcontrolador PIC24 se detalla en la figura 2.1. También posee un vector de hasta 128 interrupciones con capacidad para atender hasta 8 de ellas 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.

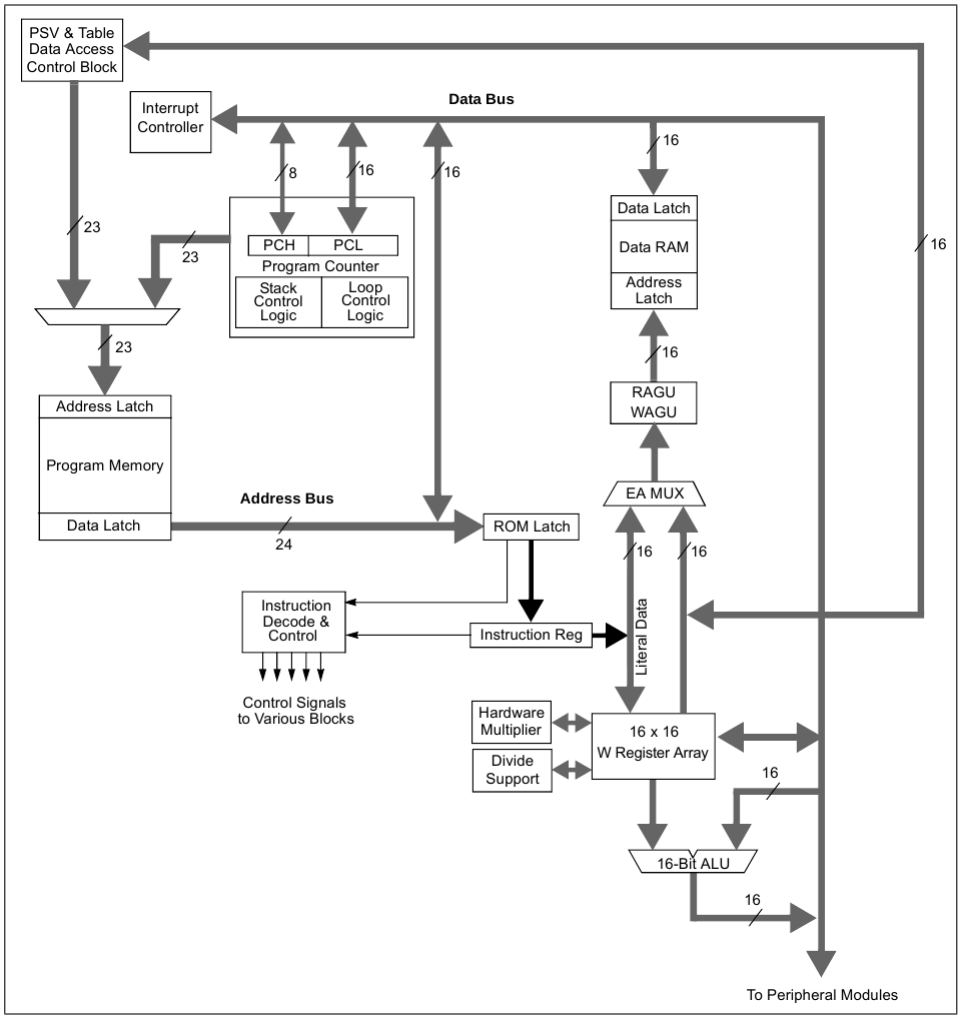


Figura 2.1: Arquitectura de la CPU del PIC24F

Periféricos La familia de microcontroladores PIC24F integra en el mismo chip una serie de periféricos que permiten realizar funciones específicas a través de hardware especialmente diseñado. Esto hace que el PIC24F se convierta en un sistema embebido capaz de ser utilizados

en aplicaciones que requieran: conversores análogos a digitales; temporizadores; comunicación síncronas y asíncronas 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. El acceso para configurar, guardar y obtener datos de estos periféricos se realiza a través de registros que están mapeados en la memoria de datos del microcontrolador por lo tanto comparten el bus de datos y no son necesarias instrucciones extras para su integración. Junto con una completa documentación de la arquitectura y funcionamiento de cada periférico, los compiladores de lenguaje C de Microchip proveen librerías para acceder a estas funcionalidades a través de una API de más alto nivel.

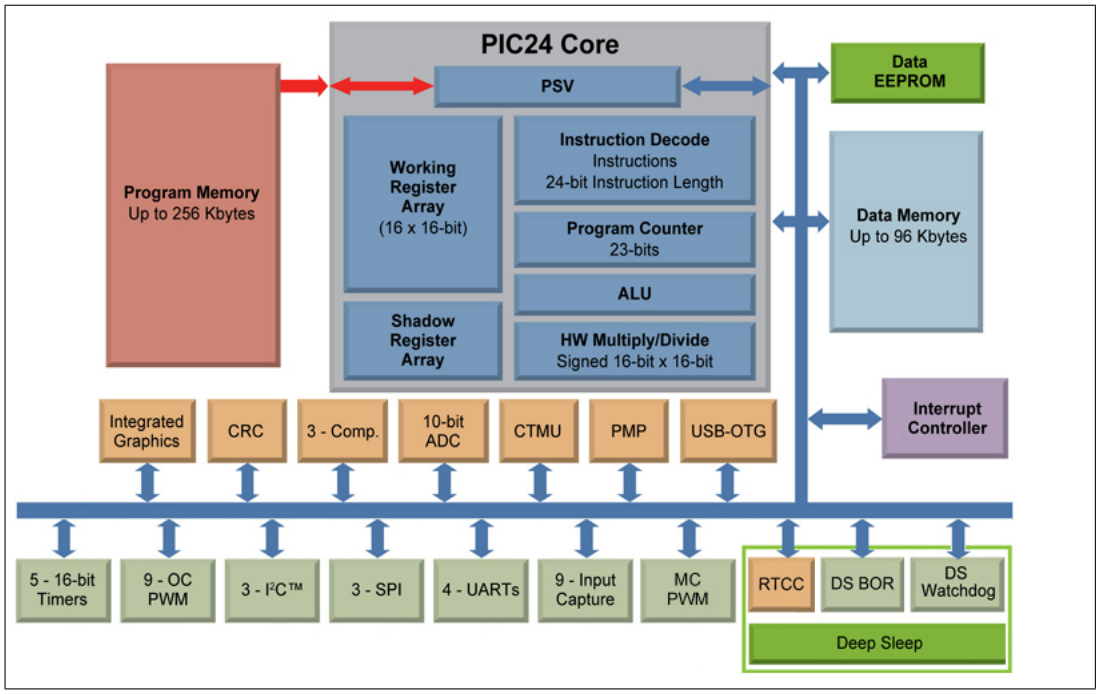


Figura 2.2: Arquitectura del PIC24F

Desarrollo

2.1.2. 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 que se ejecutan en el computador. El sistema operativo es la capa entre el hardware y las aplicaciones. El hardware puede variar considerablemente entre un sistema y otro, por eso 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. La figura 2.3 es un esquema que ilustra un sistema operativo como una capa de abstracción del hardware. Ejemplos de sistemas operativos los son UNIX, GNU/Linux, FreeRTOS, entre otros.

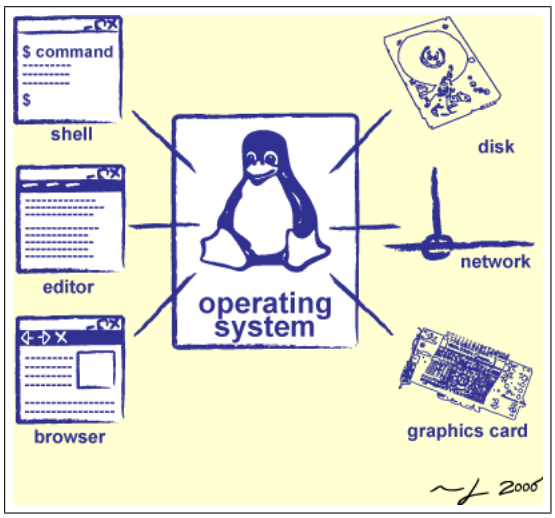


Figura 2.3: Sistema Operativo como capa de abstracción

Sistemas Operativos de Tiempo Real

Parte fundamental de un sistema operativo es su *scheduler*, módulo encargado de intercambiar entre las múltiples tareas que se deben ejecutar cediendo un espacio de tiempo para utilizar le procesador. La forma en que trabaja el *scheduler* define el tipo de sistema operativo que se posee. 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 ejecutara 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.

La figura 2.4 demuestra 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 precencia de ciertos evento el sistema debe responder de manera instantánea cambiado de contexto a la tarea correspondiente. Ciertas tareas pueden requerir un estricto *timing* ejecutándose de manera periódica, por ejemplo. En este caso se asigna una alta prioridad para asegurar que el sistema operativo siempre ejecutara esta tarea cuando corresponda.

FreeRTOS

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

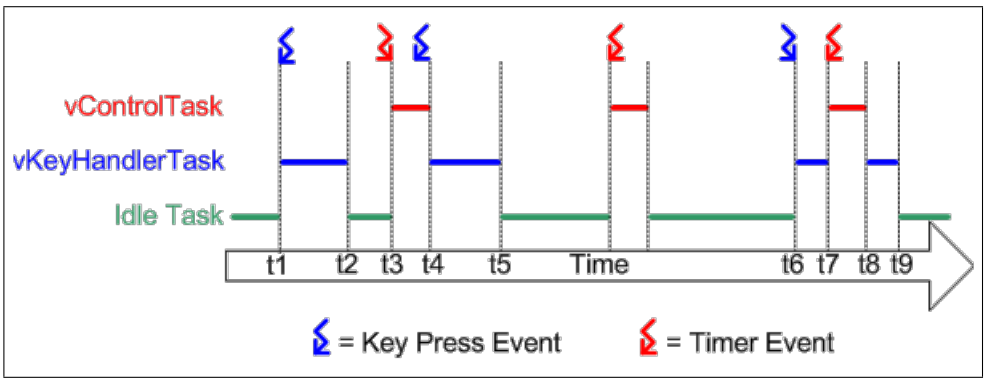


Figura 2.4: *Real time scheduling*

el como controladores para el hardware o periférico. 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. Una tarea se puede encontrar dos estados fundamentales: ejecutándose y no ejecutándose. Cuando una tarea se está ejecutando tiene el control del procesador y el código dentro de la función que representa a la tarea es procesado. El estado "no ejecutándose" en realidad consta de tres sub-estados como se observa en la figura2.5: se inicia en un estado "listo" lo que indica que la tarea está en condiciones de ser selecciona por el *scheduler* y pasar a estado "ejecutándose"; el estado "bloqueado" significa que la tarea no está disponible para ser ejecutada pues está en espera de algún evento, por ejemplo, la liberación de un mutex; cuando la tarea está en estado "suspendida" tampoco se puede ejecutar, la tarea debe explícitamente reanudarse para quedar en condiciones de ser ejecutada.

La creación de taras 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 sobre una

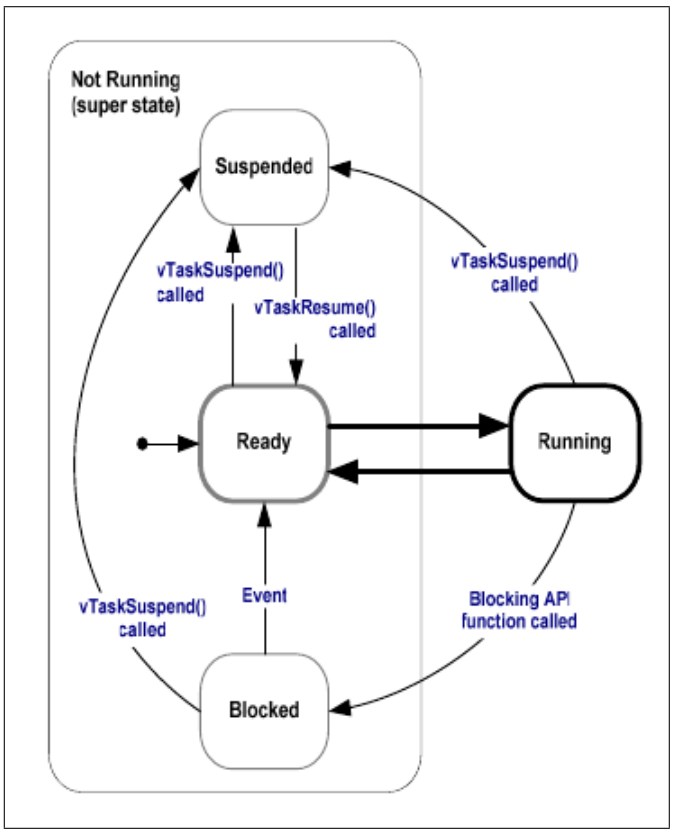


Figura 2.5: Tareas de FreeRTOS, diagrama de estados

tarea.

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 multitarea, cuando en realidad sólo una tarea puede ejecutarse a la vez ya que se posee un solo 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 este tiempo se cumple se debe guardar el estado de ejecución de esta tarea y se procede a ejecutar otra. Así cada una de las tareas se ejecuta durante una pequeña porción de tiempo, hasta que completa su trabajo; si el tiempo de proceso asignado a cada tarea es lo suficientemente pequeño pareciera que muchas cosas ocurrieron simultáneamente. Claramente una sola tarea tomaría, en términos absolutos, menos tiempo en terminar 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.

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 ejecutada. Si varias tareas en estado "listo" comparten la misma prioridad se aplica un algoritmo de *round-robing*. Las tareas 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 y haciendo un uso óptimo del procesador.

Figura 2.6: *Scheduling* de tareas

2.1.3. Ingeniería de Software

Licencias de Software

Se entiende por licencia de software a un contrato entre el desarrollador del software y el usuario final para su utilización según una serie de términos o condiciones. La licencia puede ceder ciertos derechos al usuario final; controlar la cantidad de copias que puede utilizar; el ámbito geográfico y temporal para la utilización; o bien proteger al desarrollador frente a la utilización del programa informático que se licencia. Existen al menos tres tipos de licencias de software:

- **Privativas:** El software es distribuido al usuario bajo un *End-User License Agreement* (EULA) en que el propietario fija las condiciones de uso y se reserva la propiedad del programa informático. Generalmente impide el acceso al código fuente; la realización de ingeniería inversa; el uso del software por más de un usuario; se entrega el derecho de uso por un tiempo definido y por lo general provee cierta asesoría técnica. Es común que se debe pagar por el uso de un programa bajo este tipo de licencia.
- **Libres:** Este tipo de licencias otorgan al receptor la libertad de usar, estudiar, compartir y modificar el software. Existen varias licencias que cumplen con esta definición, por ejeplo: MIT, BSD, **LGPL!** (**LGPL!**) y GPL. Se dividen en dos tipos básicos licencias con *copyleft* y sin *copyleft*. Se habla de una licencia con *copyleft* cuando esta indica que el software derivado debe mantener la misma licencia original impidiendo la generación de software privativo a partir de desarrollos libres, por ejemplo. Por lo general cumplir esta licencia requiere que se garantice el acceso al código fuente, de aquí el termino conocido como software de código abierto. La mayoría del software bajo estas licencias se distribuye de forma gratuita aunque su uso comercial no está necesariamente prohibido. No todo software gratuito es necesariamente software libre.

- **Dominio Público:** Si un programa informático se distribuye sin ningún tipo de licencia, se dice que es de dominio público. No posee ningún tipo de restricción sobre su uso, así como ninguna responsabilidad sobre sus creadores.

La aplicación de licencias se rige según las normativas legales locales. En el caso de las licencias libres por lo general basta con distribuir el texto de la licencia junto con la aplicación y agregar un encabezado indicando el tipo de licencia que se utiliza. Para atribuir autoría se suele utilizar las definiciones de la Convención de Berna, que indica que todo lo que se escribe queda automáticamente sujeto a copyright desde el momento en que la obra es fijada en un soporte material[**?**]. En Chile la legislación vigente al respecto es la Ley 17.336 de propiedad Intelectual.

GPL

La GPL es una licencia de software creada en 1989 por la *Free Software Foundation* que busca declarar que el software así licenciado es software libre y por lo tanto el usuario posee los siguientes derechos[**?**]:

- Libertad de usar el software para cualquier propósito.
- Libertad de modificar el software para adaptarlo a las propias necesidades.
- Libertad para compartir el software.
- Libertad para publicar los cambios que se han realizado.

Actualmente se encuentra en su versión 3.0 que a diferencia de versiones previas es una licencia con *copyleft* y agrega cláusulas para proteger la libertad del usuario frente a nuevas prácticas en contra del software libre, tales como[**?**]:

- Tivoización: El término se refiere a la práctica de limitar los derechos de los usuarios que compran sistemas que funcionan con software libre mediante mecanismos de hardware, por ejemplo evitando la ejecución de versiones modificadas del software embebido.
- Leyes que prohíben software libre: Se asegura de que ciertas leyes puedan limitar los derechos del usuario de un software con licencia GPL.
- Uso malicioso de patentes: Evita el uso indiscriminado de patentes, como por ejemplo, intentar obtener beneficios patentado desarrollos de software libre lo cual es una amenaza a la liberta de los usuarios.

Aplicación de la licencia Para licenciar un proyecto de software libre bajo la GPL V3.0 se deben seguir los siguientes pasos[**?**]:

1. Agregar un aviso informativo del *copyright* al inicio de cada archivo con código fuente de la aplicación. Un ejemplo es la siguiente línea: `«Copyright 2012 Universidad de Chile»`
2. Bajo el aviso de *copyright* se agrega una autorización de copia indicando que el software se distribuye bajo los términos de la licencia GPL. Un ejemplo de este aviso se cita en un ejemplo posterior.

3. Se debe incluir junto al código fuente el texto completo de la licencia en un archivo llamado `LICENSE`. El texto de la licencia se puede obtener desde el siguiente enlace: `http://www.gnu.org/licenses/gpl.txt`.

A continuación se detalla el encabezado que debería incluir cada fichero del proyecto de software hipotético llamado `Foobar` que es licenciado bajo GPL.

```
Foobar - Description - «Copyright 2012 Universidad de Chile»

This file is part of Foobar.

Foobar is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

Foobar is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with Foobar.  If not, see <http://www.gnu.org/licenses/>.
```

Calidad de software

Los requerimientos no funcionales de un proyecto de software puede definirse como los parámetros de calidad buscados en producto en sí. Como parámetros de calidad se pueden entender una serie de calificativos muy subjetivos y tal vez difícilmente medibles como: rapidez, seguridad, escalabilidad o 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 de estos conceptos. 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 norma 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 software durante las etapas de: identificación de requerimientos; validación de la integralidad de la lista de requerimientos; identificación de los objetivos del diseño del software; identificación de los objetivos de las pruebas; identificación de los criterios de calidad; o la definición de criterios para determinar si un producto de software está completo [8].

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.
  - **Completitud 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 usado bajo ciertas condiciones.
  - **Tiempo**. El grado en que el sistema cumple con 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 cumple los requerimientos.
  - **Capacidad**. Grado en que los límites máximos de un sistema cumplelos 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 productos 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 protege 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 especificadas por un determinado periodo de tiempo.
  - **Madurez**. Grado en que el sistema cumples 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 del sistema ante una interrupción o falla.

- **Seguridad**. Cómo un sistema protege la información y los datos de modo que las personas, productos o sistemas tengan el grado de acceso a los datos adecuados 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 puedes ser asociadas de manera inequívoca a esa entidad.
  - **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 componentes discretos de modo que el cambio en un componente tiene 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 en 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 las partes que deben ser modificadas.
  - **Modificable**. Grado en el sistema puede ser modificado de manera efectiva y eficiente, sin introducir defectos o degradar la calidad existente.
  - **Testeable**. 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 u 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.

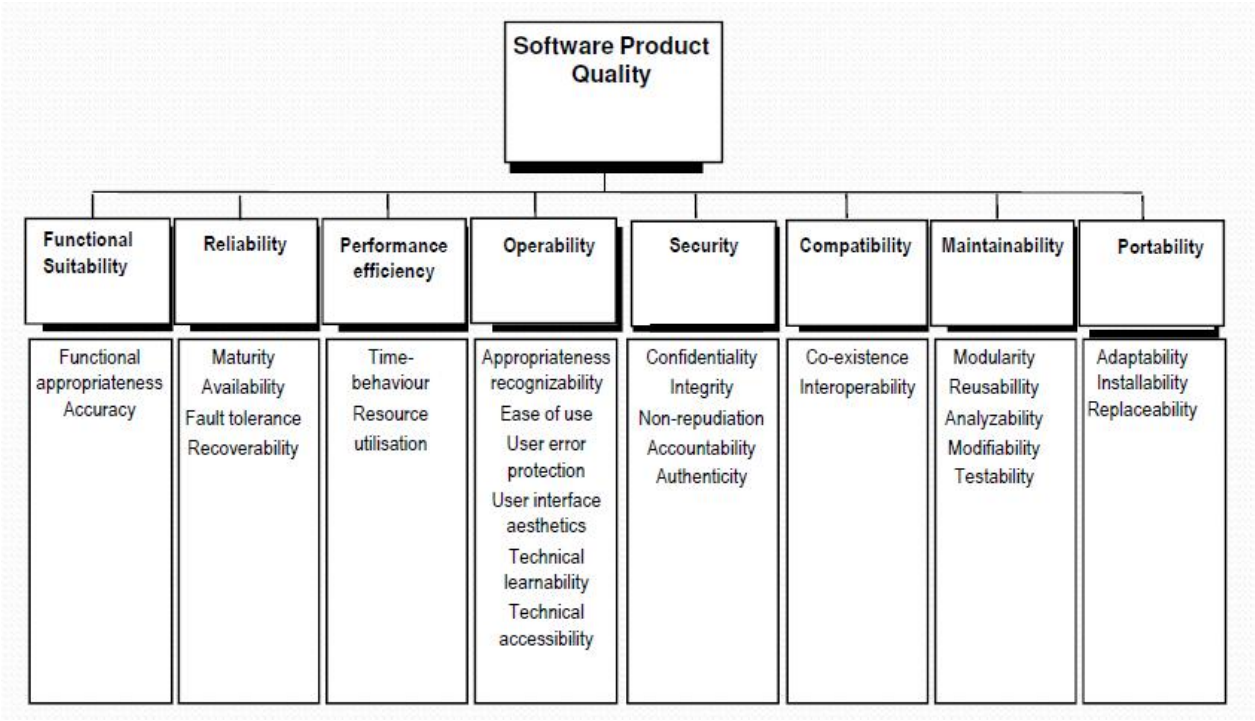


Figura 2.7: ISO/IEC 25010, categorías y subcategorías.

2.1.4. Arquitectura de Software

Patrones de diseño

Command Pattern

2.1.5. Pequeños Satélites

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 proveer acceso al espacio a pequeños *payloads* en un corto periodo de desarrollo y abajo costo se provee un nuevo estándar para pico-satélites denominado Cubesat. El estándar considera satélites de pequeñas dimensiones, acotados a un cubo de 10 [cm] de arista y un peso máximo de 1.3 [kg] como el detallado en la figura 2.8 (a). El proyecto contempla tanto las especificaciones generales del satélite así como una plataforma estándar para desplegar los satélites desde el vehículo de lanzamiento, denominada P-POD. Cada P-POD consiste en un compartimiento capaz de albergar tres Cubesat y desplegarlos mediante una sistema de resortes ante la señal generada por el vehículo de lanzamiento. La figura 2.8 (b) detalla la estructura de un P-POD.

(a) Cubesat (b) P-POD

Figura 2.8: 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*[10] y a continuación se detallan 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 2.9
  - El cubo debe tener dimensiones de 100x100x113 [mm] en x,y,z respectivamente según la figura 2.9
  - 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.
- **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 el comandos 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 despliegue desde el P-POD.
- Transmisores de rados de potencia mayor a 1mW no debe transmitir 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). Adicionalmente se pueden combinar hasta tres unidades para obtener Cubesat de 2U o 3U y así contar con mayor volumen para posicionar los componentes físicos del satélite y una mayor superficie para situar panales solares que brinden mayor autonomía energética.

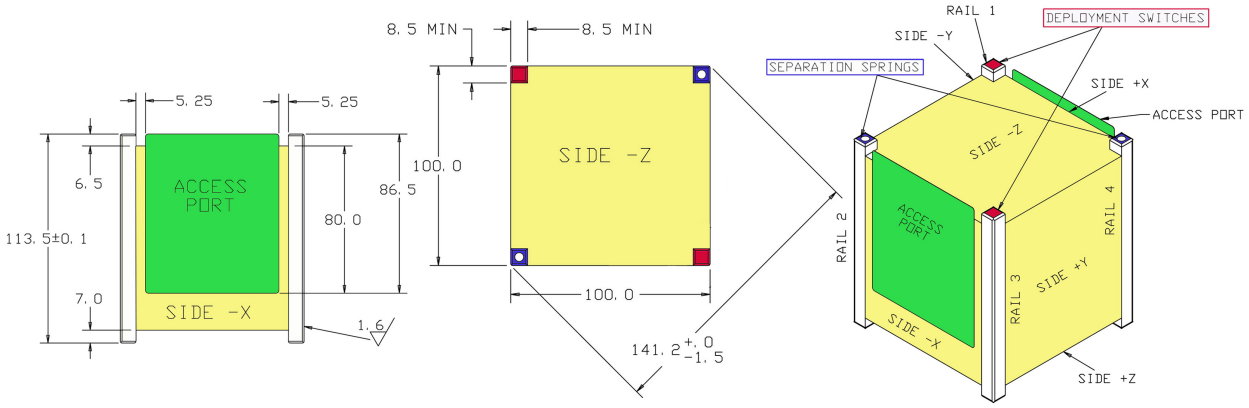


Figura 2.9: Especificaciones de diseño del estándar Cubsat

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 rápido desarrollo y a un costo comparativamente bajo, este tipo de tecnologías abre las posibilidades de investigación y desarrollo de proyectos en materia aeroespacial a instituciones educaciones y gobiernos de todo el mundo, teniendo como antecedente que la mayoría de los proyectos de satélites Cubesat han sido desarrollados por estudiantes en universidades con componentes comerciales. 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 abriendo las posibilidades de desarrollar investigación científica.

2.2. Proyecto SUCHAI

Investigaciones asociadas

Lagmuir Probe

Experimento de física

Capítulo 3

Diseño del software

3.1. Resumen

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

3.2. Requerimientos

3.2.1. Requerimientos operacionales

Los requerimientos operacionales se refieren a las funcionalidades que se espera que el computador a bordo del bus SUCHAI deba realizar. Estos requerimientos son los requisitos básicos que el sistema debe cumplir para considerar que se cuenta con un satélite capaz de llevar a cabo la misión del proyecto SUCHAI y son clasificados según el área de la misión que se ve afectada.

La lista de requerimientos proviene de una serie de reuniones sostenidas con los integrantes de los diferentes grupos de trabajo según los lineamientos del jefe de proyecto.

Área de comunicaciones

Configuración inicial del *transcevier*. El satélite debe ser capaz de fijar las configuraciones iniciales del sistema de comunicaciones como encender el *transcevier*, silenciar las comunicaciones durante determinado tiempo inmediatamente después del lanzamiento, configurar la frecuencia de transmisión a la frecuencia asignada por la IARU y la potencia, configurar y encender beacon, entre otros.

El sistema debe almacenar de manera permanente estas configuraciones iniciales, para reconfigurar el *transcevier* en caso de reinicio o falla así como permitir una reconfiguración 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 con la estación terrena. Esto se realiza mediante la activación de algún sistema eléctrico, que cuenta con cierto sistema de 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 y 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 el satélite se pueda rastrear para lo cual se debe emitir una señal de baliza o *beacon*; segundo, el satélite debe escuchar la estación terrena para determinar si se recibirán ordenes o se descargará información; y tercero, establecer un protocolo de enlace que permita realizar las operaciones de descarga y subida de datos.

Envío de telemetría. El satélite recolectará los datos requeridos por la misión, que incluyen información general sobre el estado del vehículo espacial y sus subsistemas así como los datos generados por *payloads* abordo. El envío de telemetría puede ser automático cada vez que el satélite se enlace con la estación terrena o 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 mantener estos datos 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.

Plan de vuelo. El satélite debe ser capaz de recibir y ejecutar un plan de vuelo consistente en una serie de acciones a ejecutar en momentos determinados de tiempo. El plan de vuelo puede ser precargado en el satélite antes de ser lanzado así como 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 sistema en general. Esta información será usada para determinar la salud del sistema y tomar las acciones necesarias en vuelo 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 sistema, el estado del sistema de comunicaciones, el estado del sistema de control, entre otras.

Inicialización del sistema. El software control debe poseer un algoritmo de inicio del sistema en general, que considera la inicialización del software con los parámetros adecuados a un estado adecuado, la inicialización de otros módulos o subsistemas así como las obligaciones de silencio radial durante el lanzamiento, despliegue de antenas, etc.

El software de control debe tener la capacidad de reiniciarse de manera segura y considerando variables fundamentales del 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 del satélite. Esta información debe estar a disposición como una variable del sistema para ser utilizado por el sistema de control de energía utilizada 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 del presupuesto energético disponible así como una manera de planificar el consumo energético 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 con el objetivo de contar con la

información necesaria para la ejecución de acciones dependientes de la posición real de satélite. Ejemplos de este tipo de acciones son la ejecución de un experimento en algún *payload* o en enlace con una estación terrena para descargar datos de telemetría.

Payloads

Ejecución de comandos de *payloads* . El sistema de control debe tener la capacidad de controlar diferentes subsistemas asociados a *payloads* del satélite. Se debe considerar que los *payloads* abordo del satélite pueden variar de misión en misión e incluso pueden ser descartados o agregados en etapas tardías del proyecto. Por eso el sistema de control 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 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 el funcionamiento del sistema ante fallas causadas por radiación.
- Recuperarse ante fallas de software
- Establecer estados de funcionamiento según nivel de fallas.
- Detectar y solucionar problemas al desplegar antenas.
- Capacidad de *debug* durante el desarrollo y previo lanzamiento

3.2.2. Requerimientos no funcionales

Por requerimientos no funcionales se entienden aquellos atributos asociados a la calidad del software o criterios que permiten determinar cómo debería ser el software que se está diseñando. A diferencia del los requerimientos funcionales que explican lo que el software debería hacer, los requerimientos 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 un software que sea altamente mantenible debido a dos factores básicos: primero, el desarrollo del software 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 del sistema debe proveer la capacidad de análisis del sistema a 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 el sistema, pues se considera como filosofía de trabajo en el proyecto SUCHAI el contar siempre con un sistema funcional ante la eventual posibilidad de lanzar el satélite. Así, se partirá implementado un sistema 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 'mantenible' de la norma ISO/IEC 25010[8].

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 los efectos de la radiación solar sobre los componentes electrónicos[**?**]. Esto significa que el sistema debe tener una nivel de tolerancia a fallas y ser capaz de recuperarse ante una interrupción o fallo. En lo que a software respecta, la característica de tolerancia a fallos será considera en su nivel básico, debido a que por lo general esto significa diseñar sistemas altamente redundantes con una serie de estados de funcionamiento que elevan la complejidad del diseño siguiendo una línea contraria a otros requerimientos no operacionales.

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 de tener una respuesta a cada función necesaria de implementar para ser considerada como válida. Posiblemente en las primeras iteraciones se busque cumplir con los requisitos mínimos de la misión, pero la arquitectura debe ser la idónea para delinear claramente la forma de agregar todas las funcionalidades requeridas y que las tareas requeridas sean llevadas a cabo de manera correcta.

Existen algunos características de calidad que no serán relevantes en este diseño ya que al ser la primera aproximación en la materia para el equipo se requiere mantener cierto nivel de simplicidad en la solución, luego probarla y así ganar la experiencia necesaria en el desarrollo de tecnología aeroespacial. 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 limitada de tareas; tareas de alta demanda computacional pueden ser realizadas en tierra; y no se consideran tareas que requieran una gran precisión de tiempo. La seguridad tampoco es una componente fundamental, si bien, se podría requerir 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 por ejemplo la comunidad de radioaficionados. 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 por la naturaleza de las plataformas a que se apunta se cuenta con muy bajo nivel de estandarización en los niveles más cercanos al hardware.

Por último la tabla 3.1 resume los requerimientos no funcionales del proyecto asignándole

cierta prioridad o importancia a considerar en el diseño.

Tabla 3.1: Requerimientos no funcionales

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

3.2.3. 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 sistema la simplicidad necesaria asegurando funcionalidad. La completitud de los requerimientos operacionales se obtendrán como mejoras incrementales sobre los requerimientos mínimos ya mencionados.

Tabla 3.2: Requerimientos no 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

3.3. Plataforma

La plataforma para la cual se diseña el software corresponde a un kit Cubesat de una unidad adquirido a la compañía Pumpkins Inc. El kit está compuesto por una chasis de 1U con caras perforadas, un computador abordo con placa madre tipo

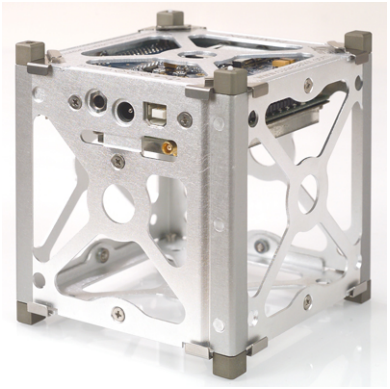
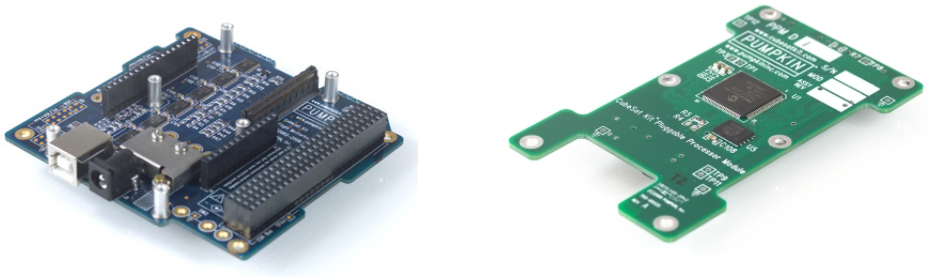


Figura 3.1: Chasis Pumpkins para Cubesat de 1U

3.3.1. Computador a bordo

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

La placa cuenta además con un reloj de tiempo real que se comunica por I2C el cual puede ser usado como reloj, alarma y/o *watchdog* externo; cuenta con una interfaz de memoria SD para contar con un medio de almacenamiento masivo de hasta 2GB; una interfaz USB 2.0 para comunicaciones previas al lanzamiento; así como la electrónica para proveer alimentación al bus PC104 y al módulo del procesador[1].



(a) Placa madre (b) Módulo del procesador

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

El módulo del procesador cuenta con un microcontrolador PIC24FJ256GA110; una memoria flash de 64Mbit con interfaz SPI; osciladores; circuitos de alimentación, protección de sobre corriente y reset. El microcontrolador posee la siguientes características[2]:

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

Como se observa las capacidades de cómputo de las plataformas son muy limitadas, así como la cantidad de memoria de programa y de datos disponible. Esto supone fuertes restricciones tanto en el diseño e implementación del software en tanto no se puede recurrir a alternativas como un sistema operativo Linux o utilizar lenguajes de programación interpretado tipo Java o Python. Las implicancias de las restricciones impuestas por 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 implementado drivers para los periféricos del microcontrolador y para cada dispositivo que se agrega al sistema.
- Sólo existen algunos sistemas operativos básicos para este tipo de dispositivos, y en general permiten organizar el software en módulos, procesos o tareas que se ejecuten de manera concurrente.
- Existe una cantidad muy limitada de código previo, reutilizable, para implementar el diseño de la aplicación, descartando de plano la posibilidad de utilizar bases de datos tipo SQL, protocolos TCP, UDP o librerías POSIX.

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.

3.4. Arquitectura de software

3.4.1. Arquitectura Global

En este nivel se propone una arquitectura de capas. Las diferentes capas agrupan funcionalidades similares y cada una interactúa sólo con la directamente superior e inferior, en una dinámica donde la capa inferior es una prestadora de servicios para la capa superior[6]. Esta arquitectura es una buena forma de proveer sistemas portables entre diferentes plataformas de hardware porque en cuanto 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 para el diseño de software en sistemas embebidos donde se pueden distinguir al menos tres niveles: capa de bajo nivel relacionada con los controladores de hardware, también llamada a menudo como capa de abstracción de hardware o HAL por sus siglas en inglés; la cada intermedia que corresponden al nivel del sistema operativo o gestor de tareas del sistema; y una capa superior que corresponde a la aplicación final del sistema.

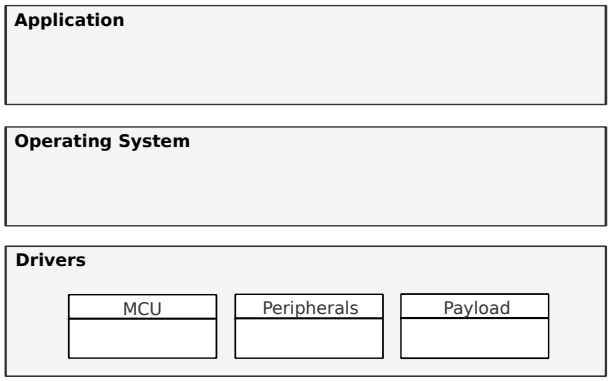


Figura 3.3: Arquitectura de tres capas para un sistema embebido.

La capa de más bajo nivel corresponde a los controladores de hardware, que pueden ser varios módulos específicos a cada pieza de hardware o subsistema presente y permite a las capas superior acceder al hardware sin conocer en detalle sus particularidades. Por ejemplo esta capa presta los servicios de acceder a periféricos de comunicaciones o subsistemas de hardware externo como puden 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*, la integración de variado hardware 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 nivel de 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 la capa de más bajo nivel y provee sus funcionalidades a la capa superior o de

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 nuevo desarrollo listo.

Una de las principales desventajas de esta arquitectura, sobre todo en sistema embebidos, es puede existir la necesidad de una comunicación entre capas no adyacentes[6] rompiendo la arquitectura. Esto se debe tratar de evitar, o al menos realizarlo de manera controlada.

3.4.2. 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 piezas de hardware. Se llaman controladores de hardware o *drivers* a aquellas librerías de software que permiten inicializar y manejar el acceso a este hardware por parte de las capas superiores de sistema operativo y aplicación[5].

Se requiere al menos un controlador para cada hardware existente en el sistema embebido, y se diseñan bajo la idea de agrupar hardware de similares características o funcionalidades bajo una interfaz común que permita abstraer las particularidades de la implementación en cada pieza de hardware. Hacia las capas superiores se maneja el "que hace" el dispositivo entregando por lo general funciones básicas que persisten entre diferentes arquitecturas de hardware como: la inicialización, encendido o habilitación del dispositivo; su configuración; la lectura de datos desde el dispositivo; la escritura de datos hacia el dispositivo; y finalmente deshabilitar o apagar este dispositivo. Además cada hardware puede poseer funcionalidades específicas que también deben ser accesibles a través del driver.

Hacia la capa de hardware el controlador maneja el "como", lo que incluye por lo general el manejo de las interfaces de entrada y salida de datos; manejo de las interrupciones; o manejo de la memoria del dispositivo. Si bien la implementación de cada controlador es específica al dispositivo, se pueden diferenciar al menos tres tipos de arquitecturas comunes en la implementación 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 el controlador 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 en la arquitectura. 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 o no 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 pieza de software también se considera parte del controlador.

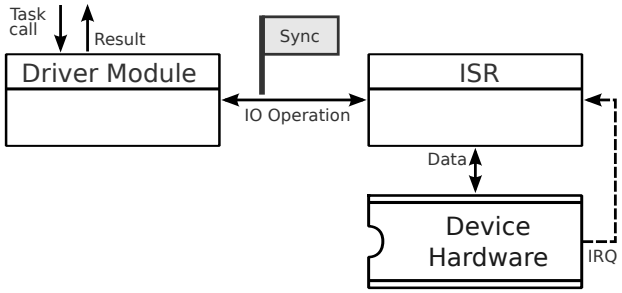


Figura 3.4: Arquitectura para controladores síncronos

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, esta se encarga de las siguientes operaciones:

- Atender la interrupción y restablecer el estado del dispositivo.
- Obtener datos desde, o agregar 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 a acciones al controlador puede continuar su ejecución sin esperar el resultado. En esta caso se habla de un controlador asíncrono. Este tipo de driver no es común de encontrar y por lo general se puede evitar cuando, para la tarea que llama al driver, no tiene mucho sentido avanzar en su ejecución antes de que se terminen las operaciones de entrada y salida. Un caso especial corresponde a la ejecución en diferentes etapas de la operaciones del driver, en donde la tarea mandante puede procesar los datos de una etapa mientras el proceso de entrada o salida continua su ejecución.

La arquitectura para este tipo de drivers 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. Acá el driver se sincroniza con la tarea a través de una cola de mensajes, donde se almacenan los resultados parciales o de cada etapa del proceso completo para que la tarea pueda procesar en paralelo esta información.

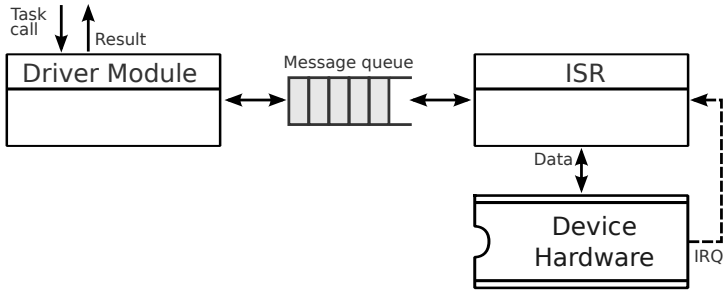


Figura 3.5: Arquitectura para controladores asíncronos

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 se observa de manera común en sistema embebidos y corresponde a la entrada de datos seriales asíncronos. En este tipo de controladores se cuenta con la llegada de una gran cantidad de datos y donde el término de la operación está determinada por la cantidad máxima de datos recibido o por algún dato indicador del término de la operación.

La arquitectura que responde a esta situación se detalla en la figura 3.6. En este caso además del módulo o función del controlador, la rutina de atención de interrupciones y una cola de mensajes (opcional) se agrega un *buffer* de memoria que es creado previo inicio de las operaciones de entrada de datos y que es accedido por referencia para evitar el uso adicional de memoria y copia de datos entre llamadas.

Las operaciones que corresponden al módulo del controlador son:

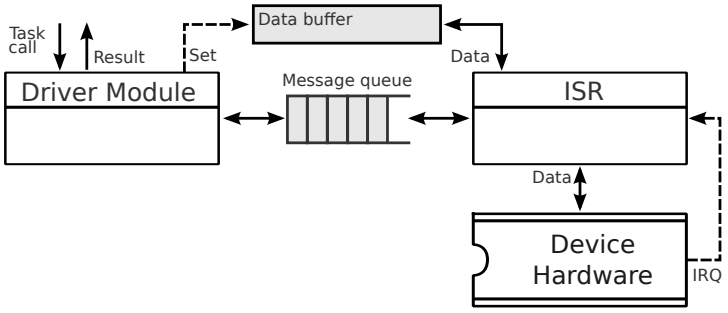


Figura 3.6: Arquitectura de un controlador de entrada serial

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

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

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

3.4.3. Sistema operativo

El sistema operativo es la capa de abstracción entre la capa de aplicación y el hardware en la arquitectura de un sistema embebido. Permite diseñar la aplicación sin considerar las particularidades de la plataforma en que se ejecuta así como estructurar la aplicación en una serie de procesos o tareas cuya gestión la proveen los servicios del sistema operativo[5]. Los servicios básicos que provee el sistema operativo a través de su kernel son:

- **Gestión de tareas:** También denominada gestión de procesos. El sistema operativo ve a la aplicación como una serie de tareas o procesos, a las cuales se debe entregar los servicios de creación, ejecución y asignación de prioridades. Cada tarea cumple objetivos específicos en la aplicación y posee sus propios recursos y limitaciones de tiempo. Varias tareas pueden existir a la vez en el sistema y el sistema operativo se encarga de proveer de tiempo de procesamiento a cada una de las tareas disponibles. Existen diferentes alternativas apara gestionar la ejecución de las tareas en el sistema, de las cuales sobresalen dos: *preemptive* y cooperativo. La implementación de la aplicación y las tareas dependen mucho del modo de gestión de tareas que utiliza el sistema operativo por lo cual se describe cada uno a continuación:

  - **Modo *preemptive*:** El sistema operativo puede interrumpir la actual tarea cualquier momento de su ejecución para realizar el cambio de contexto y ceder el procesador a una tarea diferente. Por lo generar se realiza a intervalos fijos, denominados *ticks*. Esto puede ir acompañado de un sistema de prioridades, entonces el sistema de gestión de tareas se asegura de 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 tarea. 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 de las tareas 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 tareas funcionando de manera concurrente, la verdadera utilidad nace de la posibilidad de comunicación entre tareas 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. El sistema operativo de encarga de proveer los mecanismos de comunicación y sincronización para asegurar que la información compartida no se corrompa y no existan interferencias entre tareas al acceder 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 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 el sistema operativo 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.

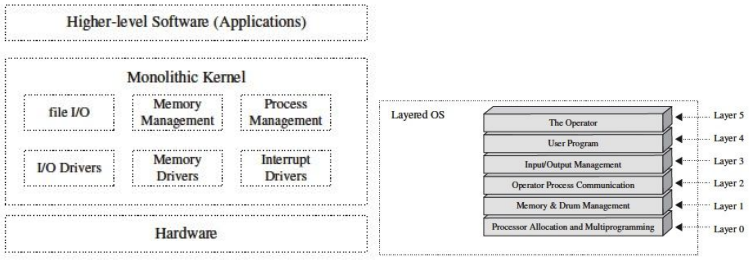
En cuanto a la arquitectura de software presente en un sistema operativo, se pueden encontrar al menos tres opciones bien conocidas: kernel monolítico, arquitectura en capas y micro kernel[5].

Kernel monolítico. En este tipo de kernel los servicios del sistema operativos se encuentran integrados a lo largo de la arquitectura que provee las cinco funcionalidades mencionadas anteriormente. Dada la dificultad de escalar y depurar este tipo de kernel, existe una variante en la cual los servicios del sistema operativo se integran como módulos. Una arquitectura básica de este tipo de sistemas se refleja en la figura 3.7 a. Algunos sistemas operativos que

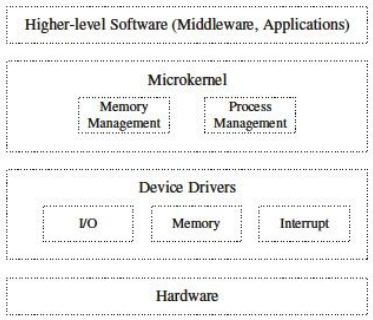
usan esta arquitectura son: Linux, Jbed RTOS y MicroC/OS-II.

Arquitectura de capas. Los servicios del sistema operativos se presentan en una arquitectura jerárquica de capas, donde cada capa depende de los funcionalidades que proveen las capas inferiores. Ejemplos son: DOS/eRTOS y VRTX. Esta arquitectura se presenta en la figura 3.7 b.

Micro kernel. Un sistema operativo con arquitectura de micro kernel provee las funcionalidades mínimas necesarias para su funcionamiento como el gestor de memorias y el gestor de procesos. El resto de las funcionalidades se proveen de manera separada usualmente en una arquitectura tipo cliente-servidor. Este tipo de sistemas es muy común en sistemas embebidos debido a que mantiene controlado el tamaño en memoria del sistema operativo y la velocidad de respuesta del sistema al no incluir componentes que son poco o para nada utilizados. Ejemplos de sistemas operativos con micro kernel son: FreeRTOS, Salvo RTOS y VxWorks. El esquema de estos sistema se aprecia en la figura 3.7 c.



(a) Kernel monolítico (b) Kernel en capas



(c) Micro kernel

Figura 3.7: Arquitecturas de sistemas operativos

En el diseño del software de vuelo el diseño de un sistema operativo está fuera del alcance del proyecto en cuanto existen una serie de alternativas ya disponibles para uso. A continuación se analiza la arquitectura y servicios que proveen algunos de los sistemas operativos para sistemas embebidos recomendados por el fabricante para la plataforma de hardware utilizada.

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

3.4.4. Aplicación

Para el diseño de la aplicación se estudió la adaptación de un patrón de diseño utilizado en la programación orientada a objetos, llamado *command pattern*. Este patrón de diseño soluciona la necesidad de realizar peticiones a diferentes objetos sin necesidad de saber cómo se ejecutaran estas acciones, de este modo la petición es encapsulada en como un comando que 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, se realizará una adaptación de la idea de este patrón de diseño, donde los objetos serán representados como módulos estáticos, con posibilidad de generar comunicación y sincronización entre estos módulos.

Los módulos fundamentales de la arquitectura son lo siguientes:

Listeners. Los *listeners* o escuchadores son los módulos en la capa superior de la arquitectura y emulan lo que serían los clientes dentro de *command pattern*. Estos módulos son los únicos encargados de generar los comandos en el sistema. Pueden existir varios *listeners* dado que implementan la inteligencia del sistema para generar la acciones deseadas ante diferentes circunstancias. Su denominación proviene del hecho de que la justificación para que un *listener* exista, es que se mantiene 'escuchando' alguna variable del sistema o el estado de un subsistema para tomar decisiones sobre los comandos que se deben ejecutar en cada momen-

to. Los *listeners* se pueden ver como las 'aplicaciones' o 'procesos' de otras arquitecturas de software para pequeños satélites[7][9] 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 un aplicación que dado ciertos parámetros, tome decisiones en tiempo real y ejecute las acciones necesarias, por ejemplo, conceptualmente se puede 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 una subsistema o variable al cual prestar atención, ya sea un temporizador, la posición actual o el arribo de un telecomando por mencionar algunos ejemplos. Una vez definido esto, el programador debe determinar bajo qué condiciones se toma la decisión de generar un determinado comando que realice las acciones requeridas.

Los *listeners* no realizan acciones que involucren directamente el acceso a otros subsistemas, evitando así el uso simultaneo de recursos de hardware -como módulos de comunicaciones- que puedan causar un estado de *data race*. Por lo tanto sólo se permite el acceso al repositorio de estados, repositorio de datos y repositorio de comandos. Tampoco en este nivel se ejecutan directamente los comandos, sino que son generados y encolados para su posterior funcionamiento. Esto plantea la limitante de que quien envía el comando no puede saber si fue ejecutado o cuál fue el resultado de su ejecución. No obstante se puede lograr la retroalimentación necesaria a través de la lectura por parte del *listener* del repositorio de estados y de la modificación de un estado en este repositorio por parte del comando generado.

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 al módulo *dispatcher* y son encolados, en este nivel se realiza un control sobre los comandos que llegan y se pueden establecer políticas de rechazo a la ejecución de comandos. 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 ejecutar comandos que usan mucha energía cuando el nivel de carga de las baterías sea crítico.
- Filtrar los comandos provenientes o hacia un determinado subsistema 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 en este apartado las estrategias de control de las operaciones que realizan en el sistema 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.

Executer. Corresponde al módulo final en el flujo de comandos del sistema, donde estos son ejecutados. Corresponde al objeto *reciver* dentro del patrón de diseño original en cuanto este módulo 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. Se ejecuta la función que realiza todas las acciones que implementa dicho comando como: leer datos, acceder a dispositivos, cálculos y guardar resultados, se espera su término y su código de retorno es notificado finalmente al *dispatcher*.

La funciones del *executer* incluyen recibir el comando, identificar y obtener la función a ejecutar, sus parámetros realizar el llamado a la función, recibir el código de retorno y notificar a al *dispatcher* el resultado del comando. 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 se ejecuta con máxima prioridad frente a los otros módulos permite brindar acceso exclusivo del sistema al comando en ejecución. Se ahorra de esta manera todos los problemas que surgen de sincronizar el uso compartido de recursos o subsistemas. Sin embargo se debe cuidar que el comando en ejecución no cause una falla que deje al sistema congelado.
- **Múltiples executers:** Se puede implementar una patrón *Thread Pool*[**?**] para permitir la ejecución de varios comandos de manera concurrente. Esto implica tener varios *executers* esperando a recibir comandos desde una cola. Cuando un *executer* 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 los recursos compartidos que utilice cada comando 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 estos sobre todo ante casos de falla o reinicio del sistema.

La arquitectura de software de la aplicación para la operación del satélite queda descrita por el diagrama de la figura 3.8 que describe el flujo de información entre los diferentes módulos que componen el software, así como sus dependencias, principales operaciones y su correspondencia con el patrón de diseño que inspira la arquitectura propuesta.

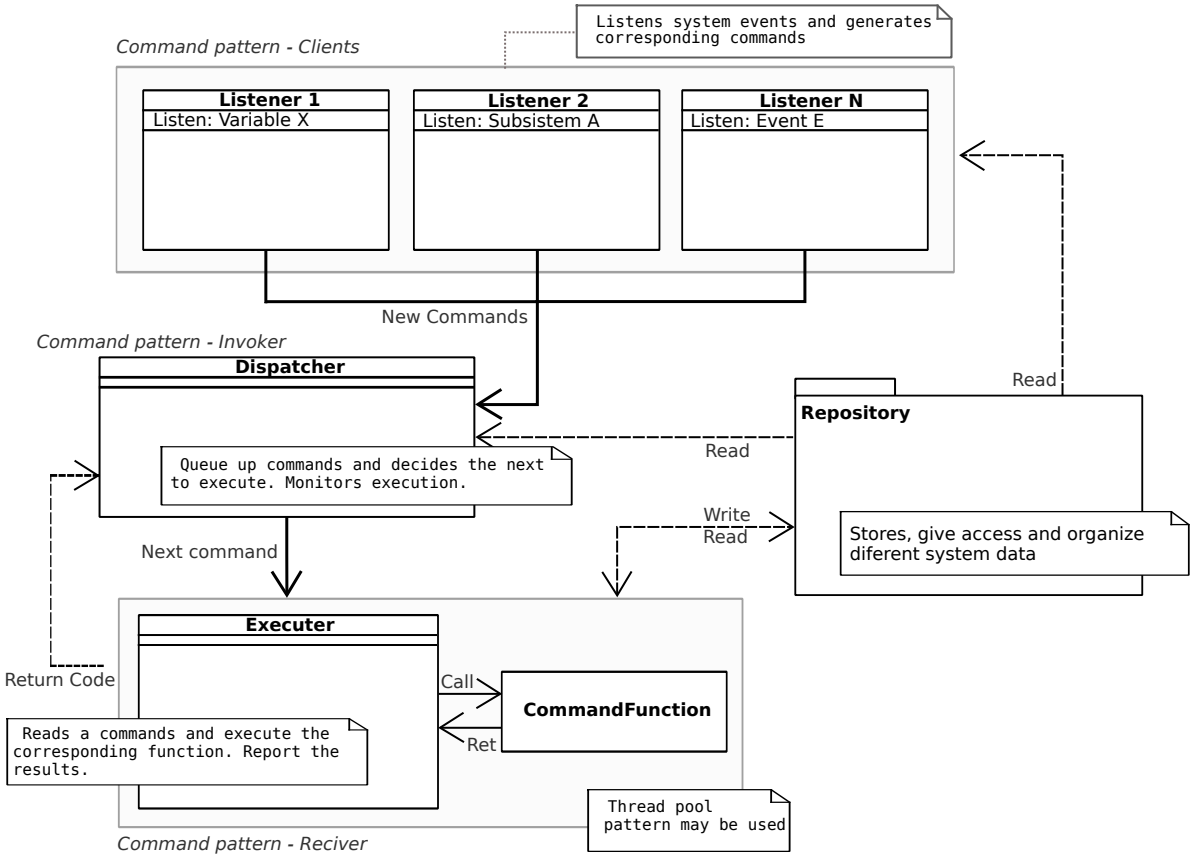


Figura 3.8: Arquitectura de software para el control del satélite

3.5. Diseño

Dado los requerimientos operacionales y no operacionales, así como la arquitectura de software en todos sus niveles, se pasa validar el diseño propuesto para poder asegurar que el software requerido y sus funcionalidades son posibles de implementar con esta solución. Para esto se vuelve 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, 3.5 y 3.7.

El resultado de este ejercicio revela los módulos que son necesarios en el diseño de la arquitectura de software a nivel de aplicación, específicamente los *listeners* que se deben

Tabla 3.4: Análisis de la arquitectura, según requerimientos operacionales, para él 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 transceiver y cuando llega un telecomando se decodifica para generar los comados del sistema correspondientes
Protocolo de enlace	Listener (Communications)	Cuando se recibe un telecomando que inicia la sesión de comunicaciones con tierra se generan comandos que responden según el protocolo y cambian el estado del sistema para establecer comunicación.
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 despligue. 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 él 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 le ejecución de comandos (o grupos de comandos) que están generando problemas en el sistema. (Lista negra). Se puede filtrar comandos desde ciertos listeners.
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 debug para tener una salida con información relevante sobre la ejecución.
Inicialización del sistema	Listener (Deployment)	Inicialmente sólo existe un listener, que genera los comandos para toda la secuencia de inicio, que implica configurar parámetros, repositorios, subsistemas, silencio radial, despligue de antenas y la activación del resto de los listeners.

implementar son:

- **Communications:** Este *listener* se encarga de controlar los eventos relacionados con

Tabla 3.6: Análisis de la arquitectura, según requerimientos operacionales, para él 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 orbita	NA	(1) La orbita se calcula en tierra y se actualiza el itinerario según las predicciones de órbita. (2) Se cuenta con un sibsistema GPS. Se ejecutan el itinerario según coordenadas espaciales.
Ejecución de comandos de Payloads	Listener (FlightPlan)	Comandos se generan según el itinerario del plan de vuelo

Tabla 3.7: Análisis de la arquitectura, según requerimientos operacionales, para él 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 reconfiguración, pasar a modos de fallo, desactivar módulos o subsistmas.
Mucho tiempo sin conexión con tierra	Listener (Communications)	Si no se estable comunicación con tierra luego de N días el sistema pasa a modo de fallo de comunicaciones, permitiendo bajar telemtría básica de manera automática o reiniciar el sistema.
Problemas con despliegue de antenas	Listener (Deployment)	Se chequean sensores que indica si las antenas se han desplegado. Se generan comandos que intenten desplegar las antenas.
Watchdog	Sistema Operativo	Se cuenta con un watchdog en el microcontrolador, que se reseta 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 payloads y hardware externo
Watchdog Externo	Listener (HouseKeeping)	Generar comandos periódicamente que reseteen el watchdog externo, reiniciando el sistema ante fallas generales en el microcontrolador.

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. Estas 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 los comandos adecuados en un modo que desplieguen 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 comandos 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 3.9 que muestra los módulos participantes y el flujo de información en esta arquitectura:

Las capas inferiores en la arquitectura global no se ven afectados por los requerimientos operaciones, 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 contar con la 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 en

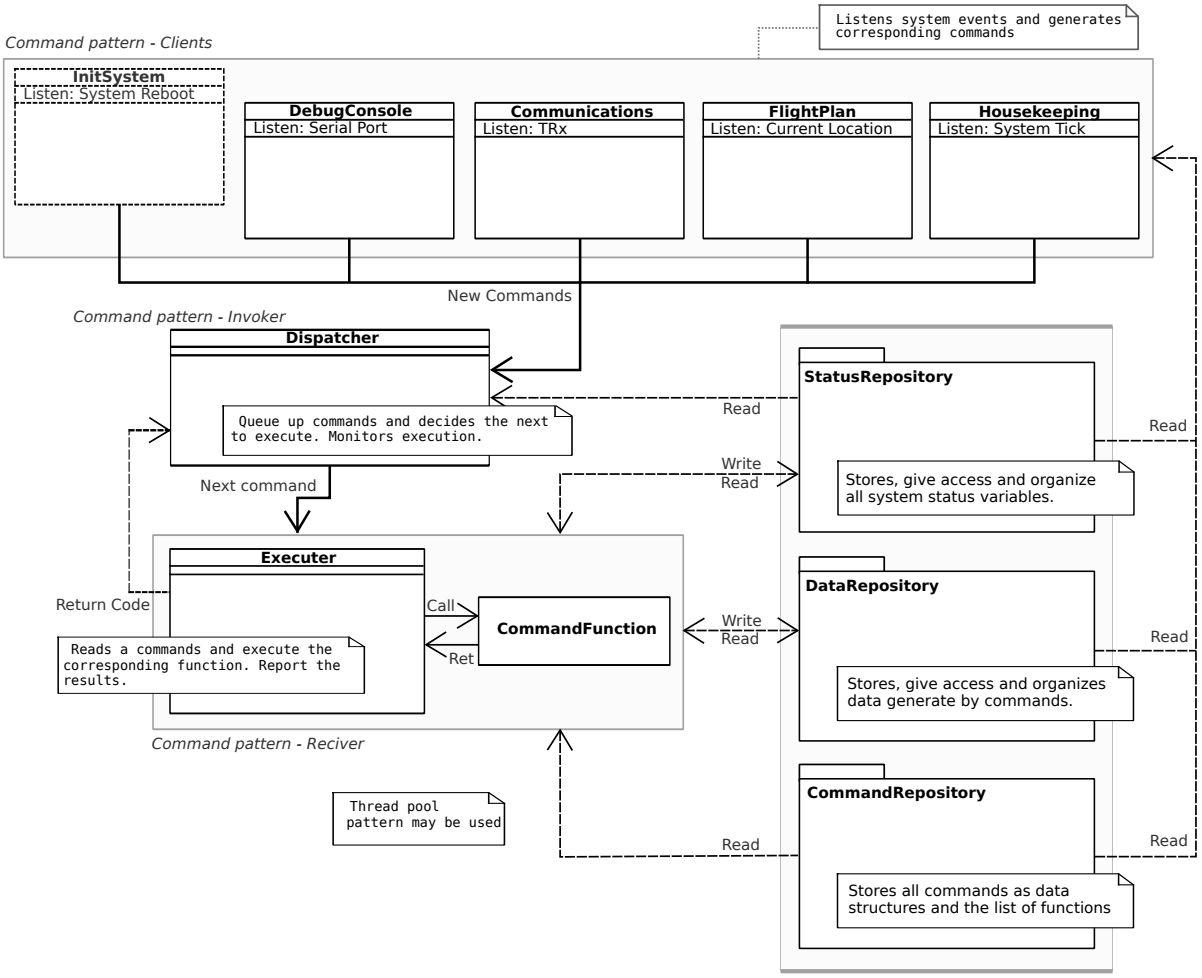


Figura 3.9: Arquitectura de software para el control del satélite

cuanto la interfaz hacia el *middelware* y hacia la capa de aplicación esté bien definida.

Capítulo 4

Implementación

En este capitulo se describe la implementación del sistema de vuelo en todos sus niveles.

Capítulo 5

Pruebas y resultados

En este capítulo se detallan y discuten los resultados obtenidos luego de la implementación del sistema. Se detalla una metodología para realizar las pruebas tanto a módulos aislados como del sistema integrando. Finalmente se analiza el funcionamiento del satélite con sus tres sistemas básicos integrados donde se realiza un checkeo de cumplimiento de estándares y funcionalidades esperadas por el equipo que dirige el proyecto SUCHAI.

5.1. Pruebas modulares

5.1.1. Pruebas a Console

5.1.2. Pruebas a Hauskeeping

5.1.3. Pruebas a Dispatcher

5.1.4. Pruebas a Executer

5.2. Pruebas de arquitectura

5.3. Pruebas de integración básica

Capítulo 6

Conclusiones

Finalmente se realizan las conclusiones del trabajo realizado, entregando información útil que se obtiene luego de realizar la investigación durante todo el proceso. También se plantea el trabajo futuro y/o pendiente. Además se entregan las ramas o líneas de investigación a seguir en base al trabajo realizado.

6.1. Conclusiones generales

6.2. Conclusiones específicas

6.3. Trabajo futuro

Bibliografía

[1] *CubeSat Kit Motherboard (MB).*

[2] *CubeSat Kit Pluggable Processor Module (PPM) D1.*

[3] *PIC24FJ256GA110 Family Data Sheet.*

[4] *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] *Embedded System Architecture.* Elseiver, 2005.

[6] *Software Engineering.* Addison-Wesley, 2011.

[7] Michael J. Dabrowski. The design a software system for small space satellite. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

[8] ISO. Iso/iec 25010:2011 systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models. Technical report, ISO, 2011.

[9] Greg Manyak. Fault tolerant and flexible cubesat software architecture. Master's thesis, California Polytechnic State University, 2011.

[10] Cal Poly SLO. Cubesat design specification. Technical report, California Polytechnic State University, 2009.