4 LA APLICACIÓN DE CONTROL DEL SISTEMA

4.1 Características generales de la aplicación

La aplicación de control del sistema se denomina “Aplicación Levitador Neumático” y el fichero ejecutable que la contiene se denomina **mypint.exe.**

La aplicación informática, que se ha realizado para este proyecto, es una aplicación de 32 bits que funciona bajo el sistema operativo Windows 95, Windows NT o también bajo Windows 98.

Para el correcto funcionamiento del programa, el ordenador debe tener instalada una tarjeta de entrada / salida de la marca DATA TRANSLATION, en particular, se ha probado el programa para el modelo de tarjeta DT2811, aunque el programa es operativo para cualquier otro modelo de tarjeta entrada / salida del fabricante mencionado que pertenezca a la familia DT30xx o DT28xx (en el caso de los 28xx ha de ser un modelo superior al 2811, es decir, del 2811 en adelante).

Asimismo, son necesarios varios ficheros adicionales que contienen las bibliotecas de enlace dinámico (DLL) de la tarjeta y que deben encontrarse en el mismo directorio que el fichero de la aplicación o en el subdirectorio “system” del directorio “Windows” del disco duro del ordenador. Estos ficheros son los siguientes:

- oldath16.dll y oldath32.dll
- olmemt16.dll y olmem32.dll
- olmem.dll
- oldaapi.dll y aldaapi32.dll
- vdtdad.386

Sin estos ficheros la aplicación no funcionará puesto que contienen las funciones de comunicación con la tarjeta que se utilizan en el programa.

Además del fichero ejecutable que contiene la aplicación se acompañan a este proyecto los ficheros que contienen el código del programa en Visual C++, de manera que se pueda modificar el programa a posteriori según futuras necesidades. El nombre, función y contenido de cada uno de estos ficheros se detalla en este mismo capítulo dedicado a la aplicación, pero en sus correspondientes apartados.

Aunque la aplicación de control del levitador neumático se ha realizado para este sistema en particular, se ha hecho de manera que sirva también para otros sistemas, siendo la única condición que la tarjeta de entrada / salida sea de la marca DATA TRANSLATION en algún modelo de los citados anteriormente; condición que viene impuesta por usar las bibliotecas particulares de este fabricante en las funciones de comunicación con la tarjeta.

En cualquier caso, el autor de la aplicación permite futuras modificaciones del programa para adaptar su uso a otros sistemas existentes en el departamento.

La aplicación de control es la encargada de controlar el sistema elevador neumático. Los pasos que se realizan en cada ejecución de esta aplicación son los siguientes:

1. El usuario abre la aplicación de control del levitador neumático.

2. La aplicación comprueba si existe una tarjeta instalada cuyo nombre coincida con el que se le ha indicado. Por defecto, busca una tarjeta modelo DT2811.

3. Si existe la tarjeta, procede a iniciar un diálogo con el usuario en la que solicita mediante diferentes ventanas de diálogo los datos que son necesarios para el experimento. Si la tarjeta no existe se genera un mensaje de error.

4. En la primera ventana pregunta el tiempo de muestreo, el tiempo final del experimento, y el tipo de control que se va utilizar, y que puede ser “Predefinido” o “PID”. Más adelante se comenta el significado de cada opción.

5. En la segunda ventana, pregunta el nombre del fichero donde se almacenarán los resultados del experimento. Esta ventana es igual a la que presenta, por ejemplo, WORD a la hora de guardar un documento, permitiendo desplazarse por el árbol de directorios en busca de un fichero en particular.

6. Se comprueba si el fichero existe, y, en tal caso, se pide confirmación antes de sobreescribirlo. Se comprueba que se puede abrir y escribir en el fichero, o, en caso contrario, se genera un mensaje de error.

7. En caso de haber seleccionado como método de control un PID se muestra una tercera ventana en la que se solicitan los valores de la constante proporcional, el tiempo integral y el tiempo derivativo.

8. Da comienzo el experimento. En cada tiempo de muestreo se captura el valor de la señal del sensor a través de la tarjeta de entrada/ salida, se calcula la señal de control según la ley de control elegida, y se saca el valor calculado a través de la tarjeta.

9. Al mismo tiempo una ventana presenta una gráfica de voltaje frente a tiempo donde se van representando los valores de la señal de referencia, la señal del sensor, la señal de control y la señal de error en valor absoluto. Cada tiempo de muestreo se dibuja un nuevo punto de cada una de estas señales que corresponde con su valor en ese instante de muestreo. No se representa directamente la altura en centímetros, para no dividir la pantalla en dos gráficas, una con valores en voltaje y otra con valores en centímetros, y poder aprovechar toda la pantalla para una sola gráfica.

10. El experimento termina cuando se llega al tiempo final. Se redibujan las señales de la pantalla con una línea continua, para facilitar su visión en pantalla. También, se almacenan los resultados en el fichero especificado, en un formato compatible con MATLAB.

11. Se puede escoger entre las opciones del menú del programa si se desea un nuevo experimento o, por el contrario, terminar y cerrar la aplicación.

12. En el primer caso, es decir, seleccionar un nuevo experimento, el proceso que se sigue es el mismo al anterior comenzando desde el paso cuatro.

13. En el segundo caso se cierra la tarjeta y se cierra la aplicación.

Deben hacerse algunos comentarios sobre algunos de los pasos anteriores.

Cuando se pregunta el tiempo de muestreo y el tiempo máximo, no se permite elegir valores que estén fuera de los límites. Si el valor está fuera de los límites se genera un mensaje de aviso y se vuelve a presentar la ventana de diálogo, de manera que el diálogo no terminará hasta que no se introduzcan valores permitidos a través de la ventana de diálogo.

Dichos límites son los siguientes :

- Una centésima de valor mínimo y un segundo de valor máximo para el tiempo de muestreo. El valor mínimo no puede ser inferior porque esa es la máxima velocidad de conversión A/D que admite la tarjeta DT2811. El valor máximo de 1 segundo es más bien indicativo, y podría ser modificado puesto que no está determinado por ningún elemento.

- Un segundo de valor mínimo y 2500 segundos de valor máximo. El valor máximo se ha fijado pensando en las limitaciones en resolución de la

pantalla, puesto que si se representase una cantidad excesiva de puntos (> 2500) no se distinguiría bien en la pantalla el comportamiento de las señales pues estarían muy comprimidas en el eje de tiempos. No obstante, estos límites pueden variarse modificando valores límite en programa. En un apartado posterior se indica cómo hacer estos cambios.

El significado de los métodos de control es el siguiente :

- **Predefinido** : significa que se aplica al sistema una señal ‘u’ de control predefinida. Esta señal, que está implementada dentro del programa es la que se ha usado durante el proyecto como señal de “calentamiento” del sistema. Es decir, es la señal que se aplica cuando el motor está en frío para calentarlo, puesto que como ya se explicó en el capítulo de identificación, es necesaria una etapa de calentamiento del sistema antes de empezar a trabajar con él.

- **PID** : significa que la ley de control que se usará para calcular la señal de control ‘u’ será la habitual de un PID. El propio usuario puede introducir los valores de las constantes del PID, aunque existen valores por defecto calculados en anteriores experimentos para permitir el control del sistema.

Además de estos dos métodos de control, durante la realización del proyecto se incluyeron dos modos más de control del sistema.

El primero de ellos es según un control predictivo en el que la ley de control que se aplicará será una ley fija análoga a la de un PID y donde los parámetros se han obtenido a partir de un modelo del sistema y de un control predictivo (GPC), del que se han derivado dichos parámetros del PID, resolviendo el problema usando una función en MATLAB. No se consideran restricciones, y el modelo del sistema considerado es el obtenido en el capítulo dedicado a la identificación del sistema.

El segundo de estos métodos es un PID, pero ajustando los parámetros del PID en función de la altura a la que se encuentra la plataforma en cada momento. En este caso, al igual que en el predictivo, el usuario no toma parte en la elección de la ley de control, que está prefijada.

Los resultados obtenidos con estos dos métodos de control se encuentran recogidos en el capítulo dedicado al control del sistema levitador.

Dado que en estos dos métodos el usuario no toma parte activa en la elección de la ley de control, se optó por eliminarlos de las opciones de tipos de control, una vez que se terminaron los experimentos del proyecto dedicados a probar controladores del sistema. No obstante, en el listado del código fuente de la aplicación se puede encontrar el código que corresponde a estos dos métodos intercalado como comentario, con el fin de mostrar cómo se realizaron ambos métodos.

Cuando se pregunta el nombre del fichero, el fichero que se elija debe tendrá la extensión ".m" tanto si se indica explícitamente en el nombre completo del fichero como si sólo se pone el nombre del fichero sin extensión. Se ha optado por forzar la extensión ".m" para indicar que el fichero es compatible con el formato de los ficheros MATLAB y porque la única utilidad de este fichero es procesar los resultados los experimentos con MATLAB.

En la ventana de diálogo de elección de tiempo de muestreo y de tiempo final, así como en la ventana de elección de parámetros de PID, se presentan valores por defecto. Estos valores por defecto, que son, por ejemplo, de 0.852 para el tiempo de muestreo, son los valores que se recomiendan para el sistema levitador neumático.

En particular, los valores del PID que están predefinidos son los de un PID que servía para controlar el sistema relativamente bien.

Antes de empezar el experimento es necesario "abrir" lógicamente la tarjeta y configurar los subsistemas de entrada y salida. Esto lo hace automáticamente el

programa y si encuentra algún error genera un mensaje de error indicando en qué punto concreto de la inicialización se produjo el error.

Igualmente, antes de cerrar la aplicación, el programa se encarga de cerrar la tarjeta y de informar de si se produjo algún error en el proceso y el punto en que se produjo.

Para facilitar la visualización del comportamiento del sistema, se ha programado la aplicación de manera que la ventana de presentación de las gráficas de las señales es reescalable a cualquier tamaño como cualquier otra ventana de Windows. Esto permite escoger el tamaño de la ventana, desde ocupar la pantalla completamente hasta reducirla o minimizarla si se desea usar la pantalla para otras aplicaciones.

Hay que recordar que la aplicación que se ha programado permite la multitarea, y, a la vez que se realiza el experimento se pueden estar utilizando otras aplicaciones, como por ejemplo MATLAB, y una sola aplicación no puede ocupar toda la pantalla en exclusiva.

La aplicación de control del levitador implementa un filtro particular para eliminar el ruido procedente de la señal del sensor. Este filtrado, cuya utilidad se explicó en el apartado referido al sensor, y también en el capítulo dedicado a la identificación del sistema, se basa en sobremuestrear la señal a cuatro veces más velocidad y aplicar después un filtro Butterworth paso de baja de orden cuatro a una frecuencia fija.

Dado que este filtro es una necesidad particular del sistema levitador y buscando que la aplicación pudiese tener un uso más general, el filtro, que es fijo, se puede habilitar o deshabilitar mediante una variable del programa, de tal manera que si se desea usar esta aplicación para otros sistemas el filtro pueda inhabilitarse sin dificultad.

Igualmente, en busca de una mayor generalidad, el rango de voltaje de la representación gráfica puede cambiarse de manera sencilla variando un par de líneas en

el programa. En el caso del sistema levitador, el rango es de 0 a 5 voltios. En un apartado posterior se detalla cómo hay que modificar el programa para cambiar el rango de voltaje de la representación gráfica.

La aplicación posee un menú principal con una serie de opciones. Todas estas opciones están disponibles en todo momento.

Dentro de la opción **Archivo** del menú principal existen dos opciones.

La primera opción es **Nuevo Experimento**, y sirve para comenzar un nuevo experimento. Se puede comenzar un nuevo experimento en cualquier momento, incluso cuando se está llevando a cabo otro experimento, pero, naturalmente, el experimento que se está realizando dejará de ejecutarse y será eliminado por el nuevo. Los datos que el primer experimento ya hubiera leído sí se almacenarán en el fichero elegido a tal efecto.

La segunda opción es **Cerrar Tarjeta**, y su función es cerrar los subsistemas de la tarjeta, detectando si existe algún fallo y avisando si se ha producido un error y en qué momento de la operación de cerrar la tarjeta ha ocurrido el error. Tras cerrar la tarjeta termina la aplicación, destruyendo la ventana de la aplicación. Cuando se elige Cerrar Tarjeta durante un experimento, el experimento termina inmediatamente, y los datos que se hubieran leído no son guardados en ningún fichero. Para evitar que el sistema quede funcionando con el último valor de señal de control del experimento abortado, antes de cerrar la tarjeta se sacan cero voltios por el canal de salida de la tarjeta, para parar el motor del ventilador.

En el menú principal de la aplicación existen dos opciones para poder ver los datos del experimento en cualquier momento.

Al seleccionar la opción **Tiempos** en el menú principal aparece en pantalla un cuadro de diálogo idéntico al que aparece al comenzar un experimento. En dicho diálogo se puede ver el tiempo de muestreo elegido, el tiempo final del experimento y el

tipo de control seleccionado. Este cuadro de diálogo es puramente informativo, y a través de él no se pueden cambiar los datos relativos al tiempo de muestreo, tiempo final o tipo de control seleccionado.

Al seleccionar la opción **Constantes** en el menú principal, aparece en pantalla un cuadro de diálogo idéntico al que se muestra cuando se tienen que elegir las constantes del PID en un control PID. En dicho cuadro se puede ver el valor de la constante proporcional, del tiempo integral y del tiempo derivativo. Como en el caso anterior, este cuadro de diálogo también es solamente informativo y a través de él no se pueden modificar las constantes de PID, que se eligieron al comenzar el experimento, si se había seleccionado PID como tipo de control.

En ambos casos, el hecho de mostrar o no en pantalla el cuadro de diálogo no afecta para nada al desarrollo del experimento, llevándose a cabo el muestreo de la señal y la salida de la señal de control de manera normal.

Otra opción que se puede seleccionar en el menú principal es **Repintar Todo**. Con esta opción se borra la ventana de la aplicación y se dibuja de nuevo la gráfica con todas las señales, pero con trazo continuo, y no como puntos, que es como se van dibujando las muestras a medida que se leen cada tiempo de muestreo. De todas maneras, al finalizar el experimento se repintan todas las señales con trazo continuo para facilitar la visión de las señales.

El color con que se pinta cada señal está predefinido. Sin embargo, se puede modificar de manera sencilla cambiando unas pocas líneas de código, de forma análoga a cuando se desea modificar la resolución en la pantalla. En el apartado dedicado al programa se verá detenidamente cómo hacer éste y otros cambios.

Por último, la aplicación tiene otra opción de menú que visualiza un cuadro de información sobre la versión y el autor del programa.

Todos los elementos del menú tienen teclas aceleradoras para poder activarse sin necesidad de utilizar el ratón.

Existen dos características del programa que pueden ser importantes en ciertos casos.

La primera característica puede ser importante un experimento de larga duración y es debida a la forma que tiene Windows de trabajar con los relojes.

Cuando se activa el salvapantallas de Windows, el sistema operativo detiene por su cuenta todos los relojes y temporizadores lógicos que estuvieran activos. Esto provoca que el temporizador que usa la aplicación del levitador también se detenga y se pare por tanto el desarrollo del experimento. Hasta que no se vuelva a activar la pantalla no se reanudará el experimento. Es decir, mientras la pantalla esté ocupada por el salvapantallas, el programa permanecerá en suspenso por estar suspendido el temporizador que se encarga de contar el tiempo de muestreo.

No obstante, esta característica, que al ser a nivel del sistema operativo no se puede solucionar en la aplicación, no es grave puesto que la propia aplicación está pintando en la pantalla cada tiempo de muestreo, con lo cual el salvapantallas no se activa ya que se está utilizando la pantalla. Sí puede afectar cuando se esté ejecutando en experimento pero se haya minimizado la ventana de la aplicación.

La segunda característica del programa se refiere a la manera de almacenar los datos en fichero. Windows utiliza el juego de caracteres ANSI para almacenar los ficheros de texto. La aplicación de control del levitador también usa ANSI para guardar los resultados del experimento.

Sin embargo, existen algunos programas como, por ejemplo, el bloc de notas de Windows, que usan OEM como tabla de caracteres. Esto provoca que, al abrir los ficheros que genera la aplicación del levitador algunos caracteres como el retorno de

carro aparezcan como símbolos diferentes en la pantalla, aunque MATLAB entiende perfectamente el contenido de estos ficheros.

4.2 PROGRAMACIÓN EN WINDOWS

Para poder entender cómo funciona la aplicación y por qué razones se ha realizado como se ha hecho, es imprescindible conocer antes algunos conceptos del sistema operativo Windows y de la programación en Windows, en especial, aquéllos en los que difiere sustancialmente de las características de MS-DOS, que era el sistema operativo corriente hace unos años.

Windows es actualmente el sistema más usado en los ordenadores personales de todo el mundo, principalmente por su facilidad de uso. Por el contrario, la programación de aplicaciones para Windows es bastante más difícil de lo que era la programación para MS-DOS.

Windows es un entorno de interfaz gráfico de usuario (GUI : Graphics User Interface), multitarea y basado en ventanas (“windows” en inglés) que se corresponden con programas.

Antes de seguir, conviene analizar esta frase. Windows es un entorno gráfico, es decir, las órdenes que da el usuario, o las opciones que elige no se teclean directamente, sino que se escogen de una serie de menús, listas desplegables o de cuadros de diálogo. Es decir, en cada momento, el usuario lo que hace es escoger entre una serie de opciones posibles, en vez de teclear órdenes como era habitual en MS-DOS.

Como consecuencia de ello, los programas en Windows no son secuenciales. Dado que se puede escoger cualquier opción de las presentadas en, por ejemplo, un menú la ejecución de un programa en Windows no puede seguir el orden lineal que era habitual en DOS, sino que se llega a una programación conducida por eventos y orientada a objetos, donde el programa se divide en pequeños módulos que definen la manera de responder a las acciones del usuario o del sistema (los eventos).

Windows es multitarea. Realmente, Windows no es totalmente multitarea, sino que existe la llamada multitarea cooperativa, en la cual todos los programas que estén funcionando deben permitir la multitarea, pero un programa puede acaparar el tiempo de cálculo del procesador. Esto suele ocurrir cuando se ejecutan aplicaciones DOS bajo Windows, puesto que las aplicaciones DOS no están preparadas para multitarea. En esos casos, el sistema operativo no puede echar a la aplicación, y el resultado más probable es que el propio sistema caiga, lo que se conoce por “cuelgue” del ordenador.

A la hora de programar hay que tener en cuenta que el sistema no estará dedicado en exclusiva a nuestro programa, y que se debe evitar acaparar el tiempo de la CPU, y evitar el acceso repetido a los puertos o a otros dispositivos que tengan un tiempo de acceso alto.

Windows funciona por medio de ventanas. Toda aplicación deben tener al menos una ventana para que el usuario pueda comunicarse con ella.

La principal ventaja de Windows es precisamente que ofrece una serie de componentes predefinidos que son iguales para todas las aplicaciones, facilitando la uniformidad y la facilidad de uso de los programas, ya que todas las ventanas se comportan de la misma forma y todas utilizan los mismos métodos para interactuar con el usuario, como son los menús descendentes, los botones, etc…

Una ventana típica de Windows tendrá siempre varios elementos que serán los mismos para todas las aplicaciones:

- **Barra de menús** : que contiene los menús disponibles para la aplicación, que se pueden activar a través del ratón o, a veces, con una combinación de teclas.

- **Botones de minimizar, maximizar y cerrar** : cuya función es, respectivamente, minimizar, maximizar y terminar con la aplicación. Al

maximizar, este botón se transforma en el de restaurar, que sirve para volver la ventana a su tamaño habitual.

- **Marco de la ventana** : que permite cambiar el tamaño de la ventana usando el ratón.

- **Área de trabajo** : es la parte de la ventana donde se colocan el texto y los gráficos, donde realmente se ven los resultados de las acciones del usuario. También se denomina área de cliente.

Existen otras partes como las barras de desplazamiento horizontal y vertical, la barra de herramientas, o la barra de estado pero que no siempre aparecen.

Windows se ocupa de manipular el tamaño, la posición y los controles de la ventana, mientras que es la propia aplicación la que gestiona el área de trabajo.

Para el desarrollo de programas, Windows ofrece una serie de funciones que permiten definir y utilizar componentes como menús, barras de desplazamiento,etc.. Estas funciones están incluidas en el SDK de Windows (SDK : Software Development Kit : Utilidades de desarrollo de programas)

Es aquí donde se produce la primera dificultad. Al ser Windows un sistema operativo mucho más complejo que MS-DOS, las funciones son muchas y muy complejas. Intentar programar directamente un aplicación por medio de las API es posible pero totalmente desaconsejable, y es la razón de que existan entornos de desarrollo como Visual C++ o Visual Basic, que automatizan en gran parte esta tarea descargando de trabajo al programador.

La siguiente característica de Windows es que se gestiona por mensajes. Cada vez que, por ejemplo, un usuario hace clic con el ratón sobre el botón de minimizar de una ventana, se genera un mensaje para notificar ese evento. Todos los mensajes generados se colocan en una gigantesca cola de mensajes que gestiona el sistema

operativo, siendo éste el que se dedica a repartir los mensajes a las aplicaciones que le corresponden. Todas las aplicaciones en ejecución en cada momento comparten esa cola de mensajes.

Esta forma de operar tiene dos consecuencias. El sistema es multitarea, puesto que el número de aplicaciones no está limitado al compartir todas la misma cola de mensajes, y que las aplicaciones pueden comunicarse entre ellas mandando y recibiendo mensajes a través de la cola. Gracias a ello, se puede programar una aplicación que colabora con otras. Una posibilidad sería, por ejemplo, combinar un programa de lectura de datos con otro que procesase datos.

Debe notarse que existe otra consecuencia de usar una sola cola de mensajes, y es que la unidad central de proceso (CPU : Central Process Unit) alternará de una aplicación a otra en muy cortos períodos de tiempo, según le lleguen los mensajes y tenga que responder a ellos. Si un programa ocupase todo el tiempo de la CPU se produciría un atasco en la cola de mensajes al no poder procesarse, y probablemente el sistema caería.

La característica más importante y que más diferencia a Windows de MS-DOS es la independencia del hardware. Windows se interpone siempre entre la aplicación y el hardware, ninguna aplicación puede acceder directamente al hardware evitando a Windows. Para interactuar con los dispositivos, ya sean impresoras, pantallas, etc.. tendrá que enviar un mensaje a Windows y será Windows quien acceda y después mande la respuesta a la aplicación.

Para acceder a los dispositivos Windows utiliza rutinas del SDK que garantizan la seguridad de la operación y el mantenimiento de la multitarea. Todo fabricante de hardware tiene que desarrollar sus productos siguiendo los estándares definidos por Microsoft si quiere que sus productos funcionen bajo Windows.

Esta característica ha de tenerse muy en cuenta, pues para programar la tarjeta de entrada / salida existente en el sistema levitador neumático no basta tener los

controladores (drivers) como en MS-DOS sino que además se debe tener un conjunto de bibliotecas de enlace dinámico (DLL) que contengan las funciones para acceder a la tarjeta.

La ventaja que presenta Windows gracias a esto es que al no acceder directamente a los dispositivos, el programa es independiente de las características particulares del dispositivo. Por ejemplo, se pinta en pantalla a o se imprime a través de un interfaz de dispositivo gráfico (GDI : Graphics Device Interface) que facilita la presentación y la independiza del hardware, es decir, bajo Windows todos los monitores o todas las impresoras son iguales.

Por último, hay que comentar el concepto de manejador o “handle”. Un manejador es un número entero y único que usa Windows para identificar un objeto determinado, ya sea una ventana, un contexto de dispositivo (usado para pintar) o un fichero abierto. El concepto de handle es parecido al de puntero en C, pero más general pues no se refiere sólo a direcciones de memoria, sino a todos los objetos que existen en cada momento en el sistema.

Todas estas características de Windows hacen que desarrollar una aplicación bajo Windows sea un proceso completamente diferente a lo que era la programación bajo MS-DOS.

4.3 VISUAL C++

4.3.1 ¿ Qué es Visual C++ ?

Visual C++ es un entorno para el desarrollo de aplicaciones escritas en C++ para el sistema operativo Windows. En concreto, para el programa que se desarrolló para este proyecto se usó el **Microsoft Developer Studio 97**, que contiene **Microsoft Visual C++ 5.0**.

Visual C++ es un entorno de programación que combina las características de orientación a objetos del lenguaje de programación **C++** y el sistema de desarrollo para crear aplicaciones gráficas para Windows **SDK**.

Mediante Visual C++ es posible escribir programas para Windows que se aprovechen de la potencia y la capacidad que ofrece el lenguaje C++ a la vez que se facilita la programación bajo Windows gracias a las MFC.

Visual C++ se permite desarrollar aplicaciones para Windows 3.x (16 bits) o para Windows 95 o NT (32 bits), incluyendo editores y asistentes que facilitan la tarea de desarrollo del software. Permite insertar objetos COM, múltiples ventanas de documentos (MDI), controles ActiveX, interacciones con Internet, etc…

Dado que enumerar siquiera todas las características y capacidades de Visual C++ sería excesivamente largo, y se aparta del objeto de esta memoria, sólo se hará referencia a las características usadas en el programa desarrollado durante el proyecto:

- Facilita la programación multihilo (multithreading), que permite separar en el programa el temporizador y el muestreo de las señales de la presentación en pantalla de los resultados.

- Permite la utilización de bibliotecas de enlace dinámico (DLL), con lo cual se pueden usar las DLL que permiten programar y operar la tarjeta de entrada / salida.

- Permite la utilización de funciones fuera de clase (funciones call-back) que permiten garantizar la exactitud de las lecturas en cada tiempo de muestreo. Más adelante se comenta qué son y por qué se usan la funciones fuera de clase.

A continuación se añade una breve introducción a la programación en Visual C++ con objeto de poder explicar cómo está hecha la aplicación del proyecto.

4.3.2 Programación en Visual C++

Al escribir una aplicación en Visual C++ se facilita enormemente el trabajo dado que Visual C++ proporciona varias herramientas para hacer más sencillo el proceso.

El primer elemento son las **MFC** (Microsoft Foundation Classes : Clases Base de Microsoft). Las MFC es una biblioteca de clases que define una cantidad inmensa de clases con sus procedimientos. Las MFC se encargan de dar soporte a los objetos usados en Windows como ventanas, menús, controles, botones, mapas de bits, etc…

Cuando se realiza una aplicación las clases que se utilizan son casi siempre clases MFC o derivadas directa o indirectamente de éstas.

Antes de las MFC, se desarrollaba una aplicación usando el lenguaje C y la biblioteca de funciones de la API (Application Program Interface) de Windows. Las funciones API se encargaban de la relación entre los objetos Windows y la aplicación en si. Con la MFC se consigue un interfaz sencillo y potente entre Windows y la aplicación.

Se puede decir que las MFC "encapsulan" las funciones de la API para facilitar su uso. No obstante, las MFC no sustituyen a las funciones API.

Mediante las MFC se pueden desarrollar gran cantidad de aplicaciones, sin prácticamente escribir ni una sola función nueva, sino sólo usando las funciones de dichas clases.

Se puede decir que aprender a utilizar las MFC es la parte más difícil de aprender a programar en Visual C++ debido a la cantidad inmensa de clases que incluye. No obstante, para facilitar su uso, Visual C++ incluye asistentes como ClassWizard.

Los asistentes para la programación son varios:

- **AppWizard** : Se puede escoger al crear un nuevo proyecto (menú File, opción New) y crea el esqueleto de una aplicación usando la biblioteca MFC. Genera los ficheros necesarios para tener inmediatamente una aplicación Windows ejecutable, pero sin ninguna funcionalidad.

- **Editores de Recursos** : Se pueden escoger desde la vista Resource View de la ventana Workspace. Permiten diseñar la barra de menú, el icono que representa la aplicación, el aspecto de los cuadros de diálogo que mostrará la aplicación, etc…

- **ClassWizard** : Se puede llamar pulsando Ctrl+W. Permite generar automáticamente las clases y funciones que gestionarán los mensajes que la aplicación reciba o mande a los objetos Windows.

- **Depurador** : Facilita la depuración del programa.

El modo más sencillo de programar en Visual C++ es generar una aplicación nueva mediante el asistente AppWizard. En el siguiente apartado se explica cómo generar con AppWizard una nueva aplicación. Una vez generada la aplicación, hay que dotarla de contenido. Para ello se definen los menús y los controles, y, en general, el aspecto de la aplicación, usando los editores de recursos, y, en paralelo, se va realizando la programación en C++ generando las clases y funciones necesarias, ya sea mediante el asistente ClassWizard o directamente, y conectándoles a los controles, menús, etc que se han definido con el editor de recursos.

Naturalmente, generar una aplicación compleja requiere un conocimiento más profundo de Visual C++ y, en especial, de las MFC, cuya explicación se escapa del objeto de esta memoria.

Por esta razón, cuando se presente el código de la aplicación en esta memoria se comentará sólo de manera sencilla la funcionalidad o las opciones de las clases utilizadas, y sólo en algunos casos se podrá explicar con detalle el código del programa. En caso de dudas sobre las funciones de Visual C++ usadas en la aplicación habrá que remitirse a la ayuda en línea que proporciona el propio entorno de desarrollo de Visual C++.

4.3.3 Creación de una aplicación en Visual C++

Visual C++ permite crear aplicaciones de diferentes tipos, hasta programas que no son aplicaciones, como bibliotecas de enlace dinámico (DLL), o controles ActiveX. Incluso dentro de lo que son aplicaciones permite crear aplicaciones de 16 ó de 32 bits.

La aplicación que se ha creado para este proyecto es una aplicación de 32 bits. La manera de crear una igual es la siguiente:

Una vez que se ha entrado en **Microsoft Visual C++ 5.0** hay que elegir la opción **New** del menú **File**. Hecho esto aparece una ventana con varias pestañas. Se debe elegir la pestaña **Projects** y dentro de las opciones que aparecen hay que escoger la opción **MFC AppWizard (exe)**. Con esta opción se indica que se desea crear una nueva aplicación ejecutable usando el asistente de aplicaciones. Para poder seguir hay que darle un nombre a la aplicación rellenando el campo **Project name** de la ventana.

A continuación comienza la primera de una serie de seis ventanas en las que Visual C++ va preguntando características de la aplicación que va a generar.

En la primera ventana pregunta si la aplicación será SDI, MDI o 'dialog based'. Una aplicación SDI (Single Document Interface) permite al usuario trabajar con sólo un documento a la vez en la aplicación. Una MDI (Multiple Document Interface) permite al usuario más de un documento de trabajo, pero es más complicada. Por último, una aplicación 'dialog based', está basada en cuadros de diálogos y no en ventanas. Se debe escoger como opción **SDI**.

La segunda ventana pregunta si se usará algún tipo de base de datos. Se debe elegir que no, que es la opción por defecto. En caso de duda, es recomendable siempre elegir la opción por defecto, que es la más común.

La tercera ventana pregunta por características del documento. También se debe elegir la opción por defecto, pulsando simplemente el botón de **Aceptar**.

La cuarta ventana se refiere a los añadidos de la ventana, como la barra de herramientas, o la barra de estado. Conviene dejarlas seleccionadas, por si en el futuro hacen falta. Por tanto, se deben elegir las opciones por defecto de esta ventana.

La quinta ventana pregunta sobre cómo trabajar con las MFC y sobre si el asistente de aplicaciones debe añadir comentarios al código que él mismo genera. Aquí también se escogen las opciones por defecto.

La sexta ventana es la que muestra las clases que va crear para la nueva aplicación y los ficheros que le corresponden, así como las clases respectivas de MFC de las que derivan. Estas clases las genera el asistente automáticamente cuando crea la nueva aplicación. Su nombre depende del nombre que se haya elegido para la aplicación. Suponiendo que el nombre elegido para la aplicación fuese **Prueba**, las clases generadas son :

- **CPruebaApp** : Esta clase es la primera que se construye cuando de ejecuta la aplicación, y se encarga de construir las demás y de gestionar los mensajes de la aplicación.

- **CmainFrame** : Esta clase se encarga de gestionar el marco de la ventana de la aplicación. En caso de una aplicación MDI existe más de un marco, por lo cual se divide en dos clases CMainFrame y CChildFrame.

- **CPruebaDoc** : Esta clase se encarga de trabajar con el documento. En la aplicación del levitador no se usa ningún documento, con lo que no se modifica esta clase para nada.

- **CPruebaView** : Esta clase se encarga de la vista del documento. Siguiendo la arquitectura documento-vista pueden existir todas las clases vista que se quiera, pero al menos debe existir una. En la aplicación del levitador sólo existe una vista.

Cada una de estas clases se define e implementa en dos ficheros diferentes para cada clase.

Un fichero con extensión **.h** (header = cabecera) donde se define la clase correspondiente. El nombre del fichero suele estar relacionado con el nombre de la clase, por ejemplo, la clase CPruebaView se define en el fichero **pruebaview.h**

En el fichero de cabecera está definida la clase a la manera de las clases en C++.

Además de la definición de la clase, el fichero de cabecera contiene la declaración del mapa de mensajes, que es la relación entre los mensajes (Ya se ha comentado que Windows funciona mediante mensajes) que le llegan a la clase y las funciones propias de dicha clase que deben ejecutarse en función de cada mensaje.

Un fichero con extensión **.cpp** (de C++ = *c plus plus*) donde se implementa la clase correspondiente. El nombre suele ser el mismo que el del fichero de cabecera con el único cambio de la extensión .cpp en vez de .h.

En el fichero .cpp se encuentra la implementación de la clase y la implementación del mapa de mensajes.

4.4 CÓDIGO FUENTE DE LA APLICACIÓN

En este apartado se incluye el código fuente de los ficheros que componen que la aplicación y que son imprimibles. Existen otros ficheros, como, por ejemplo, los de iconos, que no son imprimibles. También se han añadido imágenes con el aspecto que presentan los cuadros de diálogo y la ventana principal de la aplicación.

Para facilitar la comprensión del código se ha intercalado un gran número de comentarios dentro del código escrito.

Dado que Visual C++ genera automáticamente una parte sustancial del código, se ha optado por no alterar dichas partes. En los ficheros que no han sido alterados desde su creación automática por el asistente de Visual C++ se ha añadido un comentario al principio para indicar que no deben ser modificados.

Con el fin de reducir el número de ficheros modificados se ha procurado concentrar toda la funcionalidad del programa en la clase vista, por eso sus ficheros .cpp y .h son los más extensos con diferencia.

Los ficheros que componen el proyecto en Visual C++ del que se genera la aplicación del levitador neumático son los siguientes, (en negrita los que se han modificado) :

- MainFrm.cpp : contiene la implementación de la clase marco de la aplicación llamada DMainFrame. Deriva de la clase CFrameWnd perteneciente a las MFC.

- mypint.cpp : contiene la implementación de la clase que construye la aplicación DMypintApp, que deriva de la clase CWinApp perteneciente a las MFC.

- mypintDoc.cpp : contiene la implementación de la clase DMypintDoc, que deriva de la clase CDocument de las MFC.

- **mypintView.cpp** : contiene la implementación de la clase DMypintView que deriva de la clase CView de las MFC. Es en esta clase donde se han programado todas las funciones del programa.

- StdAfx.cpp : contiene la implementación de la biblioteca estándar de llamadas afx.

- **mypint.rc** : contiene la descripción de los recursos de los que hará uso la aplicación. Este fichero no se modifica directamente, sino que a través de los editores de recursos se modifican los recursos como menús, cuadros de diálogo, etc y es el asistente el que modifica automáticamente este fichero.

- MainFrm.h : contiene la declaración de la clase marco DMainFrame.

- **mypint.h** : contiene la declaración de la clase DMypintApp. Dado que este fichero de cabecera se incluye en todos los demás, se han hecho aquí algunas definiciones que eran necesarias para las demás clases.

- mypintDoc.h : contiene la declaración de la clase documento DMypintDoc.

- **mypintView.h** : contiene la declaración de la clase vista, DMypintView. Por tanto contiene todas las definiciones necesarias para la aplicación excepto las que se han hecho dentro de mypint.h.

- StdAfx.h : contiene las declaraciones estándares de una aplicación en Visual C++.

- **Resource.h** : contiene los identificadores de los elementos que aparecen en los recursos. Al igual que mypint.rc este fichero se modifica automáticamente para adecuarse a la definición de los recursos que se hace a través de los editores de recursos.

- **mypint.ico** : contiene el icono principal de la aplicación en formato. Es un fichero ".ico" como otro cualquiera.

- mypint.rc2 : contiene los recursos que no son editables usando los editores de recursos de Visual C++. En esta aplicación no existe ningún recurso externo, con lo cual este fichero no se ha modificado.

- mypintDoc.ico : contiene el icono que representa a los documentos usados por la aplicación. No tiene uso en la aplicación porque esta aplicación no usa la arquitectura documento-vista, con el fin de simplificar la aplicación.

- ReadMe.txt : fichero de texto generado por Visual C++ automáticamente que contiene información sobre el contenido de los demás ficheros de la aplicación.

- mypint.clw : fichero que contiene la información que necesita el entorno de Visual C++ para cargar los ficheros anteriores y poder editarlos en el entorno de programación de Visual C++. Contiene información sobre los ficheros que pertenecen proyecto **mypint**.

- mypint.dsp : fichero para construir el proyecto mypint. Automatiza el proceso de construcción de la aplicación, es decir, del fichero ejecutable **mypint.exe**.

Como se puede ver, el número de ficheros que han sido generados automáticamente por Visual C++ es bastante superior al de los ficheros que han sido creados o modificados durante la programación de la aplicación.

Hay que tener en cuenta, que, si bien son un menor número, los ficheros referentes a la clase vista son mucho más largos porque concentran casi toda la funcionalidad del programa. Esta concentración de las funciones que se usan en una sola clase se ha hecho para reducir el número de ficheros a modificar cuando se quiera alterar el programa.

También se ha renunciado al modelo documento-vista en busca de mayor sencillez en la representación de los datos y a la hora de guardar dichos datos en un fichero.

A continuación se incluye el contenido de los ficheros de la aplicación precedidos de su nombre:

MainFrm.h

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO


// MainFrm.cpp : Implementacion de la clase DMainFrame
//

#include "stdafx.h"
#include "mypint.h"

#include "MainFrm.h"
#include "mypintDoc.h"
#include "mypintView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
///////
```

```
// DMainFrame

IMPLEMENT_DYNCREATE(DMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(DMainFrame, CFrameWnd)
	//{{AFX_MSG_MAP(DMainFrame)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
///////
// DMainFrame construction/destruction

DMainFrame::DMainFrame()
{
	// TODO: add member initialization code here

}

DMainFrame::~DMainFrame()
{
}

BOOL DMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs

	return CFrameWnd::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////
///////
// DMainFrame diagnostics

#ifdef _DEBUG
void DMainFrame::AssertValid() const
{
	CFrameWnd::AssertValid();
}

void DMainFrame::Dump(CDumpContext& dc) const
{
	CFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////
///////
// DMainFrame message handlers
```

mypint.cpp

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO
```

```
// mypint.cpp : Implementacion del constructor de la aplicacion
//

#include "stdafx.h"

#include "mypint.h"

#include "MainFrm.h"
#include "mypintDoc.h"
#include "mypintView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
///////
// DMypintApp

BEGIN_MESSAGE_MAP(DMypintApp, CWinApp)
      //{{AFX_MSG_MAP(DMypintApp)
      ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// NOTE - the ClassWizard will add and remove mapping macros here.
//    DO NOT EDIT what you see in these blocks of generated code!
      //}}AFX_MSG_MAP
      // Standard file based document commands
      ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
      ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
///////
// DMypintApp construction

DMypintApp::DMypintApp()
{
      // TODO: add construction code here,
      // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////
///////
// The one and only DMypintApp object

DMypintApp theApp;

/////////////////////////////////////////////////////////////////////////
///////
// DMypintApp initialization

BOOL DMypintApp::InitInstance()
{
      AfxEnableControlContainer();
```

```
// Standard initialization
// If you are not using these features and wish to reduce the size
//  of your final executable, you should remove from the following
//  the specific initialization routines you do not need.

#ifdef _AFXDLL
      Enable3dControls();                // Call this when using MFC
in a shared DLL
#else
      Enable3dControlsStatic();    // Call this when linking to MFC
statically
#endif

      // Change the registry key under which our settings are stored.
      // You should modify this string to be something appropriate
      // such as the name of your company or organization.
      SetRegistryKey(_T("Local AppWizard-Generated Applications"));

      LoadStdProfileSettings();   // Load standard INI file options
(including MRU)

// Register the application's document templates.  Document templates
//   serve as the connection between documents, frame windows and
views.

      CSingleDocTemplate* pDocTemplate;
      pDocTemplate = new CSingleDocTemplate(
            IDR_MAINFRAME,
            RUNTIME_CLASS(DMypintDoc),
            RUNTIME_CLASS(DMainFrame),       // main SDI frame window
            RUNTIME_CLASS(DMypintView));
      AddDocTemplate(pDocTemplate);

// Parse command line for standard shell commands, DDE, file open
      CCommandLineInfo cmdInfo;
      ParseCommandLine(cmdInfo);

      // Dispatch commands specified on the command line
      if (!ProcessShellCommand(cmdInfo))
            return FALSE;

// The one and only window has been initialized, so show and update
it.
      m_pMainWnd->ShowWindow(SW_SHOW);
      m_pMainWnd->UpdateWindow();

      return TRUE;
}

////////////////////////////////////////////////////////////////////
///////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
      CAboutDlg();
```

```
// Dialog Data
	//{{AFX_DATA(CAboutDlg)
	enum { IDD = IDD_ABOUTBOX };
	//}}AFX_DATA

	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CAboutDlg)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support
	//}}AFX_VIRTUAL

// Implementation
protected:
	//{{AFX_MSG(CAboutDlg)
		// No message handlers
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
	//{{AFX_DATA_INIT(CAboutDlg)
	//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(CAboutDlg)
	//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
	//{{AFX_MSG_MAP(CAboutDlg)
		// No message handlers
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void DMypintApp::OnAppAbout()
{
	CAboutDlg aboutDlg;
	aboutDlg.DoModal();
}

/////////////////////////////////////////////////////////////////////////////
///////
// DMypintApp commands
```

mypintDoc.cpp

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO

// mypintDoc.cpp : Implementacion de la clase DMypintDoc
//

#include "stdafx.h"
#include "mypint.h"

#include "mypintDoc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
///////
// DMypintDoc

IMPLEMENT_DYNCREATE(DMypintDoc, CDocument)

BEGIN_MESSAGE_MAP(DMypintDoc, CDocument)
      //{{AFX_MSG_MAP(DMypintDoc)
// NOTE - the ClassWizard will add and remove mapping macros here.
//    DO NOT EDIT what you see in these blocks of generated code!
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
///////
// DMypintDoc construction/destruction

DMypintDoc::DMypintDoc()
{
      // TODO: add one-time construction code here

}

DMypintDoc::~DMypintDoc()
{
}

BOOL DMypintDoc::OnNewDocument()
{
      if (!CDocument::OnNewDocument())
            return FALSE;

      // TODO: add reinitialization code here
      // (SDI documents will reuse this document)

      return TRUE;
}
```

```
/////////////////////////////////////////////////////////////////////////
///////
// DMypintDoc serialization

void DMypintDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
		// TODO: add storing code here
	}
	else
	{
		// TODO: add loading code here
	}
}

/////////////////////////////////////////////////////////////////////////
///////
// DMypintDoc diagnostics

#ifdef _DEBUG
void DMypintDoc::AssertValid() const
{
	CDocument::AssertValid();
}

void DMypintDoc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////
///////
// DMypintDoc commands
```

mypintView.cpp

```
// mypintView.cpp : Implementacion de la clase DMypintView


// Este fichero contiene todas las funciones de la aplicacion que
puede
// ejecutar el usuario a traves de los menus  y los controles. Se han
// incluido todas las funciones en la vista para que solo se tenga que
// modificar un fichero cuando se altera el programa

#include "stdafx.h"
#include "mypint.h"
#include "MainFrm.h"

#include "mypintDoc.h"
#include "mypintView.h"
```

```
// Incluir la biblioteca mmsystem.h para el temporizador multimedia
#include <mmsystem.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////
///////
// DMypintView

IMPLEMENT_DYNCREATE(DMypintView, CView)

// Mapa de mensajes de la clase vista DMypintView
BEGIN_MESSAGE_MAP(DMypintView, CView)
      //{{AFX_MSG_MAP(DMypintView)
      ON_COMMAND(ID_CHEQUEO, OnChequeo)
      ON_COMMAND(ID_APP_EXIT, OnCerrar)
      ON_COMMAND(ID_ARCHIVO_NUEVO, OnNuevoExperimento)
      ON_MESSAGE(WM_MY_TIMER, OnMyTimer)
      ON_MESSAGE(WM_MY_GUARDAR, OnGuardar)
      ON_COMMAND(ID_PID, OnChequeoPid)
      ON_COMMAND(ID_REPINTAR_TODO, OnRepintarTodo)
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
///////
// DMypintView construction/destruction

// Constructor de la clase DMypintView
DMypintView::DMypintView()
{
      // Valores por defecto del experimento

      // Tiempo final en segundos
      m_fmaxtime = 250.f;
      // Tiempo de muestreo en segundos
      m_fsampletime = 0.852f;
      // Constantes del PID. Tiempos en segundos.
      m_fKp = 1.f;
      m_fTd = 0.7f;
      m_fTi = 3.f;
      m_fsum = 0.f;
      // Modo de control por defecto es un PID
      m_umode = 1;
      m_bNoFiltrar = FALSE;
}

// Destructor virtual de la clase vista de la aplicacion
DMypintView::~DMypintView()
{
}

// Creacion de la ventana de la vista
```

```
BOOL DMypintView::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs

	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////
///////
// DMypintView drawing

// OnDraw : Funcion para dibujar la ventana de la vista cada vez que
se
// cambia, se maximiza o se restaura el tamaño de la ventana. Se ha
// redefinido OnDraw para que pinte la gráfica de las señales

void DMypintView::OnDraw(CDC* pDC)
{
	DMypintDoc* pDoc = GetDocument();
	ASSERT_VALID(pDoc);

	// La funcion de dibujo que sobrecarga a OnDraw.

	// Obtener el contexto de dispositivo
	CClientDC dc(this);

	// Obtener las medidas del área de dibujo
	CRect rect; // Definir estructura de dimensiones
	GetClientRect( rect ); // Obtener las dimensiones
	int cx = ( rect.right - rect.left );
	int cy = (rect.bottom - rect.top );

	// Calcular los tamaños y la escala a usar
	dc.SetBkColor( BK_COLOR );
	dc.SetMapMode(MM_ANISOTROPIC); // Fijar modo de proyeccion
	int tam_x = (int) (RES_X*m_fmaxtime); // Tamaño del eje X
	int tam_y = (int) (RES_Y*(VAL_MAXIMO - VAL_MINIMO)); // Tamaño
del Y
	int total_x = (int) (1.2*(RES_X*m_fmaxtime)); // Tamaño más
margen
	int total_y = (int) (1.2*(RES_Y*(VAL_MAXIMO - VAL_MINIMO))); //
Idem
	int mov_x = total_x - tam_x;  // Origen de los ejes
	int mov_y = total_y - tam_y;  // Origen de los ejes
	// Numero de unidades logicas de la pantalla...
	int pant_x = (int) (total_x + 0.1*tam_x); // en el eje X...
	int pant_y = (int) (total_y + 0.1*tam_y); // y en el eje Y
	int alt_fila1 = (int) (0.8 * mov_y);  // Altura de los numeros
de tiempo
	int anch_col1 = (int) (0.8 * mov_x); // Margen de los numeros de
tiempo
	int anch_col2 = (int) (0.4 * mov_x); // Margen de los numeros de
voltaje
	int alt_fila2 = (int) (1.1 * mov_y); // Altura de los numeros de
voltaje
	int alt_fila3 = (int) ( total_y+0.07*tam_y ); // Altura de la
leyenda de las señales
```

```
	// Variables de ayuda
	int x;
	CString cadenita;
	// Colores de las señales que se van a pintar
	CPen Pluma_r( PS_SOLID, 2, COLOR_R);
	CPen Pluma_yf( PS_SOLID, 2, COLOR_YF);
	CPen Pluma_u( PS_SOLID, 2, COLOR_U);
	CPen Pluma_e( PS_SOLID, 2, COLOR_E);

	// Fijar los tamaños de la ventana
	dc.SetWindowExt( pant_x, pant_y );
	dc.SetViewportExt( cx, -cy );
	dc.SetViewportOrg( 0, cy );

	// Definir el color de los ejes
	CPen MiPluma( PS_SOLID, 2, COLOR_EJES );
	CPen *pPlumaAnterior = dc.SelectObject( &MiPluma );

	// Pintar ejes verticales
	for ( x = 0; x <=tam_x; x+=tam_x/N_EJES)
	{
		dc.MoveTo(x+mov_x,mov_y);
		dc.LineTo(x+mov_x,tam_y+mov_y);
	}
	// Pintar ejes horizontales
	for ( x = 0; x <=tam_y; x+=tam_y/N_EJES)
	{
		dc.MoveTo(mov_x,x+mov_y);
		dc.LineTo(tam_x+mov_x,x+mov_y);
	}

	// Pintar escala de tiempos
	float i=0.f;
	dc.SetTextColor( COLOR_LEYENDA );
	for ( x = 0; x <=tam_x; x+=tam_x/N_EJES)
	{
		float truco = i*m_fmaxtime/N_EJES;
		cadenita.Format("%.1f",truco);
		dc.TextOut(x+anch_col1,alt_fila1,cadenita);
		i++;
	}

	// Pintar escala de voltajes
	i=0;
	for ( x = 0; x <=tam_y; x+=tam_y/N_EJES)
	{
		float truco = i*(VAL_MAXIMO - VAL_MINIMO)/N_EJES;
		cadenita.Format("%.1f",truco);
		cadenita+=" V";
		dc.TextOut(anch_col2,x+alt_fila2,cadenita);
		i++;
	}

	// Pintar texto de leyenda
	dc.TextOut((int)(2.5*mov_x),(int)(0.5*mov_y),"Tiempo en segundos
");
	dc.SetTextColor( COLOR_U );
	dc.TextOut((int)(1.5*mov_x), alt_fila3, "señal u" );
```

```
	dc.SetTextColor( COLOR_YF );
	dc.TextOut((int)(2.5*mov_x), alt_fila3, "señal y" );
	dc.SetTextColor( COLOR_R );
	dc.TextOut((int)(3.5*mov_x), alt_fila3, "señal r" );
	dc.SetTextColor( COLOR_E);
	dc.TextOut((int)(4.5*mov_x), alt_fila3, "señal e" );


	// Bucles para pintar las señales como lineas continuas

	// Bucle para la señal de referencia 'r'
	int pos_x1 = mov_x;
	dc.SelectObject( &Pluma_r );
	for ( i=1;i<m_uIndLec;i++)
	{
		dc.MoveTo(pos_x1,(int)   (mov_y+m_signal_rvo.GetAt(i-1)   *
RES_Y ) );
		int pos_x0 = (int) (mov_x+m_fsampletime*i*RES_X);
		dc.LineTo(pos_x0,(int)  (mov_y  +  m_signal_rvo.GetAt(i)  *
RES_Y ) );
		pos_x1 = pos_x0;
	}

	// Bucle para la señal de control 'u'
	pos_x1 = mov_x;
	dc.SelectObject( &Pluma_u );
	for ( i=1;i<m_uIndLec;i++)
	{
		dc.MoveTo(pos_x1,(int) (mov_y+m_signal_u.GetAt(i-1)*  RES_Y
) );
		int pos_x0 = (int) (mov_x+m_fsampletime*i*RES_X);
		dc.LineTo(pos_x0,(int) (mov_y+m_signal_u.GetAt(i)*RES_Y));
		pos_x1 = pos_x0;
	}

	// Bucle para la señal de error 'e' en valor absoluto
	pos_x1 = mov_x;
	dc.SelectObject( &Pluma_e );
	for ( i=1;i<m_uIndLec;i++)
	{
		dc.MoveTo(pos_x1,(int)                       (mov_y+(float)
fabs(m_signal_e.GetAt(i-1)) *RES_Y));
		int pos_x0 = (int) (mov_x+m_fsampletime*i*RES_X);
		dc.LineTo(pos_x0,(int)           (mov_y+           (float)
fabs(m_signal_e.GetAt(i)) * RES_Y));
		pos_x1 = pos_x0;
	}

	// Bucle para la señal de salida filtrada 'yf'
	pos_x1 = mov_x;
	dc.SelectObject( &Pluma_yf );
	for ( i=1;i<m_uIndLec;i++)
	{
		dc.MoveTo(pos_x1,(int) (mov_y+m_signal_y.GetAt(i-1)*  RES_Y
) );
		int pos_x0 = (int) (mov_x+m_fsampletime*i*RES_X);
		dc.LineTo(pos_x0,(int) (mov_y+m_signal_y.GetAt(i)*RES_Y));
		pos_x1 = pos_x0;
```

```
	}

	// Devolver la pluma
	dc.SelectObject( pPlumaAnterior );
}

/////////////////////////////////////////////////////////////////////////
///////
// DMypintView diagnostics

// Funciones de la clase vista para su propio uso
#ifdef _DEBUG
void DMypintView::AssertValid() const
{
	CView::AssertValid();
}

void DMypintView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}

DMypintDoc* DMypintView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(DMypintDoc)));
	return (DMypintDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////
///////
// DMypintView message handlers


// OnInitialUpdate : Funcion que se ejecuta cuando se inicia la
aplicacion
// Se encarga de inicializar la tarjeta, comprobando si hubo fallos, e
// inicia una serie de dialogos para pedir al usuario los datos que
// necesita para poder comenzar un experimento
void DMypintView::OnInitialUpdate()
{
	// Llama primero a la funcion del mismo nombre de su clase base
	// Esta llamada es necesaria para inicializar bien la clase
vista
	// de la aplicacion, que deriva de una clase base CView
	CView::OnInitialUpdate();

	// Abrir tarjeta y preparar subsistemas comprobando si existen
fallos
	ECODE fallo; // Variable de fallo
	CString my_tarjeta = TARJETA;

	// Inicializacion de un subsistema ( el que se usara de A/D )
	// en la tarjeta
	fallo=olDaInitialize((LPSTR)	(LPCSTR)	my_tarjeta,(LPHDEV)
&m_board );
	// Comprobacion de si fallo la inicializacion
	if(fallo)
```

```
      {     AfxMessageBox ( "Fallo al intentar abrir la tarjeta" );
return ;    }

      // Fijar flujo del subsistema como entrada de datos a la tarjeta
inicializada
      fallo=olDaGetDASS(m_board.hdrvr,OLSS_AD,0,&m_board.hdass);
      // Comprobacion de si fallo la orden
      if(fallo)
      {     AfxMessageBox ( "Fallo al intentar leer las capacidades de
la tarjeta" );return;   }

      // Establecer el modo de funcionamiento como 'single-value'
      fallo=olDaSetDataFlow(m_board.hdass,OL_DF_SINGLEVALUE );
      // Comprobacion de posible fallo en la operacion
      if(fallo)
      {     AfxMessageBox ( "Fallo al establecer el subsistema A/D"
);return;   }

      // Configurar el subsistema A/D
      fallo=olDaConfig(m_board.hdass);
      // Comprobacion de posible fallo en la operacion
      if(fallo)
      {     AfxMessageBox ( "Fallo al configurar el subsistema A/D"
);return;   }

      // Inicializacion de un subsistema ( el de D/A ) en la tarjeta
      fallo=olDaInitialize((LPSTR)     (LPCSTR)     my_tarjeta,(LPHDEV)
&m_board2);
      // Comprobacion de posible fallo en la operacion
      if(fallo)
      {     AfxMessageBox ( "Fallo al intentar abrir la tarjeta" );
return;     }

      // Fijar flujo del subsistema como salida de datos de la tarjeta
      fallo=olDaGetDASS(m_board2.hdrvr,OLSS_DA,0,&m_board2.hdass);
      // Comprobacion de posible fallo en la operacion
      if(fallo)
      {     AfxMessageBox ( "Fallo al intentar leer las capacidades la
tarjeta" );return;      }

      // Fijar modo de operacion como 'single-value'
      fallo=olDaSetDataFlow(m_board2.hdass,OL_DF_SINGLEVALUE );
      // Comprobacion de posible fallo en la operacion
      if(fallo)
      {     AfxMessageBox ( "Fallo al establecer el subsistema D/A"
);return;   }

      // Configurar el subsistema D/A
      fallo=olDaConfig(m_board2.hdass);
      // Comprobacion de posible fallo en la operacion
      if(fallo)
      {     AfxMessageBox ( "Fallo al configurar el subsistema D/A"
);return;   }

      // Fijar el rango del subsistema A/D
      fallo=olDaGetRange(m_board.hdass,&m_admax,&m_admin);
      // Comprobacion de posible fallo en la operacion
      if(fallo)
```

```
      {     AfxMessageBox ( "Fallo al intentar leer rango para la señal
de entrada" );return;   }

      // Fijar el rango del subsistema D/A
      fallo=olDaGetRange(m_board2.hdass,&m_damax,&m_damin);
      // Comprobacion de posible fallo en la operacion
      if(fallo)
      {     AfxMessageBox ( "Fallo al intentar leer rango para la señal
de salida" );return;    }


      // Inicializacion de algunas variables miembro
      m_uIndLec = 0; // Indice para muestra presentadas en pantalla
      m_uIndEsc = 0; // Indice para muestra leidas con la tarjeta

      // Dialogo para fijar el tiempo de muestreo y el tiempo final
      DLimites dlg;
      do
      {
            dlg.m_ftiempomuest=m_fsampletime;
            dlg.m_ftiempomax=m_fmaxtime;
            dlg.m_umodo = m_umode;
      }
      while ( dlg.DoModal() == IDCANCEL );

      // Actualizacion de los datos introducidos por el usuario
      m_fmaxtime=dlg.m_ftiempomax;
      m_fsampletime=dlg.m_ftiempomuest;
      m_umode = dlg.m_umodo;

      // Dialogo para fijar el nombre del fichero donde guardar los
      // resultados del experimento
      CString nom_fichero; // variable auxiliar
      int var; // variable auxiliar
      DDialogoSalvar dlgsave( // llamo al constructor del dialogo
            FALSE,
            NULL,
            NULL,
            OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
            "Ficheros MATLAB (*.m)|*.m||",
            NULL);
      // Aparicion del dialogo en pantalla
      do
      {
            var = dlgsave.DoModal();
      }
      while ( var == IDCANCEL );

      // Asignar nombre al fichero
      nom_fichero = dlgsave.GetPathName();
      // incluyendo extension ".m" si no se ha puesto
      if( dlgsave.GetFileExt().IsEmpty() )
      {     nom_fichero += ".m";    }

      // Abrir fichero y comprobar si hay errores
      if  (  !n_myfichero.Open  (  nom_fichero,  CFile::modeCreate  |
CFile::modeWrite ) )
```

```
	{	AfxMessageBox ( "No se puede abrir el fichero" );return;
	}

	// Ajustar tamaño de los CArray para el numero de muestras
adecuado
	m_uNumMuest = (UINT) (m_fmaxtime / m_fsampletime)+1;
	m_signal_u.SetSize(m_uNumMuest+1000);
	m_signal_e.SetSize(m_uNumMuest+1000);
	m_signal_y.SetSize(m_uNumMuest+1000);
	m_signal_yf.SetSize(m_uNumMuest+1000);
	m_signal_r.SetSize(m_uNumMuest+1000);
	m_signal_rvo.SetSize(m_uNumMuest+1000);


	// Pedir los valores de las constantes del PID : Kp, Td y Ti.
	// solo si se ha elegido modo PID
	if ( m_umode == 1 )
	{
		DDialgoPID Dlg_Pid;
		do
		{
			Dlg_Pid.m_fKp_Dlg = m_fKp;
			Dlg_Pid.m_fTd_Dlg = m_fTd;
			Dlg_Pid.m_fTi_Dlg = m_fTi;
		}
		while ( Dlg_Pid.DoModal() == IDCANCEL );

		// Actualizar los valores del PID a los datos introducidos
		// por el usuario
		m_fKp = Dlg_Pid.m_fKp_Dlg;
		m_fTd = Dlg_Pid.m_fTd_Dlg;
		m_fTi = Dlg_Pid.m_fTi_Dlg;
	}

	// Generar la señal de referencia 'r'
	GenerarReferencia();

	// Fijar el tiempo de muestreo en milisegundos
	UINT tiempo = (UINT) (1000*m_fsampletime);
	// Crear el timer. En este momento comienza el experimento
	m_uIDTimer                                                    =
timeSetEvent(tiempo,50,(LPTIMECALLBACK)NuestroProc,0L,TIME_PERIODIC);

}

// OnChequeo : Esta funcion presenta un cuadro de dialogo que informa
//  de los valores que se eligieron como tiempo de muestreo, tiempo
// final y tipo de control en el experimento. Se ejecuta cuando el
// usuario activa la opcion 'Tiempos' del menú principal
void DMypintView::OnChequeo()
{
	DLimites dlg;
	dlg.m_ftiempomax=m_fmaxtime;
	dlg.m_ftiempomuest=m_fsampletime;
	dlg.DoModal();
}

// OnCerrar : Esta funcion es la encargada de guardar los datos
```

```
// obtenidos en el experimento en el fichero designado
// por el usuario antes de comenzar el experimento

// OnCerrar : Esta funcion es la encargada de cerrar a aplicacion
// Se invoca al seleccionar la opcion 'Cerrar Tarjeta' del menú
// principal. OnCerrar cierra la tarjeta comprobando y avisando
// si ha habido fallos y sacando antes de cerrar la tarjeta una
// salida de 0 voltios para parar el motor del ventilador. Tambien
// cierra el fichero que estuviera abierto aunque no guarda los
// datos, que en una sesion normal se habrian guardado ya cuando
// terminara el ultimo experimento
void DMypintView::OnCerrar()
{

      // Cerrar el fichero abierto
      n_myfichero.Close();

      // Sacar 0 voltios para parar el ventilador
    long value = (long) ((1L<<RESOLUTION)/(m_damax-m_damin) * (0.f -
m_damin));
    value = min((1L<<RESOLUTION)-1,value);
      olDaPutSingleValue(m_board2.hdass,value,0,1.0);

      // Cerrar la tarjeta comprobando si hubo fallo
      // El proceso es analogo al de abrir la tarjeta
      ECODE fallo;
      fallo=olDaReleaseDASS(m_board.hdass);
      if (!fallo)
      {     fallo=olDaReleaseDASS(m_board2.hdass);
            if(!fallo)
            {     fallo=olDaTerminate(m_board.hdrvr);
                  if(!fallo)
                  {     fallo=olDaTerminate(m_board2.hdrvr);     }
            }
      }
      if(fallo)
      {     AfxMessageBox("Ha  habido  fallo  al  cerrar  la  tarjeta",
MB_ICONSTOP );          }

      // Cierra la aplicacion al destruir la ventana de la vista
      GetParent()->DestroyWindow();
}

// OnNuevoExperimento : Esta funcion se ejecuta al seleccionar
// el usuario la opcion 'Nuevo Experimento' del menú principal
// Inicia un proceso de dialogo analogo al presentado cuando se
// inicia la aplicacion, con el fin de obtener los datos del
// nuevo experimento.
void DMypintView::OnNuevoExperimento()
{

      // Cerrar el fichero anterior
      n_myfichero.Close();

      // Inicializacion de algunas variables miembro
      m_uIndLec = 0;
      m_uIndEsc = 0;
      m_fsum = 0;
```

```
// Dialogo para fijar el tiempo de muestreo y el tiempo final
DLimites dlg;
do
{
      dlg.m_ftiempomuest=m_fsampletime;
      dlg.m_ftiempomax=m_fmaxtime;
      dlg.m_umodo = m_umode;
}
while ( dlg.DoModal() == IDCANCEL );

m_fmaxtime=dlg.m_ftiempomax;
m_fsampletime=dlg.m_ftiempomuest;
m_umode = dlg.m_umodo;


// Pedir los valores de las constantes del PID : Kp, Td y Ti.
// si se ha elegido PID como tipo de control
if ( m_umode == 1 )
{
      DDialgoPID Dlg_Pid;
      do
      {
            Dlg_Pid.m_fKp_Dlg = m_fKp;
            Dlg_Pid.m_fTd_Dlg = m_fTd;
            Dlg_Pid.m_fTi_Dlg = m_fTi;
      }
      while ( Dlg_Pid.DoModal() == IDCANCEL );
      m_fKp = Dlg_Pid.m_fKp_Dlg;
      m_fTd = Dlg_Pid.m_fTd_Dlg;
      m_fTi = Dlg_Pid.m_fTi_Dlg;
}

// Dialogo para fijar el nombre
CString nom_fichero; // variable auxiliar
int var; // variable auxiliar

DDialogoSalvar dlgsave( // llamo al constructor del dialogo
      FALSE,
      NULL,
      NULL,
      OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
      "Ficheros MATLAB (*.m)|*.m||",
      NULL);
// Aparicion del dialogo en pantalla
do
{
      var = dlgsave.DoModal();
}
while ( var == IDCANCEL );

// Asignar nombre al fichero
nom_fichero = dlgsave.GetPathName();
// incluyendo extension ".m" si no se ha puesto
if( dlgsave.GetFileExt().IsEmpty() )
{     nom_fichero += ".m";    }

// Abrir fichero y comprobar si hay errores
```

```
	if ( !n_myfichero.Open ( nom_fichero, CFile::modeCreate | CFile::modeWrite ) )
	{	AfxMessageBox ( "No se puede abrir el fichero" );return;
	}

	// Calcular el numero de muestras del experimento
	m_uNumMuest = (UINT) (m_fmaxtime / m_fsampletime)+1;

	// Limpiar los CArray y ajustar tamaño de los CArray
	// para el numero de muestras adecuado es opcional
	/*
	m_signal_u.RemoveAll();
	m_signal_e.RemoveAll();
	m_signal_y.RemoveAll();
	m_signal_yf.RemoveAll();
	m_signal_r.RemoveAll();
	m_signal_u.SetSize(m_uNumMuest+1);
	m_signal_e.SetSize(m_uNumMuest+1);
	m_signal_y.SetSize(m_uNumMuest+1);
	m_signal_yf.SetSize(m_uNumMuest+1);
	m_signal_r.SetSize(m_uNumMuest+1);
	*/

	// Generar la señal de referencia
	GenerarReferencia();

	// Redibujar la pantalla para los nuevos datos de tiempo
	OnDraw( GetDC() );

	if( m_bNoFiltrar )
	{
		// Calcular el tiempo de muestreo en milisegundos
		// En el caso de no tener activado el filtro
		UINT tiempo = (UINT) (1000*m_fsampletime);
	}
	else
	{
		// Calcular el tiempo de muestreo en milisegundos
		// En el caso de si tener activado el filtro
		UINT tiempo = (UINT) (1000*m_fsampletime/4);
	}

	// Crear el temporizador, con lo que da comienzo el experimento
	m_uIDTimer = timeSetEvent(tiempo,50,(LPTIMECALLBACK)NuestroProc,0L,TIME_PERIODIC);

}

// GenerarReferencia : Esta funcion es la encargada de generar la
// señal de referencia 'r' tanto en centimetros como en voltios. La
// señal en centimetros se guarda en la tabla m_signal_r y la señal
// en voltios en la tabla m_signal_rvo, siendo ambas variables miembro
// de la clase vista. Al ser variables miembro se puede acceder a ellas
// inmediatamente desde cualquier funcion de la clase vista
void DMypintView::GenerarReferencia()
{
```

```
	// Generar la señal de referencia : 'r' en centímetros
	// Esta funcion se puede modificar para cambiar la señal
	// de referencia y definirla como se desee. La señal
	// m_signal_r contiene la referencia en centimetros, y la
	// señal m_signal_rvo es la traduccion a voltios de la
	// señal m_signal_r usando la funcion CentsToVolts() que
	// tambien es modificable

	// Genera una señal 'r' que vale
	// 50 centimetros para 0 <= t < 0.25 * tfinal
	// 70 centimetros para 0.25 < t < 0.5 * tfinal
	// 20 centimetros para 0.5 < t < 0.75 * tfinal
	// 10 centimetros para 0.75 < t <= tfinal
	for ( UINT i=0;i<=m_uNumMuest;i++)
	{
		if(i<0.25*m_uNumMuest)	{	m_signal_r[i]	=	50;
m_signal_rvo[i] = CentsToVolts( m_signal_r[i] ); }
		else	if(i<0.5*m_uNumMuest)	{	m_signal_r[i]	=	70;
m_signal_rvo[i] = CentsToVolts( m_signal_r[i] ); }
		else	if(i<0.75*m_uNumMuest)	{	m_signal_r[i]	=	20;
m_signal_rvo[i] = CentsToVolts( m_signal_r[i] ); }
		else { m_signal_r[i] = 10; m_signal_rvo[i] = CentsToVolts(
m_signal_r[i] ) ; }
	}

	// Filtrar la referencia 'r'
	// El filtro elegido es de paso de baja de primer orden
	// con la forma H(z)= 1/(1+ a*z^-1)
	for ( i=1;i<=m_uNumMuest;i++)
	{
		m_signal_rvo[i]	=	0.2	*	m_signal_rvo[i]	+	0.8	*
m_signal_rvo[i-1];
	}

	return;
}

// CentsToVolts : Esta funcion transforma un dato de entrada de tipo
// float en un dato de salida de tipo float. La relacion entre los dos
// float es la caracteristica del sensor, de manera que
// para una altura en centimetros de la plataforma igual al valor de
// entrada a CentsToVolts, esta funcion daria un valor de salida igual
// al que da el sensor para la altura referida
float DMypintView::CentsToVolts(float dato)
{
	float salida;

	// Caracteristica del sensor
	// Para mas detalles ver el apartado dedicado al sensor
	// en la memoria del proyecto
	if ( dato < 40 )
	{
		salida = 0.0681f + 0.0146f * dato;
	}
	else
	{
		salida = -0.21f + 0.0216f * dato;
	}
```

```
	return salida;
}



// VToC : Esta funcion transforma un dato de entrada de tipo
// float en un dato de salida de tipo float. La relacion entre los dos
// float es la caracteristica inversa del sensor, de manera que para
// un valor de entrada igual al voltaje que da el sensor para una
altura
// de la plataforma VToC daria la altura en centimetros de la
plataforma
// Esta función es la inversa de CentsToVolts, y el resultado de
aplicar
// primero una y luego la siguiente usando la salida de la primera
sería
// el propio dato de entrada a la primera de las dos funciones.
float DMypintView::VToC(float dato)
{
	float salida;

	// Caracteristica inversa del sensor
	// Para mas detalles ver el apartado dedicado al sensor
	// en la memoria del proyecto
	if ( dato < 0.652f )
	{
		salida =  -0.0466f + 0.6849f * dato;
	}
	else
	{
		salida = 0.0972f + 0.4630f * dato;
	}
	salida *= 100; // Para obtener la salida en centímetros y no en
metros
	return salida;
}



// OnGuardar : Esta funcion se enccarga de guaradar en fichero
// los datos del experimento. Guarda en el fichero los datos
// siguientes, en formato MATLAB y en este orden:
// tiempo final del experimento como Tfinal
// tiempo de muestreo como 'Ts'
// señal de control aplicada como 'u'
// señal de salida medida como 'y'
// señal de referencia en centimetros como 'r'
// señal de referencia en voltios como rvo
// señal de salida medida tras ser filtrada como 'yf'
// señal de error e=r-yf como 'e'
LRESULT DMypintView::OnGuardar(WPARAM wParam, LPARAM lParam)
{
	// Crear un archivo asociado al fichero abierto
	// para poder utilizar el fichero a traves del archivo
	CArchive archivo(&n_myfichero,CArchive::store);

	// Variables de ayuda
	CString cs;
```

```
UINT i;

// Grabar el tiempo total del experimento en el fichero
cs.Format("Tfinal =  ");
archivo.WriteString( (LPCTSTR) cs);
cs.Format("%f;\n",m_fmaxtime);
archivo.WriteString( (LPCTSTR) cs);
cs.Format("]\n");

// Grabar el tiempo de muestreo en el fichero
cs.Format("Ts = ");
archivo.WriteString( (LPCTSTR) cs);
cs.Format("%f;\n",m_fsampletime);
archivo.WriteString( (LPCTSTR) cs);
cs.Format("]\n");

// Grabar señal 'u' en el fichero
cs.Format("u = [\n");
archivo.WriteString( (LPCTSTR) cs);
for (i=0;i<m_uIndEsc-1;i++)
{
      cs.Format("%f;\n",m_signal_u[i]);
      archivo.WriteString( (LPCTSTR) cs);
}
cs.Format("]\n");
archivo.WriteString( (LPCTSTR) cs);

// Grabar señal 'y' en el fichero
cs.Format("y = [\n");
archivo.WriteString( (LPCTSTR) cs);
for (i=0;i<m_uIndEsc-1;i++)
{
      cs.Format("%f;\n",m_signal_y[i]);
      archivo.WriteString( (LPCTSTR) cs);
}
cs.Format("]\n");
archivo.WriteString( (LPCTSTR) cs);

// Grabar señal 'r' en el fichero
cs.Format("r = [\n");
archivo.WriteString( (LPCTSTR) cs);
for (i=0;i<m_uIndEsc-1;i++)
{
      cs.Format("%f;\n",m_signal_r[i]);
      archivo.WriteString( (LPCTSTR) cs);
}
cs.Format("]\n");
archivo.WriteString( (LPCTSTR) cs);

// Grabar señal 'rvo' en el fichero
cs.Format("rvo = [\n");
archivo.WriteString( (LPCTSTR) cs);
for (i=0;i<m_uIndEsc-1;i++)
{
      cs.Format("%f;\n",m_signal_rvo[i]);
      archivo.WriteString( (LPCTSTR) cs);
}
cs.Format("]\n");
```

```
	archivo.WriteString( (LPCTSTR) cs);

	// Grabar señal 'yf' en el fichero
	cs.Format("yf = [\n");
	archivo.WriteString( (LPCTSTR) cs);
	for (i=0;i<m_uIndEsc-1;i++)
	{
		cs.Format("%f;\n",m_signal_yf[i]);
		archivo.WriteString( (LPCTSTR) cs);
	}
	cs.Format("]\n");
	archivo.WriteString( (LPCTSTR) cs);


	// Grabar señal 'e' en el fichero
	cs.Format("e = [\n");
	archivo.WriteString( (LPCTSTR) cs);
	for (i=0;i<m_uIndEsc-1;i++)
	{
		cs.Format("%f;\n",m_signal_e[i]);
		archivo.WriteString( (LPCTSTR) cs);
	}
	cs.Format("]\n");
	archivo.WriteString( (LPCTSTR) cs);

	return (0L);
}


// OnMyTimer : Esta funcion se encarga de pintar las nuevas
// muestras leidas cada vez que recibe un mensaje WM_MY_TIMER
// de la funcion call-back NuestroProc. Pinta en pantalla un pixel
// por muestra de cada señal, con el color escogido para cada señal
LRESULT DMypintView::OnMyTimer(WPARAM wParam, LPARAM lParam)
{
	m_dwtiempo_actual = (DWORD)lParam;
	UpdateData(FALSE);

	// Obtener el puntero al contexto de dispositivo
	CClientDC dc(this);

	// Obtener dimensiones,...
	// toda esta parte es igual a la que aparece en la
	// funcion OnDraw()
	CRect rect;
	GetClientRect( rect );
	int cx = ( rect.right - rect.left );
	int cy = (rect.bottom - rect.top );
	dc.SetBkColor( BK_COLOR );
	dc.SetMapMode(MM_ANISOTROPIC);
	int tam_x = (int) (RES_X*m_fmaxtime); // Tamaño del eje X
	int tam_y = (int) (RES_Y*(VAL_MAXIMO - VAL_MINIMO)); // Tamaño
del Y
	int total_x = (int) (1.2*(RES_X*m_fmaxtime)); // Tamaño más
margen
	int total_y = (int) (1.2*(RES_Y*(VAL_MAXIMO - VAL_MINIMO))); //
Idem
	int mov_x = total_x - tam_x;  // Origen de los ejes
```

```
	int mov_y = total_y - tam_y;  // Origen de los ejes
	// tamaño en unidades logicas de la ventana
	int pant_x = (int) (total_x + 0.1*tam_x); // tanto en el eje X
	int pant_y = (int) (total_y + 0.1*tam_y); // como en el eje Y
	int alt_fila1 = (int) (0.8 * mov_y);  // Altura de los numeros
de tiempo
	int anch_col1 = (int) (0.8 * mov_x); // Margen de los numeros de
tiempo
	int anch_col2 = (int) (0.4 * mov_x); // Margen de los numeros de
voltaje
	int alt_fila2 = (int) (1.1 * mov_y); // Altura de los numeros de
voltaje
	int alt_fila3 = (int) ( total_y+0.07*tam_y ); // Altura de la
leyenda de las señales

	// Variables de ayuda
	int i;
	CString cadenita;

	// Fijar los tamaños de la ventana
	dc.SetWindowExt( pant_x, pant_y );
	dc.SetViewportExt( cx, -cy );
	dc.SetViewportOrg( 0, cy );

	// Pintar los nuevos datos en la pantalla y refrescar los
antiguos
	for (i=0;i<=m_uIndLec;i++)
	{
		int pos_x=(int) (mov_x+m_fsampletime*i*RES_X);

		dc.SetPixel(pos_x,(int) (mov_y+ m_signal_rvo[i] *RES_Y),
COLOR_R );
		dc.SetPixel(pos_x,(int) (mov_y+ m_signal_u[i] *RES_Y),
COLOR_U );
		dc.SetPixel(pos_x,(int) (mov_y+ m_signal_y[i] *RES_Y),
COLOR_YF );
		dc.SetPixel(pos_x,(int) (mov_y+ (float) fabs(m_signal_e[i])
*RES_Y), COLOR_E);
	}

	// Incrementar el indice que indica el numero de muestras
	// que han sido representadas en la pantalla
	m_uIndLec++;

	return (0L);
}



/////////////////////////////////////////////////////////////////////
///////
// DDialgoPID dialog : Implementacion de la clase DDialgoPID, que es
la
// clase dialogo que sirve para presentar el dialogo donde se
introducen
// los valores de las constantes del PID
// No hace falta modificar esta parte
```

```
DDialgoPID::DDialgoPID(CWnd* pParent /*=NULL*/)
	: CDialog(DDialgoPID::IDD, pParent)
{
	//{{AFX_DATA_INIT(DDialgoPID)
	m_fKp_Dlg = 0.0f;
	m_fTd_Dlg = 0.0f;
	m_fTi_Dlg = 0.0f;
	//}}AFX_DATA_INIT
}

void DDialgoPID::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(DDialgoPID)
	DDX_Text(pDX, IDC_KP, m_fKp_Dlg);
	DDX_Text(pDX, IDC_TD, m_fTd_Dlg);
	DDX_Text(pDX, IDC_TI, m_fTi_Dlg);
	//}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(DDialgoPID, CDialog)
	//{{AFX_MSG_MAP(DDialgoPID)
		// NOTE: the ClassWizard will add message map macros here
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////
///////
// DDialgoPID message handlers

// OnChequeoPid : Esta funcion se ejecuta cuando el usuario selecciona
// la opcion 'Constantes' del menú principal, y sirve para que pueda
// ver un cuadro con informacion de los valores de las constantes del
PID
void DMypintView::OnChequeoPid()
{
	DDialgoPID Dlg_Pid;
	Dlg_Pid.m_fKp_Dlg = m_fKp;
	Dlg_Pid.m_fTd_Dlg = m_fTd;
	Dlg_Pid.m_fTi_Dlg = m_fTi;
	Dlg_Pid.DoModal();
}

// OnRepintarTodo : Esta funcion se ejecuta cuando el usuario
selecciona
// la opcion 'Repintar Todo' del menú. Se encarga de pintar las
señales
// con linea continua mediante una llamada a OnDraw()
void DMypintView::OnRepintarTodo()
{
	OnDraw( GetDC() );
}


/////////////////////////////////////////////////////////////////////////
///////
// DLimites dialog : Implementacion de la clase DLimites, que es la
```

```
// clase dialogo que sirve para presentar el dialogo donde se
introducen
// los valores del tiempo de muestreo, tiempo final y tipo de control
del
// experimento
// No hace falta modificar esta parte

// Declaracion de la clase DLimites
DLimites::DLimites(CWnd* pParent /*=NULL*/)
	: CDialog(DLimites::IDD, pParent)
{
	//{{AFX_DATA_INIT(DLimites)
	m_ftiempomax = 0.0f;
	m_ftiempomuest = 0.0f;
	m_umodo = 1;
	//}}AFX_DATA_INIT
}


void DLimites::DoDataExchange(CDataExchange* pDX)
{
	CDialog::DoDataExchange(pDX);
	//{{AFX_DATA_MAP(DLimites)
	DDX_Text(pDX, IDC_TIEMPO_MAXIMO, m_ftiempomax);
	DDV_MinMaxFloat(pDX, m_ftiempomax, 1.f, 820.f);
	DDX_Text(pDX, IDC_TIEMPO_MUES, m_ftiempomuest);
	DDV_MinMaxFloat(pDX, m_ftiempomuest, 1.e-002f, 100.f);
	DDX_Text(pDX, IDC_MODO, m_umodo);
	DDV_MinMaxUInt(pDX, m_umodo, 0, 3);
	//}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(DLimites, CDialog)
	//{{AFX_MSG_MAP(DLimites)
		// NOTE: the ClassWizard will add message map macros here
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

////////////////////////////////////////////////////////////////////////
///////
// DLimites message handlers



////////////////////////////////////////////////////////////////////////
///////
// DDialogoSalvar :Implementacion de la clase DDialogoSalvar, que es
la
// clase dialogo que sirve para presentar el dialogo donde se
introduce
// el nombre del fichero donde se guardaran los datos del experimento
// No hace falta modificar esta parte

IMPLEMENT_DYNAMIC(DDialogoSalvar, CFileDialog)

DDialogoSalvar::DDialogoSalvar(BOOL	bOpenFileDialog,	LPCTSTR
lpszDefExt, LPCTSTR lpszFileName,
```

```
            DWORD dwFlags, LPCTSTR lpszFilter, CWnd* pParentWnd) :
            CFileDialog(bOpenFileDialog,    lpszDefExt,    lpszFileName,
dwFlags, lpszFilter, pParentWnd)
{
}


BEGIN_MESSAGE_MAP(DDialogoSalvar, CFileDialog)
      //{{AFX_MSG_MAP(DDialogoSalvar)
            // NOTE - the ClassWizard will add and remove mapping
macros here.
      //}}AFX_MSG_MAP
END_MESSAGE_MAP()


// NuestroProc es la funcion callback que es llamada cada vez que se
// se cumple el tiempo de muestreo. Se encarga de muestrear la señal
// del sensor, calcular la señal de control y sacarla por la tarjeta
// y mandar un mensaje WM_MY_TIMER a la vista para que dibuje los
nuevos
// datos. Cuando se ha cumplido el tiempo final del experimento se
// encarga de matar el temporizador y avisar a la vista para que
// almacene los datos en un fichero y para que actualiza la grafica
// y pinte las señales con linea continua.
// Para avisar a la vista de que debe guardar los datos en fichero
// manda un mensaje WM_MY_GUARDAR que ejecuta OnGuardar()
// y para refrescar la pantalla manda un mensaje WM_PAINT que hace
// que se ejecute OnDraw()
void   CALLBACK   NuestroProc(UINT   uID,UINT   uMsg,DWORD   dwUser,DWORD
dw1,DWORD dw2)
{
      // Obtener punteros a la aplicacion, el marco y a la vista
      DWORD tiempo_actual=timeGetTime();
      DMypintApp *pApp = (DMypintApp *)AfxGetApp();
      DMainFrame     *pMain     =     (DMainFrame     *)pApp->m_pMainWnd;
//AfxGetMainWnd();
      DMypintView *pView = (DMypintView *)pMain->GetActiveView();

      // Copiar datos de la clase vista a variables locales para
      // simplificar la notacion
      DBL min1 = pView->m_admin;
      DBL max1 = pView->m_admax;
      DBL min2 = pView->m_damin;
      DBL max2 = pView->m_damax;
      float volts;
      long value;

      // Cuando se ha llegado al final del experimento
      if(pView->m_uIndEsc == pView->m_uNumMuest )
      {
            // Acaba con el temporizador
            timeKillEvent(pView->m_uIDTimer);
            // Saca 0 voltios por la tarjeta para parar el ventilador
            value = (long) ((1L<<RESOLUTION)/((float)max2-(float)min2)
* (0.f - (float)min2));
            value = min((1L<<RESOLUTION)-1,value);
            olDaPutSingleValue(pView->m_board2.hdass,value,0,1.0);
            pView->m_uIndEsc++;
```

```
            // Llamada a pintar los datos finales y repintar todo
            pView->PostMessage(WM_PAINT,0,0);
            // Llamada a guardar datos leidos
            pView->PostMessage(WM_MY_GUARDAR,0,0);

            return;
      }

      // Si aun no termino el experimento...

      // Muestrear la señal del sensor
      olDaGetSingleValue(pView->m_board.hdass,&value,0,1);
      // Obtener el valor si la codificacion no fuera binaria
      // En el caso de la DT2811 siempre lo es, pero para que el
      // programa sea mas general y valga para otras tarjetas, se
      // ha incluido este bloque
      if ( ENCODING != OL_ENC_BINARY )
      {
            value ^= 1L << (RESOLUTION-1);
            value &= (1L << RESOLUTION) -1;
      }
    volts  =  ((float)max1-(float)min1)/(1L<<RESOLUTION)  *  value  +
(float)min1;

      // Si es la cuarta muestra del sobremuestreo
      if ( pView->m_uSub == 3 || pView->m_bNoFiltrar )
      {

      // Escribir el valor de la muestra de la señal 'y' en la tabla
      pView->m_signal_y[pView->m_uIndEsc]=volts;

      // Limitarla y filtrarla...
      if ( volts > 1.843f ) { volts = 1.843f; }

      if ( !pView->m_bNoFiltrar )
      {
            // Filtrar yf
            if(m_uIndEsc > 3)
            {
            pView->m_signal_yf[pView->m_uIndEsc]=volts;
            float m1 = pView->m_sub[0];
            float m2 = pView->m_sub[1];
            float m3 = pView->m_sub[2];
            float m4 = volts;
            UINT l = pView->m_uIndEsc;
            float numer = -0.438f*m1–1.3139f*m2-1.3139f*m3-0.438f* m4;
            float  denom  =  5.2f  *  pView->m_signal_yf[l-1]  -8.6962f  *
            pView->m_signal_yf[l-2] + 7.9994f *pView->m_signal_yf[l-3];
            m_signal_yf[l]= numer + denom;
            }
            else
            {
                  pView->m_signal_yf[pView->m_uIndEsc]=volts;
            }

      }
      else
```

```
      {
            pView->m_signal_yf[pView->m_uIndEsc]=volts;
      }

      // Calcular e
      pView->m_signal_e[pView->m_uIndEsc]    =    (pView->m_signal_rvo
[pView->m_uIndEsc]) - (pView->m_signal_yf[pView->m_uIndEsc]);

      // Calcular salida en voltios 'u'....

      // Datos necesarios para PID en variables locales para
      // mayor sencillez de notacion
    float kp = pView->m_fKp;
      float td = pView->m_fTd;
      float ti = max(pView->m_fTi,0.001f);
      float ts = pView->m_fsampletime;
      float su = pView->m_fsum;
      UINT modo = pView->m_umode;
      float valor_volts;
      int men_1 = max(0,(int)(pView->m_uIndEsc)-1);
      int in = pView->m_uIndEsc;
      // Si el modo de control es 'predefinido' la señal 'u'
      // es independiente de la señal de entrada
      if( modo == 0 )
      {
            // Se utiliza una señal 'u' prededinida. Es util
            // para calentar el motor antes de empezar los experimentos
            UINT total = pView->m_uNumMuest;
            if( in < 0.2*total )    { valor_volts = 4; }
            else if( in < 0.4*total) { valor_volts = 3.5; }
            else if( in < 0.6*total) { valor_volts = 5; }
            else if( in < 0.75*total) { valor_volts = 3; }
            else if( in < 0.9*total) { valor_volts = 2; }
            else
            { valor_volts = 0.5; }
            pView->m_signal_u[pView->m_uIndEsc]=valor_volts;
      }

      // Si el modo de control es PID
      else if( modo == 1 )
      {
            float ek = pView->m_signal_e[pView->m_uIndEsc];
            float ek1 = pView->m_signal_e[men_1];
            su+=ek;
            valor_volts = kp*( ek + su*ts/ti + (ek-ek1) * td/ts );
            pView->m_signal_u[pView->m_uIndEsc]=valor_volts;
      }

      /*
      // Este modo es el de control predictivo
      else if( modo == 2 )
      {
            DMypintView *p = pView;
            int men_2 = max(0,(int)(p->m_uIndEsc)-2);
            int men_3 = max(0,(int)(p->m_uIndEsc)-3);
            int men_4 = max(0,(int)(p->m_uIndEsc)-4);
            int mas_1 = min((int)p->m_uNumMuest,(int)(p->m_uIndEsc)+1);
            int mas_2 = min((int)p->m_uNumMuest,(int)(p->m_uIndEsc)+2);
```

```
            int mas_3 = min((int)p->m_uNumMuest,(int)(p->m_uIndEsc)+3);

            float parte_u = 0.79f * p->m_signal_u[men_1] + 0.21f * p->
m_signal_u[men_2];
            float parte_yvo = (-55.f * p->VToC(p->m_signal_yf[in])+67.f
*   p->VToC(p->m_signal_yf[men_1])-26.f*p->VToC(p->m_signal_yf[men_2])-
2.4f*p->VToC(p->m_signal_yf[men_3])+3.8f*p->VToC(p->
m_signal_yf[men_4]));
            float parte_r = (2.5f * p->m_signal_r[mas_1] +4.9f * p->
m_signal_r[mas_2]+5.2f * p->m_signal_r[mas_3]);

            valor_volts = parte_u + parte_r + parte_yvo;
            pView->m_signal_u[pView->m_uIndEsc]=valor_volts;
            valor_volts += 1.564f;

      }
      */

      /*
      // Si está definido en PIDs zonales
      // Este modo se eliminara en la version final
      // de la aplicacion porque no resulta muy util
      else if ( modo == 3 )
      {
            float ek = pView->m_signal_e[pView->m_uIndEsc];
            float ek1 = pView->m_signal_e[in];
            if ( pView->m_signal_rvo[pView->m_uIndEsc] > 1.195f )
            {
                  float kp2 = 0.3f;
                  //float sum = ti*(kp-kp2)*ek/(ts*kp2) + su*kp/kp2;
                  //su = sum;
                  kp = kp2;
            }
            else if( pView->m_signal_rvo[pView->m_uIndEsc] > 0.2871f )
            {
                  float kp2 = 0.7f;
                  //float sum = ti*(kp-kp2)*ek/(ts*kp2) + su*kp/kp2;
                  //su = sum;
                  kp = kp2;
            }
            else
            {
                  float kp2 = 0.4f;
                  //float sum = ti*(kp-kp2)*ek/(ts*kp2) + su*kp/kp2;
                  //su = sum;
                  kp = kp2;
            }
            valor_volts = kp*( ek + su*ts/ti + (ek-ek1) * td/ts );
            su+=ek;
            pView->m_signal_u[pView->m_uIndEsc]=valor_volts;

      }
      */

      // Y por defecto saca una señal 'u' de tres voltios
      else
      {
            valor_volts = 3;
```

```
            pView->m_signal_u[pView->m_uIndEsc]=valor_volts;
      }

      // Comprobar que 'u' no sale de los límites
      if ( valor_volts > max2) { valor_volts = max2; }
      if ( valor_volts < VAL_MINIMO) { valor_volts = 0; }

      // Convertir el valor de voltios a la codificacion de la tarjeta
    value   =   (long)   ((1L<<RESOLUTION)/((float)max2-(float)min2)   *
(valor_volts - (float)min2));
    value = min((1L<<RESOLUTION)-1,value);
      olDaPutSingleValue(pView->m_board2.hdass,value,0,1.0);
      pView->m_fsum = su;

      // Incrementar m_uIndEsc, numero de muestras leidas
      pView->m_uIndEsc++;
      pView->PostMessage(WM_MY_TIMER,0,(LPARAM) tiempo_actual);
      pView->m_uSub = 0;

      } // Cierre del if de submuestreo
      else
      {
            pView->m_sub[pView->m_uSub] = volts;
            pView->m_uSub++;
      }
}
```

StdAfx.cpp

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO


// stdafx.cpp : source file that includes just the standard includes
//    mypint.pch will be the pre-compiled header
//    stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

MainFrm.h

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO

// MainFrm.h : Definiciones de la clase DMainFrame
//
/////////////////////////////////////////////////////////////////////////
///////

#if
!defined(AFX_MAINFRM_H__B7DD9DBE_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED
_)
#define AFX_MAINFRM_H__B7DD9DBE_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_

#if _MSC_VER >= 1000
```

```
#pragma once
#endif // _MSC_VER >= 1000

class DMainFrame : public CFrameWnd
{
protected: // create from serialization only
      DMainFrame();
      DECLARE_DYNCREATE(DMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(DMainFrame)
      virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~DMainFrame();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
      //{{AFX_MSG(DMainFrame)
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////
///////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_MAINFRM_H__B7DD9DBE_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED
_)
```

mypint.h

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO


// mypint.h : Fichero de cabecera principal de la aplicacion
//
```

```
#if
!defined(AFX_MYPINT_H__B7DD9DBA_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_
)
#define AFX_MYPINT_H__B7DD9DBA_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#ifndef __AFXWIN_H__
      #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"       // main symbols
// Se debe añadir la biblioteca math.h para poder realizar operaciones
#include <math.h>

// definicion del mensaje del temporizador
#define           WM_MY_TIMER       (WM_USER+0)
// definicion del mensaje de almacenamiento en fichero
#define           WM_MY_GUARDAR     (WM_USER+1)

/////////////////////////////////////////////////////////////////////////
///////
// DMypintApp:
// See mypint.cpp for the implementation of this class
//

class DMypintApp : public CWinApp
{
public:
      DMypintApp();

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(DMypintApp)
      public:
      virtual BOOL InitInstance();
      //}}AFX_VIRTUAL

// Implementation

      //{{AFX_MSG(DMypintApp)
      afx_msg void OnAppAbout();
// NOTE - the ClassWizard will add and remove member functions here.
//    DO NOT EDIT what you see in these blocks of generated code !
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////
///////

//{{AFX_INSERT_LOCATION}}
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.
```

```
#endif                                                                  //
!defined(AFX_MYPINT_H__B7DD9DBA_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_
)
```

mypintDoc.h

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO

// mypintDoc.h : Definiciones para la clase DMypintDoc
//
/////////////////////////////////////////////////////////////////////////
///////

#if
!defined(AFX_MYPINTDOC_H__B7DD9DC0_ABCF_11D2_B2CE_E688AD09EF3F__INCLUD
ED_)
#define
AFX_MYPINTDOC_H__B7DD9DC0_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000


class DMypintDoc : public CDocument
{
protected: // create from serialization only
      DMypintDoc();
      DECLARE_DYNCREATE(DMypintDoc)

// Attributes
public:

// Operations
public:

// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(DMypintDoc)
      public:
      virtual BOOL OnNewDocument();
      virtual void Serialize(CArchive& ar);
      //}}AFX_VIRTUAL

// Implementation
public:
      virtual ~DMypintDoc();
#ifdef _DEBUG
      virtual void AssertValid() const;
      virtual void Dump(CDumpContext& dc) const;
#endif

protected:
```

```
// Generated message map functions
protected:
      //{{AFX_MSG(DMypintDoc)
            // NOTE - the ClassWizard will add and remove member
functions here.
            //    DO NOT EDIT what you see in these blocks of generated
code !
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////
///////

//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations
immediately before the previous line.

#endif                                                                //
!defined(AFX_MYPINTDOC_H__B7DD9DC0_ABCF_11D2_B2CE_E688AD09EF3F__INCLUD
ED_)
```

mypintView.h

```
// mypintView.h : interface of the DMypintView class


/////////////////////////////////////////////////////////////////////
///////

#if
!defined(AFX_MYPINTVIEW_H__B7DD9DC2_ABCF_11D2_B2CE_E688AD09EF3F__INCLU
DED_)
#define
AFX_MYPINTVIEW_H__B7DD9DC2_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000


// Includes necesarios para la tarjeta y sus funciones
// Olmem.h contiene las definiciones de las funciones que trabajan con
los buffer
#include <olmem.h>
// Olerrors.h contiene las definiciones de funciones y codigos para
tratar errores
#include <olerrors.h>
// Oldaapi.h contiene las definiciones de las funciones de entrada y
salida de la tarjeta
#include <oldaapi.h>

// Include necesario para trabajar con los templates, como, por
ejemplo, CArray.
```

```
#include <afxtempl.h>

// Definiciones que usa la tarjeta
#define STRLEN 80
#define TARJETA "DT2811"  // Nombre de la tarjeta
#define RESOLUTION 12 // Resolucion de la tarjeta
#define ENCODING OL_ENC_BINARY // Codificacion de los valores leidos
#define BORDE 40 // Espacio en pixeles entre el recuadro y los bordes
de la pantalla
#define N_EJES 10 // Numero de ejes que cortan el recuadro
#define RES_X 10 // Resolucion: nº unidades lógicas por segundo en eje
X
#define RES_Y 1000 // Resolucion: nº unidades lógicas por segundo en
eje Y
// La resolución define la precision con que se ven los puntos en la
// pantalla. A mayor resolucion mas precision, pero tambien esta mas
//limitado el rango del tiempo y del voltaje. Con estos valores se
puede
// tener un tiempo maximo del experimento de 250 segundos y un rango
de
// tensiones de 25 voltios

// Definicion de los colores que se van a usar en las graficas
#define VERDE RGB(0,255,0)
#define NEGRO RGB(0,0,0)
#define BLANCO RGB(255,255,255)
#define ROJO RGB(255,0,0)
#define AZUL_MAR RGB(0,0,128)
#define AMARILLO RGB(255,255,0)
#define PURPURA RGB(128,0,128)
#define VERDE_OLIVA RGB(128,128,0)
#define PLATA RGB(128,128,128)

// Definicion de los colores que usa cada elemento de la grafica
#define COLOR_EJES AZUL_MAR
#define COLOR_LEYENDA VERDE_OLIVA
#define COLOR_U PURPURA
#define COLOR_YF VERDE
#define COLOR_R ROJO
#define COLOR_E PLATA
#define BK_COLOR BLANCO

// Definicion del rango de representacion del voltaje en el eje Y
#define VAL_MAXIMO 5
#define VAL_MINIMO 0

// Definicion de la estructura logica de la tarjeta
typedef struct tag_board {
  HDRVR hdrvr;        /* driver handle            */
   HDASS hdass;        /* sub system handle        */
   ECODE status;       /* board error status       */
   HBUF  hbuf;         /* sub system buffer handle */
   WORD FAR* lpbuf;    /* buffer pointer           */
   char name[STRLEN];  /* string for board name    */
   char entry[STRLEN]; /* string for board name    */
} BOARD;

typedef BOARD FAR* LPBOARD;
```

```
// Hasta aqui llegan las definiciones necesarias


// Definicion de la clase DMypintView que es la que contiene toda la
funcionalidad del programa
class DMypintView : public CView
{
protected: // create from serialization only
      DMypintView();
      DECLARE_DYNCREATE(DMypintView)

// Attributes
public:
      DMypintDoc* GetDocument();
      float m_fmaxtime;  // Duración del experimento
      float m_fsampletime; // Tiempo de muestreo
      UINT m_umode;     // Tipo de control : PID o predictivo.
      CFile  n_myfichero;  //  Fichero  dende  se  almacenaran  los
resultados
      DWORD m_dwtiempo_actual; // Tiempo de lectura y escritura de los
valores de señales
      BOARD m_board;    //  Estructura  usada  para  trabajar  con  la
tarjeta en operaciones de A/D
      BOARD m_board2;   //  Estructura  usada  para  trabajar  con  la
tarjeta en operaciones de D/A
      DBL m_damin;      // Valor minimo de salida del conversor D/A
      DBL m_damax;      // Valor maximo de salida del conversor D/A
      DBL m_admin;      // Valor minimo de entrada del conversor A/D
      DBL m_admax;      // Valor maximo de entrada del conversor A/D

      // Tabla para guardar la señal 'u'
      CArray<float,float> m_signal_u;
      // Tabla para guardar la señal 'e'
      CArray<float,float> m_signal_e;
      // Tabla para guardar la señal 'y'
      CArray<float,float> m_signal_y;
      // Tabla para guardar la señal y filtrada
      CArray<float,float> m_signal_yf;
      // Tabla para guardar la señal de referencia en centimetros
      CArray<float,float> m_signal_r;
      // Tabla para guardar la señal de referencia en voltios
      CArray<float,float> m_signal_rvo;
      // Tabla para las submuestras
      CArray<float,float> m_sub;

      // Numero de submuestras tomadas
      UINT m_uSub;

      // Variable Lógica para activar o desctivar el filtro
      BOOL m_bNoFiltrar;

      // Identificador del temporizador
      UINT m_uIDTimer;

      // Numero de muestras presentadas en pantalla
      UINT m_uIndLec;
```

```
	// Numero de muestras leidas por la tarjeta
	UINT m_uIndEsc;

	// Numero total de muestras a tomar en el experimento
	UINT m_uNumMuest;

	// Valores de las constantes del PID
	float m_fKp;	// Valor de la constante proporcional
	float m_fTd;	// Valor del tiempo derivativo en segundos
	float m_fTi;	// Valor del tiempo integral en segundos
	float m_fsum;	// Sumatorio de los errores en cada tiempo de
muestreo

// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(DMypintView)
	public:
	virtual void OnDraw(CDC* pDC);  //  Redefinida para el programa
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
	virtual void OnInitialUpdate();
	//}}AFX_VIRTUAL


// Implementation
public:
	virtual ~DMypintView();
	float CentsToVolts( float dato );
	float VToC( float dato );
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Funciones del mapa de mensajes de la clase vista de la aplicacion
protected:
	//{{AFX_MSG(DMypintView)
	afx_msg void OnChequeo();
	afx_msg void OnCerrar();
	afx_msg void OnNuevoExperimento();
	afx_msg LRESULT OnMyTimer(WPARAM wParam, LPARAM lParam);
	afx_msg LRESULT OnGuardar(WPARAM wParam, LPARAM lParam);
	afx_msg void OnChequeoPid();
	afx_msg void OnRepintarTodo();
	//}}AFX_MSG
	void GenerarReferencia();
	DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG  // debug version in mypintView.cpp
inline DMypintDoc* DMypintView::GetDocument()
   { return (DMypintDoc*)m_pDocument; }
#endif
```

```
/////////////////////////////////////////////////////////////////////////
///////



/////////////////////////////////////////////////////////////////////////
/
// DLimites dialog : Dialogo para fijar tiempo muestreo, duración y
//                   tipo de control

// Definicion de la clase DLimites, que es un dialogo para fijar el
tiempo
// de muestreo, la duracion del experimento y el tipo de control
aplicado

class DLimites : public CDialog
{
// Construction
public:
      DLimites(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
      //{{AFX_DATA(DLimites)
      enum { IDD = IDD_DIALOG2 };
      float m_ftiempomax;
      float m_ftiempomuest;
      UINT  m_umodo;
      //}}AFX_DATA


// Overrides
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(DLimites)
      protected:
      virtual void DoDataExchange(CDataExchange* pDX);     // DDX/DDV
support
      //}}AFX_VIRTUAL

// Implementation
protected:

      // Generated message map functions
      //{{AFX_MSG(DLimites)
            // NOTE: the ClassWizard will add member functions here
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};



/////////////////////////////////////////////////////////////////////////
///////
// DDialogoSalvar dialog : Dialogo para elegir el fichero donde
guardar
//                                    los resultados del experimento

// Definicion de la clase DDialogoSalvar, que es un dialogo para fijar
```

```
// el nombre del fichero donde se almacenaran los resultados

class DDialogoSalvar : public CFileDialog
{
      DECLARE_DYNAMIC(DDialogoSalvar)

public:
      DDialogoSalvar(BOOL bOpenFileDialog, // TRUE for FileOpen, FALSE
for FileSaveAs
            LPCTSTR lpszDefExt = NULL,
            LPCTSTR lpszFileName = NULL,
            DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
            LPCTSTR lpszFilter = NULL,
            CWnd* pParentWnd = NULL);

protected:
      //{{AFX_MSG(DDialogoSalvar)
            // NOTE - the ClassWizard will add and remove member
functions here.
      //}}AFX_MSG
      DECLARE_MESSAGE_MAP()
};


// Prototipo de la funcion callback que usa la aplicacion
// Esta funcion no pertenece a ninguna clase y por tanto esta fuera
// de todo excepto de la aplicacion aunque se defina en uno de los
// ficheros de definicion clases
void CALLBACK NuestroProc(UINT uID,UINT uMsg,DWORD dwUser,DWORD
dw1,DWORD dw2);


/////////////////////////////////////////////////////////////////////
///////
// DDialgoPID dialog : Dialogo para fijar los valores de las contantes
//                               del PID


// Definicion de la clase DDialgoPID, que es un dialogo para poder
fijar
// los valores de las constantes del PID

class DDialgoPID : public CDialog
{
// Construction
public:
      DDialgoPID(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
      //{{AFX_DATA(DDialgoPID)
      enum { IDD = IDD_DIALOG3 };
      float m_fKp_Dlg;
      float m_fTd_Dlg;
      float m_fTi_Dlg;
      //}}AFX_DATA


// Overrides
```

```
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(DDialgoPID)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV
support
	//}}AFX_VIRTUAL

// Implementation
protected:

	// Generated message map functions
	//{{AFX_MSG(DDialgoPID)
		// NOTE: the ClassWizard will add member functions here
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
//{{AFX_INSERT_LOCATION}}
//  Microsoft  Developer  Studio  will  insert  additional  declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_MYPINTVIEW_H__B7DD9DC2_ABCF_11D2_B2CE_E688AD09EF3F__INCLU
DED_)
```

StdAfx.h

```
// NO SE DEBE TOCAR EL CONTENIDO DE ESTE FICHERO

//stdafx.h : fichero de includes donde incluir los ficheros de include
//del sistema mas habituales, o aquellos especificos de la aplicacion
//que se usen mucho pero que se modifiquen con poca frecuencia
//

#if
!defined(AFX_STDAFX_H__B7DD9DBC_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_
)
#define AFX_STDAFX_H__B7DD9DBC_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_

#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000

#define VC_EXTRALEAN		// Exclude rarely-used stuff from Windows
headers

#include <afxwin.h>		// MFC core and standard components
#include <afxext.h>		// MFC extensions
#include <afxdisp.h>		// MFC OLE automation classes
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>			// MFC support for Windows Common
Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
```

```
// Microsoft Developer Studio will insert additional declarations
immediately before the previous line.

#endif                                                                  //
!defined(AFX_STDAFX_H__B7DD9DBC_ABCF_11D2_B2CE_E688AD09EF3F__INCLUDED_
)
```

ReadMe.txt

```
========================================================================
       MICROSOFT FOUNDATION CLASS LIBRARY : mypint
========================================================================


AppWizard has created this mypint application for you.  This
application not only demonstrates the basics of using the Microsoft
Foundation classes but is also a starting point for writing your
application.

This file contains a summary of what you will find in each of the
files that make up your mypint application.

mypint.h
    This is the main header file for the application.  It includes
other project specific headers (including Resource.h) and declares the
DMypintApp application class.

mypint.cpp
    This is the main application source file that contains the
application class DMypintApp.

mypint.rc
    This is a listing of all of the Microsoft Windows resources that
the program uses.  It includes the icons, bitmaps, and cursors that
are stored in the RES subdirectory.  This file can be directly edited
in Microsoft Developer Studio.

res\mypint.ico
    This is an icon file, which is used as the application's icon.
This icon is included by the main resource file mypint.rc.

res\mypint.rc2
    This file contains resources that are not edited by Microsoft
Developer Studio.  You should place all resources not editable by the
resource editor in this file.

mypint.clw
    This file contains information used by ClassWizard to edit
existing classes or add new classes.  ClassWizard also uses this file
to store information needed to create and edit message maps and dialog
data maps and to create prototype member functions.

/////////////////////////////////////////////////////////////////////
///////
```

```
For the main frame window:

MainFrm.h, MainFrm.cpp
    These files contain the frame class DMainFrame, which is derived
from CFrameWnd and controls all SDI frame features.


/////////////////////////////////////////////////////////////////////////
///////

AppWizard creates one document type and one view:

mypintDoc.h, mypintDoc.cpp - the document
    These files contain your DMypintDoc class.  Edit these files to
add your special document data and to implement file saving and
loading (via DMypintDoc::Serialize).

mypintView.h, mypintView.cpp - the view of the document
    These files contain your DMypintView class. DMypintView objects
are used to view DMypintDoc objects.



/////////////////////////////////////////////////////////////////////////
///////
Other standard files:

StdAfx.h, StdAfx.cpp
    These files are used to build a precompiled header (PCH) file
    named mypint.pch and a precompiled types file named StdAfx.obj.

Resource.h
    This is the standard header file, which defines new resource IDs.
    Microsoft Developer Studio reads and updates this file.

/////////////////////////////////////////////////////////////////////////
///////
Other notes:

AppWizard uses "TODO:" to indicate parts of the source code you
should add to or customize.

If your application uses MFC in a shared DLL, and your application is
in a language other than the operating system's current language, you
will need to copy the corresponding localized resources MFC40XXX.DLL
from the Microsoft Visual C++ CD-ROM onto the system or system32
directory,
and rename it to be MFCLOC.DLL.   ("XXX" stands for the language
abbreviation.
For example, MFC40DEU.DLL contains resources translated to German.)
If you don't do this, some of the UI elements of your application will
remain in the language of the operating system.

/////////////////////////////////////////////////////////////////////////
```

4.5 FUNCIONAMIENTO DE LA APLICACIÓN

En este apartado se explica cómo está construida la aplicación. Se comentan las clases que se han utilizado, y las variables y funciones de dichas clases.

No se comentan las clases generadas automáticamente por el asistente de Visual C++, puesto que puede decirse que su funcionamiento es transparente al usuario, y que no existe ninguna necesidad de conocer cómo están construidas ya que no deben ser modificadas. Sólo en el caso de la clase vista, que sí ha sido modificada durante el desarrollo de la aplicación, se hace un análisis de dicha clase.

Las clases generadas por el asistente son:

- **DMypintApp** : clase que genera la aplicación y construye las demás cuando se ejecuta el programa. Deriva de la clase CWinApp de las MFC. Se define en el fichero mypint.h, y se implementa en el fichero mypint.cpp.

- **DMainFrame** : clase marco de la aplicación. Deriva de la clase CFrameWnd de las MFC. Se define en el fichero MainFrm.h, y se implementa en el fichero MainFrm.cpp.

- **DMypintDoc** : clase documento de la aplicación. Deriva de la clase CDocument de las MFC. Se define en el fichero mypintDoc.h, y se implementa en el fichero mypintDoc.cpp.

- **DMypintView** : clase vista de la aplicación. Deriva de la clase CView de las MFC. Se define en el fichero mypintView.h, y se implementa en el fichero mypintView.cpp.

Nótese que la letra inicial del nombre de estas clases es una ‘D’ y no ‘C’. Es conveniente para facilitar la legibilidad del código, usar como letra inicial la ‘D’ para

las clases que se deriven de las clases de MFC, y mantener así la letra ‘C’ como inicial sólo de las clases pertenecientes a las MFC.

Además existen otras tres clases, que son:

- **DLimites** : clase que se utiliza para presentar el diálogo inicial de elección de tiempo de muestreo, tiempo final y tipo de control. Como es un diálogo, deriva de la clase CDialog de las MFC. Se define en el fichero mypintView.h, y se implementa en el fichero mypintView.cpp. A pesar de estar definida e implementada en los mismos ficheros que la clase vista, ambas clases son totalmente independientes.

- **DDialogoPID** : clase que se utiliza para presentar el diálogo de elección de los parámetros del PID. También se deriva de CDialog puesto que es un diálogo. Se define en el fichero mypintView.h, y se implementa en el fichero mypintView.cpp.

- **DDialogoSalvar** : clase que se utiliza para presentar el diálogo de elección del fichero donde guardar los resultados. Se deriva de la clase CFileDialog de las MFC. Se define en el fichero mypintView.h, y se implementa en el fichero mypintView.cpp.

Dado que casi todas las funciones interesantes, desde el punto de vista de control del sistema levitador, están en la clase vista, es conveniente analizar las principales funciones y variables que contiene esta clase mediante el análisis de los pasos que sigue la aplicación cuando se ejecuta..

Cuando se llama a la aplicación, se construye por primera vez la clase vista. En ese momento se ejecuta la función **OnInitialUpdate** perteneciente a la clase vista. En esta función primero se realiza el proceso de apertura de la tarjeta y comprobación de

errores de la tarjeta. A continuación, se crea un objeto de la clase DLimites y se presenta el diálogo de elección de tiempo de muestreo,...

Después la función OnInitialUpdate presenta el diálogo para elegir el fichero donde guardar los resultados. A medida que el usuario va introduciendo los datos requeridos por los cuadros de diálogo, dichos datos sirven para dar valor a varias variables de la clase vista. En el listado del fichero mypintView.h se puede ver al lado de la definición de cada variable cuál es la utilidad que tiene.

Luego, la función OnInitialUpdate llama a la función **GenerarReferencia**.

La función GenerarReferencia, perteneciente a la clase vista, se encarga de generar la señal de referencia. La señal de referencia es una variable de la clase vista, en concreto, una variable del tipo CArray (equivalente a una tabla en C) llamada m_signal_rvo. En m_signal_rvo es donde debe estar la señal de referencia que se pretende usar, y la función GenerarReferencia lo que hace es darle valor a los elementos de m_signal_rvo. Como la referencia se desea definir en centímetros, y m_signal_rvo se refiere a valores en voltios, se utiliza un segundo CArray, m_signal_r, para guardar el valor de la referencia en centímetros. Para convertir de centímetros a voltios se usa la función **CentsToVolts**.

Así pues, dentro de GenerarReferencia, lo que se hace es dar valores a los elementos de m_signal_r, y, a la vez, ir dando valores a los elementos de m_signal_rvo usando la función CentsToVolts con los elementos de m_signal_r como argumentos de entrada a esta función.

Por último, OnInitialUpdate dibuja la ventana gráfica donde se representarán los datos llamando a la función **OnDraw** , crea el temporizador que medirá el intervalo de muestreo y termina.

Tras terminar OnInitialUpdate, comienza el experimento. El temporizador, que corre en su propio hilo de proceso, se encarga de detectar cuando se cumple el intervalo

de muestreo. Cada vez que vence el temporizador, se llama a la función **NuestroProc**, que es la función encargada de leer la nueva muestra, calcular la señal de control a aplicar, y sacar la señal de control por el canal de salida de la tarjeta.

La función NuestroProc no pertenece a ninguna clase. Se comunica con las clases por medio de mensajes. Cuando la función NuestroProc termina su cometido, envía un mensaje llamado WM_MY_TIMER que avisa a la clase vista de que tiene que pintar un nuevo punto. Al recibir la clase vista el mensaje WM_MY_TIMER (que está definido en el fichero mypint.h) se ejecuta la función **OnMyTimer** que dibuja las nuevas muestras en la pantalla.

Al llegar al tiempo final del experimento, se destruye el temporizador (en la función NuestroProc), y se llama a OnDraw para que repinte las señales con una línea continua, y también desde NuestroProc se envía un mensaje WM_GUARDAR a la clase vista, para que guarde los datos en fichero.

Cuando la clase vista recibe el mensaje WM_GUARDAR, se ejecuta la función **OnGuardar**, que escribe los datos del experimento en el fichero previamente especificado.

Los datos que escribe son el tiempo de muestreo, el tiempo final del experimento, la señal de control ‘u’ aplicada, la señal de referencia ‘r’ tanto en voltios como en centímetros, la señal de salida ‘y’ antes y después de ser filtrada, y la señal de error ‘e’ diferencia de ‘r’ menos ‘y’.

En respuesta a cada opción del menú se ejecuta una función. Así, para la opción NuevoExperimento, se ejecuta la función de la clase vista **OnNuevoExperimento**, que se encarga de preparar un nuevo experimento (mostrando los cuadros de diálogo necesarios) y comenzarlo (creando el temporizador).

En respuesta a la selección de Cerrar Tarjeta en el menú, se ejecuta la función **OnCerrar**, que cierra la tarjeta comprobando si hubo fallo, y termina la aplicación.

En respuesta a la opción Repintar Todo se ejecuta la función **OnRepintarTodo**, que redibuja la gráfica de la ventana de la aplicación, pintando las señales con línea continua.

En respuesta a la opción Tiempos del menú, se ejecuta la función **OnChequeo**, y en respuesta a la opción Constantes del menú se ejecuta la función OnChequeoPid.

Ya se comentó lo que hacen estas funciones cuando se explicó que ocurría cuando se seleccionaba cada una de las órdenes del menú.

De todas manera, el código fuente contiene gran cantidad de comentarios con el fin de facilitar la comprensión del funcionamiento de la aplicación, y conviene consultarlo cuando se tenga alguna duda, junto con la ayuda en línea de Visual C++.

4.6 MODIFICACIÓN DE LA APLICACIÓN

En este apartado se explica cómo cargar los ficheros de la aplicación en el entorno de desarrollo de **Microsoft Developer Studio** para realizar modificaciones sobre el programa.

Asimismo se indica qué partes específicas del programa son las que hay que modificar para cambiar algunas de las características de la aplicación, como la duración máxima de los experimentos, o el color de las señales en la pantalla.

Para cargar un proyecto ya existente, hay que abrir el programa **Microsoft Visual C++**. Una vez abierto, en el menú **File** hay que seleccionar la opción **Open Workspace**. Entonces aparecerá un cuadro de diálogo para que se seleccione el fichero de proyecto a cargar, es decir, el fichero del proyecto con extensión **.clw**. El fichero de proyecto de la aplicación del levitador neumático es **mypint.clw**, por tanto éste es el fichero que se debe seleccionar.

Una vez cargado, se pueden distinguir tres ventanas en la pantalla.

A la izquierda está la ventana de proyecto (**Workspace**) donde aparece información sobre el proyecto en general. Según la pestaña que se tenga seleccionada de las cuatro que tiene, se puede ver en estructura de árbol las clases existentes en la aplicación, los ficheros de la aplicación, los recursos de la aplicación, o los temas de ayuda en línea. Mediante está ventana se puede encontrar y seleccionar rápidamente el elemento que se esté buscando.

Ocupando el centro y la derecha de la pantalla está la ventana de ficheros. Si se selecciona un fichero en la ventana de proyecto se abrirá una ventana de fichero donde aparecerá el código de dicho fichero. Modificando el código que aparece en esta ventana se modifica el código del fichero correspondiente.

Por último, en la parte inferior de la pantalla está la ventana de depuración en la cual aparecen los mensajes de aviso y error cuando se compila y/o se enlaza algún fichero del proyecto. Para compilar y ejecutar un proyecto hay que seleccionar la orden **Execute** del menú **Build**, o activarla pulsando Ctrl+F5.

Existen otras ventanas que también tienen funciones específicas, pero que no se usan habitualmente.

En general, el uso del entorno de programación es muy intuitivo y sencillo, y resulta fácil de manejar con un poco de práctica.

A continuación se comenta que se debe modificar en el programa para cambiar algunas características del programa.

Para variar los límites de las variables tiempo de muestreo y tiempo máximo del experimento, se puede actuar de dos maneras, cambiando directamente el trozo de código donde se definen los valores extremos de ambas variables, o usando el asistente del entorno. Por simplicidad, se explica cómo hacerlo usando el asistente.

Primero se invoca al asistente pulsando Ctrl+W. A continuación se selecciona la pestaña **Member Variables** en la ventana que ha aparecido al llamar al asistente. En el campo **Class Name** se debe elegir la clase **DLimites**, que es la clase que se encarga de los datos generales del experimento.

Al seleccionar la clase DLimites aparecerán en el campo **Control ID´s** varios identificadores. El identificador relativo al tiempo de muestreo es **IDC_TIEMPO_MUES**. Si se selecciona esta identificador en la parte inferior se verán dos campos, con nombre **Minimun Value** y **Maximun Value** que indican, respectivamente, el valor mínimo y máximo que puede adoptar el tiempo de muestreo.

Para cambiar estos valores no hay más que escribir los nuevos valores en los campos anteriores y pulsar el botón de Aceptar.

Para variar el rango permitido para el tiempo máximo del experimento hay que seguir los mismos pasos, con la diferencia de que en este caso el nombre del identificador es **IDC_TIEMPO_MAXIMO**.

Es importante recordar que el tiempo máximo que puede durar un experimento está limitado por cuestiones relativas a la representación de los datos en pantalla.

La pantalla posee un número de unidades físicas, o píxeles, y la aplicación divide la pantalla en unidades lógicas a la hora de representar los datos. La relación entre el número de unidades lógicas y el número de unidades físicas está prefijada por dos “defines” llamados **RES_X** y **RES_Y**.

El valor de duración del experimento multiplicado por el valor de RES_X no debe sobrepasar las 25.000 unidades, o la gráfica de la aplicación se estropeará por problemas de desbordamiento de las variables. Esta limitación viene impuesta por la propia implementación de las funciones gráficas de las MFC de Visual C++, y no puede evitarse. Igualmente el valor de RES_Y por el máximo valor de voltaje (definido como VAL_MAXIMO – VAL_MINIMO) tampoco debe superar las 25.00 unidades.

Los valores de RES_X y de RES_Y se definen (mediante la orden “define” de C) en el fichero **mypintview.h** y son, respectivamente, 10 y 1000. Para alterarlos simplemente basta con cambiar el valor de definición en la línea de “define” correspondiente.

Para cambiar el valor máximo y el valor mínimo de voltaje que se representa en el eje Y en la gráfica de pantalla, sólo hay que cambiar los valores en los “defines” de **VAL_MAXIMO** y **VAL_MINIMO** respectivamente, en el fichero mypintview.h. Actualmente valen cinco voltios y cero voltios.

En este mismo fichero también están las definiciones de los colores usados para representar las señales. Son respectivamente, **COLOR_EJES**, **COLOR_LEYENDA**, **COLOR_R**, **COLOR_U**, **COLOR_YF**, **COLOR_E** y **COLOR_BK** para el color de los ejes, el de la leyenda de y las escalas de la gráfica, el de la señal de referencia, el de la señal de control, el de la señal de altura (ya filtrada), el de la señal de error y el color de fondo.

Asimismo, el número de divisiones de la gráfica también se puede variar cambiando el valor del "define" de **N_EJES**, que ahora vale diez.

La señal de referencia 'r' se puede cambiar modificando la función **GenerarReferencia**, que es una función miembro de la clase vista. En esta función es donde se le dan valores a los elementos de **m_signal_rvo**, que es el CArray miembro de la clase vista que se utiliza como señal de referencia. Un CArray es una clase de las MFC equivalente a una tabla en C, con la diferencia de que puede cambiar su tamaño dinámicamente.

La aplicación del levitador neumático usa varios CArray, uno para cada una de las señales: m_signal_u para la señal de control, m_signal_r para la de referencia, m_signal_rvo para la señal de referencia convertida a voltios, m_signal_y y m_signal_yf para la señal de salida antes y después de ser filtrada, y m_signal_e para la señal de error.

La ley de control está definida en la función **NuestroProc**, dentro del bucle relativo al modo de control. Según el valor de la variable **modo** la ley de control será predefinido (modo == 0) o PID (modo == 1). Como ya se comentó, las otras dos opciones que existían (PID por zonas y Predictivo) se eliminaron tras terminar los experimentos de este proyecto. Lo que hacen todas las leyes de control es darle valor a la variable **valor_volts** que será el valor que se saque por el conversor D/A de la tarjeta.

La función NuestroProc es una función que no pertenece a ninguna clase de las existentes en la aplicación, sino que es global. Es necesario hacerlo así para poder usar

el temporizador multimedia, que es el único que garantiza suficiente precisión en la medida del tiempo de muestreo.

Estas funciones que no pertenecen a ninguna clase (funciones call-back) guardan un cierto parecido con las excepciones o las interrupciones, puesto que mientras dura la ejecución de una función call-back el sistema operativo está limitado y la mayoría de las órdenes no se pueden usar. Esto hay que tenerlo en cuenta cuando se implementa una función de este tipo, pues la duración de su ejecución no debe ser elevada, ya que es posible que el sistema se colapse y el ordenador se “cuelgue”.