Capítulo 4

Sonares en el DSP

1.Introducción

En primer lugar hay que decir que a diferencia del controlador romeo4b, en este no había ningún software implementado para el manejo de los sonares por lo tanto se tuvo que implementar todo ese software partiendo de cero. Además es importante destacar que el software del controlador empotrado está escrito casi completamente en C y tiene otra estructura distinta a la del romeo4b, que se explicará en el siguiente apartado. Sin embargo se ha intentado mantener una gran similitud entre las funciones implementadas en ambos controladores para el manejo de los sonares, con el objetivo de facilitar el diseño de algoritmos de control para ambos controladores.

En lo que respecta al algoritmo de sincronización comentar que se basa en la misma idea y procedimiento y por lo tanto tiene las mismas características de flexibilidad, dinamismo y cambio en tiempo real. Sin embargo como ya se comentó en el Capítulo 2 (Apartado 2.3.2.2.2) en el controlador empotrado sólo se contaba con 4 salidas digitales lo cual obliga a que no se pueda tener la libertad total que se tenía en el controlador romeo4b, ya que esta circunstancia obligó a la creación de una serie de grupos fijos que siempre se tendrían que activar a la vez. Los sonares de los distintos grupos fijos se han asignado a partir del criterio de minimizar la interferencia entre sonares.

Por lo tanto se han unido en 4 grupos fijos de activación los 12 sonares (Tabla 4.1), aunque sólo 10 están instalados como ya se ha comentado en varias ocasiones.

Número del grupo fijo de sonares	***Sonares que forman parte del grupo***
1	1, 3, 6[1]
2	2, 4, 9
3	7, 10, 11
4	5, 8, 12

Tabla 4.1:Grupos de activación de sonares para el controlador empotrado

Finalmente decir que el código implementado tiene la particularidad de que en su mayoría está codificado en punto fijo debido a que el hardware del DSP no soporta operaciones en punto flotante y por lo tanto si se utiliza codificación en punto flotante las operaciones serán unas 100 veces más lentas ya que se deben emular por software. Este hecho será comentado más detalladamente en el siguiente capítulo.

1 Se recuerda que la numeración de los sonares corresponde a la que aparece en la Figura 2.2

2.Arquitectura software del controlador empotrado

Esta arquitectura fue desarrollada íntegramente por Víctor Manuel Blanco González en el proyecto fin de carrera "Controlador empotrado para vehículo autónomo ROMEO-4R" y con este apartado se quiere dar una visión general del software en el que se ha implementado el manejo de los sonares para el controlador empotrado.

2.1.Estructura del software

Este software está implementado de forma modular, pudiendo distinguir 3 niveles en los que se divide (Figura 4.1).

1. **Nivel hardware:** es el nivel más bajo y está formado por el hardware del controlador. En este nivel es necesario un alto nivel de conocimiento de la electrónica del controlador.
2. **Nivel del núcleo del sistema:** este nivel solventa de forma transparente al programador de alto nivel 3 cuestiones básicas:
    - La gestión de la ejecución de tareas con exigencias temporales.
    - El acceso a zonas de memoria compartida por varias tareas, evitando condiciones de carrera.
    - La gestión de operaciones de entrada/salida, es decir, acceso al hardware mediante una serie de drivers de bajo nivel.
3. **Nivel de tareas:** en este nivel se encuentran las diferentes tareas que debe realizar el controlador. Cada una de estas tareas tendrán una periodicidad y una prioridad dentro del sistema y será el *gestor de tareas*, que forma parte del núcleo del sistema, el encargado de iniciarlas según esa periodicidad y prioridad.

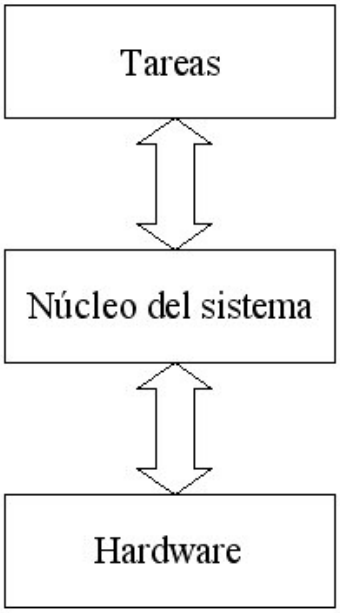

Figura 4.1:Estructura software del controlador empotrado

2.2.Nivel hardware

Debido a la gran relación existente entre el hardware y el software en el controlador empotrado, se ha visto interesante dar una visión general de los componentes hardware de este.

Este nivel está formado por 4 placas:

1. **Placa de evaluación EVM** de Spectrum Digital para el DSP que se va a utilizar que es el TMS320LF2407. Este procesador trae integrado en el propio chip el siguiente hardware:
    a. ***Memorias***: en total dispone de 544 words de memoria RAM y 32Kwords de Flash-EPROM.
    b. ***Watchdog Timer:*** se utiliza para evitar el bloqueo del programa a causa del algún error software o hardware.
    c. ***2 unidades gestoras de eventos:*** se puede utilizar para generar las señales PWM y para la interpretación de la señales procedentes de los codificadores con salida en cuadratura al llevar incluido un circuito QEP.
    d. ***Puertos de entrada/salida digitales de propósito general:*** se poseen 39 pines que pueden ser configurados como entrada o salida digital, aunque algunos de ellos

están reservados para los periféricos internos en el caso de que se usen. Finalmente tras dicha reserva quedaron 19 libres de los cuales 13 se han configurado de entrada digital y 6 de salida. De las 6 salidas 2 se utilizan para habilitar o deshabilitar los motores de tracción y dirección quedándonos 4 salidas digitales libres como se había comentado anteriormente.

e. ***Convertidor analógico/digital:*** con 16 canales de conversión y 10 bits de resolución, siendo el que se utilice para la conversión de las señales analógicas provenientes de los sonares.

f. ***Módulo controlador de interfaz con bus CAN:*** este módulo no se utiliza y sirve para comunicaciones serie multi-maestro en aplicaciones en tiempo real a partir del estándar CAN.

g. ***Puerto serie asíncrono (SCI):*** soporta comunicación serie asíncrona full-duplex con cualquier dispositivo que use formato NRZ. Para usar el interfaz estándar RS-232 es necesario añadir un circuito de conversión de niveles de tensión de TTL a ±12V (MAX-232), que se encuentra en la placa EVM.

h. ***Puerto serie síncrono (SPI):*** este dispositivo ha sido utilizado junto con electrónica implementada en las otras placas para gestionar 4 puertos series asíncronos con interfaz RS-232 necesario para la comunicación con los diversos sensores que tiene implementado el vehículo ROMEO-4R.

La placa EVM además de alimentar al chip, proporcionar la señal de reloj, etc. trae una serie de dispositivos hardware implementados.

a. ***Memorias externas:*** 2 módulos de memoria RAM, 64Kwords para memoria de programa y 64Kwords para memoria de datos.

b. ***Convertidor digital/analógico:*** tiene 4 canales de conversión simultánea y 12 bits de resolución y se utiliza para el control de los sistemas de tracción y dirección de ROMEO-4R.

c. ***Conectores y circuitos de adaptación de señal:***

- *Conector DB9* junto con un circuito de conversión MAX232, que junto con el SCI conforma un puerto serie asíncrono con interfaz estándar RS-232.
- *Conector JTAG* para la depuración y trasvase del código del DSP desde el PC.
- *Conector CAN* hembra mini-Dim de 4 pines.

- *4 conectores de expansión* donde se hacen accesibles las líneas de los buses de dirección, de datos y de control del DSP y los pines de los periféricos internos.

d. ***Switches y Led's:*** que están mapeados en el espacio de direccionamiento del DSP reservado para le entrada/salida.

2. **Módulo I y II del DSP:** estas 2 placas integran el hardware necesario para cubrir ciertas carencias de la placa EVM del DSP.

   a. ***Acondicionamiento de la alimentación del controlador:*** ya que el controlador se alimenta a 24V y no coincide con las alimentaciones necesarias para los dispositivos que es de 5V o de ±12V. La circuitería que realiza estas conversiones se encuentra en el módulo I.

   b. ***Interfaz analógica con los motores:*** aunque como se ha comentado la placa EVM tenga un convertidor digital/analógico el rango de salida es de 0 a 5V y sin embargo para el rango de entrada de los sistemas de tracción y dirección de ROMEO-4R es de ±10V. Por lo tanto en el módulo I se implementa la electrónica para mapear el rango de salida del convertidor con el de entrada de los sistemas de control.

   c. ***Interfaz digital para el control de motores:*** se han colocado puertas lógicas tipo buffer entre los codificadores ópticos y el DSP para proteger a este de sobretensiones. Esta interfaz también se encuentra implementada en el módulo I.

   d. ***Protección de para las entradas y salidas digitales:*** se han incluido una electrónica de protección y de conversión de niveles de tensión para todas las entradas y salidas digitales. Esta electrónica se encuentra distribuida entre los 2 módulos.

   e. ***Protección de entradas analógicas:*** se ha implementado una electrónica en el módulo II de tal forma que se asegure que las señal analógicas que lleguen al DSP estén en el rango de 0 a 3.3V y además también se ha limitado la corriente máxima de entrada para proteger al DSP.

   f. ***Ampliación de la capacidad de comunicación:*** como se comentó anteriormente a partir del SPI se van a gestionar 4 puertos series asíncronos con interfaz RS-232 debido a la necesidad de este tipo de puertos para la comunicación con los sensores que tiene implementado ROMEO-4R. Para implementar esta ampliación de puertos RS-232 es necesario una electrónica que se encuentra en el módulo II del DSP y que está formada por dispositivos programables GAL22V10 y dispositivos MAX232.

3. **Módulo III del DSP:** este módulo ya ha sido comentado en el Capítulo II (Apartado 2.3.2.2.1) y cuyas principales características son:

    a. Tiene la posibilidad de conectar 16 entradas analógicas que se conectarán con el conversor A/D del DSP, de estas 16 cuatro de ellas se les puede ajustar el offset y la ganancia en la misma placa de forma individual mediante potenciómetros (la circuitería utilizada es muy parecida a la que se utiliza en la caja de los sonares para ajustar la ganancia y el offset). Las otras 12 se conectan directamente al DSP sin llevar a cabo ningún tipo de ajuste.

    b. Posibilidad de conectar 2 salidas analógicas en las que se le puede ajustar el offset y la ganancia de manera individual en la misma placa, también por medio de potenciómetros.

    c. Jumpers de selección para seleccionar entre las 12 señales provenientes de los sonares o entradas analógicas alternativas pero sin poder ajustar ni el offset ni la ganancia. Estos jumpers actualmente seleccionan las señales de los sonares al ser las que se utilizan.

2.3.Nivel del núcleo del sistema

2.3.1.El gestor de tareas

Esta basado en la interrupción de tiempo real que tiene una cadencia de 1 milisegundo y que es generada por el timer-contador 1 del módulo gestor de evento EVA. En realidad al terminar la cuenta el timer-contador se genera una señal de conversión para el convertidor A/D, el cual después de realizar la conversión de los 16 canales genera la interrupción de tiempo real.

Por lo tanto la unidad de tiempo mínima que maneja el controlador es de 1 milisegundo, cuyo valor es de compromiso entre la frecuencia máxima de muestreo necesaria para el control del vehículo y el consumo de CPU ya que cada vez que se ejecuta la rutina de interrupción hay que guardar el estado completo de la CPU.

La interrupción de tiempo real realiza 2 operaciones fundamentales:

1. Actualiza el valor de una variable que indica el transcurso del tiempo desde el momento de arranque del sistema. Debido a la importancia de esta variable en un sistema de control en tiempo real existe una llamada al núcleo del sistema para obtener su valor y así evitar que el programador de tareas la maneje directamente.

2. Ejecutar el gestor de tareas.

El gestor de tareas es una rutina que decide en cada milisegundo cuales de las tareas existentes deben iniciar su ejecución, y en caso afirmativo las ejecuta. Esta decisión se toma en función de las 2 propiedades básicas de las tareas, que son la cadencia y la prioridad.

Cada tarea en realidad es una función en C que tienen que cumplir las siguientes particularidades:

- No deben tomar ningún parámetro ni devolver valor alguno.
- No pueden contener bucles infinitos, ya que esto provocaría que las tareas de baja prioridad nunca lleguen a ejecutarse.

Además hay que aclarar que la prioridad es un número que identifica unívocamente a una tarea, es decir, que no pueden existir 2 tareas con la misma prioridad. Actualmente existen 20 niveles de prioridad aunque se podría aumentar si fuese necesario.

El funcionamiento del gestor de tareas es el siguiente en cada milisegundo:

- Calcula cuales de las tareas deben ser ejecutadas en el milisegundo actual. Esto se calcula a partir de la última vez que se ejecutaron las tareas y la cadencia de cada una.
- Ordena las tareas por orden de prioridad y las ejecuta según ese orden.

Las tareas pueden ser interrumpidas por el gestor de tareas para ejecutar una tarea más prioritaria (Figura 4.2). Esto se ha diseñado así para asegurar que las tareas de mayor prioridad cumplen sus exigencias temporales de forma más estricta que las menos prioritarias. Cuando una tarea es interrumpida, se queda latente hasta que se termine la ejecución de todas la rutinas de mayor prioridad , inmediatamente después, continúa su ejecución por el punto en que se interrumpió.

Existen funciones del núcleo del sistema para *arrancar* tareas (introducir la tarea en el gestor de tareas con una cadencia y prioridad para que se empiece a ejecutar periódicamente) y para *matar* tareas (sacar una tarea del gestor de tareas y por lo tanto ya no se volverá a ejecutar).

Todas las tareas que son ejecutadas por el gestor de tareas de forma periódica se le llama *proceso foreground* (Figura 4.2) y el resto de tiempo sobrante en cada milisegundo se le llama *proceso background* sobre el cual el gestor de tareas no tiene control alguno y su velocidad es muy dependiente de la carga del sistema, pudiendo llegar a no ejecutarse jamás. En la actualidad este tiempo restante de CPU se usa para ejecutar una línea de comandos a través de un terminal serie que en realidad es un PC conectado por RS-232 al controlador y haciendo uso del programa HyperTerminal.

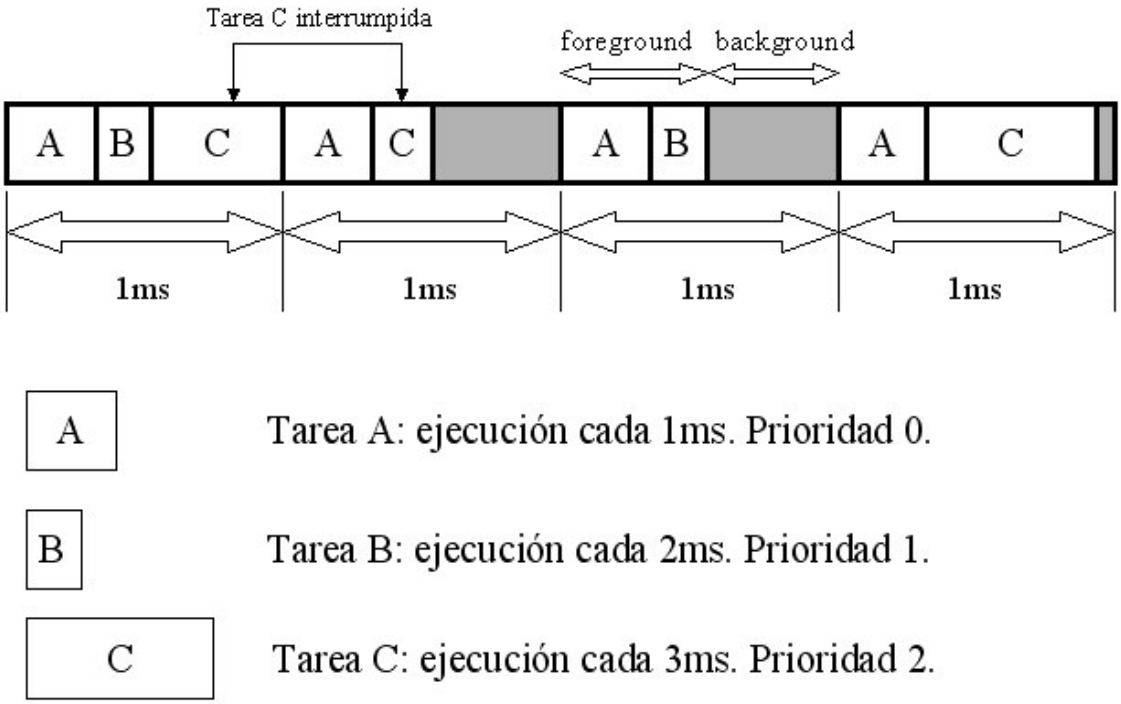

Figura 4.2:Funcionamiento del gestor de tareas

2.3.2.Gestión de accesos a memoria compartida

Dado en primer lugar que las tareas para comunicarse comparten memoria y que además las tareas pueden ser interrumpidas en cualquier momento, se pueden producir condiciones de carrera.

Para solucionar este problema se ha optado por implementar operaciones atómicas, que son aquellas que se ejecutan en su totalidad sin posibilidad de ser interrumpidas en mitad de su operación. Para implementar operaciones atómicas en el DSP lo que se ha hecho es que cada vez que se escribe o lee en memoria compartida se deshabilitan las interrupciones ya que la única manera de interrumpir una tarea es mediante la ejecución del gestor de tarea que se ejecuta con la interrupción de tiempo real.

Sin embargo la habilitación y deshabilitación de interrupciones por parte del programador de tareas puede ser peligroso para el correcto funcionamiento del sistema, por ello se crearon una serie de funciones para escribir y leer en memoria compartida para distintos tipos de datos.

2.3.3.Llamada a los drivers de lectura y escritura de bajo nivel

Como se ha visto en la descripción del nivel hardware del controlador empotrado existen multitud de periféricos, teniendo cada uno sus particularidades y una forma muy concreta de manejo por parte de la CPU. Entonces con el objetivo de facilitar el manejo de los dispositivos para las aplicaciones de alto nivel se crearon una serie de "drivers" de bajo nivel, además se quiso dar un paso más y se consiguió que todos esos drivers tuvieran una única interfaz.

Por lo tanto se han reducido a 2 llamadas del núcleo para realizar cualquier tipo de acceso a los dispositivos, una para leer y otra para escribir.

2.4.Nivel de tareas

En este nivel se incluyen las siguientes tareas:

- Gestor de emergencias (prioridad 0 y cadencia 1ms).
- Decodificador de mensajes del giróscopo (prioridad 1 y cadencia 1ms).
- PID dirección (prioridad 2 y cadencia 5ms).
- PID tracción (prioridad 3 y cadencia 400ms).
- Odometría (prioridad 4 y cadencia 1ms).
- Gestión de comunicaciones con otros nodos (prioridad 5 y cadencia 1ms).
- Actualización de los valores de los sonares y gestión de la sincronización de estos (prioridad 6 y cadencia 200ms).

Esta última tarea ha sido implementada en este proyecto y como su nombre indica se encarga de actualizar los valores de los sonares, siguiendo la estrategia de sincronización que se haya programado.

Además habría que incluir las tareas que implementan los algoritmos de control, actualmente están implementados 2 algoritmos:

- **Pure pursuit:** algoritmo codificado en punto flotante para el seguimiento de camino por medio de un sistema de control distribuido donde un PC le va comunicando al controlador empotrado el camino a seguir.
- **Aparcamiento:** algoritmo de aparcamiento planificado y codificado en punto fijo que ha sido desarrollado en el presente proyecto y del que se hablará más detalladamente en el Capítulo 5.

2.4.1.Utilidades disponibles

Existen una serie de funciones que permiten facilitar el trabajo al programador de tareas. Se dividen en 2 tipos:

1. **Llamadas de E/S por consola:** estas funciones permiten imprimir cadenas de caracteres y tomar cadenas del terminal serie conectado al controlador empotrado, que como ya se dijo anteriormente era un PC conectado por el puerto RS-232 al controlador empotrado y ejecutándose el programa HyperTerminal.
2. **Funciones de conversión punto fijo-punto flotante:** estas funciones permiten pasar de un número entero codificado en punto fijo a un número con el mismo valor codificado en punto flotante y viceversa.

2.4.2.Línea de comandos

Esta línea de comandos se ha desarrollado con el objetivo de facilitar la depuración del código que se desarrolle para el DSP. Esta línea de comandos se ejecuta en el DSP a través del terminal serie ya comentado anteriormente.

La línea de comandos interpreta un reducido número de órdenes y al no ser un proceso crítico se ejecuta dentro del *proceso background.* Las características de la línea de comandos son:

1. Es capaz de mostrar 10 variables globales (numeradas del 0 al 9) y que pueden ser usadas por los algoritmos que se quieran depurar.
2. Se pueden modificar los valores de esas 10 variables anteriores.
3. Se pueden ejecutar 10 rutinas diferentes como parte del *proceso background.*

3.Software de los sonares

3.1.Introducción

Para este controlador no sólo se ha implementado el algoritmo de sincronización de los sonares sino que se ha desarrollado todo el software necesario para la gestión de los sonares por medio del controlador empotrado.

Por otra parte se ha intentado que tanto las funciones implementadas como sus relaciones para ambos controladores sean muy parecidas, con la idea de facilitar el desarrollo de algoritmos para los dos controladores. Sin embargo siempre hay diferencias insalvables como por ejemplo que el software del DSP trabaja casi en su totalidad en punto fijo.

También hay que destacar que como ya se ha comentado anteriormente, por falta de salidas digitales se han tenido que crear una serie de grupos fijos de activación de los sonares lo cual provoca que el programador de tareas no tenga tanta libertad como en el romeo4b. Salvando esta diferencia el algoritmo de sincronización es exactamente igual que el implementado en el PC.

Finalmente comentar que todo el software que se va a comentar en este apartado se encuentra en los ficheros *sonares.c* y *sonares.h.*

3.2.Variables

Se han definido una serie de variables necesarias para el correcto funcionamiento de las funciones. Hay que destacar que el uso de estas variables son transparentes al programador de tareas y no necesita conocerlas para obtener los datos de los sonares y gestionar los grupos de sincronización, sin embargo se explican por si alguien en el futuro necesitara modificar el contenido de las funciones que posteriormente se van a explicar.

- **int cadena_sonares[NUM_JOINT_SONARS+1][NUM_JOINT_SONARS+1]:** en esta tabla de 2 dimensiones se van a guardar los grupos de sincronización que se van a utilizar en el algoritmo de sincronización. A diferencia del PC, en cada grupo de sincronización no se especifican cada uno de los sonares que forman parte, sino los grupos fijos de activación (ver Tabla 4.1). Esto se ha hecho así ya que en este controlador, como ya se ha comentado, no se pueden activar los sonares por separado por limitaciones en el número de salidas digitales existentes, por lo que se ve más conveniente que se especifiquen los grupos fijos de sonares que se quieren activar dentro del grupo de sincronización ya que de esta forma el programador de tareas será consciente de que está activando conjuntos de sonares y no sonares independientes.

Por otra parte el número máximo de grupos de sincronización es NUM_JOINT_SONAR, al igual que el número máximo de grupos fijos de activación por grupo de sincronización. Aunque pueda parecer una limitación, no los es porque:

- Puede haber tantos grupos de sincronización como grupos fijos de activación hay.
- En cada grupo de sincronización se pueden activar todos los grupos fijos de activación que hay y por lo tanto todos los sonares, no teniendo sentido activar 2 veces en un mismo grupo de sincronización un grupo fijo de activación ya que de los sonares se va a leer un único valor, aunque aparezca 2 o más veces en un mismo grupo de sincronización.

Sin embargo siempre se podría variar este valor de manera sencilla si se diera el caso que fuera necesario, sólo con variar el define de NUM_JOINT_SONARS.
Por último comentar que el tamaño de *cadena_sonares* es NUM_JOINT_SONARS+1 y no NUM_JOINT_SONARS porque se sigue la misma "sintaxis" de configurar los grupos de sincronización que en el PC, es decir, el último grupo de sincronización debe ser un grupo ficticio con un único grupo de activación fijo con el número 0 y en cada grupo de sincronización el último grupo de activación fijo debe ser uno ficticio con el número 0 también (si existe alguna duda consultar el apartado 3.4 del Capítulo 3 cambiando el número de sonares por el número del grupo de activación fijo).

- **int num_groups:** en esta variable se almacena el número actualizado de grupos de sincronización.
- **SONAR_DATA sonar_data[SONAR_SIZE_OF_QUEUE]:** cola donde se guardan en la estructura SONAR_DATA los datos relacionados con los sonares. La estructura SONAR_DATA está formada por los siguientes variables:
  - *long_real sonar[NUM_SONAR]:* tabla donde se guardan los valores obtenidos de los distintos sonares en metros y codificados en punto fijo (con 16 bits para la parte fraccionaria o con escalado 2^{16}, ver Apartado 3.4).
  - *unsigned long int time_sonar_global:* instante de tiempo en el que se leyeron las medidas de los sonares guardadas en la tabla anterior.
  - *unsigned long int time_group_sonars[NUM_JOINT_SONARS]:* instante de tiempo en el que se leyeron las medidas de los sonares que forman parte de cada uno de los distintos grupos de sincronización.
- **int new_sonar_data_flag:** indica si hay nuevos datos disponibles de los sonares.

- **int flag_tiempo:** esta variable se utiliza en la función *update_sonars* y sirve indicar si tengo que seguir esperando el tiempo de disponibilidad (tiempo mínimo para que el sonar pueda obtener una medida en el peor de los casos, que se da cuando el objeto está en el límite de la zona de detección del sonar, 380ms) o si se puede pasar al siguiente grupo de sincronización.
- **unsigned long int aux_tiempo:** variable auxiliar donde se almacena temporalmente el tiempo en el que he activado los grupos fijos de activación que forman parte del grupo de sincronización activo en cada momento.
- **int num_group_sonars_activo:** contiene el grupo de sincronización que en cada momento está activo.
- **int sonar_index:** indica la posición actual dentro del buffer de datos de los sonares.

3.3.Funciones implementadas

1. **act_sonar:** activa los sonares que forman parte de un grupo fijo. Mientras el grupo fijo esté activo los sonares que lo forman seguirán funcionando.

   Parámetros:

   - int joint_sonars: grupo fijo de sonares que queremos activar (los grupos fijos están numerados del 1 al 4).

   Devuelve:

   - void.

2. **deact_sonar:** desactiva los sonares que forman parte de un grupo fijo. Mientras el grupo fijo esté desactivado los sonares que lo forman no estarán funcionando.

   Parámetros:

   - int joint_sonars: grupo fijo de sonares que queremos desactivar (los grupos fijos están numerados del 1 al 4).

   Devuelve:

   - void.

3. **inicializa_cadena_sonares:** como en el controlador empotrado no tenemos la posibilidad de tener un archivo de configuración a partir del cual inicializar los grupos de sincronización, se ha creado está función que se debe ejecutar en la función

inicializa_Juliet en el fichero *inicia.c*. Por lo tanto está función inicializa los grupos de sincronización de los sonares, además también se utiliza para inicializar algunas de las variables globales anteriormente descritas. Por último comentar que la inicialización de los grupos de los sonares la deberá realizar el programador de tareas y deberá seguir la sintaxis indicada en el apartado anterior.

Parámetros:

- void.

Devuelve:

- void.

4. **finaliza_sonares:** desactiva todos los sonares y se deberá ejecutar al finalizar el programa principal del controlador empotrado. Se ha creado esta función porque al acabar el programa principal las salidas digitales conservan su valor hasta que no se desconecte la alimentación y puede darse el caso de que aunque el programa principal haya terminado algunos sonares sigan activos.

   Parámetros:

   - void.

   Devuelve

   - void.

5. **update_sonars:** actualiza el valor de los sonares en metros y codificado en punto fijo (con 16 bits para la parte fraccionaria) siguiendo el algoritmo de sincronización, además esta función será la que se introduzca como tarea en el planificador. Por otra parte en el caso de que haya algún error a la hora de obtener los valores de los sonares se imprimirá por la pantalla del terminal serie un mensaje de error con 3 números, donde cada número indica el tipo de error para cada uno de los 3 sonares que forman parte del grupo de activación fijo que está activo en ese momento. Estos errores no se devuelven ya que las funciones que implementen tareas no pueden tener parámetros ni devolver ningún valor, como ya se comentó en el apartado 2.3.1 de este capítulo.

   Parámetros:

   - void.

Devuelve:

- void.

6. **read_sonar:** lee el valor de un sonar y lo devuelve en metros y codificado en punto fijo.

   Parámetros:

   - int sonar: número del sonar que se quiere leer, siendo el primero el número 1.
   - int *error: código de error que se pasa por referencia. Si vale 0 no ha habido errores, si vale 1 indica error en la lectura del sonar, si vale 2 indica número de sonar no válido.

   Devuelve:

   - long_real: valor del sonar en metros y codificado en punto fijo con 16 bits para la parte fraccionaria.

7. **new_sonar_data:** comprueba si hay nuevos datos de los sonares.

   Parámetros:

   - void.

   Devuelve:

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

8. **get_sonar_data:** devuelve la estructura de datos de los sonares (SONAR_DATA).

   Parámetros:

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

   Devuelve:

   - SONAR_DATA: estructura de datos de los sonares ya comentada.

9. **erase_all_groups:** borra todos los grupos de sincronización de los sonares. Hay que tener cuidado con el uso de esta función ya que si después de ejecutarse no se crea ningún grupo de sincronización, entonces ningún grupo fijo de sonares se activará.

   Parámetros:

   - void.

Devuelve:

- void.

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

    Parámetros:

    - int group: número del grupo de sincronización a borrar. El primer grupo es el número 1.

    Devuelve:

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

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

    Parámetros:

    - int first_joint_sonara: número del primer grupo de activación fijo que va a formar parte del grupo de sincronización que se va a crear (numerados del 1 al 4).

    Devuelve:

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

12. **put_joint_sonars_in_group:** añade un grupo de activación fijo al final del grupo de sincronización especificado.

    Parámetros:

    - int group: número del grupo de sincronización al que se quiere añadir el grupo de activación fijo, teniendo en cuenta que el primer grupo de sincronización es el número 1.
    - int joint_sonars_number: número del grupo de activación fijo a añadir (numerados del 1 al 4).

    Devuelve:

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

vale -2 indica que el número del grupo de activación fijo no está dentro del rango válido.

13. **erase_sonar_group:** elimina un grupo de activación fijo del grupo de sincronización especificado.

    Parámetros:

    - int group: número del grupo de sincronización del que se quiere eliminar el grupo de activación fijo, teniendo en cuenta que el primer grupo de sincronización es el número 1.
    - int joint_sonars_number: número del grupo de activación fijo a eliminar (numerados del 1 al 4).

    Devuelve:

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

3.4.Aritmética de punto fijo

En los apartados anteriores se ha comentado que el valor de los sonares se almacenaba en la estructura SONAR_DATA en metros y codificado en punto fijo, esto se debe a que el DSP no tiene hardware específico para operaciones con números en punto flotante y por lo tanto si se quiere trabajar con esta codificación se tienen que emular las operaciones por software pero tiene el inconveniente de que las operaciones son unas 100 veces más lentas. Por ello casi todo el código del DSP está escrito usando aritmética de punto fijo ya que de esta forma se consigue representar números reales con números enteros, para los cuales el DSP sí tiene hardware implementado.

Uno se preguntará cómo es posible almacenar números con decimales en un entero y además realizar operaciones entre ellos y que después el resultado sea correcto. En primer lugar vamos a explicar con un ejemplo como se pasa de un número con decimales a un número entero y viceversa con la codificación en punto fijo. Supongamos que tenemos el número 1,256, si lo multiplico por un número grande como 20 me da 25,12 y si me quedo con la parte entera me quedaría el número 25. Entonces 25 sería 1.256 codificado en punto fijo, para volver a obtener el número decimal lo que hago es la operación contraria 25/20 que me da 1,250 que es el número que teníamos antes pero con una pequeña pérdida en la precisión, esto hay que tenerlo siempre en cuenta ya que cuando se

trabaje con números codificados en punto fijo se van a producir pérdidas en la precisión de los resultados, aunque como se verá si se hace correctamente estas pérdidas son mínimas.

En el código del DSP se hace básicamente lo mismo para pasar de codificación en punto fijo a punto flotante y viceversa, sin embargo el número por el que se multiplica o divide es una potencia de 2 (Ecuación 4.1).

$$num\_punto\_fijo = \lfloor num\_real \cdot 2^{exp} \rfloor \tag{4.1}$$

Existen funciones para realizar estas operaciones las cuales son:

1. **pasa_a_long_real:** pasa un número flotante a un entero con codificación en punto fijo.

   Parámetros:

   - float x: el número codificado en punto flotante.
   - int escalado: exponente de la potencia de 2.

   Devuelve:

   - long_real: número codificado en punto fijo, donde long_real es en realidad un typedef a long que en el DSP representa un entero de 32 bits (31 bits + signo). Esto se ha hecho para hacer más legible el código y diferenciar cuando un entero representa a un entero o a un real codificado en punto fijo.

2. **pasa_a_float:** pasa un número entero codificado en punto fijo a un número flotante.

   Parámetros:

   - long_real x: número codificado en punto fijo que representa a un número real.
   - int escalado: exponente del factor de escalado con que fue codificado x.

   Devuelve:

   - float: el número codificado en punto flotante.

Una vez que ya hemos entendido como se realiza la codificación en punto fijo vamos a explicar como se realizan las operaciones aritméticas básicas entre números con esta codificación. En realidad estas operaciones se realizan directamente, sólo hay que tener en cuenta una serie de precauciones, que son:

1. Al sumar o restar 2 números ambos deben tener el mismo escalado.
2. Al multiplicar 2 números no tienen porque tener el mismo escalado, sin embargo el resultado tendrá tanto escalado como la suma de los escalados de los números que hemos multiplicado.
3. Si se tiene que dividir se aconseja multiplicar por el inverso de uno de los 2 números y por lo tanto hay que tener cuidado con la precaución anterior.

Por otra parte quiero destacar que en el escalado, el exponente de la potencia de 2, coincide con el número de bits que se utilizan para la parte decimal del número.

Con respecto a la necesidad de utilizar funciones trigonométricas existen una serie de funciones desarrolladas por Texas Instruments que permiten hacer uso de estas funciones en punto fijo, además también existe una función para invertir los números en esta codificación. Estas funciones son:

1. **qsin:** calcula el seno de un número codificado en punto fijo.

   Parámetros:

   - int x: ángulo codificado en punto fijo con 15 bits para la parte fraccionaria, por lo tanto x varía entre -1 y 1 (int es un entero de 16 bits). Para realizar la correspondencia se escala el ángulo haciendo coincidir $-\pi$ con -1 y π con 1.

   Devuelve:

   - int: número codificado en punto fijo con 15 bits para la parte fraccionaria, que nos da el resultado del seno.

2. **qcos:** calcula el coseno de un número codificado en punto fijo.

   Parámetros:

   - int x: ángulo codificado en punto fijo con 15 bits para la parte fraccionaria.

   Devuelve:

   - int: número codificado en punto fijo con 15 bits para la parte fraccionaria, que nos da el resultado del coseno.

3. **qatan:** calcula el arcotangente de un número codificado en punto fijo.

   Parámetros:

   - long int x: número codificado en punto fijo con 16 bits para la parte fraccionaria (long int es entero de 32 bits).

   Devuelve:

   - int: ángulo resultado codificado en punto fijo con 15 bits para la parte fraccionaria (varía entre -1 y 1).

4. **qinv:** calcula el inverso de un número codificado en punto fijo.

   Parámetros:

   - int x: número a invertir codificado en punto fijo con X bits para la parte fraccionaria.

   Devuelve:

   - long int: número invertido codificado en punto fijo con 31-X para la parte fraccionaria.

Por último comentar que cuando se trabaja en punto fijo es muy importante conocer el rango dentro del cual van a variar los números con los que vamos a trabajar y en función de eso aplicar más bits a la parte entera o a la parte decimal, ya que cuanto más bits tengamos para la parte decimal mayor será la precisión pero menor será el rango.

3.5.Funcionamiento general

La función *update_sonars* se ejecuta periódicamente por medio del planificador, por lo tanto el programador de tareas o de algoritmos de control sólo debe llamar a la función *get_sonar_data* consultando anteriormente a la función new_sonar_data para saber si hay datos nuevos. Y ya después consultar de la estructura SONAR_DATA los valores que le interesen, teniendo en cuenta que los valores de los sonares están en metros y codificados en punto fijo con 16 bits para la parte fraccionaria. Además con respecto a la sincronización de los sonares deberá modificar la función *inicializa_cadena_sonares* para configurar los grupos iniciales de sincronización y después hacer uso de las funciones de gestión de la sincronización de los sonares para modificar los grupos de sincronización, si es necesario.

4.Conclusiones

En este capítulo se ha explicado como se ha integrado en la arquitectura del controlador empotrado el software para la gestión de los sonares que tiene las siguientes características:

1. No necesita que el programador de tareas se preocupe de la actualización de los valores de los sonares ya que esta la realiza el planificador llamando a la función *update_sonars*.

2. Se ha incorporado un método de sincronización de los sonares flexible, dinámico y que puede ir cambiando en tiempo real, sin embargo no se ha podido tener toda la flexibilidad que se tiene por ejemplo en el PC debido a la falta de salidas digitales libres en el controlador empotrado.

3. Para el cambio en tiempo real de los grupos de sincronización se han creado una serie de funciones que permiten la gestión de los grupos de manera sencilla para el programador de tareas.

4. Los resultados de los distintos sonares se obtienen en metros y codificados en punto fijo con 16 bits para la parte fraccionaria, lo cual es muy importante recordar si se quieren manipular esos datos o realizar operaciones con ellos.

En resumen se ha conseguido un sistema de gestión de los sonares de características muy parecidas al que se tenía con el PC pero con un controlador empotrado que tiene ventajas con respecto al controlador basado en PC de precio, tamaño, peso y consumo. La única diferencia fundamental en ambos sistemas es que en el PC se pueden sincronizar los sonares individualmente y en el controlador empotrado se sincronizan grupos fijos de sonares por no haber suficientes salidas digitales para todos los sonares.