Universidad Central “Marta Abreu” de Las Villas

Facultad de Ingeniería Eléctrica

Departamento de Automática y Sistemas Computacionales

TRABAJO DE DIPLOMA

“Software de comunicación de bajo nivel para aplicación multisensorial”

Autor: Zedhi Linares Santana

Tutor: MSc. Alain Sebastián Martínez Laguardia

Santa Clara

2007

"Año 49 de la Revolución"

Hago constar que el presente trabajo de diploma fue realizado en la Universidad Central “Marta Abreu” de Las Villas como parte de la culminación de estudios de la especialidad de Ingeniería en Automática, autorizando a que el mismo sea utilizado por la Institución, para los fines que estime conveniente, tanto de forma parcial como total y que además, no podrá ser presentado en eventos, ni publicados sin autorización de la Universidad.

_____

Firma del Autor

Los abajo firmantes certificamos que el presente trabajo ha sido realizado según acuerdo de la dirección de nuestro centro y el mismo cumple con los requisitos que debe tener un trabajo de esta envergadura referido a la temática señalada.

_____

Firma del Autor

_____

Firma del Jefe de Departamento donde se defiende el trabajo

_____

Firma del Responsable de Información Científico-Técnica

PENSAMIENTO

Emplearse en lo estéril cuando se puede hacer lo útil; ocuparse en lo fácil cuando se tienen bríos para intentar lo difícil, es despojar de su dignidad al talento.

José Martí

DEDICATORIA

A mis padres…

A mi hermano…

A mis amigos…

AGRADECIMIENTOS

Es muy difícil incluir aquí a toda la gente que de una u otra forma me ha a apoyado durante este periodo, pero quiero agradecer de forma muy especial a mis padres, que siempre me apoyaron para seguir adelante, a mi tutor Alain Martínez por servirme de guía en todo momento, a mis compañeros de aula por todos sus consejos, apoyo y excelentes momentos que hemos compartido, ya que junto a ellos aprendí muchísimas cosas. También me gustaría agradecer profundamente mi profesora Ileana y Sergio ayudarme tanto en el transcurso de la carrera y por tener tanta paciencia. Al domino por hacerme perder el tiempo. A todos gracias.

TAREA TÉCNICA

1. Análisis de los elementos significativos en la arquitectura de software de una aplicación multisensorial, con ejemplo particular en el diseño de un vehículo autónomo.

2. Determinación de los factores comunes e imprescindibles en la comunicación de los distintos elementos que componen la arquitectura de hardware de la aplicación.

3. Desarrollo del software de bajo nivel de comunicación entre los sensores y la PC que componen la arquitectura de hardware básica de un vehículo autónomo.

Firma del Autor	Firma del Tutor

RESUMEN

Los vehículos autónomos “AV” (“Autonomous Vehicle”) juegan un papel importante en el desarrollo tenológico, lo que ha suscitado la atención de muchos investigadores. Dentro de ellos reviste particular importancia el dessarrollo de un vehículo autónomo aéreo (UAV0 pues los mismos presentan una gran cantidad de aplicaciones fundamentalmente en la toma de imágenes de alta precisión georeferenciadas que pueden ser empleadas para toma de decisiones, así como para el control de reservas naturales y fotografía cartográfica entre otras posibles aplicaciones.

En este trabajo, se presenta la arquitectura básica de Hardware para un vehículo autónomo aéreo, así como es planteado un análisis de acda uno de los elementos que la componen. Igualmente son presentadas las consideraciones para la arquitectura básica de software que combina la fiabilidad con la robustez dado el alto nivel de exigencias a que va a estar sometido el sistema.

Finalmente, se presenta el diseño del software de bajo nivel para la comunicación serie de los sistemas de sensores con la unidad central y que fue confeccionada en lenguaje C y empleando código de redundancia cíclica para garantizar la tolerancia a fallos.

TABLA DE CONTENIDOS

INTRODUCCIÓN

Los vehículos autónomos “AV” (“Autonomous Vehicle”) juegan un papel importante en el desarrollo tecnológico, lo que ha suscitado la atención de muchos investigadores. Los AV se han creado con el objetivo de facilitar al ser humano un conjunto de tareas difíciles de llevar a cabo, ya sean monótonas o aquellas que conllevan un riesgo humano. Se han desarrollado muchos tipos de AV, desde los caseros que se utilizan para experimentar nuevas sensaciones y juegos, hasta los más complejos que se emplean para labores de investigación. Sin embargo, todos los proyectos tienen tres pilares comunes que son imprescindibles para dotar de autonomía a un vehículo: un sistema motriz adecuado, capacidad de análisis del medio y posibilidad de calcular una reacción consecuente, es decir, cálculo del movimiento.

El presente trabajo parte de la experiencia acumulada en los últimos años por el grupo GIMAS en colaboración con las universidades: Politécnica de Madrid y Libre de Bruselas, en el desarrollo de un vehículo autónomo aéreo (UAV). La aplicación fundamental futura de este UAV sería la toma de imágenes de alta precisión geo-referenciadas que puede ser empleada para toma de decisiones, así como para el control de reservas naturales y fotografía cartográfica.

Una vez terminada la aplicación, la misma puede ser generalizada a otras aplicaciones incluidas algunas de carácter militar, como la seguridad y la exploración.

Como parte de este proceso se logró obtener la arquitectura básica de hardware necesaria para la aplicación, así como ejemplos claros de cada uno de los elementos que la compondrían cumpliendo con los requerimientos de peso y

volumen imprescindibles para la misma, quedando por definir una arquitectura de software capaz de unificar el sistema con fiabilidad y robustez dado el alto nivel de exigencias a que va a estar sometida la aplicación.

El objetivo de este trabajo es analizar los distintos elementos que debe de tener en cuenta la arquitectura del software que será empleada en la aplicación para diseñar el software de comunicación que unirá los distintos sensores con la PC, cumpliendo con los requerimientos de tiempo, prioridades y manejo de fallos imprescindibles en una aplicación de tiempo real.

Además se analizarán los distintos componentes de hardware que componen la aplicación en función de los métodos de comunicación que emplean, sus frecuencias de muestreo y su nivel de significación en la estructura del conjunto para posteriormente, con el sistema de control, adecuar la arquitectura de software en función de los objetivos y restricciones descubiertos.

La estructura del trabajo es la siguiente: resumen, introducción, capítulos, conclusiones, recomendaciones, bibliografía y anexos.

En el Capítulo 1 se realiza un análisis de los elementos imprescindibles a considerar, en el diseño de una arquitectura de software, encaminada a operar en tiempo real en una aplicación multisensorial colocando, como caso de uso, la implementación de un vehículo autónomo.

En el Capítulo 2 se hace el análisis de la arquitectura de hardware, sobre el que va a ser desarrollado el software de comunicación de bajo nivel, poniendo énfasis en los distintos elementos que componen dicha arquitectura y sus interfases de comunicación.

En el Capítulo 3 se muestra el diseño del software de bajo nivel que unirá la PC con los sensores y que garantizará el proceso de comunicación.

CAPÍTULO 1. Arquitectura de Software

En este capítulo se analizan los factores que debe tener en cuenta la arquitectura de software que englobará la aplicación aéreo-autónoma. Además, se realiza un estudio de la definición de la arquitectura de software así, como un análisis detallado de los tipos de software a emplear y los pasos a seguir para el desarrollo de esta.

1.1 Qué es la Arquitectura de Software

Existen muchas definiciones de Arquitectura del Software y no parece que ninguna de ellas haya sido totalmente aceptada. En un sentido amplio se podría estar de acuerdo en que la Arquitectura del Software (Fielding 2000) es el diseño de más alto nivel de la estructura de un sistema, programa o aplicación y tiene la responsabilidad de:

1. Definir los módulos principales.
2. Definir las responsabilidades que tendrá cada uno de estos módulos.
3. Definir la interacción que existirá entre dichos módulos.
4. Control y flujo de datos.
5. Secuencia de la información.
6. Protocolos de interacción y de comunicación.
7. Ubicación en el hardware.

La definición oficial de Arquitectura del Software, según la IEEE Std 1471-2000 (Hilliard, 2000) es la siguiente: “La Arquitectura del Software es la organización

fundamental de un sistema formado por sus componentes, las relaciones entre ellos y el contexto en el que se implantarán y los principios que orientan su diseño y evolución".

1.2 Tipos de software

Las operaciones que debe realizar el hardware son especificadas por una lista de instrucciones, llamadas programas o software.

EL software se divide en dos grandes grupos:

1. Software del sistema.
2. Software de aplicaciones.

El software del sistema (JACOBSON, 1999) es el conjunto de programas indispensables para que la máquina funcione y se denominan también programas del sistema.(Jacobson,1999) Estos programas son básicamente:

1. Sistema operativo: DOS, WINDOWS y LINUX.
2. Editores de texto: EDIT, PADWORD.
3. Compiladores/intérpretes (lenguajes de programación).
4. Programas de utilitarios.

Uno de los programas más importantes es el sistema operativo que sirve, esencialmente, para facilitar la escritura y uso de sus propios programas.

El sistema operativo según Tanenbaum y Woodhull (2003): dirige las operaciones globales de la computadora, instruye a la computadora para ejecutar otros programas y controla el almacenamiento y recuperación de archivos (programas y datos) de cintas y discos.

Gracias al sistema operativo es posible que el programador pueda introducir y grabar nuevos programas, así como instruir a la computadora para que los ejecute. Los sistemas operativos pueden ser:

1. Monousuarios (un solo usuario) y multiusuarios o de tiempo compartido (diferentes usuarios), atendiendo al número de usuarios.

2. Monotarea (una sola tarea) o multitarea (múltiples tareas) según las tareas (procesos) que puede realizar simultáneamente.

Se debe diferenciar entre el acto de crear un programa y la acción de la computadora cuando ejecuta las instrucciones del programa.

La creación de un programa es realizada por un ser humano en base a ideas y objetivos, siguiendo una determinada metodología, que generara algún tipo de código, a continuación el mismo pasa a un compilador que lo convierte a un lenguaje entendible por la computadora.(Benitez, 2003)

La figura 1.1 muestra el proceso general de ejecución de un programa:

1. Aplicación de una entrada (datos) al programa.
2. Obtención de una salida (resultados).

La entrada puede tener una variedad de formas, tales como números o caracteres alfabéticos.

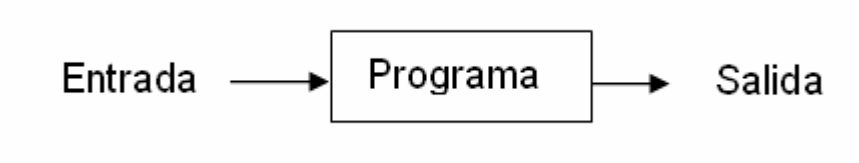


Figura 1.1: Acción de un programador

La salida puede también tener las siguientes formas:

1. Datos numéricos o caracteres.
2. Señales de control.

La ejecución de un programa según se muestra en la figura 1.2, requiere generalmente de unos datos como entrada, además del propio programa, para poder producir una salida.

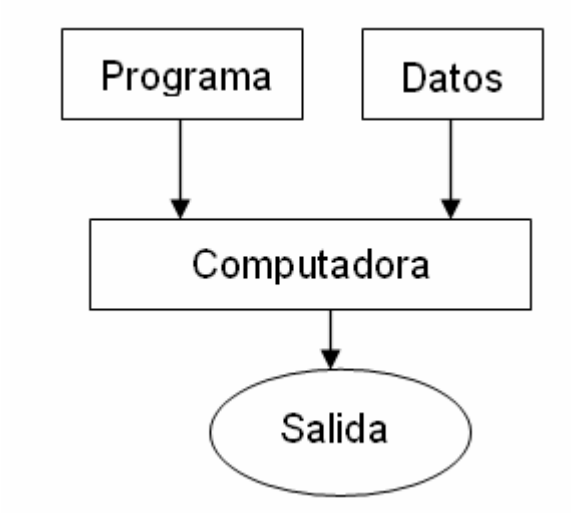


Figura 1.2: Ejecución de un programa.

1.3 Pasos para el desarrollo para la arquitectura de software

El uso de una arquitectura de software (HILLIARD, 2000) permite desarrollar aplicaciones de alta calidad de forma acelerada ayudando a los desarrolladores de software a tener una visión de aplicación, en lugar de subconjuntos aislados de código.

Los pasos más comunes del desarrollo de un software fueron planteados por Eric (2006) y son los mostrados en la figura 1.3. En el primer paso de esta estructura se produce una arquitectura de alto nivel donde se describen los propósitos y funciones de los componentes de la arquitectura. En un segundo paso, se crea un diseño en base a las especificaciones solicitadas. En esta fase también deben ser creados los modelos de los objetos, sus interfases y serán definidos los diagramas de secuencia para luego pasar a la fase de implementación donde se afinan los detalles, la relación con los sistemas físicos y las tecnologías seleccionadas.

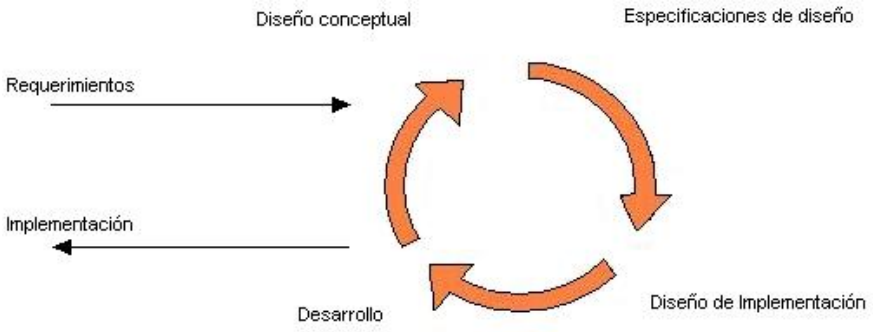


Figura 1.3: Pasos para el desarrollo de software.

Como las necesidades pueden cambiar en la fase de desarrollo o posteriormente por ampliación o actualizaciones se puede decir que el ciclo de vida del software es como el que se muestra en la figura 1.4.

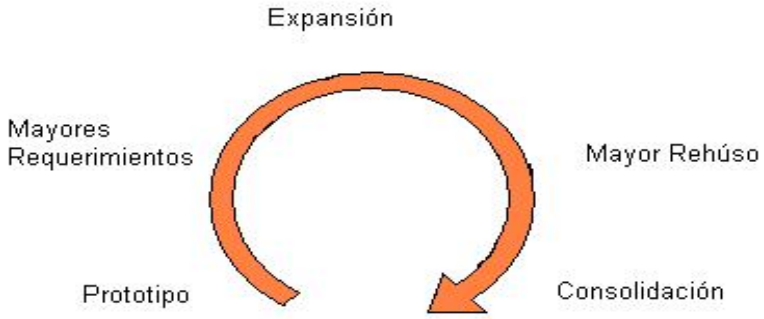


Figura 1.3: Fases del ciclo de vida del software.

El diseño de la arquitectura y los requerimientos deben llevar a una apropiada identificación de la tecnología a emplear. Siempre que sea posible la selección de determinada tecnología o conjunto de herramientas, no debe limitar la arquitectura, por esto deben ser tomados en cuenta algunos factores significativos del software tales como:

1. Sistema operativo.
2. Tecnología de redes.
3. Modelo de comunicación.
4. Lenguajes de Programación y Modelación.

5. Portabilidad.
6. Modularidad.
7. Integración a sistemas existentes.

1.3.1 Sistema Operativo

La selección del sistema operativo está relacionada directamente con la aplicación, es posible que los módulos de bajo nivel requieran de tiempo real (RT) ya que demoras en el proceso de comunicación pueden crear inestabilidad en el lazo de control. Los sistemas de Tiempo Real se clasifican en duros (Hard Real Time), que son aquellos que dependen de eventos o tienen requerimientos de tiempo inviolables y suaves (soft real time), que no sufren grandes afectaciones si los tiempos de operación son incumplidos. (Forssell, 1991) En los primeros, los tiempos de control están en el orden de 1ms a 10ms, mientras que para los segundos, varía de 50 a 500 ms.

La implementación de un vehículo autónomo exige, por una parte, un sistema de control en lazo cerrado, que hace necesario poder garantizar que la acción de control se dé en períodos de tiempo constantes. Se necesita además, una actualización periódica de ciertas variables de estado, por lo que la información debe ser llevada a la unidad de cómputo para su procesamiento e inserción en el sistema de control de forma rigurosa.

De los múltiples sistemas operativos para tiempo real, que pueden ser empleados en la aplicación que se trata, se decidió emplear el Linux con sus extensiones para tiempo real RT-Linux, tomando en consideración la solidez del mismo y la gran experiencia en su uso.

Las razones que argumentan el empleo de Linux y RT-Linux en esta aplicación son:

1. El sistema operativo aprovecha al máximo el hardware de la PC y es muy robusto.
2. El código fuente es público y por tanto fácilmente modificable y ampliable.
3. El coste es muy reducido y nulo en la fase de desarrollo.
4. Dispone de una potente extensión de tiempo real de coste nulo, con unas prestaciones equiparables a sistemas comerciales equivalentes.
5. Existe gran documentación al respecto.

El funcionamiento de RTLinux (Fagg et al., 1993, Rubini and Corbet, 2001) consiste en un pequeño parche que se aplica al kernel de Linux. El objetivo del mismo es permitir al sistema poder utilizar los servicios mas allá del planificador de tareas tradicional, permitiendo el desarrollo de funciones en tiempo real en un entorno predecible y con el menor retraso posible.

RT-Linux se apoya en Linux para arrancar, trabajar en red, manejar el sistema de ficheros y para cargar los módulos de tiempo real haciendo el sistema extensible y fácilmente modificable (Muñoz, 2003).

Como se aprecia en la figura 1.5, la solución de RT-Linux (Ripoll and Acosta, 2000) es un modo de ejecución simple de tareas con un kernel que funciona en tiempo real, actuando sobre otro kernel Linux, que no funciona en tiempo real (no RT), tomando a este como su tarea de menor prioridad. Este último kernel es el que permite utilizar los programas existentes para Linux, mientras que el anterior asegura la actuación en tiempo real.

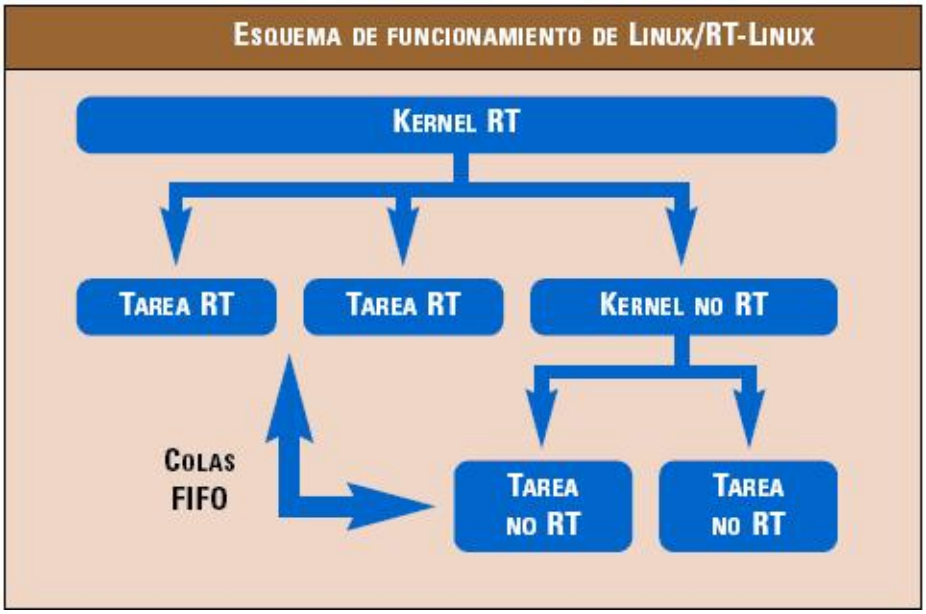


Figura 1.4: Funcionamiento de Linux/RT-Linux

1.3.2 Tecnología de comunicación

La tecnología de comunicación es de vital importancia en aplicaciones multi-sensoriales como la presente, donde una red de sensores es conectada a uno o varios procesadores. El desarrollo de un sistema flexible y fiable es todo un reto y para ello se deben tener en cuenta factores como:

1. Servicio de identificación y direcciones de red.
2. Conversiones de presentación (encriptación, compresión, reordenamiento de bytes, etc.).
3. Sincronización entre procesos.
4. Control de errores.
5. Procedimiento de llamada a subrutinas y bibliotecas.

1.3.3 Modelos de comunicación

La comunicación entre los distintos componentes puede ser implementada según alguno de los siguientes modelos.

1. Maestro/Esclavo (el maestro controla la comunicación con los clientes.)

2. Cliente/Servidor (el servidor a través de una interfase da servicio a los clientes)
3. Punto a Punto (P2P cada punto es a la vez cliente y servidor).
4. Grupo: Todos escuchan (Broadcast, Multicast).

Para la aplicación que se trata en concreto se ha pensado en una arquitectura dividida en dos partes, una que se ejecuta sobre el sistema operativo Linux y otra cuyos requerimientos temporales obligan a ejecutarse en tiempo real (RT-Linux). En la parte de Linux RT se encuentra el servidor implementado sobre el PC montado en el vehículo. El cliente (en tierra y abordo), que es la otra aplicación sin requerimientos temporales, puede emplear Linux.

El módulo de tiempo real (Muñoz, 2003).está formado por varias tareas periódicas de tiempo real (sensado de posición, comandos de movimiento y tarea de supervisión watchdog), y varias esporádicas (manejador y stop). (Muñoz, 2003).

Por su parte, la aplicación servidor está permanentemente esperando conexiones por parte de aplicaciones cliente. Una vez establecida una conexión, el servidor hace de intermediario intercambiando mensajes entre la aplicación cliente y el módulo de tiempo real que ejecuta las tareas de control.

1.3.4 Lenguajes de programación

Los lenguajes de programación, de acuerdo a Benítez (2003), sirven para escribir programas que permitan la comunicación usuario/máquina. Los programas especiales llamados traductores (compiladores e intérpretes) convierten las instrucciones escritas en lenguaje de programación, en instrucciones escritas en lenguaje máquina (0 y 1 bits), para que esta pueda entender.

Los lenguajes utilizados para escribir programas de computadoras que puedan ser entendidos por ellas se denominan lenguajes de programación.

Los lenguajes de programación se clasifican en tres grandes categorías:

1. Máquina.
2. Bajo nivel.

3. Alto nivel.

1.3.4.1 Lenguaje de máquina

Los lenguajes máquina (Maria, 2005), son aquellos cuyas instrucciones son directamente entendibles por la computadora y no necesitan traducción posterior para que la unidad central de procesamiento (UCP), pueda entender y ejecutar el programa.

Las instrucciones en lenguaje de máquina, se expresan en términos de la unidad de memoria más pequeña, el bit (dígito binario 0 o 1), en esencia una secuencia de bits que especifican la operación y las celdas implicadas en una operación. Este lenguaje es totalmente abstracto para el programador y por eso se hace necesario el empleo de un programa interfase.

1.3.4.2 Lenguaje de bajo nivel

Los lenguajes de bajo nivel han sido diseñados con el fin de adaptar la forma organizada del pensamiento humano a la forma abstracta del código máquina. Estos lenguajes dependen de la máquina, es decir, dependen de un conjunto de instrucciones específicas de la computadora.

Un lenguaje típico de bajo nivel es el lenguaje ensamblador. En este lenguaje las instrucciones se escriben en códigos alfabéticos conocidos como nemotécnicos (abreviaturas de palabras inglesas o españolas). El lenguaje ensamblador constituye la representación más directa del código máquina específico para cada arquitectura de computadoras legible por un programador. (Rene, 2005)

Este tipo de lenguaje fue usado ampliamente en el pasado para el desarrollo de software, pero actualmente sólo se utiliza cuando se requiere la manipulación directa del hardware o se pretenden rendimientos inusuales de los equipos.

1.3.4.3 Lenguaje de alto nivel

Los lenguajes de programación de alto nivel (BIC & SHAW,2003) (“C”, Ada, BASIC, FORTRAN, Pascal, etc.) son aquellos en los que las instrucciones o sentencias a la computadora son escritas con palabras similares a los lenguajes

humanos (en general lenguaje inglés), lo que facilita la escritura y compresión por el programador.

El C (Waite and Prata, 1990) es el lenguaje de programación de alto nivel más cercano al bajo nivel que se dispone y al igual que el ensamblador, es un lenguaje orientado a la implementación de **Sistemas Operativos**, concretamente Unix. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

El C dispone de las estructuras típicas de los lenguajes de alto nivel pero, a su vez, dispone de construcciones del lenguaje que permiten un control a muy bajo nivel. Los compiladores suelen ofrecer extensiones al lenguaje que posibilitan mezclar código en ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

La primera estandarización del lenguaje C fue por la ANSI, con el estándar X3.159-1989. El lenguaje que define este estándar fue conocido vulgarmente como ANSI C. Posteriormente, en 1990, fue ratificado como estándar ISO (ISO/IEC 9899:1990). La adopción de este estándar es muy amplia por lo que, si los programas creados lo siguen, el código es portable entre plataformas y/o arquitecturas.

1.3.5 Portabilidad

La portabilidad se alcanza al lograr un alto nivel de abstracción de los dispositivos de hardware y el sistema operativo. Se logra típicamente con el empleo de lenguajes universales como el Java y el Pitón. Estos lenguajes son independientes de la máquina, es decir, las sentencias del programa no dependen del diseño o hardware de una computadora específica.

1.3.6 Modularidad

La modularidad es una consideración importante cuando se diseñan grandes y complejos sistemas donde es muy útil poder desarrollarlos por segmentos para

probarlos de forma independiente. Esto también es importante cuando se considera el uso de simuladores de hardware.

1.3.7 Integración a sistemas existentes

La integración a sistemas ya desarrollados se basa en implementar con la mayor compatibilidad a los ya existentes, que típicamente tienen sus librerías en “C” o “C++” y sus comunicaciones orientadas a interfases serie.

1.3.8 Tolerancia a Fallos

La tolerancia a fallos (Pablo and Rogina, 1999, Smith, 1988) es otro de los factores a considerar en el desarrollo de la aplicación ya que un error, causado quizás por un problema de diseño, construcción, programación, un daño físico, condiciones ambientales adversas o un error humano, pueden aparecer tanto en el hardware como en el software.

EL fallo de un componente del sistema no conduce directamente al fallo del sistema, pero puede ser el comienzo de una serie de fallos que quizás sí terminen con el fallo del sistema.

El método general para la tolerancia de fallos es el uso de redundancia. Hay tres tipos posibles de redundancia:

- De información, donde se agrega un código como el de Hamming para transmitir los datos y recuperarse del ruido en la línea.
- De tiempo, en el que se realiza una acción y de ser necesario, se vuelve a realizar. Es de particular utilidad cuando los fallos son transitorios o intermitentes.
- Física, cuando se agrega un equipo adicional para permitir que el sistema tolere la pérdida o mal funcionamiento de algunos componentes.

1.4 Conclusiones

En el desarrollo de este capítulo se mostraron los distintos elementos que deben ser tenidos en cuenta en el desarrollo de la arquitectura de software, para cumplir los objetivos planteados, así como se hacen las valoraciones iniciales de los medios a emplear en cada una de las fases de desarrollo de la misma para su integración al sistema.

CAPÍTULO 2. Arquitectura de Hardware

El objetivo principal de este capítulo es el análisis de la arquitectura de hardware de una posible aplicación en la que se desplegarán los elementos de arquitectura de software planteados en el capítulo anterior. También se analizan los sistemas de sensores que la conforman, para el caso particular de un helicóptero, con capacidad de vuelo autónomo y sistema de visión.

2.1 Esquema general de la Arquitectura de Hardware

La arquitectura propuesta es la que se muestra en la figura 2.1, donde se destaca que la misma debe permitir alcanzar vuelo autónomo con un bajo coste de operación y un alto desempeño en el seguimiento de trayectorias para la toma de imágenes. El sistema de control de vuelo está basado en la combinación de los sistemas de posicionamiento global (GPS "Global Positioning System") y unidades de medición inercial (IMU "Inertial Measurement Unit").

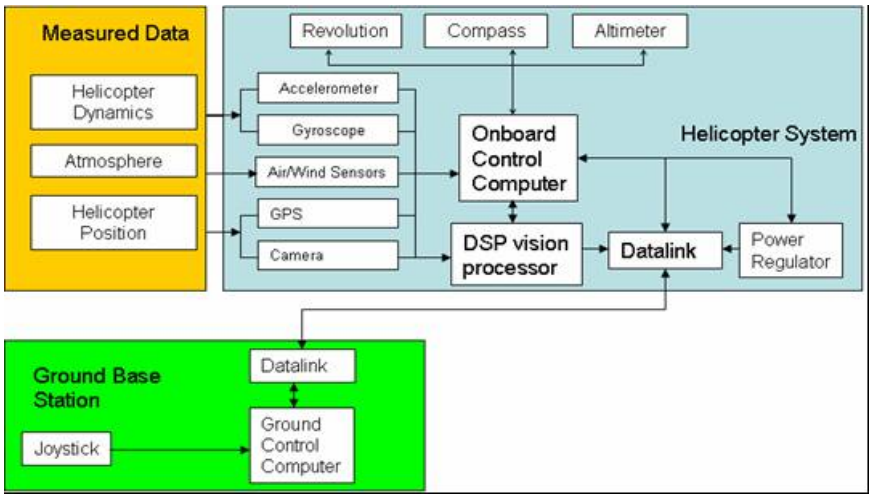


Figura 2.1: Arquitectura de hardware de un helicóptero autónomo

2.2 Sistemas de sensores que componen el sistema

Los sensores del sistema son los encargados de proveer todos los datos necesarios para el sistema de control de vuelo y pueden ser agrupados en los siguientes grupos:

1. Dinámica de vuelo.
2. Posición, altura y velocidad
3. Condiciones mecánicas.
4. Condiciones atmosféricas

2.2.1 Elementos que componen los sistemas de sensores

Los elementos que componen el sistema:

- GPS.
- IMU.
- Altímetros.
- Multi Sensor de estado.
- Sensor atmosférico.

Los mismos pueden ser agrupados de la siguiente forma:

- Dinámica de vuelo. (IMU)
- Posición, altura y velocidad (GPS, Altímetros)
- Condiciones mecánicas. (Multi Sensor de estado)
- Condiciones atmosféricas (Sensor atmosférico)

A esto debe ser sumada la PC industrial donde corre el software de control y sus elementos de comunicación con los sensores en tierra y los operadores remotos.

2.2.2 Característícas de los sensores

Todos los sensores empleados en la aplicación son de tipo inteligente, siendo sus interfaces de comunicación las que se muestran en la tabla 2.1 junto con su significación para la lógica de control.

Tabla 2.1: Interfaz de comunicación de los sensores empleados

Elemento	RS232	USB	Significación
GPS	X	X	Baja
IMU	X		Alta
Altímetros	X		Media
MODEM Radio		X	Media
LAN		X	Baja
Multi sensor de estado		X	Alta
Sensor atmosférico	X		Alta

2.2.2.1 GPS

El GPS, o sistema de posicionamiento global, es hoy en día el sensor más común a usar en la medición de posición y velocidad. Trabaja bajo un principio de triangulación con satélites en órbita que le permiten determinar en cualquier momento, los parámetros de longitud, latitud y altura (referida al nivel del mar) además de la velocidad del sensor que pide dicha información (Campoy et al., 2000). Los datos del GPS se pueden utilizar para ayudar a un INS (sistema de medición inercial) en derivar los datos de actitud y obtener velocidad filtrada y los datos filtrados de la posición.

El sistema GPS (Amidi et al., 1999) puede obtener excelentes precisiones inferiores a un metro de resolución en cualquiera de los ejes, no obstante este desempeño puede ser ampliamente mejorado con el uso de sistemas diferenciales (DGPS) como es el caso de la presente aplicación, donde una estación terrena en posición conocida retransmite mediante enlace inalámbrico correcciones al móvil. Esta acción mejora las precisiones al orden de los centímetros. Para el caso actual se decidió emplear un GPS modelo OEM4-G2-RT2W de la firma NOVATEL del que se muestran las características en la tabla 2.2.

Tabla 2.2: Características de un GPS modelo OEM4-G2-RT2W

Desempeño	
Precisión en la Posición	Punto Simple 1.8 m CEP
	Punto Simple L1/L2 1.5 m CEP
	DGPS (L1, C/A) 0.45 m CEP
	RT20 2 < 20 cm CEP
	RT2 1 cm + 1 ppm
Comunicación	1 Puerto RS232 velocidad de 300 a 921,600 bps 1 Puerto RS232 velocidad de 300 a 230,400 bps 1 Puerto USB velocidad 5 Mbps
Razón de Datos	Medición 20 Hz
	Posición 20 Hz
Precisión de velocidad 1	0.03 m/s RMS

2.2.2.2 IMU

La IMU (Inertial Measurement Unit) es el núcleo del sistema de estabilización y control de vuelo del helicóptero, integrando las funciones descritas en la figura 2.2

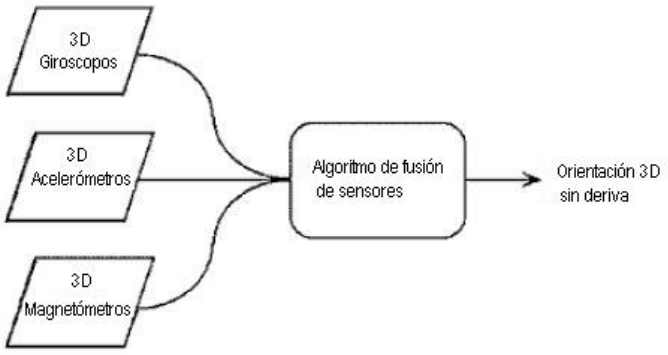


Figura 2. 2: Esquema funcional de la “IMU”

Para la presente aplicación se decidió emplear el modelo MTI de la firma Xsens, dicho sistema integra giróscopos de referencia para la medición de altura y orientación (AHRS en inglés), utilizando sensores inerciales de tecnología MEMS. Su procesador interno, de bajo consumo, provee una orientación 3D sin deriva, junto con las aceleraciones triaxiales, las velocidades angulares 3D y la orientación en el campo magnético 3D. El MTi es una unidad de medida de probado valor en la estabilización y control de cámaras, robots, vehículos y otro

equipamiento. Sus principales características tecnológicas son las que se muestran en la tabla 2.3.

Tabla 2.3: Características tecnológicas del MTI

Rango Dinámico	3D en todos los ángulos
Resolución angular	0.05 grados RMS
Precisión estática	<1.0 grados
Precisión dinámica	3.0 grados RMS
Comunicación	RS-232

2.2.2.3 Altímetros

Los altímetros (radares), al igual que la IMU, emplean comunicación serie RS232 y su función en la aplicación varía en dependencia de la ubicación física en el móvil, en un caso detectando la altura sobre el terreno para apoyar el vuelo controlado y las funciones de aterrizaje (típicamente láser) y en el caso de ser colocado en la parte frontal, su uso es como radar para prevenir colisiones con obstáculos.

Este sensor de tipo ultrasónico (Talaya et al., 2004) de bajo coste, se basa en la emisión de un pulso de ultrasonido cuyo lóbulo, o campo de acción, es de forma cónica. Midiendo el tiempo que transcurre entre la emisión del sonido y la percepción del eco se puede establecer la distancia a la que se encuentra el obstáculo que ha producido la reflexión de la onda. Estos elementos no requieren altas frecuencias de muestreo (<1Hz) debido a la baja velocidad de traslación del móvil.

2.2.2.4 Multisensor de estado

El multisensor (Cerro.I,2000) de estado se encarga de medir los distintos parámetros propios del móvil, tales como: las revoluciones por minuto del motor (RPM), temperatura del mismo, voltaje de las baterías y nivel de combustible, entre otros.

2.2.2.5 Sensor atmosférico

El sensor atmosférico (Barbariol, 1995) es el encargado de sensar distintos parámetros atmosféricos que pueden constituir disturbios para el sistema de control. Entre los mismos se encuentra fuerza del viento, presión atmosférica, temperatura y humedad.

2.2.3 PC y enlaces de comunicación

Estos elementos tienen especial significación debido a que sobre ellos pesa la estabilidad del sistema ya que son los encargados de recoger y transmitir las señales muestreadas integrándolas al software de control y de supervisión de usuario. Para la PC se seleccionó un formato industrial estándar tipo PC104 con ampliación de puertos de comunicación y para el caso de los enlaces de comunicación, un MODEM Radio y una LAN inalámbrica que brindan diversidad de velocidades y posibilidades de redundancia en el canal de comunicaciones con tierra.

2.3 Conclusiones

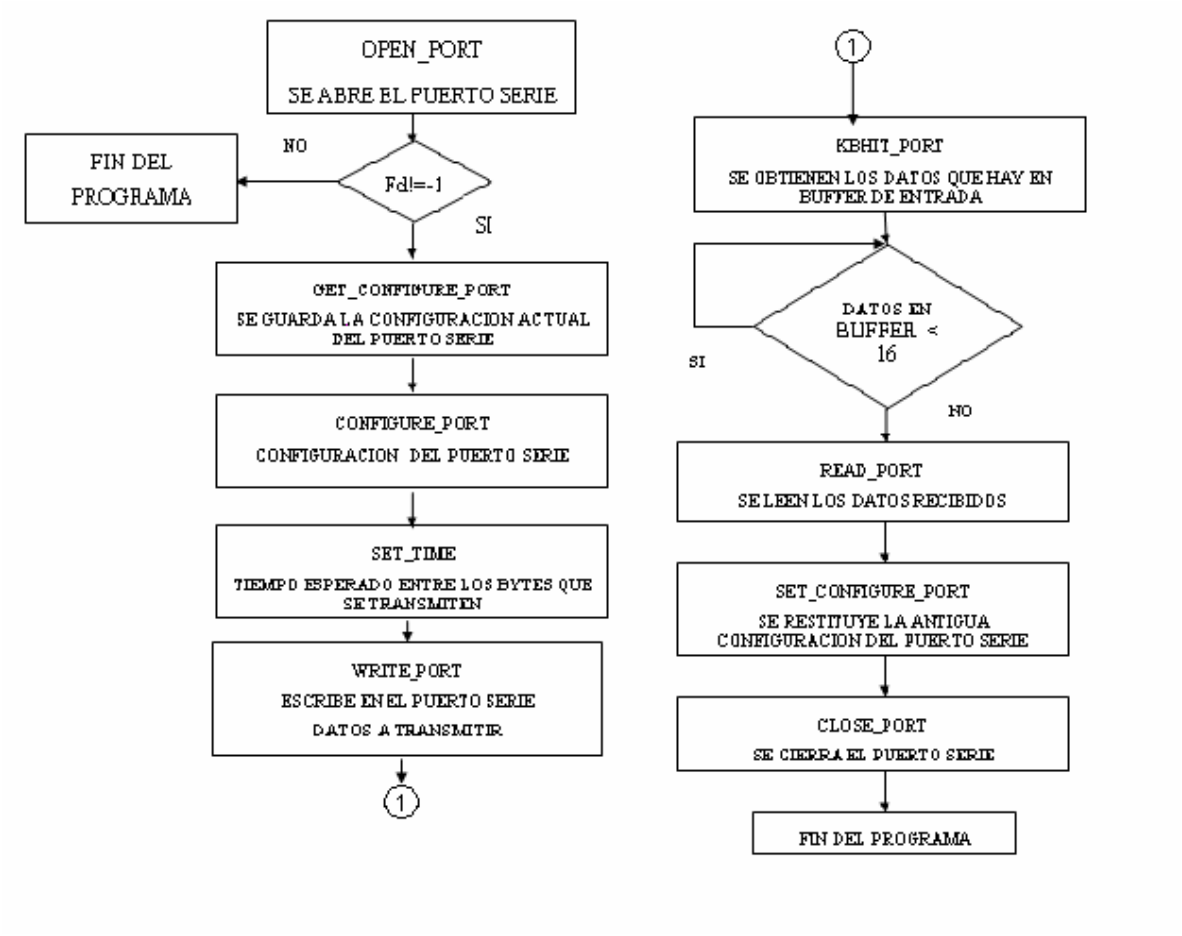
En este capítulo se muestra la arquitectura de hardware donde se va a desarrollar e implementar el software de control de vuelo, señalando las interfases de los distintos sensores y su relación con la lógica general del sistema. Es de destacar que el estudio de estos sensores es la base para desarrollar el software de comunicación de bajo nivel encargado de transportar la información de los sensores a la PC.

CAPÍTULO 3. IMPLEMENTACION DE SOFTWARE DE COMUNICACION

Este capítulo tiene como objetivo realizar el código de programación de un software capaz de lograr la comunicación por puerto serie para así poder conectar los elementos de hardware ya analizados en el capítulo anterior, además de lograr un código que sea capaz de realizar la tolerancia a fallos.

3.1 Diagrama de flujo del programa

En la figura 3.1 se muestra el digrama de flujo del programa para la comunicación seire, el cual comienza con la apertura de puerto. Una vez abierto el se toma la configuracion actual y se guarda la misma. Luego el mismo es configurado de acuerdo a las necesidades del proceso y a continuación se establece el tiempo de espera entre byte. A partir de aquí comienza el proceso de comunicación. Una vezculminado el mismo se restituye la configuración inicial del puerto y termina el programa.



Figuara 3.1: Esquema general del diagrama de flujo para la comunicación serie

3.2 Software de comunicación de por puerto serie bajo linux

Una vez descrito el programa de flujo del algoritmo para la comunicación serie, se procedió a su programación en lenguaje C. A continuación se presenta el programa desarrollado, empleando las instrucciones del C. Se ha considerado que la mejor forma de explicar cada paso del programa es a través de lso comentarios expuestos a través del programa.

El programa sigue los mimos pasos detallados en el diagrama de flujo. A continuación se plasma el programa desarrollado. Es importante resaltar que en el

Anexo I se presenta la librería “seria.h” cuyas funciomes son invocadas desde el programa principal.

```
#define __LINUX_COM__
Software de comunicacion por puerto serie bajo Linux

#include "serial.h"

int main()
{
   HANDLE fd;
   DCB OldConf;
   char cad[16]="Enviando Texto";
   int n,TIME=2;
fd=Open_Port("/dev/ttyS0");          /* se abre el puerto de
                                       comunicaciones,recibe como
                                       parámetro una cadena con el
                                       nombre del puerto y devuelve
                                       una variable de tipo HANDLE
                                       que es el manejador del puerto
                                      */
if(fd!=INVALID_HANDLE_VALUE)
{

OldConf=Get_Configure_Port(fd);     /* se guarda la configuración del
                                       Puerto,esta función devuelve
                                      una variable de tipo DCB con la
                                      configuración actual del puerto
                                      serie la función recibe un
                                      parámetro de tipo HANDLE
```

```
                                    que es el manejador devuelto
                                    por la función Open Port */

Configure_Port(fd,B115200,"8N1");   /*se configura el puerto serie
                                     Esta función configura el puerto
                                    serie con los parámetros fd,
                                     BautRate,CharParity.
                                    fd:Es el manejador del puerto
                                    devuelto por Open_port.
                                    BaudRate:Es la velocidad del
                                    puerto. (B115200, B19200, B9600,
                                     ...)
                                    CharParity:indica el número de
                                    bits de la transmisión.
                                    ("8N1","7E1","7O1","7S1")*/
Set_Time(fd,TIME);                  /* time-out entre caracteres es
                                     TIME*0.1
                                     Recive como variable el
                                     manejador del puerto y el máximo
                                     tiempo entre bytes en
                                    milisegundos*/
                                     fd:Es el manejador del puerto
                                    devuelto por Open_port.
                                    Time:multiplicador en m-seg, para
                                    el tamaño total de time-out en
                                    read y write.
                                    Timeout = (Time-m-seg *
                                     number_of_bytes)

n=Write_Port(fd,cad,16);            /* Escribo en el puerto serie
```

```
                                    los SizeData
                                    fd: Es el manejador del puerto
                                    devuelto por Open_port.
                                    Data : Es el dato a mandar(cad).
                                    SizeData: es el numero de bytes
                                    que se quieren escribir(16). */
while(Kbhit_Port(fd)<16);           /* Espero a leer hasta que se
                                    tengan 16 bytes en el buffer de
                                    entrada */
 n=Read_Port(fd,cad,16);           /* Lee los SizeData primeros
                                    caracteres del puerto serie y
                                    los carga en Data
                                    fd:Es el manejador del puerto
                                    devuelto por Open_port.
                                    Data:Es la variable en donde se
                                    reciben los datos.
                                    SizeData: es el numero de bytes
                                    que se quieren recibir.
                                   */
Set_Configure_Port(fd,OldConf);    /* Restituye la antigua
                                      Configuracion del puerto,
                                      Los parámetros serán pasados
                                      mediante una variable tipo DCB
                                    */
 Close_Port(fd);                    /* recibo la variable fd y
  }                                  Cierro el puerto serie */
printf("Presione ENTER para terminar\n");
   getchar();

   return 0;
}
```

3.3 Tolerancia a fallas

Mediante el control de errores de la información se analizara la tolerancia a fallos implementando un pequeño código de redundancia cíclica (CRC)

Los cuatro últimos bytes enviados contienen información redundante del resto de la trama, codificada en los bytes CRC3, CRC2, CRC1, CRC0. Para generar la redundancia se utiliza un Código de Redundancia Cíclica de 16 bits (CRC)

El polinomio utilizado es el CRC-CCITT(1021h): x16+x12+x5+1

```
/* Check received CRC */
/* cz2100c.lib/cz2_crc */
int cz2_crc(char read_frame[],char crca[])
{
static unsigned short i,data;
static pos;
static unsigned short accum,crc,genpoly=0x1021;
accum=0;
crc=0;
pos=0;
if (strlen(read_frame)>4)
{
while (pos!=strlen(read_frame)-4)
{
data=read_frame[pos++]<<8;
for (i=8;i>0;i--)
{
```

```
if ((data^accum)&0x8000)
accum = (accum<<1)^genpoly;
else
accum<<=1;
data<<=1;
}
}
for ( pos=4; pos>=1 ;pos-- )
{
crc<<=4;
crc|=read_frame[strlen(read_frame)-pos] & 0xf;
}
}
else
accum=1;
return ( accum == crc );
}
```

3.4 Análisis económico

El presente trabajo tributa al proyecto del grupo GIMAS “Desarrollo de un vehiculo autónomo aéreo” en el mismo se tiene como meta la implementación de un helicóptero autónomo capaz de desplazarse a través de la atmósfera por sí mismo, por trayectorias definidas, con posicionamiento georeferenciado de una elevada precisión, capaz de portar una cámara.

Dicho proyecto esta valorado económicamente para su desarrollo en aproximadamente 30000 USD y esta tarea en concreto pertenece al cronograma del primer año de ejecución. En el que se debe alcanzar las arquitecturas de Hardware y Software de la aplicación. Dicho proyecto involucra a nuestra Universidad, La universidad Libre de Bruselas y la empresa GEOCUBA.

3.5 Conclusiones del capítulo

El software demostró su valides en las pruebas de campo ejecutas mediante el establecimiento de una comunicación PC-PC y PC-IMU comprobándose que puede ser ajustado a las distintas necesidades de los sensores empleados, satisfaciendo el proceso de comunicación en entorno Linux. Es altamente portable debido a que se a seguido las reglas de programación en C y resulta de reducido gasto computacional.

CONCLUSIONES Y RECOMENDACIONES

Conclusiones

Como resultado de este trabajo, se presenta la arquitectura básica de Hardware para el Helicóptero, así como es planteado un análisis de cada uno de los elementos que la componen, cumpliendo con los requerimientos de peso y volumen imprescindibles para este tipo de aplicación.

Como complemento preciso son presentadas las consideraciones para la arquitectura básica de software capaz de unificar el sistema con fiabilidad y robustez dado el alto nivel de exigencias a que va a estar sometido el sistema.

El estudio realizado de los distintos componentes de hardware que conforman la aplicación está basado en elementos tales como: métodos de comunicación que emplean, sus frecuencias de muestreo y su nivel de significación en la estructura del conjunto. Estos factores, unidos al sistema de control, constituyen la plataforma central para la conformación de la arquitectura de software en base a los objetivos y restricciones descubiertos.

Finalmente, se logró hacer el diseño del software para la comunicación serie de los sistemas de sensores con la unidad central que van a ser colocados en el helicóptero autónomo usando el sistema operativo Linux y el lenguaje de programación C, así como el análisis de la tolerancia a fallo mediante la programación en C de un código de redundancia cíclica (CRC).

Recomendaciones

Se debe comenzar a trabajar en el planificador de tareas que manipulara el software de comunicación serie diseñado en esta tesis, para lograr la robustez y el desempeño necesario en la atención simultánea a los distintos sensores de la aplicación.

REFERENCIAS BIBLIOGRÁFICAS

AGUIRRE, CERRO, D., BARRIENTOS & GAMBAO (1998) Vehículos Aéreos Autónomos. *Revista de Ingeniería Industrial***,** 12-18.

AMIDI, O., KANADE, T. & FUJITA, K. (1999) A visual odometer for autonomous helicopter ight. *Robotics and Autonomous Systems*.

BARBARIOL (1995) *Implementation of an Autonomous Helicopter Flight Controller* Universidad de Linköping, Suecia

BENITEZ, K. (2003) Sistema Operativo del Pc http://mail.umc.edu.ve/opsu/contenidos/sistemaoperativo_computador.htm,accedido el 20-5-2007

BIC & SHAW (2003) Operating Systems Principles

BURLESON, HARING & PERGANTIS (1999) Reporte técnico del sistema SRS-99. USA, Universidad Politécnica Estatal del Sur.

CAMPOY, P., BARRIENTOS, A., GARCIA-PARDO, P. J., CERRO, J. D. & AGUIRRE, I. (2000) An autonomous helicopter guided by computer vision for visual inspection of overhead power cable. *In Proceedings of the 5th International Conference on Live Maintenance.* Madrid.

CERRO, I. J. D. & A, B. (2000) *Development of an autonomous minihelicopter for power lines inspection,* Boston.

ERICK, C. (2006) *CoRoBa, a Framework for Multi-Sensor Robotic Systems Integration* Vrije Universiteit Brussel, Belgium.

FAGG, LEWIS, MONTGOMERY & BEKEY (1993) The USC Autonomous Flying Vehicle: An Experiment in Real-Time Behavior-Based Control. *Conferencia Internacional de Sistemas y Robots Inteligentes.*

FARRELL, J. A. (1998) The Global Positioning System & Inertial Navigation. *McGraw-Hill Professional*.

FIELDING, R. T. (2000) *Architectural styles and the design of network-based software architectures* University of California, Irvine

FORSSELL, B. (1991) Radionavigation Systems. *Prentice Hall International*.

GARCIA, P. J., SUKHATME, G. S. & MONTGOMERY, J. F. (2001) Towards vision-based safe landing for an autonomous helicopter. *IEEE*.

GETTING, I. A. (1993) The Global Positioning System. *IEEE Spectrum*.

HILLIARD, R. (2000) Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE*.

JACOBSON, I., BOOCH, G. & RUMBAUGH, J. (1999) El Proceso Unificado de Desarrollo de Software. *Addison-Wesley*.

KRUCHTEN, P. (1995) Architectural Blueprints--The 4+1 View Model of Software Architecture. *IEEE Software***,** 42-50.

MUÑOZ, J. J. C. (2003) Sistemas de Tiempo Real. www.led.uc.edu.py,accedido el 20-6-2007

PABLO & ROGINA (1999) *Tolerancia a Fallos en Sistemas de Tiempo Real* Buenos Aires, Argentina

RIPOLL, I. & ACOSTA, E. (2000) Receptor de Mando a Distancia con RTLinux http://www.linuxfocus.org,accedido el 14-5-2007

RUBINI, A. & CORBERT, J. (2001) Linux device drivers. *IEEE*.

SMITH, J. (1988) *A Survey of Software Fault Tolerance Techniques* Columbia University, USA

TALAYA, J., ALAMÚS, R., BOSCH, E., SERRA, A., KORNUS, W. & BARON, A. (2004) Integración de un escáner láser terrestre con sensores de orientación GPS/IMU. *IEEE*.

TANENBAUM & WOODHULL (2003) *Sistemas Operativos Modernos,* Madrid.

WAITE, M. & PRATA, S. (1990) *Programación en C,* Madrid.

RENE, R (2005) Definiciones y conceptos preliminares.
http://homepage.mac.com/eravila/asmix861.html el 7 -4-2007

Maria, E. (2005) Lenguajes de Programacion Universidad de Madrid

ANEXOS

Anexo I libreria ‘seria.h’

```
/** \file
 * \brief Archivo usado para SO Linux.
 *
 * Este archivo contiene la definicon de las funciones de
 * de acceso al puerto, destinadas a SO tipo Linux
 *
 */

/** \defgroup HeaderLinux Funciones para Linux
 * @{
 */

#ifndef __SERIE_LINUX__
#define __SERIE_LINUX__

 #include <stdio.h> /* Standard input/output definitions */
 #include <string.h> /* String function definitions */
 #include <unistd.h> /* UNIX standard function definitions */
 #include <fcntl.h> /* File control definitions */
 #include <sys/ioctl.h>

 #include <termios.h> /* POSIX terminal control definitions */

 #ifndef FALSE
 #define FALSE 0
 #endif

 #ifndef TRUE
```

```
#define TRUE 1
#endif

#ifndef BOOL
#define BOOL int
#endif

#define INVALID_HANDLE_VALUE -1
#define NONE 0
#define RTSCTS 1
#define HARD 1
#define XONXOFF 2
#define SOFT 2

BOOL ERROR_CONFIGURE_PORT=FALSE;

typedef struct termios       DCB;
typedef int         HANDLE;

/**
 * @}
 */

 /** \var HANDLE Open_Port(char COMx[])
 * \brief Abre el puerto de comunicaciones
 *       \param COMx es el puerto a abrir "/dev/ttyS0","/dev/ttyS1","/dev/ttyACM0","/dev/ttyUSB0",...
 * \return El manejador del puerto
 * \ingroup HeaderLinux
 */
 HANDLE Open_Port(char COMx[])
 {
        int fd; // File descriptor for the port
```

```
fd = open(COMx, O_RDWR | O_NOCTTY );//| O_NDELAY);

        if (fd <0)
        {
                printf("open_port:fd=%d: No se puede abrir %s\n",fd,COMx);
                return INVALID_HANDLE_VALUE;
        }
        printf("open_port:fd=%d: Abierto puerto %s\n",fd,COMx);
        tcflush(fd, TCIOFLUSH);
        return (fd);
}


/** \var DCB Get_Configure_Port(HANDLE fd)
* \brief Devuelve la configuracion Actual del Puerto serie
* \param fd Es el manejador del puerto
* \return Una estructuracon DCB con la configuracion actual del puerto
* \ingroup HeaderLinux
*/
DCB Get_Configure_Port(HANDLE fd)
{

        struct termios oldtio;
        if(tcgetattr(fd,&oldtio)!=0) /* almacenamos la configuracion actual del puerto */
        {
printf("Error Pidiendo configuracion de Puerto\n");
ERROR_CONFIGURE_PORT=TRUE;
return oldtio;
        }
        ERROR_CONFIGURE_PORT=FALSE;
        return oldtio;
}
```

```
/** \var DCB Configure_Port(HANDLE fd,unsigned int BaudRate,char CharParity[])
 * \brief Configura el puerto serie
 * \param fd Es el manejador del puerto
 * \param BaudRate Es la velocidad del puerto
 * \param CharParity indica el numero de bits de la transmision "8N1","7E1","7O1","7S1"
 * \return Una estructuracon Bcd con la configuracion del puerto
 * \ingroup HeaderLinux
 */
DCB Configure_Port(HANDLE fd,unsigned int BaudRate,char CharParity[])
{
    DCB newtio;
    bzero(&newtio, sizeof(newtio)); //limpiamos struct para recibir los
//nuevos parametros del puerto
    //tcflush(fd, TCIOFLUSH);


//CLOCAL : conexion local, sin control de modem
    //CREAD : activa recepcion de caracteres
    newtio.c_cflag =CLOCAL | CREAD ;

    cfsetispeed(&newtio,BaudRate);
    cfsetospeed(&newtio,BaudRate);

    if(strncmp(CharParity,"8N1",3)=0)     //CS: 8n1(8bit,no paridad,1bitde
parada)
    {
        newtio.c_cflag &= ~PARENB;
        newtio.c_cflag &= ~CSTOPB;
        newtio.c_cflag &= ~CSIZE;
        newtio.c_cflag |= CS8;
```

```
        }
        if(strncmp(CharParity,"7E1",3)==0)
        {
                newtio.c_cflag |= PARENB;
                newtio.c_cflag &= ~PARODD;
                newtio.c_cflag &= ~CSTOPB;
                newtio.c_cflag &= ~CSIZE;
                newtio.c_cflag |= CS7;
        }
        if(strncmp(CharParity,"7O1",3)==0)
        {
                newtio.c_cflag |= PARENB;
                newtio.c_cflag |= PARODD;
                newtio.c_cflag &= ~CSTOPB;
                newtio.c_cflag &= ~CSIZE;
                newtio.c_cflag |= CS7;
        }
        if(strncmp(CharParity,"7S1",3)==0)
        {
                newtio.c_cflag &= ~PARENB;
                newtio.c_cflag &= ~CSTOPB;
                newtio.c_cflag &= ~CSIZE;
                newtio.c_cflag |= CS8;
        }

// IGNPAR : ignora los bytes con error de paridad
// ICRNL : mapea CR a NL (en otro caso una entrada CR del otro ordenador
// no terminaria la entrada) en otro caso hace un dispositivo en bruto
// (sin otro proceso de entrada)
        newtio.c_iflag = 0;
         //newtio.c_iflag = IGNPAR;
         //newtio.c_iflag |= ICRNL;
```

```
//Salida en bruto.
        newtio.c_oflag = 0;

//ICANON : activa entrada canonica(Modo texto)
//desactiva todas las funcionalidades del eco, y no envia segnales al
//programa llamador
         //newtio.c_lflag = ICANON;
        newtio.c_lflag = 0;

// inicializa todos los caracteres de control
// los valores por defecto se pueden encontrar en /usr/include/termios.h,
// y vienen dadas en los comentarios, pero no los necesitamos aqui

        newtio.c_cc[VTIME] = 0; /* temporizador entre caracter, no usado */
        newtio.c_cc[VMIN] = 1; /* bloqu.lectura hasta llegada de caracter. 1 */



        if(tcsetattr(fd,TCSANOW,&newtio)!=0)
        {
                printf("ERROR: No se pudo poner Comfiguracion del Puerto\n" );
                ERROR_CONFIGURE_PORT=TRUE;
                return newtio;
        }

        return newtio;
}


/** \var int Set_Configure_Port(HANDLE fd,DCB PortDCB)
* \brief Coloca la configuracion en el puerto serie
* \param fd Es el manejador del puerto
* \param PortDCB es la configuracion del puerto
```

```
* \return TRUE si todo fue bien o FALSE si no
* \ingroup HeaderLinux
*/
int Set_Configure_Port(HANDLE fd,DCB newtio)
{
      // ahora limpiamos el buffer de entrada y salidadel modem y activamos
// la configuracion del puerto
//tcflush(fd, TCIOFLUSH);

      if(tcsetattr(fd,TCSANOW,&newtio)!=0)
      {
            printf("ERROR: No se pudo poner Comfiguracion del Puerto\n" );
      ERROR_CONFIGURE_PORT=TRUE;
      return FALSE;
      }
       ERROR_CONFIGURE_PORT=FALSE;

      return TRUE;
}


/** \var long Write_Port(HANDLE fd,char Data[],int SizeData)
* \brief Escribe en el puerto serie
* \param fd Es el manejador del puerto
* \param Data Es el dato a mandar
* \param SizeData es el tamaño del dato
* \return En caso de éxito, se devuelve el número de bytes escritos (cero
* indica que no se ha escrito nada). En caso de error, se devuelve -1
* \ingroup HeaderLinux
*/
long Write_Port(HANDLE fd,char Data[],int SizeData)
{
      return write(fd,Data,SizeData);
```

```
}


/** \var long Read_Port(HANDLE fd,char *Data,int SizeData)
* \brief Recibe datos en el puerto serie
* \param fd Es el manejador del puerto
* \param Data Es el arreglo donde se almacenarán los datos recibidos
* \param SizeData es el tamaño del arreglo
* \return En caso de éxito, se devuelve el número de bytes recibidos (cero
* indica que no se ha recibido nada). En caso de error, se devuelve -1
* \ingroup HeaderLinux
*/
long Read_Port(HANDLE fd,char *Data,int SizeData)
{
	struct termios newtio;
	int bytes;

	if(tcgetattr(fd,&newtio)!=0) return -1;
	do
	{ioctl(fd, FIONREAD, &bytes); }
	while((bytes<SizeData)&&(newtio.c_cc[VMIN]!=0));

	if(bytes>0)return read(fd,Data,SizeData);
	else		return 0;
}

/** \var long Gets_Port(HANDLE fd,char *Data,int SizeData)
* \brief Recibe datos en el puerto serie,lee hasta encontrar un 0x0A,0x0D
* (rellenando el siguiente byte con un cero - "solo si existe")
* o hasta completar SizeData caracteres.
* \param fd Es el manejador del puerto
* \param Data Es el arreglo donde se almacenarán los datos recibidos
* \param SizeData es el tamaño maximo del arreglo
```

```
* \return En caso de éxito, se devuelve el número de bytes recibidos (cero
* indica que no se ha recibido nada). En caso de error, se devuelve -1
* \ingroup HeaderLinux
*/
long Gets_Port(HANDLE fd,char *Data,int SizeData)
{
long n=0,i=0;

for(i=0;i<SizeData;i++)
{
read(fd,&Data[i],1);
if((Data[i]==13)||(Data[i]==10)&&(i!=0))
{
n=i+1;
if(n<SizeData) Data[n]=0;
i=SizeData;
}
}

return n;
}

/** \var Getc_Port(HANDLE fd,char *Data)
* \brief Recibe un caracter en el puerto serie.
* \param fd Es el manejador del puerto
* \param Data Es el dato(8-bit) a mandar
* \return En caso de éxito, se devuelve el número de bytes recibidos (cero
* indica que no se ha recibido nada). En caso de error, se devuelve -1
* \ingroup HeaderLinux
*/
long Getc_Port(HANDLE fd,char *Data)
{
long n;
```

```
n=read(fd,Data,1);

return n;
}


/** \var int Kbhit_Port(HANDLE fd)
* \brief Indica el numero de caracteres disponible en el buffer de entrada.
* \param fd Es el manejador del puerto.
* \return El numero de caracteres en el buffer de recepcion.
* \ingroup HeaderLinux
*/
int Kbhit_Port(HANDLE fd)
{
int bytes;
ioctl(fd, FIONREAD, &bytes);
return bytes;

}


/** \var int Close_Port(HANDLE hComm)
* \brief Cierra el puerto serie.
* \param fd Es el manejador del puerto.
* \return TRUE si se ha cerrado el Puerto y FALSE en el caso contrario.
* \ingroup HeaderLinux
*/
int Close_Port(HANDLE fd)
{
        if (fd != INVALID_HANDLE_VALUE)
{ // Close the communication port.
```

```
            // ahora limpiamos el buffer de entrada y salidadel modem y activamos
            // la configuracion del puerto
            //tcflush(fd, TCIOFLUSH);
 if (close(fd)!=0)
 {printf("Error cerrando archivo\n");return FALSE;}
 else
 {fd = INVALID_HANDLE_VALUE;return TRUE;}
 }
        return FALSE;
 }


 /** \var int Set_Hands_Haking(HANDLE fd,int FlowControl)
 * \brief Configura el control de flujo en el puerto serie
 * \param fd Es el manejador del puerto.
 * \param FlowControl
 * 0 Ninguno<br>
 * 1 RTS/CTS<br>
 * 2 Xon/Xoff<br>
 * 3 DTR/DSR
 * \return TRUE si todo fue bien y FALSE si no lo fue.
 * \ingroup HeaderLinux
 */
 int Set_Hands_Haking(HANDLE fd,int FlowControl)
 {
        struct termios newtio;
        tcgetattr(fd,&newtio); /* almacenamos la configuracion actual del puerto */
        switch (FlowControl)
        {
                case 0://NONE
                {
                        newtio.c_cflag &= ~CRTSCTS;
```

```
                newtio.c_iflag &= ~(IXON | IXOFF | IXANY);
                newtio.c_cc[VSTART] = 0; /* Ctrl-q */
                newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
                break;
        }
        case 1://RTS/CTS - HARD
        {
                newtio.c_cflag |= CRTSCTS;
                newtio.c_iflag &= ~(IXON | IXOFF | IXANY);
                newtio.c_cc[VSTART] = 0; /* Ctrl-q */
                newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
                break;
        }
        case 2://XON/XOFF - SOFT
        {
                newtio.c_cflag &= ~CRTSCTS;
                newtio.c_iflag |= (IXON | IXOFF );//| IXANY);
                newtio.c_cc[VSTART] = 17; /* Ctrl-q */
                newtio.c_cc[VSTOP] = 19; /* Ctrl-s */
                break;
        }
    }
    tcsetattr(fd, TCSANOW, &newtio);
    return 0;
}
```

/ \var int Set_RThreshold(HANDLE fd,int n)**
*** \brief configura el numero minimo de caracteres que permitira**
*** que se ejecute la lectura del puerto**
*** \param fd Es el manejador del puerto.**
*** \param n es el numero de caracteres que activara la lectura**
*** \return TRUE si todo fue bien y FALSE si no lo fue.**

```
* \ingroup HeaderLinux
*/
int Set_RThreshold(HANDLE fd,int n)
{
        return TRUE;
}

/** \var int Set_BaudRate(HANDLE fd,unsigned int BaudRate)
* \brief configura la velocidad puerto serie
* \param fd Es el manejador del puerto.
* \param BaudRate Es la velocidad del PUerto
* \return TRUE si todo fue bien y FALSE si no lo fue.
* \ingroup HeaderLinux
*/
int Set_BaudRate(HANDLE fd,unsigned int BaudRate)
{
        struct termios newtio;

        if(tcgetattr(fd,&newtio)!=0)
{
printf("Error obteniendo configuracion time-out actual\n");
return FALSE;
}

        cfsetispeed(&newtio, BaudRate);
        cfsetospeed(&newtio, BaudRate);

        if(tcsetattr(fd, TCSANOW, &newtio)!=0)
{
printf("Error estableciendo nueva configuracion time-out\n");
return FALSE;
}
```

```
        return TRUE;
}


/** \var int Set_Time(HANDLE fd,unsigned int Time)
* \brief configura Temporizador para read y write
* \param fd Es el manejador del puerto.
* \param Time Tiempo entre bits, T=Time*0.1s, para tamaño total de time-out
* en read y write.<br>
* Timeout = (Time *0.1* number_of_bytes) seg
* \return TRUE si todo fue bien y FALSE si no lo fue.
* \ingroup HeaderLinux
*/
int Set_Time(HANDLE fd,unsigned int Time) //t =Time*0.1 s)
{
        struct termios newtio;
/* almacenamos la configuracion actual del puerto */
        if(tcgetattr(fd,&newtio)!=0)
{
printf("Error obteniendo configuracion time-out actual\n");
return FALSE;
}

        newtio.c_cc[VTIME] = Time; /*temporizador entre caracter*/
        newtio.c_cc[VMIN] = 0; /*bloquea lectura hasta llegada de
caracter 1 */

        if(tcsetattr(fd, TCSANOW, &newtio)!=0)
{
printf("Error estableciendo nueva configuracion time-out\n");
return FALSE;
}
```

```
        return TRUE;
}

/** \var int IO_Blocking(HANDLE fd,int Modo)
* \brief configura Temporizador para read y write
* \param fd Es el manejador del puerto.
* \param Modo<br>
* TRUE : Modo bloqueante<br>
* FALSE: Modo no bloqueante
* \return TRUE si todo fue bien y FALSE si no lo fue.
* \ingroup HeaderLinux
*/
int IO_Blocking(HANDLE fd,int Modo)
{
      struct termios newtio;
/* almacenamos la configuracion actual del puerto */
      if(tcgetattr(fd,&newtio)!=0)
{
printf("Error obteniendo configuracion time-out actual\n");
return FALSE;
}

      if(Modo==FALSE)
      {
            newtio.c_cc[VTIME] = 0; /* temporizador entre caracter*/
            newtio.c_cc[VMIN] = 0; /* bloqu.lectura hasta llegada de caracter. 1 */
      }
      if(Modo==TRUE)
      {
            newtio.c_cc[VTIME] = 0; /* temporizador entre caracter*/
            newtio.c_cc[VMIN] = 1; /* bloqu.lectura hasta llegada de caracter. 1 */
      }
```

```
	if(tcsetattr(fd, TCSANOW, &newtio)!=0)
{
printf("Error estableciendo nueva configuracion time-out\n");
return FALSE;
}

	return TRUE;
}
/** \var int Clean_Buffer(HANDLE fd)
* \brief Termina las operaciones de lectura y escritura pendientes y limpia
* las colas de recepción y de transmisión.
* \param fd Es el manejador del puerto.
* \return TRUE si todo fue bien y FALSE si no lo fue.
* \ingroup HeaderLinux
*/
int Clean_Buffer(HANDLE fd)
{
	if(tcflush(fd, TCIOFLUSH)!=0)
{
printf("Error Limpiando el Buffer de entrada y salida\n");
return FALSE;
}
return TRUE;
}

/** \var int Setup_Comm(HANDLE fd,unsigned long InQueue,unsigned long OutQueue)
* \brief Especifica el tamaño en Bytes del buffer de entrada y salida
* \param fd Es el manejador del puerto.
* \param InQueue Especifica el tamaño en Bytes del buffer de entrada, se
* Recomienda el uso de numero pares.
* \param OutQueue Especifica el tamaño en Bytes del buffer de salida, se
* Recomienda el uso de numero pares.
```

```
* \return TRUE si todo fue bien y FALSE si no lo fue.
*
*/
int Setup_Buffer(HANDLE fd,unsigned long InQueue,unsigned long OutQueue)
{
return TRUE;
}

/** \var HANDLE Create_Thread_Port(HANDLE *fd)
* \brief Se usa para crear un hilo que ejecuta la funcion <br>
* SERIAL_PORT_EVENT(HANDLE *hPort) <br>
* cuando se recibe un caracter por el puerto serie.
* \param fd Es el manejador del puerto.
* \return El manejador del hilo creado
* \ingroup HeaderLinux
*/
#ifdef ENABLE_SERIAL_PORT_EVENT
#include <pthread.h>
void *Thread_Port(void *hPort)
{
int n=0;
HANDLE *fd;
fd=(HANDLE *)hPort;

printf("SERIAL_PORT_EVENT [OK]\n");

do {
if(Kbhit_Port(*fd)!=0)
SERIAL_PORT_EVENT(fd);
} while(TRUE);
}
pthread_t Create_Thread_Port(HANDLE *fd)
{
```

```
 pthread_t idHilo;
 pthread_create (&idHilo, NULL, Thread_Port, fd);
 return idHilo;
 }
 #endif

#endif
```

Anexo II Inserte título del segundo anexo

Puede añadir tantos anexos como le sean necesarios.