CAPÍTULO 3

Software

3.1. Sistema operativo elegido: GNU/Linux.-

3.1.1. Una breve reseña histórica.-

El sistema operativo Linux procede del sistema operativo UNIX, al que toma como modelo. El sistema operativo UNIX se desarrolla a principios de la década de los 70 en los laboratorios de Bell Telephone, una subdivisión de AT&T.

En el año 1991 surge Linux. Debe su nombre a su creador, Linus Thorvalds. Se trata de un clon, escrito desde cero, del sistema operativo UNIX y una de sus características más importantes es que está *"soportado por comunidad"*, esto es, existe una comunidad de cientos de programadores repartidos por todo el mundo que se dedica desinteresadamente a desarrollar el sistema operativo Linux.

Muchas de las aplicaciones que se incluyen en las distintas distribuciones de Linux se encuentran dentro del proyecto GNU. Este proyecto se inicia en 1984, bajo el mando de Richard Stallman, con el objetivo de crear un sistema operativo avanzado, portable y totalmente libre, basado en UNIX.

En el año 1990 el sistema GNU estaba casi listo; sólo faltaba el núcleo. Se decidió desarrollar un núcleo para el sistema operativo llamado HURD, basado en Mach, un microkernel implementado en las Universidades de Carnegie-Mellon y Utah. Sin embargo, debido a múltiples imprevistos, el núcleo HURD se está retrasando y a día de hoy sigue sin estar completo. Afortunadamente, existe otro núcleo disponible: Linux. De la combinación del proyecto GNU y del núcleo Linux surge un sistema operativo totalmente libre: GNU/Linux.

3.1.2. Distribuciones.-

Una vez conseguido el sistema operativo Linux había que distribuirlo, hacerlo llegar al público. Aparecen varias empresas de software que se dedican a empaquetar un núcleo de Linux y un conjunto de utilidades compiladas para funcionar bajo el mismo, surgiendo de este modo las primeras *distribuciones* de Linux. En la actualidad hay un gran número de ellas, entre las que se debe citar:

Red Hat, SuSe, Mandrake, Slackware, Debian...

Las distribuciones se diferencian entre sí sobre todo en las utilidades desarrolladas por cada empresa para facilitar las tareas de configuración y administración del sistema. También existen leves diferencias en cuanto a estructura de directorios y distribución de archivos del sistema. Sin embargo todas comparten un punto en común: todas ejecutan un núcleo Linux, aunque las versiones de éste pueden variar.

3.1.3. ¿Por qué se ha elegido Linux?

Se ha elegido Linux por la fiabilidad, robustez y seguridad por la que es conocido. También resulta atractiva la idea de tener las fuentes del sistema operativo puesto que Linux se distribuye bajo la licencia GPL. Ésta es una licencia escrita por la fundación de Software Libre para evitar que se pongan restricciones a la distribución del software. La licencia consiste en que el

vendedor puede cobrar cuanto quiera por una copia, pero no se puede impedir al comprador que la distribuya gratuitamente una vez comprada. Por otro lado, como ya se apuntaba anteriormente, el código fuente tiene que estar disponible, lo que resulta muy útil para los programadores.

3.1.4. ***Debian GNU/Linux 3.0r1 (Woody)***.

De entre todas las distribuciones de linux que existen en la actualidad se ha elegido ***Debian GNU/Linux 3.0r1*** con el sobrenombre ***Woody***.

El punto fuerte de esta distribución es que se trata de una distribución soportada por una comunidad de programadores en vez de por una empresa: por lo tanto su objetivo no es el marketing ni la competencia económica sino el desarrollar la distribución de Linux más estable, fiable y segura de todas.

La mayor desventaja es la dificultad que las tareas de configuración y administración del sistema suponen al usuario debido a que no incluye herramientas que automaticen estos procesos como ocurre en otras distribuciones. Otra desventaja es la lentitud en las nuevas versiones y a la hora de incluir nuevos paquetes en la rama Woody o estable de debian: antes de incluir un paquete en la rama estable ha de pasar por las fases *inestable (**unstable**) y de pruebas (**testing**)*. Estas desventajas lo son hasta cierto punto pues la primera de ellas tiene como consecuencia un mayor control del proceso de instalación, configuración y administración del sistema por parte del usuario: se instala solo aquello que se quiera usar y se configura al gusto del usuario. En cuanto a la desventaja de la lentitud se puede decir que se pierde en velocidad pero se gana en estabilidad ya que los paquetes incluidos en la rama estable están probados hasta la saciedad.

Sin embargo el núcleo de Linux no está preparado para aplicaciones de tiempo real, o no lo estaba en el momento en que se inició este proyecto. Para proveer a Linux de capacidad de procesamiento en tiempo real se recurrió a la versión de ***Timesys Linux*** que se distribuye bajo licencia ***GPL*** en primer

lugar y, cuando resultó evidente que no satisfacía las exigencias de nuestra aplicación, se optó por utilizar la versión del núcleo de Linux más avanzada que existía en aquel momento: el ***núcleo 2.6.5.***

Antes de profundizar en el núcleo de Timesys y el 2.6.5 se verá que es exactamente un núcleo de Linux y cuales son sus funciones.

3.1.5 El núcleo de Linux.-

En cualquier sistema UNIX, existen varios procesos concurrentes atendiendo a distintas tareas. Cada uno de estos procesos necesita determinados recursos del sistema, ya sean procesador, memoria, recursos de red o cualquier otro. El núcleo es aquella entidad compuesta por código ejecutable que se encaraga de administrar todas las peticiones de recursos del sistema. Se trata de un programa que se arranca cuando se inicia el sistema operativo, se ejecuta en *"background"* o de fondo y se cierra cuando se apaga el ordenador.

Las tareas del núcleo son las siguientes:

- Manejo de procesos:
  El núcleo se encarga de la creación y destrucción de procesos, así como de la entrada y salida de datos a los mismos. También controla la comunicación entre procesos. Dentro del núcleo existe el llamado *planificador* que controla como los procesos comparten el procesador.

- Manejo de memoria.-
  El núcleo habilita un espacio de direccionamiento virtual para cada proceso en la parte superior de la memoria disponible.

- Sistema de ficheros.-
  En UNIX todos los dispositivos se tratan como ficheros de los que se puede leer y/o escribir. El núcleo habilita un sistema de ficheros

estructurado por encima del *hardware* sin estructurar.

Linux soporta gran variedad de sistemas de ficheros, esto es, diferentes maneras de organizar los datos en el medio físico como son *EXT2 y EXT3* para Linux, *FAT y VFAT* para windows, *etc.*

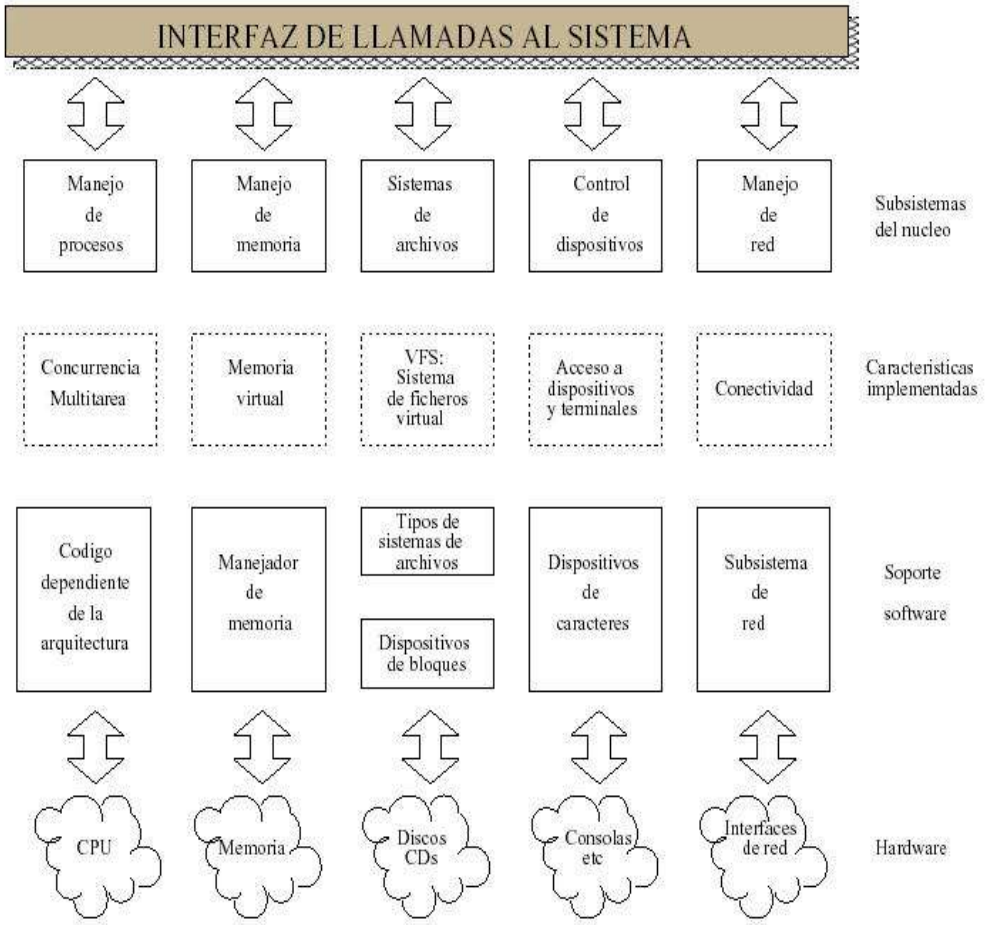
- Control de dispositivos.-

  Casi todas las operaciones del sistema requieren el acceso a algún dispositivo. A excepción del procesador, la memoria y alguna otra entidad más, las operaciones de control de dispositivos se llevan a cabo mediante un código que será específico del dispositivo. Este código se conoce como *driver* del dispositivo. El núcleo debe incluir un *driver* para cada periférico presente en el sistema.

- Manejo de red.- La red debe ser manejada por el núcleo del sistema operativo porque la mayoría de operaciones de red no son específicas de un proceso. Los paquetes entrantes son eventos asíncronos y, por lo tanto, deben ser recogidos, identificados y procesados antes de que lleguen a un proceso determinado. El núcleo está a cargo de manejar los paquetes de datos generados por los programas y por los interfaces de red, y debe controlar la ejecución de los programas en función de su actividad de red. Asi mismo, las tareas de enrutamiento y resolución de direcciones se implementan dentro del núcleo.

Una importante característica de Linux es la habilidad de extender en tiempo de ejecución las funcionalidades ofrecidas por el núcleo. Cada bloque de código que puede añadirse al núcleo en tiempo de ejecución se conoce como *"módulo"*. Cada módulo está formado por código objeto (no enlazado en un ejecutable) que puede enlazarse dinámicamente en el núcleo en ejecución gracias al programa *insmod* . Para desenlazarlo se usa el programa *rmmod.* Este sistema modular permite probar distintos drivers hasta encontrar el adecuado sin necesidad de *recompilar* todo el núcleo ni de reiniciar el equipo, lo que supone un ahorro considerable de tiempo.

En la siguiente figura se pueden apreciar las distintas partes del núcleo y las tareas que desempeñan:



A continuación se verán las mejoras que TimeSys Linux GPL o el núcleo 2.6.5 aportan para un sistema que debe funcionar en tiempo real. Se empezará con Timesys.

3.1.6. Timesys Linux GPL.

Timesys es una empresa que ha hecho modificaciones al núcleo de Linux

para adaptarlo a las necesidades de tiempo real. Cualquiera puede hacer modificaciones al núcleo de Linux e incluso distribuir sus propias modificaciones, siempre que el código se mantenga bajo el mismo copyright. Existen otras versiones de Timesys que no se distribuyen bajo licencia GPL y que, por tanto, son de pago. Es el caso de *Timesys RT Linux* que proporciona mejores capacidades de tiempo real con relojes y temporizadores de alta precisión y herencia de prioridades.

El núcleo de TimeSys Linux/GPL debe compilarse encima de un núcleo de linux 2.4 cuya versión sea superior a la 2.4.7. Esto se debe a que ***Timesys Linux/GPL*** utiliza las librerías de programación (APIs) disponibles en el núcleo de linux estándar.

Las mejoras que hace Timesys al núcleo de linux son las siguientes:

- Núcleo con expulsión(*preemptible kernel)*.- Esto significa que en cualquier momento un proceso puede bloquear a otro menos prioritario y tomar el control de la CPU.

- Interrupciones planificables y procesamiento de interrupciones software: en la salida del comando *ps* o Process status aparecen IRQ y softIRQ.

- Planificador de tiempo constante.

- Se reduce la latencia ,o tiempo transcurrido entre la ocurrencia de un evento y el momento en que la petición es finalmente servida, a la mínima cantidad posible.

- Tiempos de deshabilitación de interrupciones reducidos.

- Se añaden unos comandos para ajustar las prioridades de los procesos: *rtnice get* para obtener la política y la prioridad del proceso y *rtnice set* para ajustar la prioridad y la política al valor que nos interese. Las

políticas pueden ser de tiempo real como *fifo* y *rr* (round robin) o no como *other*. En FIFO y Round Robin se atiende a los procesos por orden de prioridad y para una misma prioridad se atiende antes a los procesos que antes lo pidieron; la diferencia consiste en que en FIFO un proceso toma el control hasta que se bloquea o hasta que lo interrumpe otro proceso más prioritario, mientras que en Round Robin un proceso no puede apropiarse de la CPU por más tiempo que un periodo determinado y terminado ese periodo debe desalojarla aunque no se haya bloqueado. Las prioridades de usuario en FIFO y Round Robin van de 1 a 99 mientras que en other van de -20 a 19.

- Los manejadores de interrupciones funcionan como hilos del núcleo, cada uno con su propia prioridad, que se puede modificar estática o dinámicamente, e individualmente. Por defecto tienen la prioridad máxima que es 509.

Se estuvo trabajando varios meses con el núcleo de Timesys hasta que llegó el momento de hacer las primeras pruebas del software desarrollado, momento en que surgió un problema insalvable: cuando se enviaban gran cantidad de caracteres a grandes velocidades desde el CBN hasta el CANa través del puerto serie, algunos de ellos se perdían. Se comprobó mediante el *Hiperterminal*© que el CBN realmente enviaba los caracteres. Por otro lado, en la línea no podía haber pérdida de caracteres. Se podía confundir algún bit con lo que se generaría un error de *checksum* pero no tenía sentido que se perdieran varios caracteres de vez en cuando.

Se concluyó que el problema tenía que estar en el CAN, ya sea en el software o en el hardware.

Al principio se pensó que se trataba de un problema de hardware y que los puertos serie no eran de mucha calidad. Tras investigar resultó que los puertos serie eran de los mejores que había: el puerto serie tenía una UART (Universal Asyncronus Receiver Transmitter) FIFO 16550A con un búffer de

16 bytes, mientras que las más antiguas sólo tienen un byte de FIFO. Por lo general la UART da una interrupción a la CPU cada vez que se recibe o se envía un byte. Entonces la CPU mueve el byte recibido desde el búffer de la UART a algún lugar de la memoria, o le da a la UART otro byte para enviar. En las UART antiguas es necesario interrumpir a la CPU cada vez que se envía o recibe un byte puesto que el búffer de la UART tiene un solo byte.

La UART del PC104 tiene un búffer de 16 bytes, lo que significa que puede recibir o transmitir hasta 16 bytes antes de interrumpir a la CPU y cuando la interrumpen, se transfieren todos los bytes de una sola vez. La CPU recibe menos interrupciones y está más libre para hacer otras cosas. Cuando se supera cierto umbral de caracteres en el búffer del puerto serie, por lo general 14 bytes, se envía la interrupción a la CPU. Si llegan más de dos caracteres en el tiempo que se tarda en atender la interrupción, los caracteres se sobrescriben y hay pérdida de datos.

Descartado el hardware como fuente del problema, sólo podía tratarse de un problema software. Se repasó bien el código de la aplicación pero ahí no estaba el fallo pues los caracteres sólo se perdían de vez en cuando. Se aumentó la prioridad del hilo de lectura de caracteres del puerto serie y se separó la lectura de caracteres y el procesamiento de los mismos en dos hilos distintos, pero esto no solucionó el problema.

Finalmente se descubrió que el problema estaba en que el núcleo de Timesys no estaba capacitado para proporcionarnos un tiempo real tan estricto: cuando la carga del sistema era alta, la CPU tardaba demasiado en atender las interrupciones de la UART y la FIFO interna de la UART se llenaba, con la consiguiente sobreescritura y pérdida de caracteres.

Una vez descubierta la causa de la pérdida de caracteres se barajaron dos opciones: migrar la aplicación al sistema operativo *QNX* o seguir utilizando Debian con el nuevo núcleo 2.6.

Se decidió seguir la segunda opción con resultados más que satisfactorios en todos los sentidos. El núcleo compilado es el ***2.6.5***.

3.1.7 El núcleo 2.6.5.-

El núcleo 2.6 ha convertido a Linux en un serio competidor para las compañías que desarrollan sistemas operativos de tiempo real, como VxWorks y WinCE. Además Linux se transforma en un sistema operativo excelente para ordenadores empotrados (*embedded*).

Entre las nuevas características que incorpora el núcleo 2.6 cabe destacar: capacidades mejoradas de tiempo real, fácil portabilidad a nuevos ordenadores, soporte para grandes cantidades de memoria, soporte para microcontroladores y un sistema mejorado de entrada/salida. Las características subrayadas son especialmente útiles para nuestra aplicación.

El sistema operativo en el que se ejecute nuestra aplicación debe funcionar de forma eficiente y fiable incluso en casos de extrema carga y se ha comprobado que TimeSys no cumple estos requisitos. Linux 2.6 es una mejor opción a este respecto.

Aunque Linux 2.6 no es todavía un verdadero sistema operativo en tiempo real, contiene mejoras que lo convierten en una buena plataforma cuando se requiere una respuesta rápida. Las características que hacen de Linux 2.6 una buena aproximación a un sistema operativo en tiempo real son las siguientes:

- Núcleo con desalojo *(kernel preemption*).-

Como ocurre con la mayoría de sistemas operativos de propósito general, Linux siempre ha prohibido que el planificador de procesos se ejecute cuando un proceso está haciendo una llamada al sistema. Por lo tanto, una vez una tarea se encuentra en una llamada al sistema, esa tarea controla el

procesador hasta que termina la llamada al sistema, sin importar el tiempo que esto pueda llevar. En lo que respecta al núcleo 2.6, se puede decir que se trata de un núcleo con desalojo. Ahora una tarea del núcleo puede ser desalojada para que una importante aplicación de usuario pueda continuar ejecutándose. En Linux 2.6 el código del núcleo ha sido dotado de *puntos de desalojo (preemption points*) , instrucciones que permiten al planificador ejecutarse y bloquear al proceso actual para dar paso a un proceso más prioritario. Linux 2.6 evita retardos excesivos en las llamadas al sistema mediante el chequeo periódico de un punto de desalojo. Durante estos chequeos, el proceso que hizo la llamada al sistema puede bloquearse y dejar que se ejecute otro proceso. Por lo tanto, bajo Linux 2.6 el kernel puede ser interrumpido a mitad de tarea para que otras aplicaciones puedan continuar su ejecución incluso si algo de bajo nivel y complicado se está ejecutando de fondo *(background)*.

- Un planificador eficiente.-

El planificador de procesos ha sido reescrito en el núcleo 2.6 para eliminar los lentos algoritmos de versiones previas. Antiguamente, para decidir que tarea debía ejecutarse a continuación, el planificador debía comprobar cada tarea que estaba preparada para ejecutarse y hacer un cálculo para determinar la importancia relativa de esa tarea. Después de hacer todos esos cálculos se elegía la tarea con la mayor puntuación. A causa de que el tiempo requerido para este algoritmo dependía directamente del número de tareas, las aplicaciones multitarea complejas tenían una planificación lenta.

El planificador de Linux 2.6 no comprueba todas las tareas antes de elegir aquella que va a tomar control de la CPU. En vez de eso, cuando una tarea pasa a estar preparada para ejecutarse, se la introduce en cierta posición de una cola determinada llamada *cola actual (current queue).* Entonces, cuando se ejecuta el planificador, elige la tarea que se encuentre en la posición más favorable en la cola. Como resultado de todo esto, la

planificación se realiza en una cantidad de tiempo constante. Cuando la tarea está ejecutándose, se le da un cuanto de tiempo, o un periodo de tiempo en el que puede ejecutarse antes de dar paso a otro hilo. Cuando su cuanto de tiempo ha expirado, la tarea se pasa a otra cola llamada la *cola expirada (expired queue).* La tarea toma posición en esta cola expirada de acuerdo con su prioridad. Este nuevo procedimiento es sustancialmente más rápido que el antiguo y trabaja igual de bien independientemente del número de tareas que haya en cola.

- Nuevas primitivas para sincronización.-

Las aplicaciones que involucren el uso de recursos compartidos como memoria compartida o dispositivos compartidos, tienen que ser desarrolladas cuidadosamente para evitar condiciones de carrera. La solución implementada en Linux, llamada Mútex, aseguraba que solamente una tarea estaba usando el recurso en un momento determinado. El Mútex provocaba una llamada al sistema para decidir si bloquear el hilo o dejar que continúe ejecutándose. Pero cuando la decisión era continuar, la innecesaria llamada al sistema consumía tiempo de ejecución y esto no era eficiente. La nueva implementación de Linux 2.6 soporta Mutexes rápidos en espacio de usuario o *Futex (Fast User-Space Mutex)* . Estas funciones pueden chequear desde el espacio de usuario si el bloqueo es necesario y efectuar la llamada al sistema para bloquear al hilo sólo cuando sea necesario. Si no se requiere el bloqueo se evita una llamada al sistema innecesaria y se ahorra tiempo. También se permite el establecimiento de prioridades para permitir a las aplicaciones o hilos de mayor prioridad acceder primero al recurso por el que se contiende.

- Compatibilidad con la norma POSIX y mejoras en el modelo de hilos.- LinuxThread, la actual librería de hilos en Linux, es mala. El nuevo modelo de hilos del núcleo 2.6 está basado en un modelo de hilos 1:1, esto es, un hilo en el núcleo por cada hilo de usuario. Además se incluye soporte en el núcleo para la nueva *librería de hilos de POSIX nativo (NPTL o Native*

Posix Thread Library). La infraestructura interna del núcleo ha sido reescrita para permitir que NPTL se ejecute sobre ella.

Para hacernos una idea de cuanto ha mejorado el modelo de hilos se puede decir que Linux, con su nuevo modelo, es capaz de iniciar y terminar 100.000 hilos simultáneamente en dos segundos. Hacer esto con el antiguo modelo se requerían 15 minutos.

Además de hilos POSIX, el núcleo 2.6 proporciona señales POSIX y temporizadores de alta resolución POSIX como parte del núcleo. Las señales POSIX son una mejora sobre las señales UNIX que existían en versiones anteriores del núcleo. Al contrario de las señales UNIX, las señales POSIX no se pueden perder y pueden llevar información como argumento. Además, las señales POSIX pueden ser enviadas de un hilo a otro, en vez de sólo de proceso a proceso como ocurre en las señales UNIX. Los temporizadores POSIX tienen una mayor precisión y permiten al programador planificar determinadas tareas para que se ejecuten periódicamente. El problema es que estos temporizadores de alta resolución todavía no están disponibles para que los utilice el usuario en sus aplicaciones. Sin embargo se está desarrollando una librería para estos temporizadores llamada *hrtimers* que estará disponible en breve.

3.2 Lenguaje de programación empleado: C++.-

El lenguaje de programación empleado es C++ por dos razones principalmente:

la primera es que facilita mucho la tarea del programador cuando el código de la aplicación es muy extenso como ocurre en el caso de esta aplicación; la segunda es la reutilización del código desarrollado para el Romeo 4R(*), del que se ha aprovechado alguna clase como por ejemplo la que controla el puerto serie.

El lenguaje de programación C++ es un lenguaje orientado a objetos. La forma de programar a ido evolucionando con el paso del tiempo. En los albores de la informática se utilizaba el lenguaje máquina que resultaba muy trabajoso. Para facilitar las cosas se inventó el ensamblador que utilizaba *mnemónicos*, un conjunto de instrucciones sencillas y fáciles de recordar. Posteriormente aparecieron los lenguajes de alto nivel como el BASIC o el FORTRAN que incluían instrucciones más elaboradas, específicas y fáciles de usar. El siguiente paso fue la programación estructurada con lenguajes como C o Pascal que permitían crear programas bastante complejos. Sin embargo, cuando se superaba cierto número de líneas de código resultaba muy complejo para el programador la gestión del proyecto. Para facilitar la gestión de proyectos grandes surgen los lenguajes orientados a objetos que facilitan la modularidad y reutilización de código.

---

* Romeo 4R es una plataforma robótica de investigación basada en un vehículo elétrico convencional. Ha sido desarrollado por el grupo de Visión, Robótica y Control en el Departamento de Ingeniería de Sistemas y Automática de la Universidad de Sevilla. Para más información léase el PFC de Rafael Martín de Agar Tirado que se encuentra en la bibliografía.

Todos los lenguajes de programación tienen tres características en común: encapsulación, herencia y polimorfismo:

- Encapsulación.-

  Es el mecanismo que relaciona el código y los datos, a la vez que los protege de interferencias y fallos externos. La entidad que encapsula el código y los datos se denomina *objeto*. Los datos encapsulados dentro de un objeto se denominan *atributos* y el código del objeto son los *métodos*. Los atributos no son más que variables encapsuladas en un objeto y los métodos son funciones también encapsuladas en dicho objeto. Parte de los atributos y métodos pueden declararse *privados* al objeto de forma que no se pueda acceder a ellos desde fuera. Este es el modo en que se protege al objeto de interferencias y fallos ocurridos en otras partes del programa. Así, por ejemplo, si se sale de los límites de una tabla no sería posible sobreescribir la zona de memoria destinada al objeto, porque es privada.

  Antes de utilizar un objeto, hay que definir una *clase*. Una clase es un nuevo tipo de dato definido por el usuario y no tiene entidad física: solo contiene la definición de ciertos atributos y métodos. El objeto sí tiene entidad física: es una *instancia* de esa clase. El concepto de objeto es parecido al de estructura, típico de C, pero con las ventajas de que se puede encapsular código además de datos y que métodos y atributos se pueden declarar privados.

- Polimorfismo.-

  El polimorfismo consiste en que se puede utilizar la misma interfaz para varios métodos distintos, es decir, pueden existir varios métodos con el mismo nombre y que se diferencien en los parámetros que se le pasen.

Según los parámetros que se usen para llamar al método, se utilizará uno u otro de los que tienen el mismo nombre.

- Herencia.-

  Característica que permite a una clase adquirir los atributos y métodos de otra. Esto facilita la reutilización de código y se ahorra tiempo. Además el programa se estructura de una forma mucho más jerárquica.

La característica de la programación orientada a objetos que más se ha usado de las citadas es la encapsulación: las variables y funciones que estaban relacionadas se han encapsulado en una misma clase. Cada clase se encuentra en un fichero independiente para facilitar la modularidad. Las características de polimorfismo y herencia no han resultado útiles para esta aplicación.

Los atributos y los métodos serán privados en la medida de lo posible. Solamente se declararan como públicos aquellos métodos que han sido desarrollados para el uso del usuario.

3.3. Descripción del funcionamiento orientada a usuario.-

3.3.1. Modos de funcionamiento.-

El software que conforma este proyecto consta de un servidor que se ejecuta a bordo del CAN y de clientes que se ejecutarán en el ordenador de tierra (PST) y/o en el ordenador de percepción y supervisión (PPSE). Existen dos modos de funcionamiento claramente diferenciados:

- El primer modo de funcionamiento se denominará *"modo puente"* y en este modo el PC104 actuará como un *puente* entre el ordenador de tierra (PST) y el CBN: recogerá cadenas de caracteres a través de ethernet y las enviará de forma transparente hacia el CBN a través del puerto serie. No se realiza ningún procesamiento en el ordenador de a bordo, es decir, su única misión es la de proveer de comunicaciones ethernet al CBN. Se utilizará este modo de funcionamiento en la identificación del modelo del helicóptero. El bucle de control se ejecutará en el cliente de tierra y no existe un cliente en el ordenador de percepción y supervisión (PPSE).

- El segundo modo de funcionamiento se denomina *"modo procesamiento a bordo"* y consiste en que el bucle de control se ejecuta en el ordenador de a bordo. Los clientes del ordenador de tierra (PST) y del PPSE tendrán una labor de supervisión. Para facilitar esta tarea se proporcionan al *usuario* una serie de métodos para el envío y recepción de cadenas de caracteres cualesquiera. El usuario es el investigador que desarrollará algoritmos de control de alto nivel que se ejecutarán en el PC de control. Para efectuar tareas de supervisión, el usuario deberá generar un protocolo para encapsular los datos de supervisión en cadenas y para interpretarlos correctamente en el otro extremo de la comunicación. El formateado de las cadenas es labor del usuario ya que sólo se le proporcionan métodos para el envío y recepción de dichas cadenas.

A continuación se describen los métodos de interés para el usuario clasificados por el programa en el que se hallan: servidor, cliente en *modo puente* y cliente en modo procesamiento a bordo:

3.3.2. Métodos de interés para el usuario.-

3.3.2.1.Servidor.-

El servidor tiene dos modos de funcionamiento seleccionables en el fichero de configuración “can.conf”. Sin embargo, en el servidor, solo el modo de funcionamiento con procesamiento a bordo proporciona métodos al usuario. En el *modo puente* los datos que llegan desde la red se desvían hacia el puerto serie de forma transparente al usuario y viceversa.

3.3.2.1.1. Métodos destinados al usuario para la comunicación con el CBN a través del puerto serie.-

En este apartado se recogen los métodos que utilizará el usuario para comunicarse con el CBN. Son métodos para obtener datos que serán necesarios en el algoritmo de control y para aplicar las consignas que resulten de éste.

- `int Control::envia_datos_camino_xyz (double x, double y , double z)`:

Descripción:

Este método envía al CBN un punto en coordenadas cartesianas.

Parámetros:

Los parámetros que se le pasan a este método son tres números flotantes de doble precisión correspondientes a las coordenadas x, y, z.

Valor devuelto:

0 si todo va bien y -1 si se desborda la cola de puntos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `int Control::envia_datos_camino_geo(double latitud, double longitud, double altura)`:

  Descripción:

  Este método envía al CBN un punto en coordenadas geográficas.

  Parámetros:

  Los parámetros que se le pasan a este método son tres números flotantes de doble precisión correspondientes a latitud, longitud y altura. Para la latitud y la longitud se ha definido un código de signos que nos indica el cuadrante en el se halla el helicóptero:

  - ✔ latitud:

    Si es positiva es norte y si es negativa es sur.

  - ✔ longitud:

    Si es positiva es este y si es negativa es oeste.

  Valor devuelto:

  0 si todo va bien y -1 si se desborda la cola de puntos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `int Control::leer_de_puerto_16(int puerto, short int *puntero)`:

  Descripción:

  Este método lee un valor de 16 bits del puerto del CBN que se le indique. Ha sido dotado de un “timeout” para que no se quede eternamente esperando una lectura.

Parámetros:

Los parámetros que se le pasan son un número entero correspondiente al número del puerto y un puntero a short int para pasarle por referencia la dirección del short int en que se quiere recoger el resultado.

Valor devuelto:

0 si todo va bien, -1 si se desborda la cola de comandos (en cuyo caso se escribe un código de error en fichero y en la cola de errores) y -2 si expira el "timeout" sin haber podido leer el dato (se produce un código de error distinto que recibe el mismo tratamiento que en el caso del -1).

- `int Control::leer_de_puerto_32(int puerto, float *puntero)`:

Descripción:

Este método lee un valor de 32 bits del puerto del CBN que se le indique. Ha sido dotado de un "timeout" para que no se quede eternamente esperando una lectura.

Parámetros:

Los parámetros que se le pasan son un número entero correspondiente al número del puerto y un puntero a flotante de precisión simple para pasarle por referencia la dirección del número en coma flotante en que se quiere recoger el resultado.

Valor devuelto:

0 si todo va bien, -1 si se desborda la cola de comandos (en cuyo caso se

escribe un código de error en fichero y en la cola de errores) y -2 si expira el "timeout" sin haber podido leer el dato (se produce un código de error distinto que recibe el mismo tratamiento que en el caso de que el valor devuelto sea -1).

- `int Control::escribir_en_puerto_16(int puerto, short int valor)`:

  Descripción:

  Este método escribe un valor de 16 bits en el puerto del CBN que se le indique. Sirve de base para otras funciones de más alto nivel ya que dependiendo del puerto en que se escriba se tendrán distintas funcionalidades.

  Parámetros:

  Los parámetros que se le pasan a este método son el número de puerto en el que se quiere escribir y el valor de 16 bits (short int) que se quiere escribir en ese puerto.

  Valor devuelto:

  0 si todo va bien y -1 si se desborda la cola de comandos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `int Control::escribir_en_puerto_32(int puerto, float valor)`:

  Descripción:

  Este método escribe un valor de 32 bits en el puerto del CBN que se le indique. Sirve de base para otras funciones de más alto nivel ya que dependiendo del puerto en que se escriba se tendrán distintas funcionalidades.

Parámetros:

Los parámetros que se le pasan a este método son el número de puerto en el que se quiere escribir y el valor de 32 bits (float) que se quiere escribir en ese puerto.

Valor devuelto:

0 si todo va bien y -1 si se desborda la cola de comandos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `void Control::activar_tx_de_estado():`

Descripción:

Este método ordena al CBN que envíe el estado del helicóptero cada periodo de muestreo, que valdrá normalmente 20 ms. Por defecto se envía el estado completo pero se puede configurar para obtener únicamente aquéllos campos del estado que nos interesen de acuerdo con una máscara de bits que será preciso facilitar al CBN con la función “envia_mascara”.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::envia_mascara(int mascara):`

Descripción:

Este método llama a la función “escribir_en_puerto_16” para escribir la máscara en un puerto determinado del CBN (el 10000). La máscara es un patrón binario que indica cuales de las variables del estado del helicoptero (que mantiene el CBN) deben ser transmitidas hacia el CAN. En el CBN se sobreescribe la máscara por defecto, que es 0xffff, con el valor que le se ha pasado y a partir de este momento solo se recibirán en el CAN aquellos campos del estado del helicóptero que estén incluidos en la máscara.

Parámetros:

Un entero que corresponde a la nueva máscara.

Valor devuelto:

No tiene.

- `void Control::desactivar_tx_estado():`

Descripción:

Este método da una orden al CBN para que deje de enviar el estado del helicóptero.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::limpiar_codigo_de_error():`

Descripción:

Este método se utiliza para limpiar las colas de errores del CBN. Se ignoran todos los errores que se hayan producido hasta este momento. Debe incluirse en la inicialización del bucle de control.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::invalidar_camino():`

Descripción:

Este método se usa para invalidar el camino que tenía almacenado el CBN. Los puntos del camino ("waypoints") que el CBN tenía almacenados son desechados. Estos puntos son los que se envían con los métodos *envia_datos_camino_xyz* y *envia_datos_camino_geo*. Se debe incluir una llamada a esta función en la inicialización del bucle de control.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::activar_seguimiento():`

Descripción:

Este método se usa para activar el seguimiento de trayectorias por parte del CBN. Cuando el CBN recibe el comando correspondiente a este método, empieza a seguir secuencialmente los puntos del camino que tenía almacenados.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::desactivar_seguimiento():`

Descripción:

Este método se usa para desactivar el seguimiento de trayectorias por parte del CBN. Cuando el CBN recibe el comando correspondiente a este método deja de seguir los puntos del camino que tenía almacenados.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::habilitar_coordenadas_geograficas():`

Descripción:

Este método hace una llamada a “escribir_en_puerto_16” para escribir un 1 en un puerto determinado del CBN (el 10001). De este modo se indica al CBN que los puntos del camino están en coordenadas geográficas y no en cartesianas (valor por defecto).

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::habilitar_coordenadas_cartesianas():`

Descripción:

Este método hace una llamada a “escribir_en_puerto_16” para escribir un 0 en un puerto determinado del CBN (el 10001). De este modo se indica al CBN que los puntos del camino están en coordenadas cartesianas (valor por defecto) y no en geográficas .

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::activar_aceleraciones_y_velocidades():`

Descripción:

Este método hace una llamada a “escribir_en_puerto_16” para escribir un 1 en un puerto determinado del CBN (el 10002). De este modo se indica al CBN que debe recoger ángulos de Euler, aceleraciones y velocidades de la IMU. Por defecto solo se recogen los ángulos de Euler de la IMU de modo que sea posible tener un dato del estado del helicóptero cada 20 ms. Si se recoge todo no es posible tener un dato del estado del helicóptero cada 20 ms.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::desactivar_aceleraciones_y_velocidades():`

Descripción:

Este método indica al CBN que recoja únicamente los ángulos de Euler de la IMU. Éste es el comportamiento por defecto. Internamente, y de forma transparente al usuario, se hace una llamada a “escribir_en_puerto_16” para escribir un 0 en un puerto determinado del CBN (el 10002).

Parámetros:

No tiene.

Valor devuelto:

No tiene.

3.3.2.1.2. Métodos destinados al usuario para la comunicación con el ordenador de tierra (PST) o el ordenador de percepción y supervisión (PPSE) a través de ethernet.-

En este apartado se recogen métodos para el envío y recepción de cadenas cualesquiera a través de ethernet. La interpretación de las cadenas es responsabilidad del usuario. El protocolo proporciona conexiones punto a punto con tierra y con el PPSE y métodos para el envío y recepción de cadenas de caracteres.

- `int Control::leer_cadena_usr_tierra(char *cadena):`

Descripción:

Este método recoge una cadena enviada desde tierra a través de ethernet y la mete en la tabla "cadena".

Parámetros:

La cadena en la que se quiere recoger la lectura.

Valor devuelto:

0 si todo va bien y -1 si no había nada que leer.

- `int Control::leer_cadena_usr_ppse(char *cadena):`

Descripción:

Este método recoge una cadena enviada desde PPSE a través de ethernet y la mete en la tabla "cadena".

Parámetros:

La cadena en la que se quiere recoger la lectura.

Valor devuelto:

0 si todo va bien y -1 si no había nada que leer.

- `int Control::escribir_cadena_usr_tierra_y_esperar_respuesta (char *cadena_enviada,char *cadena_recibida):`

Descripción:

Este método envía una cadena hacia tierra y se queda bloqueado esperando una respuesta.

Parámetros:

La cadena que se envía y aquella en la que se quiere recoger la lectura.

Valor devuelto:

0 si todo va bien.

- `int Control::escribir_cadena_usr_ppse_y_esperar_respuesta (char *cadena_enviada,char *cadena_recibida):`

Descripción:

Este método envía una cadena hacia PPSE y se queda bloqueado esperando una respuesta.

Parámetros:

La cadena que se envía y aquella en la que se quiere recoger la lectura.

Valor devuelto:

0 si todo va bien.

- `int Control::escribir_cadena_usr_tierra_y_no_esperar_respuesta (char *cadena_enviada):`

  Descripción:

  Este método envía una cadena hacia tierra.

  Parámetros:

  La cadena que se envía y aquella en la que se quiere recoger la lectura.

  Valor devuelto:

  0 si todo va bien y -1 si no había nada que leer.

- `int Control::escribir_cadena_usr_ppse_y_no_esperar_respuesta (char *cadena_enviada):`

  Descripción:

  Este método envía una cadena hacia PPSE .

  Parámetros:

  La cadena que se envía y aquella en la que se quiere recoger la lectura.

  Valor devuelto:

0 si todo va bien y -1 si no había nada que leer.

3.3.2.1.3. Métodos destinados al usuario para la adquisición de datos sobre el estado del helicóptero y supervisión de errores.-

En este apartado se recogen algunos métodos públicos de la clase “Estimated” para la obtención de datos sobre el estado del helicóptero y para sacar errores de una cola. Se guardan las últimas estructuras de estado y se ponen disponibles al usuario para que las use en sus algoritmos de control. El número de estados que se guarda es configurable con la macro TAM_TABLA_ESTADOS.

- `void Estimated::actualizar(Control *p_control);`

Descripción:

Este método saca las últimas respuestas recibidas del CBN. Si se recibe un estado del helicóptero se envía a tierra sólo si así se especificó en el fichero de configuración *"can.conf"* y se guarda en una tabla circular siempre. Si se reciben errores se envían a tierra si así lo determina el fichero de configuración, se actualizan ciertas banderas y se meten los errores en una cola destinada a tal propósito. El usuario puede sacar los errores de la cola en el bucle de control y actuar en consecuencia. Este método debe utilizarse una vez en cada pasada del bucle de control, antes de tomar ninguna decisión. Los datos obtenidos sobre errores y estado del helicóptero se utilizarán para tomar decisiones en el bucle de control.

Parámetros:

Un puntero a un objeto “Control” para tener acceso a las colas y métodos del objeto que se le pase por referencia.

Valor devuelto:

No tiene.

- `bool Estimated::errores();`

Descripción:

Esta función devuelve verdadero si se recogieron errores procedentes del PC104 o del CBN en la última actualización.

Parámetros:

No tiene

Valor devuelto:

true (verdadero) si hubo errores y false (falso) si no los hubo.

- `bool Estimated::errores_cbn();`

Descripción:

Esta función devuelve verdadero si se recogieron errores procedentes del CBN en la última actualización.

Parámetros:

No tiene

Valor devuelto:

true (verdadero) si hubo errores y false (falso) si no los hubo.

- `bool Estimated::errores_can();`

  Descripción:

  Esta función devuelve verdadero si se recogieron errores procedentes del PC104 en la última actualización.

  Parámetros:

  No tiene.

  Valor devuelto:

  true (verdadero) si hubo errores y false (falso) si no los hubo.

- `bool Estimated::nuevo_estado();`

  Descripción:

  Esta función devuelve verdadero si se recogió algún estado nuevo en la última actualización.

  Parámetros:

  No tiene.

  Valor devuelto:

  true (verdadero) si se recogió un nuevo estado y false (falso) si no.

- `COORDENADAS_GEO Estimated::obtener_coordenadas_geo(int numestado);`

```
typedef struct{
  float latitud;   // + = norte ; - = sur
  float longitud;  // + = este  ; - = oeste
  float altura_gps;} COORDENADAS_GEO;
```

Descripción:

Esta función recoge una estructura de COORDENADAS_GEO del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura COORDENADAS_GEO .

Valor devuelto:

Una estructura COORDENADAS_GEO que contiene las coordenadas geográficas pedidas: latitud, longitud y altura medida por el GPS.

- `COORDENADAS_XY Estimated::obtener_coordenadas_xy(int numestado);`

```
typedef struct{
        float x;
        float y;} COORDENADAS_XY;
```

Descripción:

Esta función recoge una estructura de COORDENADAS_XY del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura COORDENADAS_XY .

Valor devuelto:

Una estructura COORDENADAS_XY que contiene las coordenadas cartesianas pedidas.

- `bool Estimated::obtener_estado_del_switch_automático_manual(int numestado);`

Descripción:

Esta función recoge el valor del switch automático manual del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo y lo devuelve como variable booleana.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger el estado del switch automático-manual.

Valor devuelto:

Un valor booleano true (verdadero) si el switch está en la posición de automático y false (falso) si está en manual.

- `GPS_QOS Estimated::obtener_gps_qos(int numestado);`

```
typedef struct{
          char numero_de_satelites;
          char rt20_status;
          char sol_status;
             } GPS_QOS;
```

Descripción:

Esta función recoge una estructura GPS_QOS relativa a la calidad del servicio del GPS del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura GPS_QOS.

Valor devuelto:

Una estructura GPS_QOS que contiene la calidad de servicio del GPS.

- `short int Estimated::obtener_medida_sonar(int numestado);`

Descripción:

Esta función recoge un número entero de 16 bits (short int) correspondiente a la medida del sonar del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la medida del sonar.

Valor devuelto:

Un entero de 16 bits que contiene la medida del sonar.

- `short int Estimated::obtener_nivel_baterias(int numestado);`

Descripción:

Esta función recoge un número entero de 16 bits (short int) correspondiente al nivel de las baterías del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger el nivel de las baterías.

Valor devuelto:

Un entero de 16 bits que contiene el nivel de las baterías.

- `char Estimated::obtener_nivel_combustible(int numestado);`

Descripción:

Esta función recoge un carácter de 8 bits correspondiente al nivel del combustible del estado que se indica como parámetro siendo 0 el estado más

actual y (TAM_TABLA_ESTADOS -1) el más antiguo. Su contenido está aún por definir.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger el nivel del combustible.

Valor devuelto:

Un carácter de 8 bits que corresponde al nivel del combustible.

- `ORIENTACION Estimated::obtener_orientacion(int numestado);`

```
typedef struct{
  float pitch;
  float roll;
  float yaw;} ORIENTACION;
```

Descripción:

Esta función recoge una estructura ORIENTACION que contiene pitch, roll y yaw medidos por la IMU del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura ORIENTACION.

Valor devuelto:

Una estructura ORIENTACION que contiene pitch, roll y yaw.

- `ACELERACIONES Estimated::obtener_aceleraciones(int numestado);`

```
typedef struct{
          float aceleracion_x;
          float aceleracion_y;
          float aceleracion_z;} ACELERACIONES;
```

Descripción:

Esta función recoge una estructura ACELERACIONES que contiene tres números flotantes de precisión simple correspondientes a las aceleraciones lineales del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura ACELERACIONES.

Valor devuelto:

Una estructura ACELERACIONES que contiene tres números reales de precisión simple correspondientes a las aceleraciones lineales en cada uno de los ejes cartesianos.

- `PAN_TILT Estimated::obtener_pan_tilt(int numestado);`

```
typedef struct{
      int  pan;
      int tilt;} PAN_TILT;
```

Descripción:

Esta función recoge una estructura PAN_TILT que contiene un entero correspondiente a la medida del ángulo relativo al pan y otro entero correspondiente a la medida del ángulo relativo al tilt del estado que se indica

como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura PAN_TILT.

Valor devuelto:

Una estructura PAN_TILT que contiene dos enteros: el ángulo del pan y el del tilt.

- `PWM Estimated::obtener_pwm(int numestado);`

```
typedef struct{
      short int pwm_medidos[7];} PWM;
```

Descripción:

Esta función recoge una estructura pwm que contiene una tabla de siete enteros de 16 bits correspondientes a las medidas de los PWM del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura PWM.

Valor devuelto:

Una estructura PWM que contiene las medidas de los PWM como enteros de 16 bits.

- `float Estimated::obtener_rpm_rotor(int numestado);`

  Descripción:

  Esta función recoge un número en coma flotante de precisión simple que corresponde a las revoluciones por minuto del rotor del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

  Parámetros:

  Un entero que corresponde al número de estado del que se quiere recoger las revoluciones por minuto del rotor.

  Valor devuelto:

  Un número flotante de precisión simple que contiene las revoluciones por minuto del rotor.

- `unsigned int Estimated::obtener_tiempo_local_cbn(int numestado);`

  Descripción:

  Esta función recoge un valor de tiempo medido en milisegundos por el CBN desde su arranque correspondiente al estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

  Parámetros:

  Un entero que corresponde al número de estado del que se quiere recoger el valor del tiempo local al CBN.

Valor devuelto:

Un entero sin signo que contiene el valor del tiempo leído por el CBN en milisegundos.

- `VELOCIDADES_ANGULARES`
  `Estimated::obtener_velocidades_angulares(int numestado);`

```
typedef struct{
          float velocidad_angular_x;
          float velocidad_angular_y;
          float velocidad_angular_z;
        } VELOCIDADES_ANGULARES;
```

Descripción:

Esta función recoge las velocidades angulares medidas por el CBN del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger las velocidades angulares.

Valor devuelto:

Una estructura VELOCIDADES_ANGULARES que contiene las velocidades angulares leídas por el CBN.

- `VELOCIDADES_Y_HEADING`
  `Estimated::obtener_velocidades_y_heading(int numestado);`

```
typedef struct{
  float direccion_con_respecto_al_norte;
  float velocidad_horizontal;
```

```
float velocidad_vertical;} VELOCIDADES_Y_HEADING;
```

Descripción:

Esta función recoge la dirección con respecto al norte, velocidad horizontal y velocidad vertical del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger velocidades y heading.

Valor devuelto:

Una estructura VELOCIDADES_Y_HEADING que contiene tres números reales en punto flotante de precisión simple que corresponden a la dirección con respecto al norte, velocidad horizontal y velocidad vertical.

- `int Estimated::sacar_de_cola_errores(ERRORES *p_dato);`

Descripción:

Esta función saca un error de una cola circular de errores que se encuentra dentro del objeto Estimated. El error se recoge por referencia. Sería aconsejable para el usuariosacar todos los errores de esta cola al comienzo de cada periodo del bucle de control para tener toda la información sobre los errores que se han producido y actuar en consecuencia.

Parámetros:

Un puntero a una estructura ERRORES para recoger por referencia el dato del error.

Valor devuelto:

0 si se consiguió sacar el error correctamente y -1 si no había ningún error en la cola.

3.3.2.2. Cliente con procesamiento a bordo.-

En el cliente con procesamiento a bordo se proporcionan únicamente dos métodos de interés para el usuario: envío y recepción de cadenas. Se proporciona información acerca de si el servidor se encuentra bloqueado en espera de una respuesta por parte del cliente y la también se proporciona la posibilidad de desbloquear al servidor en este caso.

- `int Procesamiento::enviar_cadena(char *cadena, bool desbloquear_can);`

Descripción:

Esta función envía una cadena cualquiera al CAN y tiene la posibilidad de desbloquearlo.

Parámetros:

La cadena enviada y una variable booleana que indica si se quiere desbloquear el CAN (si vale true queremos desbloquear al CAN y si vale false no).

Valor devuelto:

Devuelve un entero que corresponde al número de caracteres que se han enviado. Si se produce un error el valor devuelto será -1 y si se rompe la conexión de red se devuelve 0.

- `int Procesamiento::recibir_cadena(char *cadena, bool * p_desbloquear);`

Descripción:

Esta función comprueba si ha llegado una cadena cualquiera procedente del CAN y en tal caso recoge dicha cadena.También recoge por referencia una variable booleana que nos indica si el CAN está bloqueado en espera de una respuesta por parte del cliente. Esta función no se bloquea, si no que realiza un sondeo.

Parámetros:

La cadena recibida y una variable booleana que indica si el CAN está bloqueado en espera de una respuesta por parte del cliente..

Valor devuelto:

Devuelve un entero que corresponde al número de caracteres que se han recibido. Si se produce un error el valor devuelto será -1 y si se rompe la conexión de red se devuelve 0.

3.3.2.3.Cliente con modo puente.-

Los métodos que se proporcionan al usuario en el cliente en *modo puente* son los mismos que se proporcionan en el servidor en modo procesamiento. En el *modo puente* el procesamiento se realiza en el cliente. Para el usuario es transparente programar el bucle de control en el cliente en *modo puente* o en el servidor de a bordo en modo procesamiento. Por eso los métodos son los mismos.

3.3.2.3.1. Métodos destinados al usuario para la comunicación con el CBN a través de ethernet y del puerto serie.-

En este apartado se recogen los métodos que utilizará el usuario para comunicarse con el CBN.

- `int Puente::envia_datos_camino_xyz (double x, double y , double z)`:

  Descripción:

  Este método envía al CBN un punto en coordenadas cartesianas.

  Parámetros:

  Los parámetros que se le pasan a este método son tres números flotantes de doble precisión correspondientes a las coordenadas x, y, z.

  Valor devuelto:

  0 si todo va bien y -1 si se desborda la cola de puntos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `int Puente::envia_datos_camino_geo(double latitud, double longitud, double altura)`:

  Descripción:

  Este método envía al CBN un punto en coordenadas geográficas.

  Parámetros:

Los parámetros que se le pasan a este método son tres números flotantes de doble precisión correspondientes a latitud, longitud y altura. Para la latitud y la longitud se ha definido un código de signos que nos indica el cuadrante en el que se encuentra el helicóptero:

- ✔ latitud:

  Si es positiva es norte y si es negativa es sur.

- ✔ longitud:

  Si es positiva es este y si es negativa es oeste.

Valor devuelto:

0 si todo va bien y -1 si se desborda la cola de puntos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `int Puente::leer_de_puerto_16(int puerto, short int *puntero)`:

Descripción:

Este método lee un valor de 16 bits del puerto del CBN que se le indique. Ha sido dotado de un “timeout” para que no se quede eternamente esperando una lectura.

Parámetros:

Los parámetros que se le pasan son un número entero correspondiente al número del puerto y un puntero a short int para pasarle por referencia la dirección del short int en que se quiere recoger el resultado.

Valor devuelto:

0 si todo va bien, -1 si se desborda la cola de comandos (en cuyo caso se escribe un código de error en fichero y en la cola de errores) y -2 si expira el

“timeout” sin haber podido leer el dato (se produce un código de error distinto que recibe el mismo tratamiento que en el caso del -1).

- `int Puente::leer_de_puerto_32(int puerto, float *puntero)`:

  Descripción:

  Este método lee un valor de 32 bits del puerto del CBN que se le indique. Ha sido dotado de un “timeout” para que no se quede eternamente esperando una lectura.

  Parámetros:

  Los parámetros que se le pasan son un número entero correspondiente al número del puerto y un puntero a flotante de precisión simple para pasarle por referencia la dirección del número en coma flotante en que se quiere recoger el resultado.

  Valor devuelto:

  0 si todo va bien, -1 si se desborda la cola de comandos (en cuyo caso se escribe un código de error en fichero y en la cola de errores) y -2 si expira el “timeout” sin haber podido leer el dato (se produce un código de error distinto que recibe el mismo tratamiento que en el caso de que el valor devuelto sea -1).

- `int Puente::escribir_en_puerto_16(int puerto, short int valor)`:

  Descripción:

  Este método escribe un valor de 16 bits en el puerto del CBN que se le

indique. Sirve de base para otras funciones de más alto nivel ya que dependiendo del puerto en que se escriba se tendrán distintas funcionalidades.

Parámetros:

Los parámetros que se le pasan a este método son el número de puerto en el que se quiere escribir y el valor de 16 bits (short int) que se quiere escribir en ese puerto.

Valor devuelto:

0 si todo va bien y -1 si se desborda la cola de comandos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `int Puente::escribir_en_puerto_32(int puerto, float valor)`:

Descripción:

Este método escribe un valor de 32 bits en el puerto del CBN que se le indique. Sirve de base para otras funciones de más alto nivel ya que dependiendo del puerto en que se escriba se tendrán distintas funcionalidades.

Parámetros:

Los parámetros que se le pasan a este método son el número de puerto en el que se quiere escribir y el valor de 32 bits (float) que se quiere escribir en ese puerto.

Valor devuelto:

0 si todo va bien y -1 si se desborda la cola de comandos, en cuyo caso se escribe un código de error en fichero y en la cola de errores.

- `void Puente::activar_tx_de_estado():`

  Descripción:

  Este método ordena al CBN que envíe el estado del helicóptero cada periodo de muestreo, que valdrá típicamente 20 ms. Por defecto se envía el estado completo pero se puede configurar para obtener únicamente aquéllos campos del estado que nos interesen de acuerdo con una máscara binaria que será preciso facilitar al CBN con la función “envia_mascara”.

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

- `void Puente::envia_mascara(int mascara):`

  Descripción:

  Este método llama a la función “escribir_en_puerto_16” para escribir la máscara en un puerto determinado del CBN (el 10000). La máscara es un patrón binario que indica cuales de las variables del estado del helicoptero (que mantiene el CBN) deben ser transmitidas hacia el CAN. En el CBN se sobreescribe la máscara por defecto, que es 0xffff, con el valor que le se ha pasado y a partir de este momento solo se recibirán en el CAN aquellos campos del estado del helicóptero que estén incluidos en la máscara.

  Parámetros:

  Un entero que corresponde a la nueva máscara.

Valor devuelto:

No tiene.

- `void Puente::desactivar_tx_estado():`

Descripción:

Este método da una orden al CBN para que deje de enviar el estado del helicóptero.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Control::limpiar_codigo_de_error():`

Descripción:

Este método se utiliza para limpiar las colas de errores del CBN. Se ignoran todos los errores que se hayan producido hasta este momento. Debe incluirse en la inicialización del bucle de control.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puente::invalidar_camino():`

Descripción:

Este método se usa para invalidar el camino que tenía almacenado el CBN. Los puntos del camino (“waypoints”) que el CBN tenía almacenados son desechados. Estos puntos son los que se envían con los métodos *envia_datos_camino_xyz* y *envia_datos_camino_geo.* Se debe incluir una llamada a esta función en la inicialización del bucle de control.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puente::activar_seguimiento():`

Descripción:

Este método se usa para activar el seguimiento de trayectorias por parte del CBN. Cuando el CBN recibe el comando correspondiente a este método, empieza a seguir secuencialmente los puntos del camino que tenía almacenados.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puente::desactivar_seguimiento():`

Descripción:

Este método se usa para desactivar el seguimiento de trayectorias por parte del CBN. Cuando el CBN recibe el comando correspondiente a este método deja de seguir los puntos del camino que tenía almacenados.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puente::habilitar_coordenadas_geograficas():`

Descripción:

Este método hace una llamada a "escribir_en_puerto_16" para escribir un 1 en un puerto determinado del CBN (el 10001). De este modo se indica al CBN que los puntos del camino están en coordenadas geográficas y no en cartesianas (valor por defecto).

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puente::habilitar_coordenadas_cartesianas():`

Descripción:

Este método hace una llamada a “escribir_en_puerto_16” para escribir un 0 en un puerto determinado del CBN (el 10001). De este modo se indica al CBN que los puntos del camino están en coordenadas cartesianas (valor por defecto) y no en geográficas .

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puente::activar_aceleraciones_y_velocidades():`

Descripción:

Este método hace una llamada a “escribir_en_puerto_16” para escribir un 1 en un puerto determinado del CBN (el 10002). De este modo se indica al CBN que debe recoger ángulos de Euler, aceleraciones y velocidades de la IMU . Por defecto solo se recogen los ángulos de Euler de la IMU de modo que sea posible tener un dato del estado del helicóptero cada 20 ms. Si se recoge todo no es posible tener un dato del estado del helicóptero cada 20 ms.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puente::desactivar_aceleraciones_y_velocidades():`

Descripción:

Este método hace una llamada a “escribir_en_puerto_16” para escribir un 0 en un puerto determinado del CBN (el 10002). De este modo se indica al CBN que recoja únicamente los ángulos de Euler de la IMU. Éste es el comportamiento por defecto.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

3.3.2.3.2. Métodos destinados al usuario para la adquisición de datos sobre el estado del helicóptero y supervisión de errores.-

En este apartado se recogen algunos métodos públicos de la clase “Estimated” para la obtención de datos sobre el estado del helicóptero y para sacar errores de una cola. Se guardan las últimas estructuras de estado y se ponen disponibles al usuario para que las use en sus algoritmos de control. El número de estados que se guarda es configurable con la macro TAM_TABLA_ESTADOS.

- `void Estimated::actualizar(Puente *p_puente);`

Descripción:

Este método saca las últimas respuestas recibidas del CBN. Si se recibe un estado del helicóptero se guarda en una tabla circular. Si se reciben errores se actualizan ciertas banderas y se meten los errores en una cola destinada a tal propósito. El usuario puede sacar los errores de la cola en el bucle de control y actuar en consecuencia. Este método debe utilizarse una vez en cada pasada del bucle de control, antes de tomar ninguna decisión. Los datos obtenidos sobre errores y estado del helicóptero se utilizarán para tomar decisiones en el bucle de control.

Parámetros:

Un puntero a un objeto “Puente” para tener acceso a las colas y métodos del objeto que se le pase por referencia.

Valor devuelto:

No tiene.

- `bool Estimated::errores();`

Descripción:

Esta función devuelve verdadero si se recogieron errores procedentes del ordenador de tierra (PST) o del CBN en la última actualización.

Parámetros:

No tiene

Valor devuelto:

true (verdadero) si hubo errores y false (falso) si no los hubo.

- `bool Estimated::errores_cbn();`

Descripción:

Esta función devuelve verdadero si se recogieron errores procedentes del CBN en la última actualización.

Parámetros:

No tiene

Valor devuelto:

true (verdadero) si hubo errores y false (falso) si no los hubo.

- `bool Estimated::errores_tierra();`

Descripción:

Esta función devuelve verdadero si se recogieron errores procedentes del ordenador de tierra (PST) en la última actualización.

Parámetros:

No tiene.

Valor devuelto:

true (verdadero) si hubo errores y (false)falso si no los hubo.

- `bool Estimated::nuevo_estado();`

  Descripción:

  Esta función devuelve verdadero si se recogió algún estado nuevo en la última actualización.

  Parámetros:

  No tiene.

  Valor devuelto:

  true (verdadero) si se recogió un nuevo estado y false (falso) si no.

- `COORDENADAS_GEO Estimated::obtener_coordenadas_geo(int numestado);`

```
typedef struct{
  float latitud;   // + = norte ; - = sur
  float longitud;  // + = este  ; - = oeste
  float altura_gps;} COORDENADAS_GEO;
```

  Descripción:

  Esta función recoge una estructura de COORDENADAS_GEO del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

  Parámetros:

  Un entero que corresponde al número de estado del que se quiere recoger la estructura COORDENADAS_GEO .

Valor devuelto:

Una estructura COORDENADAS_GEO que contiene las coordenadas geográficas pedidas: latitud, longitud y altura medida por el GPS.

- `COORDENADAS_XY Estimated::obtener_coordenadas_xy(int numestado);`

```
typedef struct{
        float x;
        float y;} COORDENADAS_XY;
```

Descripción:

Esta función recoge una estructura de COORDENADAS_XY del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura COORDENADAS_XY .

Valor devuelto:

Una estructura COORDENADAS_XY que contiene las coordenadas cartesianas pedidas.

- `bool Estimated::obtener_estado_del_switch_automático_manual(int numestado);`

Descripción:

Esta función recoge el valor del switch automático manualdel estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo y lo devuelve como variable booleana.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger el estado del switch automático-manual.

Valor devuelto:

Un valor booleano true (verdadero) si el switch está en la posición de automático y false (falso) si está en manual.

- `GPS_QOS Estimated::obtener_gps_qos(int numestado);`

```
typedef struct{
          char numero_de_satelites;
          char rt20_status;
          char sol_status;
             } GPS_QOS;
```

Descripción:

Esta función recoge una estructura GPS_QOS relativa a la calidad del servicio del GPS del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura GPS_QOS.

Valor devuelto:

Una estructura GPS_QOS que contiene la calidad de servicio del GPS.

- `short int Estimated::obtener_medida_sonar(int numestado);`

Descripción:

Esta función recoge un número entero de 16 bits (short int) correspondiente a la medida del sonar del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la medida del sonar.

Valor devuelto:

Un entero de 16 bits que contiene la medida del sonar.

- `short int Estimated::obtener_nivel_baterias(int numestado);`

Descripción:

Esta función recoge un número entero de 16 bits (short int) correspondiente al nivel de las baterías del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger el nivel de las baterías.

Valor devuelto:

Un entero de 16 bits que contiene el nivel de las baterías.

- `char Estimated::obtener_nivel_combustible(int numestado);`

Descripción:

Esta función recoge un carácter de 8 bits correspondiente al nivel del combustible del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo. El contenido de este carácter está aún por definir.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger el nivel del combustible.

Valor devuelto:

Un carácter de 8 bits que corresponde al nivel del combustible.

- `ORIENTACION Estimated::obtener_orientacion(int numestado);`

```
typedef struct{
   float pitch;
   float roll;
   float yaw;} ORIENTACION;
```

Descripción:

Esta función recoge una estructura ORIENTACION que contiene pitch, roll y yaw medidos por la IMU del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura ORIENTACION.

Valor devuelto:

Una estructura ORIENTACION que contiene pitch, roll y yaw.

- `ACELERACIONES Estimated::obtener_aceleraciones(int numestado);`

```
typedef struct{
          float aceleracion_x;
          float aceleracion_y;
          float aceleracion_z;} ACELERACIONES;
```

Descripción:

Esta función recoge una estructura ACELERACIONES que contiene tres números flotantes de precisión simple correspondientes a las aceleraciones lineales del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura ACELERACIONES.

Valor devuelto:

Una estructura ACELERACIONES que contiene tres números reales de precisión simple correspondientes a las aceleraciones lineales en cada uno de los ejes cartesianos.

- `PAN_TILT Estimated::obtener_pan_tilt(int numestado);`

```
typedef struct{
      int  pan;
      int tilt;} PAN_TILT;
```

Descripción:

Esta función recoge una estructura PAN_TILT que contiene un entero correspondiente al ángulo relativo al pan y otro al ángulo relativo al tilt del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura PAN_TILT.

Valor devuelto:

Una estructura PAN_TILT que contiene dos enteros: el ángulo del pan y el del tilt.

- `PWM Estimated::obtener_pwm(int numestado);`

```
typedef struct{
      short int pwm_medidos[7];} PWM;
```

Descripción:

Esta función recoge una estructura pwm que contiene una tabla de siete enteros de 16 bits correspondientes a las medidas de los PWM del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger la estructura PWM.

Valor devuelto:

Una estructura PWM que contiene las medidas de los PWM como enteros de 16 bits.

- `float Estimated::obtener_rpm_rotor(int numestado);`

Descripción:

Esta función recoge un número en coma flotante de precisión simple que corresponde a las revoluciones por minuto del rotor del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger las revoluciones por minuto del rotor.

Valor devuelto:

Un número flotante de precisión simple que contiene las revoluciones por minuto del rotor.

- `unsigned int Estimated::obtener_tiempo_local_cbn(int numestado);`

Descripción:

Esta función recoge un valor de tiempo en milisegundos medido por el CBN desde su arranque. Este valor de tiempo corresponde al estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger el valor del tiempo local al CBN.

Valor devuelto:

Un entero sin signo que contiene el valor del tiempo en milisegundos leído por el CBN desde su arranque.

- `VELOCIDADES_ANGULARES Estimated::obtener_velocidades_angulares(int numestado);`

```
typedef struct{
          float velocidad_angular_x;
          float velocidad_angular_y;
          float velocidad_angular_z;
        } VELOCIDADES_ANGULARES;
```

Descripción:

Esta función recoge las velocidades angulares medidas por el CBN del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger las velocidades angulares.

Valor devuelto:

Una estructura VELOCIDADES_ANGULARES que contiene las velocidades angulares leídas por el CBN.

- `VELOCIDADES_Y_HEADING Estimated::obtener_velocidades_y_heading(int numestado);`

```
typedef struct{
  float direccion_con_respecto_al_norte;
  float velocidad_horizontal;
  float velocidad_vertical;} VELOCIDADES_Y_HEADING;
```

Descripción:

Esta función recoge la dirección con respecto al norte, velocidad horizontal y velocidad vertical del estado que se indica como parámetro siendo 0 el estado más actual y (TAM_TABLA_ESTADOS -1) el más antiguo.

Parámetros:

Un entero que corresponde al número de estado del que se quiere recoger velocidades y heading.

Valor devuelto:

Una estructura VELOCIDADES_Y_HEADING que contiene tres números reales en punto flotante de precisión simple que corresponden a la dirección con respecto al norte, velocidad horizontal y velocidad vertical.

- `int Estimated::sacar_de_cola_errores(ERRORES *p_dato);`

Descripción:

Esta función saca un error de una cola circular de errores que se

encuentra dentro del objeto Estimated. El error se recoge por referencia. Sería aconsejable para el usuariosacar todos los errores de esta cola al comienzo de cada periodo del bucle de control para tener toda la información sobre los errores que se han producido y actuar en consecuencia.

Parámetros:

Un puntero a una estructura ERRORES para recoger por referencia el dato del error.

Valor devuelto:

0 si se consiguió sacar el error correctamente y -1 si no había ningún error en la cola.

3.4. Descripción exhaustiva del programa.-

Hasta ahora se ha dado una visión del software implementado muy *"orientada a usuario"*, es decir, se ha explicado lo indispensable que debe conocer el investigador para poder probar sus algoritmos de control en el helicóptero. Se han explicado los métodos o funciones declarados como públicos en las distintas clases. Estos métodos pertenecen a la capa de aplicación y permiten obtener información de los distintos sensores (sónar, GPS, IMU) y ejercer actuaciones (sobre los servomecanismos, etc).

La parte del software que se ha desarrollado hasta el momento es la que el usuario debe conocer en profundidad, para poder modificar, utilizar y adaptar el software según sus necesidades.

En el apartado anterior se ha visto métodos que han sido creados específicamente para el uso del usuario. Sin embargo existen un gran número de métodos relacionados con el funcionamiento interno del programa que no serán accesibles al usuario. Estos métodos no deberán ser modificados pues puede peligrar la estabilidad y el buen funcionamiento del programa.

Se puede decir que el programa se divide en dos partes bien diferenciadas:

- La parte del programador:

  Se trata de la parte básica del programa o el esqueleto del mismo. Realiza las tareas básicas necesarias: lectura del fichero de configuración, lanzamiento de hilos, temporización, transmisión y recepción de mensajes tanto por Ethernet como por el puerto serie, concurrencia, etc. Se trata de código de cierta complejidad que no deberá ser modificado por el usuario; esta misión corresponderá a un programador que conozca a fondo el código del programa.

- La parte del usuario:

  Esta parte del código debe ser implementada por el usuario, dotando al programa de alguna funcionalidad o aplicación específica para el helicóptero. Existen apartados dentro del hilo principal destinados a que el usuario escriba su código: declaración de variables locales, inicialización del bucle de control, algoritmo de control, etc. Estos apartados están perfectamente delimitados para que el usuario no modifique el código del programador.

3.4.1. Estructura del programa.-

Cada uno de los programas, el servidor y los clientes, constan de un fichero main.cpp y de una serie de ficheros que contienen los hilos. El usuario deberá escribir su código en ciertos apartados del fichero main.cpp que han sido destinados a tal efecto y en ningún caso deberá modificar el código de los hilos. Para que no existan posibles confusiones que provoquen modificaciones del código del programador por parte del usuario se ha delimitado fuertemente cada uno de los bloques del fichero main.cpp.

La estructura será la misma en todos los programas (servidor, cliente en *modo puente* y cliente en modo procesamiento) pero se diferenciarán en el contenido y funcionalidad de algunos de dichos bloques. Se indicarán las diferencias cuando existan.

En la siguiente tabla puede apreciarse la estructura del hilo main en los programas.

Ficheros de cabecera
Declaración de variables globales
Declaración de variables locales
Captura de las señales SIGINT y SIGTERM

<table>
<tr><td colspan="3">Configuración</td></tr>
<tr><td colspan="3">Recogida de punteros para pasárselos al resto de los hilos</td></tr>
<tr><td colspan="3">Lanzamiento de hilos</td></tr>
<tr><td colspan="3">Temporización</td></tr>
<tr><td colspan="3">Inicialización de usuario</td></tr>
<tr><td colspan="3">Bucle de control</td></tr>
<tr><td>Estimación y registro de datos</td><td>Algoritmo de control</td><td>Actuación</td></tr>
<tr><td colspan="3">Terminación</td></tr>
</table>

1. Ficheros de cabecera.-

Contienen el prototipo de funciones del sistema y del usuario y la definición de constantes y estructuras que se utilizarán a lo largo del programa.

2. Declaración de variables globales.-

Se han tratado de suprimir en la medida de lo posible pues no son aconsejables. Sólo existen dos variables globales: una estructura de tipo PUNTEROS que sólo es global al fichero main.cpp (de modo que se sigue teniendo modularidad) y una variable booleana llamada *fin* que es global a todos los ficheros y sirve para indicar que el bucle de control debe terminar.

3. Declaración de variables locales.-

Este bloque del fichero main será compartido por el programador y por el usuarioexistiendo una zona delimitada para cada uno de ellos.

El programador declara aquí una variable para dar parámetros (política de planificación y prioridad) al hilo principal e instancia los objetos cuyos punteros se dará como parámetros al resto de los hilos. Los objetos podrían haberse declarado globales pues van a ser usados por todos los hilos y con esto se habría reducido la carga de trabajo en el desarrollo; sin embargo las

variables globales no son aconsejables y se ha tratado, en la medida de lo posible, de evitarlas.

El usuariotiene un apartado dentro de este bloque para declarar las variables que necesite en el código de su controlador.

4. Captura de las señales SIGINT y SIGTERM.-

En este bloque se capturan las señales SIGINT y SIGTERM para ser tratadas por un manejador. Cuando se produzca una de estas señales por pulsar CTRL+C o por matar el proceso respectivamente se llamará a una rutina que efectuará una salida ordenada del programa (cerrando ficheros y descriptores de socket y destruyendo los hilos).

5. Configuración.-

En este bloque se lleva a cabo la lectura de un fichero de configuración que se llamará según los casos: can.conf para el servidor, puente.conf para cliente en *modo puente* y procesamiento.conf para cliente en modo procesamiento. Los datos de configuración leídos se almacenaran en una estructura de tipo CONFIG cuyos campos variarán en función del programa que se esté tratando. El contenido del fichero de configuración varía en función del programa al que pertenezca:

5.a) Servidor.- Los campos son los siguientes:

- Modo de funcionamiento: variable booleana que indica funcionamiento como puente si es verdadera y como procesamiento si es falsa.
- Enviar errores a tierra: variable booleana que indica que hay que enviar los errores a tierra si es verdadera.
- Enviar estado a tierra: variable booleana que indica que hay que enviar las estructuras ESTADO_DEL_HELICOPTERO recogidas a tierra si es verdadera.
- Periodo del bucle de control: Tiempo mínimo que debe durar una pasada del bucle de control.
- Puerto tierra: Puerto en el que se escuchan las conexiones del

ordenador de tierra (PST).

- Puerto ppse: Puerto en el que se escuchan las conexiones del ordenador de percepción y supervisión (PPSE).

- IP tierra: Dirección IP desde la que se conecta el ordenador de tierra (PST).

- IP ppse: Dirección IP desde la que se conecta el ordenador de percepción y supervisión (PPSE).

5.b) Cliente .- Tanto para el cliente en *modo puente* como para el cliente en *modo procesamiento* los campos de la estructura CONFIG son los siguientes:

- Periodo del bucle de control: Tiempo mínimo que debe durar una pasada del bucle de control.

- Límite del fichero de logs: número máximo de escrituras que se pueden hacer en el fichero.

- Puerto: Puerto al que se conecta el cliente.

- IP can: Dirección IP del CAN que contiene el programa servidor.

En el bloque de configuración también se establece el valor por defecto de la máscara de estados, que será 0xFFFF. Por lo tanto el comportamiento por defecto es recibir el estado completo del helicóptero.

6. Recogida de punteros para pasárselos al resto de los hilos.-

En este bloque se recogen los punteros a los objetos que han sido instanciados de forma local en la función main y que se deberán usar también en el resto de los hilos. Los punteros de dichos objetos se encapsularán en una estructura de tipo PUNTEROS que se dará como parámetro al resto de los hilos.

7. Lanzamiento de hilos.-

En este bloque se efectúa la configuración del hilo principal en cuanto a política de planificación y prioridad se refiere y se lanzan el resto de los hilos. Los hilos lanzados dependerán del programa que se esté tratando:

7.a) Servidor.-

- Hilo lectura serie.
- Hilo procesa carácter.
- Hilo escritura serie.
- Hilo conexión ethernet PPSE.
- Hilo conexión ethernet tierra.

7.b) Cliente.- Tanto en modo procesamiento como en *modo puente* los hilos serán:

- Hilo escritura ethernet
- Hilo lectura ethernet

8. Temporización.-

Este bloque crea un temporizador que expira con un periodo que será típicamente de 10 ms (aunque se puede configurar) y que sirve de base para solventar nuestras necesidades de temporización: periodo del bucle, tiempo entre lecturas del puerto serie y tiempo de espera de la lectura de un puerto determinado del CBN (TIMEOUT).

9. Inicialización de usuario.-

En este bloque el usuariohará las llamadas a funciones que sean necesarias para inicializar el bucle de control. Estas funciones pueden servir para configurar el comportamiento que se espera del CBN (indicarle qué campos del estado se quiere, activar la transmisión de estados o activar el seguimiento de puntos) o para inicializar el algoritmo de control que se va a ejecutar en el bucle. Todo en el caso del servidor en *modo procesamiento* o el cliente en *modo puente*.

El bloque de inicialización de usuario para el cliente en *modo procesamiento* se encargará de inicializar el bucle de supervisión.

10. Bucle de control.-

Tiene sentido hablar de bucle de control en el servidor en *modo procesamiento* o en el cliente en *modo puente*. Para el cliente en modo procesamiento sería más correcto hablar de bucle de supervisión. El bucle de supervisión es responsabilidad del usuario que lo podrá rellenar a su gusto. El bucle de control tiene a su vez varios subapartados:

10.1. Estimación y registro de datos.- En este subapartado se actualizan los datos sobre el estado del helicóptero (se actualiza una tabla que contiene las últimas estructuras de ESTADO_DEL_HELICOPTERO recibidas desde el CBN) y, si así se indica en el fichero de configuración, se envían estos datos a tierra para su registro en un fichero de *logs*. También se mantiene una cola con los errores que se han producido tanto en el CBN como en el servidor (o cliente en modo puente). Estos datos de errores también pueden enviarse al ordenador de tierra (PST) (en el caso del programa *servidor)* para su tratamiento y/o registro.

10.2. Algoritmo de control.- Partiendo de los datos obtenidos en el apartado anterior, se aplica el algoritmo de control diseñado por el investigador. Se tendrá como resultado unas actuaciones que habrá que enviar al CBN para que las aplique. Estas actuaciones se comunican al CBN a través de escrituras en determinados puertos.

10.3. Actuación.- En este subapartado se envían al CBN las consignas que se ha obtenido en el algoritmo de control.

11. Bloque de terminación.-

Este bloque puede ejecutarse por dos motivos:

1) Se ha producido un error que ha provocado que la variable booleana *fin* tome el valor true (verdadero). En este caso se sale del bucle de control y se ejecuta la función *terminacion* correspondiente al bloque de terminación.

2) Se ha producido la señal SIGINT o SIGTERM (por pulsar CTRL+C o por ejecutar desde la Shell la orden *kill pid*) y el manejador hace una llamada a la función *terminacion.*

En cualquiera de los dos casos el resultado es el mismo: se cierran los ficheros y los descriptores de socket que haya abiertos y se destruyen los hilos. Todo esto se realiza de forma automática haciendo una llamada a la función *exit*.

3.4.2. El protocolo de comunicaciones del puerto serie.-

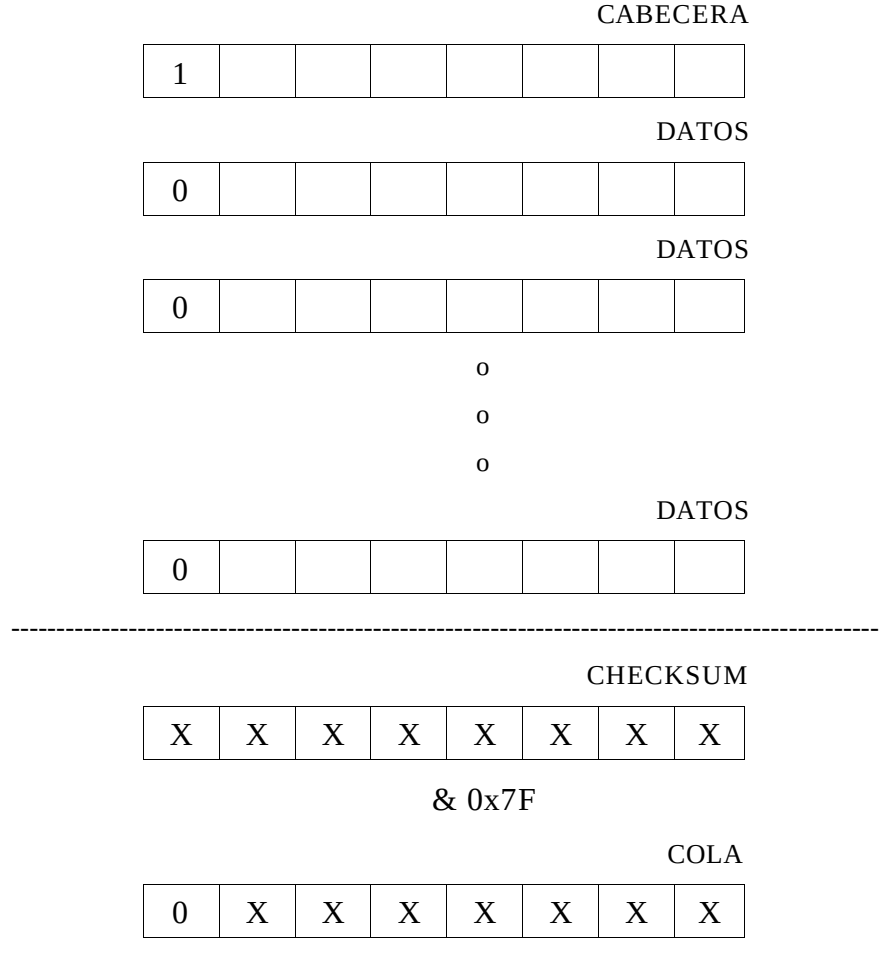
El protocolo de comunicaciones que se ha implementado para comunicar el PC de control con el CBN está basado en el que se diseñó para el vehículo robótico *Romeo4R*(*). El funcionamiento se basa en la transmisión y recepción de *mensajes* o *telegramas*, que no son más que cadenas de caracteres que siguen una estructura determinada. Esta estructura es la misma para todos los *mensajes:*

- Una CABECERA (un carácter).
- Un conjunto de DATOS (opcional, de 0 a 133 caracteres).
- Una COLA (un carácter) que contiene bits de paridad calculados a partir del mensaje completo, esto es, cabecera + datos.

---

* Para más información sobre el protocolo de comunicaciones desarrollado para el vehículo autónomo Romeo 4R léase el PFC titulado "Controlador empotrado para el vehículo autónomo Romeo 4R" de Víctor Manuel Blanco González.

El cálculo del carácter de cola se detalla en la siguiente figura:

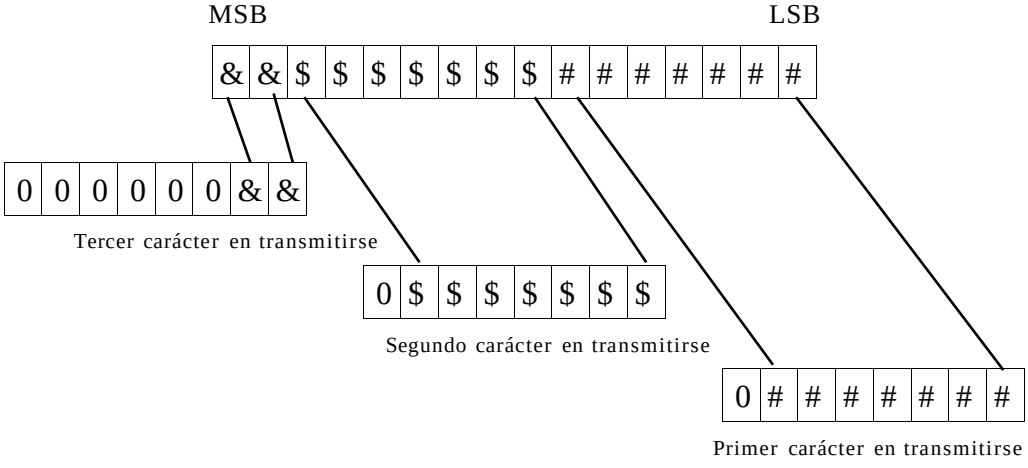


Tanto los mensajes que circulan del Controlador de Bajo Nivel (CBN) al Controlador de Alto Nivel (CAN) como los que circulan en sentido contrario tienen la misma estructura citada anteriormente. Pero estos dos tipos de mensajes difieren en la forma de codificar el campo DATOS. La diferencia consiste en la manera de encapsular los números de 16 y 32 bits en caracteres, que luego serán enviados. La razón de la diferente codificación del campo datos en función del sentido en el que se transmita es que se consigue una menor carga computacional para el CBN, que interesa que esté poco cargado.

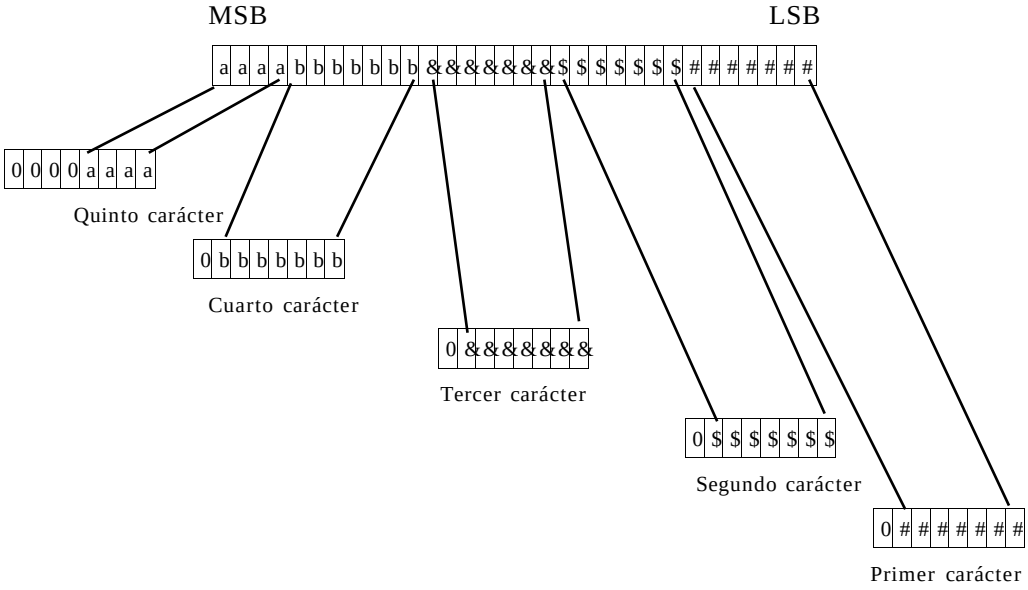
En las siguientes figuras se ilustra la codificación del campo datos desde el punto de vista del controlador de bajo nivel y desde el punto de vista del controlador de alto nivel:

a) Punto de vista del controlador de bajo nivel:

a.1) Desglose de un número de 16 bits en caracteres:

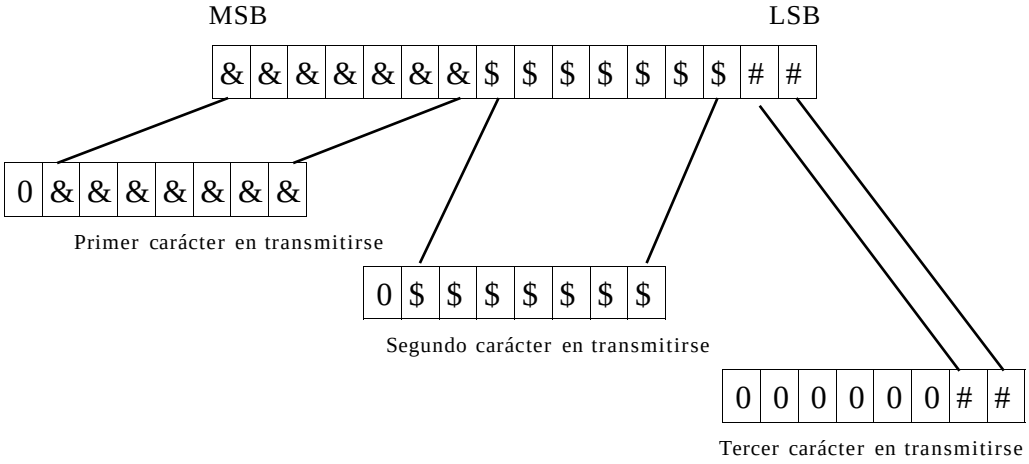


a.2) Desglose de un número de 32 bits en caracteres:

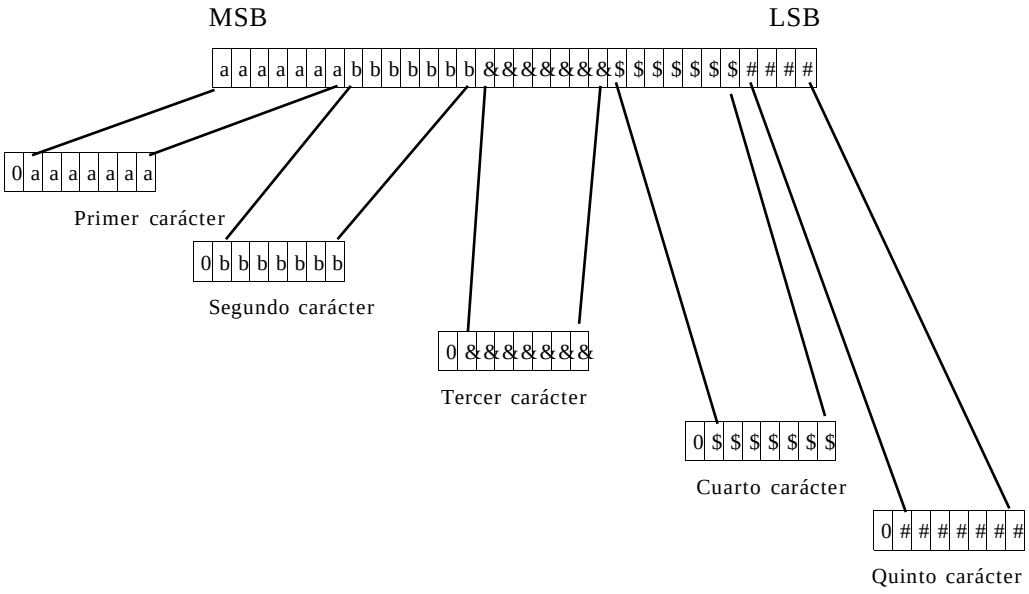


b) Punto de vista del controlador de alto nivel:

b.1) Desglose de un número de 16 bits en caracteres:



b.2) Desglose de un número de 32 bits en caracteres:



Como se puede observar en las figuras, el único carácter del mensaje que tiene su bit más significativo con valor "1" es el carácter de CABECERA. Esta es la manera que tiene el protocolo de identificar el comienzo de un nuevo mensaje. Por el contrario, el resto de caracteres (DATOS + COLA) tendrán su bit más significativo a "0".

Si el mensaje no precisa la transmisión de ningún dato, sólo se transmitirán dos caracteres: la CABECERA y la COLA. En este caso la COLA solo dependerá de la cabecera y será una copia de ésta excepto por el bit más significativo, que valdrá "0" en vez de "1".

A continuación se detallan los mensajes que pueden enviar el controlador de alto nivel y el controlador empotrado.

3.4.2.1. Mensajes procedentes del controlador de alto nivel.-

El intérprete de mensajes que hay incoporado en el software del controlador empotrado podría ser capaz de reconocer hasta 16 tipos de órdenes o comandos. En realidad no se han definido todos los comandos posibles hasta la fecha y se deja abierta la posibilidad de ampliar el número de comandos. Cada uno de estos comandos será un mensaje con las características que se vieron en el apartado anterior. Lo que distinguirá a unos comandos de otros será el carácter de CABECERA. Estos comandos pueden ir acompañados de:

- Un parámetro que será un número de 16 bits.
- Dos parámetros que serán o bien dos números de 16 bits o bien uno de 16 bits y otro de 32 bits.
- Ningún parámetro.

Dependiendo de esto, el campo DATOS del mensaje tendrá diferente longitud.

Todos los comandos enviados al controlador empotrado reciben una respuesta por parte de éste, es decir, son asentidos. El mensaje de asentimiento puede contener un dato de respuesta si se requiere.

Los comandos que no requieren la transmisión de ningún parámetro tienen una cabecera que se contruye de la siguiente manera:

CABECERA= (0x80) OR (nº de comando <0-15>)

Los comandos que requieren la transmisión de un parámetro de 16 bits tienen una cabecera que se contruye de la siguiente manera:

CABECERA= (0xC0) OR (nº de comando <0-15>)

Los comandos que requieren la transmisión de dos parámetros (ya sean dos números de 16 bits o uno de 16 bits y otro de 32 bits) tienen una cabecera que se contruye de la siguiente manera:

CABECERA= (0xE0) OR (nº de comando <0-15>)

Como puede observarse el número de comando se codifica en los 4 bits menos significativos de la cabecera. Con cuatro bits existe la posibilidad de codificar hasta 16 comandos distintos pero, como ya se ha hecho notar, no se han agotado todas las posibilidades de codificación. Aún en el caso de que se agotasen las 16 posibilidades de codificación de comandos, no habría ningún problema porque el software del CBN tiene implementada la entrada y salida mediante una serie de puertos y podría aprovecharse esta característica para definir una serie de puertos virtuales que tendrían asociada una funcionalidad específica, esta vez de comunicación con el programa de control que se ejecuta en el CAN en vez de la funcionalidad de entrada/salida que es la ordinaria. Así se podría crear un nuevo comando sin más que escribir en determinado puerto.

A continuación se detalla una lista de comandos con sus parámetros correspondientes:

Nº de comando	***Acción***	***Parámetro 1 (16 bits)***	***Parámetro 2 (16 ó 32 bits)***
0	Leer nº de 16 bits de puerto E/S	Nº de puerto	X
1	Escribir nº de 16 bits en puerto E/S	Nº de puerto	Valor (16 bits)
2	Leer nº de 32 bits de puerto E/S	Nº de puerto	X
3	Escribir nº de 32 bits en puerto E/S	Nº de puerto	Valor (32 bits)
4	Limpiar código de error	X	X
5	Activar/desactivar tx mensajes de estado	1-> activar 0-> desactivar	X
6	Invalidar camino	X	X
7	Activar/ desactivar seguimiento	1-> activar 0-> desactivar	X

Se han tratado los mensajes de tipo comando que el nodo de control de alto nivel puede enviar al controlador empotrado. Sin embargo el controlador de alto nivel también puede enviar mensajes que no contienen comandos, sino puntos de una trayectoria que se quiere que siga el helicóptero. Estos puntos pueden estar codificados en coordenadas cartesianas o en coordenadas geográficas. En cualquiera de los dos casos el mensaje contendrá tres números de 32 bits codificados en formato de punto flotante. El control de flujo de estos mensajes será tarea del CBN, que solicitará puntos de la trayectoria al controlador de alto nivel hasta completar una cola de puntos. A medida que se vaya vaciando esta cola por haberse alcanzado los puntos en cuestión, el CBN solicitará nuevos puntos al controlador de alto nivel.

Estos mensajes se codifican de la siguiente manera:

Sistema de coordenadas	***CABECERA***	***1ª componente (32 bits)***	***2ª componente (32 bits)***	***3ª componente (32 bits)***
Cartesianas	0xA0	X	Y	Z
Geográficas	0xB0	Latitud	Longitud	Altura

3.4.2.2. Mensajes procedentes del controlador empotrado.-

Existen cuatro tipos de mensajes que el CBN puede enviar al controlador de alto nivel:

A) Mensaje de asentimiento y respuesta a comando.-

Se trata del asentimiento a un comando enviado por el controlador de alto nivel. Como se vio anteriormente, el asentimiento puede contener información si es necesario. Dentro de este grupo de mensajes existen tres subgrupos:

a) Mensajes que no requieren respuesta alguna:
CABECERA= (0xA0) OR (nº de comando < 0-15>)

b) Mensajes cuya respuesta es un número de 16 bits:
CABECERA= (0x90) OR (nº de comando < 0-15>)

c) Mensajes cuya respuesta es un número de 32 bits:
CABECERA= (0x80) OR (nº de comando < 0-15>)

El campo número de comando codifica el número del comando al que se está asintiendo o respondiendo.

En la siguiente tabla se detallan los posibles asentimientos:

Nº de comando	***Respuesta a:***	***Parámetro 1 (16 bits)***	***Parámetro 2 (32 bits)***
0	Leer nº de 16 bits de puerto E/S	Valor leído	X
1	Escribir nº de 16 bits en puerto E/S	Valor escrito	X
2	Leer nº de 32 bits de puerto E/S	X	Valor leído
3	Escribir nº de 32 bits en puerto E/S	X	Valor escrito
4	Limpiar código de error	X	X
5	Activar/desactivar tx mensajes de estado	1-> activar 0-> desactivar	X
6	Invalidar camino	X	X
7	Activar/ desactivar seguimiento	1-> activar 0-> desactivar	X

Uno de los comandos que puede enviar el controlador de alto nivel es la transmisión periódica del estado del helicóptero (comando nº 5). Este mensaje, llamado mensaje de ESTADO, es el siguiente mensaje que se verá de los que envía el CBN.

B) Mensaje de estado del helicóptero:

Está compuesto de:

- CABECERA= 0xFF
- DATOS: Variarán en función de la máscara de estados.

A continuación se explica el significado de cada uno de los bits de la máscara de estados ordenados de bit menos significativo a bit más significativo:

1. Tiempo local al CBN (32 bits)
2. Coordenadas geográficas (3 números de 32 bits)
3. Coordenadas cartesianas (3 números de 32 bits)
4. Velocidades y orientación con respecto al norte (3 números de 32 bits)
5. Calidad de servicio GPS (tres caracteres)
6. Orientación: pitch, roll y yaw (3 números de 32 bits)
7. Aceleraciones (3 números de 32 bits)
8. Velocidades angulares (3 números de 32 bits)
9. Medidas de los PWM (7 números de 16 bits)
10. Medida del sonar (16 bits)
11. Nivel del las baterías (16 bits)
12. Nivel de combustible (8 bits)
13. Estado del conmutador automático- manual (8 bits)
14. Pan & Tilt (dos números de 32 bits)
15. Revoluciones por minuto del rotor (32 bits)

El tamaño de un estado del helicóptero completo, una vez encapsulado en un mensaje, es de 133 caracteres. Normalmente un mensaje de estado tendrá un tamaño menor pues no se envía el estado completo.

- COLA: calculada como en todos los mensajes

C) Mensajes de error:

El CBN lleva a cabo un tratamiento de errores. Cuando se produce algún error, éste es enviado al controlador de alto nivel.

- CABECERA= 0xE0

- DATOS: entero de 16 bits que codifica el tipo de error.
- COLA: como en el resto de los casos.

D) Mensajes de control de flujo:

Este tipo de mensaje sirve para implementar el control de flujo en la transmisión, por parte del controlador de alto nivel, de puntos de la trayectoria que debe seguir el helicóptero. El funcionamiento es como sigue: el CBN realiza una petición de transmisión y el controlador de alto nivel le envía un punto. Una vez recibido el punto, el CBN manda un mensaje de asentimiento, especificando en él si desea que el controlador de alto nivel le envíe otro punto o no.

D.1) Asentimiento con petición de transmisión de un nuevo punto:

- CABECERA = 0xC0
- COLA

D.1) Asentimiento sin petición de transmisión de un nuevo punto:

- CABECERA = 0xCF
- COLA

3.4.3. Hilos de ejecución.-

3.4.3.1. Definición.-

El programa debe efectuar varias tareas de forma simultánea: bucle de control, escucha de conexiones TCP/IP entrantes y lectura/escritura de datos por Ethernet o por el puerto serie. La solución adoptada para tener simultaneidad de tareas son los llamados *"hilos de ejecución"*. Dentro del mismo programa se crearán varios hilos de ejecución, cada uno de los cuales se encargará de una tarea determinada. La comunicación entre los distintos hilos se realizará a través de colas circulares. Cuando un hilo no tenga nada que hacer se bloqueará para no consumir tiempo de procesamiento y despertará cuando sea necesario.

Los hilos de ejecución se pueden definir como cada una de las actividades concurrentes provocadas por un proceso. De modo que la existencia de varios hilos en un proceso implica la ejecución "simultánea" de varias actividades dentro de este proceso. En realidad la ejecución de los hilos no es simultánea sino que el planificador divide el tiempo de CPU entre los distintos hilos y va alternando entre ellos dependiendo de sus prioridades. Los hilos comparten el espacio de direccionamiento pues pertenecen al mismo proceso. Por lo tanto, las variables globales serán comunes a todos los hilos. Sin embargo, cada hilo posee una pila o *stack* separada, que le permite conservar de manera independiente las direcciones de retorno en las llamadas a subrutina, y crear variables locales.

Se ha optado por crear hilos pero se podría haber utilizado procesos. La diferencia fundamental existente entre ambos estriba en la cantidad de recursos que consume cada uno a la hora de crearse. Cuando un proceso crea un nuevo proceso o proceso hijo, se crea en memoria una copia exacta del proceso creador pero con distinto identificador de proceso (PID) y, a partir de ese momento, cada uno continúa la ejecución de manera completamente independiente. La única manera que tienen de comunicarse los procesos padre e hijo es mediante mecanismos IPC (Inter Process Communication o

Comunicación entre procesos), que consumen muchos recursos. Además de las dificultades para comunicar ambos procesos, se tiene el problema de la sobrecarga derivada de la creación de un nuevo proceso, que ocuparía la misma cantidad de memoria que el proceso original.

Estos dos problemas se resuelven mediante el empleo de hilos. La creación de un hilo supone una mucho menor carga para el sistema puesto que al crear un hilo no se duplica la estructura de datos asociada al proceso; solamente es preciso almacenar separadamente para cada hilo la información que le permita continuar su actividad cuando sea necesario (registros de la máquina, incluyendo contador de programa y puntero de pila). Por otro lado la comunicación entre los hilos se resuelve fácilmente mediante el uso de colas circulares, a las que se ha dotado de mecanismos para evitar las condiciones de carrera (situación en el que el resultado depende del orden de ejecución).

Sin embargo, el empleo de hilos tiene un inconveniente: como los hilos comparten el mismo espacio de direcciones, un error de programación podría corromper los datos globales. Este inconveniente no se daría si se usaran procesos con mecanismos IPC *(Inter Proccess Comunication).* Para evitar este problema se ha reducido al mínimo el número de variables globales en el programa.

3.4.3.2. Hilos del programa.-

Los hilos que se han utilizado en el programa están definidos en la norma 1003.1c (POSIX.4a) y se conocen habitualmente como *pthreads* (POSIX threads o hilos POSIX). POSIX (*Portable Operating System Interface*) es un conjunto de normas generadas por el IEEE y estandarizadas por ANSI e ISO. La ventaja de utilizar esta norma es que se consigue un código *portable* e independiente del sistema operativo: el código que se desarrolle podrá en gran medida recompilarse y funcionar en diferentes máquinas y diferentes sistemas operativos.

Crear un hilo POSIX es más simple que activar un nuevo proceso. Del mismo modo que una variable de tipo *pid_t* identifica a un proceso, la variable *pthread_t* identifica a un hilo. Para crear un hilo POSIX y obtener la variable pthread_t que lo identifica es preciso suministrar los siguientes datos:

-Un puntero a la variable pthread_t para permitir su modificación.

-Un conjunto de atributos, agrupados en una variable de tipo pthread_attr_t.

-Un puntero a la rutina de arranque, el *main* del nuevo proceso.

-Un argumento a la rutina de arranque, con tipo de puntero a void. Este puntero podrá apuntar a una estructura que contenga tantos parámetros como se quiera, lo único que se tiene que hacer es interpretar correctamente (casting) el puntero dentro del hilo.

Algunos atributos fijan el comportamiento del hilo en lo que se refiere a planificación de tareas (prioridad, algoritmo empleado); otros fijan el tamaño y la ubicación de la pila, único segmento de datos *privado* del nuevo hilo.

Durante la fase de inicialización, una de las acciones que se realiza es el inicio y configuración de los hilos. De esta manera, cuando finaliza el bloque de inicialización, los hilos se están ejecutando en paralelo con el bucle de control. Cuando se inicia el bucle de control, los hilos ya están listos para aceptar conexiones TCP/IP y para hacer lecturas o escrituras por el puerto serie o por ethernet. El bucle de control intercambia información con los hilos a través de colas circulares.

Cuando ocurre algún evento que provoque la finalización del programa (se ha pulsado Ctrl + C o se ha producido algún error), el programa pasa a la fase de terminación, que se encarga, entre otras cosas, de destruir los hilos que estaban en ejecución.

En el hilo principal del servidor en *modo procesamiento* o del cliente en *modo puente* existen regiones especiales para que los investigadores declaren

sus variables o escriban el código de sus controladores. El resto de los hilos no se debe modificar a menos que se conozca a fondo el funcionamiento del programa.

Existen dos modos de funcionamiento: se empezará por desarrollar los hilos del modo de funcionamiento con procesamiento a bordo y se continuará con el *modo puente*.

3.4.3.3. Descripción de los hilos del programa.-

3.4.3.3.1. Modo de funcionamiento con procesamiento a bordo:

Lo más relevante de este modo de funcionamiento es que el bucle de control se ejecuta en el CAN mientras que los ordenadores de tierra y de percepción y supervisión (PPSE) se encargan de labores de supervisión.

El programa consta de varios hilos de ejecución. En primer lugar se verán los hilos del servidor:

- ➔ Hilo principal: Sus funciones son:
  - Lectura del fichero de configuración para inicializar ciertas variables del programa.
  - Establecimiento de la captura de las señales SIGINT y SIGTERM con un manejador
  - Creación de hilos.
  - Inicialización del temporizador.
  - Bucle de control.
- ➔ Hilo de lectura serie:

Su función es leer un carácter del puerto serie e introducirlo en una cola circular de caracteres. Si no hay caracteres que recoger del puerto serie, el hilo se bloquea hasta que llegue un carácter.

➔ Hilo procesa carácter:

Su función es sacar caracteres de la cola anterior y procesarlos de acuerdo con el protocolo de comunicaciones del puerto serie para obtener una estructura de datos del tipo RESPUESTA, procedente del CBN. Estos dos hilos estaban juntos en un principio pero se separaron para mejorar el funcionamiento del programa. Si no hay caracteres dentro de la cola de caracteres, el hilo se bloquea hasta que haya al menos un carácter en la cola.

➔ Hilo escritura serie:

Su misión es sacar un punto o un comando de su correspondiente cola y enviarlos carácter a carácter hacia el CBN a través del puerto serie. Si tanto la cola de puntos como la cola de comandos están vacías, el hilo se bloquea. Si hay comandos y/o puntos que enviar al CBN, el hilo despierta y los envía a través del puerto serie de acuerdo con el protocolo de comunicaciones que ha sido diseñado para ello.

➔ Hilo conexión ethernet tierra:

Este hilo se queda escuchando conexiones TCP/IP entrantes en un puerto determinado, que se lee de un fichero de configuración, y desde una IP determinada, la del ordenador de tierra (PST) (que también se lee del mismo fichero). Si se trata de acceder desde otra IP, la conexión es rechazada. Si se realiza una conexión exitosa con tierra, este mismo hilo lanza el hilo de escritura ethernet tierra y el hilo de lectura ethernet tierra, para envío de datos hacia tierra y para la recepción de mensajes de supervisión desde tierra respectivamente. En caso de error se finaliza el programa ordenadamente, es decir, cerrando ficheros, descriptores de socket, etc. En caso de ruptura de la conexión, hilo conexión ethernet

tierra destruye los hilos que ha creado y se queda escuchando conexiones entrantes como en un principio.

- Hilo lectura ethernet tierra:

  Este hilo es iniciado por hilo conexión ethernet tierra. Su función es recoger mensajes de supervisión del ordenador de tierra (PST) a través de ethernet y meterlos en una cola. Existen métodos del usuariopara sacar estos mensajes de la cola de manera que estén disponibles en el bucle de control para la toma de decisiones.

- Hilo escritura ethernet tierra:

  Este hilo es iniciado por hilo conexión ethernet tierra. Su función es el envío de datos y respuestas a los mensajes de supervisión a través de ethernet hacia tierra. El usuario, en el bucle de control del hilo principal, encola el mensaje que quiera hacer llegar hacia tierra y el hilo escritura ethernet tierra lo toma de esta cola y lo envía a tierra.

➔ Hilo conexión ethernet PPSE:

Este hilo se queda escuchando conexiones TCP/IP entrantes en un puerto determinado que se lee de un fichero de configuración, y desde una IP determinada, la del PPSE (que también se lee del mismo fichero). Si se trata de acceder desde otra IP, la conexión es rechazada. Si se realiza una conexión exitosa con el ordenador de percepción y supervisión (PPSE), este mismo hilo lanza el hilo de escritura ethernet PPSE y el hilo de lectura ethernet PPSE, para envío de datos hacia el ordenador de percepción y supervisión (PPSE) y para la recepción de mensajes de supervisión desde el ordenador de percepción y supervisión (PPSE) respectivamente. En caso de error se finaliza el programa ordenadamente, es decir, cerrando ficheros, descriptores de socket, etc. En caso de ruptura de la conexión, hilo conexión ethernet PPSE destruye los hilos que ha creado y se queda escuchando conexiones entrantes como en un principio.

- Hilo lectura ethernet PPSE:

  Este hilo es iniciado por hilo conexión ethernet PPSE. Su función es recoger mensajes de supervisión del ordenador de percepción y supervisión (PPSE) a través de ethernet y meterlos en una cola. Existen métodos del usuariopara sacar estos mensajes de la cola de manera que estén disponibles en el bucle de control para la toma de decisiones.

- Hilo escritura ethernet PPSE:

  Este hilo es iniciado por hilo conexión ethernet PPSE. Su función es el envío de datos y respuestas a los mensajes de supervisión a través de ethernet hacia el ordenador de percepción y supervisión (PPSE). El usuario, en el bucle de control del hilo principal, encola el mensaje que quiera hacer llegar hacia el ordenador de percepción y supervisión (PPSE) y el hilo escritura ethernet PPSE lo toma de esta cola y lo envía al ordenador de percepción y supervisión (PPSE).

A continuación se verán los hilos del cliente situado en los PCs de tierra y de visión. La estructura de ambos clientes es la misma y tienen los mismos hilos. Se diferencian en el puerto al que se conectan, en su dirección IP y en el uso que el usuariohaga de los métodos que se le han proporcionado.

➔ Hilo principal: Sus funciones son:

- Lectura del fichero de configuración e inicialización de determinadas variables.

- Establecimiento de la captura de las señales SIGINT y SIGTERM con un manejador

- Conexión con el CAN.

- Creación de los hilos de escritura y de lectura ethernet.

- Inicialización del temporizador.

- Intercambio con el CAN de mensajes de supervisión del usuario a través de colas circulares.

➔ Hilo lectura ethernet:

Su función es recoger una respuesta a un mensaje de supervisión previamente enviado o algún otro tipo de dato procedente del CAN y meterlo en una cola circular para ponerlo a disposición del usuario en el bucle principal.

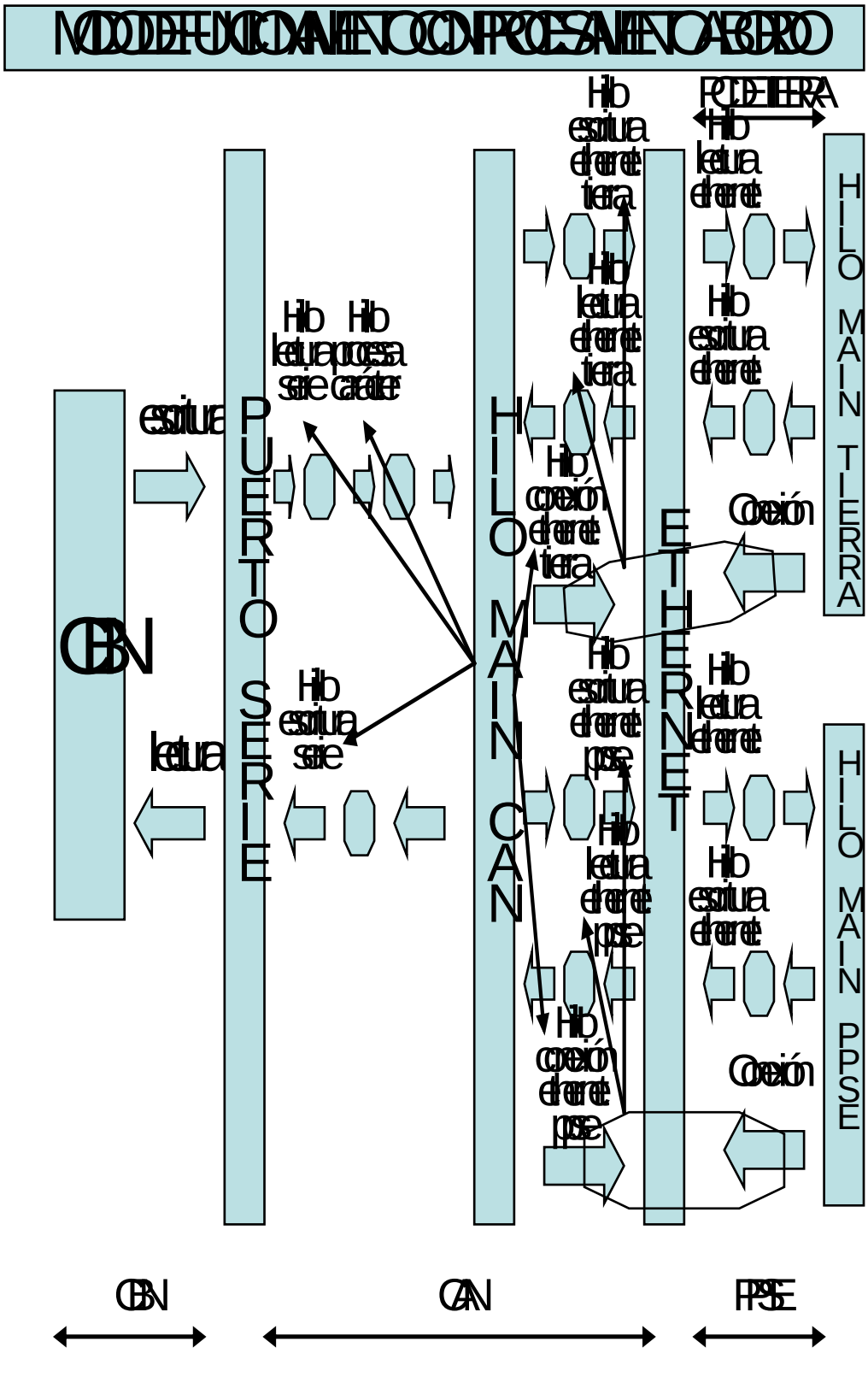
➔ Hilo escritura ethernet:

Su función es vaciar la cola de mensajes de supervisión que llena el hilo principal y enviarlos al CAN a través de ethernet.

En la siguiente figura se puede observar un esquema de este modo de funcionamiento. Las colas circulares se representan mediante círculos. Hago notar que cada cola es de un tipo determinado: cola de caracteres, cola de puntos, cola de comandos, etc. Las colas tienen mecanismos de exclusión mutua para evitar condiciones de carrera (situaciones en las que el resultado depende del orden de ejecución).

Como se puede apreciar en la figura, el hilo principal del CAN inicia los hilos de escritura y lectura serie, el hilo procesa carácter y los hilos de conexión ethernet con tierra y con PPSE. Si, y sólo si, se produce una conexión con éxito, los hilos de conexión ethernet tierra y PPSE lanzan a su vez los respectivos hilos de lectura y escritura ethernet.

Los clientes son más sencillos: realizan la conexión con el servidor del CAN y lanzan los hilos de lectura y escritura ethernet.


PUERTO SERIE
HILO MAIN CAN
ETHERNET
HILO MAIN TIERRA
HILO MAIN PPSE
CBN
CAN
PPSE

3.4.3.3.2. Modo de funcionamiento como puente con tierra:

Este modo de funcionamiento se caracteriza porque el bucle de control se ejecuta en el cliente y no hay interacción con el PC de percepción y supervisión (PPSE). El servidor, es decir, el CAN tiene la única misión de recoger caracteres del puerto serie y enviarlos al cliente a través de la red y de recoger cadenas del cliente y enviarlas al CBN a través del puerto serie.

A continuación se verán los hilos del servidor:

➔ Hilo principal: Sus funciones son:

- Lectura del fichero de configuración para inicializar ciertas variables del programa.
- Establecimiento de la captura de las señales SIGINT y SIGTERM con un manejador
- Creación del hilo de lectura serie, del hilo procesa carácter, del hilo de escritura serie y del hilo de conexión ethernet con tierra.
- Inicialización del temporizador.

➔ Hilo de lectura serie: Su función es leer un carácter del puerto serie e introducirlo en una cola circular de caracteres. Si no hay caracteres que recoger del puerto serie, el hilo se bloquea hasta que llegue un carácter.

➔ Hilo procesa carácter: Su función es sacar caracteres de la cola anterior hasta tener el número de caracteres que constituyen una estructura

RESPUESTA correcta. Cuando se tienen todos los caracteres se meten en una cola de estructuras PETICION. De este modo se mandarán cadenas en vez de caracteres individuales a través de la red. El objetivo de esto es minimizar el número de entradas y salidas puesto que consumen mucho tiempo de procesamiento. Si no hay caracteres en la cola, el hilo se bloquea hasta que los haya.

- Hilo conexión ethernet tierra: Este hilo se queda escuchando conexiones TCP/IP entrantes en un puerto determinado, el puerto 3490, y desde una IP determinada, la del ordenador de tierra (PST). Si se trata de acceder desde otra IP, la conexión es rechazada. Si se realiza una conexión exitosa con tierra, este mismo hilo lanza el hilo de escritura ethernet tierra y el hilo de lectura ethernet tierra. En caso de error se finaliza el programa ordenadamente, es decir, cerrando ficheros, descriptores de socket, etc. En caso de ruptura de la conexión, hilo conexión ethernet tierra destruye los hilos que ha creado y se queda escuchando conexiones entrantes como en un principio.

  - Hilo escritura ethernet tierra: Este hilo es iniciado por hilo conexión ethernet tierra. Su misión es recoger estructuras PETICION de una cola determinada y escribir en ethernet la cadena que hay en uno de sus campos. Si la cola está vacía, el hilo se bloquea hasta que haya al menos un elemento en la cola.

  - Hilo lectura ethernet tierra: Este hilo es iniciado por hilo conexión ethernet tierra. Su misión es recoger una cadena que proviene de ethernet y meterla en una cola circular. Si no hay nada que recoger de la red, el hilo se bloquea.

- Hilo escritura serie: Su función es sacar una cadena de la cola referenciada en hilo lectura ethernet tierra y enviarla por el puerto serie. Si no hay cadenas en la cola, el hilo se bloquea y despierta cuando haya cadenas.

Acto seguido se exponen los hilos del cliente:

➔ Hilo principal:

Sus funciones son:

- Lectura del fichero de configuración para inicializar ciertas variables del programa.
- Establecimiento de la captura de las señales SIGINT y SIGTERM con un manejador
- Conexión con el CAN.
- Creación de los hilos de lectura y escritura ethernet
- Inicialización del temporizador.
- Bucle de control

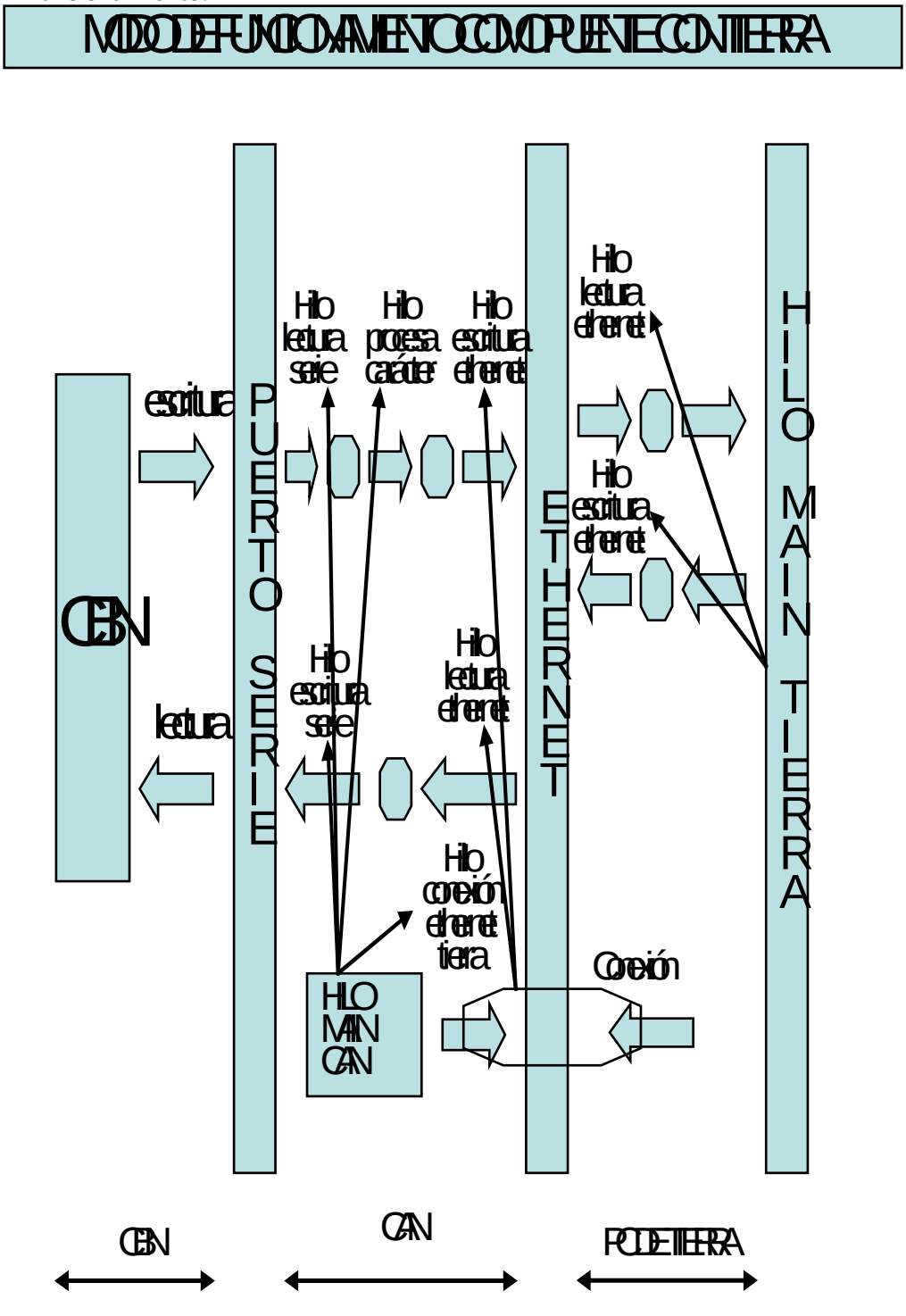
➔ Hilo lectura ethernet:

Su misión es recoger una cadena de caracteres de la red y aplicarle un procesamiento para obtener una estructura RESPUESTA que se mete en una cola para ponerla a disposición del usuarioen el bucle de control. Si no hay nada que recoger de la red, el hilo se bloquea.

➔ Hilo escritura ethernet:

Su función es enviar puntos o comandos al CBN pasando primero por ethernet y luego por el puerto serie. Si no hay puntos ni comandos que enviar, el hilo se bloquea.

En la siguiente figura se puede observar un esquema de este modo de funcionamiento.



Las colas circulares se han representado mediante círculos. Hago notar que cada cola es de un tipo determinado: cola de caracteres, cola de puntos, cola de comandos, etc. Las colas tienen mecanismos de exclusión mutua para evitar condiciones de carrera (situaciones en las que el resultado depende del orden de ejecución).

Como se puede apreciar en la figura, el hilo principal del CAN inicia los hilos de escritura y lectura serie, el hilo procesa carácter y el hilo de conexión ethernet con tierra. Si, y sólo si, se produce una conexión con éxito, el hilo de conexión ethernet tierra lanza a su vez los hilos de lectura y escritura ethernet.

El cliente es más sencillo: realiza la conexión con el servidor del CAN y lanza los hilos de lectura y escritura ethernet.

3.4.4. Modularidad.-

A la hora de escribir el programa se ha puesto especial atención en la modularidad: se han encapsulado los métodos que tenían algo en común en la misma clase y cada clase posee un fichero o módulo independiente.

Cada hilo tiene un fichero independiente. Se han evitado por todos los medios las variables globales: las variables comunes a varios hilos se han declarado dentro de un objeto local al hilo principal, y se ha pasado un puntero a dicho objeto al resto de los hilos. De esta forma el código esta más estructurado y ordenado y facilita posibles nuevas ampliaciones. Si en el futuro se crean nuevas clases que deban ser utilizadas por los hilos, el mecanismo será el mismo: se declaran objetos locales en el hilo principal y se pasa el puntero del objeto al hilo que lo necesite.

3.4.5. Temporización y salida del programa: Señales de 1003.1a.-

En el programa se ha hecho uso de las señales de la norma 1003.1a (POSIX.1). Estas señales tienen pocas posibilidades y son poco apropiadas para tiempo real pero han resultado suficientes para nuestra aplicación.

Su uso más apropiado es la notificación de acontecimientos asíncronos y excepcionales, de modo que se asemeja a una interrupción. El programa desarrollado usa las señales de este modo para efectuar una salida del proceso de manera ordenada. También se utilizarán señales para la temporización.

Las señales pueden ser recogidas por un manejador. En tal caso, al recoger la señal se hace una llamada asíncrona a una función manejadora. Si no se recoge la señal, la acción por defecto es la terminación del programa.

Algunas señales, definidas en < signal.h> son las siguientes:

- SIGFPE: excepción de punto flotante.

-SIGKILL: Terminación de proceso. No se puede bloquear, ignorar o recibir con un manejador, y suele ser la 9, como en "kill -9 <pid>" desde el shell.

- SIGTERM: Terminar proceso, como en "kill <pid>" desde el shell.

- SIGINT: Interrupción, provocada por un carácter especial del terminal de control (normalmente Ctrl + C).

- SIGQUIT: Señal de terminación (también provocada por un carácter).

- SIGUSR1 y SIGUSR2: Señales reservadas para el usuario.

- SIGALRM: Utilizada para indicar fin de temporizador.

De todas estas señales se ha tratado en la aplicación desarrollada con tres:

- SIGINT y SIGTERM han sido recogidas por un manejador que llama a una función para realizar una salida ordenada del programa (cerrando ficheros y descriptores de socket, y destruyendo hilos).

-SIGALARM es generada cada vez que expira un temporizador que se ha creado y es recogida por un manejador que llama a la función "envio_periodico". Esta función incrementa unos contadores (variables estáticas) que al llegar a un determinado valor provocan el inicio de un nuevo periodo del bucle de control, hacen que se realice una nueva lectura del puerto serie o indican que ha expirado el tiempo que se debe esperar a que llegue una lectura (TIMEOUT).

3.4.6. Sincronización.-

En la aplicación desarrollada existen varios hilos ejecutándose simultáneamente. La comunicación entre estos hilos se realiza a través de colas circulares que han sido dotadas de un mecanismo de **"exclusión mutua"**. El objetivo de este mecanismo es evitar las "condiciones de carrera" que son situaciones en las que el resultado depende del orden de ejecución. Cuando varios hilos pueden acceder a la misma posición de memoria, es necesario asegurar que el contenido de ésta siempre sea coherente lo que implica normalmente el restringir el acceso simultáneo a dicha posición de memoria. Esto se podría producir, por ejemplo, si dos hilos tratasen de acceder a la misma posición de memoria de la cola, uno para introducir un dato y otro para sacarlo, o los dos para introducirlo o sacarlo. Como no se sabe en qué momento el manejador interrumpirá a un hilo para dar paso a otro, en determinadas circunstancias las modificaciones que comenzó el primero de los hilos podrán aparecer incompletas para el segundo. El resultado sería impredecible y esto sería desastroso para la aplicación que se quiere desarrollar.

Para proporcionar exclusión mutua se utilizan variables de "mútex". Cuando un hilo quiere acceder a la cola, bloquea la variable de mútex para que ningún otro hilo pueda acceder. Cuando el hilo termina de usar la cola, desbloquea la variable de mutex para que otro hilo pueda acceder a ella. El acceso a la cola se trata como una **"acción atómica"**, o indivisible, que tiene lugar sin que ningún otro hilo detecte los pasos intermedios, ni pueda interrumpirlos.

Además de proporcionar sincronización por exclusión mutua, en la aplicación se requiere **sincronización con condiciones**. Esto consiste en que un hilo debe conocer si se ha producido un determinado acontecimiento como, por ejemplo, que determinada cola circular deje de estar vacía. La sincronización con condiciones permite bloquear a un hilo determinado hasta

que se cumpla cierta condición: de este modo este hilo no consume tiempo de procesamiento y se consigue una mayor eficiencia.

El procedimiento normal y recomendado para esperar una condición es el siguiente:

- Acceder a los datos compartidos a través de un mútex.
- Repetir mientras no se cumpla la condición binaria:
  Esperar la activación de la variable condición (**pthread_cond_wait**).
- Fin Repetir.
- Realizar el proceso necesario.
- Liberar el mútex.

Cuando se hace la llamada **pthread_cond_wait** se libera el mútex asociado a la variable condición y se bloquea el hilo hasta que algún otro hilo active la variable condición con **pthread_cond_signal** o **pthread_cond_broadcast.** Cuando esto sucede, al menos uno de los hilos bloqueado por la señal continuará ejecutándose y su primera acción será intentar bloquear el mútex para tener acceso exclusivo a los datos compartidos. El bucle de comprobación repetida es necesario porque el estándar no garantiza que el hilo desbloqueado sea único.

El proceso normal para desbloquear una condición es el siguiente:

- Acceder a los datos compartidos a través de un mútex.
- Proceso que activa la condición binaria
- Activación de la variable condición (**pthread_cond_signal** o **pthread_cond_broadcast).**
- Liberar el mútex.

La llamada **pthread_cond_signal** desbloquea al menos uno de los hilos que esperan la condición (pero no se garantiza que sea sólo uno). Si no hay ningún hilo esperando, la llamada no tiene ningún efecto. La otra llamada,

pthread_cond_broadcast hace que todos los hilos bloqueados continuen.

El sistema elegirá el hilo que se desbloquea y que consigue el mútex en función del algoritmo de planificación elegido y la prioridad de cada hilo.

3.4.7. Registro de datos.-

Una de las tareas que se quiere que realice la aplicación es el registro de datos sobre el estado del helicóptero que puedan servir al investigador para identificar el modelo. Con esta finalidad se ha decidido que cada vez que se ejecute el programa, se impriman los estados del helicóptero recibidos en un "fichero de logs" en el ordenador de tierra (PST). Todos los ficheros de log se almacenan en el directorio *log*.Para evitar confundir unos ficheros con otros y llevar un histórico de los experimentos se ha decidido guardar los ficheros de log con el formato: DIA_MES_AÑO-HORA_MINUTO_SEGUNDO.log. También se ha decidido limitar el tamaño de los ficheros de log para evitar que se llene el disco duro del ordenador de tierra. El tamaño de estos ficheros de logs es configurable mediante el fichero de configuración del cliente de tierra: *puente.conf* o *procesamiento.conf*.

3.4.8.-Posibilidad de ejecutar los programas como "demonios".-

Se ha proporcionado la posibilidad de ejecutar el programa servidor del ordenador empotrado como un *demonio*. Los demonios son procesos que normalmente se ponen en marcha con el arranque del sistema y que se ejecutan en segundo plano (*background*), es decir, que no necesitan de ninguna intervención directa (a través de la pantalla o el teclado ...) con el usuario. Como los demonios no necesitan de intervención por parte del usuario, no dispondrán de un terminal (tty) asociado. De hecho, uno de los pasos para crear un demonio es la desconexión del proceso de su terminal asociado. Los demonios tienen la ventaja de que una vez que se tiene un código estable para el servidor del CAN, este servidor se ejecutará sin más que arrancar el ordenador. Esto resulta bastante cómodo.

El código para crear el demonio se encuentra comentado en el programa servidor. Para utilizar el demonio habrá que quitar los comentarios y compilar el código resultante. De este modo se obtendrá un ejecutable llamado *server* por ejemplo. Para que se ejecute el demonio al arrancar el programa habrá que colocar un script que contenga la orden de ejecución del demonio en el directorio /etc/init.d/ . Considérese que el script tiene el nombre *server*. El siguiente paso es crear un enlace simbólico a dicho script en los directorios / etc/rc2.d/ y /etc/rc6.d/:

/etc/rc2.d/S99server ---> /etc/init.d/server (script)
/etc/rc2.d/K01server ---> /etc/init.d/server (script)

Con esto el demonio estaría listo y se ejecutaría al arrancar el ordenador.

3.4.9. Métodos de interés para el programador.-

En un apartado anterior se desarrollaban los métodos de interés para el usuario. En este apartado se profundizará en los métodos de interés para el programador, para posibles mejoras o ampliaciones.

Los métodos se clasificarán según el programa en que se puede encontrarlos, esto es, servidor, cliente en *modo puente* o cliente en modo procesamiento. Dentro de cada programa se distinguirán los métodos según la clase a la que pertenezcan. También existen funciones que no pertenecen a ninguna clase.

3.4.9.1 Servidor.-

El servidor tiene dos modos de funcionamiento seleccionables en el fichero de configuración “can.conf”. Se han definido cuatro clases cuyos métodos se detallan a continuación. La mayoría de estos métodos son privados a la clase para que no puedan ser usados desde fuera de la clase. Los métodos públicos y orientados al usuario ya se vieron en un apartado anterior.

3.4.9.1.1.Clase Control.-

En primer lugar se verán métodos para el manejo de las colas

- `int Control::meter_comando(PETICION comando);`

Descripción:

Este método introduce un comando en una cola circular de comandos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION.

Parámetros:

Una estructura de tipo PETICION correspondiente al comando que se quiere meter en la cola.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_punto(PETICION punto);`

Descripción:

Este método introduce un punto en una cola circular de puntos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION.

Parámetros:

Una estructura de tipo PETICION correspondiente al punto que se quiere meter en la cola.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::sacar_comando(PETICION *comando);`

Descripción:

Este método saca un comando de una cola circular de comandos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION.

Parámetros:

Un puntero a una estructura de tipo PETICION para recoger por referencia el comando.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_punto(PETICION *punto);`

  Descripción:

  Este método saca un punto de una cola circular de puntos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION. Esta función tiene un mecanismo de control de flujo y se queda bloqueada si no hay peticiones de transmisión por parte del CBN y si la cola de comandos está vacía. Si la cola de comandos no está vacía y no hay petición de transmisión de puntos, no se permite que se bloquee la función, que simplemente no saca el punto y devuelve -1. Si llega una petición de transmisión se desbloquea la función si estaba bloqueada, y se saca un punto de la cola de puntos si ésta no está vacía.

  Parámetros:

  Un puntero a una estructura de tipo PETICION para recoger por referencia el punto.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba vacía o si no había petición de transmisión y la cola de comandos no estaba vacía.

- `void Control::desbloquea_sacar_punto();`

  Descripción:

  Comprueba si hay peticiones de transmisión de puntos, y en tal caso desbloquea las funciones sacar_punto y sacar_comando.

  Parámetros:

  No tiene.

  Valor devuelto:

No tiene.

- `bool Control::cola_puntos_no_llena();`

  Descripción:

  Comprueba si la cola de puntos no está llena.

  Parámetros:

  No tiene.

  Valor devuelto:

  true (verdadero) si la cola de puntos no está llena y falso si sí lo está.

- `int Control::meter_msg_para_enviar(MENSAJE dato,bool true_tierra_false_ppse);`

  Descripción:

  Este método introduce una estructura de tipo MENSAJE en la cola circular correspondiente, ya sea la cola de mensajes para enviar a tierra o la de PPSE.

  Parámetros:

  Una estructura de tipo MENSAJE correspondiente al mensaje que se quiere meter en la cola y una variable booleana que será verdadera si el mensaje se envía a tierra y falsa si se envía hacia PPSE.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_msg_rcv_blq(MENSAJE dato,bool true_tierra_false_ppse);`

  Descripción:

  Este método introduce una estructura de tipo MENSAJE en la cola circular correspondiente, ya sea la cola de mensajes desbloqueantes recibidos de tierra o de PPSE. Estos mensajes se caracterizan porque son necesarios para que

continúe la ejecución del bucle de control. El bucle de control se bloquea a la espera de estos mensajes con la función sacar_msg_rcv_blq.

Parámetros:

Una estructura de tipo MENSAJE correspondiente al mensaje que se quiere meter en la cola y una variable booleana que será verdadera si el mensaje procede de tierra y falsa si procede de PPSE.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_msg_rcv_no_blq(MENSAJE dato,bool true_tierra_false_ppse);`

Descripción:

Este método introduce una estructura de tipo MENSAJE en la cola circular correspondiente, ya sea la cola de mensajes no desbloqueantes recibidos de tierra o de PPSE. Estos mensajes no son necesarios para que continúe la ejecución del bucle de control, es decir, el bucle de control no se queda bloqueado esperándolos.

Parámetros:

Una estructura de tipo MENSAJE correspondiente al mensaje que se quiere meter en la cola y una variable booleana que será verdadera si el mensaje procede de tierra y falsa si procede de PPSE.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_puente_con_cbn(PETICION pet);`

Descripción:

Este método introduce una estructura de tipo PETICION en una cola circular en la que se almacenan hasta que se envían hacia el CBN.

Parámetros:

Una estructura de tipo PETICION correspondiente a los datos que se quiere enviar hacia el CBN.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_puente_con_tierra(PETICION pet);`

Descripción:

Este método introduce una estructura de tipo PETICION en una cola circular en la que se almacenan hasta que se envían hacia el PC de tierra (PST).

Parámetros:

Una estructura de tipo PETICION correspondiente a los datos que se quiere enviar hacia el PC de tierra (PST).

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_respuesta_tierra(RESPUESTA resp);`

Descripción:

Este método introduce una estructura de tipo RESPUESTA en una cola circular en la que se almacenan hasta que se envían hacia el PC de tierra.

Parámetros:

Una estructura de tipo RESPUESTA correspondiente a los datos que se quiere enviar hacia el PC de tierra.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_respuesta_can(RESPUESTA resp);`

  Descripción:

  Este método introduce una estructura de tipo RESPUESTA en una cola circular en la que se almacenan hasta que se recogen en el bucle de control del CAN.

  Parámetros:

  Una estructura de tipo RESPUESTA correspondiente a los datos que se quiere hacer llegar al bucle de control del CAN.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_valor_16(short int valor);`

  Descripción:

  Este método introduce un entero de 16 bits en una cola circular en la que se almacenan hasta que son recogidos por la función sacar_valor_16. Este método se utiliza para recoger lecturas del CBN con los hilos de lectura serie y procesa carácter y ponerlas a disposición del bucle de control a través de la función leer_de_puerto_16.

  Parámetros:

  Un entero de 16 bits correspondiente a la lectura que se quiere poner a disposición del bucle de control.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_valor_32(float valor);`

  Descripción:

Este método introduce un número real de punto flotante de 32 bits en una cola circular en la que se almacenan hasta que son recogidos por la función sacar_valor_32. Este método se utiliza para recoger lecturas del CBN con los hilos de lectura serie y procesa carácter y ponerlas a disposición del bucle de control a través de la función leer_de_puerto_32.

Parámetros:

Un número real de punto flotante de 32 bits correspondiente a la lectura que se quiere poner a disposición del bucle de control.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_ack(ACK ack);`

Descripción:

Este método introduce en una cola circular una estructura de tipo ACK correspondiente a un asentimiento que se espera de un comando que se ha enviado hacia el CBN.

Parámetros:

Una estructura de tipo ACK correspondiente a un asentimiento que se espera de un comando que se ha enviado hacia el CBN.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_caracter(char valor);`

Descripción:

Este método introduce un carácter leído del puerto serie en una cola circular de caracteres para ponerlo a disposición del hilo procesa carácter.

Parámetros:

El carácter que se quiere introducir en la cola.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::sacar_msg_para_enviar(MENSAJE *dato,bool true_tierra_false_ppse);`

  Descripción:

  Este método saca una estructura de tipo MENSAJE en la cola circular correspondiente, ya sea la cola de mensajes para enviar a tierra o la de PPSE.

  Parámetros:

  Un puntero a una estructura de tipo MENSAJE para recoger por referencia el mensaje que se quiere sacar de la cola y una variable booleana que será verdadera si el mensaje se envía a tierra y falsa si se envía hacia PPSE.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_msg_rcv_blq(MENSAJE *dato,bool true_tierra_false_ppse);`

  Descripción:

  Este método saca una estructura de tipo MENSAJE de la cola circular correspondiente, ya sea la cola de mensajes bloqueantes recibidos de tierra o de PPSE. Este método se queda bloqueado hasta que recibe un mensaje que lo desbloquea. Se utiliza para obtener datos de tierra o de PPSE sin los cuales no podría continuar la ejecución del bucle de control.

  Parámetros:

  Un puntero a una estructura de tipo MENSAJE para recoger por referencia el mensaje que se quiere sacar de la cola y una variable booleana que será verdadera si el mensaje procede de tierra y falsa si procede de PPSE.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_msg_rcv_no_blq(MENSAJE *dato,bool true_tierra_false_ppse);`

Descripción:

Este método saca una estructura de tipo MENSAJE de la cola circular correspondiente, ya sea la cola de mensajes no bloqueantes recibidos de tierra o de PPSE. Este método no se queda bloqueado en espera del mensaje.

Parámetros:

Un puntero a una estructura de tipo MENSAJE para recoger por referencia el mensaje que se quiere sacar de la cola y una variable booleana que será verdadera si el mensaje procede de tierra y falsa si procede de PPSE.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_puente_con_cbn(PETICION *pet);`

Descripción:

Este método saca una estructura de tipo PETICION de una cola circular en la que se almacenan hasta que se envían hacia el CBN.

Parámetros:

Un puntero a una estructura de tipo PETICION para recoger por referencia los datos que se quiere enviar hacia el CBN.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_puente_con_tierra(PETICION *pet);`

Descripción:

Este método saca una estructura de tipo PETICION de una cola circular en la que se almacenan hasta que se envían hacia el PC de tierra.

Parámetros:

Un puntero a una estructura de tipo PETICION para recoger por referencia los datos que se quiere enviar hacia el PC de tierra.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_respuesta_tierra(RESPUESTA *resp);`

Descripción:

Este método saca una estructura de tipo RESPUESTA de una cola circular en la que se almacenan hasta que se envían hacia el PC de tierra.

Parámetros:

Un puntero a una estructura de tipo RESPUESTA para recoger por referencia los datos que se quiere enviar hacia el PC de tierra.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_respuesta_can(RESPUESTA *resp);`

Descripción:

Este método saca una estructura de tipo RESPUESTA de una cola circular en la que se almacenan hasta que se recogen en el bucle de control del CAN..

Parámetros:

Un puntero a una estructura de tipo RESPUESTA para recoger por referencia los datos que se necesitan en el bucle de control del CAN.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_valor_16(short int *p_valor);`

Descripción:

Este método se queda bloqueado tratando de sacar un número entero de 16 bits de una cola determinada. Se desbloquea si llega un valor de 16 bits o si expira un temporizador, lo que ocurra antes. Si expira el temporizador se desiste en el intento de sacar el valor de la cola.

Parámetros:

Un puntero a un número entero de 16 bits para recoger por referencia el valor sacado de la cola y ponerlo a disposición del bucle de control a través de la función leer_de_puerto_16.

Valor devuelto:

Devuelve 0 si todo va bien, -1 si la cola estaba vacía y -2 si expiró el temporizador.

- `int Control::sacar_valor_32(float *p_valor);`

Descripción:

Este método se queda bloqueado tratando de sacar un número en punto flotante de 32 bits de una cola circular determinada. Se desbloquea si llega un valor de 32 bits o si expira un temporizador, lo que ocurra antes. Si expira el temporizador se desiste en el intento de sacar el valor de la cola.

Parámetros:

Un puntero a un número en punto flotante de 32 bits para recoger por referencia el valor sacado de la cola y ponerlo a disposición del bucle de control a través de la función leer_de_puerto_32.

Valor devuelto:

Devuelve 0 si todo va bien, -1 si la cola estaba vacía y -2 si expiró el temporizador.

- `int Control::sacar_ack(ACK *p_ack);`

Descripción:

Este método saca de una cola circular el asentimiento que se espera correspondiente a un comando que se ha enviado. Si se recibe un asentimiento del CBN, será comparado con los que se van sacando de la cola, y si no coinciden se da un mensaje de error con un código determinado que se escribirá en fichero y/o se enviará a tierra.

Parámetros:

Un puntero a una estructura del tipo ACK para recoger por referencia el asentimiento esperado que se ha sacado de la cola.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_caracter(char *p_valor);`

Descripción:

Este método saca un carácter de una cola circular. Esta cola sirve de intermediaria entre los hilos de lectura serie y procesa carácter .

Parámetros:

Un puntero a caracteres para recoger por referencia el carácter que se ha sacado de la cola.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

Lectura y escritura de máscara:

- `int Control::lectura_atomica_mask();`

Descripción:

Este método realiza una lectura atómica de la variable máscara de estados (patrón binario que indica qué campos del estado se envían del CBN al CAN). Se ha proporcionado exclusión mutua entre los métodos de lectura y escritura de la variable máscara para evitar condiciones de carrera (situaciones en las que el resultado que se obtiene depende del orden de ejecución).

Parámetros:

No tiene.

Valor devuelto:

Un número entero que corresponde al valor de la máscara.

- `void Control::escritura_atomica_mask(int mask);`

Descripción:

Este método realiza una escritura atómica en la variable máscara de estados (patrón binario que indica qué campos del estado se envían del CBN al CAN). Se ha proporcionado exclusión mutua entre los métodos de lectura y escritura de la variable máscara para evitar condiciones de carrera (situaciones en las que el resultado que se obtiene depende del orden de ejecución).

Parámetros:

Un número entero correspondiente al valor que se quiere escribir en la máscara.

Valor devuelto:

No tiene.

Comprobación de asentimiento

- `bool Control::compara_ack(ACK ack1,ACK ack2);`

Descripción:

Este método realiza la comparación entre dos estructuras del tipo ACK.

Parámetros:

Dos estructuras del tipo ACK.

Valor devuelto:

true (verdadero) si las estructuras ACK son iguales y falso si son distintas.

Control de flujo para el envío de waypoints

- `bool Control::lee_pet_tx();`

Descripción:

Este método lee la variable pet_de_tx para comprobar si el CBN ha hecho una petición de transmisión de un nuevo punto de la trayectoria que debe seguir el helicóptero.

Parámetros:

No tiene.

Valor devuelto:

true (verdadero) si el CBN ha pedido la transmisión de un nuevo punto de la trayectoria y falso si no.

- `void Control::escribe_pet_tx(bool valor);`

Descripción:

Este método escribe la variable pet_de_tx con un valor booleano true (verdadero) si se ha recibido del CBN una petición de transmisión de un nuevo punto de la trayectoria y falso cuando se ha efectuado la transmisión de dicho punto.

Parámetros:

El valor booleano que se quiere escribir en la variable pet_de_tx.

Valor devuelto:

No tiene.

Tratamiento de errores

- `void Control::procesa_error(short int codigo);`

Descripción:

Este método escribe un código de error en un fichero, rellena el campo error de una estructura RESPUESTA con el error e introduce esta última en la cola de respuestas del CAN.

Parámetros:

Un entero de 16 bits correspondiente al código del error que se ha producido.

Valor devuelto:

No tiene.

- `void Control::escribe_error_fichero(short int error_code);`

Descripción:

Este método abre un fichero de errores si no estaba abierto y escribe en él el código de error que se le pase. Cuando el tamaño del fichero de errores sobrepasa un valor determinado, éste se cierra.

Parámetros:

Un entero de 16 bits correspondiente al código del error que se ha producido.

Valor devuelto:

No tiene.

Métodos privados:

- `void Control::inicializa_cola_msg(COLA_MENSAJE *c);`

  Descripción:
  Este método inicializa la cola circular de estructuras MENSAJE que se le pase por referencia.

  Parámetros:
  Un puntero a una cola circular de estructuras de tipo MENSAJE

  Valor devuelto:
  No tiene.

- `void Control::inicializa_cola_ack();`

  Descripción:
  Este método inicializa la cola circular de estructuras ACK.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Control::inicializa_cola_respuesta(COLA_RESPUESTA *c);`

  Descripción:
  Este método inicializa la cola circular de estructuras RESPUESTA que se le pase por referencia.

  Parámetros:
  Un puntero a una cola circular de estructuras del tipo RESPUESTA.

  Valor devuelto:

No tiene.

- `void Control::inicializa_cola_peticion(COLA_PETICION *c);`

  Descripción:
  Este método inicializa la cola circular de estructuras PETICION que se le pase por referencia.

  Parámetros:
  Un puntero a una cola circular de estructuras del tipo PETICION.

  Valor devuelto:
  No tiene.

- `void Control::inicializa_cola_caracter();`

  Descripción:
  Este método inicializa la cola circular de caracteres.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Control::inicializa_cola_valor_16();`

  Descripción:
  Este método inicializa la cola circular de enteros de 16 bits.

  Parámetros:
  No tiene.

  Valor devuelto:

No tiene.

- `void Control::inicializa_cola_valor_32();`

  Descripción:

  Este método inicializa la cola circular de números de punto flotante de 32 bits..

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

- `int Control::enviar_cadena_usr(char *cadena_enviada, bool esperar_respuesta, char *cadena_recibida,bool true_tierra_false_ppse);`

  Descripción:

  Este método envía una cadena de usuario, que deberá ser formateada por el usuario, al PC de tierra o al de percepción y supervisión (PPSE) y se queda bloqueado esperando una respuesta si así se le indica. Esta función sirve de base a otras de más alto nivel.

  Parámetros:

  -La cadena que se quiere enviar.

  -Una variable booleana que indica si el bucle de control está bloqueado en espera de una respuesta por parte del cliente.

  -Una cadena en la que se recoge la respuesta del cliente.

  -Una variable booleana que indica si la cadena se envía al PC de tierra o al PC de percepción y supervisión (PPSE).

  Valor devuelto:

  Se devuelve 0 si todo va bien y -1 si hubo algún error.

- `int Control::enviar_msg_usr(MENSAJE msg_enviado,MENSAJE *msg_recibido, bool true_tierra_false_ppse);`

  Descripción:

  Este método envía una estructura de tipo mensaje que tiene tres campos:el primero es un carácter de cabecera, 'm', que sirve para distinguirlo de otros tipos de mensaje; el segundo indica si el bucle de control se ha bloqueado en espera de una respuesta y el tercero es una cadena de usuario , que deberá ser formateada por el usuario. Esta estructura se envía al PC de tierra o al de percepción y supervisión (PPSE). Esta función sirve de base a otras de más alto nivel y hace uso del método que se vio en el apartado anterior.

  Parámetros:

  -Una estructura de tipo MENSAJE que se envía al cliente

  -Un puntero a una estructura de tipo MENSAJE para recoger la respuesta del cliente.

  -Una variable booleana que indica si el mensaje se envía al PC de tierra o al PC de percepción y supervisión (PPSE).

  Valor devuelto:

  Se devuelve 0 si todo va bien y -1 si hubo algún error.

- `int Control::meter_en_cola_respuesta(COLA_RESPUESTA *c, RESPUESTA dato,pthread_mutex_t *mutex_respuesta);`

  Descripción:

  Este método introduce una estructura de tipo RESPUESTA en una cola circular que se le pase por referencia.

  Parámetros:

  -Un puntero a la cola de estructuras de tipo RESPUESTA (paso de parámetros por referencia).

  -Una estructura de tipo RESPUESTA que se quiere introducir en la cola.

-Un puntero a la variable de exclusión mutua o "mútex" que corresponde a la cola circular que se ha pasado por referencia.

Valor devuelto:

Se devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::meter_en_cola_peticion(COLA_PETICION *c, PETICION dato);`

Descripción:

Este método introduce una estructura de tipo PETICION en una cola circular que se le pase por referencia.

Parámetros:

-Un puntero a la cola de estructuras de tipo PETICION (paso de parámetros por referencia).

-Una estructura de tipo PETICION que se quiere introducir en la cola.

-Un puntero a la variable de exclusión mutua o "mútex" que corresponde a la cola circular que se ha pasado por referencia.

Valor devuelto:

Se devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Control::sacar_cadena_usr(char *cadena,bool true_tierra_false_ppse);`

Descripción:

Este método saca una cadena de usuario (que sigue un formato diseñado por el usuario) de una cola circular determinada. La cadena de usuario puede proceder del PC de percepción y supervisión (PPSE) o del de tierra (PST).

Parámetros:

Una cadena para recoger la cadena de usuario y una variable booleana que

indica si la cadena procede del PC de tierra (PST) o del de percepción y supervisión (PPSE).

Valor devuelto:

Se devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_de_cola_respuesta(COLA_RESPUESTA *c, RESPUESTA *dato,pthread_mutex_t *mutex_respuesta);`

Descripción:

Este método saca una estructura de tipo RESPUESTA de una cola circular que se le pase por referencia.

Parámetros:

-Un puntero a la cola de estructuras de tipo RESPUESTA (paso de parámetros por referencia).

-Un puntero a una estructura de tipo RESPUESTA para recoger por referencia un dato de la cola.

-Un puntero a la variable de exclusión mutua o "mútex" que corresponde a la cola circular que se ha pasado por referencia.

Valor devuelto:

Se devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::sacar_de_cola_peticion(COLA_PETICION *c, PETICION *dato);`

Descripción:

Este método saca una estructura de tipo PETICION de una cola circular que se le pase por referencia.

Parámetros:

-Un puntero a la cola de estructuras de tipo PETICION (paso de parámetros por referencia).

-Un puntero a una estructura de tipo PETICION para recoger por

referencia un dato de la cola.

Valor devuelto:

Se devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Control::empaqueta_comando(char *cadena,int comando,int n_valores,unsigned short valor_a,float valor_b,int tam_valor_b);`

  Descripción:

  Este método introduce un comando, un valor entero de 16 bits y un valor flotante de 32 bits en una cadena de acuerdo con el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie. Los valores de 16 bits y de 32 bits son parámetros que van con el comando y en algunas ocasiones no estarán incluidos puesto que el comando no los necesitará.El valor flotante puede usarse para enviar un entero de 16 bits.

  Parámetros:

  -La cadena que se va a enviar por el puerto serie (que contendrá el comando y los parámetros formateados según el protocolo de comunicaciones).

  -Un entero que contiene el número de comando.

  -Un entero que contiene el número de parámetros que acompañan al comando: ninguno, uno (que será entero de 16 bits) o dos (que pueden ser dos enteros de 16 bits o un entero de 16 bits y un flotante de 32 bits).

  -Un entero de 16 bits que contiene el primero de los posibles parámetros que acompañan al comando.

  -Un flotante de 32 bits que contiene el segundo de los posibles parámetros que acompañan al comando.

  -Una variable entera que indica si el segundo de los parámetros se debe formatear como un número de 16 bits o de 32 bits.

  Valor devuelto:

  Un número entero que contiene el número de caracteres que se ha introducido en la cadena.

- `int Control::empaqueta_datos_camino_xyz(double x, double y, double z, char *cadena);`

  Descripción:

  Este método formatea las tres componentes de un punto en coordenadas cartesianas según el protocolo de comunicaciones del puerto serie. El resultado es una cadena formateada que se envía por el puerto serie carácter a carácter.

  Parámetros:

  Tres números reales de precisión doble correspondientes a las tres componentes del punto en coordenadas cartesianas y la cadena que recoge el punto una vez formateado para ser enviado por el puerto serie.

  Valor devuelto:

  Un número entero que contiene el número de caracteres que se ha introducido en la cadena.

- `int Control::empaqueta_datos_camino_geo(double longitud, double latitud, double altura,char *mensaje_al_cbn);`

  Descripción:

  Este método formatea las tres componentes de un punto en coordenadas geográficas según el protocolo de comunicaciones del puerto serie. El resultado es una cadena formateada que se envía por el puerto serie carácter a carácter.

  Parámetros:

  Tres números reales de precisión doble correspondientes a longitud, latitud y altura y la cadena que recoge el punto una vez formateado para ser enviado por el puerto serie.

  Valor devuelto:

Un número entero que contiene el número de caracteres que se ha introducido en la cadena.

- `void Control::mete_8_en_cadena(char caracter,char *cadena,int *ind,char *check);`

Descripción:

Este método mete un carácter en una posición de una cadena indicada por un índice que se pasa por referencia y actualiza un carácter de checksum que también se pasa por referencia. Se usa para formatear datos según el protocolo de comunicaciones del puerto serie.

Parámetros:

-El carácter que se quiere introducir en la cadena.

-La cadena en que se recoge el resultado (ya formateado para ser enviado por el puerto serie).

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

-Un puntero a carácter para pasar por referencia el carácter de checksum, que será modificado por el método

Valor devuelto:

No tiene

- `void Control::mete_16_en_cadena(unsigned short valor,char *cad,int *ind,char *check);`

Descripción:

Este método mete un entero de 16 bits en una posición de una cadena indicada por un índice que se pasa por referencia y actualiza un carácter de checksum que también se pasa por referencia. Se usa para formatear datos según el protocolo de comunicaciones del puerto serie.

Parámetros:

-El entero de 16 bits que se quiere introducir en la cadena.

-La cadena en que se recoge el resultado (ya formateado para ser enviado por el puerto serie).

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

-Un puntero a carácter para pasar por referencia el carácter de checksum, que será modificado por el método

Valor devuelto:

No tiene

- `void Control::mete_32_en_cadena(float * puntero,char *cad,int *ind,char *check);`

  Descripción:

  Este método mete un número en punto flotante en una posición de una cadena indicada por un índice que se pasa por referencia y actualiza un carácter de checksum que también se pasa por referencia. Se usa para formatear datos según el protocolo de comunicaciones del puerto serie.

  Parámetros:

  -Un puntero a número flotante para pasar por referencia el número que se quiere introducir en la cadena.

  -La cadena en que se recoge el resultado (ya formateado para ser enviado por el puerto serie).

  -Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

  -Un puntero a carácter para pasar por referencia el carácter de checksum, que será modificado por el método

  Valor devuelto:

  No tiene

- `char Control::desempaqueta_numero_de_8_bits(char *p_elemento, int *indice);`

Descripción:

Este método recoge un carácter de una posición de una cadena formateada según el protocolo de comunicaciones del puerto serie. Se usa para desempaquetar los datos contenidos en las cadenas recibidas por el puerto serie. Estas cadenas siguen el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie.

Parámetros:

-La cadena recibida por el puerto serie.

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

Valor devuelto:

El carácter que se ha recogido de la cadena.

- `short int Control::desempaqueta_numero_de_16_bits(char *p_elemento,int *indice);`

Descripción:

Este método recoge un entero de 16 bits de una posición de una cadena formateada según el protocolo de comunicaciones del puerto serie. Se usa para desempaquetar los datos contenidos en las cadenas recibidas por el puerto serie. Estas cadenas siguen el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie.

Parámetros:

-La cadena recibida por el puerto serie.

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

Valor devuelto:

El entero de 16 bits que se ha recogido de la cadena.

- `int Control::desempaqueta_numero_de_32_bits(char *p_elemento,int *indice);`

  Descripción:

  Este método recoge un número entero de 32 bits de una posición de una cadena formateada según el protocolo de comunicaciones del puerto serie. Se usa para desempaquetar los datos contenidos en las cadenas recibidas por el puerto serie. Estas cadenas siguen el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie.

  Parámetros:

  -La cadena recibida por el puerto serie.

  -Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

  Valor devuelto:

  El número entero de 32 bits que se ha recogido de la cadena.

- `int Control::calcula_tam(int mascara);`

  Descripción:

  Esta función calcula el número de caracteres que debe tener una cadena recibida por el puerto serie que contenga el estado del helicóptero. El número de caracteres dependerá de la máscara de estados ya que en función de dicha máscara se enviarán unos campos del estado u otros.

  Parámetros:

  Un entero que contiene la máscara de estados del helicóptero.

  Valor devuelto:

  Un entero que contiene el número de caracteres que debe tener una cadena que contenga el estado del helicóptero.

- `void Control::desempaqueta_mensaje(char *p_mensaje, RESPUESTA *p_respuesta_actual,int mascara);`

Descripción:

Este método desempaqueta un estado del helicóptero que estaba encapsulado en una cadena procedente del CBN y lo introduce en el campo status de la estructura RESPUESTA que se le pase por referencia.

Parámetros:

-La cadena recibida por el puerto serie.

-Un puntero a una estructura RESPUESTA para recoger por referencia el resultado.

-La máscara de estado del helicóptero.

Valor devuelto:

No tiene.

- `int Control::procesa_caracter_recibido(char caracter,RESPUESTA *p_respuesta_actual);`

Descripción:

Este método rellena uno de los campos de la estructura RESPUESTA que se le pase por referencia con los datos que se van recibiendo carácter a carácter por el puerto serie. Se trata de una máquina de estados que guarda memoria de la última vez que se llamó a la función y continúa en el punto en el que lo dejó.

Parámetros:

-El carácter recibido por el puerto serie.

-Un puntero a una estructura RESPUESTA para recoger por referencia el resultado.

Valor devuelto:

Se devuelve 0 si se consiguió desempaquetar con éxito la cadena y rellenar el campo adecuado de la estructura RESPUESTA, 1 si todavía no se tienen suficientes caracteres para obtener una estructura completa y -1 si hubo

error de checksum.

- `void Control::espera_fin_bucle();`

  Descripción:
  Este método sirve para bloquear al bucle de control en espera de que el temporizador dé la orden de inicio de un nuevo periodo del bucle de control. Si en las operaciones realizadas en el bucle de control se supera el tiempo de un periodo de bucle de control, no existe bloqueo. Se pone una cota inferior al tiempo que puede durar una pasada del bucle de control.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Control::fin_bucle();`

  Descripción:
  Este método sirve para desbloquear el bucle de control cuando el temporizador dé la orden de inicio de un nuevo periodo del bucle de control.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `bool Control::comprueba_timeout_16();`

  Descripción:
  Este método comprueba si el bucle de control está bloqueado en espera de una lectura de un puerto de 16 bits.

Parámetros:
No tiene.

Valor devuelto:
true (verdadero) si el bucle de control está bloqueado en espera de una lectura de un puerto de 16 bits y falso si no.

- `void Control::esperar_timeout_16();`

Descripción:
Este método bloquea al bucle de control en espera de una lectura de un puerto de 16 bits. Se trata de una espera temporizada, esto es, no se esperará un tiempo mayor que el indicado por el *TIMEOUT*.

Parámetros:
No tiene.

Valor devuelto:
No tiene.

- `void Control::deja_de_esperar_timeout_16();`

Descripción:
Este método desbloquea al bucle de control si estaba en espera de una lectura de un puerto de 16 bits. Este método se llama cuando expira el temporizador para la lectura, es decir, cuando se alcanza el tiempo de *TIMEOUT*.

Parámetros:
No tiene.

Valor devuelto:
No tiene.

- `bool Control::comprueba_timeout_32();`

  Descripción:
  Este método comprueba si el bucle de control está bloqueado en espera de una lectura de un puerto de 32 bits.

  Parámetros:
  No tiene.

  Valor devuelto:
  true (verdadero) si el bucle de control está bloqueado en espera de una lectura de un puerto de 32 bits y falso si no.

- `void Control::esperar_timeout_32();`

  Descripción:
  Este método bloquea al bucle de control en espera de una lectura de un puerto de 32 bits. Se trata de una espera temporizada, esto es, no se esperará un tiempo mayor que el indicado por el *TIMEOUT*.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Control::deja_de_esperar_timeout_32();`

  Descripción:
  Este método desbloquea al bucle de control si estaba en espera de una lectura de un puerto de 32 bits. Este método se llama cuando expira el temporizador para la lectura, es decir, cuando se alcanza el tiempo de *TIMEOUT*.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

3.4.9.1.2. Clase Estimated.-

La clase Estimated tiene métodos que son en su mayoría públicos y que ya han sido comentados en el capítulo de métodos orientados al usuario. En este apartado se verán los pocos métodos privados que todavía restan por explicar.

- `void Estimated::meter_estado_tabla(ESTADO_DEL_HELICÓPTERO estado);`

  Descripción:

  Este método introduce un nuevo estado del helicóptero en la tabla de estados del helicóptero, desplazando en una posición a los que había antes e incluso expulsando de la tabla al más antiguo si fuera necesario.

  Parámetros:

  Una estructura ESTADO_DEL_HELICOPTERO que se quiere meter en la tabla.

  Valor devuelto:

  No tiene.

- `void Estimated::inicializa_cola_errores();`

  Descripción:

  Este método inicializa la cola circular de estructuras ERRORES que hay encapsulada dentro de la clase Estimated..

Parámetros:
No tiene.

Valor devuelto:
No tiene.

- `int Estimated::meter_en_cola_errores(ERRORES dato);`

Descripción:
Este método introduce una estructura del tipo ERRORES en la cola circular que hay encapsulada dentro de la clase Estimated.

Parámetros:
Una estructura de tipo ERRORES que se quiere meter en la cola.

Valor devuelto:
Se devuelve 0 si todo va bien y -1 si la cola estaba llena.

3.4.9.1.3. Clase Conexión.-

Esta clase contiene métodos relacionados con las comunicaciones de red con los clientes: establecimiento de conexiones TCP/IP, envío y recepción de mensajes y ruptura de las conexiones creadas.

- `void Conexion::conectar(int puerto,int *p_fd);`

Descripción:
Este método escucha conexiones en el puerto que se le indica como parámetro y recoge el descriptor de socket de la conexión en el entero que se le pase por referencia. Este método es privado y sirve de soporte para los métodos conexion_tierra y conexión_PPSE.

Parámetros:

Un entero que indica el puerto en el que se debe escuchar conexiones entrantes y un puntero a entero para recoger por referencia el descriptor de socket de la conexión.

Valor devuelto:

No tiene.

- `void Conexion::conexion_tierra(CONFIG *p_datos_configuracion);`

  Descripción:

  Este método escucha conexiones procedentes desde tierra. La estructura CONFIG que se pasa por referencia contiene el puerto en que deben escucharse las conexiones de tierra y la IP del ordenador de tierra (PST). Estos datos se han obtenido de la lectura de un fichero de configuración *"can.conf"*. Esta función realiza a su vez una llamada a la función conectar.

  Parámetros:

  Una estructura CONFIG que se pasa por referencia y que contiene el puerto en que deben escucharse las conexiones de tierra y la IP del ordenador de tierra (PST).

  Valor devuelto:

  No tiene.

- `void Conexion::conexion_ppse(CONFIG *p_datos_configuracion);`

  Descripción:

  Este método escucha conexiones procedentes del ordenador de percepción y supervisión (PPSE). La estructura CONFIG que se pasa por referencia contiene el puerto en que deben escucharse las conexiones de argso y la IP de PPSE. Estos datos se han obtenido de la lectura de un fichero de

configuración *"can.conf"*. Esta función realiza a su vez una llamada a la función conectar.

Parámetros:

Una estructura CONFIG que se pasa por referencia y que contiene el puerto en que deben escucharse las conexiones de PPSE y la IP de PPSE

Valor devuelto:

No tiene.

- `int Conexion::enviar_a_tierra(void *puntero, int longitud);`

Descripción:

Este método envía al PC de tierra (PST) *longitud* bytes a partir de la dirección de memoria indicada por *puntero.*

Parámetros:

El puntero a la dirección de memoria donde comienzan los datos que se quiere enviar al PC de tierra (PST) y un entero que indica el número bytes que se quiere enviar al PC de tierra (PST).

Valor devuelto:

Un entero que contiene el número de bytes que se han podido enviar.

- `int Conexion::enviar_a_ppse(void *puntero, int longitud);`

Descripción:

Este método envía al PC de percepción y supervisión (PPSE) *longitud* bytes a partir de la dirección de memoria indicada por *puntero.*

Parámetros:

El puntero a la dirección de memoria donde comienzan los datos que se quiere enviar al PC de percepción y supervisión (PPSE) y un entero que indica

el número bytes que se quiere enviar al PC de percepción y supervisión (PPSE).

Valor devuelto:

Un entero que contiene el número de bytes que se han podido enviar.

- `int Conexion::recibir_de_tierra(void *puntero, int longitud);`

Descripción:

Este método recibe del PC de tierra (PST) *longitud* bytes y los recoge a partir de la dirección de memoria indicada por *puntero.*

Parámetros:

El puntero a la dirección de memoria donde se quiere recoger los datos recibidos desde el PC de tierra (PST) y un entero que indica el número bytes que se quiere recoger.

Valor devuelto:

Un entero que contiene el número de bytes que se han podido recibir.

- `int Conexion::recibir_de_ppse(void *puntero, int longitud);`

Descripción:

Este método recibe del PC de percepción y supervisión (PPSE) *longitud* bytes y los recoge a partir de la dirección de memoria indicada por *puntero.*

Parámetros:

El puntero a la dirección de memoria donde se quiere recoger los datos recibidos desde el PC de percepción y supervisión (PPSE) y un entero que indica el número bytes que se quiere recoger.

Valor devuelto:

Un entero que contiene el número de bytes que se han podido recibir.

- `void Conexion::esperar_ruptura_con_tierra();`

  Descripción:
  Este método bloquea al hilo que lo llama en espera de que se rompa la conexión con el ordenador de tierra (PST).

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Conexion::esperar_ruptura_con_ppse();`

  Descripción:
  Este método bloquea al hilo que lo llama en espera de que se rompa la conexión con el ordenador de percepción y supervisión (PPSE).

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Conexion::romper_conexion_con_tierra();`

  Descripción:
  Este método da la señal que desbloquea a los hilos que estaban esperando la ruptura de la conexión con el ordenador de tierra (PST).

  Parámetros:
  No tiene.

Valor devuelto:

No tiene.

- `void Conexion::romper_conexion_con_ppse();`

Descripción:

Este método da la señal que desbloquea a los hilos que estaban esperando la ruptura de la conexión con el ordenador de percepción y supervisión (PPSE).

Parámetros:

No tiene.

Valor devuelto:

No tiene.

3.4.9.1.4. Clase Puerto_serie.-

Esta clase contiene métodos relacionados con el manejo del puerto serie: inicialización, lectura y escritura y cierre del puerto serie.

- `int Puerto_serie::init_puerto_serie(int baud);`

Descripción:

Este método inicializa el puerto serie y lo configura para una velocidad de transmisión y recepción que se le pase como parámetro. Es un método privado que sirve de base a inicia_puerto_serie.

Parámetros:

Un entero que contiene la velocidad de transmisión y recepción en baudios.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si hubo algún error.

- `void Puerto_serie::inicia_puerto_serie();`

  Descripción:
  Comprueba si el puerto serie está inicializado y, si no es así, lo inicializa. La velocidad de transmisión y recepción se define como una constante.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Puerto_serie::posicionar_principio_puerto_serie ();`

  Descripción:
  Este método se sitúa al principio del puerto serie.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Puerto_serie::escribir_caracter_puerto_serie(char ch);`

  Descripción:
  Escribe el carácter que se le pase como parámetro en el puerto serie.

  Parámetros:
  El carácter que se quiere escribir en el puerto serie.

Valor devuelto:

No tiene.

- `unsigned char Puerto_serie::leer_caracter_puerto_serie();`

  Descripción:

  Lee un carácter del puerto serie.

  Parámetros:

  No tiene.

  Valor devuelto:

  El carácter leído.

- `void Puerto_serie::enviar_cadena_puerto_serie( char *cadena, int length);`

  Descripción:

  Escribe en el puerto serie tantos caracteres de la cadena que se le pase como indique el segundo parámetro.

  Parámetros:

  La cadena que se quiere escribir en el puerto serie y un entero que indica el número de bytes que se ha escrito.

  Valor devuelto:

  No tiene.

- `int Puerto_serie::recibir_cadena_puerto_serie ( char *cadena,int length);`

  Descripción:

  Recoge del puerto serie *length* caracteres y los guarda en la cadena que se le pase como parámetro.

Parámetros:

La cadena en la que se recogen los caracteres y un entero que indica el número de bytes que se quiere recoger.

Valor devuelto:

No tiene.

- `void Puerto_serie::esperar();`

Descripción:

Este método bloquea el hilo de lectura serie hasta que expira un temporizador. De este modo se acumulan caracteres en el puerto serie que luego se leen de una sola vez. Se consigue así disminuir el número de entraadas y salidas y reducir la carga del procesador

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `void Puerto_serie::dejar_de_esperar();`

Descripción:

Este método desbloquea el hilo de lectura serie hasta cuando expira un temporizador para que pueda leer los caracteres que se han acumulado en el puerto serie.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

- `int Puerto_serie::close_puerto_serie();`

Descripción:

Cierra el manejador del puerto serie.

Parámetros:

No tiene.

Valor devuelto:

Devuelve 0 si tuvo éxito y -1 si hubo un error.

3.4.9.1.5. Otras funciones.-

En este apartado se discutirán funciones que no pertenecen a ninguna clase. Se empezará con las funciones de los hilos:

- `void iniciar_hilo_procesa_caracter(PUNTEROS *p_punteros);`
- `void *funcion_hilo_procesa_caracter(void *p_punteros);`
- `void destruir_hilo_procesa_caracter();`

- `void iniciar_hilo_lectura_serie(PUNTEROS *p_punteros);`
- `void *funcion_hilo_lectura_serie(void *p_punteros);`
- `void destruir_hilo_lectura_serie();`

- `void iniciar_hilo_lectura_eth_tierra(PUNTEROS *p_punteros);`
- `void *funcion_hilo_lectura_eth_tierra(void *p_punteros);`
- `void destruir_hilo_lectura_eth_tierra();`

- `void iniciar_hilo_lectura_eth_ppse(PUNTEROS *p_punteros);`
- `void *funcion_hilo_lectura_eth_ppse(void *p_punteros);`
- `void destruir_hilo_lectura_eth_ppse();`

- `void iniciar_hilo_escritura_serie(PUNTEROS *p_punteros);`
- `void * funcion_hilo_escritura_serie(void *p_punteros);`
- `void destruir_hilo_escritura_serie();`

- `void iniciar_hilo_escritura_eth_tierra(PUNTEROS *p_punteros);`
- `void *funcion_hilo_escritura_eth_tierra(void *p_punteros);`

- `void destruir_hilo_escritura_eth_tierra();`

- `void iniciar_hilo_escritura_eth_ppse(PUNTEROS *p_punteros);`
- `void *funcion_hilo_escritura_eth_ppse(void *p_punteros);`
- `void destruir_hilo_escritura_eth_ppse();`

- `void iniciar_hilo_conexion_eth_tierra(PUNTEROS *p_punteros);`
- `void *funcion_hilo_conexion_eth_tierra(void *p_punteros);`
- `void destruir_hilo_conexion_eth_tierra();`

- `void iniciar_hilo_conexion_eth_ppse(PUNTEROS *p_punteros);`
- `void *funcion_hilo_conexion_eth_ppse(void *puntero);`
- `void destruir_hilo_conexion_eth_ppse();`

Todas estas funciones son similares así que explicaré una de cada tipo:

- `void iniciar_hilo(PUNTEROS *p_punteros);`

Descripción:

Esta función configura los atributos del hilo y efectúa su lanzamiento pasando como parámetro del hilo un puntero a una estructura PUNTEROS. La estructura PUNTEROS contiene punteros a objetos y estructuras que se han declarado como locales en el main y se necesitan en los hilos. Esto es un artificio para evitar el uso de variables globales que resultan poco elegantes y dificultan la modularidad del código.

Parámetros:

Un puntero a una estructura de tipo PUNTEROS para hacer llegar a los hilos objetos y estructuras creados en el main.

Valor devuelto:

No tiene.

- `void *funcion_hilo(void *puntero);`

Descripción:

Esta función es el hilo de ejecución propiamente dicho. Tiene como parámetro un puntero a void que se puede interpretar como se desee para

pasarle al hilo cualquier número de parámetros. Este parámetro se ha usado para pasarle al hilo un puntero a una estructura de tipo PUNTEROS que contiene a su vez punteros a cada uno de los objetos y estructuras que se necesitan en el hilo.

Parámetros:

Un puntero a void que se debe interpretar adecuadamente para recoger cualquier número de argumentos.

Valor devuelto:

No tiene.

- `void destruir_hilo();`

Descripción:

Esta función destruye un hilo.

Parámetros:

No tiene.

Valor devuelto:

No tiene.

A continuación se desarrollan el resto de funciones:

- `CONFIG configuracion();`

Descripción:

Esta función lee el fichero de configuración *"can.conf"* y rellena una estructura del tipo CONFIG.

Parámetros:

No tiene.

Valor devuelto:

La estructura de tipo CONFIG que contiene los datos de configuración que se han leído en el fichero.

- `void inicializa_timer(int periodo_ms);`

Descripción:

Esta función arranca un temporizador con un periodo que se le da como parámetro. Cada vez que pase un periodo del temporizador se dará la señal SIGALRM y se llamará a la función envio_periodico que corresponde al manejador que se ha definido para esta señal.

Parámetros:

Un entero que corresponde al periodo del temporizador en milisegundos.

Valor devuelto:

No tiene.

- `void envio_periodico(int param);`

Descripción:

Esta función corresponde al manejador que se ha creado para la señal SIGALRM que lanza el temporizador. Cada vez que pasa un periodo del temporizador se llama a la función envio_periodico. El periodo del temporizador será típicamente de 10 ms. Cada vez que se llama a la función envio_periodico se incrementan unas variables estáticas que al llegar a un valor determinado desbloquean algunas funciones:

-Si pasan 50 (u otro número definido en una constante TIMEOUT) periodos de temporizador sin que llegue la lectura que se está esperando de un puerto, se deja de esperar dicha lectura, es decir, ha expirado el TIMEOUT.
-Cada vez que pasa un determinado múltiplo de periodos del temporizador (cuyo valor está definido en una constante) se desbloquea el hilo de lectura serie para que lea de una vez todos los caracteres almacenados en el puerto serie.

--Cada vez que pasa un determinado múltiplo de periodos del temporizador (cuyo número se lee del fichero de configuración *"can.conf"*) se desbloquea el bucle de control para que dé una nueva pasada.

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador.

Valor devuelto:

No tiene.

- `sighandler_t signal(int ,sighandler_t );`

Descripción:

Instala un manejador para la señal que se le indique como primer parámetro.

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador y una función que manejará dicha señal.

Valor devuelto:

La función que corresponde al manejador instalado.

- `void terminacion(int numsem);`

Descripción:

Esta función cierra el puerto serie, los descriptores de fichero, los descriptores de socket y destruye los hilos. Efectúa una salida del programa de forma ordenada. Funciona también como manejador para las señales SIGINT y SIGTERM que se producen pulsando Ctrl+C o ejecutando el comando *kill pid*(donde pid es el identificador del proceso que se quiere matar).

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador (cuando esta función se use como manejador).

Valor devuelto:

No tiene.

- `void captura_sigint_y_sigterm (void (*handler)(int));`

Descripción:

Esta función instala un manejador para las señales SIGINT y SIGTERM que se producen pulsando Ctrl+C o ejecutando el comando *kill pid* (donde pid es el identificador del proceso que se quiere matar) respectivamente. El manejador que se instala es la función terminacion.

Parámetros:

Una función que toma como parámetro un número entero y devuelve void, correspondiente al manejador de las señales SIGINT y SIGTERM.

Valor devuelto:

No tiene.

3.4.9.2. Cliente en *modo puente*.-

En el modo de funcionamiento en *modo puente* el bucle de control se encuentra en el cliente y el servidor tiene la única función de traspasar de modo transparente los datos del cliente al CBN y viceversa. Se han definido tres clases cuyos métodos se detallan a continuación. La mayoría de estos métodos son privados a la clase para que no puedan ser usados desde fuera de la clase. Los métodos públicos y orientados al usuarioya se vieron en un apartado anterior. Muchos de los métodos son idénticos a los que se incluyen en el servidor en modo procesamiento, lo que por otro lado es lógico puesto que la funcionalidad es la misma.

3.4.9.2.1. Clase Puente.-

En primer lugar se verán métodos para el manejo de las colas

- `int Puente::meter_comando(PETICION comando);`

Descripción:

Este método introduce un comando en una cola circular de comandos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION.

Parámetros:

Una estructura de tipo PETICION correspondiente al comando que se quiere meter en la cola.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::meter_punto(PETICION punto);`

Descripción:

Este método introduce un punto en una cola circular de puntos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION.

Parámetros:

Una estructura de tipo PETICION correspondiente al punto que se quiere meter en la cola.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::sacar_comando(PETICION *comando);`

  Descripción:

  Este método saca un comando de una cola circular de comandos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION.

  Parámetros:

  Un puntero a una estructura de tipo PETICION para recoger por referencia el comando.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Puente::sacar_punto(PETICION *punto);`

  Descripción:

  Este método saca un punto de una cola circular de puntos. Tanto los comandos como los puntos pertenecen a una estructura de datos común llamada PETICION. Esta función tiene un mecanismo de control de flujo y se queda bloqueada si no hay peticiones de transmisión por parte del CBN y si la cola de comandos está vacía. Si la cola de comandos no está vacía y no hay petición de transmisión de puntos, no se permite que se bloquee la función, que simplemente no saca el punto y devuelve -1. Si llega una petición de transmisión se desbloquea la función si estaba bloqueada, y se saca un punto de la cola de puntos si ésta no está vacía.

  Parámetros:

  Un puntero a una estructura de tipo PETICION para recoger por referencia el punto.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba vacía o si no había petición de transmisión y la cola de comandos no estaba vacía.

- `void Puente::desbloquea_sacar_punto();`

  Descripción:

  Comprueba si hay peticiones de transmisión de puntos, y en tal caso desbloquea las funciones sacar_punto y sacar_comando.

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

- `int Puente::meter_respuesta(RESPUESTA resp);`

  Descripción:

  Este método mete una estructura de tipo RESPUESTA en una cola circular en la que se almacenan hasta que se recogen en el bucle de control de tierra.

  Parámetros:

  Una estructura de tipo RESPUESTA que se quiere hacer llegar al bucle de control de tierra.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::meter_valor_16(short int valor);`

  Descripción:

  Este método introduce un entero de 16 bits en una cola circular en la que se almacenan hasta que son recogidos por la función sacar_valor_16. Este método se utiliza para recoger lecturas del CBN con los hilos de lectura serie y procesa carácter y ponerlas a disposición del bucle de control a través de la función leer_de_puerto_16.

Parámetros:

Un entero de 16 bits correspondiente a la lectura que se quiere poner a disposición del bucle de control.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::meter_valor_32(float valor);`

Descripción:

Este método introduce un número real de punto flotante de 32 bits en una cola circular en la que se almacenan hasta que son recogidos por la función sacar_valor_32. Este método se utiliza para recoger lecturas del CBN con los hilos de lectura serie y procesa carácter y ponerlas a disposición del bucle de control a través de la función leer_de_puerto_32.

Parámetros:

Un número real de punto flotante de 32 bits correspondiente a la lectura que se quiere poner a disposición del bucle de control.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::meter_ack(ACK ack);`

Descripción:

Este método introduce en una cola circular una estructura de tipo ACK correspondiente a un asentimiento que se espera de un comando que se ha enviado hacia el CBN.

Parámetros:

Una estructura de tipo ACK correspondiente a un asentimiento que se espera de un comando que se ha enviado hacia el CBN.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::sacar_respuesta(RESPUESTA *resp);`

Descripción:

Este método saca una estructura de tipo RESPUESTA de una cola circular en la que se almacenan hasta que se recogen en el bucle de control de tierra.

Parámetros:

Un puntero a una estructura de tipo RESPUESTA para recoger por referencia los datos que se necesitan en el bucle de control de tierra.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Puente::sacar_valor_16(short int *p_valor);`

Descripción:

Este método se queda bloqueado tratando de sacar un número entero de 16 bits de una cola determinada. Se desbloquea si llega un valor de 16 bits o si expira un temporizador, lo que ocurra antes. Si expira el temporizador se desiste en el intento de sacar el valor de la cola.

Parámetros:

Un puntero a un número entero de 16 bits para recoger por referencia el valor sacado de la cola y ponerlo a disposición del bucle de control a través de la función leer_de_puerto_16.

Valor devuelto:

Devuelve 0 si todo va bien, -1 si la cola estaba vacía y -2 si expiró el temporizador.

- `int Puente::sacar_valor_32(float *p_valor);`

  Descripción:

  Este método se queda bloqueado tratando de sacar un número en punto flotante de 32 bits de una cola circular determinada. Se desbloquea si llega un valor de 32 bits o si expira un temporizador, lo que ocurra antes. Si expira el temporizador se desiste en el intento de sacar el valor de la cola.

  Parámetros:

  Un puntero a un número en punto flotante de 32 bits para recoger por referencia el valor sacado de la cola y ponerlo a disposición del bucle de control a través de la función leer_de_puerto_32.

  Valor devuelto:

  Devuelve 0 si todo va bien, -1 si la cola estaba vacía y -2 si expiró el temporizador.

- `int Puente::sacar_ack(ACK *p_ack);`

  Descripción:

  Este método saca de una cola circular el asentimiento que se espera correspondiente a un comando que se ha enviado. Si se recibe un asentimiento del CBN, se comparará con los que se van sacando de la cola, y si no coinciden se da un mensaje de error con un código determinado que se escribirá en fichero y/o se enviará a tierra.

  Parámetros:

  Un puntero a una estructura del tipo ACK para recoger por referencia el asentimiento esperado que se ha sacado de la cola.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

  Lectura y escritura de máscara:

- `int Puente::lectura_atomica_mask();`

  Descripción:

  Este método realiza una lectura atómica de la variable máscara de estados. Se ha proporcionado exclusión mutua entre los métodos de lectura y escritura de la variable máscara para evitar condiciones de carrera (situaciones en las que el resultado que se obtiene depende del orden de ejecución).

  Parámetros:

  No tiene.

  Valor devuelto:

  Un número entero que corresponde al valor de la máscara.

- `void Puente::escritura_atomica_mask(int mask);`

  Descripción:

  Este método realiza una escritura atómica en la variable máscara de estados. Se ha proporcionado exclusión mutua entre los métodos de lectura y escritura de la variable máscara para evitar condiciones de carrera (situaciones en las que el resultado que se obtiene depende del orden de ejecución).

  Parámetros:

  Un número entero correspondiente al valor que se quiere escribir en la máscara.

  Valor devuelto:

  No tiene.

  Comprobación de asentimiento

- `bool Puente::compara_ack(ACK ack1,ACK ack2);`

  Descripción:

Este método realiza la comparación entre dos estructuras del tipo ACK.

Parámetros:
Dos estructuras del tipo ACK.

Valor devuelto:
true (verdadero) si las estructuras ACK son iguales y falso si son distintas.

Control de flujo para el envío de waypoints

- `bool Puente::lee_pet_tx();`

Descripción:
Este método lee la variable pet_de_tx para comprobar si el CBN ha hecho una petición de transmisión de un nuevo punto de la trayectoria que debe seguir el helicóptero.

Parámetros:
No tiene.

Valor devuelto:
true (verdadero) si el CBN ha pedido la transmisión de un nuevo punto de la trayectoria y falso si no.

- `void Puente::escribe_pet_tx(bool valor);`

Descripción:
Este método escribe la variable pet_de_tx con un valor booleano true (verdadero) si se ha recibido del CBN una petición de transmisión de un nuevo punto de la trayectoria y falso cuando se ha efectuado la transmisión de dicho punto.

Parámetros:
El valor booleano que se quiere escribir en la variable pet_de_tx.

Valor devuelto:

No tiene.

Tratamiento de errores

- `void Puente::procesa_error(short int codigo);`

Descripción:

Este método escribe un código de error en un fichero, rellena el campo error de una estructura RESPUESTA con el error e introduce esta última en la cola de respuestas de tierra (para hacérsela llegar al bucle de control).

Parámetros:

Un entero de 16 bits correspondiente al código del error que se ha producido.

Valor devuelto:

No tiene.

- `void Puente::escribe_error_fichero(short int error_code);`

Descripción:

Este método abre un fichero de errores si no estaba abierto y escribe en él el código de error que se le pase. Cuando el tamaño del fichero de errores sobrepasa un valor determinado, éste se cierra.

Parámetros:

Un entero de 16 bits correspondiente al código del error que se ha producido.

Valor devuelto:

No tiene.

Métodos privados:

- `void Puente::inicializa_cola_ack();`

  Descripción:
  Este método inicializa la cola circular de estructuras ACK.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Puente::inicializa_cola_respuesta(COLA_RESPUESTA *c);`

  Descripción:
  Este método inicializa la cola circular de estructuras RESPUESTA que se le pase por referencia.

  Parámetros:
  Un puntero a una cola circular de estructuras del tipo RESPUESTA.

  Valor devuelto:
  No tiene.

- `void Puente::inicializa_cola_peticion(COLA_PETICION *c);`

  Descripción:
  Este método inicializa la cola circular de estructuras PETICION que se le pase por referencia.

  Parámetros:
  Un puntero a una cola circular de estructuras del tipo PETICION.

  Valor devuelto:
  No tiene.

- `void Puente::inicializa_cola_valor_16();`

  Descripción:

  Este método inicializa la cola circular de enteros de 16 bits.

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

- `void Puente::inicializa_cola_valor_32();`

  Descripción:

  Este método inicializa la cola circular de números de punto flotante de 32 bits..

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

- `int Puente::meter_en_cola_respuesta(COLA_RESPUESTA *c, RESPUESTA dato,pthread_mutex_t *mutex_respuesta);`

  Descripción:

  Este método introduce una estructura de tipo RESPUESTA en una cola circular que se le pase por referencia.

  Parámetros:

  -Un puntero a la cola de estructuras de tipo RESPUESTA (paso de parámetros por referencia).

-Una estructura de tipo RESPUESTA que se quiere introducir en la cola.

-Un puntero a la variable de exclusión mutua o "mútex" que corresponde a la cola circular que se ha pasado por referencia.

Valor devuelto:

Se devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::meter_en_cola_peticion(COLA_PETICION *c, PETICION dato);`

  Descripción:

  Este método introduce una estructura de tipo PETICION en una cola circular que se le pase por referencia.

  Parámetros:

  -Un puntero a la cola de estructuras de tipo PETICION (paso de parámetros por referencia).

  -Una estructura de tipo PETICION que se quiere introducir en la cola.

  -Un puntero a la variable de exclusión mutua o "mútex" que corresponde a la cola circular que se ha pasado por referencia.

  Valor devuelto:

  Se devuelve 0 si todo va bien y -1 si la cola estaba llena.

- `int Puente::sacar_de_cola_respuesta(COLA_RESPUESTA *c, RESPUESTA *dato,pthread_mutex_t *mutex_respuesta);`

  Descripción:

  Este método saca una estructura de tipo RESPUESTA de una cola circular que se le pase por referencia.

  Parámetros:

  -Un puntero a la cola de estructuras de tipo RESPUESTA (paso de parámetros por referencia).

-Un puntero a una estructura de tipo RESPUESTA para recoger por referencia un dato de la cola.

-Un puntero a la variable de exclusión mutua o "mútex" que corresponde a la cola circular que se ha pasado por referencia.

Valor devuelto:

Se devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Puente::sacar_de_cola_peticion(COLA_PETICION *c, PETICION *dato);`

  Descripción:

  Este método saca una estructura de tipo PETICION de una cola circular que se le pase por referencia.

  Parámetros:

  -Un puntero a la cola de estructuras de tipo PETICION (paso de parámetros por referencia).

  -Un puntero a una estructura de tipo PETICION para recoger por referencia un dato de la cola.

  Valor devuelto:

  Se devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Puente::empaqueta_comando(char *cadena,int comando,int n_valores,unsigned short valor_a,float valor_b,int tam_valor_b);`

  Descripción:

  Este método introduce un comando, un valor entero de 16 bits y un valor flotante de 32 bits en una cadena de acuerdo con el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie. Los valores de 16 bits y de 32 bits son parámetros que van con el comando y en algunas ocasiones no estarán incluidos puesto que el comando no los necesitará. El

valor flotante puede usarse para enviar un entero de 16 bits.

Parámetros:

-La cadena que se va a enviar por el puerto serie (que contendrá el comando y los parámetros formateados según el protocolo de comunicaciones).

-Un entero que contiene el número de comando.

-Un entero que contiene el número de parámetros que acompañan al comando: ninguno, uno (que será entero de 16 bits) o dos (que pueden ser dos enteros de 16 bits o un entero de 16 bits y un flotante de 32 bits).

-Un entero de 16 bits que contiene el primero de los posibles parámetros que acompañan al comando.

-Un flotante de 32 bits que contiene el segundo de los posibles parámetros que acompañan al comando.

-Una variable entera que indica si el segundo de los parámetros se debe formatear como un número de 16 bits o de 32 bits.

Valor devuelto:

Un número entero que contiene el número de caracteres que se ha introducido en la cadena.

- `int Puente::empaqueta_datos_camino_xyz(double x, double y, double z, char *cadena);`

Descripción:

Este método formatea las tres componentes de un punto en coordenadas cartesianas según el protocolo de comunicaciones del puerto serie. El resultado es una cadena formateada que se envía por el puerto serie carácter a carácter,pasando antes de forma transparente a través de la red y el servidor.

Parámetros:

Tres números reales de precisión doble correspondientes a las tres componentes del punto en coordenadas cartesianas y la cadena que recoge el punto una vez formateado para ser enviado por el puerto serie.

Valor devuelto:

Un número entero que contiene el número de caracteres que se ha introducido en la cadena.

- `int Puente::empaqueta_datos_camino_geo(double longitud, double latitud, double altura,char *mensaje_al_cbn);`

Descripción:

Este método formatea las tres componentes de un punto en coordenadas geográficas según el protocolo de comunicaciones del puerto serie. El resultado es una cadena formateada que se envía por el puerto serie carácter a carácter, pasando antes de forma transparente a través de la red y el servidor.

Parámetros:

Tres números reales de precisión doble correspondientes a longitud, latitud y altura y la cadena que recoge el punto una vez formateado para ser enviado por el puerto serie.

Valor devuelto:

Un número entero que contiene el número de caracteres que se ha introducido en la cadena.

- `void Puente::mete_8_en_cadena(char caracter,char *cadena,int *ind,char *check);`

Descripción:

Este método mete un carácter en una posición de una cadena indicada por un índice que se pasa por referencia y actualiza un carácter de checksum que también se pasa por referencia. Se usa para formatear datos según el protocolo de comunicaciones del puerto serie.

Parámetros:

-El carácter que se quiere introducir en la cadena.

-La cadena en que se recoge el resultado (ya formateado para ser enviado por el puerto serie).

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

-Un puntero a carácter para pasar por referencia el carácter de checksum, que será modificado por el método

Valor devuelto:

No tiene

- `void Puente::mete_16_en_cadena(unsigned short valor,char *cad,int *ind,char *check);`

  Descripción:

  Este método mete un entero de 16 bits en una posición de una cadena indicada por un índice que se pasa por referencia y actualiza un carácter de checksum que también se pasa por referencia. Se usa para formatear datos según el protocolo de comunicaciones del puerto serie.

  Parámetros:

  -El entero de 16 bits que se quiere introducir en la cadena.

  -La cadena en que se recoge el resultado (ya formateado para ser enviado por el puerto serie).

  -Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

  -Un puntero a carácter para pasar por referencia el carácter de checksum, que será modificado por el método

  Valor devuelto:

  No tiene

- `void Puente::mete_32_en_cadena(float * puntero,char *cad,int *ind,char *check);`

Descripción:

Este método mete un número en punto flotante en una posición de una cadena indicada por un índice que se pasa por referencia y actualiza un carácter de checksum que también se pasa por referencia. Se usa para formatear datos según el protocolo de comunicaciones del puerto serie.

Parámetros:

-Un puntero a número flotante para pasar por referencia el número que se quiere introducir en la cadena.

-La cadena en que se recoge el resultado (ya formateado para ser enviado por el puerto serie).

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

-Un puntero a carácter para pasar por referencia el carácter de checksum, que será modificado por el método

Valor devuelto:

No tiene

- `char Puente::desempaqueta_numero_de_8_bits(char *p_elemento, int *indice);`

Descripción:

Este método recoge un carácter de una posición de una cadena formateada según el protocolo de comunicaciones del puerto serie. Se usa para desempaquetar los datos contenidos en las cadenas recibidas por el puerto serie. Estas cadenas siguen el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie. Las cadenas son recogidas por el servidor y enviadas al cliente a través de la red de forma transparente.

Parámetros:

-La cadena recibida por el puerto serie.

-Un puntero a entero para pasar por referencia el índice para recorrer la

cadena.

Valor devuelto:

El carácter que se ha recogido de la cadena.

- `short int Puente::desempaqueta_numero_de_16_bits(char *p_elemento,int *indice);`

Descripción:

Este método recoge un entero de 16 bits de una posición de una cadena formateada según el protocolo de comunicaciones del puerto serie. Se usa para desempaquetar los datos contenidos en las cadenas recibidas por el puerto serie. Estas cadenas siguen el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie. Las cadenas son recogidas por el servidor y enviadas al cliente a través de la red de forma transparente.

Parámetros:

-La cadena recibida por el puerto serie.

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

Valor devuelto:

El entero de 16 bits que se ha recogido de la cadena.

- `int Puente::desempaqueta_numero_de_32_bits(char *p_elemento,int *indice);`

Descripción:

Este método recoge un número entero de 32 bits de una posición de una cadena formateada según el protocolo de comunicaciones del puerto serie. Se usa para desempaquetar los datos contenidos en las cadenas recibidas por el puerto serie. Estas cadenas siguen el formato del protocolo de comunicaciones que se ha diseñado para el puerto serie. Las cadenas son

recogidas por el servidor y enviadas al cliente a través de la red de forma transparente.

Parámetros:

-La cadena recibida por el puerto serie.

-Un puntero a entero para pasar por referencia el índice para recorrer la cadena.

Valor devuelto:

El número entero de 32 bits que se ha recogido de la cadena.

- `int Puente::calcula_tam(int mascara);`

Descripción:

Esta función calcula el número de caracteres que debe tener una cadena recibida por el puerto serie que contenga el estado del helicóptero. El número de caracteres dependerá de la máscara de estados ya que en función de dicha máscara se enviarán unos campos del estado u otros.

Parámetros:

Un entero que contiene la máscara de estados del helicóptero.

Valor devuelto:

Un entero que contiene el número de caracteres que debe tener una cadena que contenga el estado del helicóptero.

- `void Puente::desempaqueta_mensaje(char *p_mensaje, RESPUESTA *p_respuesta_actual,int mascara);`

Descripción:

Este método desempaqueta un estado del helicóptero que estaba encapsulado en una cadena procedente del CBN y lo introduce en el campo status de la estructura RESPUESTA que se le pase por referencia.

Parámetros:

-La cadena recibida por el puerto serie.

-Un puntero a una estructura RESPUESTA para recoger por referencia el resultado.

-La máscara de estado del helicóptero.

Valor devuelto:

No tiene.

- `int Puente::procesa_caracter_recibido(char caracter,RESPUESTA *p_respuesta_actual);`

Descripción:

Este método rellena uno de los campos de la estructura RESPUESTA que se le pase por referencia con los datos que se van recibiendo carácter a carácter por el puerto serie. Se trata de una máquina de estados que guarda memoria de la última vez que se llamó a la función y continúa en el punto en el que lo dejó.

Parámetros:

-El carácter recibido por el puerto serie.

-Un puntero a una estructura RESPUESTA para recoger por referencia el resultado.

Valor devuelto:

Se devuelve 0 si se consiguió desempaquetar con éxito la cadena y rellenar el campo adecuado de la estructura RESPUESTA, 1 si todavía no se tienen suficientes caracteres para obtener una estructura completa y -1 si hubo error de checksum.

- `void Puente::abrir_fichero_de_logs();`

Descripción:

Este método abre un fichero de logs en el que el investigador puede

almacenar datos sobre el estado del helicóptero que le servirán para la identificación del modelo. Todos los ficheros de log se almacenan en el directorio *log.*Para evitar confundir unos ficheros con otros y llevar un histórico de los experimentos se ha decidido guardar los ficheros de log con el formato: DIA_MES_AÑO-HORA_MINUTO_SEGUNDO.log

Parámetros:
No tiene.

Valor devuelto:
No tiene.

- `void Puente::imprime_estado(ESTADO_DEL_HELICOPTERO estado);`

Descripción:
Este método escribe datos sobre el estado del helicóptero en un fichero de logs con el objetivo de proporcionar datos al investigador que le permitan identificar el modelo del helicóptero. Se ha puesto un límite al tamaño que puede ocupar un fichero de logs para evitar que se llene el disco duro del ordenador de tierra (PST). El tamaño máximo de este fichero de logs se puede configurar a través del fichero de configuración *puente.conf*.

Parámetros:
Una estructura ESTADO_DEL_HELICOPTERO que contiene los datos que se van a registrar en el fichero de logs.

Valor devuelto:
No tiene.

- `void Puente::espera_fin_bucle();`

Descripción:
Este método sirve para bloquear al bucle de control en espera de que el

temporizador dé la orden de inicio de un nuevo periodo del bucle de control. Si en las operaciones realizadas en el bucle de control se supera el tiempo de un periodo de bucle de control, no existe bloqueo. Se pone una cota inferior al tiempo que puede durar una pasada del bucle de control.

Parámetros:
No tiene.

Valor devuelto:
No tiene.

- `void` Puente `::fin_bucle();`

Descripción:
Este método sirve para desbloquear el bucle de control cuando el temporizador dé la orden de inicio de un nuevo periodo del bucle de control.

Parámetros:
No tiene.

Valor devuelto:
No tiene.

- `bool` Puente `::comprueba_timeout_16();`

Descripción:
Este método comprueba si el bucle de control está bloqueado en espera de una lectura de un puerto de 16 bits.

Parámetros:
No tiene.

Valor devuelto:

true (verdadero) si el bucle de control está bloqueado en espera de una lectura de un puerto de 16 bits y falso si no.

- `void` Puente `::esperar_timeout_16();`

  Descripción:
  Este método bloquea al bucle de control en espera de una lectura de un puerto de 16 bits. Se trata de una espera temporizada, esto es, no se esperará un tiempo mayor que el indicado por el *TIMEOUT*.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void` Puente `::deja_de_esperar_timeout_16();`

  Descripción:
  Este método desbloquea al bucle de control si estaba en espera de una lectura de un puerto de 16 bits. Este método se llama cuando expira el temporizador para la lectura, es decir, cuando se alcanza el tiempo de *TIMEOUT*.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `bool` Puente `::comprueba_timeout_32();`

  Descripción:
  Este método comprueba si el bucle de control está bloqueado en espera de

una lectura de un puerto de 32 bits.

Parámetros:
No tiene.

Valor devuelto:
true (verdadero) si el bucle de control está bloqueado en espera de una lectura de un puerto de 32 bits y falso si no.

- `void` Puente `::esperar_timeout_32();`

Descripción:
Este método bloquea al bucle de control en espera de una lectura de un puerto de 32 bits. Se trata de una espera temporizada, esto es, no se esperará un tiempo mayor que el indicado por el *TIMEOUT*.

Parámetros:
No tiene.

Valor devuelto:
No tiene.

- `void` Puente `::deja_de_esperar_timeout_32();`

Descripción:
Este método desbloquea al bucle de control si estaba en espera de una lectura de un puerto de 32 bits. Este método se llama cuando expira el temporizador para la lectura, es decir, cuando se alcanza el tiempo de *TIMEOUT*.

Parámetros:
No tiene.

Valor devuelto:

No tiene.

3.4.9.2.2. Clase Estimated.-

La clase Estimated tiene métodos que son en su mayoría públicos y que ya han sido comentados en el capítulo de métodos orientados al usuario. En este apartado se verán los pocos métodos privados que todavía restan por explicar.

- `void Estimated::meter_estado_tabla(ESTADO_DEL_HELICÓPTERO estado);`

  Descripción:

  Este método introduce un nuevo estado del helicóptero en la tabla de estados del helicóptero, desplazando en una posición a los que había antes e incluso expulsando de la tabla al más antiguo si fuera necesario.

  Parámetros:

  Una estructura ESTADO_DEL_HELICOPTERO que se quiere meter en la tabla.

  Valor devuelto:

  No tiene.

- `void Estimated::inicializa_cola_errores();`

  Descripción:

  Este método inicializa la cola circular de estructuras ERRORES que hay encapsulada dentro de la clase Estimated..

  Parámetros:

  No tiene.

  Valor devuelto:

No tiene.

- `int Estimated::meter_en_cola_errores(ERRORES dato);`

Descripción:
Este método introduce una estructura del tipo ERRORES en la cola circular que hay encapsulada dentro de la clase Estimated.

Parámetros:
Una estructura de tipo ERRORES que se quiere meter en la cola.

Valor devuelto:
Se devuelve 0 si todo va bien y -1 si la cola estaba llena.

3.4.9.2.3. Clase Conexión.-

Esta clase contiene métodos relacionados con las comunicaciones de red: establecimiento de conexiones TCP/IP, envío de cadenas de caracteres y recepción de caracteres individuales.

- `void Conexion::conectar(CONFIG *p_datos_configuracion);`

Descripción:
Este método trata de efectuar una conexión TCP/IP con el programa que se ejecuta en el CAN. La estructura CONFIG que se pasa por referencia contiene el puerto en el que se debe hacer la conexión y la IP del ordenador de control. Estos datos se han obtenido de la lectura de un fichero de configuración *"puente.conf"*.

Parámetros:
Una estructura CONFIG que se pasa por referencia y que contiene el puerto en el que se debe efectuar la conexión y la IP del ordenador de control (CAN).

Valor devuelto:

No tiene.

- `int Conexion::enviar_peticion(PETICION peticion);`

Descripción:

Este método envía hacia el CAN tantos bytes de la cadena contenida en el campo mensaje como indique el campo longitud. La estructura PETICION contiene un comando, los parámetros de este y el resultado de encapsularlos en una cadena formateada para ser enviada por el puerto serie. También contiene la longitud de esta cadena. El tener los datos encapsulados y sin encapsular en la misma estructura obedece a razones de comodidad para la programación.

Parámetros:

Una estructura PETICION que contiene un comando, los parámetros de este y el resultado de encapsularlos en una cadena formateada para ser enviada por el puerto serie. También contiene la longitud de esta cadena. De esta estructura solo se utilizarán en este método los campos mensaje y longitud.

Valor devuelto:

Un entero correspondiente al número de caracteres que se han enviado a través de la red, 0 si se rompe la conexión y -1 si hubo un error..

- `int Conexion::recibir_caracter(char *p_caracter);`

Descripción:

Este método recoge un carácter procedente del CBN y que ha sido recogido por el PC de control y enviado a tierra de forma transparente a través de la red.

Parámetros:

Un puntero a carácter para recoger por referencia el carácter leído.

Valor devuelto:

Un entero correspondiente al número de caracteres leídos si todo va bien, 0 si se rompe la conexión y -1 si hubo un error.

3.4.9.2.4. Otras funciones.-

En este apartado se discutirán funciones que no pertenecen a ninguna clase. Se empezará con las funciones de los hilos:

- `void iniciar_hilo_lectura_ethernet(PUNTEROS *p_punteros);`
- `void *funcion_hilo_lectura_ethernet(void *args);`
- `void destruir_hilo_lectura_ethernet();`
- `void iniciar_hilo_escritura_ethernet(PUNTEROS *p_punteros);`
- `void *funcion_hilo_escritura_ethernet(void *args);`
- `void destruir_hilo_escritura_ethernet();`

Todas estas funciones son similares así que explicaré una de cada tipo:

- `void iniciar_hilo(PUNTEROS *p_punteros);`

Descripción:

Esta función configura los atributos del hilo y efectúa su lanzamiento pasando como parámetro del hilo un puntero a una estructura PUNTEROS. La estructura PUNTEROS contiene punteros a objetos y estructuras que se han declarado como locales en el main y se necesitan en los hilos. Esto es un artificio para evitar el uso de variables globales que resultan poco elegantes y dificultan la modularidad del código.

Parámetros:

Un puntero a una estructura de tipo PUNTEROS para hacer llegar a los hilos objetos y estructuras creados en el main.

Valor devuelto:

No tiene.

- `void *funcion_hilo(void *args);`

  Descripción:

  Esta función es el hilo de ejecución propiamente dicho. Tiene como parámetro un puntero a void que se puede interpretar como se desee para pasarle al hilo cualquier número de parámetros. Este parámetro se ha usado para pasarle al hilo un puntero a una estructura de tipo PUNTEROS que contiene a su vez punteros a cada uno de los objetos y estructuras que se necesitan en el hilo.

  Parámetros:

  Un puntero a void que se debe interpretar adecuadamente para recoger cualquier número de argumentos.

  Valor devuelto:

  No tiene.

- `void destruir_hilo();`

  Descripción:

  Esta función destruye un hilo.

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

  A continuación se desarrollan el resto de funciones:

- `CONFIG configuracion();`

  Descripción:

  Esta función lee el fichero de configuración *"puente.conf"*y rellena una

estructura del tipo CONFIG.

Parámetros:

No tiene.

Valor devuelto:

La estructura de tipo CONFIG que contiene los datos de configuración que se han leído en el fichero.

- `void inicializa_timer(int periodo_ms);`

Descripción:

Esta función arranca un temporizador con un periodo que se le da como parámetro. Cada vez que pase un periodo del temporizador se dará la señal SIGALRM y se llamará a la función envio_periodico que corresponde al manejador que se ha definido para esta señal.

Parámetros:

Un entero que corresponde al periodo del temporizador en milisegundos.

Valor devuelto:

No tiene.

- `void envio_periodico(int param);`

Descripción:

Esta función corresponde al manejador que se ha creado para la señal SIGALRM que lanza el temporizador. Cada vez que pasa un periodo del temporizador se llama a la función envio_periodico. El periodo del temporizador será típicamente de 10 ms. Cada vez que se llama a la función envio_periodico se incrementan unas variables estáticas que al llegar a un valor determinado desbloquean algunas funciones:

-Si pasan 50 (u otro número definido en una constante TIMEOUT) periodos

de temporizador sin que llegue la lectura que se espera de un puerto, se deja de esperar dicha lectura, es decir, ha expirado el TIMEOUT.

--Cada vez que pasa un determinado múltiplo de periodos del temporizador (cuyo número se lee del fichero de configuración*"puente.conf"*) se desbloquea el bucle de control para que dé una nueva pasada.

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador.

Valor devuelto:

No tiene.

- `sighandler_t signal(int ,sighandler_t );`

Descripción:

Instala un manejador para la señal que se le indique como primer parámetro.

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador y una función que manejará dicha señal.

Valor devuelto:

La función que corresponde al manejador instalado.

- `void terminacion(int numsem);`

Descripción:

Esta función cierra el puerto serie, los descriptores de fichero, los descriptores de socket y destruye los hilos. Efectúa una salida del programa de forma ordenada. Funciona también como manejador para las señales SIGINT y SIGTERM que se producen pulsando Ctrl+C o ejecutando el

comando *kill pid* (donde pid es el identificador del proceso que se quiere matar).

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador (cuando esta función se use como manejador).

Valor devuelto:

No tiene.

- `void captura_sigint_y_sigterm (void (*handler)(int));`

Descripción:

Esta función instala un manejador para las señales SIGINT y SIGTERM que se producen pulsando Ctrl+C o ejecutando el comando *kill pid* (donde pid es el identificador del proceso que se quiere matar) respectivamente. El manejador que se instala es la función terminacion.

Parámetros:

Una función que toma como parámetro un número entero y devuelve void, correspondiente al manejador de las señales SIGINT y SIGTERM.

Valor devuelto:

No tiene.

3.4.9.3. Cliente en *modo procesamiento*.-

En el modo de funcionamiento en *modo procesamiento* el bucle de control se encuentra en el servidor y el cliente tiene realiza funciones de supervisión y registro de datos. Por simplicidad se ha definido una única clase que contiene la mayoría de los métodos. Esta clase se llama Procesamiento.

3.4.9.3.1. Clase Procesamiento.-

En un capítulo anterior se explicaron dos métodos orientados al usuario: enviar_cadena y recibir cadena. En este apartado se explicarán el resto de los métodos de la clase, que están más orientados al programador.

- `void Procesamiento::configuracion();`

  Descripción:
  Esta función lee el fichero de configuración *"procesamiento.conf"*y rellena una estructura del tipo CONFIG llamada datos_configuracion, que es privada a la clase Procesamiento

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Procesamiento::conexion();`

  Descripción:
  Este método trata de efectuar una conexión TCP/IP con el programa que se ejecuta en el CAN. La estructura de tipo CONFIG llamada datos configuracion que ha sido declarada como local a la clase Procesamiento contiene el puerto al que se trata de conectar y la IP del ordenador de control. Estos datos se han obtenido de la lectura de un fichero de configuración *"procesamiento.conf"*.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Procesamiento::abrir_fichero_de_logs();`

  Descripción:

  Este método abre un fichero de logs en el que el investigador puede almacenar datos sobre el estado del helicóptero que le servirán para la identificación del modelo. Todos los ficheros de log se almacenan en el directorio *log*.Para evitar confundir unos ficheros con otros y llevar un histórico de los experimentos se ha decidido guardar los ficheros de log con el formato: DIA_MES_AÑO-HORA_MINUTO_SEGUNDO.log

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

- `void` Procesamiento `::imprime_estado(ESTADO_DEL_HELICOPTERO estado);`

  Descripción:

  Este método escribe datos sobre el estado del helicóptero en un fichero de logs con el objetivo de proporcionar datos al investigador que le permitan identificar el modelo del helicóptero. Se ha puesto un límite al tamaño que puede ocupar un fichero de logs para evitar que se llene el disco duro del ordenador de tierra (PST). Este tamaño es configurable mediante el fichero de configuración *procesamiento.conf*.

  Parámetros:

  Una estructura ESTADO_DEL_HELICOPTERO que contiene los datos que se van a registrar en el fichero de logs.

  Valor devuelto:

  No tiene.

- `void` Procesamiento`::espera_fin_bucle();`

  Descripción:
  Este método sirve para bloquear al bucle del hilo principal en espera de que el temporizador dé la orden de inicio de un nuevo periodo. Si en las operaciones realizadas en el bucle se supera el tiempo de un periodo de bucle , no existe bloqueo. Se pone una cota inferior al tiempo que puede durar una pasada del bucle .

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void` Procesamiento`::fin_bucle();`

  Descripción:
  Este método sirve para desbloquear el bucle del hilo principal cuando el temporizador dé la orden de inicio de un nuevo periodo del bucle.

  Parámetros:
  No tiene.

  Valor devuelto:
  No tiene.

- `void Procesamiento::escribe_error_fichero(short int error_code);`

  Descripción:
  Este método abre un fichero de errores si no estaba abierto y escribe en él el código de error que se le pase. Cuando el tamaño del fichero de errores

sobrepasa un valor determinado, éste se cierra.

Parámetros:

Un entero de 16 bits correspondiente al código del error que se ha producido.

Valor devuelto:

No tiene.

- `void Procesamiento::escritura_ethernet();`

Descripción:

Este método saca una estructura MENSAJE de la cola de mensajes para enviar, la encapsula en una cadena que empieza por el carácter 'm' y continua con un carácter de desbloqueo ('s' o 'n') y envía la cadena resultante al servidor.

Parámetros:

No tiene

Valor devuelto:

No tiene.

- `void Procesamiento::lectura_ethernet();`

Descripción:

Este método recoge una cadena procedente del servidor y la interpreta convenientemente:

-Si la cadena recogida empieza por el carácter 's' se trata de un mensaje de estado del helicóptero y, por consiguiente, se registra en el fichero de logs.

-Si empieza por el carácter 'e' se trata de un mensaje de error y se escribe en el fichero de errores.

-Si empieza por el carácter 'm' se trata de un mensaje del usuarioy, por lo

tanto, se introduce en la cola de mensajes recibidos para hacerlo llegar al bucle del hilo principal que será el encargado de interpretarlo.

Parámetros:
No tiene

Valor devuelto:
No tiene.

Métodos privados:

- `void Procesamiento::inicializa_cola_msg_para_enviar();`

Descripción:
Este método inicializa la cola circular de estructuras MENSAJE que se utiliza para comunicar el hilo principal y el hilo de escritura ethernet.

Parámetros:
No tiene

Valor devuelto:
No tiene.

- `void Procesamiento::inicializa_cola_msg_recibido();`

Descripción:
Este método inicializa la cola circular de estructuras MENSAJE que se utiliza para comunicar el hilo de lectura ethernet y el hilo principal.

Parámetros:
No tiene

Valor devuelto:
No tiene.

- `int Procesamiento::sacar_msg_para_enviar(MENSAJE *dato);`

  Descripción:

  Este método saca una estructura de tipo MENSAJE de la cola circular correspondiente para que pueda ser enviada al PC de control.

  Parámetros:

  Un puntero a una estructura de tipo MENSAJE para recoger por referencia el mensaje que se quiere sacar de la cola.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Procesamiento::sacar_msg_recibido(MENSAJE *dato);`

  Descripción:

  Este método saca una estructura de tipo MENSAJE de la cola circular correspondiente para que pueda ser interpretada por el bucle del hilo principal. El MENSAJE procede del ordenador de control, el CAN.

  Parámetros:

  Un puntero a una estructura de tipo MENSAJE para recoger por referencia el mensaje que se quiere sacar de la cola.

  Valor devuelto:

  Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Procesamiento::meter_msg_para_enviar(MENSAJE dato);`

  Descripción:

  Este método introduce una estructura de tipo MENSAJE en la cola circular correspondiente para que pueda ser enviada al PC de control.

Parámetros:

La estructura de tipo MENSAJE que se quiere enviar al servidor.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

- `int Procesamiento::meter_msg_recibido(MENSAJE dato);`

Descripción:

Este método introduce una estructura de tipo MENSAJE en la cola circular correspondiente para hacérsela llegar al bucle del hilo principal que la interpretará adecuadamente.

Parámetros:

La estructura de tipo MENSAJE que se ha recibido del servidor y se quiere poner a disposición del bucle del hilo principal.

Valor devuelto:

Devuelve 0 si todo va bien y -1 si la cola estaba vacía.

3.4.9.3.2. Otras funciones.-

En este apartado se discutirán funciones que no pertenecen a ninguna clase. Se empezará con las funciones de los hilos:

- `void iniciar_hilo_lectura_ethernet(Procesamiento *p_procesamiento);`
- `void *funcion_hilo_lectura_ethernet(void *args);`
- `void destruir_hilo_lectura_ethernet();`

- `void iniciar_hilo_escritura_ethernet(Procesamiento *p_procesamiento);`
- `void *funcion_hilo_escritura_ethernet(void *args);`
- `void destruir_hilo_escritura_ethernet();`

Todas estas funciones son similares así que explicaré una de cada tipo:

- `void iniciar_hilo(Procesamiento *p_procesamiento);`

  Descripción:

  Esta función configura los atributos del hilo y efectúa su lanzamiento pasando como parámetro del hilo un puntero a un objeto del tipo Procesamiento que tiene encapsuladas en su interior todas las variables y parámetros que requiere el hilo. Esto es un artificio para evitar el uso de variables globales que resultan poco elegantes y dificultan la modularidad del código.

  Parámetros:

  Un puntero a un objeto Procesamiento que se ha creado localmente en el main.

  Valor devuelto:

  No tiene.

- `void *funcion_hilo(void *args);`

  Descripción:

  Esta función es el hilo de ejecución propiamente dicho. Tiene como parámetro un puntero a void que se puede interpretar como se desee para pasarle al hilo cualquier número de argumentos. Este parámetro se ha usado para pasarle al hilo un puntero a un objeto de tipo Procesamiento que tiene encapsulados en su interior todas las variables y parámetros que se requieren en el hilo.

  Parámetros:

  Un puntero a void que se debe interpretar adecuadamente para recoger cualquier número y tipo de argumentos.

  Valor devuelto:

  No tiene.

- `void destruir_hilo();`

  Descripción:

  Esta función destruye un hilo.

  Parámetros:

  No tiene.

  Valor devuelto:

  No tiene.

  A continuación se desarrollan el resto de funciones:

- `void captura_sigint_y_sigterm (void (*handler)(int));`

  Descripción:

  Esta función instala un manejador para las señales SIGINT y SIGTERM que se producen pulsando Ctrl+C o ejecutando el comando *kill pid* (donde pid es el identificador del proceso que se quiere matar) respectivamente. El manejador que se instala es la función terminacion.

  Parámetros:

  Una función que toma como parámetro un número entero y devuelve void, correspondiente al manejador de las señales SIGINT y SIGTERM.

  Valor devuelto:

  No tiene.

- `void terminacion(int numsem);`

  Descripción:

  Esta función cierra los descriptores de fichero, los descriptores de socket y destruye los hilos. Efectúa una salida del programa de forma ordenada. Funciona también como manejador para las señales SIGINT y SIGTERM que se producen pulsando Ctrl+C o ejecutando el comando *kill pid* (donde

pid es el identificador del proceso que se quiere matar).

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador (cuando esta función se use como manejador).

Valor devuelto:

No tiene.

- `void inicializa_timer(int periodo_ms,Procesamiento *p_procesamiento);`

  Descripción:

  Esta función arranca un temporizador con un periodo que se le da como parámetro. Cada vez que pase un periodo del temporizador se dará la señal SIGALRM y se llamará a la función envio_periodico que corresponde al manejador que se ha definido para esta señal.

  Parámetros:

  Un entero que corresponde al periodo del temporizador en milisegundos y un puntero a un objeto de tipo Procesamiento que encapsula la variable *fin* de la aplicación.

  Valor devuelto:

  No tiene.

- `void envio_periodico(int param);`

  Descripción:

  Esta función corresponde al manejador que se ha creado para la señal SIGALRM que lanza el temporizador. Cada vez que pasa un periodo del temporizador se llama a la función envio_periodico. El periodo del temporizador será típicamente de 10 ms. Cada vez que se llama a la función envio_periodico se incrementan una variable estática que al llegar a un

valor determinado desbloquea la función espera_fin_bucle. Cada vez que pasa un determinado múltiplo de periodos del temporizador (cuyo número se lee del fichero de configuración *"procesamiento"*) se desbloquea el bucle de hilo principal para que dé una nueva pasada.

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador.

Valor devuelto:

No tiene.

- `sighandler_t signal(int ,sighandler_t );`

Descripción:

Instala un manejador para la señal que se le indique como primer parámetro.

Parámetros:

Un entero que corresponde al código de la señal que corresponde al manejador y una función que manejará dicha señal.

Valor devuelto:

La función que corresponde al manejador instalado.