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

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

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL 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

Marco Tórico

2.1. Sistemas embebidos

Los sistemas embebidos, a diferencia de un computador personal que es usado con fines generales para una amplia variedad de tareas, son sistemas computacionales normalmente utilizados para atender una cantidad limitada de procesos, realizar tareas específicas o dotar de determinada inteligencia a un sistema más complejo. Está compuesto por uno o más microcontroladores pequeños que cuentan con periféricos para manejar diferentes protocolos de comunicación, conversores análogo-digital, *timers*, puertos de entrada y salida digitales, todo en integrado en un mismo chip para ahorrar espacio y energía. Parte fundamental de un sistema embebido es el *software* que provee la funcionalidad final, usualmente se usa el término *firmware* para referirse a este código con que se programa el microcontrolador el cual por lo general es específico para la plataforma de *hardware* y se relaciona a muy bajo nivel con ella. 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, acá no tiene la capacidad de re-programarlo 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[**?**]:

- Se mantiene siempre funcionando y debe proveer respuesta a entradas y órdenes externas en tiempo real. Se debe diseñar considerando una operación continua y una posible reconfiguración del sistema estando ya en marcha.
- Las interacciones con el sistema pueden ser impredecibles y no se tiene control sobre ellas. Existen sistemas que son controlados por el usuario mediante una interfaz preparada para ello, mientras que otros 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.

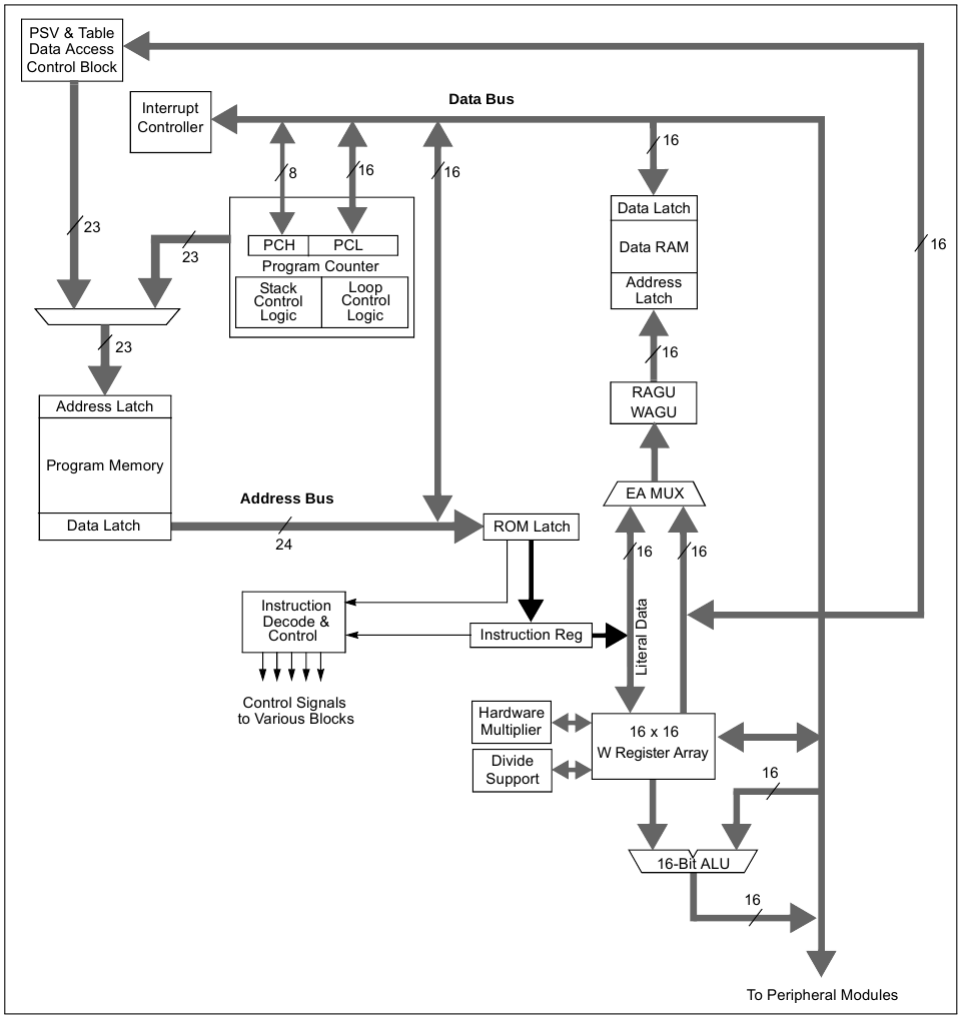


Figura 2.1: Arquitectura de la CPU del PIC24F

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.

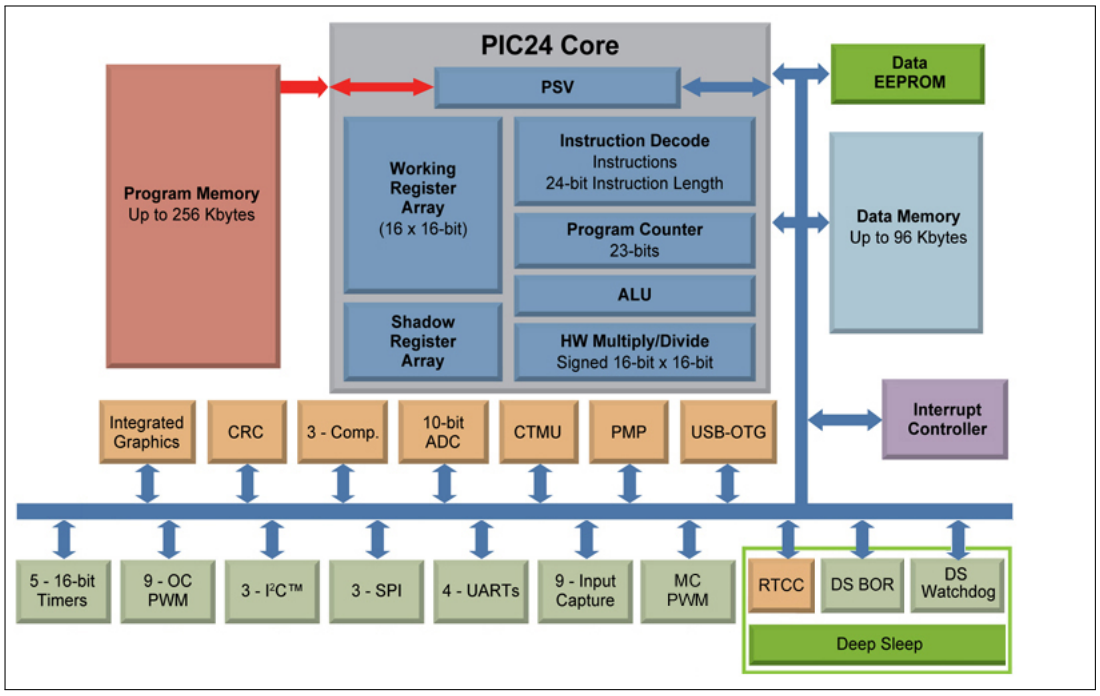


Figura 2.2: Arquitectura del PIC24F

Desarrollo

2.1.1. 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.

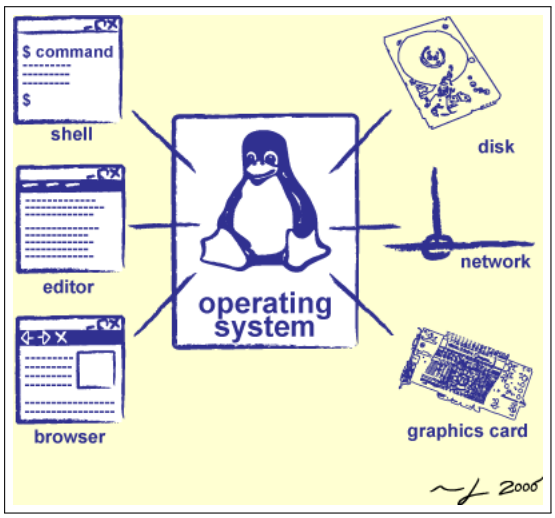


Figura 2.3: Sistema Operativo como capa de abstracción

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 nivel

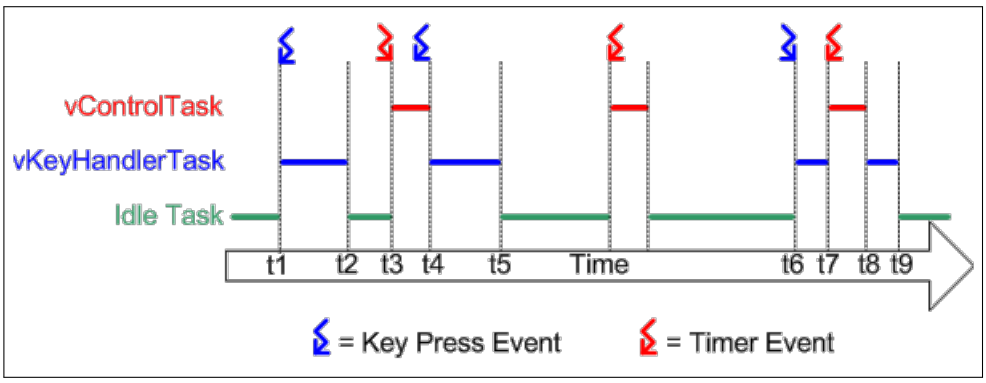


Figura 2.4: *Real time scheduling*

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

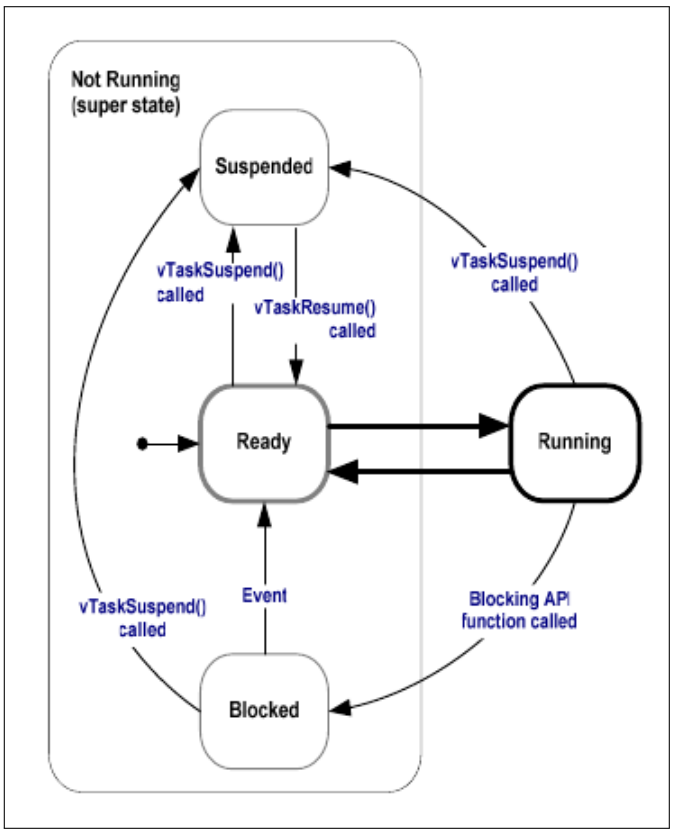


Figura 2.5: Tareas de FreeRTOS, diagrama de estados

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

2.1.2. 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 \textit{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 [**?**].

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.

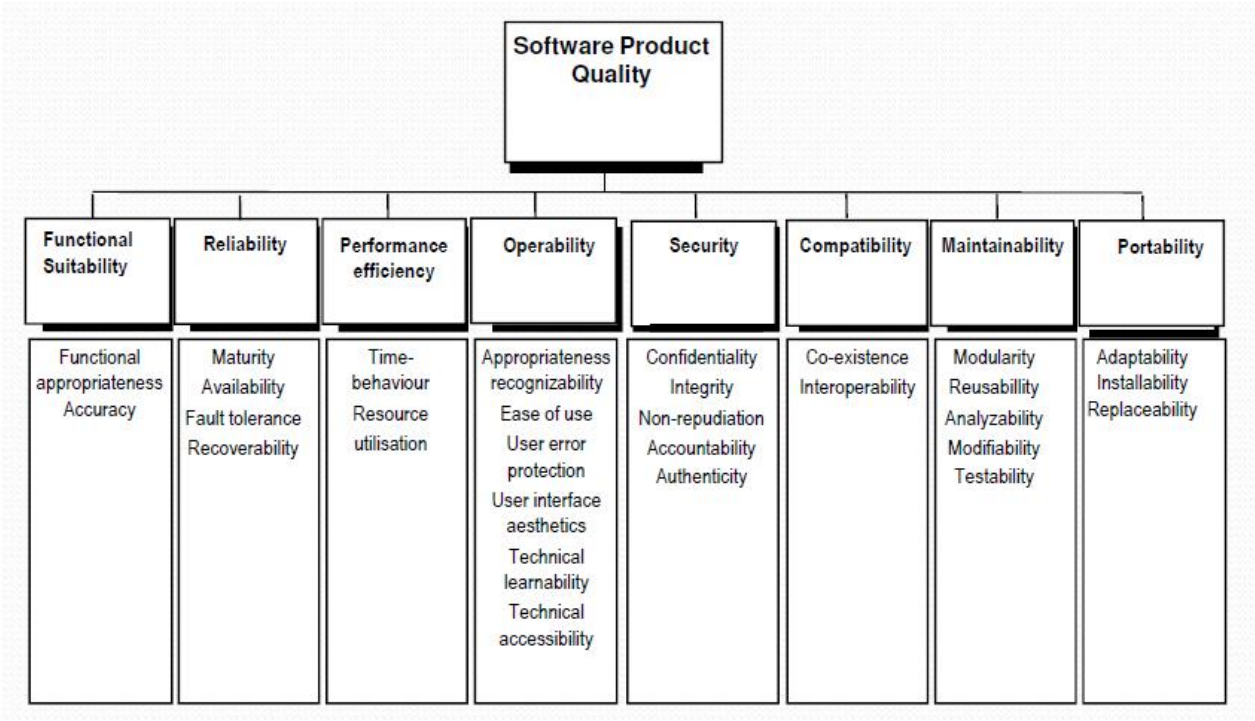


Figura 2.7: ISO/IEC 25010, categorías y subcategorías.

2.1.3. Patrones de diseño

En computación, especialmente en la programación orientada a objetos, se utilizan patrones para sortear la difuctad de generar buenos diseños seleccionando los objetos pertinentes y sus correspondientes relaciones dentro de a aplicación. Los programadores experimentados tienden a reutilizar soluciones bien probadas para luego apadtarlas al contexto específico de un nuevo problema generando una metodología general para resolver cierta clase de problemas, estas metodologías son llamadas patrones de diseño y se encuentran bien documentas para su aplicación sucesiva en cada nueva aplicación.[?]

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

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

Modelo-Vista-Controlador. Este patrón de diseño divide una aplicación interactiva en tres componentes: el modelo, que contiene la funcionalidad base y los datos; la vista, que despliega la información al usuario; y el controlador, que maneja la entrada del usuario para

cambiar el estado de la aplicación. Así, la interfaz gráfica se refleja en el controlador y la vista, donde cada acción se propaga a través del controlador hacia el modelo, actualizando a la vez la vista.[?][?]

- **Contexto:** aplicaciones interactivas son un interfaz de usuario flexible.
- **Problema:** el diseño de una interfaz gráfica de usuario que esté desacoplada del programa principal. La interfaz gráfica por lo general debe ser flexible, permitiendo cambios en su distribución o la forma de llamar a las funcionalidades o varios métodos de entrada que llamando a la misma funcionalidad como botones, linea de comandos o menús. También se puede requerir portar la interfaz gráfica a diferentes ambientes o estándares manteniendo la funcionalidad de la aplicación. Otro caso típico corresponde a la visualización de la misma información de diferentes maneras o en diferentes ventana, como lo pueden ser vistas de gráficos o tablas para los mismos datos, por lo tanto se requiere desacoplar los datos de la vista para evitar la duplicación o corrupción de la información.
- **Solución:** dividir la aplicación en tres áreas: el modelo, la vista y el controlador. Donde el modelo controla la lógica y funcionalidades internas de la aplicación, la vista gestiona la representación de la aplicación hacia el usuario y el controlador maneja las entradas del usuario.

  El usuario sólo interactúa con el controlador, y los cambios realizados en el modelo debe ser propagados hacia la vista. Como cada vista comparte el modelo, los cambios se verán actualizados en cada una de ellas.
- **Estructura:** los componentes que completan la estructura del patrón MVC se detallan a continuación:
  - **Modelo:** encapsula la información y todas las funcionalidades de la aplicación, de manera independiente de su representación gráfica. Provee los métodos para realizar los procesos específicos de la aplicación así como para acceder a los datos de esta. A la vez implementa el mecanismo de propagación de los cambios hacia las diferentes vista suscritas a este modelo.
  - **Vista:** despliega la información al usuario, a partir de los datos del modelo. Permite múltiples formas de visualizar el modelo. Provee un método de actualización que es activado por el mecanismo de propagación de cambios del modelo suscrito.
  - **Controlador:** cada vista tiene asociado un controlador que recibe la entrada del usuario, como presionar un botón o seleccionar una entrada de menú. El controlador solicita el servicio correspondiente al modelo, causando a la vez una actualización de la vista si es necesario.

  Para clarificar las relaciones entre los componentes del patrón se cuenta con el diagrama de la figura **??**. La esencia dinámica del patrón en ejecución se observa a través del diagrama de colaboración detallado en la figura **??**
- **Consecuencias:** alguno de los beneficios que provee son: múltiples vistas para el mismo modelo; vistas sincronizadas dado que los cambios en el modelo se propagan a las vistas que lo comparten; vista y controlador intercambiable de manera independiente del modelo, permitiendo portar la aplicación a diferentes plataformas sin cambiar la funcionalidad. Por otro lado los inconvenientes que genera son: aumento de complejidad, al necesitar adaptar cada elemento de la vista al controlador y funcionalidades; excesivo

número de actualizaciones al propagar cambios del modelo a todas las vistas; alto acoplamiento entre la vista y el controlador, aún como elementos separados su diseño está muy relacionado; acoplamiento entre el modelo y el conjunto vista-controlador, ya que cualquier cambios en la interfaz de los servicios que provee el modelo tiene un efecto directo en el funcionamiento de la vista y el controlado pues acceden directamente a estos servicios.

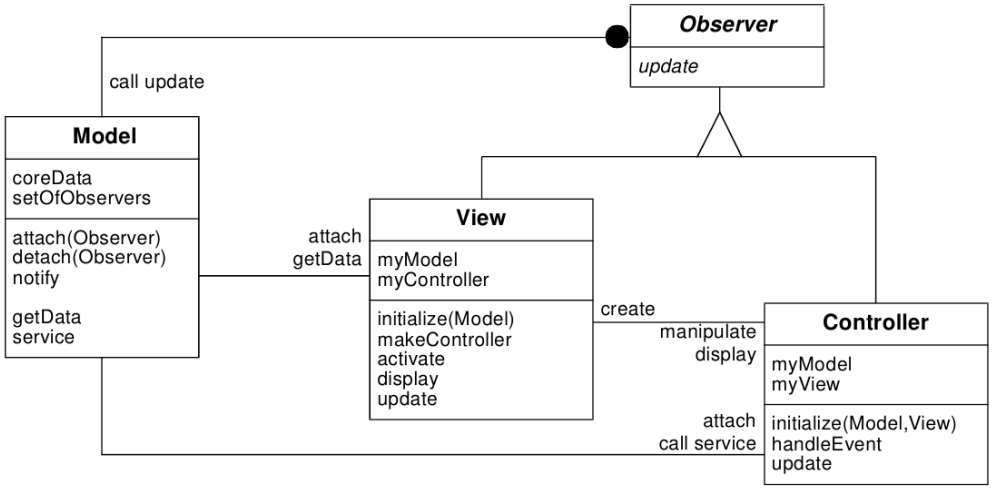


Figura 2.8: Diagrama de clases del patrón de diseño MVC.

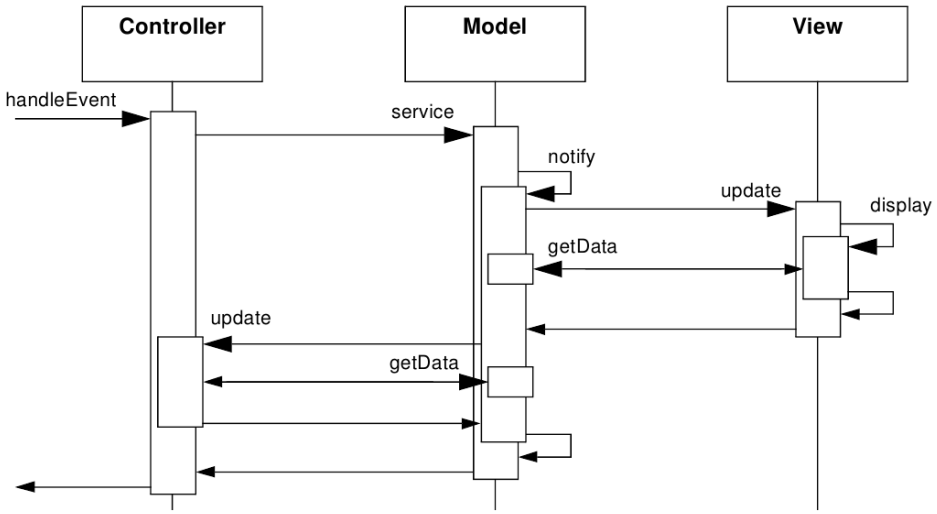


Figura 2.9: Esquema de colaboración del patrón de diseño MVC.

Como se ve, el proceso de diseño de una aplicación, parte por la busca de patrones de diseño claramente documentados y que soluciones un tipo de problema similar al buscado. Comparar patrones de diseño cuando se tienen varios candidatos a solucionar el problema también se hace sencillo en cuanto la forma de presentar cada patrón permite obtener sus principales características y posible implementación. Además una aplicación puede hacer uso de varios patrones de diseño en diferentes niveles de su arquitectura para cumplir la totalidad de las especificaciones.

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

Arquitecturas de *software* basadas en patrones

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

Command Pattern

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

- **Contexto:** Aplicaciones que requieren flexibilidad al añadir funcionalidades. Interfaces gráficas flexibles y extensibles. Aplicaciones orientadas a la ejecución de funciones por parte del usuario soportando características como llevar un registro de ejecución o deshacer acciones. Aplicaciones donde el instante de generación de un requerimiento es independiente del instante en que se ejecuta.
- **Problema:** Una aplicación que requiere dar soporte a una gran cantidad de funcionalidades, se puede ver beneficiada del aislar la forma genérica en que se realiza el llamado a esta función de su implementación misma. También cuando se requiere implementar funcionalidades como registro de eventos, deshacer, macros o suspender cierta ejecución de un proceso, se ve la necesidad de separar las funcionalidades del núcleo de la aplicación que las gestiona. Una aplicación que requiere ser altamente escalable sin afectar el código existente debe encontrar una manera homogénea de agregar estas funcionalidades nuevas.
- **Solución:** Encapsular los requerimientos en objetos llamados comandos, así cada llamada a una función específica crea un nuevo comando, también específico a esa llamada que implementa la funcionalidad. El patrón describe la estructura de gestión para la generación y ejecución de los comandos la cual es homogénea para cada comandos

requerido. Las adaptabilidad y extensión de la aplicación se realiza mediante la implementación de nuevos comandos sobre el sistema de gestión base.

- **Estructura:** La arquitectura está compuesta por los siguientes módulos.
  - **Comando:** Todos los comandos poseen una estructura básica, implementando un método para ejecutar el comando. Para cada función requerida en la aplicación se crea una derivación de la estructura base del comando que implementa la lógica requerida del comando.
  - **Controlador o cliente:** Representa la interfaz de la aplicación y para cada requerimiento crea el comando adecuado el que es transferido al procesador de los comandos.
  - **Procesador de comandos o *invoker*:** Recibe los comandos, agenda e inicia su ejecución. Este módulo es independiente de cada comando, en cuanto sólo utiliza la interfaz genérica que cada comando debe respetar.
  - **Proveedor o receptor:** Provee la funcionalidad del comando en sí ya que la lógica de ejecución del comando incluye el llamado uno o varios del los servicios entregados por el proveedor.

  El diagrama de la figura 2.10 detalla las relaciones entre los diferentes módulos o clases del patrón. La dinámica de la arquitectura se rescata en en la figura 2.11 que corresponde a un diagrama de colaboración entre los elementos de la estructura descrita.
- **Consecuencias:** La aplicación de este patrón de diseño implica los siguientes beneficios:
  - Los comandos pueden ser requeridos de manera flexible, según se implemente el controlador. De hecho podrían existir más de un controlador o cliente generando el mismo comando desde diferentes modos.
  - La aplicación es fácil de modificar y extender a través de la implementación de nuevos comandos dado que el procesador de comandos sólo trabaja con la interfaz genérica del comandos. Es posible agregar comandos más complejos combinando los ya existentes en una especie de macro.
  - El procesador de comandos al centralizar la gestión de estos es el lugar adecuado para implementar servicios como:registro de comandos ejecutados; filtrar la ejecución de comandos; posponer, repetir o deshacer la ejecución de comandos; o gestionar un sistema de prioridades entre comandos.

  Entre las ventajas y limitaciones de este diseño conviene mencionar:
  - La cantidad de comandos puede crecer considerablemente aumentando la complejidad de la aplicación.
  - La retroalimentación entre la ejecución del comando y el cliente que lo crea puede ser limitada.
  - Al desacoplar el momento de generar el comando con el momento de la ejecución en sí, pueden existir limitaciones en implementar aplicaciones basadas en eventos pues el comando obtendrá algunos parámetros de la aplicación sólo durante su ejecución.

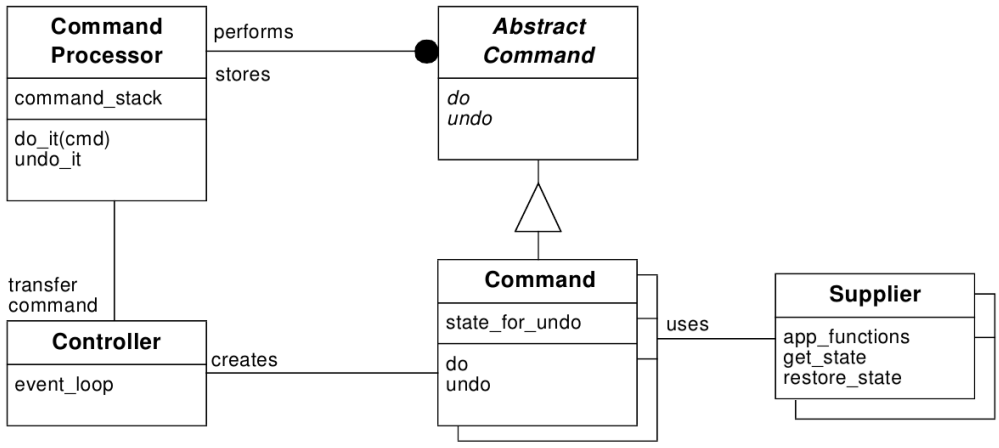


Figura 2.10: Diagrama de clases del patrón de diseño procesador de comandos.

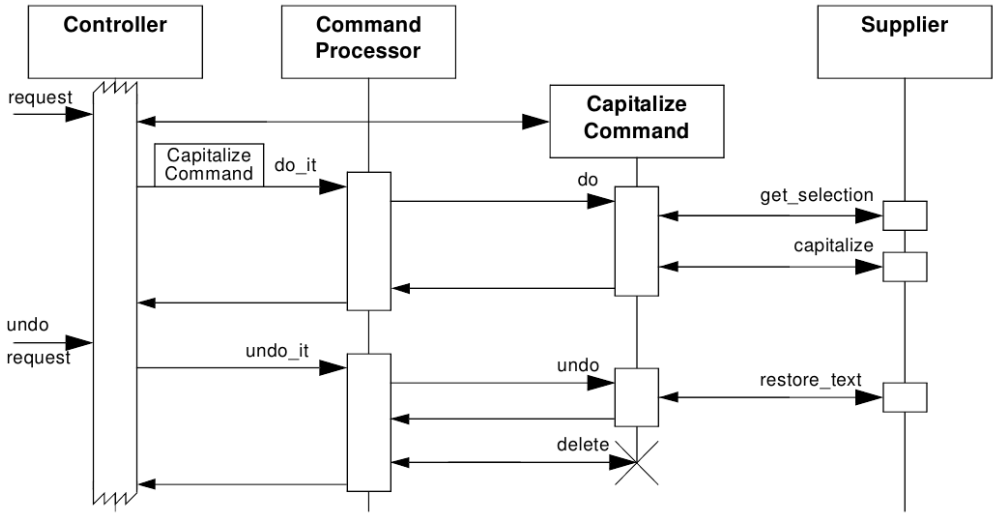


Figura 2.11: Esquema de colaboración del patrón de diseño procesador de comandos.

2.1.4. 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 2.12 (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 2.12 (b) detalla la estructura de un P-POD.

(a) Cubesat (b) P-POD

Figura 2.12: Satélite tipo Cubsat

Estándar. Las restricciones que fija el estándar Cubesat se detallan formalmente en las especificaciones de diseño desarrolladas por *California Polytechnic State University*[**?**] 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 2.13
  - El cubo debe tener dimensiones de 100x100x113 [mm] en x,y,z respectivamente según la figura 2.13
  - Los componentes no deben sobresalir más de 6.5 [mm] en dirección normal a cada cara.
  - Sólo los rieles exteriores de la estructura pueden tener contacto con el P-POD.
  - El satélite no debe superar los 1.33 [kg] de masa.
  - La estructura externa debe estar construida en aluminio 7075 o 6061.
- **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.

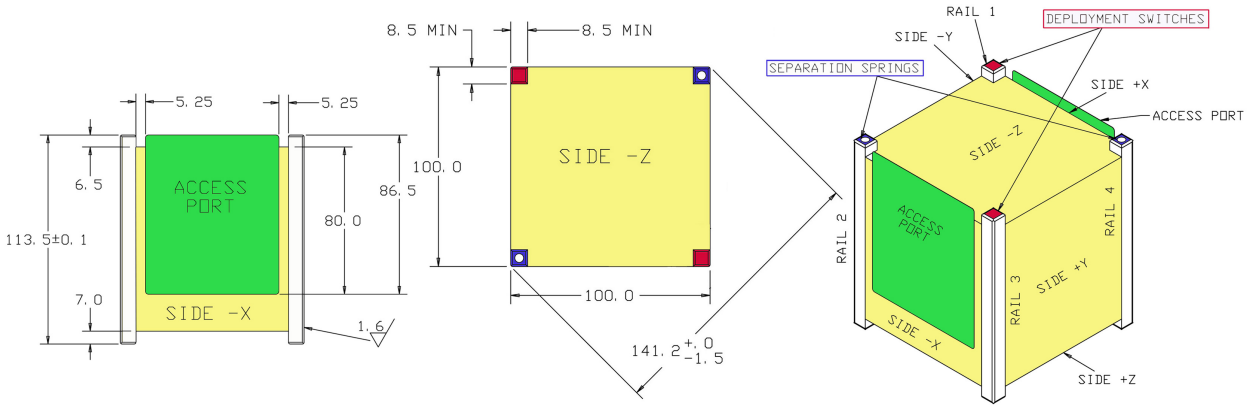


Figura 2.13: Especificaciones de diseño del estándar Cubsat

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 utilizando componentes comerciales o desarrollos propios. El valor educacional de un proyecto satelital de estas características incluye: el desarrollo de nueva tecnología aeroespacial, que puede ser probada a pequeña escala en ambientes realistas; la formación de profesionales entrenados en el área mediante la experiencia práctica que brinda la ejecución de este tipo proyectos; la aplicación de metodologías en la formación académica que incluyan el trabajo en equipos multidisciplinarios.

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. Entre las aplicaciones más destacadas se encuentra la observación remota de la tierra mediante el uso de sensores para

tomar datos sobre la ionosfera y termosfera, o el uso de cámaras fotográficas que provean imágenes para un estudio posterior. Este tipo de misiones y sus datos pueden ser llegar incluso a poner a prueba sistemas de predicción o alerta temprana de desastres[**?**].

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

2.2. Proyecto SUCHAI

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

Figura 2.14: Logo del proyecto SUCHAI

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

El satélite se compone de tres sistemas básicos: el computador a bordo, que consiste en un microcontrolador de gama media para ejecutar el *software* que controla la operación del satélite en vuelo; el sistema de comunicaciones compuesto por un transmisor y receptor de radio UHF así como antenas a desplegar durante la operación; y un sistema de control de energía o EPS que incluya baterías y panales solares para proveer la energía eléctrica que permite operar al satélite así como cargar las baterías. Como carga útil se considera la integración de: una cámara digital que pueda realizar tomas de la tierra; sensores de temperatura; giróscopos; un sensor para medir la densidad y temperatura de la ionosfera

tipo *langmuir probe*; dispositivo para realizar un estudio estadístico de la transferencia de potencia en ambientes de microgravedad y la disipación de calor de la electrónica en ambientes escasos de aire.

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[?].

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

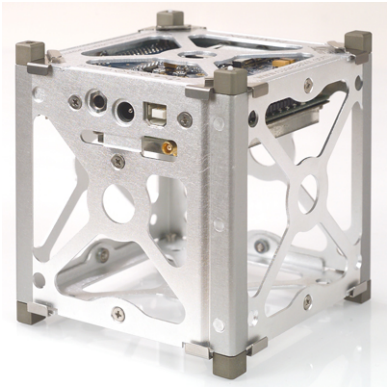
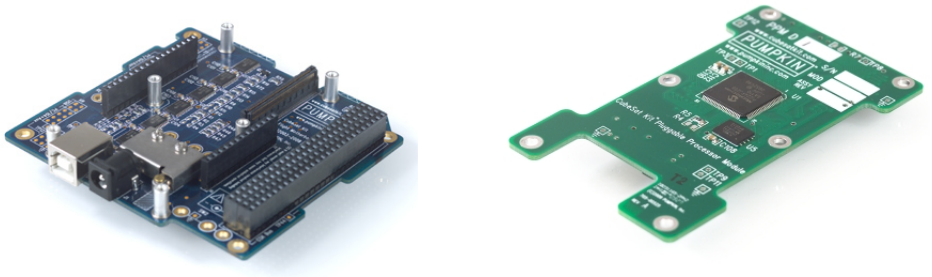


Figura 3.1: Chasis Pumpkins para Cubesat de 1U

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[**?**].



(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[**?**]:

- 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[?]. 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.

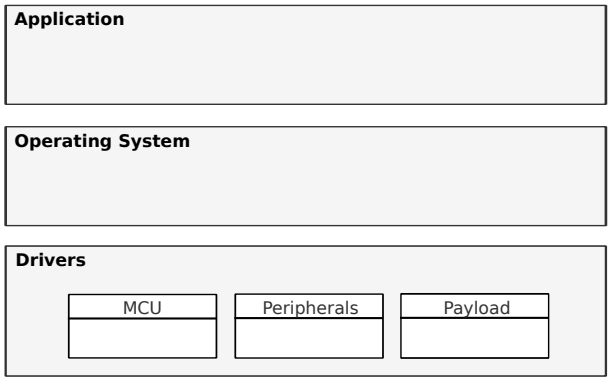


Figura 3.3: Arquitectura de tres capas para un sistema embebido.

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[**?**] 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[**?**].

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.

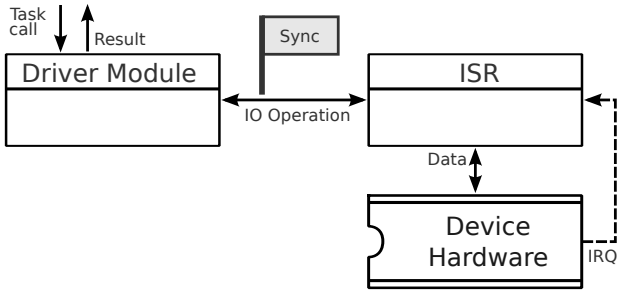


Figura 3.4: Arquitectura para controladores síncronos

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.

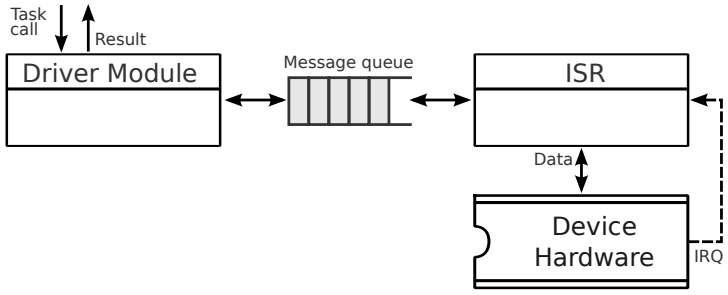


Figura 3.5: Arquitectura para controladores asíncronos

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:

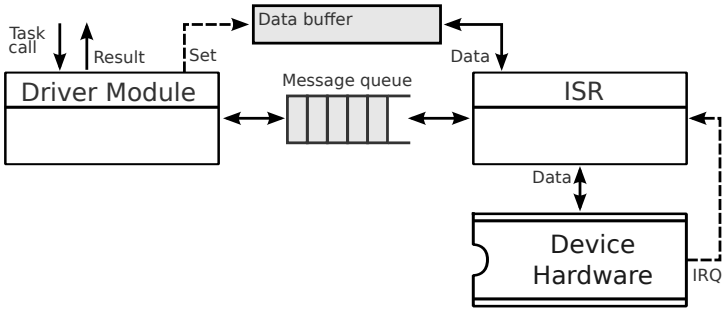


Figura 3.6: Arquitectura de un controlador de entrada serial

- 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[**?**]. 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.

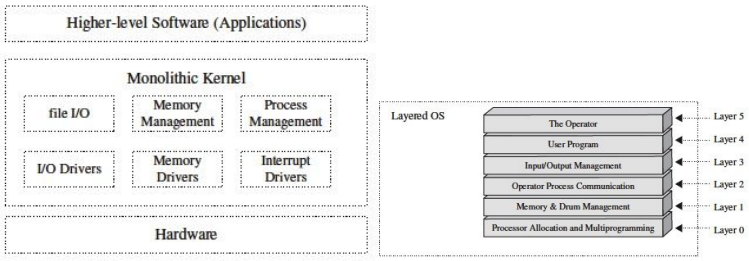
En cuanto a la arquitectura de software presente en un sistema operativo, se pueden encontrar al menos tres opciones bien conocidas: kernel monolítico, arquitectura en capas y micro kernel[**?**].

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

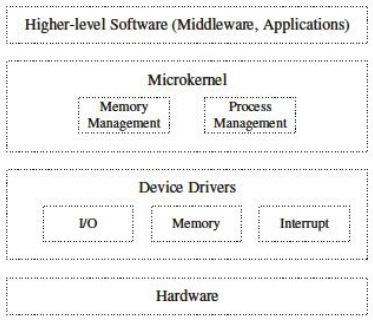
usan esta arquitectura son: Linux, Jbed RTOS y MicroC/OS-II.

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

momento. Los *listeners* se pueden ver como las 'aplicaciones' o 'procesos' de otras arquitecturas de software para pequeños satélites[?][?] en cuanto acá se realizan procesos de manera periódica y se mantienen activos durante todo el funcionamiento del sistema. Estos módulos permiten extender las funcionalidades del satélite en cuanto se requiera 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.

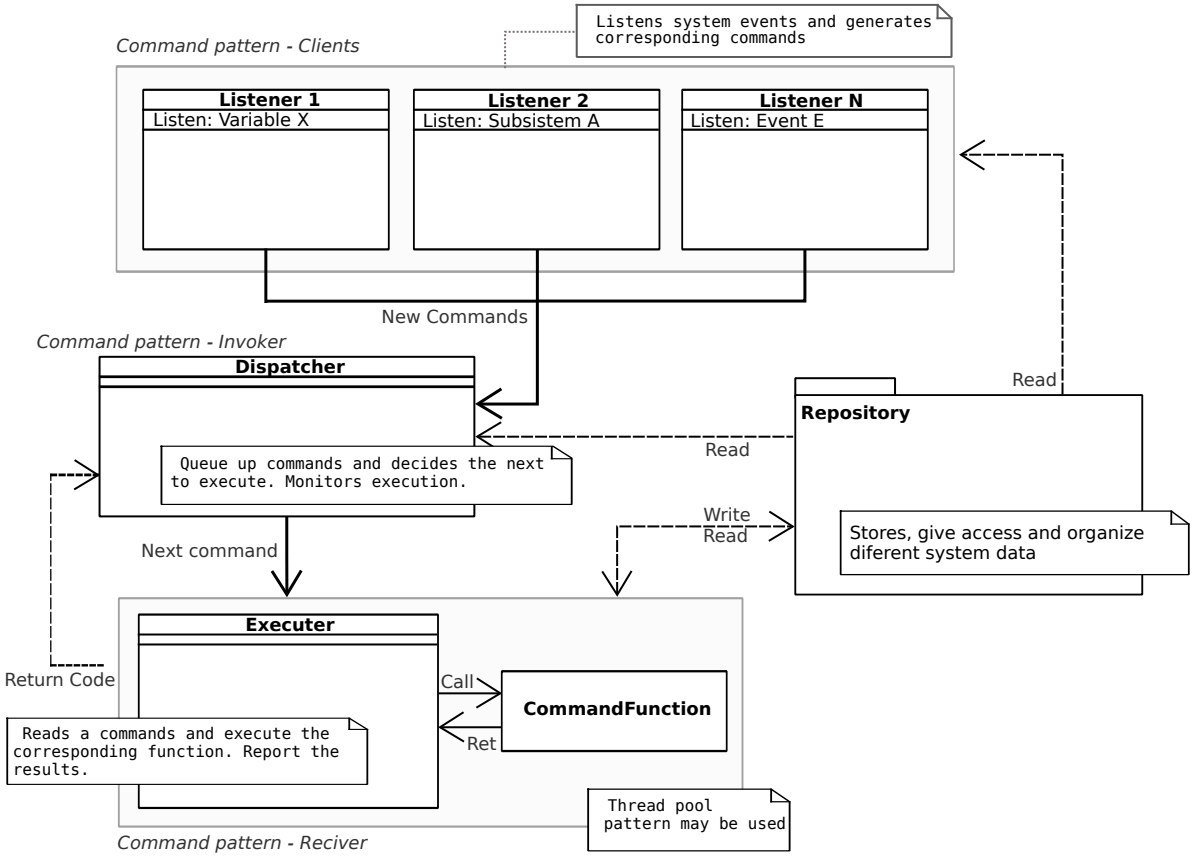


Figura 3.8: Arquitectura de software para el control del satélite

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 implementar son:

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.

- **Communications:** Este *listener* se encarga de controlar los eventos relacionados con el subsistema de comunicaciones, específicamente presta atención a la llegada de telecomandos que implican la ejecución de comandos en el sistema.

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.

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

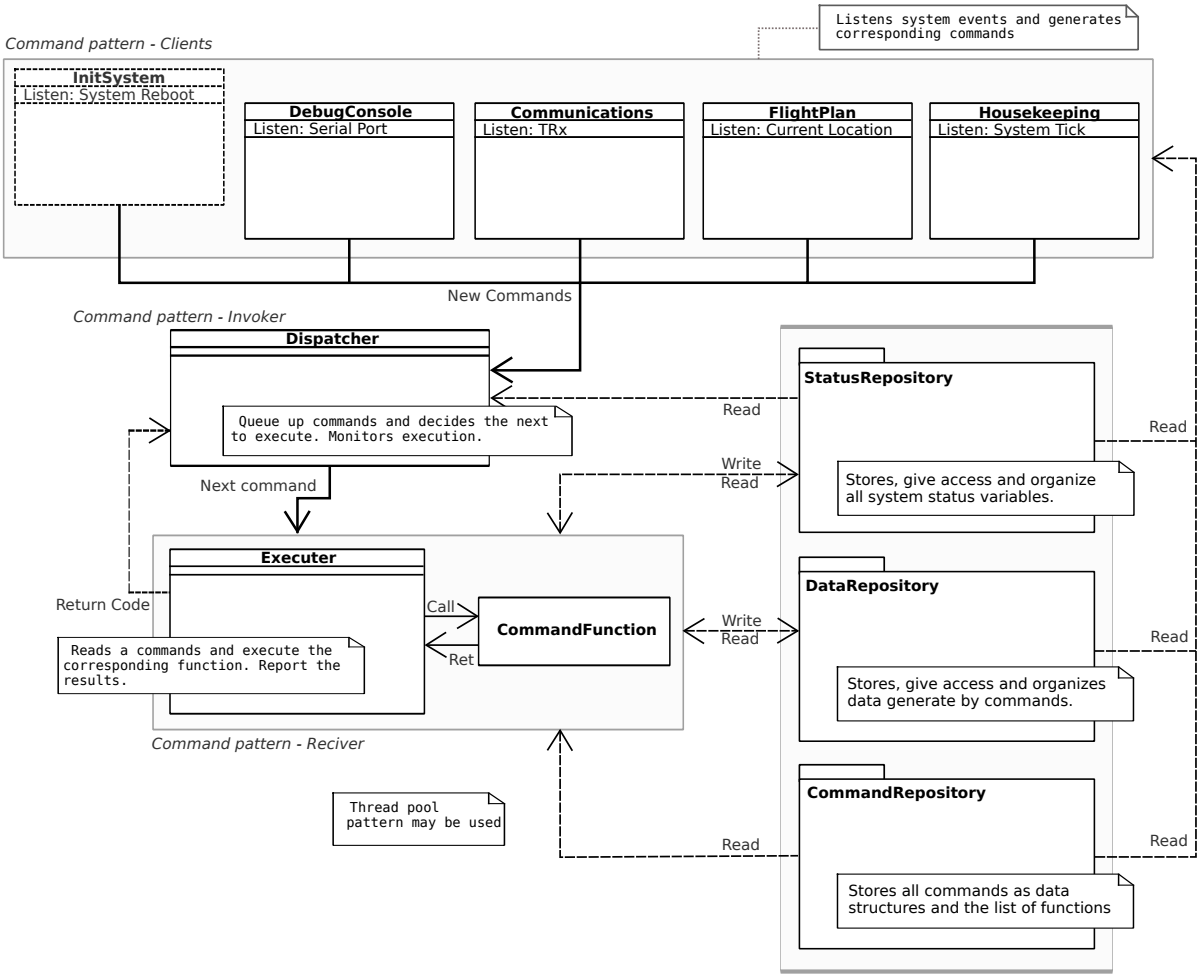


Figura 3.9: Arquitectura de software para el control del satélite

Capítulo 4

Implementación

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

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

4.1. Ambiente de desarrollo

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

Computadores. En general para el desarrollo de software no se tienen requerimientos de hardware elevados, considerando que bastaría con ejecutar un procesador de texto, una aplicación de linea de comandos para ejecutar el compilar y la disponibilidad de al menos un puerto USB para utilizar el programador del microcontrolador. La aplicación más demandante en recursos es el IDE (ver 4.1) cuyos requerimientos recomendados de hardware se encuentran en la tabla 4.4.

Tabla 4.1: Requerimientos de hardware recomendados para desarrollo

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

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

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

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

editor de texto, con funcionalidades de control de cambios locales, múltiples configuraciones para un mismo proyecto, integración con múltiples compiladores y acceso directo a la programación del dispositivo todo desde la misma aplicación centralizando todo el proceso de desarrollo en un ambiente adecuado.

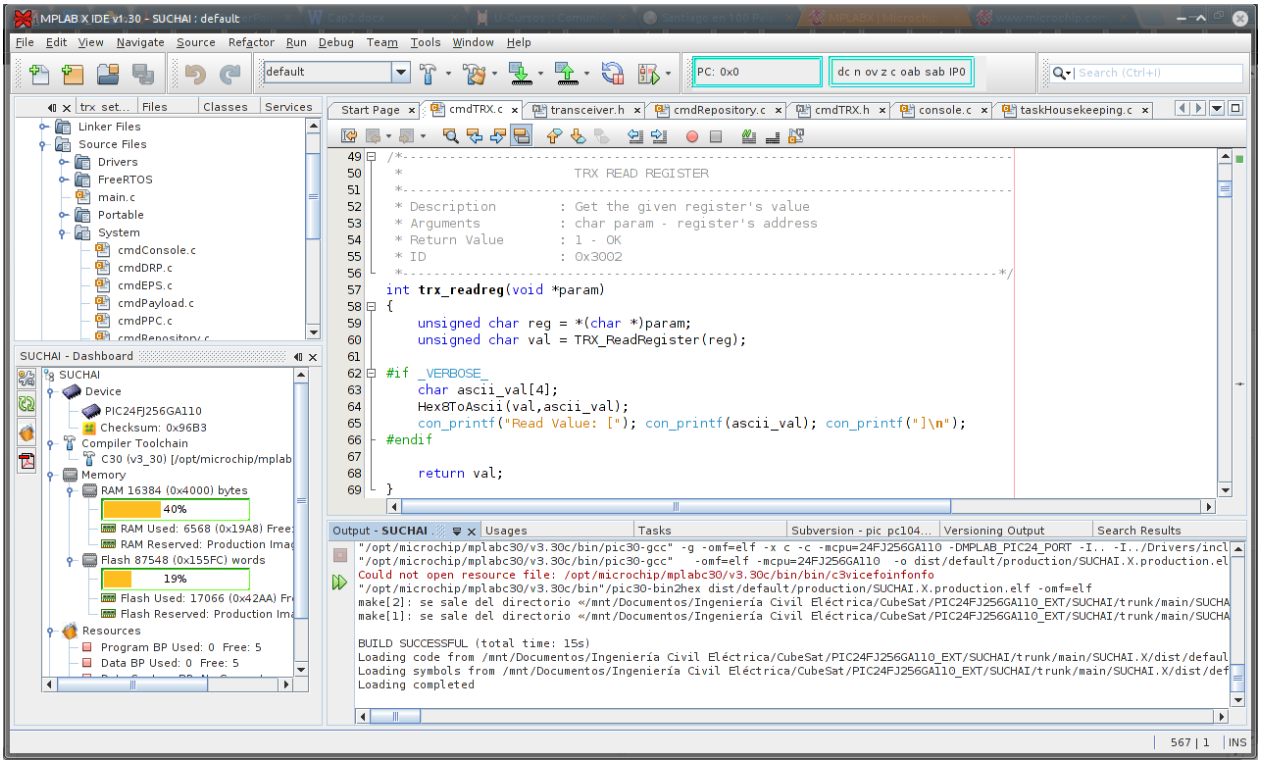


Figura 4.1: Entorno de desarrollo integrado MPLABX

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

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

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

y pruebas del sistema. Se debe hacer hincapié en lo fundamental de esta herramienta en el proceso de desarrollo de un sistema embebido debido a que: la aplicación compilada es específica para el dispositivo objetivo; las herramientas de simulación de microcontroladores no son suficientes para testear las reales condiciones de ejecución de la aplicación; y porque el ciclo de desarrollo se completa con la resolución y ajuste de problemas observados durante la ejecución de la aplicación en su sistema objetivo y de manera dinámica como resultado de respuestas a entradas no deterministas.

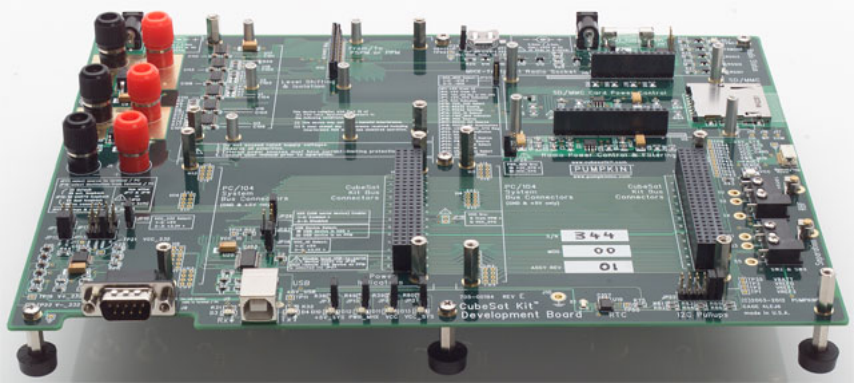


Figura 4.2: Tarjeta de desarrollo para Cubesat Kit de Pumpkins

4.2. Organización del proyecto

4.2.1. Directorios

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

```
+-main/
|
+-Drivers/
| |
| +-include/
|
+-<RTOS>/
|
+-Payloads/
| |
| +-Cmd/
| | |
| | +-include/
| |
```

```
| +-Drivers/
|   |
|   +-include/
|
+-<Proyecto>.X/
|
+-System/
| |
| +-include/
|
|_main.c
```

Los desarrollos deben seguir esta estructura al momento de agregar archivo con código fuente al sistema. En la tabla 4.2 se detalla la funcionalidad de cada directorio.

Tabla 4.2: Organización de directorios del proyecto

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

4.2.2. IDE

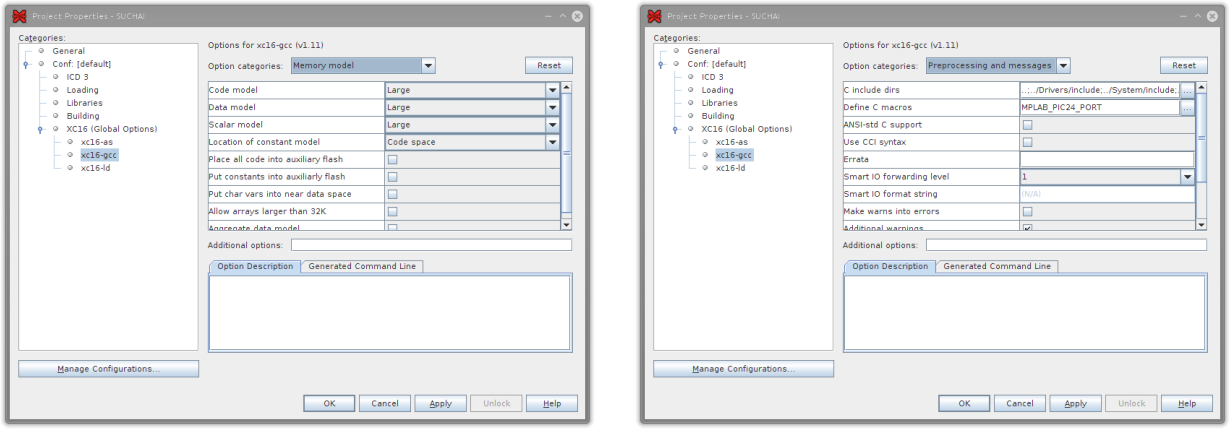
Esta estructura de directorios creada es la base para configurar adecuadamente los archivos con las fuentes del proyecto en el IDE, en este caso MPLAB X. Para la correcta construcción de software en el IDE se deben ajustar las configuraciones del compilador según las indicaciones de la tabla 4.3 (parámetros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

Ejemplos del dialogo para configurar las opciones del compilador en MPLAB X se detallan

Tabla 4.3: Configuración del compilador XC16

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

en la figura 4.3.



(a) (b)

Figura 4.3: Diálogo de configuración del compilador XC16 en MPLABX

4.2.3. Documentación

4.3. Controladores de hardware

La primera capa de la aplicación corresponde a una serie de módulos que implementan los controladores de hardware. En este caso un módulo significa una librería que consiste en una serie de funciones escritas en lenguaje C las cuales acceden a funcionalidades específica de cierto hardware respetando sus protocolos o API. La librería consta entonces de su correspondiente archivo fuente con extensión `.c` y un archivo de cabecera con extensión `.h` que contiene la declaración de las funciones.

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

1. Programar en lenguaje ensamblador (*assembler*). Se maneja directamente el juego de instrucciones del procesador (también conocido como lenguaje máquina) para configurar sus registros, manejar periféricos o ejecutar un programa general. Requiere de un compilador de *assembler* para generar el ejecutable binario.

2. Utilizar un compilador en C. El compilador provee un nivel mayor de abstracción al permitir programar en un lenguaje de alto nivel y portable como C. Además provee librerías básicas para las funciones específicas de cada dispositivo.

3. Utilizar una librería externa. Un controlador de hardware puede requerir los servicios de otro controlador para su funcionamiento. Es el caso típico de dispositivos que utilizan algún protocolo de comunicación (RS232, SPI, I2C) y su controlador consiste en implementar una API de llamadas sobre este protocolo.

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

4.3.1. Microcontrolador

4.3.2. Periféricos

En la sección **??** se revisaron los periféricos disponibles en el PIC24F256GA110 en la tabla **??** se detalla la lista de drivers a implementar y la arquitectura que se utilizará en cada uno de ellos.

Tabla 4.4: Drivers para periféricos del microcontrolador

Periférico	**Arquitectura**
Timers	
PWM	
I2C	
SPI	
UART	
Input Capture	
RTCC	
CRC	
Comparadores	
ADC	
hline	

Implementación de un controlador síncrono

Implementación de un controlador asíncrono

4.4. Sistema operativo

En la capa de sistema operativo se ha optado por utilizar FreeRTOS. Este sistema operativo está diseñado específicamente para sistema embebidos y provee la capacidad de implementar tareas que son módulos de software que funciona de manera concurrente y puede compartir información a través de diferentes estructuras de sincronización. FreeRTOS soporta una gran variedad de microcontroladores, entre los cuales se encuentra la familia PIC24F, a través de *ports* y aplicaciones demo que se obtienen al descargar el software. FreeRTOS consiste básicamente de 5 archivos fuentes en lenguaje C (sólo tres son necesarios para la utilidad básica), 11 archivos de cabecera y una capa portable dependiente del dispositivo sobre el cual se trabaja. Fuera de los demos y diferentes ports incluidos con la descarga el siguiente árbol de directorio se agrega a la carpeta del proyecto y en la configuración del proyecto en MPLABX:

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

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

Tabla 4.5: Configuración del compilador XC16 para FreeRTOS

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

El primer paso para integrar el sistema operativo es crear el archivo de cabecera con su configuración llamado `FreeRTOSConfig.h`. El archivo se compone de una serie de *defines* que cambian el comportamiento del sistema operativo y lo ajustan a las necesidades de la aplicación. Una plantilla con los posibles valores a configurar se encuentra en la página web de FreeRTOS: `http://www.freertos.org/a00110.html`. Un ejemplo de este archivo se detalla en el código 4.1

En segundo lugar se deben crear las tareas que ejecutará el sistema. En FreeRTOS una tarea es una función con una firma específica y que por lo general entrará en un ciclo de permanente para mantener su ejecución en el tiempo. Toda tarea debe poseer la siguiente firma: `void taskName(void *)`, donde el nombre de la función varia de tarea en tarea, pero siempre debe retornar *void* y recibir un puntero *void* como parámetro. Un simple prototipo de tarea en FreeRTOS se detalla en el código 4.2. La tarea se denomina `taskTest` y recibe a través de su parámetro una cadena de texto. La tarea entra en un ciclo e imprime de manera periódica el *string* entregado como parámetro. Aunque la tarea está en un ciclo infinito, no se encuentra en una situación de *busy waiting* dado que utiliza la función `vtaskDelay` que detiene la ejecución de la tarea durante un periodo de tiempo liberando los recursos del procesador; en este caso se dice que la tarea se encuentra dormida.

Una vez programada la función que corresponde a la tarea, se puede configurar el sistema para que la ejecute. Esto se realiza en el archivo `main.c` donde se realizan las configuraciones del hardware, ser cargan las configuraciones del sistema operativo contenida en el archivo `FreeRTOSConfig.h`, se crean las tareas y se inicia el sistema operativo. Para crear tareas se utiliza la función `xTaskCreate` indicando la función a ejecutar, un nombre, prioridad, memoria asignada y parámetros; para detalles referirse a la documentación de FreeRTOS[**?**]. El sistema operativo se inicia con la función `vTaskStartScheduler()` y una vez alcanzado este punto, el control de los procesos a ejecutar queda en manos de FreeRTOS quien según su algoritmo de `scheduling` seleccionará la tarea que debe ejecutarse en cada instante. El llamado a la función `vTaskStartScheduler` no retorna a menos que se produzca un error en la ejecución del sistema operativo. El el código 4.3 se ilustra la forma de iniciar FreeRTOS con dos tareas ejecutándose de manera simultanea, para mantener la generalidad no se incluyen configuraciones de hardware específicas.

Listing 4.1: FreeRTOSConfig.h

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#include <xc16.h>

#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             1
#define configUSE_TICK_HOOK             0
#define configTICK_RATE_HZ              250
#define configCPU_CLOCK_HZ              16000000
#define configMAX_PRIORITIES            4
#define configMINIMAL_STACK_SIZE        115
#define configTOTAL_HEAP_SIZE           5120
#define configMAX_TASK_NAME_LEN         8
#define configUSE_TRACE_FACILITY        0
#define configUSE_16_BIT_TICKS          1
#define configIDLE_SHOULD_YIELD         1

#define INCLUDE_vTaskPrioritySet        1
#define INCLUDE_uxTaskPriorityGet       0
#define INCLUDE_vTaskDelete             0
#define INCLUDE_vTaskSuspend            1
#define INCLUDE_vTaskDelayUntil         1
#define INCLUDE_vTaskDelay              1

#define configKERNEL_INTERRUPT_PRIORITY 0x01

#endif /* FREERTOS_CONFIG_H */
```

Listing 4.2: taskTest.c

```
#include taskTest.h

void taskTest(void *param)
{
    const unsigned long Delayms = 500 / portTICK_RATE_MS;
    char *msg = (char *)param;

    while(1)
    {
        vTaskDelay(Delayms);
        printf("[taskTest] %s\n", msg);
    }
}
```

Listing 4.3: main.c

```
1  /* RTOS Includes */
2  #include "FreeRTOSConfig.h"
3  #include "FreeRTOS.h"
4  #include "task.h"
5
6  /* Task includes */
7  #include "taskTest.h"
8
9  int main(void)
10 {
11     /* Crating all task */
12     xTaskCreate(taskTest, (signed char*)"taskTest",
           configMINIMAL_STACK_SIZE, (void *)"T1 Running...", 1, NULL);
13     xTaskCreate(taskTest, (signed char*)"taskTest",
           configMINIMAL_STACK_SIZE, (void *)"T2 Running...", 2, NULL);
14
15     /* Start the scheduler. Should never return */
16     printf(">>Starting FreeRTOS\n");
17     vTaskStartScheduler();
18
19     /* Never get here */
20     return 0;
21 }
```

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

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

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

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

4.5. Aplicación

En la capa de aplicación se debe implementar la arquitectura de software detallada en la figura 3.9 que permite el cumplimiento de todos los requerimientos de operacionales del satélite. El proceso incluye interpretar el patrón de diseño que inspira la arquitectura base para ser implementado en un lenguaje *procedural* como C. Conviene también implementar la arquitectura base del patrón de ejecutor de comandos para contar con un sistema base que sea general y bien probado sobre el cual implementar las funcionalidades específicas del proyecto. Las funciones específicas de un proyecto satelital estarán implementadas en la capa de *listeners* y la lista de comandos por lo que finalmente se detalla la implementación de los distintos módulos que competan los requerimientos del proyecto SUCHAI en específico.

4.5.1. Implementación del patrón de diseño

Cómo se describe en la sección 2.1.4 se debe implementar un patrón de diseño, específicamente *Command Pattern* en un lenguaje procedural como C. El problema radica en la programación basada en patrones de diseño es una técnica utilizada principalmente en lenguajes de programación orientados a objetos y los patrones se describen según diagramas de colaboración entre objetos incluyendo técnicas como herencia o polimorfismo. No obstante el diseño de una arquitectura de software debería ser independiente del lenguaje de programación a utilizar, así como también, el diseño de la arquitectura basándose en patrones[**?**]. Para sortear esta situación se procede a definir cómo se traduce cada elemento del patrón de diseño deseado a la plataforma objetivo.

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

Listing 4.4: Prototipo de un comando

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

- **Repositorios:** Los repositorios y proveedores de servicios se implementaran como librerías que acceden a funciones a las capas inferiores del sistema como drivers a submódulos o dispositivos de entrada y salida.

4.5.2. Comandos

Los comandos se han implementado como funciones que deben respetar una firma específica que se convierte en un un nuevo tipo de dato. En este caso la función debe retornar un entero y recibir un parámetro, como se detalla en el código 4.4, lo que define dos aspectos fundamentales del software:

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

Para encapsular la información asociada a los comandos se crea una nueva estructura de datos definida como en el código 4.5 que representa a un comando que es entregado al módulo *executer*. La estructura contiene dos campos: el puntero a la función que implementa el comando según la definición del código 4.4; y el parámetro del comando.

Con el objetivo de implementar una estrategia simple y flexible que permita tanto generar comandos así como determinar la función que se asocia a cada comandos se ha definidos una nueva estructura de datos que representa a los comandos que son generados por los *listeners* cuya definición se detalla en el código 4.6. Si se utiliza la estructura definida en el código 4.5 significa que cada *listener* debe conocer todos los comandos disponibles en el sistema y cada vez que se requiere generar un comando se debe agregar a mano la función que corresponde en la estructura. Esto genera inconvenientes ya que al separar la definición de cada comando

Listing 4.5: Estructura de comandos para *executer*

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

Listing 4.6: Estructura de comandos para *dispatcher*

```
/**
 * Structure that represent a command passed to dispatcher. Contains
 * only a code that represent the function to call, a paremeter and
 * other command's metadata
 */
typedef struct ctrl_command{
    int cmdId;      ///< Command id, represent the desired command
    int param;      ///< Command parameter
    int idOrig;     ///< Metadata: Id of sender subsystem
    int sysReq;     ///< Metadata: Level of energy the command requires
}DispCmd;
```

archivos diferentes se debe realizar una serie de *includes*, también hace poco automatizado la generación de comandos en serie y no es una forma práctica de generar comandos de manera remota.

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

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

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

Listing 4.7: Ejemplo de comando

```
/**
 * Performs debug taks over current RTOS. Get rtos memory usage in bytes
 *
 * @param param Not used
 * @return Availible heap memory in bytes
 */
int obc_get_rtos_memory(void *param)
{
    size_t mem_heap = xPortGetFreeHeapSize();
    printf("Free RTOS memory: %d", mem_heap);

    return mem_heap;
}
```

La implementación de un comando en específico requiere la definición de una función que cumpla con la firma definida en el código 4.4, un ejemplo es el código 4.7 donde se implementa el comando `get_rtos_memory`; notar que este comando forma parte del archivo `cmdOBC.c`, por lo tanto cada función y definición incluye el prefijo `obc_` como parte del estándar de código utilizado en el proyecto. Este comando que será utilizado como ejemplo es, sin embargo, una importante funcionalidad dentro del software de vuelo que nos permitirá obtener información sobre el uso de memoria del sistema operativo. El objetivo es utilizar la función `xPortGetFreeHeapSize` de FreeRTOS que entrega la cantidad de memoria en *bytes* disponible en el *heap*[**?**] y desplegarla en consola. La convención indica que retornar un valor cero desde el comando significa error lo cual demuestra su conveniencia en este caso pues el comando puede retornar directamente la cantidad de memoria disponible.

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

1. Un arreglo único de comandos centralizado en el repositorio de comandos. Se llena manualmente un arreglo de punteros a funciones tipo *cmdFunction* y para registrar un nuevo comando se agrega dentro del arreglo.
    - **Ventajas:** Se puede ahorrar memoria RAM creando un arreglo de tipo *const* que se almacena en memoria de programa.
    - **Desventajas:** Complejo seguir cambios en el orden en que se registran los comandos y por lo tanto su código asociado. Ofrece pocas posibilidades de agrupar comandos según funciones. Alto impacto al agregar un nuevo comando pues implica la modificación de múltiples archivos fuente, incluyendo el repositorio de comandos.
2. Varios arreglos con comandos centralizados en el repositorio de comandos. Se agrupan comandos con funcionalidades relacionadas en varios arreglos, así cuando se agrega

un nuevo comando sólo se modifica el arreglo relacionado a una determinada serie de comandos.

- **Ventajas:** Mejor agrupación de los comandos, localizando cambios. Los códigos numéricos se pueden diferenciar por grupo.
- **Desventajas:** Agregar un comando requiere modificar varios archivos, incluyendo el archivo donde se implementa y el repositorio de comandos.

3. Descentralizar cada arreglo de comandos en el archivo correspondiente. Cada archivo que implementa un grupo de comandos cuenta con un arreglo con las funciones. El repositorio de comandos utiliza este arreglo como una variable externa.
   - **Ventajas:** Mejor agrupación de comandos. Bajo impacto en el código al agregar un comando, pues sólo se modifica el archivo dónde se encuentra el comando sin intervenir el repositorio de datos.
   - **Desventajas:** Se dificulta el seguimiento de cambios en los códigos de cada comando. Agregar un nuevo grupo de comandos, en un nuevo archivo, requiere modificaciones de código en el repositorio de comandos.
4. Descentralizar comandos y utilizar `enums` para asignar códigos. Similar a la alternativa anterior pero los códigos de cada comando y la forma de llenar los arreglos de comandos se basan en una estructura de enumeración de C que abstrae el uso de códigos numéricos por sentencias textuales.
   - **Ventjas:** Agregar un comando sólo requiere cambios locales al archivo que agrupa un determinado tipo de comandos. Se facilita el seguimiento de cambios en los comandos disponibles pues los la definición textual de la enumeración no cambia.
   - **Desventajas:** Aún se requiere modificar el repositorio de comando cuando se agrega un archivo con un nuevo grupo de funciones. Al completar los arreglos de funciones a través de las enumeraciones no se puede crear un arreglo a mano en memoria de datos, se requiere una función que inicialice los arreglos cada vez que se inicia el sistema.

Todas las estrategias mencionadas fueron implementadas en algún momento, sin embargo, la cuarta alternativa ha demostrado ser la opción más flexible y conveniente. De este modo el proceso de registro del comando creado sólo requiere modificaciones en los archivos relacionados a su implementación, en este caso `cmdOBC.c` y `cmdOBC.h`.

Cada archivo con un grupo de comandos debe tener una estructura de enumeración (`enum` en C) que representan los códigos de cada comando. El primer valor de la enumeración debe ser diferente para cada grupo de comandos para no generar ambigüedades y para agrupar los comandos también por códigos. El último valor de la enumeración es un valor *dummy* que sólo es utilizado para controlar el tamaño del arreglo de comandos. Luego, por cada función que implemente un comando se agrega un valor en la enumeración, como en el código 4.8.

En este caso se registran dos comandos, `obc_reset` y `obc_get_rtos_memory`, la convención de sintaxis utilizada indica que se debe utilizar el prefijo del grupo de comandos `obc_` y un prefijo que indique que se trata de un código de comando `id_`. El código 4.8 también muestra como se utiliza Doxygen para mantener en la documentación del proyecto una lista actualizada con el valor de cada enumeración que permita de manera directa asociar el código de un comando cuando estos se generen de manera remota.

Listing 4.8: Lista de comandos disponibles

```
/**
 * List of availible commands
 */
typedef enum{
    obc_id_reset = 0x1000,  ///< @cmd_first
    obc_id_get_rtos_memory, ///< @cmd

    ppc_id_last_one    // Dummy element
}OBC_CmdIndx;
```

Listing 4.9: Registro de comandos

```
cmdFunction obc_Function[OBC_NCMD];

/**
 * This function registers the list of command in the system,
 * initializing the functions array. This function must be called
 * at every system start up.
 */
void obc_onResetCmdOBC(void)
{
    obc_Function[(unsigned char)obc_id_reset] = obc_reset;
    obc_Function[(unsigned char)obc_id_get_rtos_memory] =
       obc_get_rtos_memory;
}
```

También en el archivo `cmdOBC.c` se debe crear el arreglo que contiene la lista de funciones, en este caso denominado `obc_Function` cuyo tamaño se determina a través del último elemento de la enumeración. Esta lista debe ser inicializada a través de una función que se ejecutará al inicio del sistema, en este caso se denomina `obc_onResetCmdOBC`. El código 4.9 detalla este proceso.

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

4.5.3. Repositorio de comandos

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

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

Inicializar el repositorio de comandos significa llamar a la función de inicialización presente en cada archivo donde se implementan las funciones, esto se realiza en la función `repo_onResetCmdRepo` disponible en el código 4.10.

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

```
                          |>> N° de comando
Comando ->          0 x A A B B
            N° de grupo <<|
```

Por esta razón la función `repo_getCmd`, encargada de retornar la función asociada a un código de comando, divide este número para identificar el arreglo desde el cual obtener el comando y la posición dentro del arreglo que ocupa el puntero a la función buscada. Esto se implementa en el código 4.10.

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

4.5.4. Repositorios de estados

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

Las operaciones básicas que provee el repositorio son: leer una variable de estado, lo cual se realiza en la función `dat_setCubesatVar`; escribir el valor de una variable de estado, lo cual se realiza en la función `dat_getCubesatVar`; e inicializar el repositorio de estados ante un reinicio del sistema, a través de la función `dat_onResetCubesatVar`. Estas funciones se encuentra implementadas en el código 4.11.

Este repositorio es leído y escrito de manera concurrente por una serie de tareas por lo cual se puede generar una condición de *data race*. Esto se hace mas evidente cuando la implementación del repositorio en una memoria externa puede resultar operaciones de lectura o escritura no atómicas, donde además se requiere utilizar recursos compartidos del sistema

Listing 4.10: cmdRepository.c

```c
/* Add external cmd arrays */
extern cmdFunction obc_Function;

/**
 * Returns a pointer with the function asociated to each cmdID.
 * @param cmdID Command id
 * @return Pointer to function of type cmdFunction
 */
cmdFunction repo_getCmd(int cmdID)
{
    int cmdOwn, cmdNum;
    cmdFunction result;

    cmdNum = (unsigned char)cmdID;
    cmdOwn = (unsigned char)(cmdID>>8);

    switch (cmdOwn)
    {
        case CMD_OBC:
            if(cmdNum >= OBC_NCMD)
                result=cmdNULL;
            else
                result = obc_Function[cmdNum];
            break;

        default:
            result = cmdNULL;
            break;
    }

    return result;
}

/**
 * Initializes all cmd arrays
 * @return 1, allways successful
 */
int repo_onResetCmdRepo(void)
{
    obc_onResetCmdOBC();
    return 1;
}

/**
 * Null command, just print to stdout
 * @param param Not used
 * @return 1, allways successful
 */
int cmdNULL(void *param)
{
    int arg=*( (int *)param );
    printf("cmdNULL was used with param %d\n", arg);
    return 1;
}
```

como periféricos de entrada y salida. Por esta razón se debe implementar una estructura de sincronización que provea la exclusión mutua entre las diferentes tareas que acceden a este repositorio. Esta situación se observa en el código 4.11 cuando se usan las funciones `xSemaphoreTake` y `xSemaphoreGive` de FreeRTOS[?].

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

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

En general cualquier tipo de memoria no volátil se puede utilizar para estos fines, aunque si se utilizan lo métodos 2 o 3, es conveniente utilizar una memoria de acceso aleatorio en

Listing 4.11: dataRepository.c

```c
#include "dataRepository.h"

extern xSemaphoreHandle dataRepositorySem;  // Mutex for status repository
int DAT_CUBESAT_VAR_BUFF[dat_cubesatVar_last_one]; // Internal buffer

/**
 * Sets a status variable's value
 * @param indxVar Variable to set @sa DAT_CubesatVar
 * @param value Value to set
 */
void dat_setCubesatVar(DAT_CubesatVar indxVar, int value)
{
    xSemaphoreTake(dataRepositorySem, portMAX_DELAY);
    DAT_CUBESAT_VAR_BUFF[indxVar] = value;
    xSemaphoreGive(dataRepositorySem);
}

/**
 * Returns a status variable's value
 * @param indxVar Variable to set @sa DAT_CubesatVar
 * @return Variable value
 */
int dat_getCubesatVar(DAT_CubesatVar indxVar)
{
    int value = 0;
    xSemaphoreTake(dataRepositorySem, portMAX_DELAY);
    value = DAT_CUBESAT_VAR_BUFF[indxVar];
    xSemaphoreGive(dataRepositorySem);
    return value;
}

/**
 * Initializes status repository
 */
void dat_onResetCubesatVar(void)
{
    int i;
    for(i=0; i<dat_cubesatVar_last_one; i++)
    {
        dat_setCubesatVar(i,0xFFFF);
    }
}
```

lugar de dispositivos que leen o escriben por bloques ya que el uso de este repositorio es bastante intensivo dentro del diseño del sistema.

El primer método es el más simple en cuanto no requiere de controladores externos para su funcionamiento y la sincronización no es crítica. Por esto y para mantener la generalidad en el código del ejemplo 4.11 se implementa este diseño en el repositorio de estados. Sin embargo el no contar con un almacenamiento persistente de los datos es una limitación crítica ya que este módulo permite que el sistema de vuelo sea tolerante a reinicio inesperados volviendo a funcionar según sus últimos estados almacenados. Este método es factible de utilizar para pruebas o bien como un sistema de respaldo en caso de que una falla fuerce al sistema a trabajar con funcionalidades mínimas.

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

4.5.5. Dispatcher

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

Una parte fundamental de este módulo es la cola de comandos. Los comandos generados por cada *listener* llegan a esta cola en espera de ser procesados. Esta cola es un recurso compartido y presenta el esquema típico del productor-consumidor con múltiples productores y un único consumidor, por lo que el acceso concurrente debe estar sincronizado (ver figura 4.4). La solución a este tipo de esquemas es bien conocido y se logra mediante la utilización de semáforos o mutexes.

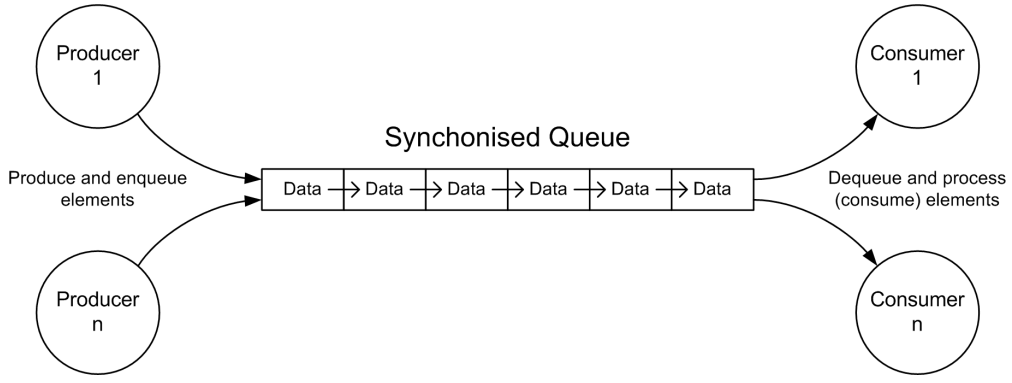


Figura 4.4: Problema del productor-consumidor

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

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

El flujo de trabajo cuando se utilizan colas en FreeRTOS se ilustra en la figura 4.5.

La implementación de este módulo corresponde al diagrama de flujo de la figura 4.6. Las tres funcionalidades básicas del procesador de comandos son: recibir un comandos y obtener sus meta datos; determinar si su ejecución es posible; construir el comando que se entregará al ejecutor. En el código 4.12 se implementan este diagrama de flujo. Se tiene entonces una tarea de FreeRTOS basada en eventos, pues despierta solamente cuándo existe algún comando disponible en la cola y desarma la estructura de datos obtenida. La función `check_if_executable` implementa la lógica que determina si el comando es puede ejecutar o no, en base a los meta datos de comandos y el estado actual del sistema. Finalmente se consulta al repositorio de comandos por la función asociada al código entrega y se construye la estructura de datos que representa un comando para el ejecutor que contiene el puntero a la función y su parámetro. Como sólo existe un proceso que ejecuta los comandos el *dispatcher* debe esperar a que termine su ejecución antes recuperar uno nuevo. Esta comunicación también se logra a través de una cola, de un sólo elemento, donde el ejecutor envía el resultado que retornó la función que acaba de ejecutar.

4.5.6. Executer

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

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

Para la comunicación entre tareas, involucrando el intercambio de mensajes se utiliza nuevamente la estructura `Queue` de FreeRTOS. En este caso se tienen dos mensajes que entregar para sincronizar dos estados del proceso total, por lo tanto se implementan dos colas de largo igual a un elemento. Al ser las colas de largo uno, estas actúan como *mutexes* bloqueando un proceso hasta que el otro haya completado sus operaciones.

Listing 4.12: taskDispatcher.c

```c
#include "taskDispatcher.h"

extern xQueueHandle cmdQueue; /* Commands queue */
extern xQueueHandle executerCmdQueue; /* Executer commands queue */
extern xQueueHandle executerStatQueue; /* Executer result queue */

void taskDispatcher(void *param)
{
    printf(">>[Dispatcher] Started\n");
    portBASE_TYPE status; /* Status of cmd reading operation */

    DispCmd newCmd; /* The new cmd readed */
    ExeCmd exeCmd; /* Strucutre to executer */
    int cmdId, idOrig, sysReq, cmdParam, cmdResult; /* Cmd metadata */

    while(1)
    {
        /* Read newCmd from Queue - Blocking */
        status = xQueueReceive(cmdQueue, &newCmd, portMAX_DELAY);

        if(status == pdPASS)
        {
            /* Gets command metadata*/
            cmdId = newCmd.cmdId;
            idOrig = newCmd.idOrig;
            sysReq = newCmd.sysReq;
            cmdParam = newCmd.param;

            /* Check if command is eecutable */
            if(check_if_executable(&newCmd))
            {
                printf("[Dispatcher] Cmd: %X, Param: %d, Orig: %X\n",
                    cmdId, cmdParam, idOrig);

                /* Fill the executer command */
                exeCmd.fnct = repo_getCmd(newCmd.cmdId);
                exeCmd.param = param;

                /* Send the command to executer Queue - BLOCKING */
                xQueueSend(executerCmdQueue,&exeCmd,portMAX_DELAY);

                /* Get the result from Executer Stat Queue - BLOCKING */
                xQueueReceive(executerStatQueue,&cmdResult,portMAX_DELAY);
            }
        }
    }
}
```

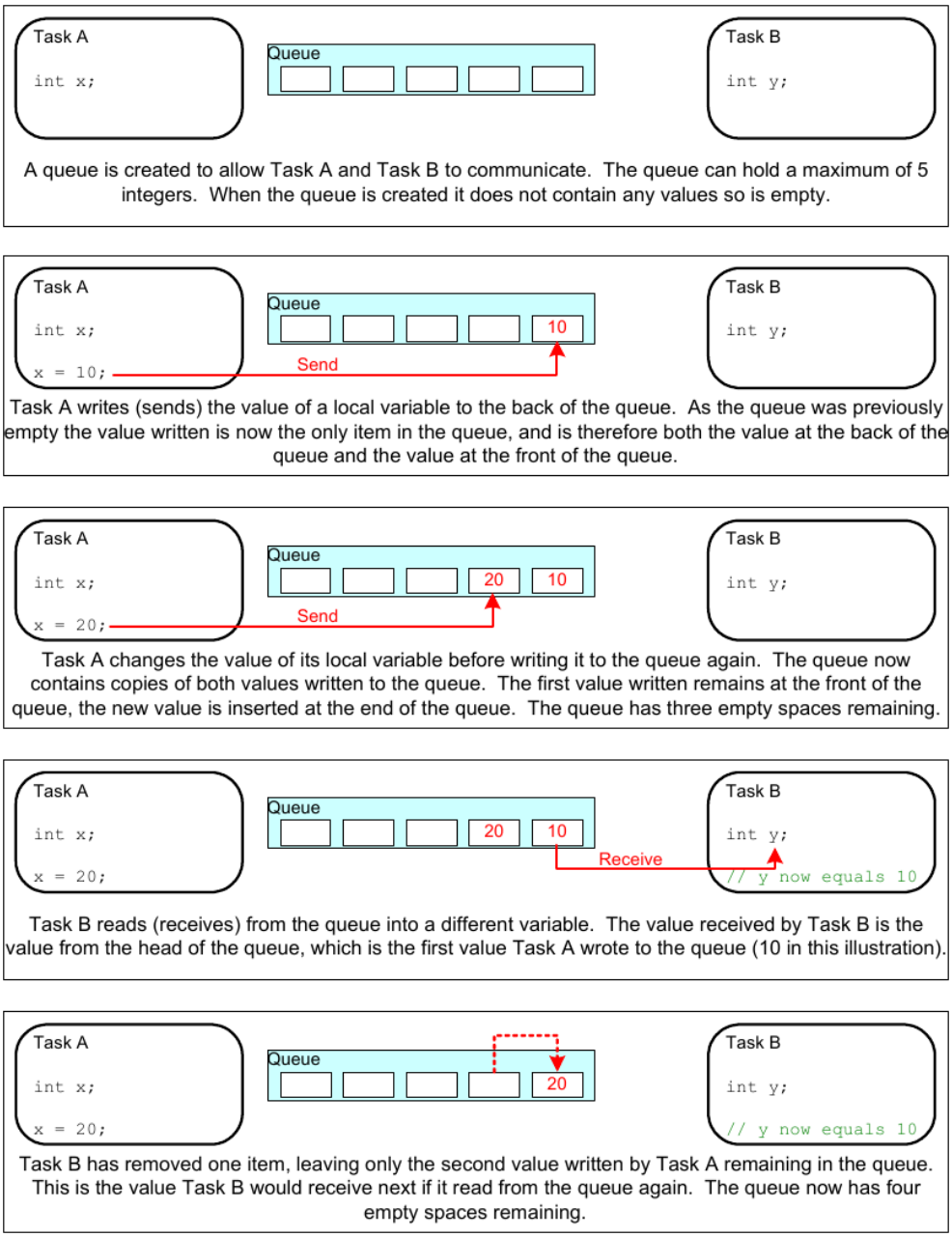


Figura 4.5: FreeRTOS Queue

El flujo de operación del *executer* o ejecutor se detalla en la figura 4.7 y es bastante simple en cuánto cumple sus tres obligaciones de manera secuencial. Esta simplicidad, sin embargo, no revela la verdadera utilidad de este módulo: proveer un ambiente de ejecución exclusivo al comando. Por diseño sólo un módulo ejecutor existe en el sistema, por lo tanto, una vez aquí el comando toma el control del procesador, de todos los recursos de hardware y software para realizar su tarea. Además aprovecha la generalidad definida por la interfaz de los comandos, para permitir a este módulo ejecutar cualquier función que implemente la interfaz requerida.

El código 4.13 detalla la implementación de este módulo. Al momento de crear esta tarea se debe tener en cuenta las siguientes consideraciones:

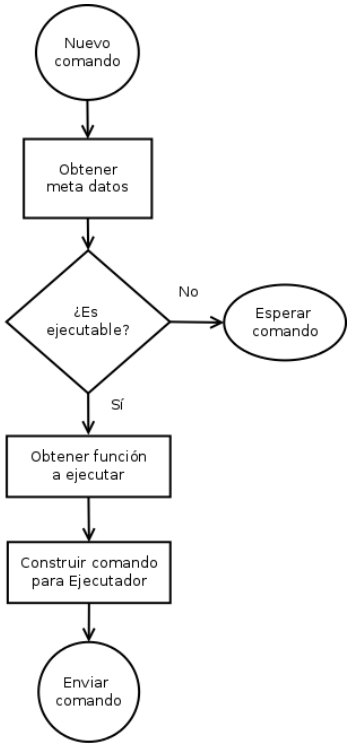


Figura 4.6: Diagrama de flujo para *dispatcher*

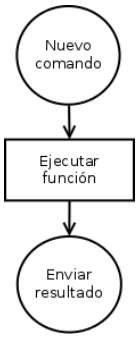


Figura 4.7: Diagrama de flujo para *executer*

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

Listing 4.13: taskExecuter.c

```c
#include "taskExecuter.h"

extern xQueueHandle executerCmdQueue; /* Comands queue*/
extern xQueueHandle executerStatQueue; /* Comands queue*/

void taskExecuter(void *param)
{
    printf(">>[Executer] Started\n");
    ExeCmd RunCmd;
    int cmdStat, queueStat, cmdParam;

    while(1)
    {
        /* Read the CMD that Dispatcher sent - BLOCKING */
        queueStat = xQueueReceive(executerCmdQueue,&RunCmd,portMAX_DELAY);
        if(queueStat == pdPASS)
        {
            printf("[Executer] Running a command...\n");
            ClrWdt();

            /* Execute the command */
            cmdParam = RunCmd.param;
            cmdStat = RunCmd.fnct((void *)&cmdParam);

            ClrWdt();
            printf("[Executer] Command result: %d\n", cmdStat);

            /* Send the result to Dispatcher - BLOCKING */
            xQueueSend(executerStatQueue, &cmdStat, portMAX_DELAY);
        }
    }
}
```

4.5.7. Listeners

Los *listeners* son los módulos encargados de implementar la lógica de generación de comandos dentro del sistema de vuelo. Existen una serie de *listeners* bajo la lógica de que a cada uno le corresponde controlar un determinado subsistema del satélite. Estos módulos se implementan como tareas de FreeRTOS que se activan de manera periódica, ejecutando operaciones de control que tienen como salida comandos que son agregados a la cola del *dispatcher*. La lógica de funcionamiento estas tareas se detalla en el diagrama de la figura 4.8. FreeRTOS ofrece dos funciones que implementan la periodicidad en las ejecución de las tareas, mediante la técnica de suspender su ejecución por una cantidad determinada de *ticks*: `vTaskDelay` y `vTaskDelayUntil`. Cada una de estas funciones cambia el estado de la tarea a suspendida, un estado dónde la tarea no utiliza ningún recurso del sistema, luego del tiempo específica la tarea retoma su ejecución. La diferencia entre ambas funciones es que la primera siempre suspende la tarea durante un tiempo fijo, mientras que la segunda mantiene fijo el tiempo que transcurre entre ambas llamadas[?]. `vTaskDelayUntil` es más adecuada para tareas de tipo *hard real-time* y es la utilizada en los *listeners* para proveer una resolución de tiempo más precisa.

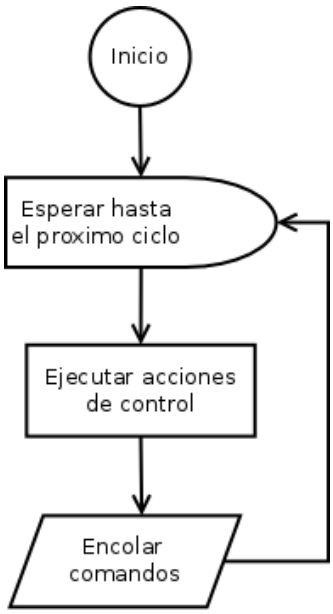


Figura 4.8: Diagrama de flujo para *listeners*

Los *listeners* y su implementación pueden ser bastantes específicos a cada sistema. La regla general es que se crea uno por cada subsistema que se debe controlar, pero la complejidad de cada implementación se esconde bajo la lógica programa como "operaciones de control" del diagrama 4.8. A continuación se desarrolla el ejemplo de la implementación de un *listener*.

Una de las funcionalidades principales que se puede esperar del sistema es la ejecución de acciones de manera periódica, acciones que incluyen tareas de control interno del propio subsistema al que corresponde el computador a bordo. Este módulo es denominado *housekeeping* y su lógica de control es: ejecutar comandos de control interno a periodos fijos de tiempo. Según las necesidades del sistema, se puede implementar varios periodos fijos, en este

caso ejecutarán comandos cada 1, 10, y 30 segundos. Para guiar la implementación de este módulo, su lógica se detalla en la figura 4.9

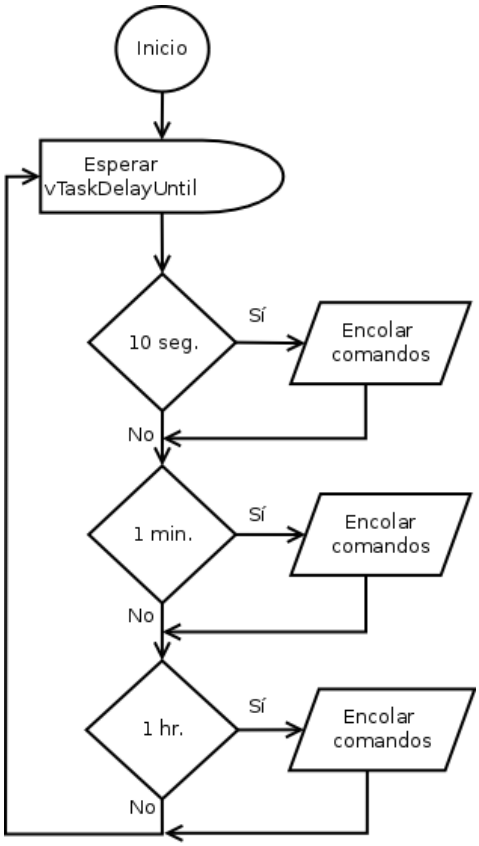


Figura 4.9: Diagrama de flujo para *housekeeping*

Los comandos que se ejecutan en cada intervalo de tiempo, dependerán de la aplicación, en este caso se ejecutarán una serie de comandos cuya función es actualizar las variables de estado del sistema. La implementación de esta tarea se detalla en el código 4.14.

Con la implementación del primer *listener* y la correcta inicialización del sistema operativo, las tareas y los repositorios, se tiene la primera versión funcional del sistema de vuelo. Si bien no se realiza ninguna tarea fundamental para lo que significa el proyecto satelital, la generalidad de la implementación responde principalmente a los requerimientos no operacionales del proyecto y es la base para cualquier proyecto derivado.

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

```
>>Starting FreeRTOS [->]
>>[Executer] Started
>>[Dispatcher] Started
>>[Houskeeping] Started
[Houskeeping] _10sec_check
[Dispatcher] Cmd: 1001, Param: 0, Orig: 1001
```

Listing 4.14: taskHouskeeping.c

```c
#include "taskHouskeeping.h"
extern xQueueHandle dispatcherQueue; /* Commands queue */

void taskHouskeeping(void *param)
{
    printf(">>[Houskeeping] Started\r\n");
    portTickType delay_ms    = 1000;    //Task period in [ms]
    portTickType delay_ticks = delay_ms / portTICK_RATE_MS; //Task period

    unsigned int elapsed_sec = 0;       // Seconds count
    unsigned int _10sec_check = 10;     //10[s] condition
    unsigned int _10min_check = 10*60;  //10[m] condition
    unsigned int _1hour_check = 60*60;  //1[h] condition

    DispCmd NewCmd;
    NewCmd.idOrig = CMD_IDORIG_THOUSEKEEPING; //Housekeeping

    portTickType xLastWakeTime = xTaskGetTickCount();
    while(1)
    {
        vTaskDelayUntil(&xLastWakeTime, delay_ticks); //Suspend task
        elapsed_sec += delay_ms/1000; //Update seconds counts

        /* 10 seconds actions */
        if((elapsed_sec % _10sec_check) == 0)
        {
            printf("[Houskeeping] _10sec_check\n");
            NewCmd.cmdId = obc_id_get_rtos_memory;
            NewCmd.param = 0;
            xQueueSend(dispatcherQueue, &NewCmd, portMAX_DELAY);
        }

        /* 10 minutes actions */
        if((elapsed_sec % _10min_check) == 0)
        {
            printf("[Houskeeping] _10min_check\n");
            NewCmd.cmdId = drp_id_print_CubesatVar;
            NewCmd.param = 0;
            xQueueSend(dispatcherQueue, &NewCmd, portMAX_DELAY);
        }

        /* 1 hours actions */
        if((elapsed_sec % _1hour_check) == 0)
        {
            printf("[Houskeeping] _1hour_check\n");
            NewCmd.cmdId = drp_id_update_dat_CubesatVar_hoursWithoutReset;
            NewCmd.param = 1; //Add 1 hour
            xQueueSend(dispatcherQueue, &NewCmd, portMAX_DELAY);
        }
    }
}
```

```
[Executer] Running a command...
Free RTOS memory: 2282
[Executer] Command result: 2282
[Houskeeping] _10sec_check
[Dispatcher] Cmd: 1001, Param: 0, Orig: 1001
[Executer] Running a command...
Free RTOS memory: 2282
[Executer] Command result: 2282
(...)
[Houskeeping] _10min_check
[Dispatcher] Cmd: 5002, Param: 0, Orig: 1001
[Executer] Running a command...
==================================
        Status repository
==================================
0, -1
1, -1
2, -1
3, -1
4, -1
5, -1
6, -1
7, -1
8, -1
9, -1
10, -1
(...)
[Executer] Command result: 1
(...)
[Houskeeping] _1hour_check
[Dispatcher] Cmd: 5000, Param: 1, Orig: 1001
[Executer] Running a command...
[Executer] Command result: 1
(...)
[Houskeeping] _1hour_check
[Dispatcher] Cmd: 5000, Param: 1, Orig: 1001
[Executer] Running a command...
[Executer] Command result: 1
(...)
[Houskeeping] _10min_check
[Dispatcher] Cmd: 5002, Param: 0, Orig: 1001
[Executer] Running a command...
==================================
        Status repository
==================================
0, -1
1, 2
```

```
2, 2
3, -1
4, -1
5, -1
6, -1
7, -1
8, -1
9, -1
10, -1
(...)
```

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

4.6. Específico al proyecto SUCHAI

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

4.6.1. Consola serial

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

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

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

La entrada de datos requiere mayor detalle en la implementación en cuanto debe permitir el reconocimiento de las cadenas de caracteres que se ingresan como órdenes al sistema. Para esto se implementa un *listener* con dedicación exclusiva tomar los caracteres de la consola serial, formar las cadenas de texto, separar parámetros e interpretar la orden como comandos del sistema de vuelo que serán encolados para su ejecución. El detalle de la implementación de esta tarea se ilustra a través del diagrama de flujo de la figura 4.10

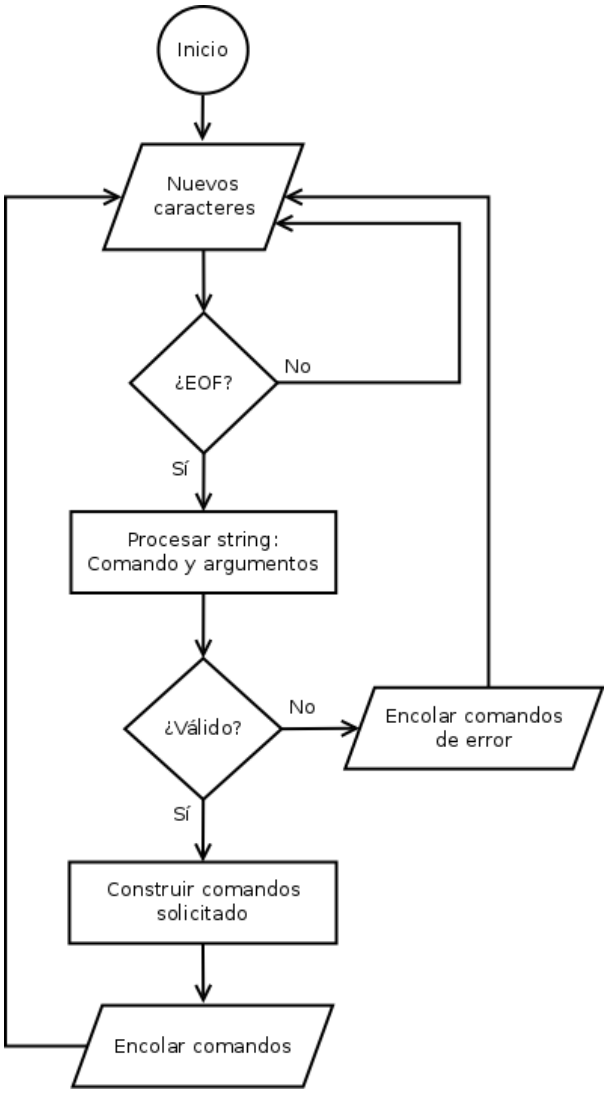


Figura 4.10: Diagrama de flujo para *taskConsole*

4.6.2. Plan de vuelo

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

La solución consiste en implementar un *listener* que monitorice la hora y fecha del sistema para obtener desde el plan de vuelo el comando adecuado a ser encolado para su ejecución.

El plan de vuelo en sí, consiste e una lista ordenada temporalmente con códigos de comandos y sus parámetros. La lista se implementa en una memoria permanente externa, como: un arreglo espaciado con cierta resolución de tiempo; un arreglo que incluya además la hora correspondiente al comando; o un archivo con una lista de comando, parámetro y fecha y hora de ejecución. El flujo de operación de la tarea corresponde al diagrama de la figura 4.11.

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

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

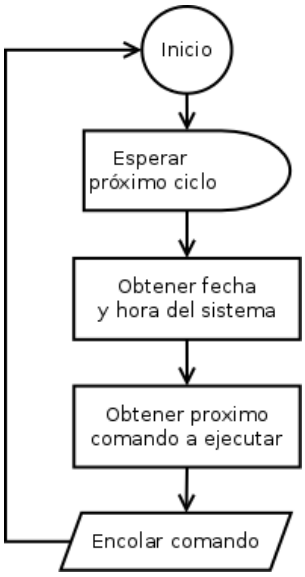


Figura 4.11: Diagrama de flujo para *taskFlightPlan*

4.6.3. Comunicaciones

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

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

- Configurar hardware de comunicaciones.
- Leer configuraciones de hardware de comunicaciones.
- Leer telecomandos recibidos por el subsistema de comunicaciones.

- Escribir telemetría para ser transmitida por el subsistema de comunicaciones.
- Configurar un *beacon* y transmitirlo.

Al igual que el caso de la consola serial, se requiere la implementación de un *listener* para controlar las operaciones de lectura y procesamiento de las ordenes enviadas de manera remota al satélite. El estado de funcionamiento del sistema de comunicaciones está disponible como variables de estado a través del repositorio de comandos, por lo tanto, las operaciones del *listener* de comunicaciones incluyen monitorizar estas variables de estado para realizar las operaciones correspondientes. Estas incluyen monitorizar la llegada de nuevos *frames* de telecomandos para comandar su lectura; procesar los telecomandos leídos generando los comandos del sistema solicitados; y generar a intervalos fijos el *beacon* del satélite con la información adecuada. La implementación de este *listener* sigue responde al diagrama de flujo de la figura 4.12

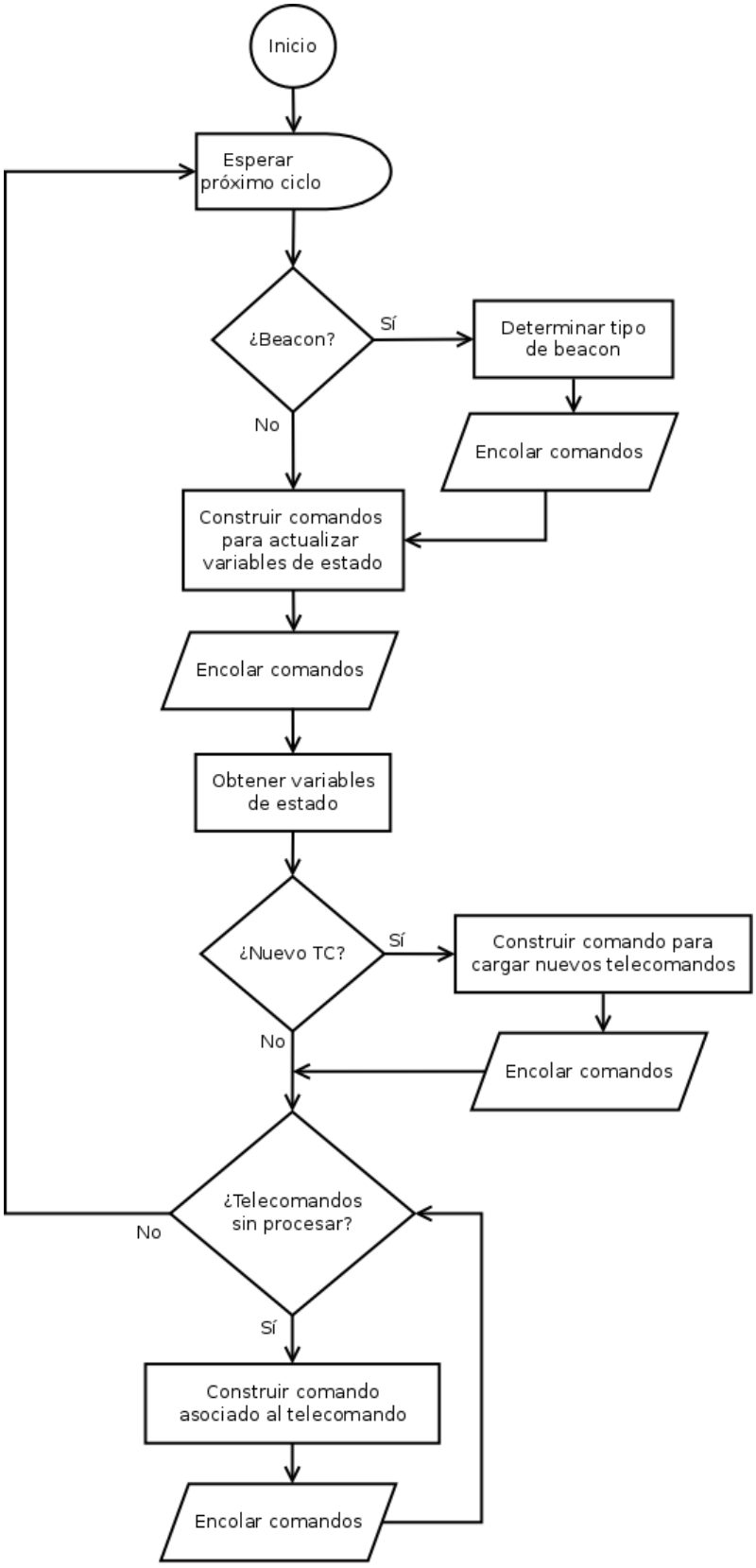


Figura 4.12: Diagrama de flujo para *taskCommunications*

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