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

DISEÑO E IMPLEMENTACIÓN DEL SOFTWARE DE CONTROL PARA EL COMPUTADOR A BORDO DE UN PICO-SATÉLITE

TESIS PARA OPTAR AL TÍTULO DE TÍTULO DE INGENIERO CIVIL ELECTRICISTA

CARLOS EDUARDO GONZÁLEZ CORTÉS

PROFESOR GUÍA:
MARCOS DÍAZ QUEZADA

MIEMBROS DE LA COMISIÓN:

SANTIAGO DE CHILE
MARZO 2013

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1. Fundamentación y objetivos generales

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

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

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

1.2. Objetivos específicos

Los objetivos específicos del proyecto se enumeran a continuación

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

El listado de objetivos presenta un orden temporal en la ejecución de estas tareas puesto que objetivo es dependiente del cumplimiento de los objetivos anteriores. El trabajo se puede considerar terminado cuando se ha probado la implementación e integración del *software* con los módulos de comunicaciones y energía obteniendo un sistema satelital con las mínimas funcionalidades.

Capítulo 2

Contextualización

2.1. Marco Teórico

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

2.1.1. Sistemas embebidos

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

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

- Un sistema embebido se mantiene siempre funcionando y debe proveer respuesta en tiempo real. Se debe diseñar considerando una operación continua y una posible reconfiguración del sistema estando ya en marcha.
- Las interacciones con el sistema pueden ser impredecibles y no se tiene control so-

bre ellas. Existen sistemas que son controlados por el usuario mediante una interfaz preparada para ellos, mientras que otros sistemas deben atender eventos imprevistos sin dejar de realizar tareas rutinarias.

- Existen limitaciones físicas. Normalmente estos sistemas poseen limitadas características de: poder de cómputo, memoria de datos y de programa; espacio físico; y disponibilidad de energía.
- El diseño de software para sistemas embebidos requiere una interacción de bajo nivel. Existe una amplia gama de plataformas de hardware para desarrollar sistemas embebidos y se requiere interactuar también con una amplia gama de dispositivos externos. Por esto se requiere desarrollar capas de drivers de periféricos que oculten las diferencias de hardware a la aplicación final del sistema.
- Es importante considerar aspectos de seguridad y confiabilidad del sistema durante todo su desarrollo debido a que la mayoría de los sistemas embebidos son usados para controlar otros sistemas críticos en diversos procesos.

Microcontroladores PIC

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

Tabla 2.1: Guía de microcontroladores PIC

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

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

Arquitectura Poseen un juego de instrucciones *Reduced Instruction Set Computing* (RISC) (80 instrucciones) de ancho fijo en 24 bits que en su mayoría se ejecutan en un solo ciclo excepto: divisiones; cambios de contexto; y acceso por tabla a memoria de programa[**?**]. Se basa en una arquitectura Harvard modificada de 16 bits de datos[**?**] lo que significa que el dispositivo posee una memoria de datos tipo *Random Access Memory* (RAM) separada de la memoria de datos (FLASH) pudiendo acceder de manera independiente e incluso simultáneamente a las instrucciones del programa y a los datos de este alojados en RAM. La arquitectura de la CPU la completan una *Arithmetic Logic Unit* (ALU) con *hardware* dedicado para realizar multiplicaciones y divisiones. El detalle de la arquitectura del microcontrolador PIC24 se detalla en la figura 2.1. También posee un vector de hasta 128 interrupciones con capacidad para atender hasta 8 de ellas lo que permite liberar al procesador de la espera de sucesos asíncronos ya que son notificados y atendidos de manera específica en una rutina de atención de la interrupción.

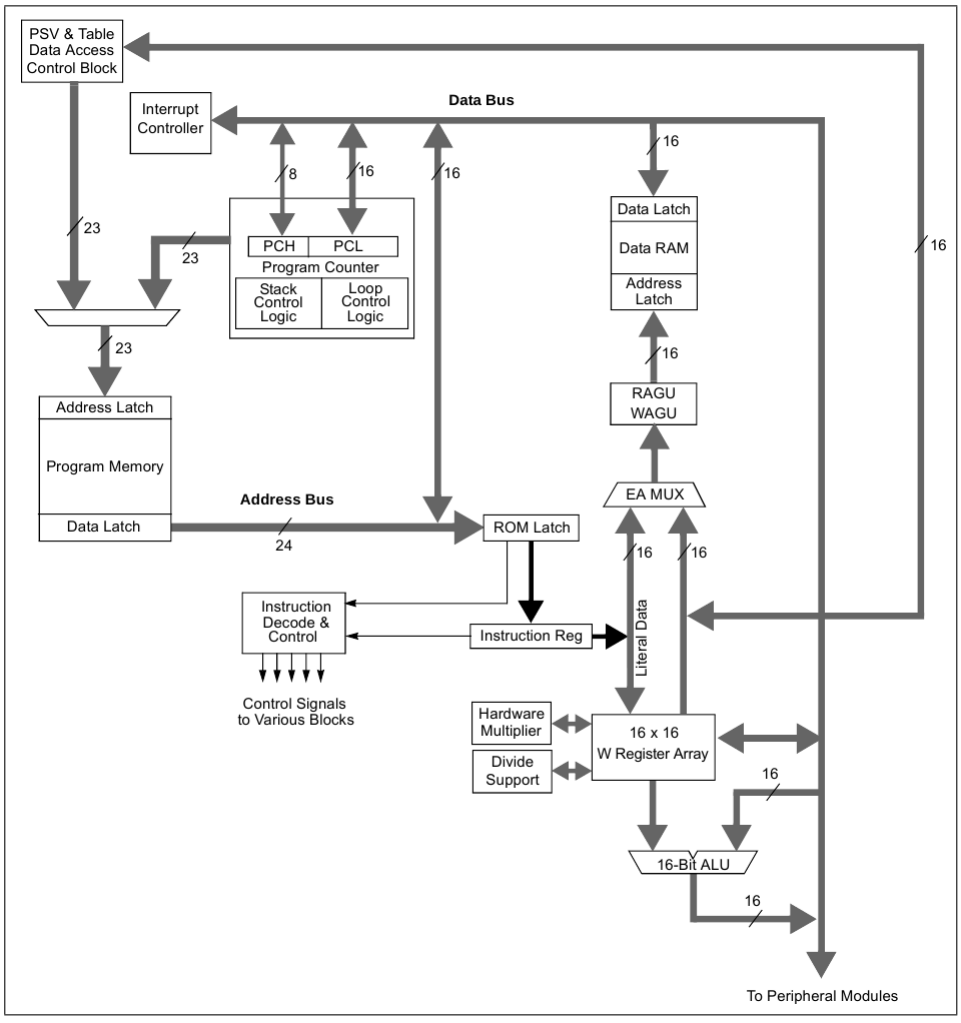

Figura 2.1: Arquitectura de la CPU del PIC24F

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

en aplicaciones que requieran: conversores análogos a digitales; temporizadores; comunicación síncronas y asíncronas como RS232, SPI o I2C; USB o Ethernet; manteniendo acotados los costos del sistema. Una lista de los periféricos disponibles para estos microcontroladores se detalla en la figura 2.2. El acceso para configurar, guardar y obtener datos de estos periféricos se realiza a través de registros que están mapeados en la memoria de datos del microcontrolador por lo tanto comparten el bus de datos y no son necesarias instrucciones extras para su integración. Junto con una completa documentación de la arquitectura y funcionamiento de cada periférico, los compiladores de lenguaje C de Microchip proveen librerías para acceder a estas funcionalidades a través de una API de más alto nivel.

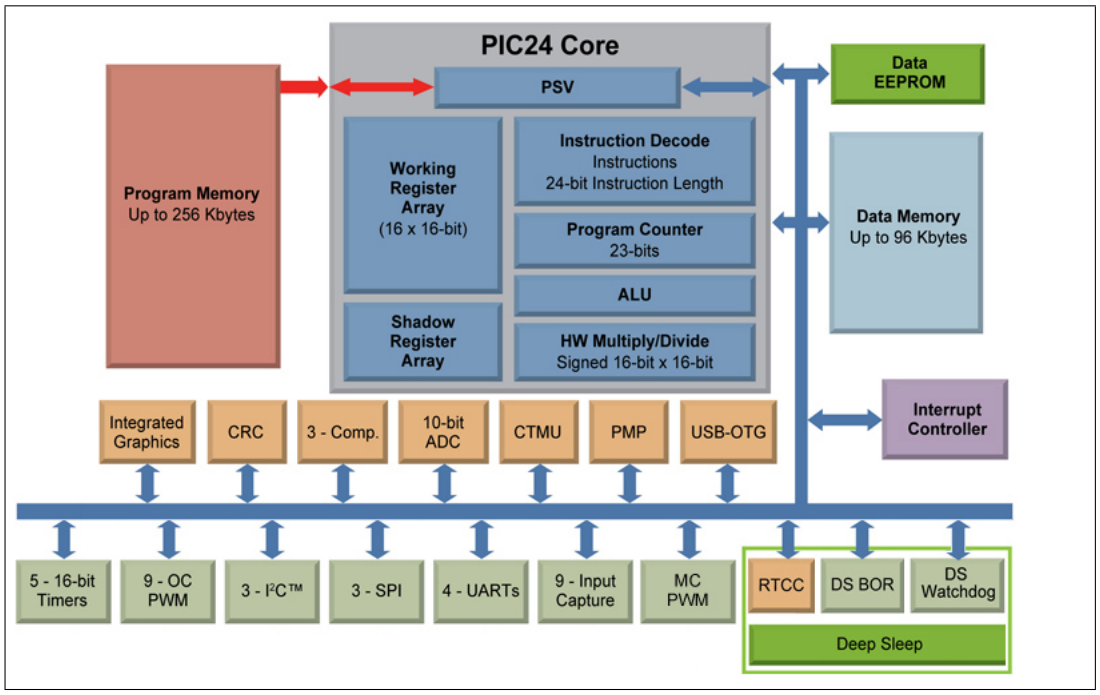

Figura 2.2: Arquitectura del PIC24F

Desarrollo

2.1.2. Sistemas Operativos

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

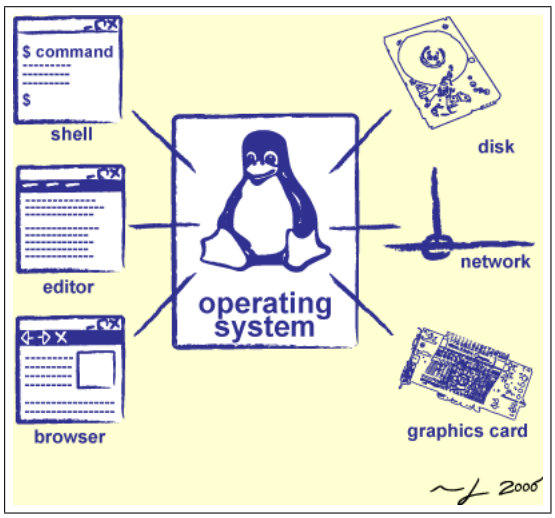

Figura 2.3: Sistema Operativo como capa de abstracción

Sistemas Operativos de Tiempo Real

Parte fundamental de un sistema operativo es su *scheduler*, módulo encargado de intercambiar entre las múltiples tareas que se deben ejecutar cediendo un espacio de tiempo para utilizar le procesador. La forma en que trabaja el *scheduler* define el tipo de sistema operativo que se posee. Un sistema operativo de tiempo real posee un *scheduler* diseñado para proveer un flujo de ejecución determinista, pues solo sabiendo con exactitud la tarea que el sistema ejecutara en un determinado momento se pueden cumplir los requerimientos estrictos de *timing*[?]. Esto es un aspecto de especial interés en sistemas embebidos que normalmente requieren respuesta en tiempo real ante eventos no predecibles.

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

FreeRTOS

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

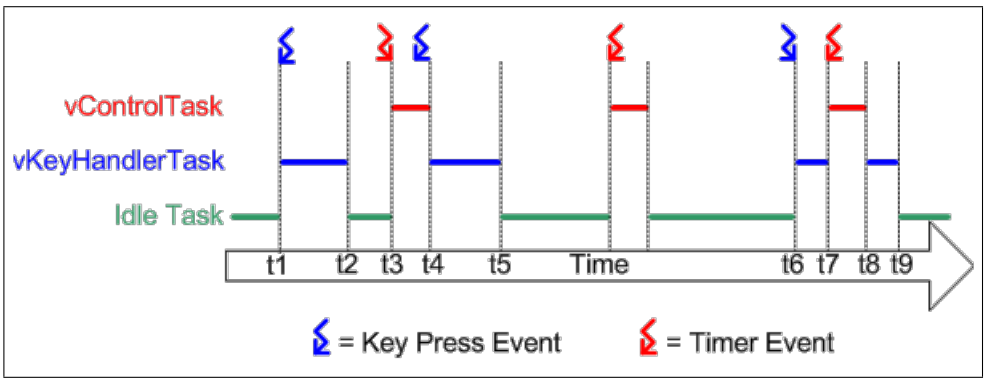

Figura 2.4: *Real time scheduling*

el como controladores para el hardware o periférico. Entre sus principales características se encuentran:

- *Scheduler pre-emptive* o cooperativo.
- Sincronización y comunicación entre tareas a través de colas, semáforos, semáforos binarios y mutexes.
- Mutexes con herencia de prioridades.
- Software *timers*.
- Bajo consumo de memoria (Entre 6K y 10K en ROM).
- Altamente configurable.
- Detección de *stack overflow*
- Soporte oficial a 33 arquitecturas de sistemas embebidos.
- Estructura de código portable, escrito en C.
- Licenciado bajo *General Public License* (GPL) modificada que permite su uso comercial sin publicar código fuente.
- Gratuito
- Amplia documentación, foros y asistencia técnica.

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

La creación de taras y el control de sus estados se realiza a través de la API de FreeRTOS que documenta claramente todas las posibles operaciones que se pueden realizar sobre una

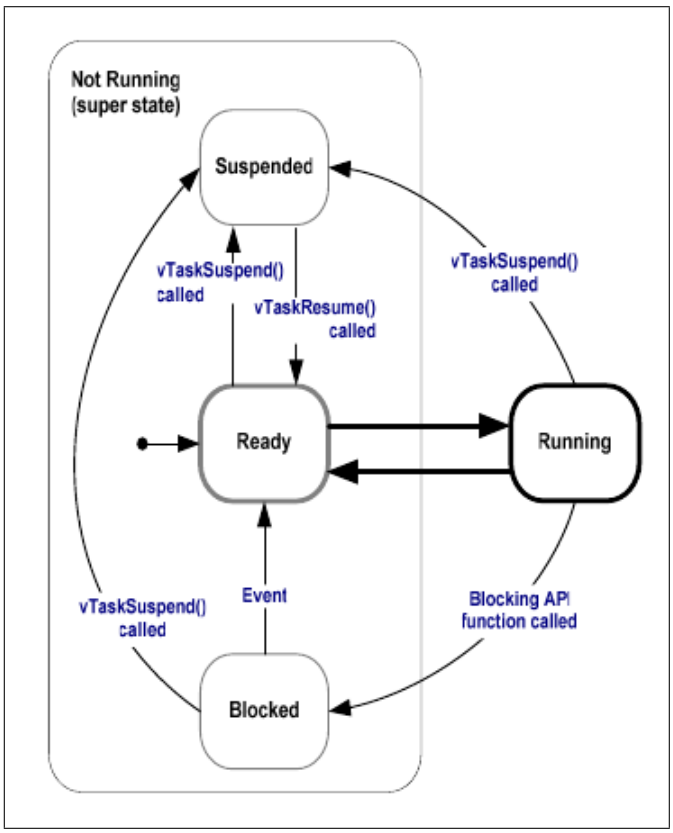

Figura 2.5: Tareas de FreeRTOS, diagrama de estados

tarea.

El *scheduler* es la parte fundamental del kernel que controla la ejecución de las diferentes tareas disponibles. Su objetivo es generar la sensación de estar en un ambiente multitarea, cuando en realidad sólo una tarea puede ejecutarse a la vez ya que se posee un solo procesador. Como se detalla en la figura 2.6 la función del *scheduler* es entregar una porción de tiempo de ejecución fijo a una tarea y una vez que este tiempo se cumple se debe guardar el estado de ejecución de esta tarea y se procede a ejecutar otra. Así cada una de las tareas se ejecuta durante una pequeña porción de tiempo, hasta que completa su trabajo; si el tiempo de proceso asignado a cada tarea es lo suficientemente pequeño pareciera que muchas cosas ocurrieron simultáneamente. Claramente una sola tarea tomaría, en términos absolutos, menos tiempo en terminar si no fuera interrumpida, pero se gana un sistema más fluido cuando se deben ejecutar en conjunto tareas que toman mucho tiempo de proceso y otras relativamente cortas.

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

Figura 2.6: *Scheduling* de tareas

2.1.3. Ingeniería de Software

Licencias de Software

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

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

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

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

GPL

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

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

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

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

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

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

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

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

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

This file is part of Foobar.

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

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

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

Calidad de software

Los requerimientos no funcionales de un proyecto de software puede definirse como los parámetros de calidad buscados en producto en sí. Como parámetros de calidad se pueden entender una serie de calificativos muy subjetivos y tal vez difícilmente medibles como: rapidez, seguridad, escalabilidad o modularidad. Algunos conceptos pueden ser utilizados de manera poco clara o equivalentes como lo extensible o escalable que puede ser un software, sin dejar claro las diferencias o acotaciones de estos conceptos. Es por esto que se han creado normas en torno a las metodologías para desarrollar y utilizar modelos de calidad de software que permitan establecer de manera clara los parámetros a medir.

En especial se tratará la norma ISO/IEC 25010, una actualización a la antigua norma ISO/IEC 9126, que plantea un modelo de calidad software que consta de ocho características con sus respectivos sub-atributos, y puede ser utilizado para evaluación o especificación de software durante las etapas de: identificación de requerimientos; validación de la integralidad de la lista de requerimientos; identificación de los objetivos del diseño del software; identificación de los objetivos de las pruebas; identificación de los criterios de calidad; o la definición de criterios para determinar si un producto de software está completo [1].

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.

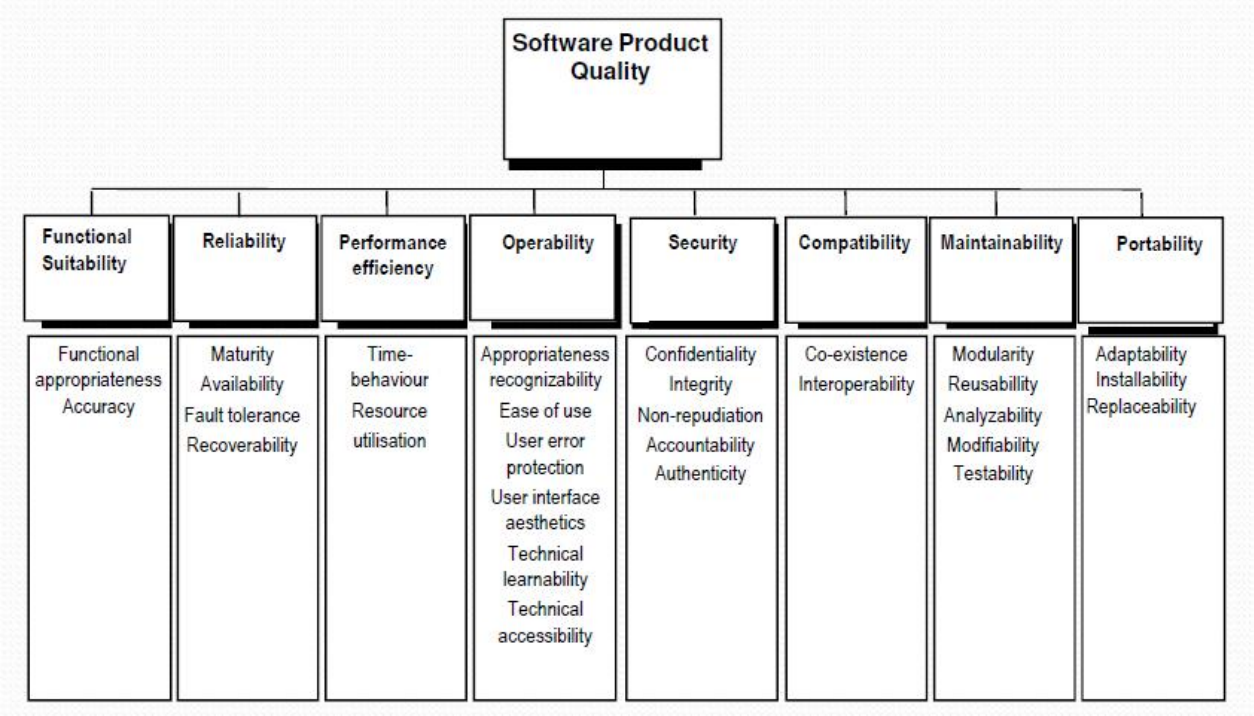

Figura 2.7: ISO/IEC 25010, categorías y subcategorías.

Patrones de diseño

Command Pattern

2.1.4. Satélites

En general sobre lo qué es una satélite y tópicos transversales a todo proyecto aeroespacial.

Pico-satélites

Satélites tipo Cubesat

Estándar

Aplicaciones

2.2. Antecedentes generales

Carrera satelital chilena

2.3. Antecedentes específicos

SUCHAI proposal

Investigaciones asociadas

Lagmuir Probe

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 funcionales de la misión así, requerimientos generales de desarrollo, así como 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 en base a módulo y se centra en específico en la aplicación de un determinado patrón de diseño que guía la solución propuesta en forma de una arquitectura basada en *command patern*

Para el diseño de la arquitectura se ha utilizado un patrón de diseño llamado *command pattern* el cual se programa en lenguaje C para el compilador Microchip MPLAB C30 V3.3x utilizando el sistema operativo FreeRTOS. Todo esto sobre una plataforma de desarrollo CubesatKit que cuenta con un procesador PIC24FJ256GA110.

3.2. Requerimientos

3.2.1. Requerimientos generales

3.2.2. 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.

Tolerancia a fallos de software. SURE? El sistema debe ser capaz de recuperarse ante fallas del software de control, reinicios o módulos de software funcionando de manera incorrecta. Por ejemplo, debe evitar la ejecución de acciones que han causado fallas sucesivas veces o no activar módulos que han fallado anteriormente.

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

Estado de salud del sistema.

Mucho tiempo sin conexión a tierra.

Problemas al desplegar la antena.

Watchdog.

Fallos de hardware.

3.2.3. Requerimientos mínimos

3.2.4. 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[1].

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

3.3. Plataforma

Se describe la plataforma final donde se ejecutará la aplicación, en este caso un sistema embebido consistente de un PIC24FJ256GA110

3.4. Arquitectura de software

3.4.1. Arquitectura Global

3.4.2. Controladores de hardware

3.4.3. Sistema operativo

3.4.4. Aplicación

Capítulo 4

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.

4.1. Pruebas modulares

4.1.1. Pruebas a Console

4.1.2. Pruebas a Hauskeeping

4.1.3. Pruebas a Dispatcher

4.1.4. Pruebas a Executer

4.2. Pruebas de arquitectura

4.3. Pruebas de integración básica

Capítulo 5

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.

5.1. Conclusiones generales

5.2. Conclusiones específicas

5.3. Trabajo futuro

Bibliografía

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