

CAPÍTULO 4. ADQUISICIÓN DE DATOS

En este capítulo se describe la primera etapa del proceso de generación de un MDT, la **adquisición de datos**. El LD-MRS y MTi-G deberán trabajar de forma coordinada y sincronizada y generar un fichero ordenado con los datos adquiridos. Este fichero es único para cada vuelo y es la fuente de datos para un posterior procesamiento y representación del MDT.

A lo largo de este capítulo, se explica el código que se ha desarrollado para realizar la adquisición de los datos, el cual coordina el funcionamiento del LD-MRS y del MTi-G. El lenguaje de programación utilizado ha sido **C/C++**. Es el mismo lenguaje en el cual el fabricante de ambos dispositivos proporciona las librerías de funciones. El entorno de desarrollo usado ha sido **Eclipse** con su extensión para el lenguaje C/C++, instalado sobre un sistema operativo Linux, concretamente una distribución **Kubuntu**.

Para la **adquisición de datos** se han desarrollado **dos sub-programas distintos**, cada uno de ellos gobierna el funcionamiento de uno de los dispositivos. Uno de ellos se encarga del **MTi-G**, de configurarlo y de leer las estimaciones que proporcionan sus sensores, los cuales son guardados en una zona de memoria compartida. El segundo se encarga del funcionamiento del **LD-MRS**, lo configura y recibe los datos de los barridos que este realiza (*scans*). Éste programa también lee los datos del MTi-G almacenados en la zona de memoria compartida y lo hace de forma sincronizada, los asocia al último *scan* del LD-MRS obtenido y escribe la información de ambos dispositivos de forma conjunta y ordenada en un fichero de salida.

El fichero de salida que se genera se encuentra ordenado por *scans* del LD-MRS. Cada *scan* adquirido se asocia con los datos sincronizados del MTi-G, y se escriben en el fichero de salida. Los *scans* previos y consecutivos se encuentra en el mismo fichero pero separado entre ellos mediante líneas en blanco (caracteres de retorno de carro). Más adelante se explica con más detalle la estructura de estos ficheros.

4.1. Software del MTi-G

El código que controla al MTi-G se estructura en **varias partes**. La primera parte se ocupa de **configurar** el dispositivo, es decir, da valores a los parámetros necesarios del mismo para que éste realice las estimaciones y envíe la información que se necesita en el formato deseado. La segunda parte es un **bucle** responsable de realizar lecturas periódicas de los valores estimados por el filtro de Kalman y **actualizar una zona de memoria compartida** con los valores más recientes de los mismos.

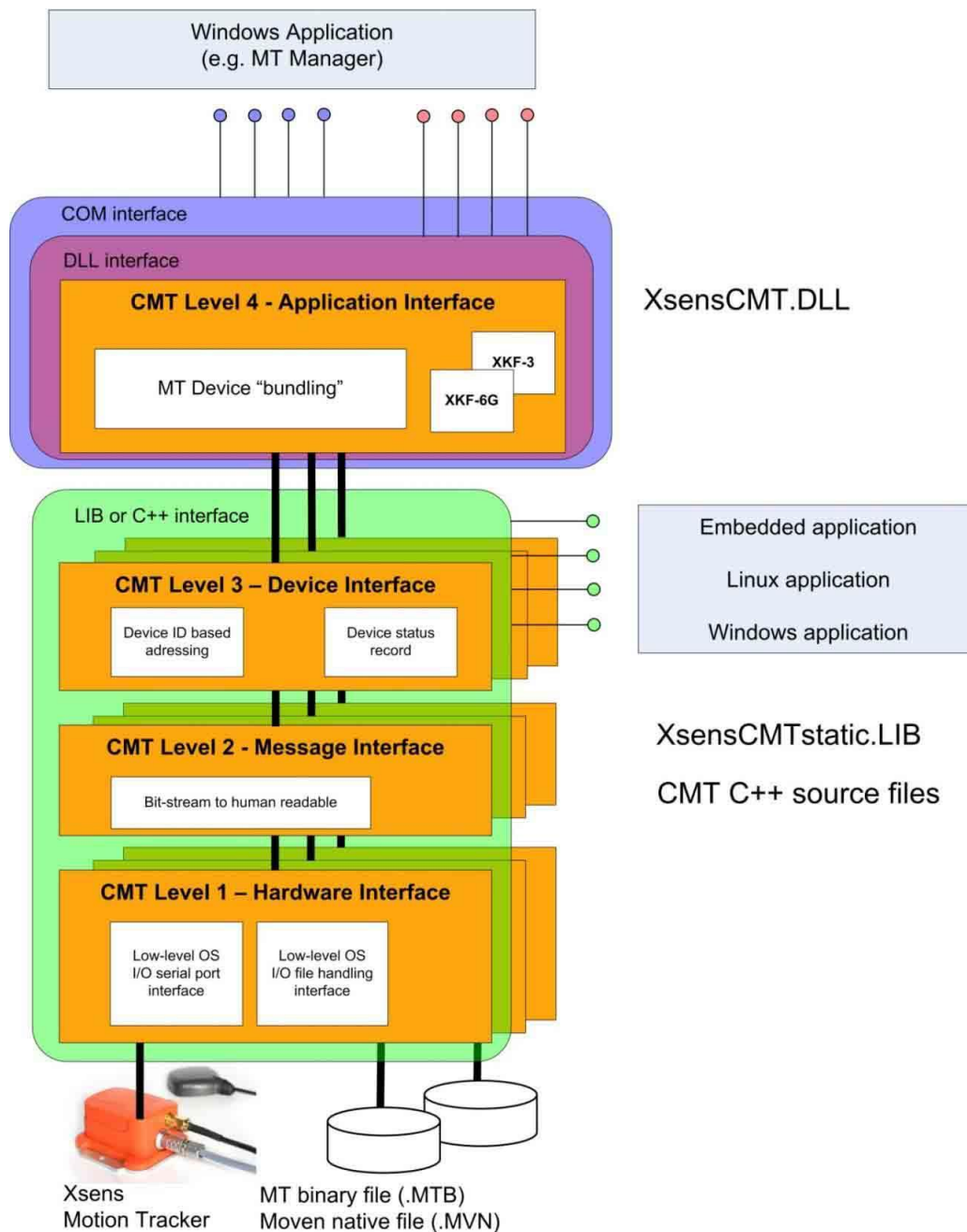


Figura 38. Niveles de programación del MTi-G (13)

A lo largo de todo este código se usa la librería de funciones que proporciona el fabricante del dispositivo. Según la arquitectura software del mismo **se puede trabajar con él a distintos niveles** (Figura 38), realizando medidas independientes con cada uno de los sensores a petición del usuario (nivel más bajo), o recibiendo paquetes de datos que el dispositivo genera como salida de su procesador y que contienen toda la información a la salida del filtro XKF-6G (nivel más alto). Los niveles más bajos tienen más importancia en aplicaciones de tiempo real y con gran criticidad de tiempos; como en el caso del sistema diseñado en este proyecto los tiempos los marca el LD-MRS **se trabaja con el nivel más alto**.

Al escribir un código utilizando el nivel más alto de este dispositivo, la configuración del MTi-G toma mucha importancia porque es el momento en el que se le comunica al dispositivo la forma y la velocidad a la que se quiere que nos entregue los paquetes de datos. Una vez que se ha hecho esto el resto del código se encarga de recibir los paquetes, leer los datos que hay dentro y almacenarlos en la zona de memoria compartida.

4.1.1. Main

Toda la funcionalidad del código desarrollado para el manejo de este dispositivo se engloba dentro de la función main.

4.1.1.1. Configuración del dispositivo

Empieza con la definición de las librerías que se necesitan para configurar el dispositivo. De entre ellas destaca la definición de la estructura de la **zona de memoria compartida** que se utiliza para intercambiar datos entre los procesos que controlan al MTi-G y al LD-MRS. Según los métodos de la programación concurrente, se necesita de un mecanismo de exclusión eficaz que regule el acceso a la zona compartida y evite inconsistencias en los datos de la misma, es decir, que uno de los procesos escriba mientras uno está leyendo o viceversa. El mecanismo elegido para esta tarea son los **semáforos**.

La zona de memoria compartida tiene una estructura específicamente definida para almacenar los tipos de datos que va a proporcionar el MTi-G, y se muestran en la Tabla 10. En la columna de la izquierda se muestran los nombres de las variables reales que se usan en el código desarrollado.

Variable	Tamaño (Bytes)	Información que recoge
utc	12	Tiempo de la zona horaria de referencia, según el estándar UTC (Coordinated Universal Time).
euler	24	Valor de los ángulos de Euler (Roll, Pitch, Yaw) con respecto al LTP en cada instante.

LLA	24	Posición del MTi-G expresada en latitud, longitud y altitud.
veloc	24	Velocidad del MTi-G en cada uno de los ejes del sistema de coordenadas body fixed .
status	4	Valor del status byte (visto en capítulos anteriores).
valid	4	Bandera que indica si la información del tiempo que proporciona el GPS es fiable (valor 1) o no lo es (valor 0).

Tabla 10. Estructura de datos de la zona de memoria compartida

Tras estas definiciones se realiza lo que se denomina un **Hardware Scan**. Dispositivos como el MTi-G pueden trabajar de forma coordinada y en grupo, por tanto lo principal en el software es detectar cuántos de estos dispositivos están conectados a la tarjeta de procesamiento. Se sabe de antemano que en este caso sólo es uno, pero aun así es necesario hacerlo pues es la forma de iniciar la comunicación con el dispositivo.

Durante el *hardware scan* se establece un puerto para recibir los paquetes que llegan desde el dispositivo y a la vez se establece la **velocidad de transmisión** de los datos (se recuerda que el conector del dispositivo es un cable serie con un convertidor a conector USB en un extremo).

El MTi-G tiene **dos estados de funcionamiento**, el de configuración y el de medida. Para configurarlo lo primero que hay que hacer es establecer el estado de configuración. Esto se realiza con la función **gotoConfig()**.

Tras establecer el modo de configuración, en primer lugar se eligen los datos que se quiere que el dispositivo envíe a la unidad de procesamiento, esto se hace la función **setDevicemode2()** y un conjunto de máscaras. Los datos que interesan para el sistema diseñado se muestran en la Tabla 11. Algunos de esos datos están disponibles en varios formatos de representación (la orientación como ya se vio, se proporciona mediante los ángulos de Euler, cuaterniones o matrices de rotación).

Datos requeridos al MTi-G	
Orientación	Ángulos de Euler
Posición	
Velocidad	
Estado (Status Byte)	

Tabla 11. Datos requeridos al MTi-G

Seguidamente se configura lo que se llama el **LeverArm**. Éste es un **vector** que le indica al dispositivo la relación de distancias entre los sensores inerciales, situados dentro del encapsulamiento naranja, y la posición

en la que se coloca la antena GPS. El vector expresa, en el sistema de coordenadas tipo *body fixed* del MTi-G, la posición de la antena con respecto a los sensores (en los sensores se sitúa el origen de coordenadas).

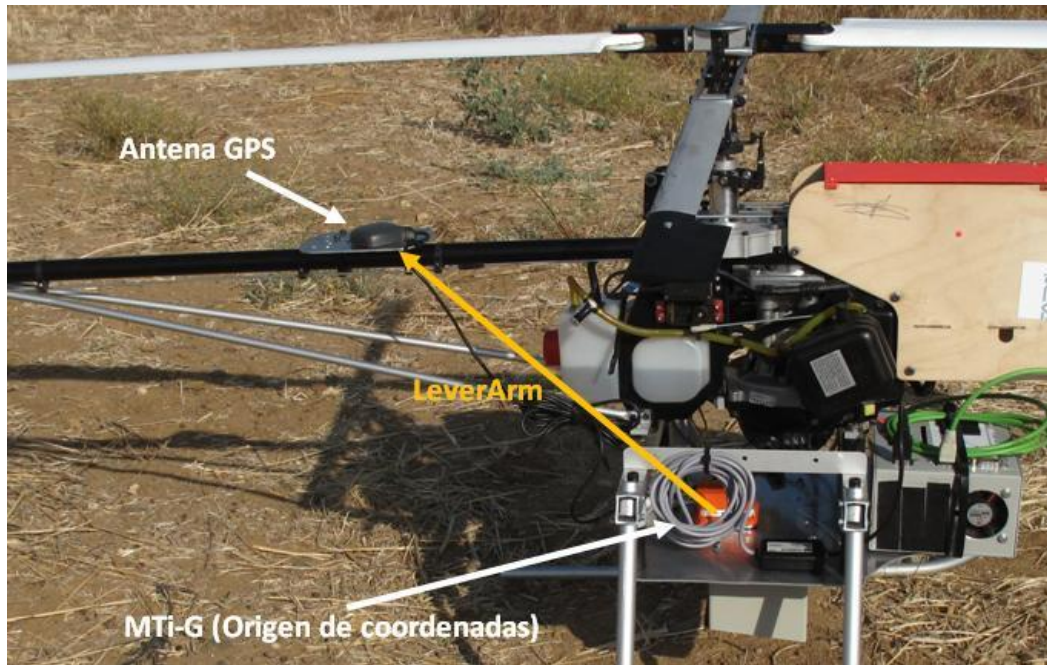


Figura 39. Ejemplo del vector *LeverArm* para el helicóptero CB 5000

Ambos dispositivos podrían colocarse en la misma posición, sin embargo **la antena receptora debe estar colocada en un lugar exterior del vehículo** aéreo, por aquello de que cuando más cerca del suelo la señal GPS es más débil, y si se obstaculiza la antena su recepción se ve fuertemente perjudicada. Por otro lado, el cuerpo del **MTi-G (sensores inerciales) debe estar colocado cercano al centro de gravedad** del vehículo, para así evitar lecturas de aceleraciones indeseadas de la dinámica del vehículo. Tal y como se aprecia en la Figura 39 esta posición está debajo de toda la mecánica del helicóptero y la antena GPS ahí no tendría una buena recepción.

A pesar de esta colocación separada de la antena y el encapsulamiento del MTi-G las lecturas de la posición, tanto del GPS como de los sensores inerciales, deben hacer referencia al mismo punto, sino se estaría cometiendo un error en la estimación de la posición por parte del XKf-6G. Por esta razón es necesario el uso del vector *LeverArm*, que internamente traslada las posiciones estimadas por la antena GPS al sistema de coordenadas *b-frame* de los sensores del MTi-G.

Debido a que el montaje del sistema diseñado es distinto en el caso del avión y del helicóptero, se tienen distintos valores del *LeverArm* (Tabla 12) y debe seleccionarse el adecuado para el tipo de vehículo que se usa y antes de realizar el vuelo.

Valores del LeverArm (m)			
Pilatus Porter		CB 5000	
x	-0.45	x	-0.34
y	-0.04	y	0.01
z	0	z	0.26

Tabla 12. Valores del LeverArm para los dos vehículos aéreos

Se recuerda de capítulos anteriores que el MTi-G dispone de varios escenarios predeterminados con los que trabaja el filtro XKF-6G, orientados a ofrecer medidas optimizadas para las características del medio que se modela en cada uno de ellos. El siguiente paso en el proceso de configuración es elegir el escenario que se va a utilizar. De entre las posibilidades que se vieron se ha escogido el **escenario Aerospace**, ya que es la que mejor se adapta para aplicaciones en las que se trabaja con vehículos aéreos. Una vez se han completado estos pasos se tiene el dispositivo perfectamente configurado.

4.1.1.2. Zona de memoria compartida

El **MTi-G es el primero** de los dos dispositivos **que se pone en marcha** según como se ha diseñado el software para la adquisición de datos. Esto es debido a que tiene un proceso de **calibrado inicial** que se lleva a cabo cada vez que se enciende, y supera en tiempo al periodo de inicialización del LD-MRS. Por esta razón es el primero de los dos procesos (un proceso se encarga del MTi-G y otro del LD-MRS) en crear la zona de memoria compartida, reservando un segmento de memoria del tamaño que se necesita y adhiriéndolo al espacio de direcciones del proceso, para poder depositar allí los valores de las variables de la Tabla 10.

Se necesita configurar la estructura de los semáforos para determinar cuántos se van a usar y los valores que se les van a dar a los mismos para su funcionamiento. En el caso que se está tratando sólo se tiene una zona de memoria compartida accedida por dos procesos, por lo que sólo se necesita **un semáforo**.

La **gestión del acceso** a la zona compartida es simple, se comprueba el valor del semáforo antes de acceder (el proceso del MTi-G solamente escribe en la zona de memoria compartida y, como ya se verá, el proceso del LD-MRS sólo lee). Si el valor del semáforo es 1, entonces el proceso del MTi-G baja el valor a 0 y accede a la zona compartida para volcar una copia de los valores estimados por el filtro XKF-6G. Cuando termine de escribir sale de la zona y establece el valor del semáforo en 1 de nuevo. En el caso en el que el LD-MRS hubiese estado leyendo cuando el proceso del MTi-G quería acceder, el semáforo hubiera estado a 0 y por lo tanto se le hubiera denegado el acceso al proceso del MTi-G. Así se protegen los datos de la zona de memoria compartida de las posibles inconsistencias, que se darían si ambos procesos accedieran a la vez, uno leyendo y el otro escribiendo.

4.1.1.3. Recepción de los paquetes

Una vez que el MTi-G está correctamente configurado y la zona de memoria compartida definida y disponible para su uso, el proceso entra en un bucle que se encarga de las tareas que se describen a continuación.

En primer lugar se le solicita al MTi-G el envío del último paquete de datos disponible, el más reciente y con los valores más actualizados. Esto se hace mediante la función ***requestData(packet)***. La variable *packet* (a partir de ahora también se hará referencia a la misma como “paquete”) es una clase definida en la librería de la API del dispositivo. Un paquete es una estructura de datos predefinida por el dispositivo y que agrupa la mayoría de los datos que éste es capaz de medir. Un paquete puede ser configurado por el usuario hasta cierto punto, decidiendo qué información está incluida en él y cual no.

También se le solicita al MTi-G la información relativa al tiempo, siguiendo el estándar **UTC (Tiempo Universal Coordinado)**. De entre el MTi-G y el LD-MRS, el dispositivo más preciso en cuanto a la medida del tiempo es sin duda el MTi-G, debido al receptor GPS que trabaja con las medidas de tiempo que recibe de los satélites del sistema GPS. Por esto mismo será el tiempo que se tome como referencia en el software del sistema diseñado.

Cada vez que se recibe un paquete desde el MTi-G se comprueba el byte de estado (Figura 30) y que sus banderas indiquen que la información que contiene el paquete es fiable (en el caso en el que el MTi-G se encuentra sin señal actualizada del GPS o el filtro de Kalman no converge, y estas banderas lo reflejan).

Según la estructura del byte de estado es necesario que el valor del mismo sea como mínimo de 5 (bits “0” y “2” con valor 1 y bit “1” con valor 0) para tener garantías de que la información contenida en el paquete es fiable; este valor de 5 en el byte de estado significaría que el test inicial del dispositivo se realizó correctamente y que la señal GPS se recibe actualizada y sin problemas, sin embargo el filtro de Kalman (la bandera “*XKF valid*” está a 0) no proporciona una salida fiable porque alguno de los sensores está saturando. En el caso de tener un byte de estado de valor 7 (bits “0”, “1” y “2” con valor 1), significaría que se tiene la misma situación que en el caso anterior pero con la diferencia de que ahora la bandera XKF está a “1” por lo que el valor de la salida del filtro de Kalman está convergiendo.

Si las banderas indican que la información que contiene el paquete de datos es fiable, se procede a leer la misma y guardarla en una estructura auxiliar con los mismos campos que los de la zona de memoria compartida. Si las banderas indican que la información no es fiable se toma el valor “0” de la variable *valid* como indicador de que esos datos inerciales no servirán para corregir la orientación de los datos del láser, cosa que se tendrá en cuenta posteriormente.

El siguiente paso es acceder a la zona de memoria compartida, para ello se intenta bajar el semáforo y, si no se detecta al otro proceso dentro de la zona de memoria compartida, el proceso del MTi-G accede y vuelca los datos del paquete en la zona de memoria compartida. Luego el proceso del MTi-G abandona la zona de memoria compartida y se sube de nuevo el valor del semáforo a “1”.

Hasta aquí se han presentado todas las tareas que se realizan dentro de este bucle, el cual se repite una y otra vez aproximadamente cada 20 ms. Esta frecuencia de ejecución del bucle viene dada por los tiempos de computación del propio MTi-G y de las funciones de la librería que se usan para leer los datos del MTi-G.

4.1.2. Pseudocódigo

En el anexo de este mismo documento se adjuntan diagramas de flujo que representan el pseudocódigo de esta parte del software, correspondiente al proceso del dispositivo MTi-G.

4.2. Software del LD-MRS

Para controlar la **adquisición de los datos del radar láser**, es decir, de cada uno de los barridos que realiza se ha desarrollado otro código. Esto realiza la adquisición de forma ordenada y asociando los datos que recoge con los que el MTi-G escribe en la zona de memoria compartida de manera **sincronizada**.

El código se divide en **dos hilos de ejecución principales, que se ejecutan de manera paralela** y se comunican entre ellos por medio de banderas. Uno de los hilos incluye la función principal o **main**, y el otro la función encargada de la recepción de los mensajes que envía el dispositivo a la tarjeta de procesamiento, esta función es conocida como **messageHandler()**. Los mensajes del LD-MRS son paquetes de datos con la información correspondiente a un barrido completo o *scan*.

Aparte, e incluida dentro del hilo de ejecución de la función *main*, existe otra función de bastante importancia, **processScan()**. Ésta es llamada desde la función *main* y se encarga de escribir en un fichero de salida la información contenida en los mensajes que llegan desde el LD-MRS, es decir la información de los *scans*, junto con la que se lee de manera sincronizada de la zona de memoria compartida, y que proviene del MTi-G.

En líneas generales estas son las características principales de este código que gestiona el dispositivo LD-MRS. En los siguientes apartados se describe más detenidamente la recepción de los mensajes, la sincronización con el proceso del MTi-G o la generación del fichero de salida, entre otros temas.

4.2.1. MessageHandler

La función ***messageHandler()*** es la encargada de la gestión de la recepción de los mensajes que envía el LD-MRS. Se ejecuta en un **hilo independiente** al de la función ***main y processScan()***. El mecanismo multi-hilo viene implementado por la API propia del LD-MRS, así que no se entra en más detalles acerca de la programación de este hilo.

Cada vez que el LD-MRS realiza un barrido y prepara los datos a enviar en forma de paquete, envía un mensaje con éstos a la tarjeta de procesamiento y es cuando la función ***messageHandler()*** **despierta** y comienza su ejecución. Lo primero que hace esta función es detectar el **tipo de mensaje** que ha llegado. Los mensajes del láser pueden ser de muchos tipos (parametrización, configuración, monitorización, datos, etc.), sin embargo, los que interesan para el sistema diseñado son los que contienen la información de las coordenadas de la superficie escaneada. Estos mensajes son del tipo ***ibeo::DataTypeScanPointList***, que también se reconocen por el código **0x2204**.

Con una estructura de control ***switch*** se comprueba si el mensaje que se ha recibido pertenece al tipo 0x2204, y si ese es el caso, se accede a **la zona de memoria compartida para tener una copia de la información del MTi-G más actualizada**. Ésta información de la zona de memoria compartida se asocia la información del *scan* recibido, de forma que se tienen asociadas las coordenadas de la superficie en un instante y la posición y orientación de la aeronave en ese mismo instante. Así se sincronizan ambos dispositivos, ya que la información que está en la zona de memoria compartida se actualiza a una tasa superior a la que se reciben los mensajes. Este paso de sincronización es de vital importancia para el posterior procesamiento de los datos y para su correcta representación.

La información leída de la zona de memoria compartida y que procede del MTi-G se almacena en una variable global que tiene la misma estructura que la memoria compartida. Esto se hace dentro del hilo de procesamiento de esta función, y hasta que se proceda a escribir los datos en el fichero de salida los datos se mantienen almacenados ahí. El hecho de que se use una variable global permite la comunicación entre los dos hilos de procesamiento antes comentados.

Tras leer la información que proviene de la zona de memoria compartida lo siguiente que se hace es almacenar el mensaje que ha llegado desde el LD-MRS. Los datos que contiene se copian en un **buffer** que actúa como mecanismo de control, para evitar que mientras se escribe en el fichero de salida los datos correspondientes al mensaje actual, llegue otro mensaje desde el LD-MRS y sobrescriba el que se estaba escribiendo en el fichero de salida (Figura 40).

Este **buffer** está formado por únicamente por **dos posiciones**. Éstas son ampliables, pero no ha sido necesario incluir un **buffer** más grande, siendo sus dos posiciones **suficientes según el esquema de**

sincronización utilizado. Este *buffer* tiene un mecanismo de funcionamiento *FIFO* (*First In First Out*). Cada una de las dos posiciones es una variable del tipo **MutexedScan**, un tipo definido en las librerías de la API del dispositivo. Estas variables tienen la característica de estar asociadas a un mecanismo de exclusión, un **mutex**, de manera que sólo se podrá acceder a ellas si el **mutex** lo permite (si no hay ningún otro proceso leyendo o escribiendo en la variable en el mismo momento).

Para gestionar el *buffer* se tienen unas variables de control y saber a qué variable **MutexedScan** hay que acceder en cada instante. La variable **next_msg** apunta siempre a la posición en la que se debe copiar el mensaje recién llegado del LD-MRS. Otra variable que se utiliza para el mismo fin es **next_proc**. Ésta se describirá más adelante.

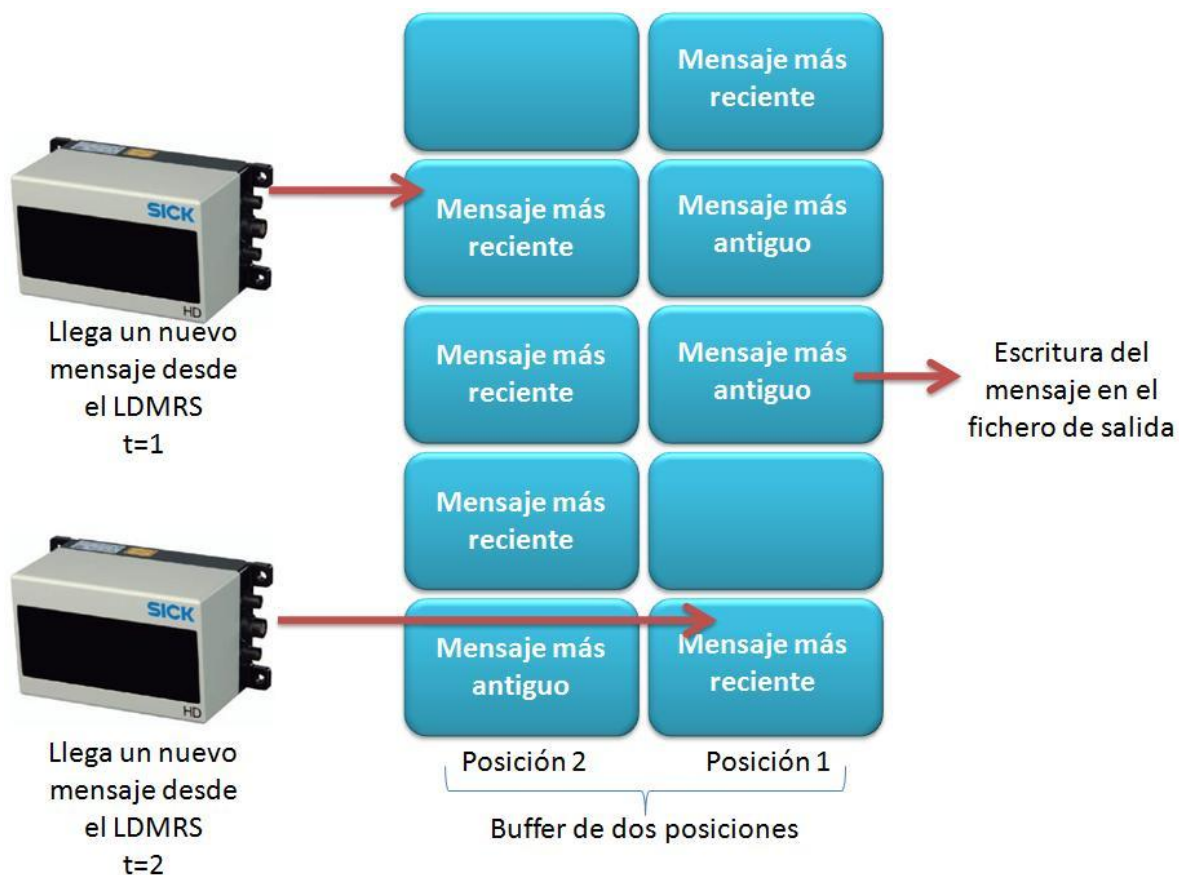


Figura 40. Buffer de recepción de mensajes del LD-MRS

En el momento en el que se realiza la **copia del mensaje recibido en el buffer**, se **levanta una bandera o señal** que actúa como **indicador para la función main**. La función **main** ejecuta un bucle a la espera de que la función **messageHandler()** le avise de que llegó un nuevo mensaje, este aviso es la bandera que se comenta, y la acción que desencadena es la llamada a la función **processScan()** que es la que escribirá los datos en el fichero de salida. Esta bandera es una variable global a la que pueden acceder ambos hilos (hilos distintos no pueden acceder el uno a las variables del otro a no ser que sean globales).

La función ***messageHandler()*** termina aquí sus tareas. Como se ha visto y de manera resumida, sus funciones son únicamente recibir el mensaje enviado por el LD-MRS, hacer una copia ordenada en el buffer (asociándole los datos del MTi-G más actuales) y avisar con una bandera o señal a la función *main* de la llegada de ese mensaje.

4.2.2. ProcessScan()

La **tarea principal** de esta función es leer la información de los mensajes que están en el *buffer* y escribirla de forma ordenada en el **fichero de salida**. Esta función está ligada al mismo hilo que la función *main*, el **hilo principal** del programa, y se encarga de **generar un fichero de texto con la información de cada scan y los datos inerciales y de posición** asociados al mismo. La razón de describir las características de esta función antes de las de la función *main* es simplemente porque sigue el hilo argumental de lo explicado en el apartado anterior con la función *MessageHandler()*.

Lo primero que hace esta función es realizar una **copia del scan más antiguo** que hay almacenado en el buffer. La **posición del buffer** que debe leer la función *processScan()* la indica la variable ***next_proc***. Esta variable apunta siempre a la posición del *buffer* que contiene el paquete más antiguo recibido del LD-MRS y que no se ha escrito todavía en el fichero de salida.

Una vez conocida la posición que se tiene que leer, se accede a la variable para poder así realizar la copia de la información del *scan* que contiene. Ya se dijo que las posiciones del *buffer* son **variables protegidas** por un mecanismo de exclusión tipo **mutex**. Si la función *messageHandler()* no está escribiendo simultáneamente en la posición que se quiere leer, se accede a la misma y se realiza la **copia** de la información que contiene en una **variable local** de la función *processScan()*; esta variable local sirve como paso intermedio entre el *buffer* y el fichero de salida. La posición que se ha leído queda libre para poder almacenar más mensajes que lleguen desde el LD-MRS. La variable ***next_proc*** se **ajusta** para que apunte a la otra posición del *buffer*, que será la leída en la próxima iteración.

A continuación se escribe en el **fichero de salida** la información del *scan* que se ha copiado en la variable local. Se genera un único fichero de salida **por cada ejecución** del programa de adquisición de datos. Éste fichero se estructura por *scans* o barridos, llevando cada uno de esos scans la información asociada de posición y medidas inerciales de la aeronave, la información correspondiente a *scans* consecutivos se separa mediante caracteres de retorno de carro.

4.2.2.1. Fichero de salida

Se genera un fichero de salida por cada ejecución del programa de adquisición de datos, teniendo éste una **estructura determinada** y fija. Éste fichero se organiza en *scans* individuales, separándolos unos de otros

mediante caracteres de retorno de carro. Las primeras líneas que corresponden a cada *scan* recogen información general del mismo, esta información aparece recogida en la Tabla 13.

Información	Descripción
Numeración del <i>scan</i>	En cada ejecución del programa los <i>scans</i> se numeran de forma incremental.
Marca de tiempos inicial	Marca de tiempo del primer pulso láser que se envía en el barrido (primer punto).
Marca de tiempos final	Marca de tiempo del último pulso láser que se envía en el barrido (último punto).
Tiempo activo del <i>scan</i>	Diferencia entre las marcas de tiempo final e inicial.
Flags	Banderas asociadas a las características del <i>scan</i> recibido.
Puntos	Número de puntos de los que está compuesto el <i>scan</i> .

Tabla 13. Información general del *scan* en el fichero de salida

La información correspondiente a la **posición de la aeronave y las medidas inerciales se imprime también** en el fichero, es decir, la información que se extrajo de la zona de memoria compartida en el momento en el que se recibió el *scan*. Al incluir esta información en el fichero de salida se hace con el formato más reducido, pero con la precisión necesaria para que en los posteriores cálculos del procesamiento de datos no arrastren errores de redondeo.

Con respecto a la **precisión** de las variables, los datos correspondientes a la posición y velocidad se imprimen con **8 dígitos decimales**. Para determinar cuántos dígitos decimales son necesarios en los cálculos se ha tenido en cuenta que un grado de latitud (la posición viene dada en latitud, longitud y altitud) equivale aproximadamente al orden de 10^5 m (unos 100 km). Algo parecido ocurre con la longitud. De manera que 10^{-5} grados de longitud o latitud equivaldrían al orden de las unidades de metros, es decir, que con cinco decimales en las variables que almacenan la información de la latitud y la longitud basta para tener la posición en el orden metros. El receptor GPS que se está usando en el dispositivo MTi-G tiene un error aproximado de 2.5 m, por ello se incrementó la precisión de las variables latitud y longitud del programa a 8 decimales, esto hace que no interfieran los redondeos de estas variables en el error de la medida, siendo éstos despreciables.

El siguiente paso es imprimir en el fichero la información propia de cada *scan*. **Cada *scan*** está compuesto por un número (más o menos variable) de **puntos o coordenadas**. Cada uno de esos puntos a su vez tiene un conjunto de valores asociados que indican las características del mismo (banderas). En el fichero de salida se utiliza una línea distinta para la información correspondiente a cada punto. Las características que acompañan a cada punto se muestran en la Tabla 14.

Característica	Definición
<i>Point num</i>	Numeración de los puntos que pertenecen a este <i>scan</i> .
<i>Cartesian</i>	Coordenadas cartesianas del punto en el sistema de referencia del LD-MRS (<i>Body b-frame</i>).
<i>EchoWidth</i>	Anchura del pulso láser recibido tras su reflexión. Útil para determinar falsas reflexiones.
<i>Flags</i>	Banderas de clasificación (punto reflejado en polvo, lluvia, etc.) (Figura 41).
<i>Channel & Subchannel</i>	Entre ambos determinan el plano de la apertura vertical al que pertenece el punto.
<i>Offset</i>	Retraso del punto en cuestión, con respecto a la marca de tiempos inicial del <i>scan</i> .
<i>Valid</i>	Resumen de las banderas del punto, con valor igual a “1” cuando el punto no tiene ninguna bandera activada.
<i>Segment</i>	Campo usado para otras aplicaciones (no se utiliza).

Tabla 14. Características de los puntos de cada *scan*

Después de la lista de puntos que componen el *scan* se imprimen dos **líneas en blanco**, para separar visualmente (y con vistas también al procesamiento posterior) un *scan* y su consecutivo.

Public Types

```
enum Flags {
    FlagGround = 0x0001, FlagDirt = 0x0002, FlagRain = 0x0004, FlagThresholdSwitching = 0x0010,
    FlagReflector = 0x0020, FlagLeftCovered = 0x0100, FlagRightCovered = 0x0200, FlagBackground = 0x0400,
    FlagMarker = 0x0800, MaskInvalid = FlagGround | FlagDirt | FlagRain | FlagBackground, MaskCovered = FlagLeftCovered | FlagRightCovered
}
```

Figura 41. Banderas posibles de cada punto de un *scan* (18)

Se buscan dos cosas con este formato del fichero de salida, una es que se ocupe el mínimo tamaño posible en memoria para que su almacenamiento no sea problemático (mucho tiempo para escribirlo y para almacenarlo), y otra, que sea fácil su posterior lectura para el procesamiento de los datos, de manera que cada uno de los campos del fichero se pueda reconocer sin ambigüedad.

El tamaño del fichero de salida puede llegar a ser problemático si no se controla, dado que la aplicación de adquisición de datos genera gran cantidad de información (alrededor de 700 puntos en un *scan* cada 80 milisegundos, sin incluir la información de sus banderas asociadas e información propia del MTi-G). Esto se debe tener en cuenta ya que la tarjeta de procesamiento almacena esta información en una tarjeta de memoria SD y en un vuelo de adquisición de datos prolongado podría superarse la capacidad máxima de una

tarjeta SD estándar. Por esta misma razón se hace necesario el uso de una tarjeta de memoria de alta capacidad y con una velocidad de escritura que sea superior a la velocidad a la que la aplicación escribe en ella.

La función *ProcessScan()* termina aquí su tareas. En la Figura 42 se puede ver un ejemplo del fichero de salida que genera este programa.

```

Scan 1
ScanStart 2008-12-15T00:11:20.178114 (683161 ms)
ScanEnd 2008-12-15T00:11:20.200396 (683183 ms)
ScanActiveTime(ms) 22
Flags 4
Points 738

UTC Y2010 M09 D10 07:30:55 m000077
Euler [-0.49730527,0.76570410,-138.17576599]
LLA [37.42611694,-6.00321293,62.19619370]
Vel [-0.27509207,-0.05818643,0.13713859]
Status 7
Valid 1

Point num; Cartesian(x,y,z); EchoWidth; Flags; Channel; Subchannel; Offset(us); Valid; Segment;
1; 0.05844890,0.06843486,-0.00062831; 1.24; 18; 1; 0; 0; 0; 0
2; 0.02623660,0.03018176,-0.00083770; 1.08; 2; 0; 0; 0; 0; 0
3; 0.09840645,0.11320368,-0.00104719; 1.56; 2; 1; 0; 0; 0; 0
4; 0.01987424,0.02246374,-0.00062827; 0.80; 18; 0; 0; 0; 0; 0
5; 0.09276454,0.10485125,-0.00097738; 1.48; 18; 1; 0; 0; 0; 0
...
...
...
730; 0.10494641,-0.12072705,-0.00335079; 1.44; 2; 0; 0; 0; 0; 0
731; 0.07216473,-0.08301603,-0.00076794; 1.64; 2; 1; 0; 0; 0; 0
732; 0.15013108,-0.17423570,0.00160569; 1.52; 2; 2; 0; 0; 0; 0
733; 0.14357564,-0.16662775,0.00460733; 1.60; 2; 3; 0; 0; 0; 0
734; 0.06493056,-0.07602391,-0.00209424; 1.16; 18; 0; 0; 0; 0; 0
735; 0.09092051,-0.10645424,-0.00097738; 1.48; 18; 1; 0; 0; 0; 0
736; 0.16798812,-0.19843559,0.00181513; 1.20; 18; 2; 0; 0; 0; 0
737; 0.14211610,-0.16787432,0.00460733; 1.36; 18; 3; 0; 0; 0; 0
738; 0.00000000,0.00000000,0.00000000; 0.00; 0; 0; 0; 0; 0; 1; 65535

Scan 2
ScanStart 2008-12-15T00:11:20.258303 (683241 ms)
ScanEnd 2008-12-15T00:11:20.280564 (683263 ms)
ScanActiveTime(ms) 22
Flags 1028
Points 739
UTC Y2010 M09 D10 07:30:55 m000147
Euler [-0.48104489,0.76557422,-138.17008972]
LLA [37.42611694,-6.00321293,62.20447159]
Vel [-0.28081167,-0.06082192,0.13890338]
Status 7
Valid 1

Point num; Cartesian(x,y,z); EchoWidth; Flags; Channel; Subchannel; Offset(us); Valid; Segment;
1; 0.05844890,0.06843486,-0.00062831; 1.24; 18; 1; 0; 0; 0; 0
2; 0.05247321,0.06036352,-0.00167539; 1.08; 2; 0; 0; 0; 0; 0
3; 0.09840645,0.11320368,-0.00104719; 1.60; 2; 1; 0; 0; 0; 0
4; 0.01987424,0.02246374,-0.00062827; 0.80; 18; 0; 0; 0; 0; 0
5; 0.09276454,0.10485125,-0.00097738; 1.48; 18; 1; 0; 0; 0; 0

```

Información del LD-MRS

Información del MTi-G

Puntos que componen un barrido o *scan* completo

Línea de separación entre un *scan* y el siguiente

Figura 42. Ejemplo de fichero de salida

4.2.3. Main

La función *main*, aunque se explica la última, es el **punto de partida** del programa y se encarga de la configuración del LD-MRS. También se encarga de comunicarse, como ahora se verá, con las funciones *messageHandler()* y *processScan()* de forma que realicen cada una su tarea de manera sincronizada.

Una tarea particular de la función *main*, y de este código a su vez, es el **control de la duración de su ejecución** y por tanto del proceso de adquisición de datos. Ésta duración **la elige el usuario en el menú inicial**, expresando los minutos que desea que dure, aunque la gestión interna de la aplicación del control del tiempo sea en segundos. La función *main* se encarga de iniciar unas variables al inicio de la ejecución que almacenarán el tiempo que lleva consumido la aplicación desde que comenzó su ejecución. Se comprueba el valor de éstas variables cada vez que se ejecuta el bucle principal del código (se describe más adelante), de manera que cuando se alcanza la duración fijada por el usuario se detiene la ejecución y se termina el programa.

4.2.3.1. Zona de memoria compartida

Debido a que este proceso de adquisición de datos del LD-MRS va a compartir una zona de memoria con el proceso del MTi-G, es necesario **definir de nuevo**, para el mapa de direcciones de este proceso del LD-MRS, la **zona de memoria compartida y el semáforo** que se encarga del control del acceso a esta zona de memoria.

Las líneas de código que realizan esta tarea son prácticamente iguales a las que se utilizaron en el código del dispositivo MTi-G, por lo tanto no se dedicará más tiempo a explicarlas.

Tras la definición de la zona de memoria compartida se entra en un **bucle** que comprueba los valores de la misma. Ya se comentó que el MTi-G escribía en una de las posiciones de esta zona de memoria el valor de una variable llamada *valid*. Cuando la variable ***valid* tiene el valor “1”**, significa que la información del MTi-G es fiable, y por consiguiente los valores de la zona de memoria compartida son válidos. Sin embargo, durante el proceso de inicialización del MTi-G esta variable suele tener el valor “0”. En el momento en el que se detecta que la variable cambia al valor “1” se termina el bucle y **comienza el proceso de configuración del LD-MRS**, y la adquisición de datos.

4.2.3.2. Configuración del LD-MRS

De la misma manera que ocurría con el dispositivo MTi-G, lo primero que se realiza en la configuración del LD-MRS es **establecer la comunicación entre el dispositivo** y la tarjeta de procesamiento. Para ello el LD-MRS, que se conecta mediante el estándar **Ethernet**, dispone de una dirección IP a la cual se lanza una petición de comunicación desde la aplicación.

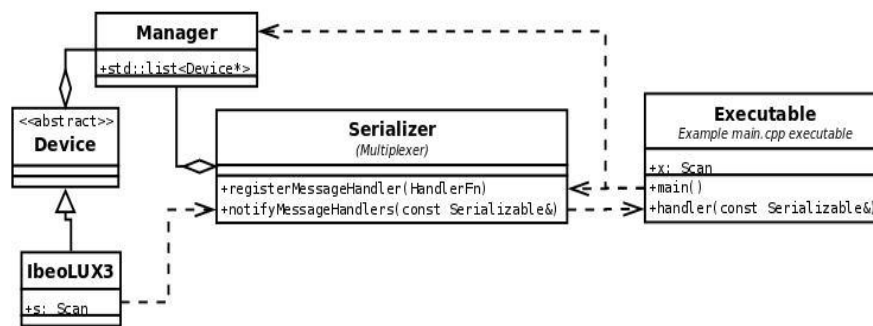
Una vez está establecida la conexión con el LD-MRS se procede a definir los **objetos y métodos software** que se encargarán de encapsular al hardware del dispositivo y a sus eventos (envío de paquetes de información, mensajes de alarma, etc.). La API del LD-MRS proporciona para ello una estructura de programación de clases que ayuda al desarrollador en esta tarea.

Detailed Description

Main class that handles devices and connections from and to the IbeoAPI.

This management object has the following tasks:

- It creates each **Interface** and **Device** object based on a configuration file and controls their object life cycle.
- It connects devices to their respective interfaces.
- It contains one **Serializer** to which all created Devices will send their messages to.
- It accepts TCP connections requests on the port number **ibeo::StandardPort** from an external ("client") program through an **IbeoAcceptor** object. If such a connection has been established, all messages from the Manager's **Serializer** will be forwarded to that connected client according to its chosen **DataTypeFilter**.



Pseudo-UML diagram of the Manager and its Serializer

Figura 43. Diagrama UML de la estructura de clases para programar el LD-MRS (18)

El LD-MRS sigue un procedimiento establecido para comunicarse con la tarjeta de procesamiento y enviarle mensajes con los datos. Este procedimiento se refleja también en una jerarquía de clases y objetos (hablando desde el punto de vista del software) como la que se ve en el diagrama UML de la Figura 43. Como se aprecia es necesario definir un **objeto** de la clase **Manager**, el cual funcionará como **nexo de unión software** entre el dispositivo físico que se encuentra al otro lado del cable Ethernet y la función **MessageHandler()** de la aplicación. Este objeto se encarga de la gestión de los dos hilos del programa, de forma totalmente transparente para el programador, haciendo que todos los mensajes que envía el LD-MRS pasen por un objeto de tipo **Serializer** (que hace las veces de multiplexor de los mensajes que van y vienen) antes de llegar a despertar a la función **MessageHandler()** del código.

Las características de configuración del LD-MRS se introdujeron ya en un capítulo previo relativo a los componentes del sistema. Concretamente, en este proyecto se ha configurado el LD-MRS con una **apertura horizontal de 100°** (desde 50° a la izquierda hasta 50° a la derecha de la normal de la ventana del LD-MRS). La apertura vertical que se usa viene determinada por defecto y se corresponde con los **cuatro planos verticales** en los que el LD-MRS emite pulsos láser, con la peculiaridad de que sólo se usan dos de los cuatro planos (los dos inferiores, con numeración 1 y 2) en la franja de apertura horizontal izquierda **desde los 50° hasta los 35°**. En esta franja sólo se pueden usar esos **dos planos verticales** por construcción interna del mismo dispositivo.

La frecuencia de funcionamiento del dispositivo, es decir, **la frecuencia a la que se realizan los barridos** que se ha elegido es la menor, **12.5Hz** con una **resolución angular constante** de **0.25°** entre pulsos láser. Elegir frecuencias superiores supone perder resolución angular, lo que no interesa. Al ser una frecuencia baja, la sincronización con el MTi-G puede realizarse de manera más sencilla y facilitando también la generación de los ficheros de salida sin que la llegada de mensajes del LD-MRS desborde el *buffer* de recepción.

El LD-MRS tiene varios modos de funcionamiento. De entre ellos nos interesan dos, el modo de configuración, en el cual se pueden establecer los parámetros que se han descrito en este apartado y el modo de medida, en el cual el LD-MRS está realizando barridos a la frecuencia programada y enviándole los *scans* a la aplicación. Durante todo el proceso de configuración el dispositivo no realiza barridos porque por defecto se inicia en modo configuración

4.2.3.3. Fichero de salida

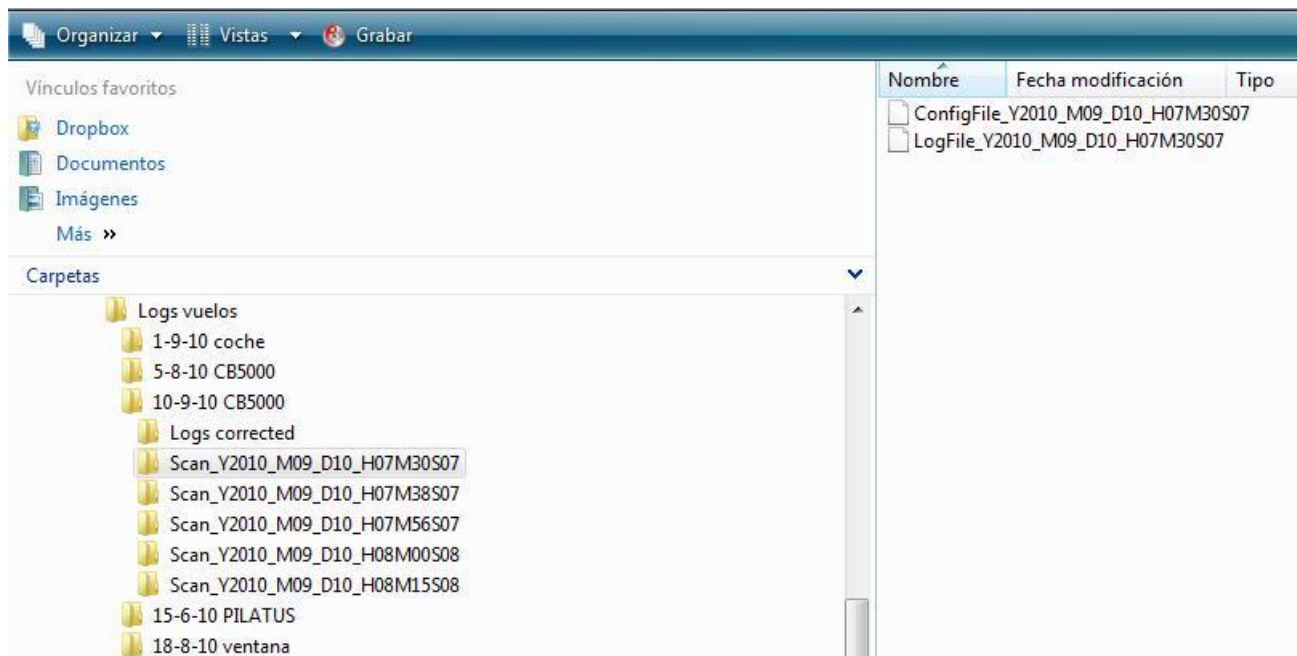


Figura 44. Ejemplo de la estructura de carpetas y ficheros de salida

El programa que se está describiendo genera un fichero de salida, el mismo que se ha mostrado en un apartado anterior. Este fichero se guarda en una carpeta que el programa crea específicamente para tal efecto, la cual recibe el nombre de **Scan_[fecha]**, y va seguida de la marca de tiempos que hay en la zona de memoria compartida al inicio del programa. El fichero de salida recibe el nombre de **LogFile_[fecha]**.

Aparte, se crea otro fichero con parámetros de configuración del LD-MRS con vistas a una posterior depuración de los resultados. Este fichero recibe el nombre de **ConfigFile_[fecha]**. Un ejemplo de esta estructura de ficheros puede verse en la Figura 44.

4.2.3.4. Bucle principal

Una vez se ha configurado el dispositivo correctamente y se han creado y abierto las carpetas y ficheros necesarios, se procede a establecer el LD-MRS en modo de medida. Esto se realiza con la instrucción *measure()*. Durante todo el proceso de configuración anterior el dispositivo no realiza barridos porque por defecto se inicia en modo configuración.

Tras establecer el modo de medida se ejecuta el bucle principal del programa. Éste se usa para gestionar los posibles eventos que se dan durante la ejecución y para coordinar las funciones *messageHandler()* y *processScan()*, que se recuerda que pertenecían a hilos de ejecución distintos.

Los eventos que pueden aparecer durante el transcurso del programa son varios. De entre ellos los que más interesan se listan a continuación:

Timeout: Está presente en el código para determinar si en algún momento la ejecución del bucle se bloquea esperando que ocurra un evento y no ocurre ninguno. Si esto sucede, puede deberse a algún tipo de error interno del dispositivo, por este motivo se fuerza este evento. Éste se vale de un **temporizador** para terminar el programa. El tiempo de espera antes de forzar un evento *Timeout* son 15 segundos.

NewScanDataEvent: Este evento proviene directamente de la función *messageHandler()*, es una **señal interna** de la API del LD-MRS que actúa a modo de bandera y que se levanta **cuando llega un mensaje nuevo** al programa. Cuando se produce este evento **se llama** de forma inmediata a la función *processScan()* para que proceda con la escritura de la información del scan recibido en el fichero de salida.

CancelEvent: este evento permite **terminar el programa a petición del usuario**. Está controlado de manera interna en la API del LD-MRS, y hace que **cuando se pulsan las teclas Ctrl+C** se genere una señal interna que provoca la terminación del programa.

Los eventos anteriores son detectados mediante un método de la librería del LD-MRS conocido con el nombre de *g_events.wait()*. El flujo del programa espera en esa función que está al inicio del bucle hasta que se produce uno de los eventos, entonces sale de la espera y se analiza qué evento ocurrió mediante una estructura de control de flujo. Seguidamente se realizan las acciones correspondientes (por ejemplo, despertar la función *processScan()*, terminar el programa, u otras).

Dentro del bucle también, **se comprueba** en cada iteración si el **tiempo total de ejecución del programa** es superior al que definió el usuario en el menú inicial. Si eso es así se procede a terminar el programa.

Para **terminar** el programa se cierran los ficheros abiertos y se coloca al **LD-MRS en modo idle**, de manera que no siga emitiendo pulsos láser.

4.2.4. Pseudocódigo

En el anexo de este mismo documento se adjuntan diagramas de flujo que representan el pseudocódigo de esta parte del software, correspondiente al proceso del dispositivo LD-MRS.