3 LIBRERÍA DE ENLACE DINÁMICO (DLL)

En este apartado se describe interfaz de programación (API) de la librería de nelace dinámico (DLL) suministrada pro AICIA a ELIMCO Sistemas para el control de una antena de dos grados de libertad del fabricante Orbit.

3.1 Entorno de trabajo.

El cliente marcó como herramienta de trabajo el entorno de desarrollo de Microsoft, Visual Studio, en su versión 6.0. Como es conocido, C++ es una extensión del estándar ANSI C que soporta programación orientada a objetos.

La versión 6.0 del entorno estaba orientada a mantener el entorno en una evolución paralela a los nuevos escenarios informáticos (especialmente, al desarrollo de aplicaciones para internet y software para sistemas de arquitectura distribuida).

El generador de proyectos del entorno contiene como opción la creación de aplicaciones de librerías de enlazado dinámico, esto es, de tipo DLL. Como ya se ha comentado, el enlazado dinámico de una librería permite cargar y enlazar en tiempo de ejecución bibliotecas construídas a tal efecto. Esto además permite que diferentes aplicaciones compartan una misma DLL, ahorrando tanto en memoria como en espacio en disco. Además, proporciona una mayor modularidad en la programación y en del desarrollo de aplicaciones. Esto, como ya se ha comentado, era de especial interés para el cliente, ya que podrá usar la DLL desarrollada para futuras aplicaciones de control, sin preocuparse de los detalles de la programación en sí, simplemente invocando los servicios que provee dicha librería.

3.2 Lista de funciones que intervienen en el interfaz

La aplicación DLL desarrollada para este proyecto consta exclusivamente de tres puntos de entrada, los cuales son:

- DLL
- inicioDLL
- finDLL

Todos los parámetros de entrada de la DLL se muestran en la tabla adjuntada en el anexo I.

Las órdenes al sistema pedestal y las consultas de estados, BIT, etc… se llevan a cabo seleccionando adecuadamente los parámetros que reciben estas funciones. En particular, DLL es la más importante para el funcionamiento, ya que ella se encarga de mapear todas la funciones relevantes del protocolo de comunicaciones de Orbit con los diferentes comandos de control

implementados. En la siguiente figura se puede ver un esquema con los puntos de entrada a la DLL.

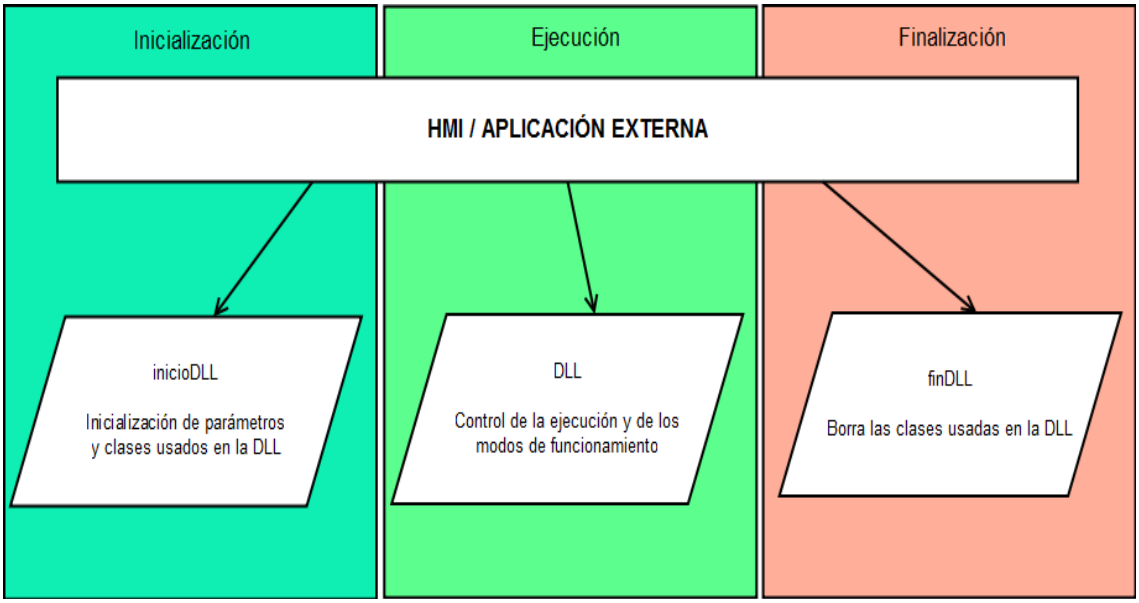

Figura 12: Esquema básico de uso y llamada de la DLL

3.3 Punto de entrada principal: DLL

Antes de describir los parámetros de entrada y salida a esta función y los posibles comportamientos cabe incidir en el hecho de que el de Orbit es un protocolo de comunicaciones full duplex y asíncrono, lo cual implica que la mayoría de las tramas se reciben sin que se solicite previamente, por lo que es preciso vaciar periódicamente los buffers de recepción.

Además, las órdenes enviadas (conocidas como PDO) son un servicio sin confirmar. E incluso aunque se trabaje con tramas con confirmación o respuesta (servicio PDO), la lectura de éstas se intercala con el flujo de paquetes periódico y por tanto los comandos pregunta-respuestas hay que integrarlos con el funcionamiento global de lectura de buffer-interpretación de tramas.

La llamada a la función principal de la DLL sería:

```
int __stdcall DLL(void * ig1, void * ig2, void * ig3, long Modo_Op, long Comando))
```

Los tres primeros parámetros son punteros a estructuras, de tipos variables dependiendo del comande que se trate. Los detalles de estas estructuras se comentan más adelante. El parámetro *Modo_op* indica en qué modo de operación se está, mientras que el modo comando es usado para seleccionar el comando que se desea ejecutar. En los siguientes apartados se comentan los detalles de dichos parámetros.

3.3.1 Descripción de los modos de funcionamiento

El manejador principal de comandos (es decir, la función DLL) gestiona los diferentes modos de operación, como los modos PDO, BIT, SDO o search. Esta función será llamada con unos determinados parámetros, que serán diferentes según el modo en que se esté invocándo.

Existen cuatro modos de operación del sistema, los cuales serán seleccionados según el valor del parámetro *Modo_op*:

MODO_PDO➔ 1
MODO_SDO➔ 2
MODO_BIT➔ 3
MODO_SEARCH_CFG➔ 5

Cabe destacar que, si bien el punto de entrada de la DLL es siempre el mismo para estos modos, la función demultiplexa el modo mediante una bifurcación tipo *switch* e invoca a la función de control correspondiente (una diferente para cada modo).

Modo PDO:

Mediante este modo se controlan las funciones básicas de la antena. Tanto la monitorización del estado como los comandos de posición y velocidad en elevación y acimut serán controlados en este modo aquí. El funcionamiento es el siguiente:

El parámetro Modo_Op se ha pasado con un valor MODO_PDO. En este caso, la función DLL llama a otra función, llamada *VControl*, que será la que demultiplexe el evento que sucedió.

El parámetro comando codificará que evento provocó la invocación de la DLL. Se pueden diferenciar dos tipos básicos: de estado y de comando. Si bien el de estado sólo es uno, y el resto es de comando, la importancia del primero justifica una explicación exhaustiva.

La función DLL usada para monitorizar el estado actual del sistema pedestal espera ser llamada cuando vence un timer programado por la aplicación externa que la invoca. Es de recalcar, pues, que debe ser la aplicación del usuario quien programe el timer y lance la dll con el comando adecuado.

Al hacer la llamada con *comando = EVENTO_TIMER*, el programa llamara a la función *OnTimer(int)*. Dicha función realizará las siguientes tareas:

- Leerá del puerto serie la información que allí haya, para poder procesarla adecuadamente

- A través de la función *evalmessage*, interpretará las diferentes tramas recibidas y modifica los valores necesarios

- Enviará los paquetes que hubiera pendientes.

- Una vez se retorna de esta función, está habrá modificado los campos de estado de las estructuras de elevación o acimut.

A pesar de que en principio la DLL se diseña como una aplicación genérica, el hecho de estar orientada principalmente a actuar con una aplicación tipo interfaz (ya sea la HMI desarrollada en el presente proyecto o cualquier otra aplicación que un potencial usuario pueda desarrollar) justifica que se realice con un enfoque claramente marcado a facilitar dicha tarea. De esta forma, en el apartado de comandos, existen diferentes eventos que provocan la actuación sobre la DSA:

- Pinchar sobre los radio buttons de los diferentes modos de control (elevación o azimuth).

- Deseleccionar el campo de texto en el que introducimos un determinado valor de posición o velocidad para azimuth o elevación.

- Utilizar las flechas tipo cursores de la pantalla.

- Pinchar sobre “puesta a cero”

- Pinchar sobre el modo search

- Pinchar sobre “parada de emergencia”

La segunda opción mandará el comando introducido en el campo deseleccionado. Las flechas mueven la antena al ser pinchadas, y paran al ser levantadas. Puesta a Cero realiza una puesta de los ejes en la posición inicial, y parada de emergencia realiza una parada. Un análisis más extenso de estas características es desarrollado en el apartado dedicado a la HMI del presente documento.

Podría entenderse que en estos primeros casos recién citados podemos encontrar las funciones que nos proporcionarían un control más directo sobre la DSA, ya que envían el comando deseado e introducido en los campos de las estructuras correspondientes. Como ya se ha comentado, se profundizará sobre las estructuras más adelante.

En los controles de flecha, se pasa como comando de posición o velocidad el valor del *slide bar* de la aplicación de Visual Basic que se encuentre en ese momento, relativo a un determinado máximo. Por otra parte, en el modo “parada de emergencia” el pedestal decelera hasta velocidad cero, tras lo cual

se apagan los motores. Finalmente, los eventos *lostfocus* son similares a los comandos, sólo que se ejecutan al cambiar el foco del cuadro de texto.

MODO SDO:

En este modo, se pueden usar una serie de comandos para configurar la DSA. Estos comandos están explicados en el apartado dedicado al protocolo de control de Orbit, del presente documento, además de ser profundamente analizados en el documento [1].

MODO BIT:

Como ya se ha comentado en el apartado dedicado al protocolo de Orbit, el sistema pedestal posee un modo de test, Built-In-Test, que registra determinadas variables para conocer el estado interno del sistema. El BIT permite monitorizar cuatro palabras de parámetros pre-programados. Según el valor de comando podremos:

- Inicializar el BIT de acimut o elevación.
- Finalizar el BIT de acimut o elevación.
- Limpiar la información del registro latente (latch). En el BIT existe un registro actual, que se va actualizando en cada consulta, y un registro latente, que mantiene los bits de error activos aún cuando ya se subsanó el error. Mediante ese comando, lo se limpia este último.
- Comprobar el estado de los temporizadores ya que, en caso de que hubieran vencido, indicaría que existe un error en la comunicación

MODO CONFIGURACIÓN SEARCH:

En este modo se realiza la configuración de los parámetros que usará *el modo search* para realizar la búsqueda del punto destino. Para ello se puede, o bien leer la información de configuración de un fichero (*comando* = *EVENTO_SEARCH_SETUP_LEER_FICHERO*), o bien pasarla como parámetro a través de la estructura correspondiente.

Comunicación con la DLL: Las estructuras:

El intercambio de información entre la aplicación externa y la DLL está pensado para ser llevado a cabo mediante la información almacenada en unas estructuras de memoria. Las estructuras serán pasadas como parámetros y se modificarán o leerán en la DLL. Se comentan los pormenores más adelante.

3.4 Punto de entrada de inicialización: inicioDLL

La function que realiza implementa las funcionalidades necesarias al comienzo de la ejecución de la DLL es la siguiente:

```
int __stdcall inicioDLL(long puerto)
```

Como se ha comentado, se trata del punto de entrada llamado al inicio de la aplicación externa. Esta función recibe el manejador del puerto creado en la aplicación usuaria de la DLL. En la aplicación HMI, cuyos detalles se comentan en el próximo capítulo, la llamada se realiza en el código del formulario *FormConfPuerto:*

```
hserialportDSA = AbrirPuerto(PuertoConfDSA)
```

y posteriormente,

```
a = inicioDLL(hserialportDSA).
```

El parámetro *puerto* es copiado a la variable global puertoserie, que almacenará en sucesivas ejecuciones de la dll el valor del manejador para las comunicaciones con la DSA. Posteriormente se utiliza la clase theUCom, diseñada para las comunicaciones, para realizar una conexión (apertura del puerto serie). Una vez realizada la conexión, se pasa a la inicialización de los parámetros de configuración del acimut. Estos parámetros reciben unos valores por defectos, aunque también pueden ser alterados mediante el fichero SetUp.prm. Se realiza una inicialización de la máquina de estados del protocolo, inicialización que sólo es necesaria en la primera llamada a la dll.

Parámetros de entrada	**Descripción**	**Detalles**
long Puerto	Manejador del puerto serie creado en una aplicación externa.	Se asigna a la variable global puertoserie, estática para sucesivas llamadas de dll60.dll
Parámetros de salida	**Descripción**	**Detalles**
	La función devuelve 1 si se cargó SetUp.prm, 0 en caso contrario	SetUp.prm es un archivo de configuración de los parámetros de azimuth

3.5 Punto de entrada de fin de ejecución: finDLL

En esta ocasión, se trata de la función destinada a cerrar la ejecución de la DLL.

```
int __stdcall finDLL(void)
```

Se trata del punto de entrada llamado al finalizar definitivamente una aplicación externa. Borra las clases *ElBitMonitor* y *AzBitMonitor*, usadas para la monitorización del Build-In-Test de la DSA de la antena. No recibe ningún parámetro y siempre retorna 1.

3.6 La comunicación con aplicaciones externas.

La comunicación se realiza a través de unas estructuras que alamacenarán todos los datos a intercambiar. De esta manera tanto la DLL como las aplicaciones que hagan uso de sus recursos irán modificando los valores en memoria de las variables que componen las diferentes estructuras.

Existen un total de siete estructuras, que engloban funcionalidades asociadas a interfaces gráficas, parámetros y comunicaciones. En la siguiente tabla se muestra un resúmen de sus características y funcionalidades.

Tabla 8: Estructuras de la DLL

Estructura	Función
graficos	Variables asociadas a la ventana de Control de la HMI. Incluye estados de los DSA y modos de control
graficosSDO	Variables asociadas a la ventana de SDO de la HMI. Almacena eje al que comunicar y parámetros actualizados o leídos.
graficosBIT	Ventana de BIT de la HMI. Contiene banderas *todo/nada* del Built-In-Test.
setupparams	Guarda los parámetros iniciales de la DSA.
graficosSRCH	Parámetros del modo Search
graficosComun	Dos variables para comprobar si se está actualizando todo correctamente y si el control está en modo manual
TipoPuertoConf	Almacena la configuración del puerto de comunicaciones PC-DSA o PC-Banco de flitros

La codifiación de estas estructuras se muestra a continuación.

```
struct graficos{
    long EstadoCCWLimit ;
    long EstadoS_CCWLimit ;
    long EstadoCWLimit ;
    long EstadoS_CWLimit ;
            float EstadoPosicion ;
    float EstadoVelocidad ;
    long EstadoModo ;


    long ComandosMdStandBy ;
    long ComandosMdPunto ;
    long ComandosMdVelocidad ;
```

```
        long ComandosMdSrch ;
        long ComandosMdPosErr ;
                    float ComandosPosicionVal ;
        float ComandosVelocidadVal ;
            //        long ControlManual;
    };



struct graficosSDO{
            long    Eje ;
            long    Indice ;
            long    Subindice ;
            long    TipoDato ;
            long    ValorLeer ;
            long    ValorEscribir ;
            long    Respuesta ;
    };



struct graficosBIT{
        long BITSafeSwitch ;
        long BITPhaseCurrentSum ;
        long BITHPhaseA ;
        long BITHPhaseB ;
        long BITHPhaseC ;
        long BITMotorCurrent ;
        long BITLowBusVoltage ;
        long BITHighBusVoltage ;
        long BITHighBusCurrent ;
        long BITLowBrkCurrent ;
        long BITHighBrkCurrent ;
        long BITRefVoltage ;
        long BITEncoderACount ;
        long BITEncoderBCount ;
        long BITEncoderAIndex ;
        long BITEncoderBIndex ;
        long BITHome ;
        long BITLimitDown ;
        long BITMotorVelocity ;
        long BITMotorLoadPos ;
        long BITInternalTemperature ;
        long BITInternalHumidity ;
        long BITMotorTemperature ;
        long BITBridgeTemperature ;
        long BITMotorBimetal ;
```

```
        long BITPowerBridgeLockOut ;
        long BITInternalPwrSupply ;
        long BITFlashError ;
        long BITRsrv1 ;
        long BITRsrv2 ;
        long BITRsrv3 ;
        long BITRsrv4 ;
        long BITActualizaOk;
};



struct setup_params{
          long    m_gear_ratio;
          long    m_load_enc;
          float   m_max_acc;
          float   m_max_vel;
          long    m_motor_enc;
          float   m_offset;
          float   m_home;
};



struct graficosSRCH{

   float m_az_cen;
   float m_az_sec;
   float m_az_stp;
   float m_az_vel;

   float m_el_cen;
   float m_el_sec;
   float m_el_stp;
   float m_el_vel;

   long Type;
          long UpdVia;

};


struct graficosComun{

      long ControlManual;
      long ComunicacionOk;
};
```

```
// Parametros de configuracion del puerto serie
struct  TipoPuertoConf{
   long numeroCOM; /* COM1=1, COM2=2...*/
   long baudios;
   long bitsdatos;
   long bitsparada;
   long paridad;
};
```