Capítulo 3

Sonares en el PC

1.Introducción

Uno de los problemas que se tienen a la hora de utilizar los sonares son las interferencias mutuas que se producen cuando se disparan a la vez sonares que se encuentran cercanos. Por lo tanto para disminuir esa interferencia todo lo posible es necesario algún tipo de mecanismo de sincronismo de tal forma que no se disparen sonares que estén cercanos a la vez.

En el capítulo anterior del presente proyecto fin de carrera se han comentado varios métodos de sincronismo vía hardware. Uno de ellos consistía en una opción que traen los sonares BERO de Siemens, que tras programarlos y conectar las señales de sincronismo de estos se consigue que al activar el grupo de sonares, estos se disparen de forma secuencial y por lo tanto disminuya la interferencia mutua. Por otra parte también se indicó que en la caja de los sonares utilizando la propiedad anteriormente comentada de los sonares BERO, se podían sincronizar los sonares en 4 grupos a partir de la conexión o desconexión de una serie de jumpers que se encontraban en la propia caja de los sonares. Sin embargo estos 2 métodos tienen el problema de que son poco flexibles y dinámicos, es decir, que no se pueden cambiar las estrategias de sincronización en tiempo real y además para realizarlo hay que reprogramar los sonares y cambiar su cableado o cambiar los jumpers de la caja de los sonares. Por lo tanto con estos métodos sólo se pueden realizar estrategias de sincronismo estáticas para toda la duración de la aplicación.

Por ello para conseguir esa flexibilidad y dinamismo se va a realizar la sincronización por software, de tal forma que los sonares se dividen en grupos donde los sonares de un mismo grupo se activan de forma simultánea mientras los demás sonares están apagados. Tras un tiempo (Capítulo 2 Apartado 2.1.4), calculado para poder detectar un objeto en el peor de los casos, se desactivan los sonares de ese grupo y se continúa con el siguiente grupo. Esta opción tiene la ventaja que los grupos pueden ser cambiados de forma dinámica y en tiempo real por software, además ofrece una flexibilidad completa a conveniencia del programador de la aplicación.

El código para gestionar el sincronismo de los sonares ha sido introducido en la arquitectura software que existía en el controlador romeo4b basado en PC, realizándose las oportunas modificaciones pero siempre teniendo en cuenta la compatibilidad del nuevo código con el ya existente.

2.Arquitectura software para el controlador romeo4b

Esta arquitectura fue definida en el proyecto fin de carrera "Control automático de un vehículo autónomo bajo el sistema operativo GNU/Linux. Implementación de drivers y software de control" realizado por Rafael Martín Agar Tirado y en este apartado se quiere dar una visión general que permita conocer el entorno en el que fue programado la gestión del sincronismo de los sonares.

2.1.Niveles de abstracción

El software está estructurado en capas, donde la capa inferior tiene un menor nivel de abstracción y la capa superior tiene el mayor nivel de abstracción. La capa inferior es lo que normalmente se llama como "software de bajo nivel" y este es el software para el cual es necesario conocer el funcionamiento de los dispositivos hardware. Sin embargo la capa superior forma el "software de alto nivel" y que permite el uso del software sin tener que saber como funcionan los distintos dispositivos hardware. Este "software de alto nivel" es ideal para un sistema que va a ser utilizado por distintos usuarios como sucede en ROMEO-4R.

Las distintas capas del software implementado en el controlador romeo4b son (Figura 3.1):

1. **ROMEO-4R:** es el nivel más bajo y para poder trabar a este nivel es necesario conocer toda la electrónica del vehículo.

2. **Nivel de driver:** es el primer nivel software y para utilizarlo es necesario conocer la interfaz del dispositivo con el PC. Esta capa es totalmente dependiente del dispositivo y además sigue siendo necesario un alto nivel de conocimiento del hardware del sistema. Por otra parte el software de este nivel es totalmente distinto para cada dispositivo.

3. **Nivel de dispositivo:** es la siguiente capa software y ya en ella no hace falta conocer el funcionamiento interno de los dispositivos. Además esta capa proporciona una interfaz genérica para que las funciones implementadas puedan ser utilizadas por cualquier aplicación de más alto nivel.

4. **Nivel de aplicación:** esta capa se encuentra íntimamente relacionada con la aplicación concreta para la que se va a utilizar el vehículo. En esta capa estarán implementadas todas aquellas funciones que el usuario va a utilizar y que van a formar la interfaz de usuario. Por lo tanto un usuario de alto nivel sólo necesita conocer esta interfaz para poder utilizar todos los dispositivos del sistema.

5. **Programa de usuario:** es el lugar donde el usuario debe desarrollar su aplicación utilizando la interfaz que le proporciona el nivel de aplicación.

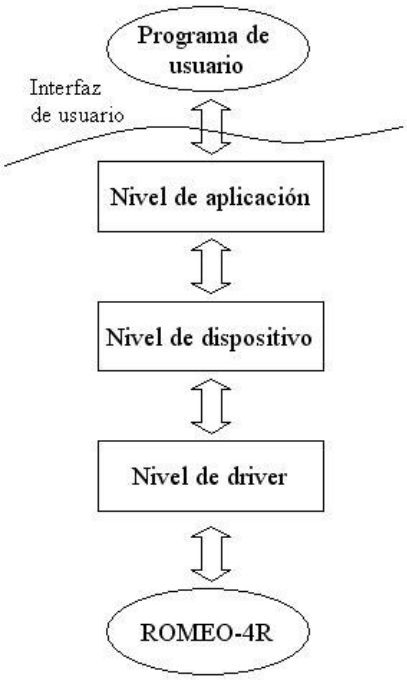

Figura 3.1:Niveles de abstracción software

2.1.1.Nivel de driver

En este nivel sólo se va a comentar brevemente la parte relacionada con los sonares debido a que este nivel es muy específico y se considera que el resto del nivel no es objeto de este proyecto.

El nivel de driver relacionado con los sonares es la implementación del driver de la tarjeta AX5411 que ya se comentó en el capítulo anterior (apartado 2.3.2.1). El código de este driver se encuentra en el directorio *ax5411/driver* dentro del directorio general del software para el controlador. En este directorio se puede encontrar el código fuente del driver escrito en C y un fichero llamado *guia_usuario.txt* donde se explica detalladamente como instalar el driver, acceder al dispositivo, escribir y leer tanto señales digitales como analógicas.

2.1.2.Nivel de dispositivo

Este nivel está escrito en C++ y se divide en una serie de clases que presentan al siguiente nivel una interfaz para el manejo de los distintos dispositivos. Las clases que se han definido en este nivel son:

1. **Clase DCX:** está asociada a la tarjeta DCX que se utiliza para el control de los motores de tracción y dirección del vehículo.

2. **Clase Puerto_serie:** esta clase implementa las funciones para el manejo del puerto serie.
3. **Clase Gps:** esta clase hereda de la clase Puerto_serie, ya que el GPS es un dispositivo que se comunica con el PC por medio del puerto serie.
4. **Clase Giroscopo:** este dispositivo también se comunica a través del puerto serie por lo que también heredará de la clase Puerto_serie.
5. **Clase Laser:** implementa las funciones para el manejo del láser escáner LMS, y como también este dispositivo se comunica por el puerto serie con el controlador también hereda de la clase Puerto_serie.
6. **Clase AX5411:** implementa las funciones para comunicarse con la tarjeta AX5411, a continuación se van a explicar las métodos públicos que forman la interfaz ofrecida a la capa superior.
    a. ***open_ax5411*:** abre el archivo del driver de la AX5411 que normalmente se encuentra en /dev/ax5411 en modo lectura y escritura. Guarda el descriptor del fichero en ax5411_handle.

        Parámetros:

        - char *file: ruta del archivo de la AX5411 si se omite se toma por defecto.

        Devuelve:

        - int: SUCCESS si no hay errores o ERROR en caso contrario.

    b. ***close_ax5411:*** cierra el archivo del driver de la AX5411.

        Parámetros:

        - void.

        Devuelve:

        - int: SUCCESS si no hay errores o ERROR en caso contrario.

    c. ***read_analog_input_ax5411:*** devuelve el valor de una entrada analógica. Lo único que hace es devolver el valor de la entrada analógica deseada que se encuentra almacenado en la variable miembro "lectura", puesto que esta variable guarda la última lectura que se hizo de la tarjeta.

Parámetros:

- int i: entrada analógica que queremos leer.

Devuelve:

- int: el valor de dicha entrada, o ERROR si nos salimos de rango.

d. ***read_digital_input_ax5411:*** devuelve el valor de una entrada digital. De la misma manera que en el caso analógico, se devuelve el valor de la entrada deseada que tenemos en la variable miembro "lectura".

Parámetros:

- int i: entrada digital que queremos leer.

Devuelve:

- int: valor de dicha entrada o ERROR si nos salimos de rango.

e. ***update_all_input_ax5411:*** actualiza el valor de la variable miembro "lectura" realizando una lectura de todas las entradas. Si queremos acceder a un valor "reciente" de alguna entrada, antes de llamar las 2 funciones anteriores, debemos llamar a esta función, para actualizar el valor de la variable "lectura".

Parámetros:

- void.

Devuelve:

- int: aunque no se devuelve nada.

f. ***write_analog_output_ax5411:*** escribe un valor en una salida analógica, comprobando que no nos salimos de rango.

Parámetros:

- int i: salida en la que queremos escribir.
- int value: valor que queremos escribir.

Devuelve:

- int: ERROR si nos salimos del rango de las salidas analógicas.

g. ***write_digital_output_ax5411:*** escribe un valor en una salida digital comprobando que no nos salimos de rango.

Parámetros:

- int i: salida digital en la que queremos escribir.
- int value: valor que queremos escribir.

Devuelve:

- int: ERROR si nos salimos del rango de las salidas digitales.

h. ***set_range_ax5411:*** establece el rango de entradas para la conversión A/D.

Parámetros:

- int min: límite inferior del rango.
- int max: límite superior del rango.

Devuelve:

- int: siempre devuelve SUCCESS.

i. ***set_gain_ax5411:*** establece la ganancia.

Parámetros:

- int gain: ganancia que queremos poner.

Devuelve:

- int: siempre devuelve SUCCESS.

2.1.3.Nivel de aplicación

Este nivel se sirve de las interfaces ofrecidas por el nivel de dispositivo y esta es la capa que tiene una relación directa con el programa de usuario. El nivel de aplicación consta de una sola clase denominada Romeo. Esta clase hereda las funciones de todas las clases del nivel de dispositivo, en la Figura 3.2 se puede observar la jerarquía de clases que está implementada en este nivel.

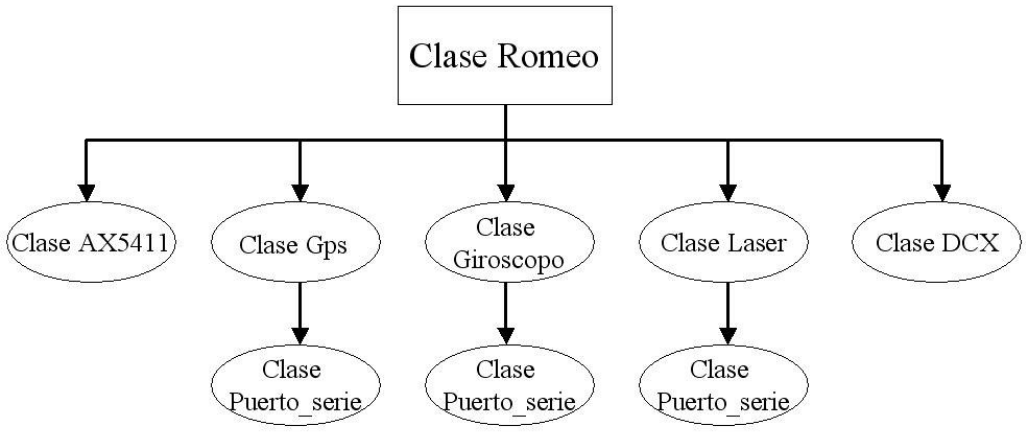

Figura 3.2:Jerarquía de clases implementadas en los niveles de aplicación y dispositivo

Por lo tanto la clase Romeo dispondrá de los métodos propios definidos en la clase Romeo y los métodos definidos públicos y que forman las diferentes interfaces de las clases que hereda (AX5411, Gps, Giroscopo, Laser y DCX).

Las funciones propias de la clase Romeo se dividen en la relación que tienen con los diferentes dispositivos, aunque sólo se van a comentar las relacionadas con la clase AX5411 al ser las de interés para el presente proyecto.

1. **Funciones relacionadas con la clase AX5411:**

    a. ***update_ax5411:*** actualiza las medidas de los sonares a partir de la lectura de los valores procedentes de la AX5411.

    Parámetros:

    - void.

    Devuelve:

    - int: ERROR si se ha producido algún fallo o SUCCESS en caso contrario.

    b. act_sonar: activa un sonar, es decir, pone la señal de disparo de un sonar a nivel alto.

    Parámetros:

    - int i: sonar que queremos activar (numerados del 1 al 12).

Devuelve:

- void.

c. ***deact_sonar:*** desactiva un sonar, es decir, pone la señal de disparo de un sonar a nivel bajo.

Parámetros:

- int i: sonar que queremos desactivar (numerados del 1 al 12).

Devuelve:

- void.

d. ***get_ax5411_data:*** devuelve una estructura del tipo AX5411_DATA que contiene una tabla con los valores leídos para cada sonar y el instante de tiempo en el que se realizó esa lectura.

Parámetros:

- int index: indica a que elemento AX5411_DATA queremos acceder de la cola circular, un 0 indica el último elemento introducido (el más reciente), un 1 el siguiente, ...

Devuelve:

- AX5411_DATA: estructura de datos de la AX5411 ya comentada.

e. ***new_ax5411_data:*** indica si se ha realizado una nueva estimación de datos por la AX5411 desde la última vez que se consultaron.

Parámetros:

- void.

Devuelve:

- int: 1 si existen nuevos datos y 0 en caso contrario.

f. ***read_sonar:*** devuelve el valor obtenido de un sonar en metros. Esta función se utiliza en update_ax5411.

Parámetros:

- int sonar: sonar del que queremos conocer su valor.

Devuelve:

- double: valor leído por el sonar en metros.

2.1.4.Estructura general del programa

En la estructura general del programa se pueden distinguir 2 partes bien diferenciadas:

1. **Programa principal o Main:** será aquí donde se va a introducir el algoritmo de control y cuya estructura se describirá detalladamente en el Capítulo 5.
2. **Hilos de ejecución adyacentes:** son una serie de hilos que se ejecutan de forma paralela al programa principal. Se distinguen 2 tipos de hilos:
   a. ***Hilos correspondiente a cada uno de los dispositivos:*** la función fundamental de estos hilos es la adquisición de datos de cada uno de los dispositivos. Llaman constantemente a la función "*update*" de la clase Romeo asociada a cada dispositivo en concreto. Normalmente al leer los datos, estos se procesan y se almacenan en una cola circular de datos del dispositivo que también pertenece a la clase Romeo. Por otra parte se activa una bandera para hacer saber al programa de control que se han leídos nuevos datos. Los hilos que corresponden a este grupo son:
      - Hilo de la DCX.
      - Hilo de la AX5411
      - Hilo del Gps
      - Hilo del Giróscopo
      - Hilo del Láser LMS
   b. ***Hilo de estimación:*** este grupo está formado por un único hilo que se encarga de realizar las estimaciones de posición y orientación para el vehículo ROMEO-4R a partir de las medidas recibidas de los dispositivos. En este hilo se llama continuamente a la función *update_estimacion* que es la función que a partir de los datos de diferentes dispositivos calcula la posición y orientación del vehículo.

La estructura del programa se ha organizado de esta modo para conseguir una mayor eficiencia del mismo y conseguir una independencia entre el algoritmo de control y el resto de tareas de tal forma que si alguna de ellas es muy lenta no se vean afectadas ni el resto de ellas ni el algoritmo de control.

3.Sincronización de los sonares por software

Una vez que ya se ha comentado la estructura del software para el controlador romeo4b y en más detalle los distintos niveles asociados a los sonares, se van a explicar las modificaciones que se introdujeron en el código para poder realizar la sincronización de estos por software, que como ya se ha comentado es mejor que las otras opciones, sobre todo por ser más flexible y poder cambiarse entre distintas estrategias de sincronización en tiempo real. Recordar que antes de realizar estas modificaciones los sonares se activaban todos a la vez y por lo tanto la interferencia mutua entre ellos era importante.

3.1.Algoritmo de sincronización

El algoritmo de sincronización implementado se basa en la creación de una serie de grupos de sonares que se van a activar de forma simultánea, es decir, que todos los sonares que forman parte de un grupo se activarán a la vez. Por lo tanto los sonares que estén muy juntos o haya peligro de interferencia mutua deben estar en distintos grupos.

Por otra parte una vez que los sonares de un grupo se activan, mientras lo sonares de los demás grupos está desactivados, se espera un tiempo de disponibilidad (Capítulo 2 Apartado 2.1.4) calculado para poder detectar un objeto en el peor de los casos, y a continuación se desactivan los sonares del grupo antes de activar los del grupo siguiente, para que no se produzca ningún tipo de interferencia. Así hasta recorrer todos los grupos de sonares que haya.

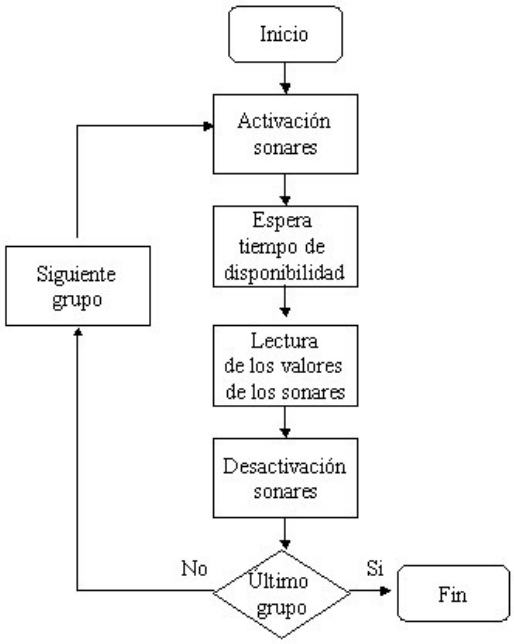

Figura 3.3: Algoritmo de sincronización

3.1.1.Implementación software de los grupos de sonares

Los grupos de sonares están implementados en una tabla bidimensional llamada *cadena_sonares* que es una variable de la clase Romeo asociada al dispositivo AX5411, en ella se encuentran los números de los sonares que forman cada grupo (numerados del 1 al 12) y siempre el último sonar debe ser uno ficticio con el número 0. Además el último grupo debe ser uno ficticio con un único sonar con el número 0. Esto se ha hecho de esta forma para que se sepa de una forma sencilla cual es el último sonar de cada grupo y cual es el último grupo, aunque como se verá más adelante esto es transparente al usuario que maneja los grupos de sincronización.

Por otra parte es importante decir que esta tabla tiene tamaño NUM_SONAR+1 en sus 2 dimensiones, siendo NUM_SONAR el número de sonares que hay (actualmente tiene valor 12 y está definido en el archivo *romeo4*.h). Esto implica que no puede haber más de NUM_SONAR grupos aunque el tamaño sea NUM_SONAR+1, ya que siempre debe haber un último grupo ficticio con un único sonar que sea el 0 y además no puede haber más de NUM_SONAR sonares por grupo, también porque siempre debe haber un último sonar ficticio de número 0 como ya se ha comentado anteriormente. Aunque parezca una limitación se ha dejado así debido a que no se ve utilidad que haya más grupos que sonares ya que si entre cada grupo se tiene que esperar un tiempo de disponibilidad de 380ms, incluso en el caso de NUM_SONAR grupos el bucle de actualización de los valores de los sonares sería de 4.56 segundos (380ms*12) lo cual es excesivo para cualquier estrategia de control que se quiera implementar utilizando los sonares. Por otra parte que en un mismo grupo haya sonares repetidos no tiene sentido ya que el sonar sólo se va a activar una vez y se va a leer un único dato por grupo y esta sería la única forma de que hubiera más de NUM_SONAR sonares en cada grupo. Por lo tanto se ve claramente que el tamaño elegido para *cadena_sonares* no implica realmente ningún tipo de limitación ya que nuca se va a necesitar un tamaño mayor.

3.1.2.Modificaciones realizadas en la estructura AX5411_DATA

Tras las modificaciones realizadas la estructura de datos AX5411_DATA está compuesta por las siguientes variables:

1. ***double sonar[NUM_SONAR]:*** tabla en la que se almacena los valores obtenidos de los distintos sonares.
2. ***double beta_remolque:*** valor obtenido del sensor que mide el ángulo del remolque.
3. ***double time:*** tiempo en el que realizaron las medidas de los sonares.

4. ***double time_groups[NUM_SONAR]:*** tiempo en el que se realizaron las medidas de los distintos grupos de sonares.

3.2.Modificación de la función *update_ax5411*

Esta función ha sido modificada para que se activen los sonares según la estrategia de sincronización que se haya implementado. Para que haya compatibilidad con códigos anteriores, en el caso de que no haya ningún grupo de sincronización, es decir, que el primer sonar del primer grupo de activación valga 0, se ejecuta el código antiguo de esta función que activaba todos los sonares a la vez. Si no se da este caso se activan los sonares por grupos, se esperan los 380ms de tiempo de disponibilidad, se leen los valores de los sonares correspondientes y por último se desactivan los sonares del grupo. Este proceso se repite con cada uno de los grupos de sincronización que haya. Además en ambos casos siempre se lee en primer lugar el sensor del remolque ya que este no está implementado con sonares y no interfiere con las demás medidas.

Por último comentar que no se ha cambiado la declaración de la función, por lo tanto la función tiene los mismos parámetros y devuelve el mismo tipo de variable con el mismo significado.

3.3.Nuevas funciones implementadas

En primer lugar hay que decir que todas las nuevas funciones que se han implementado, se han hecho en el nivel de aplicación y más concretamente dentro de las funciones propias de la clase Romeo relacionadas con la AX5411.

Estas nuevas funciones tienen como objetivo poder gestionar de forma dinámica y en tiempo real los grupos de sincronización. Por lo tanto con estas funciones no sólo se van a poder eliminar e insertar sonares dentro de un grupo sino que además se van a poder crear nuevos grupos y eliminar otros ya existentes.

Estas nuevas funciones son:

1. ***erase_all_groups:*** borra todos los grupos de sincronización de los sonares. Hay que tener en cuenta que si después de ejecutar esta función no se crea ningún grupo se ejecutará el código antiguo de la función *update_ax5411* y por lo tanto todos los sonares se activarán a la vez.

   Parámetros:

   - void.

Devuelve:

- void.

2. ***erase_group:*** borra el grupo de sincronización especificado.

   Parámetros:

   - int group: número del grupo a borrar. Hay que tener en cuenta que el primer grupo es el número 1.

   Devuelve:

   - int: código de error. Si vale 0 indica que no ha habido ningún error y si vale -1 indica que el número de grupo no está dentro del rango válido.

3. ***create_group:*** crea un grupo de sincronización el cual es añadido al final.

   Parámetros:

   - int first_sonar: número del primer sonar que va a formar parte del grupo que se va a crear, se recuerda que los sonares están numerados del 1 al 12.

   Devuelve:

   - int: código de error. Si vale 0 indica que no ha habido errores y si vale -2 indica que el número del sonar no está dentro del rango válido.

4. ***put_sonar_in_group:*** añade un sonar al final del grupo de sincronización especificado.

   Parámetros:

   - int group: número del grupo de sincronización al que se quiere añadir el sonar, teniendo en cuenta que el primer grupo es el número 1.
   - int sonar_number: número del sonar a añadir (numerados del 1 al 12).

   Devuelve:

   - int: código de error. Si vale 0 indica que no ha habido errores, si vale -1 indica que el número del grupo no está dentro del rango válido y si vale -2 indica que el número del sonar no está dentro del rango válido.

5. ***erase_sonar_group:*** elimina un sonar del grupo de sincronización especificado.

Parámetros:

- int group: número del grupo de sincronización del que se quiere eliminar el sonar, teniendo en cuenta que el primer grupo es el número 1.
- int sonar_number: número del sonar a eliminar (numerados del 1 al 12).

Devuelve:

- int: código de error. Si vale 0 indica que no ha habido errores, si vale -1 indica que el número del grupo no está dentro del rango válido, si vale -2 indica que el número del sonar no está dentro del rango válido y si vale -3 indica que el sonar especificado no pertenece al grupo de sincronización y por lo tanto no se ha podido borrar.

6. ***finaliza_sonares:*** función que desactiva todos los sonares cuando se finaliza el hilo asociado a estos. Este hilo es el que llama constantemente a *update_ax5411*.

   Parámetros:

   - void.

   Devuelve:

   - void.

7. ***copy_cadena_sonares:*** función que es transparente al programador de algoritmos de control. Copia a partir de la estructura donde se encuentran los datos obtenidos del fichero de configuración, los distintos grupos de sincronización en la tabla *cadena_sonares*. En el caso de que los grupos o los sonares no estén dentro del rango válido o que el número de grupos no coincida con el especificado se finalizará el programa y se sacará por pantalla el correspondiente mensaje de error.

   Parámetros:

   - conf_struct conf_info:estructura donde se guarda la configuración inicial, leída del fichero *romeo.conf*.

   Devuelve:

   - void.

3.4.Utilización del archivo de configuración *romeo.conf*

En este apartado se va a explicar una forma alternativa de crear inicialmente los grupos de sincronización, a la utilización de las funciones anteriormente comentadas, y que se basa en el uso del fichero de configuración *romeo.conf*.

Este fichero permite inicializar ciertas ciertas variables del vehículo como la velocidad y la orientación, también permite indicar el periodo del bucle de control y que dispositivos se quieren monitorizar. Un ejemplo del contenido del fichero se muestra a continuación:

```
// Fichero de configuracion del programa de control de ROMEO 4R.
//
// En aquellas preguntas que se contesten con sí o no:
// - 1 -> sí.
// - 2 -> no.
//
//



0.0 // Velocidad inicial (en m/s.)
0.0 // Curvatura inicial (en m/s.)

500000 // Periodo del bucle de control (en us.)

0   // ¿Usar GPS en estimacion posicion?
1   // ¿Usar GYRO en estimación posición?
0   // ¿Usar Láser?
0   // ¿Activación de motores?

1   // ¿Monitorizar DCX?
1   // ¿Monitorizar AX5411?
1   // ¿Monitorizar GPS?
1   // ¿Monitorizar GYRO?
1   // ¿Monitorizar ESTIMATED?

0   // Simulación?

0   // Teleoperación?

2         //Numero de grupos de sonares
1 3 5 0   //Sonares que forman cada grupo
2 4 6 0   //Ver "sonares.txt" para entender el formato
0
```

Además como se puede observar permite definir los grupos de sincronización de los sonares según una sintaxis determinada que está explicada en el fichero sonares.txt que se encuentra en el mismo directorio que el código fuente. El contenido de este fichero es:

```
Vamos a explicar el formato de la parte de los sonares del fichero de
configuración con un ejemplo.

Parte que nos interesa del fichero "romeo.conf":

4     //Numero de grupos de sonares
1 4 7 0 //Sonares que forman cada grupo
```

```
2 5 8 0 //Ver "sonares.txt" para entender el formato
3 6 9 0
10 0
0

Como se puede ver en la primera línea indicamos el número de grupos que va
a haber en total.
En las siguientes líneas se especifica los sonares que queremos que se
activen en cada turno (o grupo) y siempre debemos terminar la línea con el
número 0.
Además cuando se han especificado todos los grupos se debe poner un único
0 en la última línea.

Si queremos que todos los sonares se activen a la vez (que es lo que se
hacía hasta esta modificación), es decir como si sólo hubiera un único
turno (o grupo), se debe poner en el número de grupos un 0, de esta forma
se ejecutará el código que había antes (para mantener compatibilidades).
También se puede poner un único grupo con todos los sonares en ese grupo
pero eso no asegura que los algoritmos anteriores a esta modificación
vayan a funcionar bien.

AVISO: no se pueden poner un número mayor de grupos que el número de
sonares especificado en la variable NUM_SONAR.
```

La utilización de esta forma de inicializar los grupos de sincronización de los sonares tiene las siguientes ventajas:

1. Se inicializan los grupos de sincronización de los sonares sin escribir una sola línea de código.
2. Además después se pueden modificar estos grupos utilizando las mismas funciones que se han explicado anteriormente.
3. Si no se van a realizar cambios en los grupos de sincronización, permite configurar los grupos de los sonares sin tener que conocer el código implementado para la gestión de estos.

Al leer los datos de los grupos de sincronización del fichero de configuración se comprueba si el número de grupos y el de sonares está dentro del rango válido y que el número de grupos coincide con el especificado y si no es así el programa finaliza inmediatamente sacando por pantalla el correspondiente mensaje de error. Esto se ha hecho así para evitar en lo posible errores en la configuración inicial de los sonares. Sin embargo en las funciones de gestión de sincronización si hay algún tipo de error, este se devuelve en forma de código de error y debe ser el programador de la aplicación el que gestione los errores, además si hay algún error el programa no finaliza.

3.5.Funcionamiento general

Básicamente las únicas funciones que debe conocer la persona que utilice los sonares para la realización de un algoritmo de control son *new_ax5411_data*, *get_ax5411_data* y las funciones de gestión de los grupos de sincronización en el caso que los quiera cambiar durante el transcurso de su programa. Las demás funciones no hace falta utilizarlas debido a que, como se comentó anteriormente, existe un hilo encargado de llamar constantemente a la función *update_ax5411* y esta función, como se ha visto, se encarga de actualizar los valores de los sonares de acuerdo a los grupos de sincronización existentes.

Por lo tanto se deberá utilizar la función *new_ax5411_data* para ver si hay nuevos datos y si es así con la función *get_ax5411_data* guardarlos en una estructura *AX5411_DATA* y por último accediendo a los distintos campos de la estructura obtener el valor de las variables que estamos interesados para nuestro algoritmo de control.

Hay otra opción que es utilizar la función *get_estimated_status* que devuelve una estructura del tipo ESTIMATED y donde se encuentra almacenado el estado del vehículo incluyendo los valores de los sonares entre otros datos.

4.Conclusiones

En este capítulo se ha explicado las modificaciones necesarias que se han necesitado para implementar el algoritmo de sincronización por software que tiene las siguientes características:

1. Es dinámico y se pueden cambiar los grupos de sincronización en tiempo real de una forma sencilla mediante el uso de las funciones implementadas para la gestión de los grupos.

2. Permite grupos de sincronización con diferentes números de sonares.

3. El programador tiene completa libertad para crear los grupos de sincronización que necesite en cada momento y con el número de sonares que quiera en cada grupo.

4. No es necesario utilizar todos los sonares en los grupos de sincronización, lo cual permitiría por ejemplo que si en un momento dado estamos interesados en los objetos de un lateral, podemos activar sólo los sonares de ese lateral y no sólo eso sino que podríamos activarlos en 2 grupos para que no haya interferencia mutua entre los sonares que estén cercanos. Este ejemplo nos da una idea de la flexibilidad y libertad del método implementado.

Sin embargo el único problema que tiene es que a más número de grupos de sincronización mayor será el periodo de actualización de los sonares y por lo tanto habrá que tenerlo en cuenta a la hora de diseñar que cantidad de grupos permite y necesita el algoritmo a implementar.

Finalmente comentar que se ha conseguido el objetivo de crear un método de sincronización de los sonares dinámico, flexible y que pueda variar en tiempo real de una forma sencilla.