Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO FINAL DE CARRERA

TÍTULO DEL TFC: Net Zeppelin: dirigible con C# y estación de tierra

TITULACIÓN: Ingeniería Técnica de Telecomunicación, especialidad Sistemas de Telecomunicación

AUTOR: Rubén Gómez Pérez

DIRECTOR: Juan López Rubio

FECHA: 18 de julio de 2011

Título: Net Zeppelin: dirigible con C# y estación de tierra

Autor: Rubén Gómez Pérez

Director: Juan López Rubio

Fecha: 18 de julio de 2011

Resumen

Hoy en día es habitual encontrar en Internet diferentes páginas con tutoriales sobre cómo llevar a cabo proyectos aparentemente fáciles que impliquen electrónica y/o programación, como por ejemplo el dirigible BlimpDuino, que se puede comprar online y solo necesita el montaje y la subida del código a la placa para que funcione. Lo bueno de estos proyectos es que muchas veces se pueden ampliar o modificar para cambiar o mejorar su funcionamiento, y esa es la idea principal de este trabajo.

En este proyecto se ha utilizado el dirigible BlimpDuino, controlado mediante una placa de Arduino, y se ha realizado un cambio en el sistema de funcionamiento. Primero se le ha cambiado la placa base de manera que el dirigible pueda funcionar mediante un programa escrito en C# y que utilice .NET Micro Framework. Después se ha buscado la incorporación de diferentes sensores para añadirlos al dirigible y se ha cambiado el sistema de comunicación para el control remoto, haciendo que ésta se establezca mediante un módulo XBee. Por último, aprovechando ese cambio en las comunicaciones, se ha diseñado una estación base donde se puedan ver los principales datos sobre el funcionamiento del dirigible y se ha modificado el programa para que el control manual se lleve a cabo mediante un joystick conectado al ordenador de la estación de tierra.

El grupo Icarus, de la Escuela de Ingeniería de Telecomunicación y Aeroespacial de Castelldefels, utiliza C# para el control de vehículos aéreos no tripulados (en inglés, Unmanned Aerial Vehicles, UAVs), por lo que este proyecto podría ser útil para el futuro diseño e implementación de un piloto automático sencillo basado en C# que se pueda usar para docencia y como ejemplo de uso de .NET Micro Framework (C# sin sistema operativo). Todo esto además del renovado sistema de control manual y la nueva capacidad para sensores, fruto del presente trabajo de fin de carrera.

Title: Net Zeppelin: C# controlled blimp with groundstation

Author: Rubén Gómez Pérez

Director: Juan López Rubio

Date: July, 18th 2011

Overview

Nowadays it is very common to find in the Internet many different pages with tutorials about how to make apparently easy making projects involving electronics and/or programming, such as the BlimpDuino zeppelin, which can be purchased online and only needs the user to build the kit and upload the code to the board to work. The good thing about these projects is that very often they can be expanded or modified to change or improve their performance, and that is the main idea of this project.

In this project, the Arduino controlled zeppelin BlimpDuino has been changed its functioning system. First, its board has been replaced by another one, so the zeppelin can be run under a C# .NET Micro Framework program. After that, the addition of different sensors to the zeppelin has been studied and the remote control communication system has been changed, making it work with an XBee module. Finally, taking advantage of this change in the remote control communications, a ground station has been designed in which the principal performance data of the zeppelin can be seen, and the program has been modified so the manual control of the zeppelin can be carried out using a joystick connected to the computer where the ground station is.

The Icarus group, from the School of Telecommunications and Aerospace Engineering of Castelldefels (EETAC), uses C# to control Unmanned Aerial Vehicles (UAVs), so this project could be useful for the future design and implementation of a simple autopilot program based on C# that can be used for teaching purpose and as an example of the use of .NET Micro Framework (C# without operating system). All this in addition of the new manual control system and the new sensor capabilities, result of this final career project.

ÍNDICE

INTRODUCCIÓN 1

1. HARDWARE UTILIZADO EN EL PROYECTO 3
1.1. El precedente: BlimpDuino 3
1.1.1. Estructura de la placa, del código y funcionamiento 4
1.2. Placa de desarrollo FEZ Mini 6
1.3. Módulo de comunicaciones XBee Pro 6
1.4. Servo 8
1.5. Joystick SideWinder Precision 2 8
1.6. Sensor de ultrasonidos 9
1.7. Acelerómetro 10
1.8. Driver de potencia 11
1.9. Conclusiones del capítulo 12

2. SOFTWARE UTILIZADO EN EL PROYECTO 13
2.1. Microsoft Expression Studio 13
2.1.1. Microsoft Expression Blend 14
2.1.2. Microsoft Expression Design 17
2.2. Microsoft Visual Studio 18
2.3. Conclusiones del capítulo 19

3. PRIMERAS PRUEBAS DE COMPONENTES 21
3.1. La placa Embedded Master 21
3.2. Prueba del XBee 22
3.3. Prueba del acelerómetro 23
3.4. Prueba del servo 25
3.5. Conexión del sensor de ultrasonidos 26
3.6. Conclusiones del capítulo 27

4. MONTAJE DE LA PLACA FINAL 29
4.1. Funcionamiento y conexiones de la nueva placa 29
4.2. Pruebas de hardware en la placa FEZ Mini 32
4.2.1. Prueba del XBee 33
4.2.2. Prueba del acelerómetro 34
4.2.3. Prueba del servo 35
4.2.4. Prueba del sensor de ultrasonidos 35

4.2.5. Prueba del driver 36
4.3. Net Zeppelin: programa final 37
4.3.1. Clases del Net Zeppelin 38
4.4. Diseño de la placa de topos 39
4.5. Conclusiones del capítulo 41

5. CREACIÓN DE LA ESTACIÓN BASE 43
5.1. Diseño de la estación base 43
5.2. Programando la estación 44
5.2.1. Estación base en Expression Blend 44
5.2.2. Estación base en Visual Studio 47
5.2.3. Incluyendo el joystick en la estación 49
5.3. Conclusiones del capítulo 53

6. ESTUDIO DE AMBIENTALIZACIÓN Y CONCLUSIONES 55
6.1. Estudio de la ambientalización 55
6.2. Conclusiones del proyecto 57

BIBLIOGRAFÍA 59

ANEXOS 61

INTRODUCCIÓN

Hoy en día es habitual encontrar en Internet diferentes páginas con tutoriales sobre cómo llevar a cabo proyectos aparentemente fáciles que impliquen electrónica y/o programación, como por ejemplo el dirigible BlimpDuino, que se puede comprar online y solo necesita el montaje y la subida del código a la placa para que funcione. Lo bueno de estos proyectos es que muchas veces se pueden ampliar o modificar para cambiar o mejorar su funcionamiento. Utilizar este tipo de proyectos para docencia es una buena idea, ya que al ser trabajos que suelen contar con explicaciones para su desarrollo permiten a los estudiantes aprender por ellos mismos y les muestra hasta donde pueden llegar.

En este proyecto se ha utilizado el antes mencionado BlimpDuino, controlado mediante una placa de Arduino, y se ha realizado un cambio en el sistema de funcionamiento. Primero se le ha cambiado la placa base de manera que el dirigible pueda funcionar mediante un programa escrito en C# y que utilice .NET Microframework. Después se ha buscado la incorporación de diferentes sensores para añadirlos al dirigible y se ha cambiado el sistema de comunicación para el control remoto, haciendo que ésta se establezca mediante un módulo XBee. Por último, aprovechando ese cambio en las comunicaciones, se ha diseñado una estación base donde se puedan ver los principales datos sobre el funcionamiento del dirigible y se ha modificado el programa para que el control manual se lleve a cabo mediante un joystick conectado al ordenador de la estación de tierra.

De esta manera, se ha pasado de un proyecto prediseñado donde sólo hacía falta construir la estructura del zeppelín a uno donde hay que diseñar la distribución de los elementos y en el que hace falta programar tanto el dirigible como una aplicación independiente que se comunique con el primero.

Además de todo lo mencionado anteriormente, este proyecto sería útil para introducir a los alumnos en el ámbito de los Vehículos Aéreos no Tripulados, en inglés UAV (Unmanned Aerial Vehicle). El grupo Icarus, de la Escuela de Ingeniería de Telecomunicación y Aeroespacial de Castelldefels, utiliza C# para el control de UAVs, por lo que el desarrollo de trabajos como este puede aumentar el interés de los alumnos por este tema y por las actividades de Icarus. Por ejemplo, este proyecto podría ser útil para el futuro diseño e implementación de un piloto automático sencillo basado en C# que se pueda usar para docencia, como ejemplo práctico de uso de .NET Micro Framework (C# sin sistema operativo) y para cualquier tipo de tarea que requiera un UAV para poder llevarse a cabo.

La estructura de esta memoria responde al orden seguido durante la realización del proyecto, y se divide en 6 capítulos. Para empezar, en los dos primeros capítulos se podrá encontrar información básica sobre los componentes y programas utilizados durante el proyecto (incluida una explicación un poco más extensa sobre el BlimpDuino), así como justificaciones

para su uso. De esta manera se asientan unos conocimientos básicos sobre el funcionamiento de las herramientas de trabajo para proceder mejor.

En el capítulo 3 se explican en detalle las diferentes pruebas realizadas antes de comenzar con la redacción del programa principal del dirigible, basadas en utilizar los componentes de uno en uno para comprobar su buen funcionamiento, tener una primera toma de contacto práctica y escoger según qué parámetros en el hardware que permita diferentes configuraciones. Estas pruebas se llevaron a cabo utilizando una placa diferente a la final, llamada Embedded Master, que también utiliza .NET Micro Framework.

Tras estos test del equipo, comienza el desarrollo de una de las grandes mitades que forman el proyecto. Tal y como se explica en el capítulo 4, se escribe el código del dirigible para la placa FEZ Mini, después de realizar de nuevo algunas pruebas para comprobar si hay cambios entre la versión de .NET Micro Framework utilizada anteriormente (versión 3.3) y la actual (versión 4.1). Hacia el final del capítulo se explica el proceso de estructuración y montaje de la placa de topos utilizada para el ensamblaje de todas las piezas del zeppelín.

La última parte del proyecto, detallada en el capítulo 5 de este documento, consiste en la creación de una aplicación de escritorio, una estación de tierra donde poder visualizar datos extraídos por el dirigible gracias a los sensores que lleva incorporados. En este capítulo también se explican diferentes conceptos y métodos utilizados en la elaboración de la estación, así como diferentes detalles acerca del funcionamiento del joystick utilizado para el control del dirigible.

Por último, el capítulo 6 contiene un rápido estudio sobre la ambientalización del proyecto y las conclusiones extraídas a lo largo del mismo, además de un repaso de los objetivos conseguidos, como se podrían mejorar con más tiempo y financiación y posibles nuevas metas y mejoras de cara a futuras revisiones del proyecto.

1. HARDWARE UTILIZADO EN EL PROYECTO

El presente trabajo consta de dos objetivos principales, uno de los cuales es realizar un cambio en el hardware del dirigible para permitir la incorporación de nuevos sensores con los que antes no contaba y cambiar el sistema de comunicaciones utilizado anteriormente. En este primer capítulo se explicarán los principales detalles sobre el material utilizado para la realización de dicha parte del proyecto.

1.1. El precedente: BlimpDuino

Si bien no se trata de un sensor o una pieza en concreto, el BlimpDuino es el primer elemento que hará falta para poder llevar el proyecto a buen puerto, ya que hará falta comprender cómo funciona y qué componentes utiliza para poder reescribir ese código en C# y conseguir que el nuevo dirigible funcione como el original. Por ello, tras una breve explicación general del proyecto BlimpDuino, hay un apartado dedicado a explicar el funcionamiento del dirigible y la estructura tanto de la placa como del código.

BlimpDuino es un proyecto “open source” desarrollado gracias a la colaboración entre Chris Anderson (editor jefe de la revista Wired y fundador de la web DIY Drones) y Jordi Muñoz (integrante de DIY Drones) que consiste en un dirigible que, mediante una placa de Arduino, puede ser pilotado manualmente.

A través de la web DIY Drones se puede comprar un kit de montaje de BlimpDuino, que incluye todas las piezas necesarias para construir el dirigible, incluyendo la placa, el servo o un mando de radiocontrol con su respectivo receptor entre otras. El código necesario para hacer funcionar el zeppelín se encuentra alojado en Google Code, al alcance de cualquiera que esté interesado en el proyecto.

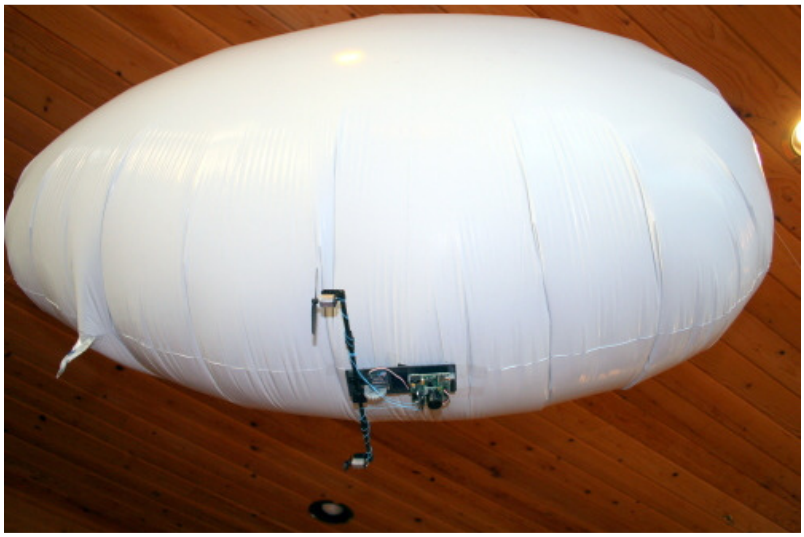

Fig. 1.1 BlimpDuino ya montado, en una imagen de la web DIY Drones

El punto fuerte del proyecto BlimpDuino es que cualquiera puede tener su propio dirigible en casa. El montaje de la góndola del dirigible está explicado paso a paso en el blog de Chris Anderson, y el código creado por Jordi Muñoz sólo necesita ser subido a la placa mediante el programa Arduino. Aparte de comprar el kit, montarlo y subirle el programa de funcionamiento, el usuario sólo debe asegurarse de poder llenar el globo aerostático de helio.

Además, puesto que el proyecto es open source, el usuario de BlimpDuino puede modificarlo con total libertad, añadiéndole nuevas funcionalidades o mejorando el código ya existente. Gracias a este aspecto se ha podido llevar a cabo el presente trabajo, y constituye la principal idea del mismo: cambiar la estructura del BlimpDuino y mejorar sus funcionalidades.

1.1.1. Estructura de la placa, del código y funcionamiento

La placa de Arduino que utiliza el BlimpDuino es una placa diseñada para ser usada únicamente en dicho proyecto. Este hecho significa dos cosas: por un lado, puesto que los componentes a utilizar serán siempre los mismos, el tamaño de la placa es el mínimo posible, lo cual evita posibles problemas en cuanto al peso que el globo del dirigible puede soportar; por el otro lado, la placa no se puede ampliar añadiéndole nuevos módulos. Tan sólo sería posible incorporar un módulo XBee. La siguiente figura muestra de manera esquemática la disposición de los elementos que conforman la placa del BlimpDuino:

Fig. 1.2 Esquema de bloques de la placa de Arduino utilizada en BlimpDuino

El código del BlimpDuino está, al igual que el del programa de este proyecto, estructurado en “clases” (el entrecomillado se debe a que en un proyecto de Arduino, puede haber diferentes ficheros .pde, pero no son clases propiamente dichas como las de C#), habiendo una por cada elemento clave en su funcionamiento. En el siguiente esquema se muestran las diferentes clases:

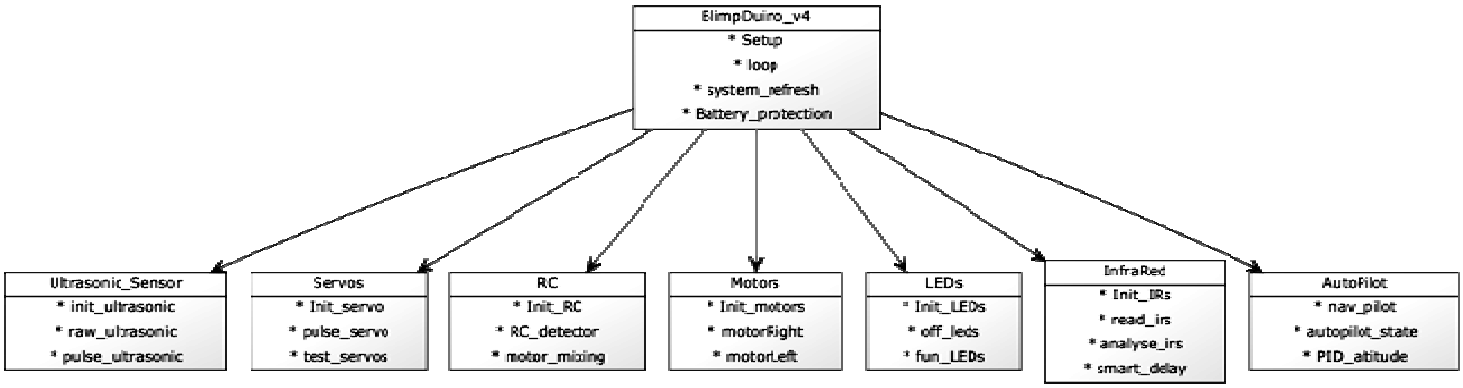

Fig. 1.3 Diagrama de clases en UML del proyecto BlimpDuino

Como se puede observar en la figura, nos encontramos con un fichero para los motores, otro para el sensor de ultrasonidos, y así cubriendo las diferentes funcionalidades del dirigible. A parte de todos estos, hay un fichero llamado BlimpDuino_v4.pde que funciona como clase principal donde establecer el bucle que irá realizando las llamadas al sensor de ultrasonidos y se encargará de hacer funcionar tanto los motores como el servo.

La diferencia respecto a C# está en que no hace falta crear objetos del tipo deseado para acceder a las funciones de cada fichero .pde. Así, cuando el BlimpDuino tiene que ejecutar, por ejemplo, la función “motorRight” que sirve para encender el motor derecho y que está incluida en el fichero Motors.pde, no hace falta crear un objeto de tipo Motors en el archivo principal y entonces hacer una llamada parecida a Motors.motorRight. Con sólo escribir el nombre de la función (sea esta o cualquier otra que haya en otro fichero del proyecto) y pasarle los parámetros pertinentes si es que necesita alguno, ya se puede ejecutar la orden.

Esta estructuración del código facilita su escritura, ya que el usuario no tiene que preocuparse de crear suficientes objetos de los tipos que necesita y establecer el tipo de acceso para cada uno de ellos. Con ir escribiendo variables que pueda necesitar e ir haciendo las llamadas a las funciones puede escribir todo el programa. Sin embargo, este mismo aspecto puede llegar a ser un problema, y es que el código queda menos estructurado, y puesto que las llamadas son directas y todo puede ir escrito en cualquier fichero dentro de un mismo proyecto, el programa es más difícil de ser interpretado por alguien ajeno y con poca experiencia.

En cuanto al funcionamiento, hace falta destacar un pequeño detalle, y es que al contrario de lo que pueda parecer, el BlimpDuino es capaz de transmitir datos. Cuando se encuentra conectado al ordenador mediante el cable serie que sirve las veces para cargar el programa a la placa, utilizando el monitor serie del programa Arduino se pueden visualizar los datos que la placa envía al usuario, establecidos mediante código. Entre esos datos se encuentran la altura a la que vuela el dirigible o la cantidad de batería disponible. El problema está, precisamente, en que la única manera de saber estos datos (si no se le

añade un módulo XBee) es teniendo conectado el dirigible al ordenador, lo que obliga al usuario a establecer turnos entre el pilotaje del zeppelín y la observación de datos.

Tras estudiar con detenimiento el BlimpDuino y sus características, es fácil ver que se trata de un proyecto cerrado que apenas deja sitio a nuevas incorporaciones de hardware, pensado en primera instancia para ser montado y utilizado pero no mejorado. Es por esto que para poder llevar el zeppelín a otro nivel será necesario cambiar la placa con la que funciona.

1.2. Placa de desarrollo FEZ Mini

La pieza de hardware más importante en el cambio general que experimentará el dirigible es, sin duda, la nueva placa de desarrollo. En esta placa se van a conectar nuevos sensores cuya incorporación antes resultaba imposible, ya que, tal y como se ha comentado en el apartado anterior, la placa inicial con la que funciona el dirigible está diseñada expresamente para ser usada como base de funcionamiento del BlimpDuino, con las restricciones en cuanto a hardware que esto implica.

La principal diferencia (sin contar el hecho de que esta es una placa estándar sin una aplicación fija) respecto a la placa de Arduino es el hecho de que la placa FEZ Mini utiliza .NET Micro Framework para sus programas, permitiendo de esta manera la ejecución de código escrito en C# sin necesidad de sistema operativo. Como la cantidad de pines disponibles aumenta con este cambio de placa, no hay problema para mantener los sensores ya presentes antes (como, por ejemplo, el sensor de ultrasonidos, que permite saber la altura a la que vuela el dirigible), sino que, además, se pueden añadir otros que antes no había, como un acelerómetro.

Básicamente, hay dos razones por las que se utiliza la placa FEZ Mini en este proyecto. La principal es el cambio en el lenguaje utilizado, ya que C# ofrece muchas más posibilidades a la hora de programar y cuenta con librerías muy útiles para el proyecto, como es la librería que permite leer los datos de un joystick o las propias librerías para C# de las placas de GHI Electronics, que facilitan mucho la identificación de puertos y aportan clases específicas relativas a algunas funciones de estas placas. La otra razón tiene que ver con las propias limitaciones del dirigible. Si bien hay placas que disponen de más pines que la FEZ Mini, el reducido tamaño de ésta la hace idónea para ser utilizada en el zeppelín, aunque sólo se le pueda añadir 1 ó 2 sensores más.

1.3. Módulo de comunicaciones XBee Pro

El cambio de la placa que hará funcionar el zeppelín acarrea otro cambio importante: el tipo de comunicación utilizada por este. Esto no solo afecta al modo de movimiento por radiocontrol, sino que ahora también servirá para

poder visualizar datos en una estación de tierra. La comunicación se llevará a cabo mediante el protocolo UART utilizando un módulo XBee, concretamente el modelo XBee Pro.

El XBee Pro permite la comunicación a distancias de hasta 100 metros en interiores con un bajo consumo de energía, por lo que es perfecto para utilizarlo en la conexión del dirigible con la estación de tierra.

Además, también se dispone de un módulo XBee Pro PKG-U, que se diferencia del XBee Pro habitual en que este se puede conectar directamente al ordenador mediante un cable USB y en que cuenta con un encapsulado más grande y robusto con una antena mayor que la del XBee Pro. El módulo PKG-U es perfecto para su uso con la estación de tierra, ya que al poder ser conectado directamente al ordenador no hará falta un adaptador para conectar un XBee Pro convencional.

Fig. 1.4 Módulos XBee Pro (izquierda) y XBee Pro PKG-U (derecha)

Estos módulos XBee ya estaban disponibles desde el inicio del proyecto, y tanto su fácil uso como el buen rendimiento que ofrecen son motivos más que suficientes para utilizarlos en el trabajo. Los módulos XBee no precisan de ninguna configuración previa para su uso, sólo es necesario especificar los diferentes parámetros del puerto serie en el programa que se va a ejecutar para que coincidan con los predeterminados en el PKG-U. Además, a pesar de que el XBee Pro cuenta con 20 pines diferentes, únicamente 4 de ellos (alimentación, transmisión, recepción y masa) son necesarios para poder establecer la comunicación inalámbrica.

1.4. Servo

Para que el dirigible pueda moverse tanto horizontal como verticalmente es necesario que pueda posicionar sus hélices de la manera apropiada, y esto se consigue mediante el servo, pieza indispensable para el funcionamiento del zeppelín que permanecerá en la nueva placa por motivos obvios.

El servo utilizado en el dirigible es el modelo TGY-1440A de la marca Turnigy. Se trata de un servo económico pero con una buena precisión. Este servo realiza un giro de 60° en 0.12 segundos y su torsión es de 0.6 kg/cm (es decir, puede levantar hasta 0.6 kg que se encuentren a una distancia de 1 cm de su eje), por lo que es suficiente para poder redireccionar los motores del zeppelín.

Este es el servo con el que funcionaba el dirigible desde el principio. Puesto que abarca todos los ángulos posibles desde 0° hasta 180° (rango necesario para poder mover el zeppelín tanto adelante y atrás en el eje horizontal como arriba y abajo en el vertical) con una buena velocidad de giro, por lo que no es necesario un cambio en esta pieza de cara al nuevo diseño del zeppelín.

1.5. Joystick SideWinder Precision 2

Cuando se adquiere el dirigible BlimpDuino, el kit incluye un receptor de radiocontrol y un mando, concretamente los modelos HK-TR6A y HK-T4A, respectivamente, ambos pertenecientes a la marca HobbyKing. Este mando opera a 2,4 GHz y cuenta con 4 canales de transmisión diferentes. El receptor tiene hasta 6 entradas diferentes, pero sólo se utilizan 3 (2 para las ordenes enviadas por el mando radiocontrol y una para el control del servo).

Sin embargo, tras la inclusión en la nueva placa de los módulos XBee, ya no es preciso el uso del mando y el receptor radiocontrol. Si tenemos en cuenta la conexión de los XBee (uno en el dirigible y otro en un ordenador para mostrar datos en la estación base), se podrían utilizar para enviar las ordenes de movimiento tanto de los motores como del servo a través de la estación de tierra. Para ello se utilizará un joystick.

El joystick que permitirá el control manual del dirigible es el modelo SideWinder Precision 2 de Microsoft. Este joystick se mueve en 3 ejes (horizontal, vertical y rotación) y cuenta con 8 botones y una pequeña palanca. Los valores a la salida del stick principal (tanto para el movimiento horizontal como para el vertical y el de rotación) y de la palanca van de 0 a 65535. Estos valores deberán ser tomados en cuenta para que se correspondan en todo momento con las posiciones que pueda tomar el servo.

Para poder utilizar el joystick en la estación de tierra ha sido necesario incluir ciertas DLL de la librería de DirectX, además de una clase con todas las funciones necesarias escrita por Mark Harris.

El uso de este joystick se debe ante todo a su disponibilidad, ya que el departamento de arquitectura de computadores ya contaba con él. Sin embargo, también disponía de otro sistema de control, el X52 Pro de Saitek. Este joystick cuenta con muchos más botones y tiene otro stick aparte del principal que se utiliza para el acelerador. Y esa es precisamente la razón de utilizar el SideWinder y no el X52 Pro.

Fig. 1.5 Comparativa de los dos joysticks y el total de características que se quedarían sin usar de cada uno

La falta de necesidad de tantos botones para manejar el dirigible es el porqué a la hora de decantarse por uno de los dos. El zeppelín no necesita un joystick con demasiadas prestaciones para poder ser controlado: sólo hace falta poder acelerar e ir marcha atrás (eje vertical del joystick), girar a izquierda y derecha (eje horizontal del joystick) y mover el servo arriba y abajo (palanca del joystick).

Por eso, utilizar un joystick con muchas prestaciones sólo se traducirá en desaprovechar la mayoría, así que el elegido es el SideWinder por la simple razón de no tener tantas funciones.

1.6. Sensor de ultrasonidos

La placa del BlimpDuino contaba con un sensor de ultrasonidos para obtener la altitud a la que volaba el dirigible. Puesto que el objetivo del trabajo no es capar las funcionalidades ya existentes en el dirigible sino aumentarlas, este sensor también estará conectado a la placa FEZ Mini.

El sensor de ultrasonidos utilizado es el LV-MaxSonar-EZ4 de la marca MaxBotix. Este sensor cuenta con 3 puertos de salida: un puerto serie con formato RS232, una salida analógica con un factor de escala de $V_{CC}/512$ y otra salida que ofrece una representación de la distancia medida mediante un pulso. Cada vez que se enciende, el sensor utiliza un ciclo de lectura para la

calibración. Este proceso consiste en detectar un objeto que, para que resulte exitoso, no debe estar a menos de 35 centímetros del sensor.

Este componente viene de serie en el BlimpDuino. Su elección por parte de los desarrolladores del proyecto se debe a las características del haz del sensor. El EZ4 pertenece a una gama de sensores de ultrasonidos, los EZX, donde X es un número del 0 al 4. La diferencia entre cada uno de ellos es que a medida que el número es más grande, el haz que lanza el sensor para poder realizar las mediciones de distancia pertinentes es más estrecho, y por tanto, más preciso.

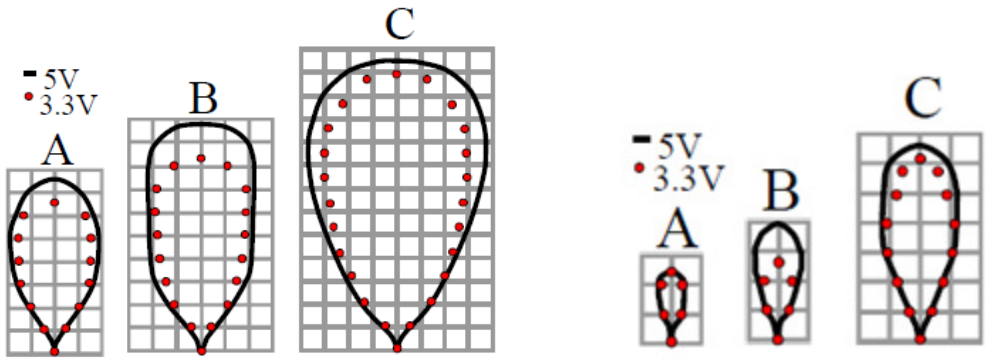

Fig. 1.6 Comparación de los haces del sensor EZ0 (izquierda) y EZ4 (derecha) en la medida de objetos a 6,35 mm (A), 2,54 cm (B) y 8,25 cm (C) en función de la alimentación administrada.

De ahí el uso del EZ4 en el dirigible: puesto que el sensor va a estar directamente enfocado al suelo y se pretende averiguar a qué altura se encuentra, un haz demasiado amplio puede llevar a rebotes en objetos cercanos que falsifiquen la medida. Pero con un haz lo más estrecho posible, es mucho más probable que la gran mayoría de la señal de ultrasonidos rebote en el objeto que esté justo debajo del dirigible y se halle la altura a la que vuela correctamente.

1.7. Acelerómetro

Ya se ha comentado anteriormente que el cambio de placa del zeppelín aumenta el número de pines disponibles para la conexión de otros módulos. La capacidad de la placa FEZ Mini ha permitido añadir un nuevo módulo (sin contar el XBee Pro), y el elegido ha sido el acelerómetro MMA7260Q de la marca Freescale Semiconductor.

El MMA7260Q tiene un consumo de corriente de 500 µA y cuenta con un modo de suspensión para cuando no hace falta que esté funcionando, por lo que no supone un gran gasto de la batería. Este acelerómetro, además, permite escoger entre 4 sensibilidades diferentes, por lo que sirve tanto para

aplicaciones que necesiten una precisión muy alta como para aplicaciones que puedan funcionar con una precisión baja.

La funcionalidad básica del sensor será informar al usuario en qué dirección se está moviendo el dirigible. Así, si se ve que los valores mostrados por el acelerómetro no se corresponden con la trayectoria descrita por el usuario mediante el joystick, se podrá saber que hay algún problema.

El uso de este acelerómetro viene condicionado, por una parte, por el hecho de que ya se contaba con él desde antes del comienzo de proyecto. Por otro lado, tras hacer las pruebas del sensor, los resultados de éstas demostraron que el funcionamiento del acelerómetro es bueno y el rango de valores en las salidas para cada eje es perfectamente adecuado para su uso en el dirigible. A todo esto se debe añadir la anteriormente mencionada posibilidad de selección de la sensibilidad, que permite afinar más las medidas realizadas por el sensor si en algún momento del vuelo es necesario.

1.8. Driver de potencia

Para poder hacer funcionar los motores hace falta suministrarles cierta potencia. Sin embargo, conectarlos directamente a salidas digitales de la placa FEZ Mini derivaría en que sólo podrían o funcionar a la máxima potencia (que tampoco sería muy grande debido al límite propio de la placa) o permanecer quietos. Es por esto que para poder activar los motores se utilizará el driver L293D de Texas Instruments.

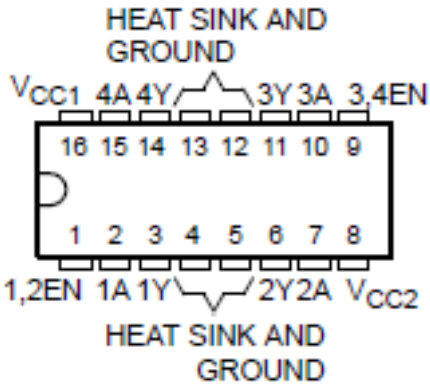

Fig. 1.7 Esquema de los pines del driver extraído del datasheet

Este componente cuenta con dos alimentaciones diferentes: una de ellas es para hacerlo funcionar, y la otra sirve para el suministro de los motores. Esta separación de alimentaciones está diseñada para minimizar la disipación de potencia en las entradas lógicas del driver.

La ventaja de utilizar un driver respecto a utilizar sólo entradas digitales es que, mediante el uso de pulsos a la entrada del componente, se pueden obtener salidas variables que dependerán del tiempo en estado alto del pulso. De esta

manera los motores podrán variar su velocidad de giro para permitir al usuario controlar mejor el dirigible.

A estos elementos hay que añadir un regulador, condensadores de desacoplo para los motores y resistencias utilizadas en un divisor de tensión para poder medir el nivel de batería (ya que el voltaje interno de la placa, 3,3 V, es menor que el de la batería, 7,4 V, y cualquier valor mayor que 3,3 sería interpretado por la placa como 3,3).

Tabla 1.1. Consumo de los módulos (voltaje de la batería: 7,4V; capacidad de la batería: 460 mAh)

Componente	Voltaje necesario	Corriente necesaria	Conexión
Driver L293D	VCC1: 4,5V – 7V VCC2: VCC1 – 36V	0,2 µA	VCC1: 3,3V (FEZ Mini) VCC2: 7,4V (Batería)
XBee Pro	2,8V – 3,4V	270 mA	3,3V (FEZ Mini)
Regulador 78L05	Para Vo = 3,3V – 9V V_i Max: 30V	Dato no disponible	7,4V (Batería)
Sensor de ultrasonidos MaxSonar EZ4	2,5V – 5,5V	Dato no disponible	3,3V (FEZ Mini)
Acelerómetro	2,2V – 3,6V	500 µA	3,3V (FEZ Mini)
FEZ Mini	5V – 9V	Dato no disponible	7,4V (Batería)

1.9. Conclusiones del capítulo

La mayoría de las piezas de hardware que se van a utilizar son sensores que ya vienen montados en un pequeño trozo de placa y con los pines necesarios para su funcionamiento diferenciados mediante zócalos. Esto agilizará bastante el montaje de la placa, ya que no habrá que emplear tiempo en prepararles un circuito acondicionador a cada uno; circuitos que por otra parte no harían sino aumentar la complejidad de la placa y precisar tiempo para distribuirlos por ella que se podría utilizar en otras tareas.

2. SOFTWARE UTILIZADO EN EL PROYECTO

Tal y como se ha comentado en el capítulo anterior, son dos los objetivos principales de este trabajo. El segundo de ellos es el de diseñar una estación de tierra que muestre al usuario toda una serie de datos recibidos desde el dirigible, tales como el nivel de batería o la aceleración que está experimentando y en qué ejes lo hace. Para la elaboración de esta estación base se han utilizado dos plataformas de desarrollo de Microsoft, Expression Studio y Visual Studio, que se explicarán de manera general en este capítulo.

2.1. Microsoft Expression Studio

Hoy en día, de cara a los usuarios es tan importante que un programa o aplicación funcione de manera fluida y sin errores como que su interfaz resulte atractiva y facilite la identificación de sus diferentes funciones y elementos interactivos. Hasta hace unos 5 ó 6 años la manera de crear estas interfaces gráficas pasaba por utilizar Visual Basic, donde las opciones de personalización se veían más o menos limitadas por las herramientas de la época. Sin embargo, en 2005 apareció la primera versión de Microsoft Expression Studio, el programa que ha permitido elaborar la estación base de este trabajo y que da al programador la posibilidad de personalizar todo el aspecto de su aplicación.

Microsoft Expression Studio es un paquete de programas orientadas al diseño de aplicaciones y páginas web. La versión utilizada es la 4, la más nueva. Algunos de los programas incluidos en Microsoft Expression Studio son:

- Microsoft Expression Blend, proporciona las herramientas necesarias para diseñar la parte gráfica de las aplicaciones que, más tarde, queramos programar en Visual Studio.

- Microsoft Expression Design, utilizado para la creación de gráficos e ilustraciones para su posterior uso en Expression Blend y en Expression Web.

- Microsoft Expression Web, mencionado antes, sirve para desarrollar fácilmente páginas web basadas en estándares compatibles.

- Microsoft Expression Encoder, que proporciona las herramientas necesarias para la edición de videos y su posterior publicación en páginas web.

Además, Microsoft Expression Blend cuenta con una herramienta extra muy útil, llamada SketchFlow. Este añadido permite el diseño de prototipos para aplicaciones utilizando diagramas de flujo que simulen el recorrido que el usuario hace a través de los diferentes estados posibles del programa.

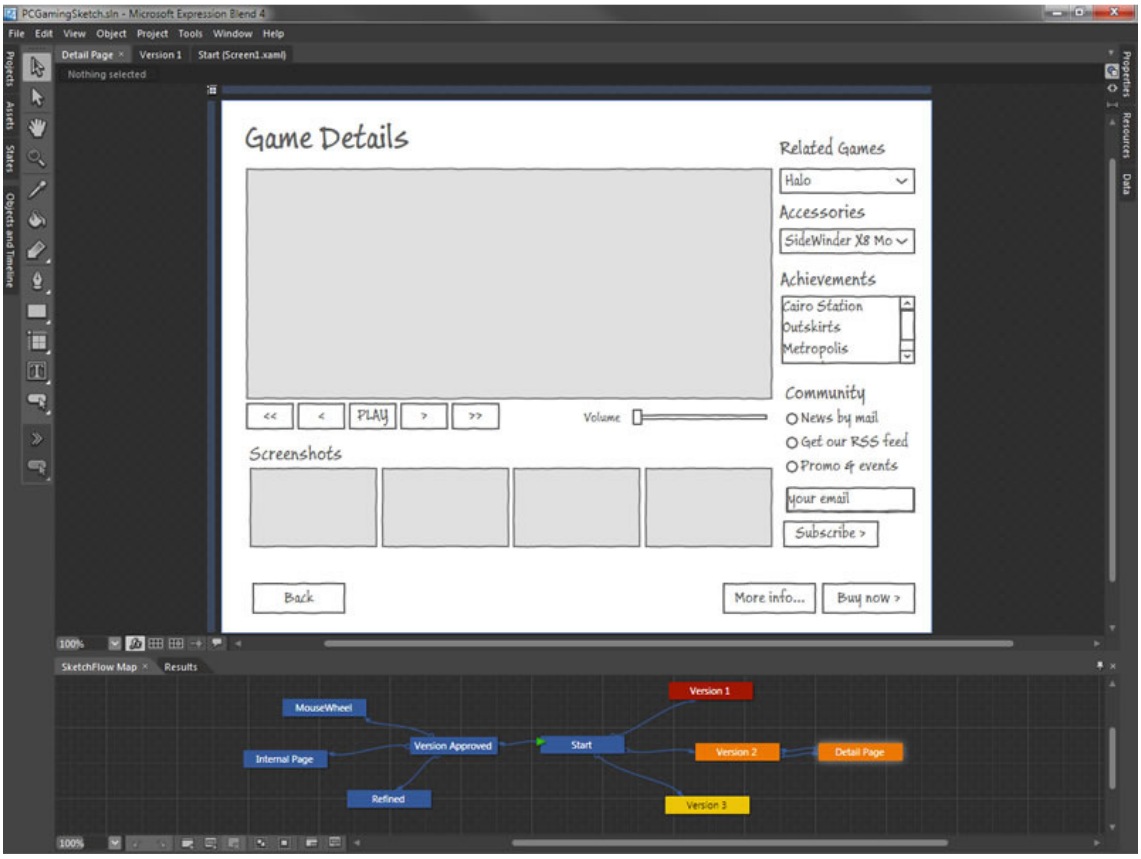

Fig. 2.1 Ejemplo de aplicación de la herramienta Sketchflow

Concretamente, para la estación base se han utilizado Expression Blend para la interfaz y Expression Design para la creación de alguna ilustración específica requerida por la estación. El uso de Expression Studio agiliza mucho la creación de la estación base y permite aplicar llamativos efectos visuales en poco tiempo y de manera sencilla y crear diseños vectoriales para su uso en el programa.

2.1.1. Microsoft Expression Blend

Microsoft Expression Blend ayuda a aunar dos aspectos aparentemente contrarios en el desarrollo de aplicaciones: diseño y código interno. Normalmente, en el ámbito profesional, es costumbre escuchar que en una empresa dedicada a la creación de aplicaciones hay un equipo que se encarga de la programación y otro que se encarga de la interfaz gráfica. También se puede escuchar que los programadores no confieren a sus aplicaciones diseños agradables o centrados en el usuario. Expression Blend proporciona al programador una manera fácil de implementar la parte gráfica de su aplicación y dejarla lista para la escritura del código.

2.1.1.1. Funcionamiento de Expression Blend

El funcionamiento de Expression Blend es muy sencillo, y recuerda mucho al funcionamiento de Visual Basic. Aquí, igual que en VB, lo único que necesitamos para diseñar la interfaz del programa es arrastrar a la ventana principal de éste los elementos que creamos oportunos. Así, si nuestra aplicación necesita un cuadro de texto y un botón, lo único que hay que hacer es buscarlos en la lista de elementos y arrastrarlos a la ventana principal o seleccionarlos en la lista y crearlos en la ventana con las medidas deseadas.

Aún así, si se genera un elemento con los valores predeterminados y se desea cambiarlos, en la parte derecha de la pantalla se sitúa la pestaña de propiedades del elemento (“Properties”), junto a las pestañas de recursos (“Resources”) y datos (“Data”).

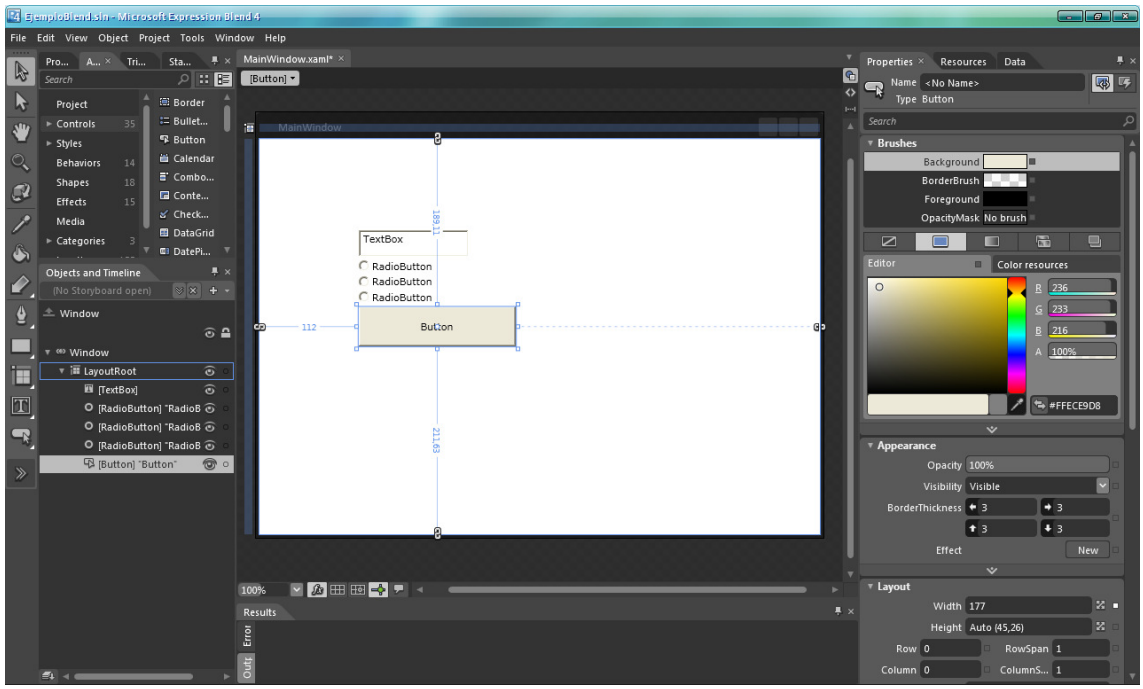

Fig. 2.2 Ventana de trabajo de Expression Blend en modo Diseño

Con echar un vistazo se puede apreciar el objetivo principal de Expression Blend. Éste es un programa orientado al diseño gráfico; el abanico de posibilidades que ofrece es muy amplio, y prácticamente todos los aspectos de los objetos mostrados por pantalla son editables, desde el color de relleno o del borde hasta la redondez de las esquinas. Es obvio que cada elemento puede tener ciertas propiedades exclusivas de su tipo, pero Expression Blend permite al diseñador escoger el estilo de hasta el más mínimo detalle de su aplicación. Además, se pueden combinar distintos objetos y aplicarles efectos de entre una lista ofrecida por el programa, a la vez que es posible guardar diferentes plantillas (“Templates”) para que un mismo elemento se vea diferente gráficamente según la situación lo requiera.

2.1.1.2. Expression Blend: el XAML

De momento, parece que Expression Blend es más o menos como un Visual Basic moderno. ¿Qué es lo que hace interesante a Expression Blend? La respuesta es: el XAML.

XAML responde a las siglas de “eXtensible Application Markup Language”, o “Lenguaje Extensible de Formato para Aplicaciones” en castellano. Como su parecida pronunciación sugiere, el XAML está basado en el XML, y es el lenguaje que utiliza Expression Blend para almacenar los objetos que hemos ido añadiendo a la ventana de nuestra aplicación.

En todo momento, en Expression Blend podemos echar un vistazo al fichero XAML donde se van haciendo las declaraciones de los elementos (denominados controles). En el código vemos que lo que se va escribiendo son las distintas propiedades de los controles, tales como el nombre, la alineación o la posición en pantalla, y estas pueden personalizarse tanto desde la barra de propiedades como desde este fichero XAML. Cuando se guarda el proyecto, se crea una solución que se puede abrir con Visual Studio 2010, donde podremos empezar a programar nuestra aplicación con toda la interfaz gráfica hecha.

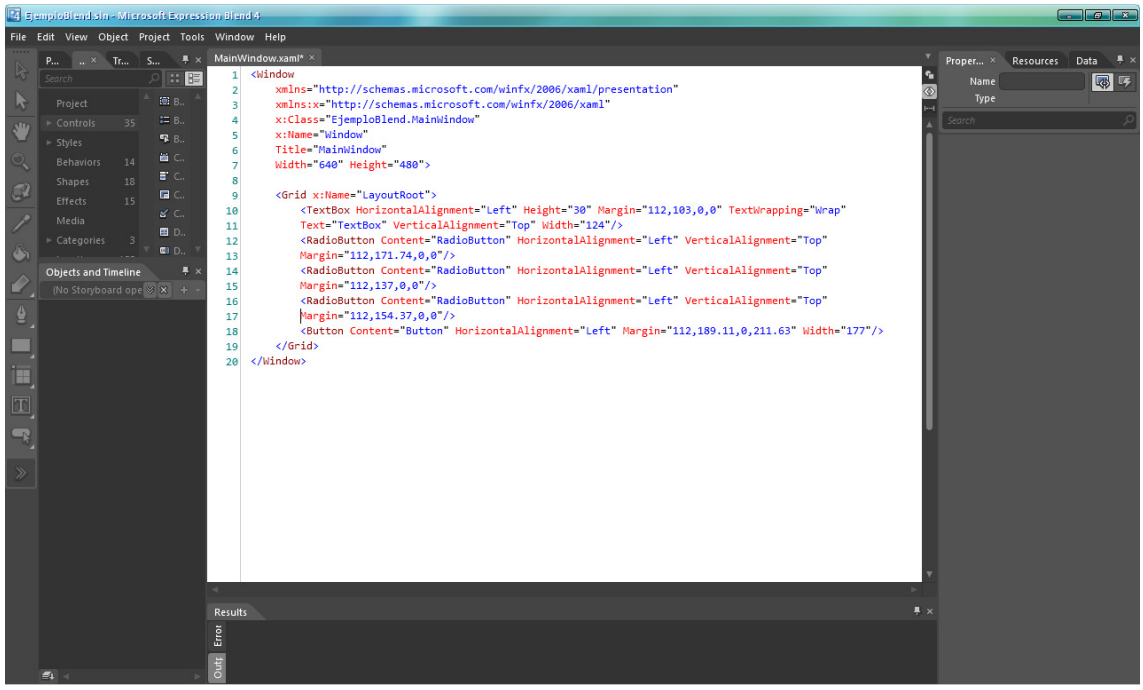

Fig. 2.3 Ventana de trabajo de Expression Blend en modo XAML

Un último aspecto a destacar es el paralelismo que ofrecen en su ejecución Expression Blend y Visual Studio 2010. Si se tienen estos dos programas abiertos a la vez y en ambos se trabaja con la misma solución, cuando se apliquen cambios en uno de los dos, el otro programa mostrará un cuadro de texto advirtiendo del cambio en la solución, dando la oportunidad de cargarla de nuevo. De esta manera, es muy sencillo ver en uno de los programas los cambios que se hayan podido hacer en el otro. Por ejemplo, si cambiamos la

apariencia de un botón en Expression Blend, con pasar a Visual Studio y aceptar la recarga de la solución, podemos probar la aplicación con el nuevo diseño de botón, y así con cualquier aspecto que se cambie dentro del programa, agilizando el proceso de creación.

2.1.2. Microsoft Expression Design

A pesar de que Expression Blend ofrece una amplia gama de controles para añadir a nuestra solución, puede pasar que estos no sean los idóneos para el proyecto. A veces su diseño no encaja con el de la aplicación, y otras, simplemente, necesitamos una imagen que no tenemos y que podríamos crear desde cero en base a unos gustos propios. Expression Design es la herramienta perfecta para estos casos.

Expression Design es un programa de creación de gráfiicos que permite tanto hacer imágenes desde cero como modificar alguna ya existente. El programa ofrece al usuario una serie de herramientas, tales como lápices, pinceles y otras opciones como diferentes estilos de trazo o las características del relleno de los gráficos.

El programa funciona por capas, al igual que otras aplicaciones de diseño como Adobe Photoshop (incluso se pueden importar archivos de dicho programa): un mismo diseño se puede componer de diferentes capas que se superponen unas a otras y cuya opacidad y posición puede ser editada.

Expression Design permite exportar los gráficos creados por el usuario a diferentes formatos, desde .JPG o .PNG, formatos muy comunes, hasta .PSD, el tipo de archivo característico de Photoshop. Incluso se puede exportar en formato XAML, haciendo que la imagen sea vectorial.

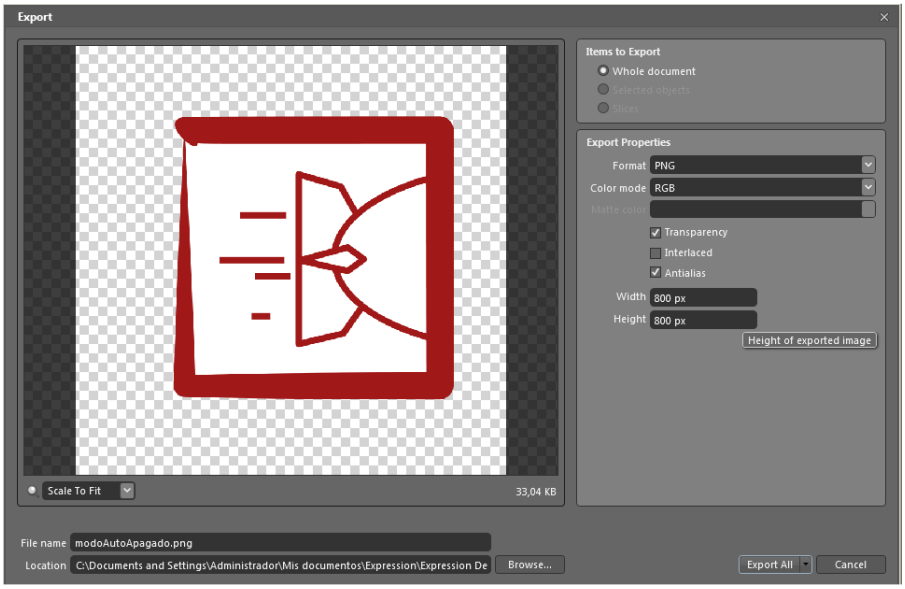

Fig. 2.4 Pantalla de exportación de gráficos de Expression Design

Expression Blend permite crear interfaces ricas en detalles y efectos visuales, pero siempre tendrá ciertos límites en cuanto al aspecto de los controles que se puedan añadir. Expression Design ayuda al programador a superar esos límites y le permite crear multitud de imágenes y gráficos para poder personalizar absolutamente cada pequeño detalle de su aplicación.

2.2. Microsoft Visual Studio

Si bien Expression Blend permite crear una aplicación completa a base de arrastrar los elementos que hagan falta a la ventana principal, no permite escribir absolutamente todo el código necesario para que el programa funcione correctamente. Para ello hará falta utilizar una herramienta básica e indispensable para la programación en C#: Visual Studio.

Visual Studio es un entorno de desarrollo que contiene múltiples herramientas que soportan distintos lenguajes de programación, tales como C++, ASP .NET o el utilizado en este trabajo, C#. Los diferentes tipos de proyecto que Visual Studio permite desarrollar incluyen aplicaciones de consola, de ventana o librerías en diferentes lenguajes. A todos los que ya vienen de serie cabe añadir los tipos que ofrece el SDK de GHI Electronics. Estas nuevas plantillas permiten crear proyectos específicos para cada una de las placas de la compañía que utilicen .NET Micro Framework, incluyendo automáticamente las referencias necesarias para trabajar con cada una de ellas.

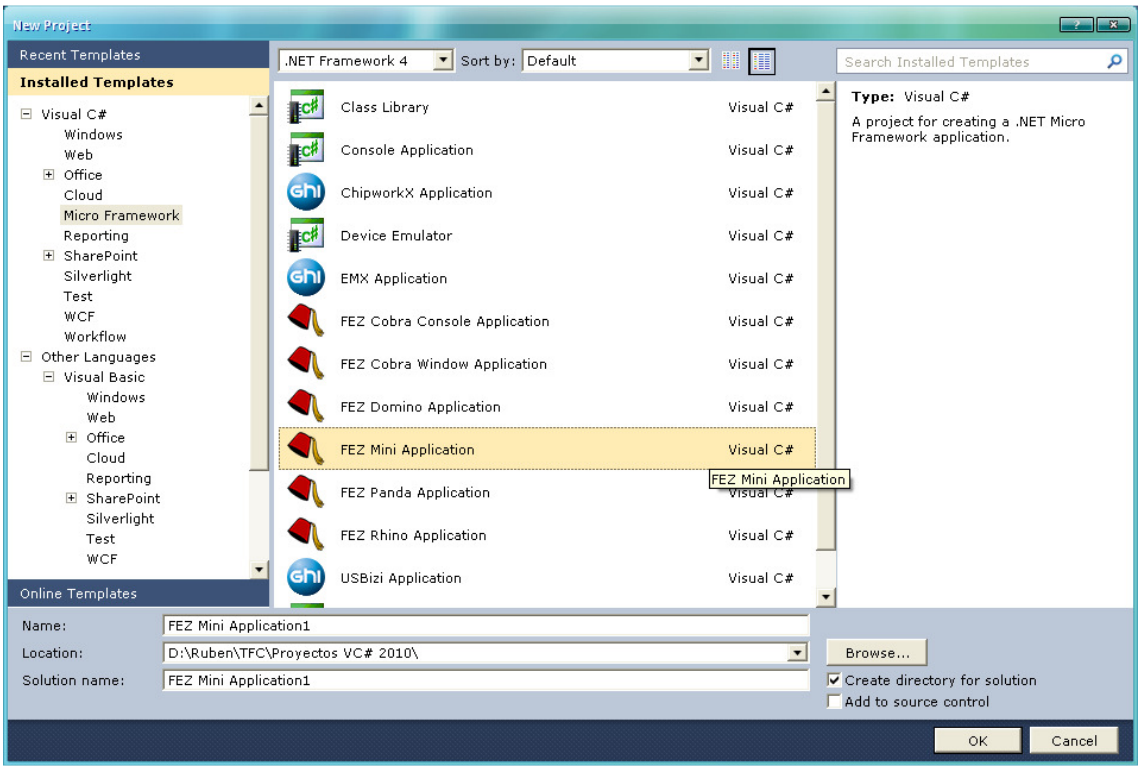

Fig. 2.5 Pantalla de creación de un nuevo proyecto. Lenguaje y tipo de soporte se escogen en la lista de la izquierda; el tipo de aplicación, en la central.

Como ya se ha comentado anteriormente en este mismo capítulo, Expression Blend y Visual Studio se complementan perfectamente, permitiendo la edición del archivo XAML en cualquiera de los dos programas. De esta manera se puede realizar cada parte del proyecto donde más fácil sea; así, en el caso de la estación de tierra, toda la parte gráfica ha sido implementada mediante Expression Blend, mientras que todo el código se ha escrito desde Visual Studio, a excepción de alguna parte del XAML correspondiente a determinados efectos visuales. En el caso del dirigible no hay alternativa, se ha desarrollado todo el código en Visual Studio, puesto que se trata de un proyecto para la placa FEZ Mini, que sólo funciona con .NET Micro Framework y que precisa de la librería de GHI Electronics.

2.3. Conclusiones del capítulo

Tanto Expression Studio como Visual Studio resultan ser herramientas perfectas para la realización del proyecto. Si bien no hay prácticamente ninguna alternativa a su uso debido a la naturaleza de la FEZ Mini, las posibilidades que ofrecen en sus respectivos usos son enormes. Expression Studio es muy fácil de usar y no presenta más límite que el criterio de diseño del programador; por su parte, Visual Studio permite realizar todo tipo de proyectos, y es un producto cuyo uso no es complicado y que cuenta con una amplia comunidad en la red, haciendo que encontrar respuesta a dudas sobre la programación en C# resulte relativamente sencillo.

3. PRIMERAS PRUEBAS DE COMPONENTES

Tras estudiar y comprender los detalles generales sobre el funcionamiento tanto del hardware como del software utilizado en el proyecto es hora de poner en práctica todos esos conocimientos. Sin embargo, no se ha optado por empezar a hacer el programa principal del dirigible desde cero, sino que primero se han realizado pruebas separadas para cada uno de los componentes del zeppelín para comprobar su buen funcionamiento y poder contar con una base sobre la que construir el programa final. Todas esas pruebas y los detalles sobre el funcionamiento observado de los módulos del zeppelín quedan recogidos en este tercer capítulo.

3.1. La placa Embedded Master

Como ya se ha dicho anteriormente, uno de los objetivos fundamentales del proyecto es el de realizar el cambio de placa que utiliza el dirigible para poder añadir sensores adicionales con los que antes no contaba. Este cambio consiste en sustituir la placa Arduino inicial por una placa FEZ Mini, capaz de ejecutar programas escritos en C# mediante el uso de .NET Micro Framework, plataforma pensada para el desarrollo de código para dispositivos de baja memoria.

Sin embargo, al comienzo del proyecto, la placa FEZ Mini no estaba disponible para su uso, por lo que hasta entonces se utilizó otra placa de desarrollo para ir probando los diferentes componentes que se iban a utilizar en el diseño final del dirigible y comprobar su funcionamiento en la práctica. El dispositivo utilizado para tal fin es la placa Embedded Master.

Embedded Master es una placa de desarrollo de la marca GHI Electronics que, al igual que la FEZ Mini, también utiliza .NET Micro Framework, razón por la cual se utilizó para las primeras pruebas del código. No es una de las placas más recientes (no tiene soporte para .NET Micro Framework 4.1, sólo dispone de soporte hasta la versión 3.3), pero cumple con su cometido, ya que lo indispensable es la existencia de ciertas conexiones necesarias para la comunicación XBee y la lectura de datos analógicos. Además de esto, Embedded Master aporta puertos USB, pantalla (no táctil), lector de tarjetas SD, un transductor piezoeléctrico y demás puertos y periféricos que la convierten (o convertían, puesto que hay placas más recientes con mejores características) en una placa muy completa a la hora de desarrollar prototipos. Y para terminar, otra buena razón para el uso de Embedded Master es el hecho de que tanto esta placa como la FEZ Mini pertenecen al mismo fabricante (GHI Electronics), por lo que muchas de las librerías que se utilicen en estas primeras pruebas podrán ser usadas también con la otra placa, permitiendo una fácil transición a FEZ Mini aprovechando todo el trabajo hecho.

3.2. Prueba del XBee

El primer módulo en ser probado es el XBee Pro, ya que es una pieza clave del proyecto. La comunicación mediante radiofrecuencia es la única manera de hacer funcionar el dirigible con el cambio de placa.

Para utilizar el módulo XBee sólo son necesarias cuatro conexiones, tal y como se puede ver en su correspondiente datasheet. Estas conexiones son:

- El pin 1 (VCC), para alimentar el módulo
- El pin 2 (DOUT), por donde salen los datos recibidos por el puerto serie
- El pin 3 (DIN), por donde entran los datos a enviar por el puerto serie
- El pin 10 (GND), la masa del módulo

Estos pines se conectan al cabezal “SV1” de la placa, y la correspondencia entre los del XBee y los que se han utilizado de la placa es la que se muestra en la tabla siguiente:

Tabla 3.1. Correspondencia entre las conexiones del módulo XBee y la placa Embedded Master

Módulo XBee Pro	Placa Embedded Master
Pin 1 (VCC)	Pin 1 (3.3V)
Pin 2 (DOUT)	Pin 4 (UART_RXD2)
Pin 3 (DIN)	Pin 3 (UART_TXD2)
Pin 10 (GND)	Pin 2 (GND)

Para realizar la prueba del XBee Pro se ha escrito un pequeño y sencillo programa que demuestre la buena recepción y transmisión tanto del módulo pequeño como el PKG-U (el módulo PKG-U no es más que un XBee con una carcasa que incluye una antena de mayor tamaño y que se puede conectar directamente al ordenador mediante un cable USB). El programa consiste en establecer una conexión serie entre la placa y el ordenador mediante los módulos XBee y enviar a la placa una letra cualquiera en minúsculas para que esta la reenvíe al ordenador, pasando la misma letra a mayúsculas. El código del programa es el siguiente:

```
public static SerialPort Serial1;

public static void Main()
{
    string frase = "Empieza a escribir...";
    Serial1 = new SerialPort("COM3", 9600, Parity.None, 8, StopBits.One);
    Serial1.Open();
    Serial1.DataReceived += new SerialDataReceivedEventHandler(Serial1_DataReceived);

    Thread.Sleep(-1);
}
```

```
   static void Serial1_DataReceived(object sender, SerialDataReceivedEventArgs e)
   {
      // Check if Chars are received
      if (e.EventType == SerialData.Chars)
      {
         // Create new buffer
         byte[] ReadBuffer = new byte[Serial1.BytesToRead];

         // Read bytes from buffer
         Serial1.Read(ReadBuffer, 0, ReadBuffer.Length);
         string letra = new string(Encoding.UTF8.GetChars(ReadBuffer));


         // Encode to string
         Serial1.Write(Encoding.UTF8.GetBytes(letra.ToUpper()),0,
Encoding.UTF8.GetChars(ReadBuffer).Length);
      }
   }
```

El código que hay después del Main() corresponde al evento de recepción de datos por el puerto serie. Así, cada vez que se reciben datos por el XBee se ejecutará el código escrito dentro de ese evento.

Mediante el programa HyperTerminal se puede comprobar que el buen funcionamiento del módulo XBee y del módem PKG-U tanto en modo de transmisión como de recepción, ya que en pantalla se observa la letra enviada en minúsculas convertida a mayúsculas. El éxito de esta prueba es importante, porque la placa del dirigible no sólo recibirá las órdenes del joystick para navegar, sino que también enviará al ordenador los diferentes datos que se mostrarán en la estación de tierra.

3.3. Prueba del acelerómetro

El acelerómetro MMA7260Q es el siguiente módulo probado para su inserción en la nueva placa del dirigible. Con este módulo será posible monitorizar la dirección en la que acelera el zeppelín, permitiendo ver si se mueve en una dirección que no debería (ya sea por un fallo de hardware, una corriente de aire o cualquier otro fenómeno) y poder corregir su rumbo para evitar accidentes.

El acelerómetro viene montado en una pequeña placa PCB que incluye conectores para utilizar los pines mínimos necesarios para su utilización. Estos son los siguientes:

- Los pines 1 y 2 (g-Select1 y g-Select2, respectivamente), utilizado para la selección de la sensibilidad del sensor
- El pin 3 (V_{DD}), para alimentar el módulo
- El pin 4 (V_{SS}), la masa del módulo
- El pin 12 ($\overline{\text{Sleep Mode}}$), que activa o suspende el módulo (como está negado, un 1 lo activa y un 0 lo suspende)

- Los pines 13, 14 y 15 (Z_{OUT}, Y_{OUT} y X_{OUT}, respectivamente), donde se leen los voltajes correspondientes a la aceleración medida en cada uno de los ejes

En este caso, los pines 1 y 2 se pueden dejar desconectados, ya que la sensibilidad que se establece dejándolos al aire (800 mV/g) funciona bien para la aplicación. Para el resto de conexiones se utilizan los cabezales "SV1" y "Analog" de la placa, con la siguiente correspondencia entre pines:

Tabla 3.2. Correspondencia entre las conexiones del acelerómetro y la placa Embedded Master

Acelerómetro	Placa Embedded Master	
	SV1	Analog
Pin 3 (V_{DD})	Pin 1 (3.3V)	---
Pin 4 (V_{SS})	Pin 2 (GND)	---
Pin 12 ($\overline{\text{Sleep Mode}}$)	Pin 10 (E26)	---
Pin 13 (Z_{OUT})	---	Pin 3 (ADC1)
Pin 14 (Y_{OUT})	---	Pin 4 (ADC2)
Pin 15 (X_{OUT})	---	Pin 1 (ADC0)

El código para probar el acelerómetro es muy sencillo; lo único que hace es leer los datos enviados por el sensor mediante los ADC de la placa e imprime por pantalla los correspondientes valores. Esta prueba es suficiente para verificar el correcto funcionamiento del sensor, ya que al permanecer quieto, este experimenta la aceleración de la gravedad, que es constante y ayuda a ver de forma más clara la diferencia entre los valores de salida de cada eje. El código del programa es el siguiente:

```
public static void Main()
    {
        AnalogIn entradaX = new AnalogIn(AnalogIn.AnalogInputPin.AIN0, 0, 500);
        AnalogIn entradaY = new AnalogIn(AnalogIn.AnalogInputPin.AIN1, 0, 500);
        AnalogIn entradaZ = new AnalogIn(AnalogIn.AnalogInputPin.AIN2, 0, 500);
        OutputPort gs1 = new OutputPort(GHIElectronics.Hardware.EmbeddedMaster.Pins.E28, false);
        OutputPort gs2 = new OutputPort(GHIElectronics.Hardware.EmbeddedMaster.Pins.E29, false);
        OutputPort sleep = new OutputPort(GHIElectronics.Hardware.EmbeddedMaster.Pins.E26x, true);

        Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth, SystemMetrics.ScreenHeight);
        sleep.Write(true);

        while (true)
        {
            LCD.Clear();
            LCD.DrawText("Lectura X: " + entradaX.Read().ToString(),
                    Resources.GetFont(Resources.FontResources.small), Color.White, 0, 0);
            LCD.DrawText("Lectura Y: " + entradaY.Read().ToString(),
                    Resources.GetFont(Resources.FontResources.small), Color.White, 0, 10);
            LCD.DrawText("Lectura Z: " + entradaZ.Read().ToString(),
                    Resources.GetFont(Resources.FontResources.small), Color.White, 0, 20);
            LCD.Flush();
        }
    }
```

Por si acaso, se han probado las diferentes sensibilidades que ofrece el acelerómetro (cambiando el estado inicial de los pines de selección en el programa de prueba), y después de realizar los tests, la sensibilidad que ofrece unos resultados más ajustados a la función que se necesita en el dirigible es la primera. Esto se debe a que cuanto menor sea la sensibilidad, mayor será la diferencia entre valores que se muestre a la salida por cada g que soporte el zeppelín; por lo tanto, puesto que el dirigible no va a velocidades muy altas, una sensibilidad baja ofrecerá valores lo suficientemente distantes entre sí para cada eje, permitiendo discernir el sentido de la aceleración.

3.4. Prueba del servo

Como se explicaba en el primer capítulo, el servo es una pieza básica en el funcionamiento del dirigible, ya que se encarga del posicionamiento de sus hélices. Sin embargo, el cambio en el modo de transmisión utilizado por la placa del dirigible implica una renovación en el funcionamiento de este módulo, debido a que antes se conectaba al receptor radiocontrol. Ahora tendrá que conectarse directamente a la placa.

El servo necesita 3 conexiones: una a la alimentación, otra a la masa y otra por la que se envíe el pulso que lo hará girar el ángulo necesario. Las conexiones en la placa Embedded Master son las siguientes:

Tabla 3.3. Correspondencia entre las conexiones del servo y la placa Embedded Master

Servo	**Placa Embedded Master**
	SV1
Pin 1 (V_{CC})	Pin 1 (3.3V)
Pin 2 (GND)	Pin 2 (GND)
Pin 3 (Control)	Pin 9 (E27)

El funcionamiento del servo es muy simple. El motor se activa mediante un pulso y rota un determinado ángulo en función de la duración del pulso enviado, permaneciendo en esa posición hasta que vuelva a recibir una nueva orden. Debido a este comportamiento, hay dos maneras de utilizar un servo mediante unas clases de la librería de GHI Electornics para .NET Micro Framework:

- La clase PWM (Pulse Width Modulation). Mediante esta clase podemos enviar pulsos con una frecuencia y un ciclo de trabajo determinados, siempre y cuando la palca utilizada disponga de un pin de tipo PWM (la placa de desarrollo Embedded Master cuenta con 2 de estos pines).

- La clase OutputCompare. Utilizando esta clase se puede configurar un pin para que retenga su salida en el estado deseado durante un tiempo concreto.

Aunque ambas soluciones son muy parecidas y lo lógico sería utilizar un pin de tipo PWM porque la placa dispone de varios, la solución final utiliza la clase OutputCompare. Esto se debe a que para cubrir el rango de 180° del servo se utilizan pulsos que van desde los 1,25 ms a los 1,75 ms en estado alto. La clase PWM incluida en la librería para la placa Embedded Master no permite realizar diferencias tan pequeñas, por lo que la única posibilidad que queda es la clase OutputCompare.

Para probar el funcionamiento del servo en la placa de desarrollo se escribió el siguiente código, que lo mueve automáticamente a 3 posiciones diferentes:

```
public static void Main()
{
   OutputCompare oc = new OutputCompare((Cpu.Pin)ChipworkX.Pin.PC5, true, 2);

   while (true)
   {
      oc.Set(true, new uint[] { 1700, 18300 }, 0, 2, true);
      Thread.Sleep(1000);

      oc.Set(true, new uint[] { 800, 19200 }, 0, 2, true);
      Thread.Sleep(1000);

      oc.Set(true, new uint[] { 1250, 18750 }, 0, 2, true);
      Thread.Sleep(1000);
   }
}
```

Los valores que aparecen en la función Set del objeto OutputCompare son los valores correspondientes a las 3 posiciones principales del servo; esto es, -90° (para hacer que el dirigible suba), 0° (hacer que vaya hacia adelante) y 90° (para que vaya hacia abajo). Gracias a estos valores de los extremos y a los que se lean del joystick se podrá establecer una función que mueva el servo al ángulo justo en función de la palanca del joystick.

3.5. Conexión del sensor de ultrasonidos

El último módulo en ser testeado mediante la placa Embedded Master es uno de los componentes iniciales del BlimpDuino junto con el servo. Como se ha podido leer en el primer capítulo, este dispositivo se utiliza para poder medir a qué altura está volando el dirigible. De esta manera, se podrán realizar aterrizajes controlados o, simplemente, vigilar si el zeppelín vuela a la altura esperada para detectar posibles problemas (en los motores o fugas de helio en el globo).

El sensor de ultrasonidos cuenta con 7 pines, de los cuales 3 ofrecen las medidas del sensor en diferentes formatos. Puesto que sólo hace falta una de las salidas, las conexiones necesarias para utilizar el sensor se reducen a 3:

alimentación, masa y la salida del sensor. Los 3 tipos de salida, tal y como se explica detalladamente en el primer capítulo, son diferentes entre sí en cuanto al formato de los datos se refiere. La decisión tomada ha sido la de utilizar la salida analógica (consultar el primer capítulo si es necesario saber cuáles son las otras dos), ya que de cara al uso del sensor en la placa FEZ Mini este tipo de salida ofrecerá cierta ventaja que se explicará en el siguiente capítulo.

En la siguiente tabla se pueden observar las correspondencias entre la placa y los pines del sensor de ultrasonidos:

Tabla 3.4. Correspondencia entre las conexiones del sensor de ultrasonidos y la placa Embedded Master

Sensor de ultrasonidos	Placa Embedded Master	
	SV1	Analog
Pin 1 (GND)	Pin 2 (GND)	---
Pin 2 (+5) (Alimentación de 2,5V a 5,5V)	Pin 1 (3.3V)	---
Pin 5 (AN)	---	Pin 2 (ADC3)

Una vez establecidas las conexiones es el momento de probar el sensor. Se ha escrito un código que obtendrá las medidas realizadas por el módulo y éstas se mostrarán por la pantalla de la placa, tal y como se hizo también con el acelerómetro. El código utilizado para la prueba es este:

```
public static void Main()
    {
        AnalogIn ultra = new AnalogIn(AnalogIn.AnalogInputPin.AIN3, 0, 300);
        Bitmap LCD = new Bitmap(SystemMetrics.ScreenWidth, SystemMetrics.ScreenHeight);

        while (true)
        {
            LCD.Clear();
            LCD.DrawText("Altura: " + ultra.Read().ToString(),
                    Resources.GetFont(Resources.FontResources.small), Color.White, 0, 0);
            LCD.Flush();
        }
    }
```

3.6. Conclusiones del capítulo

Es un programa muy sencillo, pero suficiente para comprobar el funcionamiento del sensor. Los datos obtenidos de las medidas se adaptan al valor máximo establecido en la declaración del objeto AnalogIn, además de estar expresados en pulgadas, tal y como se explica en el datasheet. Se tendrá que tener en cuenta este detalle de cara al futuro, cuando el dirigible envíe datos a la estación base. En ese momento será necesaria una conversión de pulgadas a centímetros para que se entienda mejor el dato.

Todas estas pruebas que se han realizado han servido para comprender mejor el funcionamiento de cada uno de los módulos que se añadirán al dirigible y para decidir, en algunos casos, el modo de funcionamiento del módulo. La ventaja de haber podido utilizar la placa Embedded Master es que, puesto que también utiliza .NET Micro Framework (aunque sea una versión anterior a la más reciente), todos los programas utilizados en dichos tests se van a poder reciclar de cara al programa final para el zeppelín, facilitando la escritura de código del mismo.

Ahora, una vez probados los módulos en esta placa, sólo queda disponer de la FEZ Mini para comenzar a desarrollar de verdad el proyecto, dividido en dos grandes partes.

4. MONTAJE DE LA PLACA FINAL

Después de realizar las pruebas pertinentes para cada módulo por separado, es hora de juntar todas las partes que harán funcionar el dirigible y comenzar el montaje de la nueva placa. En este capítulo se detalla la disposición de los elementos en la placa FEZ Mini, las pruebas de compatibilidad realizadas y la estructura del programa final del dirigible. Además, como ahora el zeppelín cuenta con más sensores, habrá que diseñar un soporte en el que poner tanto la placa FEZ Mini como los diferentes módulos que se le añadan.

4.1. Funcionamiento y conexiones de la nueva placa

En el primer capítulo aparecía un diagrama de bloques que mostraba de manera simplificada los componentes de la placa Arduino que utilizaba el dirigible en un principio. Es hora de hacer un pequeño repaso de cómo funcionaba esa placa y qué limitaciones presentaba. El funcionamiento es sencillo: mediante el mando de radiocontrol se envían las órdenes al dirigible (el ángulo al que debe estar el servo, si utilizar solo un motor para girar o los dos para ir en línea recta y si el movimiento debe ser hacia adelante o hacia atrás). Esta señal es captada por el receptor, que está conectado al servo y a la placa Arduino. A su vez, mediante el código del programa, la placa establece las instrucciones a seguir por el driver y éste realiza las acciones oportunas sobre los motores, mientras que el sensor de ultrasonidos va realizando medidas y enviando a la placa un voltaje en función de la distancia a la que está el zeppelín respecto al suelo.

El inconveniente de utilizar esta placa es que la comunicación, de entrada, es “unidireccional”. Cuando el zeppelín está en el aire tan solo recibe la señal del mando, ya que a menos que se le conecte un módulo XBee, no cuenta con ninguna forma de enviar datos de vuelta de manera inalámbrica. Como se ha visto en el capítulo 1, la única forma que tiene de mostrar datos relativos a su estado es mediante un cable serie conectado al ordenador. Utilizando la consola serie del programa de desarrollo Arduino se puede ver a qué distancia se encuentra el zeppelín del suelo y el nivel de batería, pero de esta manera es imposible pilotar el dirigible.

En capítulos anteriores se ha explicado varias veces que se va a utilizar un módulo XBee Pro para poder realizar las comunicaciones entre la estación base y el dirigible. En este caso sí que se podrán enviar y recibir mensajes en ambos sentidos. En la siguiente figura se puede observar esquemáticamente la distribución de los elementos que conformarán la góndola del zeppelín, incluyendo la placa FEZ Mini que funcionará como núcleo principal, y la estación base, que aunque sea una parte del proyecto externa al cuerpo del dirigible es indispensable para el manejo de éste:

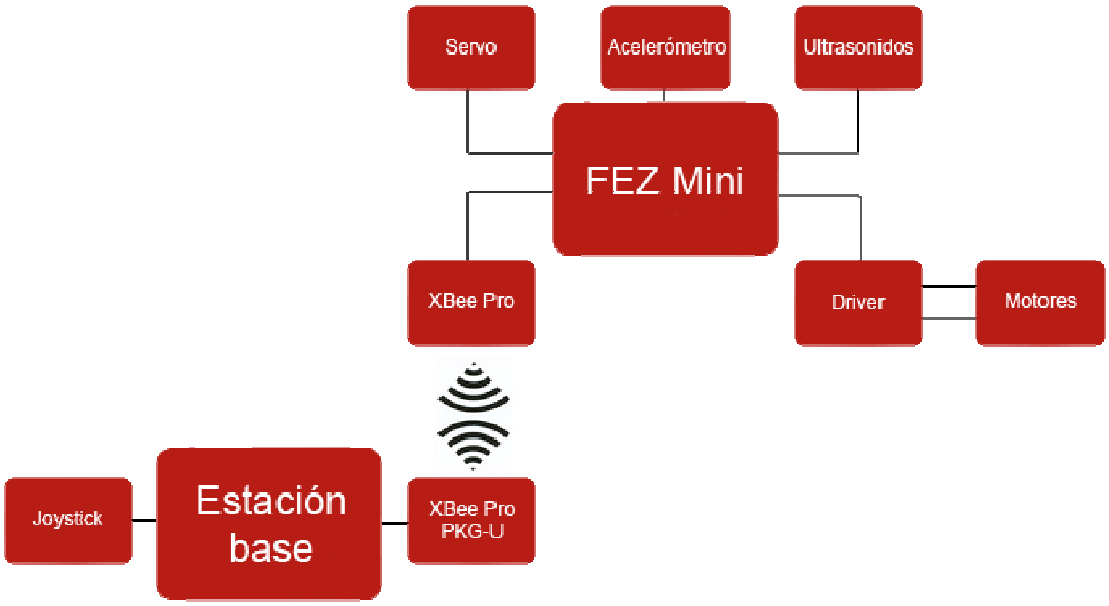

Fig. 4.1 Esquema de la nueva placa del dirigible

El funcionamiento del dirigible utilizando la FEZ Mini varía un poco respecto al modelo con Arduino. Mediante un joystick (que reemplaza al anterior mando de radiocontrol, lo que deriva en la extracción del módulo RC de la góndola del dirigible) se envían al zeppelín las mismas órdenes que en el caso anterior, solo que esta vez se utiliza el módulo XBee para la comunicación y el formato de éstas no será el mismo. El XBee pasa los comandos recibidos a la placa, y ésta mueve el servo el ángulo necesario y establece las salidas del driver para activar los motores. Además, durante todo el rato, tanto el acelerómetro como el sensor de ultrasonidos realizan medidas cuyos resultados son recopilados por la placa, que gracias de nuevo al XBee, los envía a la estación de tierra, donde el usuario puede visualizarlos.

A pesar de que por el momento sólo haya un sensor nuevo, se ha producido un cambio muy importante en el funcionamiento del dirigible. Ahora ya puede comunicarse con un dispositivo terrestre mientras se encuentra en medio de un vuelo, otorgando al usuario la capacidad de monitorizar el estado del zeppelín.

En el capítulo anterior, además de los programas utilizados en la prueba de los componentes, se adjuntaban tablas con la correspondencia entre los pines de cada uno de los módulos y la placa Embedded Master. A continuación se muestran un esquemático realizado con el programa EAGLE con todas las conexiones y una tabla que ofrece una comparación entre los actuales pines que se utilizarán en la placa FEZ Mini y los que se usaron en las pruebas del capítulo 3:

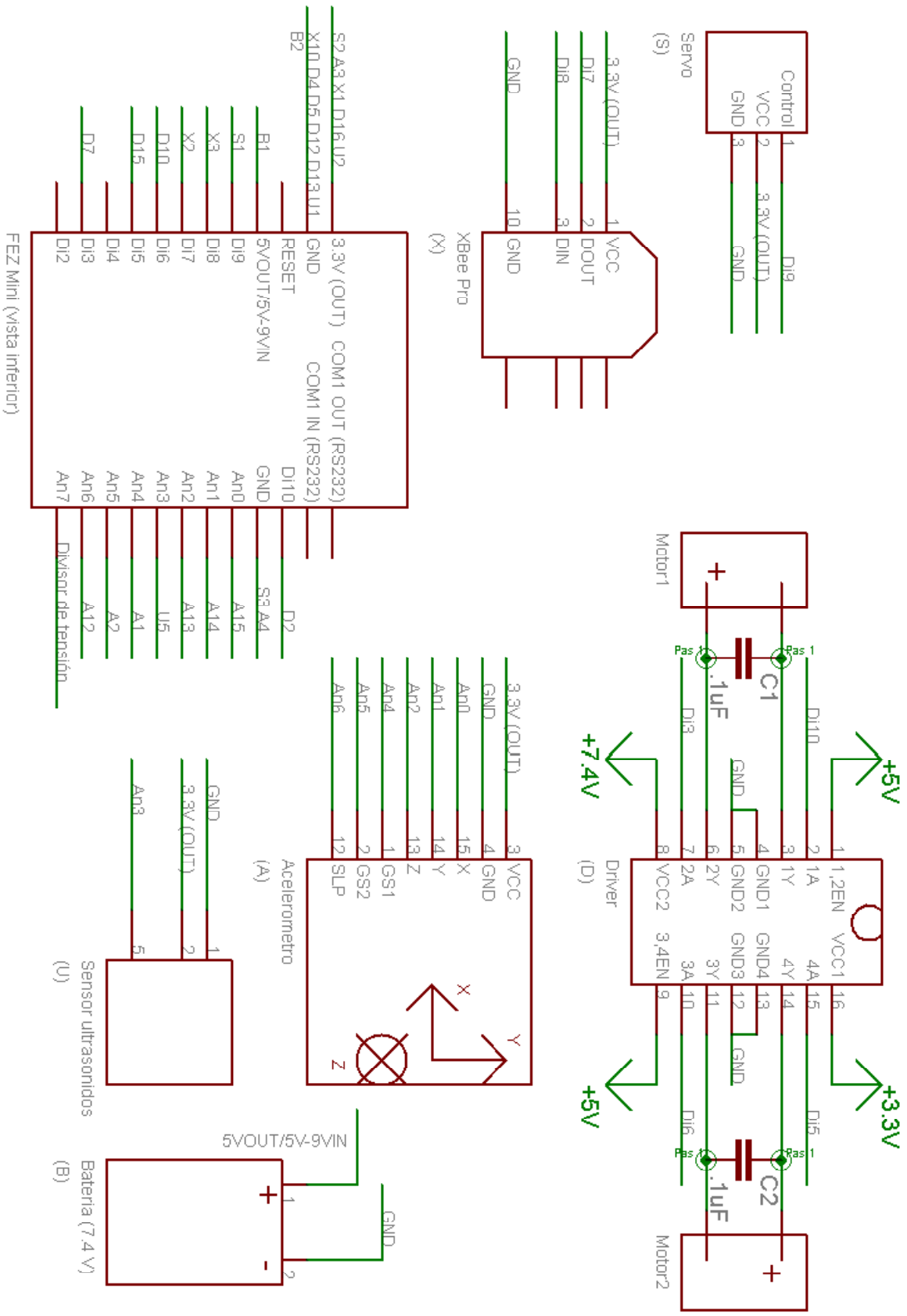

Fig. 4.2 Esquemático de todas las conexiones de los módulos del Net Zeppelin

Tabla 4.1. Correspondencia entre los pines de los diferentes módulos y los de la placa FEZ Mini

Módulo	Pin del módulo	Pin en Embedded Master	Pin en FEZ Mini
XBee	Pin 1 (VCC)	Pin 1 (SV1) (3.3V)	Pin 1 (3.3 V)
	Pin 2(DOUT)	Pin 4 (SV1) (UART_RXD2)	Pin 7 (Di7) (COM3)
	Pin 3 (DIN)	Pin 3 (SV1) (UART_TXD2)	Pin 6 (Di8) (COM3)
	Pin 10 (GND)	Pin 2 (SV1) (GND)	Pin 2 (GND)
Servo	Pin 1 (V_{CC})	Pin 1 (SV1) (3.3V)	Pin 1 (3.3 V)
	Pin 2 (GND)	Pin 2 (SV1) (GND)	Pin 2 (GND)
	Pin 3 (Control)	Pin 9 (SV1) (E27)	Pin 5 (Di 9)
Ultrasonidos	Pin 1 (GND)	Pin2 (SV1) (GND)	Pin 2 (GND)
	Pin 2 (V_{CC})	Pin 1 (SV1) (3.3V)	Pin 1 (3.3 V)
	Pin 5 (AN)	Pin 2 (Analog) (ADC3)	Pin 17 (An3)
Acelerómetro	Pin 3 (V_{DD})	Pin 1 (SV1) (3.3V)	Pin 1 (3.3 V)
	Pin 4 (V_{SS})	Pin 2 (SV1) (GND)	Pin 2 (GND)
	Pin 12 ($\overline{\text{Sleep Mode}}$)	Pin 10 (SV1) (E26)	Pin 14 (An6)
	Pin 13 (Z_{OUT})	Pin 3 (Analog) (ADC1)	Pin 20 (An0)
	Pin 14 (Y_{OUT})	Pin 4 (Analog) (ADC2)	Pin 19 (An1)
	Pin 15 (X_{OUT})	Pin 1 (Analog) (ADC0)	Pin 18 (An2)
Driver	Pin (VCC1)	-	Pin 1 (3.3 V)
	Pin (VCC2)	-	7,4V (batería)
	Pines (1,2EN y 3,4EN)	-	5V (regulador)
	Pin (1A)	-	Pin 22(Di10)
	Pin (2A)	-	Pin 11 (Di3)
	Pin (3A)	-	Pin 8 (Di6)
	Pin (4A)	-	Pin 9 (Di5)

El driver fue probado directamente en la placa FEZ Mini por razones de disponibilidad del material (tanto del driver como de la placa final) y por la incomodidad de realizar dichas pruebas con otra placa que no sea la FEZ Mini, ya que ésta se puede acoplar directamente a una protoboard, no como la Embedded Master.

4.2. Pruebas de hardware en la placa FEZ Mini

Teniendo en cuenta el cambio en la versión de .NET Micro Framework que se produce al pasar de la placa Embedded Master a la FEZ Mini, no está de más volver a hacer las mismas pruebas de funcionamiento que en el caso de la

primera placa de desarrollo. Esto ayudará a comprobar el funcionamiento de las nuevas clases que habrá que usar con la FEZ Mini y a reescribir el código en base a la nueva placa, lo cual facilitará la escritura del programa final para el dirigible, ya que habrá líneas que se podrán reutilizar.

Puesto que en el capítulo 1 ya se han explicado las características de la placa FEZ Mini, en los siguientes apartados se irán detallando los cambios que se han tenido que aplicar en el código de prueba de los diferentes componentes respecto a los utilizados en la placa Embedded Master y las razones por las cuales se han realizado.

4.2.1. Prueba del XBee

El primer módulo probado en la nueva placa es el XBee, la pieza que permite el envío de información tanto del dirigible a la base como al revés.

Antes, el XBee iba conectado al puerto COM3 de la placa Embedded Master mediante los pines específicos para esta función. En este caso, como se puede ver en la tabla 4.1, el módulo ha sido conectado a unos pines digitales en los que la función de puerto serie es secundaria. Esto se debe a que, a pesar de que la FEZ Mini cuenta con dos pines para la recepción y la transmisión mediante el puerto COM1, este puerto no ofrecía respuesta alguna; aparte de esto, en el artículo sobre XBee de la página web http://wiki.tinyclr.com se puede encontrar que el ejemplo de conexión de XBee en la placa FEZ Mini utiliza el puerto COM3 para ello.

El código utilizado para probar el módulo XBee en la nueva placa es el siguiente:

```
public static SerialPort serial1;

public static void Main()
{
    serial1 = new SerialPort("COM3", 9600, Parity.None, 8, StopBits.One);
    serial1.Open();
    serial1.DataReceived += new SerialDataReceivedEventHandler(serial1_DataReceived);

    Thread.Sleep(-1);
}

static void serial1_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
    // Check if Chars are received
    if (e.EventType == SerialData.Chars)
    {
        // Create new buffer
        byte[] readBuffer = new byte[serial1.BytesToRead];

        // Read bytes from buffer
        serial1.Read(readBuffer, 0, readBuffer.Length);
        string letra = new string(Encoding.UTF8.GetChars(readBuffer));

        serial1.Write(Encoding.UTF8.GetBytes(letra.ToUpper()), 0, letra.Length);
    }
}
```

Comparando este programa con el utilizado en la placa Embedded Master se puede ver que el código es exactamente igual. Entre las versiones 3.3 y 4.1 de .NET Micro Framework no hay cambios en cuanto a las instrucciones que hacen falta para abrir puertos serie y utilizarlos para el envío y recepción de información y las referencias necesarias son las mismas, por lo que el código anterior se puede aprovechar por completo para su uso en la FEZ Mini.

4.2.2. Prueba del acelerómetro

Siguiendo el mismo orden de exposición que en el capítulo 3, el siguiente componente a probar es el acelerómetro. Contando alimentación y masa, este módulo necesita 6 conexiones para su funcionamiento. Hay 4 imprescindibles, que son las 3 salidas analógicas para cada uno de los ejes en los que puede ir la aceleración y el puerto de suspensión, que al estar negado, hace falta dejar en estado alto para que el acelerómetro realice medidas. Los otros 2 pines son opcionales, puesto que sirven para la selección de la sensibilidad. Sin embargo, en este proyecto no es necesario cambiarla en ningún momento debido al cometido del acelerómetro (una explicación con más detalle sobre este asunto puede encontrarse en el capítulo 3, apartado 3).

El programa escrito para utilizar el acelerómetro en la FEZ Mini es el siguiente:

```
public static void Main()
    {
        AnalogIn inX = new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An0);
        AnalogIn inY = new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An1);
        AnalogIn inZ = new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An2);
        OutputPort sleep = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.An6,true);
        OutputPort gs1 = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.An4, false);
        OutputPort gs2 = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.An5, false);

        while (true)
        {
            Debug.Print("**********************************");
            Debug.Print("inX: " + inX.Read().ToString());
            Debug.Print("inY: " + inY.Read().ToString());
            Debug.Print("inZ: " + inZ.Read().ToString());
            Debug.Print("**********************************");
        }
    }
```

Como en el caso anterior, el código es prácticamente igual. En la versión de Embedded Master se mostraban los datos leídos de las entradas analógicas por la pantalla de la que disponía la placa. Como ahora no se dispone de ninguna, se muestran a través de la pantalla de Debug de Visual Studio.

El único cambio significativo es que ahora, cuando se crea una nueva variable de tipo AnalogIn, no hace falta especificar el rango de valores entre los que se moverá la salida. Esto se debe a que las referencias necesarias son diferentes: en el caso de la placa Embedded Master la clase AnalogIn pertenecía a la referencia GHIElectronics.Hardware, mientras que en el caso de FEZ Mini pertenece a GHIElectronics.NETMF.FEZ.

4.2.3. Prueba del servo

Este componente tan sólo necesitaba una conexión (sin contar alimentación y masa) para su funcionamiento, que como ya se ha dicho anteriormente, utiliza pulsos para mover el servo un ángulo determinado.

La placa FEZ Mini dispone de varios pines de tipo PWM, que como se detalla en el capítulo 3, ofrece pulsos de un determinado ciclo de trabajo. Sin embargo, por motivos de comodidad y por haberse demostrado el buen funcionamiento del recurso ya utilizado, se usará de nuevo la clase OutputCompare para mover el servo, tal y como se hizo en la placa Embedded Master.

El nuevo código utilizado es el que se muestra a continuación:

```
public static void Main()
    {
        OutputCompare oc = new OutputCompare((Cpu.Pin)FEZ_Pin.Digital.Di9, true, 2);

        while (true)
        {
            oc.Set(true, new uint[] { 1700, 18300 }, 0, 2, true);   // Servo hacia abajo
            Thread.Sleep(1000);
            oc.Set(true, new uint[] { 800, 19200 }, 0, 2, true);    // Servo hacia arriba
            Thread.Sleep(1000);
            oc.Set(true, new uint[] { 1250, 18750 }, 0, 2, true);   // Servo en el centro
            Thread.Sleep(1000);
        }
    }
```

Una vez más, el código no ha cambiado ni un ápice. La clase OutputCompare utiliza los mismos argumentos tanto en la versión 3.3 de .NET Micro Framework como en la 4.1, y por supuesto, los valores necesarios en estado alto para mover los diferentes ángulos son los mismos, ya que este aspecto sólo depende del propio servo.

4.2.4. Prueba del sensor de ultrasonidos

El sensor de ultrasonidos es el siguiente módulo en ser probado en la placa FEZ Mini. Como se ve en la tabla 4.1 del presente capítulo, este sensor necesita 3 conexiones mínimas: la de alimentación, la de masa y la salida de datos. El programa utilizado para la prueba es el siguiente:

```
public static void Main()
    {
        AnalogIn ultra = new AnalogIn((AnalogIn.Pin)FEZ_Pin.AnalogIn.An3);

        while (true)
        {
            int distancia = Convert.ToInt32(ultra.Read().ToString());
            Debug.Print("Distancia (cm): " + distancia);
        }
    }
```

Comparando con el programa del capítulo 3, lo único que cambia es la manera de asignar el pin de la placa. También en el capítulo anterior se justificó la elección de la salida analógica argumentando que, de cara a utilizar la placa FEZ Mini, dicha salida presentaría una ventaja. Pues bien, tal y como se puede deducir de la línea en la que se declara el objeto de tipo AnalogIn al ser comparada con la misma línea del otro programa (y tal y como se ha explicado en el apartado 4.2.2 de este mismo capítulo, porque el acelerómetro también cuenta con este tipo de pines), en esta versión del SDK de GHI Electronics el constructor de la clase AnalogIn no requiere que se introduzca un valor máximo de escala. De esa manera se obtienen las lecturas analógicas directamente en la escala adecuada. De ahí que los resultados mostrados en el ordenador durante la prueba sean las medidas exactas reales, sólo que en pulgadas.

Ese pequeño detalle sobre el valor máximo puede parecer no tener importancia, pero el hecho de no tener que calcular factores de escala cada vez que se precisen lecturas analógicas ayuda a no tener que estar pendiente de qué valor pasar como parámetro a qué puerto analógico, simplificando el código y el trabajo del programador.

4.2.5. Prueba del driver

El último elemento de hardware en ser probado en la FEZ Mini es el driver que permitirá regular la potencia recibida por los motores para que estos operen de forma adecuada. Este componente se ha probado directamente en esta placa por cuestiones de disponibilidad. Un aspecto importante sobre el funcionamiento de los motores ya comentado en capítulos anteriores es que lo ideal sería que éstos no fuesen siempre a la máxima potencia, sino que ésta se regulara en función de cuan inclinado está el joystick. Por eso el control del driver se llevará a cabo como el servo: mediante pulsos. El driver recibirá los pulsos y filtrará la señal, resultando un nivel más alto o más bajo en función del tiempo en estado alto de los pulsos.

Cuando se realizaron las pruebas del servo se decidió utilizar la clase OutputCompare porque el código ya estaba escrito de antes utilizando esta clase. Sin embargo, como en esta versión de Micro Framework la clase PWM acepta tiempos de ns para establecer los pulsos y puesto que el driver se va a probar por primera vez, se utilizará la clase PWM para comprobar su funcionamiento.

El código escrito para probar el driver es el que se muestra a continuación:

```
public static void Main()
    {
        PWM pwm = new PWM((PWM.Pin)FEZ_Pin.PWM.Di5);
        while (true)
        {
            pwm.SetPulse(50000000, 2500000);    // velocidad 1
            Thread.Sleep(1000);
            pwm.SetPulse(50000000, 4000000);    // velocidad 2
            Thread.Sleep(1000);
            pwm.SetPulse(50000000, 6000000);    // velocidad 3
```

```
            Thread.Sleep(1000);
            pwm.SetPulse(50000000, 8000000);    // velocidad 4
            Thread.Sleep(1000);
            pwm.SetPulse(50000000, 10000000);   // velocidad 5
            Thread.Sleep(1000);
        }
    }
```

Los valores de tiempo que pide la función SetPulse de la clase PWM están, como ya se ha dicho antes, expresados en ns, por lo que el periodo de cada pulso es de 50 ms. A partir de ahí, cuanta menos velocidad se requiera de los motores, más corto deberá ser el tiempo en estado alto. Tal y como está conectado el driver (a 7,4V para mover los motores), utilizar todo el periodo en estado alto hace que los motores funcionen a una velocidad demasiado alta como para poder controlarlo. Mediante diferentes pruebas, se ha fijado el periodo alto máximo de 10 ms, que deriva en una velocidad más o menos alta para el tamaño del dirigible. En el siguiente capítulo se especificará cómo se controla la asignación de potencia en función de la posición del joystick.

Como se ha podido ver en todos los apartados, los códigos utilizados para la prueba de los módulos en la nueva placa son exactamente o prácticamente iguales que los usados en la placa Embedded Master. Esto ha facilitado mucho la transición de una placa a otra, ya que todo el código se ha podido reutilizar para el programa final del dirigible. Los únicos aspectos que se han modificado en estos programas han sido las asignaciones de los pines, que por supuesto no son iguales de una placa a la otra.

Lo que sí es diferente es la manera de asignarlos, ya que GHI Electronics ha creado un enumerador dentro de la clase GHIElectronics.NETMF.FEZ, llamada FEZ_Pins. Este enumerador clasifica los diferentes pines de las placas FEZ según su tipo (AnalogIn, Digital, PWM…), y para cada uno de ellos ofrece los diferentes pines disponibles con el mismo nombre con el que aparecen en la placa. En el caso de Embedded Master hacía falta consultar el datasheet para saber el identificador de cada pin, puesto que no estaban escritos en la placa. Si bien en ambos casos se pueden identificar los pines sin problemas, en el caso de FEZ Mini, el enumerador ofrece cierta comodidad y ahorra tiempo de consulta de documentos. Es un pequeño detalle que aunque no afecta en absoluto al trabajo final, agiliza el trabajo un cierto punto.

4.3. Net Zeppelin: programa final

Las pruebas anteriormente descritas han servido para dos cosas: para comprobar que los módulos del dirigible funcionaban igual de bien que con la placa Embedded Master y para ir disponiendo de trozos de código que se pudiesen reutilizar en este punto del proyecto. Ahora que ya se dispone de todo el material necesario, es hora de escribir el programa principal del nuevo dirigible: Net Zeppelin.

4.3.1. Clases del Net Zeppelin

El código completo del Net Zeppelin está incluido en el apartado de anexos, dado que es demasiado largo como para aparecer aquí. Sin embargo, lo que si se expondrán a continuación son las diferentes clases que lo componen y cuál es su cometido.

En el siguiente diagrama de clases se puede ver el número de clases y sus funciones y campos:

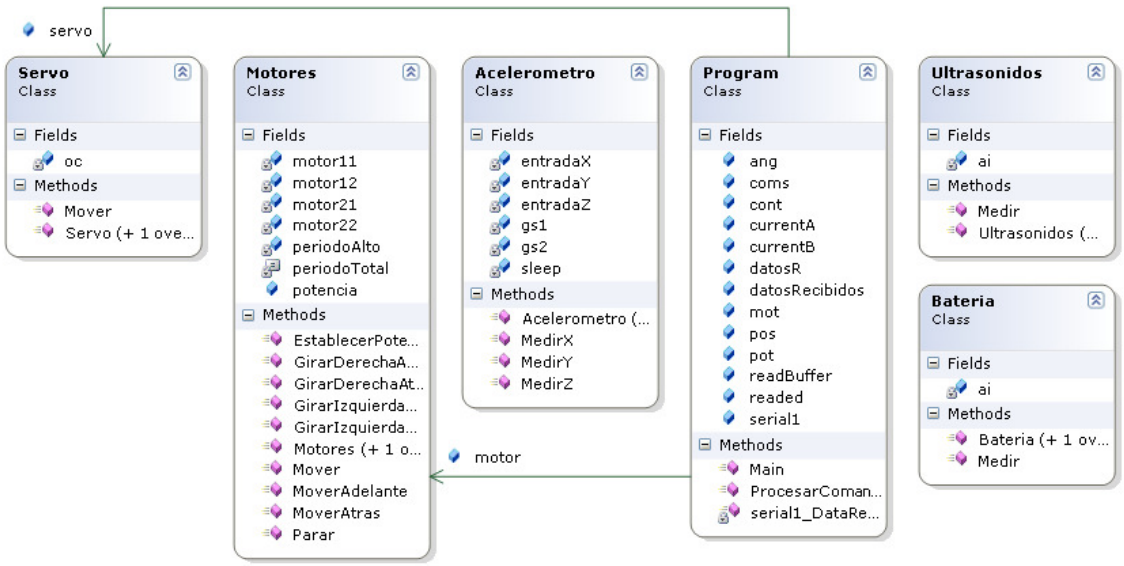

Fig. 4.3 Diagrama de clases del proyecto Net Zeppelin

Tras ver qué clases hay dentro del proyecto, toca saber para qué sirve cada una:

- **Program**: esta es la clase principal del programa. Aquí se encuentra el bucle de envío de datos a la estación base y el evento de recepción de datos por el puerto serie. También cuenta con una función (ProcesarComando) que se encarga de convertir los mensajes recibidos en parámetros interpretables por las clases Servo y Motores, y realiza llamadas a todas las demás clases para poder enviar los datos obtenidos por el dirigible y hacer que éste se mueva en función de las órdenes recibidas.

- **Bateria**: se trata de una clase muy simple, cuenta con un campo de tipo AnalogIn y una función de lectura de dicho puerto. Esta clase se utiliza para realizar las mediciones del nivel de batería, dato que se transmite más tarde a la estación de tierra.

- **Servo**: esta clase contiene un campo OutputCompare al cual se le asigna el pin digital que se le pase como parámetro al constructor de la clase. La clase dispone de una función que se encarga de mover el servo mediante la instrucción Set de la clase OutputCompare.

- **Motores**: en esta clase se encuentran 4 campos de tipo PWM (dos por motor, ya que estos disponen de dos entradas), un uint con un valor determinado para el periodo de los pulsos, un int que representa el porcentaje de potencia a utilizar y otro uint que guarda el tiempo exacto que permanecerá el pulso en estado (calculado mediante los dos campos anteriores con la función EstablecerPotencia). Las funciones de las que dispone, sin contar la mencionada antes, son 8: 7 que encienden según que pin o pines de los motores para los 7 movimientos diferentes que puede realizar el dirigible (MoverAdelante, MoverAtras, GirarIzquierdaAdelante, GirarIzquierdaAtras, GirarDerechaAdelante, GirarDerechaAtras y Parar) y una que llama a la función EstablecerPotencia y a la función que corresponda de las 7 anteriores según el comando recibido de la estación.

- **Ultrasonidos**: clase con una estructura igual que la de la clase Bateria: un campo de tipo AnalogIn y una función de lectura de ese mismo puerto. Mediante esta clase se obtiene la altura a la que vuela el zeppelín.

- **Acelerómetro**: esta clase dispone de 6 campos: 3 son de tipo AnalogIn, y se usan para obtener las medidas del acelerómetro en los 3 ejes; los otros 3 son de tipo OutputPort y se utilizan para activar o desactivar los pines de selección de sensibilidad y modo Sleep del sensor. Dispone de una función para obtener cada una de las 3 medidas que realiza el acelerómetro.

Este es el programa que lleva el dirigible almacenado en la placa FEZ Mini. Su estructura evita que el dirigible se cuelgue en mitad del vuelo, y las clases son simples gracias a la división por módulos que se ha hecho, que evitan grandes conjuntos de código y ayudan a comprender más fácilmente qué hace cada una de ellas.

4.4. Diseño de la placa de topos

Con el programa principal hecho es el momento de completar la góndola del zeppelín. Puesto que la placa FEZ Mini va a llevar conectados varios sensores y dispositivos, hace falta darles un soporte para que no queden suspendidos una vez se acople la placa al globo del dirigible. Por esta razón, se ha diseñado

y soldado una placa de topos que sostenga todos los componentes que conforman la góndola del zeppelín.

La siguiente figura, realizada con el programa EAGLE, muestra esquemáticamente la distribución de los componentes en la placa de topos:

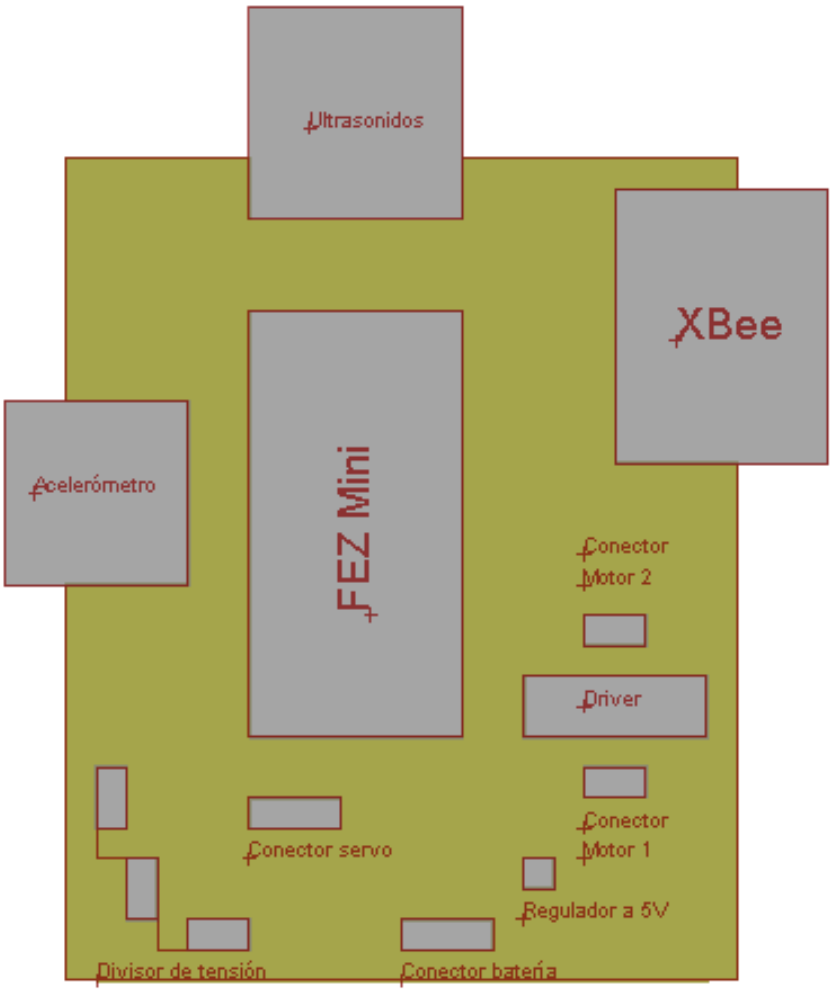

Fig. 4.4 Esquemático de la placa de topos del dirigible

Tanto la placa FEZ Mini como el acelerómetro o el XBee son componentes que pueden ser utilizados en infinidad de proyectos diferentes; sin embargo, soldar sus pines directamente sobre la placa de topos implicaría tener que sacrificar su carácter modular y tener que utilizarlos sólo para el funcionamiento del dirigible. Para evitar esta situación, se han soldado zócalos y pines sueltos (en función del tipo de conexión que lleve incorporado cada módulo) en la placa de topos. Así, todos los elementos serán extraíbles, permitiendo su uso en otros proyectos que se lleven a cabo al mismo tiempo y facilitando el reemplazo por nuevos módulos de mayor calidad si se dispone de ellos (siempre y cuando compartan el tipo de encapsulado, ya que si la distancia entre pines del sensor nuevo no es la misma que la del viejo hará falta cambiar el zócalo).

Además de estos elementos de conexión de los sensores y componentes, la placa cuenta con un regulador que, a partir de los 7,4 V que da la batería, ofrece una salida de 5 V utilizada por el driver para dar potencia a los motores; y también hay un divisor de tensión, gracias al cual se puede medir el nivel de batería (como la batería da 7,4 V y la alimentación interna de la FEZ Mini es de 3,3 V, hace falta un divisor de tensión que ofrezca un voltaje proporcional al de la batería pero adaptado al fondo de escala causado por la alimentación interna de la placa).

Las medidas de esta placa son mayores que la de Arduino. La original medía 41 mm x 41 mm, mientras que la placa de topos mide 66 mm x 95 mm. El peso de la góndola con esta placa es de unos 135 gramos, lo cual no supone problema alguno para el dirigible, ya que cuando se utilizaba la placa de Arduino hacía falta añadirle lastre; este lastre elevaba el peso total a casi 150 gramos y aún así el globo era capaz de mantener el dirigible suspendido en el aire.

4.5. Conclusiones del capítulo

A pesar de ser más grande, el tamaño de la nueva placa no supone un gran problema para el zeppelín, si bien hará falta llenar el globo con más helio que aire para que éste se mantenga suspendido a una altura constante.

Con esto se da por finalizada esta primera gran parte del proyecto. Con el programa del dirigible terminado y la placa de topos estructurada y montada, es hora de diseñar la estación de tierra.

5. CREACIÓN DE LA ESTACIÓN BASE

Ahora que el dirigible está terminado y cuenta con una conexión inalámbrica gracias al módulo XBee, sería útil poder recibir datos sobre su estado para controlar el buen funcionamiento del zeppelín y detectar posibles anomalías. Con esta finalidad se ha diseñado una estación base que muestre por pantalla la información recibida del dirigible a la vez que le envía de vuelta las directrices de movimiento que queremos que siga. Su desarrollo será explicado en este quinto capítulo del trabajo.

5.1. Diseño de la estación base

El propósito principal de la estación de tierra es el de controlar el estado del dirigible, incluyendo aspectos como el nivel de batería o la altura a la que vuela entre otros. Por lo tanto, será de agradecer poder comprobar dichos datos con un vistazo rápido, sobre todo para no perder de vista el zeppelín (esto se podría remediar incorporándole una cámara). A partir de esta idea, el diseño de la estación base se muestra en la siguiente figura, correspondiente a la estación ya completa:

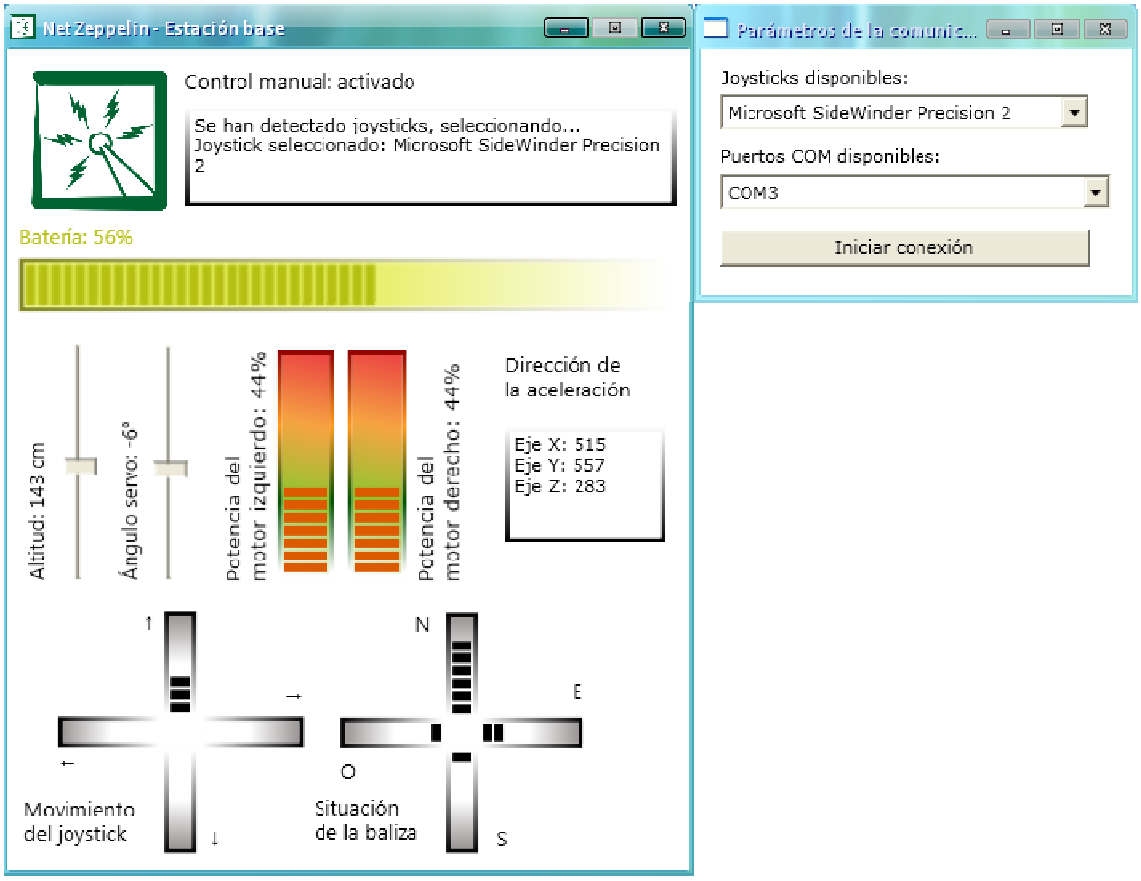

Fig. 5.1 Estación de tierra del Net Zeppelin

El que se muestra es un diseño simple y directo, sin demasiados adornos visuales. A primera vista se aprecian las barras de batería, temperatura y humedad (estas dos últimas añadidas por si surge la posibilidad de sumar los sensores necesarios para esas mediciones a los que ya hay en el dirigible), que además cambiarán su aspecto y su color a medida que adquieran diferentes valores para resaltar el significado de sus cifras. También resaltan dos cruces, una que representa las lecturas obtenidas del joystick, para comprobar que el dirigible responde como debe al movimiento de este, y otra que muestra la situación de la baliza de infrarrojos (perteneciente al sistema de piloto automático original del BlimpDuino, que no se ha añadido por ahora pero que se podría implementar en un futuro) respecto a la posición del zeppelín.

A parte de estos elementos, la estación también cuenta con unos indicadores de altitud y del ángulo del servo, otros tres que muestran el modo de funcionamiento activo, un indicador de los niveles de aceleración que está experimentando el dirigible y un cuadro de texto donde se muestran mensajes enviados por este en determinadas situaciones.

5.2. Programando la estación

Para construir la estación base se ha utilizado tanto Expression Blend como Visual Studio. El primero ha sido empleado para la parte gráfica, mientras que el cometido del segundo ha sido el de escribir las líneas de código necesarias para que la estación pudiese llevar a cabo ciertas acciones.

5.2.1. Estación base en Expression Blend

Como se comentó en el capítulo 2, Expression Blend hace uso del lenguaje XAML para crear aplicaciones de Windows Presentation Fundation. Es en estos ficheros donde se recogen las propiedades de cada uno de los elementos añadidos a la aplicación, las animaciones creadas por el usuario y otros aspectos. El método para la creación de la aplicación es muy simple (arrastrar los elementos deseados al formulario para que se genere automáticamente el código en el fichero XAML y cambiar las propiedades deseadas desde ahí o desde la barra de propiedades de Expression Blend), pero hay un par de características utilizadas en el proyecto que cabe mencionar.

5.2.1.1. Templates

A veces puede ocurrir que la aplicación que se está diseñando necesite que los elementos que en ella se muestran guarden cierto parecido externo (el color de fondo, el de los bordes…), ya sea por una cuestión estética o por cualquier otra razón. Los “templates” (“plantillas” en castellano) de Expression Blend permiten guardar una misma configuración visual aplicable a todos los elementos del mismo tipo que el de la plantilla editada que se desee.

Para ver como funcionan se utilizará como ejemplo las barras que forman la cruz de estado del joystick.
Hay dos maneras de hacer una nueva plantilla: utilizando una existente o creándola desde cero. Esta última opción es perfecta para cambiar crear controles inexistentes en Expression Blend, ya que al estar la plantilla vacía se pueden añadir todos los elementos necesarios para crear un control exclusivo en función de los gustos del usuario. Sin embargo, para cambiar solo algunos aspectos gráficos (como es el caso de este proyecto), es recomendable editar una plantilla ya existente.
En este ejemplo, la plantilla a editar corresponde a la de una barra de progreso. Para que su aspecto vaya en consonancia con el del resto de la aplicación será necesario cambiar el grosor del borde y los colores de toda la barra. Como se puede observar, la plantilla utiliza un "Border" y dentro de él introduce un elemento de tipo "ProgressBar". Para cambiar el aspecto del borde se debe modificar la propiedad "BorderBrush" del primer elemento, mientras que para el fondo y el color de la propia barra hay que cambiar los parámetros "Background" y "Foreground" del segundo. Aunque en este caso no es el objetivo que se busca, también sería posible dejar los campos que se deseen con la misma apariencia que en su origen. Esto se consigue mediante la opción "Template Binding", que hará referencia al campo seleccionado de la plantilla original.
Ahora, con la plantilla ya acabada, es posible aplicarla a cualquier "ProgressBar". Para ello, solo es necesario hacer click derecho sobre la barra cuyo aspecto queremos cambiar, y en el menú emergente que aparece, seleccionar "Edit Template > Apply Resource > Nombre_de_la_plantilla".
Gracias a los "Templates" es posible cambiar el aspecto de varios objetos del mismo tipo a la vez sin necesidad de ir editándolos de uno en uno, permitiendo dar cierta uniformidad visual a los elementos de la aplicación en menos tiempo.

5.2.1.2. Animaciones

Cuando se está utilizando una aplicación en la que todo permanece con las mismas formas o parecidas, una manera muy útil de llamar la atención del usuario es mediante una animación. Un elemento que se mueve, un cambio de colores… Todo vale para captar la atención del usuario cuando la situación lo requiere. Expression Blend ofrece una manera sencilla y con bastante potencial para crear animaciones. Se trata de los Storyboards.
Los Storyboards son animaciones creadas a partir de un estado inicial y uno final de un elemento cualquiera, cuya transición es creada por el propio programa. Las animaciones pueden ir desde cambios en los colores de los elementos hasta que estos se muevan hasta un punto de la pantalla o que desaparezcan y vuelvan a aparecer. Cualquier cambio que se pueda hacer moviendo los elementos o cambiando propiedades puede ser registrado como una animación para crear una aplicación dinámica.
El ejemplo que ayudará a ilustrar el proceso de creación de Storyboards consiste en hacer que la barra de batería parpadee cuando se encuentra en un estado de bajo nivel.

El primer paso es crear un nuevo Storyboard mediante la opción de la barra de Storyboards. Se le otorga un nombre, como por ejemplo “StoryPrueba”, y automáticamente se crea uno completamente vacío.

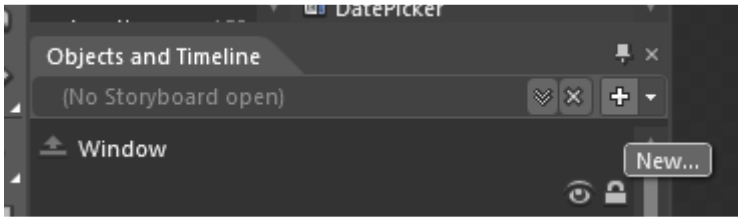

Fig. 5.2 Barra de storyboards, donde se muestran los ya existentes y se pueden crear nuevos

Como la intención es que la barra de batería aparezca y desaparezca, la animación consta de dos pasos:

- Hacer que la barra pase de ser visible a no serlo
- Hacer que vuelva a ser visible y repetir el proceso anterior

Para ello, lo que hay que hacer es seleccionar la barra de batería, situarse sobre el segundo 0 del “Timeline” y hacer click sobre la opción “Record Keyframe”. Esto establecerá el estado actual de la barra como el estado en el que deberá estar al principio de la animación.

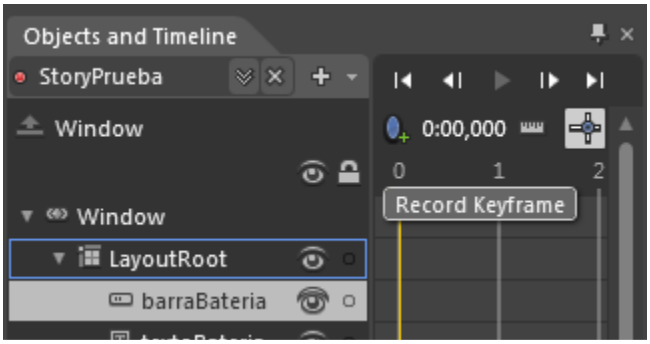

Fig. 5.3 Pantalla de grabación y edición de storyboards

Ahora solo queda hacer lo propio con el estado final de la animación, es decir, que la barra no se vea (no hace falta que se vuelva a ver por una razón que se verá más adelante). Para ello, con la barra de batería seleccionada, hay que situarse sobre el segundo final de la animación (unos 2 segundos ya están bien), bajar la visibilidad de la barra al 0% y seleccionar “Record Keyframe”.
A continuación se selecciona el nombre del Storyboard que se está editando y, en la barra de propiedades, se seleccionan las opciones “AutoReverse” y “Forever” de RepeatBehavior. De esta manera no hace falta crear la parte de la animación que devuelve la barra a su estado inicial, ya que la opción de

"AutoReverse" lo hace por sí sola. Además, la opción "Forever" hace que la animación se repita indefinidamente mientras ésta esté activa.
En el proyecto se han usado los Storyboards para realizar los cambios gráficos en las barras de batería, temperatura y humedad y en los indicadores de modo de funcionamiento. La adición de estas animaciones hace la aplicación más atractiva visualmente y no representa un gran problema, puesto que el método de creación de Storyboards es muy intuitivo.

5.2.2. Estación base en Visual Studio

Tras crear la estación base y añadir todos los elementos visuales que se precisen, le toca el turno a la programación propiamente dicha. Puesto que se ha utilizado Expression Blend para el diseño de la interfaz gráfica, la estación base se ha guardado en una solución que puede abrirse desde Visual Studio, y es ahí donde se escribirá el código necesario para el buen funcionamiento de la estación.

La versión de .NET Framework utilizada para la estación es la 4, la más reciente. La estación de tierra cuenta con dos clases, la de la ventana principal y la del joystick, que crea una interfaz para poder seleccionar un joystick conectado al ordenador para poder utilizarlo en el control del dirigible.

5.2.2.1. La clase principal

En la clase principal del programa está el código para todos los cambios que se han de realizar en la interfaz de la estación base, como los cambios visuales en la barra de batería o mostrar la inclinación del joystick. Además, también están las rutinas de envío y recepción de información. El dirigible necesita estar recibiendo directrices constantemente para poder mantener la posición del servo o el sentido en el que giran las hélices. Para ello, se ha utilizado un objeto de tipo DispatcherTimer.

La clase DispatcherTimer proporciona un temporizador que, tras recorrer cierto intervalo de tiempo establecido por el usuario, llama a una función específica. Esta función consiste en tomar los valores de salida necesarios del joystick, enviarlos mediante el puerto serie al zeppelín y actualizar una parte de la estación de tierra que muestra la inclinación del joystick. Para poder llevar a cabo el envío de datos sería suficiente con un Timer corriente, pero el hecho de actualizar la interfaz requiere el uso del DispatcherTimer. Si no se utiliza este último, la interfaz se acabará congelando, ya que pertenece a un Thread diferente, y al Timer normal no le da tiempo a refrescarla. Siempre que se esté realizando una aplicación que utilice WPF y sea necesario un temporizador, la única salida posible es utilizar el DispatcherTimer.

Utilizando objetos de tipo Dispatcher (esta vez sin necesidad de usar un temporizador) se renuevan el resto de elementos de la interfaz cuando estas actualizaciones son requeridas a través de una función que no pertenece al

programa principal (y, por lo tanto, no pertenece al mismo Thread que la interfaz de usuario).

Por último, la estación base es la encargada de realizar los cálculos necesarios para el movimiento del dirigible. Estos son el ángulo que debe tomar el servo o el sentido de rotación de los motores. Realizar esta serie de cálculos no hace sino ralentizar el procesado de los comandos recibidos en el zeppelín, provocando fallos en el sistema que resulten en un vuelo poco fluido e incluso en un paro completo de la placa del dirigible.

5.2.2.2. La clase JoystickInterface

La segunda clase que conforma la estación base es la clase JoystickInterface. Esta clase está escrita por Mark Harris y es necesaria para poder utilizar el joystick en esta o cualquier otra aplicación que lo requiera.

El funcionamiento de la clase es sencillo. La función FindJoysticks, como su propio nombre indica, se encarga de detectar todos los joysticks que haya conectados al ordenador; para poder seleccionar uno y referenciarlo a la variable de tipo JoystickInterface que se haya creado, se debe llamar a la función AcquireJoystick y pasarle como parámetro el nombre de cualquiera de los controladores detectados mediante la función descrita anteriormente.

El reconocimiento de la posición del joystick se establece mediante la función UpdateStatus. Esta función obtiene las posiciones de los ejes y los estados de los botones del joystick en el momento en que se realiza la llamada. Para luego obtener dichos datos lo único que hace falta es acceder a los diferentes parámetros del objeto JoystickInterface, como AxisA (eje horizontal), AxisB (eje vertical), Slider (en el caso del joystick utilizado en el proyecto, la palanca auxiliar) o Buttons (devuelve un array con el estado de los botones, “true” si están pulsados o “false” si no es así).

La clase también ofrece funciones para detener o pausar el funcionamiento del joystick. Igual que se dispone de una función UpdateStatus para actualizar las variables del objeto JoystickInterface y así poder leer los valores de los ejes o los botones del controlador, también hay disponible una función contraria, llamada StopUpdateJoystick, que termina el thread principal de la clase. Además, también es posible desconectar el joystick de la aplicación y dejarlo libre para su uso en otra. Esto se hace mediante la función ReleaseJoystick.

Como se puede ver, aunque la clase parece simple y con pocas funciones, ofrece todas las necesarias para poder utilizar correctamente cualquier joystick más o menos sencillo. En el caso de este proyecto, la clase ofrece parámetros de sobra, ya que tan sólo necesita 3 ejes: el horizontal, el vertical y el de la palanca auxiliar.

5.2.3. Incluyendo el joystick en la estación

Una vez diseñada la estación de tierra, tan sólo hace falta un último elemento para que esté completa, y esa pieza es el joystick. Este dispositivo es vital para poder manejar el dirigible, ya que éste ya no cuenta con la comunicación por radio que ofrecían el mando del BlimpDuino y su receptor.

5.2.3.1. Prueba del joystick

Antes de utilizar el dirigible en la estación base un aspecto que hace falta conocer para poder programar luego las funciones de movimiento del zeppelín. Se trata de los valores de salida del joystick. Para poder establecer los valores umbral entre los que decidir, por ejemplo, si cierta inclinación del joystick indica que el dirigible debe ir hacia adelante o hacia atrás, hace falta saber cuáles son esos valores y cuál es el origen de coordenadas a partir del cual se miden.

Para ello se ha utilizado un código sencillo que lo único que hace es leer las salidas de los diferentes ejes y de la palanca (que se utilizará para el servo) del dispositivo. Este código utiliza una clase llamada JoystickInterface, que junto a las librerías de DirectX permite listar todos los joysticks conectados al ordenador, seleccionar uno de ellos y obtener los datos que envía. Si bien es posible leer tanto los ejes horizontal y vertical como los de rotación además de los botones, el programa de prueba sólo lee los dos primeros y la palanca auxiliar, que son las partes que se utilizarán.

El programa de prueba es el siguiente:

```
static void Main(string[] args)
    {
        JoystickInterface.Joystick device = new JoystickInterface.Joystick();
        string[] deviceList = device.FindJoysticks();

        foreach (string s in deviceList)
            Console.WriteLine(s);

        device.AcquireJoystick(deviceList[0]);

        do
        {
            device.UpdateStatus();
            Thread.Sleep(100);
            Console.Clear();
            Console.WriteLine("Eje X: " + device.AxisA.ToString());
            Console.WriteLine("Eje Y: " + device.AxisB.ToString());
            Console.WriteLine("Eje Z: " + device.AxisC.ToString());
            Console.WriteLine("Slider: " + device.Slider.ToString());
            Console.WriteLine("");
        } while (true);
    }
```

El joystick SideWinder Precision 2 ofrece valores que van desde 0 hasta 65535, tanto para el movimiento del stick como el de la palanca auxiliar. El punto de inicio de la palanca es la inclinación total hacia arriba, es decir, cuando está inclinada en esa posición la salida es 0, cuando está hacia abajo es 65535, y en el punto medio es 32767. El stick principal, pos su parte, tiene su centro de

coordenadas en la posición superior izquierda. Así, para el eje horizontal, la salida del dispositivo es 0 en la izquierda y 65535 en la derecha, mientras que para el eje vertical las salidas son iguales que con la palanca auxiliar.

Gracias a esta prueba se pueden establecer los valores mínimos para cada una de las acciones que pueda realizar el dirigible. En la siguiente tabla se detallan los valores de cada eje y cuál sería la orden a seguir por el zeppelín:

Tabla 5.1. Valores del joystick y correspondiente orden para el dirigible

Eje horizontal	Eje vertical	Orden
16384 < X < 49152	X < 32767	Mover hacia adelante
16384 < X < 49152	X > 32767	Mover hacia atrás
X < 16384	X < 32767	Giro a la izquierda (hacia adelante)
X > 49152	X < 32767	Giro a la derecha (hacia adelante)
X < 16384	X > 32767	Giro a la izquierda (hacia atrás)
X > 49152	X > 32767	Giro a la derecha (hacia atrás)

Además de establecer estos valores, también hace falta concretar una fórmula que adapte los valores de salida del joystick a los ángulos correspondientes del servo. Dicha fórmula se establece gracias a los valores máximos obtenidos del joystick, y a continuación se explica ese cálculo que hay que hacer a partir del valor máximo de la palanca auxiliar del joystick.

El servo cubre un arco de 180°, por lo que si la salida máxima del stick es 65535, para cada grado tiene que haber una variación de unas 364 unidades (65535 / 180). Por lo tanto, como el programa del dirigible está estructurado en módulos y la función que mueve el servo recibe como argumento la salida directa del joystick, hace falta realizar la siguiente conversión para obtener el ángulo exacto:

$$\text{Ángulo} = (65535 - \text{salida del joystick}) / 364.08333333333 \qquad \textbf{(5.1)}$$

Sin embargo, sólo con esto no se puede mover el servo. Puesto que utiliza la clase OutputCompare, hace falta darle un entero como parámetro, que representa los µs que se mantendrá el pulso en estado alto, que será el que determine el ángulo a girar por el servo.

Gracias a las pruebas hechas en la placa Embedded Master (capítulo 3, apartado 4) se ha podido establecer el margen de tiempos entre 800 µs y 1700 µs, es decir, 900 µs entre valor mínimo y máximo. De esta manera, los 900 µs repartidos entre los 180° que cubre el servo dan un total de 5 µs entre cada

grado. Sin embargo, la fórmula no es tan fácil como multiplicar 5 por el ángulo deseado, ya que de esta manera el margen de tiempos iría de los 0 µs a los 900 µs. Por lo tanto, la fórmula final queda de la siguiente manera:

$$\text{Tiempo en estado alto} = \text{Ángulo} * 5 + 800 \qquad \textbf{(5.2)}$$

Ahora, con ese tiempo calculado en µs, se hace una llamada a la función Set del objeto OutputCompare, pasándole como parámetros el tiempo calculado y la diferencia entre 20 ms y el periodo en estado alto (siempre en ms), además de otros datos que necesita.

5.2.3.2. Estructura de envío al dirigible

A través de la fórmula calculada en el apartado anterior, se puede obtener el ángulo que debe girar el servo. Sin embargo, este no es el único dato que necesita el dirigible para su navegación. También hace falta decirle que motor (uno o los dos) hace falta mover, en qué sentido tiene que hacerlo y a qué velocidad.

Sería conveniente enviarle a la placa cuantos menos caracteres posibles, ya que de esta manera se reducen el tiempo que estará procesando el mensaje y las posibilidades de que salte una excepción en el código y deje de funcionar. El ángulo del servo depende de un número, que se corresponde con la posición de la palanca auxiliar, y la orden para los motores se basa en dos valores, los ejes horizontal y vertical del stick de control. Sin embargo, sería más fácil calcular, mediante esos dos valores, qué orden enviar a los motores. Esto se puede conseguir estableciendo un carácter específico para cada una de las posibles órdenes que se podría enviar a los motores. En la siguiente tabla se muestra la correspondencia entre las órdenes y el carácter enviado:

Tabla 5.2. Asignación de caracteres a las diferentes ordenes para el dirigible

Orden	Carácter asignado
Mover hacia adelante	F
Mover hacia atrás	B
Giro a la izquierda (hacia adelante)	I
Giro a la derecha (hacia adelante)	D
Giro a la izquierda (hacia atrás)	L
Giro a la derecha (hacia atrás)	R

En el caso de la potencia que se administrará a los motores, también hace falta asignar algún código. Como se pudo ver en el capítulo anterior, la potencia que dará el driver varía en función del estado alto de los pulsos que se le mandan. De esta manera, cuanto más largo sea el tiempo en estado alto, los motores operarán con más intensidad.

Una manera fácil de convertir la posición del joystick a un parámetro que represente la potencia a suministrar es convertir esa salida del joystick en un porcentaje. De esta manera se podrá utilizar este mismo número para calcular en el zeppelín la duración del estado alto del pulso en función de su periodo total.

En la siguiente tabla se muestra la correspondencia entre los valores máximos y mínimos del joystick para cada comando y sus respectivos porcentajes:

Tabla 5.3. Valores de salida del joystick y porcentajes de potencia correspondientes

Porcentaje	Comando	Salida del joystick
0%	I, D, L, R	Eje X: 32767
	F, B	Eje Y: 32767
100%	F	Eje Y: 0
	B	Eje Y: 65535
	I, L	Eje X: 0
	D, R	Eje X: 65535

Gracias a esta tabla se pueden establecer un par de fórmulas que permitirán convertir los niveles del joystick en sus respectivos porcentajes. La primera fórmula será para calcular la potencia cuando el máximo porcentaje se produce con la salida mínima del joystick, es decir, para los comandos F, I y L. La fórmula utilizada para calcular el porcentaje de potencia en función de la salida del joystick para estos comandos es la siguiente:

$$\text{Porcentaje} = 100 - (\text{salida del joystick} / 327{,}66) \qquad \textbf{(5.3)}$$

Para estos casos, al aumentar la salida del joystick (es decir, al acercarlo hacia su posición central), el porcentaje se irá reduciendo, yendo del 100% al estar en los extremos izquierdo o superior, al 0% por estar en el centro. Como se puede ver a partir de la tabla, aquí la salida del joystick irá de 0 a 32766.

La otra fórmula sirve para los casos en que el máximo porcentaje se produce con la máxima salida del joystick, que serán los de los comandos B, D y R. Esta vez la fórmula es justamente al revés que la anterior:

$$\text{Porcentaje} = (\text{salida del joystick} / 327{,}66) - 100 \qquad \textbf{(5.4)}$$

En esta situación, aumentar la salida del joystick (es decir, pasar de la posición central a la derecha o abajo) hará aumentar el porcentaje de potencia, aproximándose al 100% a medida que la posición del joystick se acerca a la parte inferior o la derecha. Ahora, la salida irá de 32768 a 65535.

Una vez decididos los caracteres para cada comando y los valores para diferenciar la potencia deseada, hace falta establecer una cadena en la que añadir ambos datos, ángulo del servo y comando para los motores. La cadena utilizada es la siguiente:

000#F#100! **(5.5)**

En el mensaje anterior, “000” se corresponde con el ángulo del servo, que puede ir de 0 a 180. Se ha establecido por código que el ángulo ocupe siempre 3 dígitos, incluso cuando el ángulo sólo llega a las decenas o las unidades. Esto se debe a ciertos parámetros de recepción de la placa del dirigible, que facilitan la recepción.

La F, como se puede deducir gracias a la tabla anterior, ocupa la posición de la orden recibida por el zeppelín. A partir del carácter que reciba, la placa FEZ Mini encenderá según qué terminales de los motores para llevar a cabo el movimiento deseado.

Por último, el número del final de la sentencia es el que expresa la marcha del dirigible, y por tanto, qué potencia se suministrará a los motores.

La almohadilla se utiliza para separar los datos y facilitar el filtrado en la FEZ Mini. Así, tras almacenar el comando completo y realizar un Split utilizando la almohadilla como separador, se obtiene un array de tres elementos, siendo uno el ángulo del servo, otro la letra que necesitan los motores para determinar cuáles girarán y el sentido en el que deben hacerlo, y el último la potencia que el driver le tiene que administrar a los motores. La exclamación del final realiza la función de marcar el fin de la línea, de manera que sólo cuando el comando recibido incluya una exclamación se puede pasar al procesado del mensaje y, posteriormente, a las funciones de movimiento del dirigible.

5.3. Conclusiones del capítulo

Gracias al establecimiento de esta estructura para los mensajes enviados al dirigible se consigue enviar las órdenes necesarias en una línea de longitud muy reducida, lo cual beneficiará a la estabilidad en el funcionamiento del dirigible evitando agotar la memoria, como ya se ha comentado anteriormente.

Tras implementar esta última parte de la estación base, el proyecto ya está completo. El dirigible puede volar gracias a las directrices enviadas desde la estación de tierra y ésta muestra al usuario los datos obtenidos por el zeppelín. Ahora que el proyecto está finalizado es hora de hacer un pequeño estudio sobre el impacto medioambiental del trabajo y un repaso de las conclusiones a las que se han llegado durante su elaboración.

6. ESTUDIO DE AMBIENTALIZACIÓN Y CONCLUSIONES

Con el capítulo anterior se ha explicado la segunda gran parte del proyecto, y este documento ha expuesto diferentes explicaciones sobre el material utilizado, el funcionamiento de las aplicaciones creadas y la estructura del código de éstas. Sin embargo, queda un último punto antes de exponer mis conclusiones personales, y es realizar un pequeño estudio ambiental sobre el proyecto. En este último capítulo se expondrá ese apartado sobre la ambientalización del trabajo, seguido de las diferentes conclusiones a las que he llegado en relación al proyecto, los objetivos alcanzados y los aspectos que se podrían mejorar o hacia los que se podrían enfocar futuras revisiones del trabajo.

6.1. Estudio de la ambientalización

En los tiempos que corren es tan importante la calidad de un proyecto como el hecho de que no suponga grandes costes, tanto monetarios como medioambientales. A la hora de diseñar un proyecto un ingeniero ha de tener en cuenta el impacto medioambiental de su trabajo, y cada día se tiene más en cuenta este aspecto de la ingeniería. Por ello, este trabajo también cuenta con un pequeño apartado en el que se intentará analizar el impacto ambiental del proyecto y si podría haber alternativas para reducirlo.

De entrada, el dirigible no altera de ninguna manera el entorno. No realiza vertidos de ningún tipo, no contiene materiales tóxicos y la comunicación inalámbrica no afecta de ninguna manera al medio. Aún así, hay algunos elementos que podrían presentar algún problema en determinadas situaciones y son los siguientes:

- Helio del globo
- Batería LiPo
- Placa de topos
- Piezas de plástico de la góndola

El helio es un gas noble no tóxico. Una manera (posiblemente la única) de que hubiese algún problema relacionado con el helio sería una fuga del globo del dirigible. Este gas sólo podría representar un problema en el ámbito de la salud del usuario, ya que una gran ingesta de dicho gas puede provocar asfixia por reemplazo de oxigeno. Aparte de este no hay ningún otro problema que se pudiese derivar de un escape de helio, por lo que el impacto medioambiental es nulo.

La batería del zeppelín es de polímero de litio (LiPo). Estas baterías ofrecen más tiempo de autonomía que las de ion de litio (Li-ion) o las de níquel-cadmio (Ni-Cd), aunque no pueden ser descargadas del todo, ya que si no quedan

inservibles. En este caso el problema viene en el momento de tirarlas porque ya no se pueden utilizar más, inconveniente recurrente en baterías y pilas.

La placa de topos es el elemento más problemático en potencia. La placa de topos está hecha de baquelita, un plástico de tipo termoestable. Este tipo de plásticos son infusibles e insolubles, y si bien presentan ventajas en cuanto a su resistencia a los golpes frente a los termoplásticos (deformables a temperatura ambiente), son mucho más difíciles de reciclar.

Por último, hay que recordar que la góndola del zeppelín se sostiene sobre una lámina de plástico y dispone de unas cuantas piezas de Lego para la sujeción de los motores y para poder aplicar la rotación del servo a estos. Estas piezas están hechas de ABS, otro plástico de tipo termoestable como la baquelita, y el material de la base de la góndola, aunque no es especificado por los creadores del BlimpDuino, seguramente también lo sea. Esto, al igual que en el caso anterior, ofrece ventajas respecto a otros plásticos como una mayor resistencia a los golpes y demás características propias de los termoestables, pero dificulta su reutilización.

Por lo tanto, en estos 3 últimos casos el inconveniente siempre es el mismo: la dificultad del proceso de reciclaje de dichos materiales o componentes. Por suerte, hoy en día no es difícil encontrar instalaciones de recogida y reciclado de residuos para los que no se dispone de contenedores. La siguiente figura, extraída de http://www.ecolec.es/puntoslimpios.asp, muestra la situación de los 956 centros de recogida selectiva que hay repartidos por todo el estado español:

Fig. 6.1 Mapa con la situación de los centros de recogida selectiva españoles

En estos lugares también se pueden depositar baterías, elementos electrónicos y otros plásticos, por lo que, para el usuario, es fácil deshacerse de estos elementos sin perjudicar el medio ambiente.

En resumidas cuentas, el proyecto Net Zeppelin no perjudica al medio ambiente más allá del gasto energético empleado en el reciclaje de los materiales y el combustible utilizado en poder acceder a una de estas instalaciones en el caso de que el usuario no disponga de una cerca de su domicilio.

6.2. Conclusiones del proyecto

La realización de este proyecto, las fases por las que ha pasado su desarrollo y los imprevistos que han ido surgiendo durante el trabajo me han llevado a ciertas conclusiones que me dispongo a exponer en este apartado.

La primera de todas ellas es que a pesar de que este proyecto haya podido tener más o menos complicaciones, se trata de un proyecto que cualquier alumno de la universidad, si dispone del capital para obtener el material, podría llevar a cabo personalmente, sin tener que ser un trabajo académico. La base del trabajo ha sido un kit de montaje de un dirigible operado mediante Arduino que se vende por Internet. La nueva placa que se ha utilizado para el cambio de Arduino a C# también se puede comprar por Internet, y su coste no es muy elevado. El resto de componentes se pueden adquirir en tiendas especializadas o también en la red; su coste es otro tema. Los programas utilizados para el desarrollo de las aplicaciones es el mismo que se ha podido utilizar en anteriores asignaturas sobre programación de la carrera, y cualquier estudiante sabe cómo funcionan.

Este es un aspecto que me atrae mucho de este tipo de proyectos: cualquiera, con ciertos conocimientos, puede desarrollarlos. Desde luego, la calidad final dependerá de los niveles de conocimientos de cada uno, pero pienso que este tipo de proyectos atraen a la gente que no está interesada en la ingeniería o la programación hacia este terreno, introduciéndolos poco a poco en el montaje de circuitos y el desarrollo de programas. Además, este tipo de trabajos, una vez finalizados, le otorgan al usuario una mayor sensación de haber acabado algo tangible, algo que puede manipular y con lo que puede estar experimentando después de todo el tiempo que ha empleado en el montaje y el desarrollo.

Otro de los aspectos que me gustaría destacar del proyecto es la variedad del mismo. Gracias a haber decidido cambiar por completo el BlimpDuino original para que pudiese funcionar con otra placa, con un lenguaje de programación diferente y con nuevos elementos de hardware, el trabajo no se ha centrado en programar o en el montaje de una placa, sino que ha tenido parte de los, lo cual lo ha convertido en un proyecto bastante completo en el que siempre ha habido algo que hacer.

Desde mi opinión personal, estoy satisfecho con el resultado final del trabajo. Pese a que la base del proyecto era el BlimpDuino, nada del código original se ha utilizado para el trabajo, todo se ha reescrito desde 0. A veces se ha consultado el código del BlimpDuino para ver cómo se ejecutaba alguna acción, pero el código se ha escrito teniendo en mente cómo debería comportarse un zeppelín, lo cual ha convertido el proyecto en un trabajo duro pero muy satisfactorio.

El próximo paso sería añadirle ese piloto automático que se mencionaba en la introducción para disponer de un dirigible capaz de moverse automáticamente o manualmente según lo requiera la situación, funcionalidad con la que el BlimpDuino cuenta pero que por ahora no se ha podido implementar en el Net Zeppelin.

Otra buena idea sería elaborar una placa PCB para el dirigible. Como el objetivo del proyecto no era la creación de una placa perfecta, el ensamblaje de piezas se ha acabado realizando en una placa de topos, que aunque funciona perfectamente, no es la mejor opción tanto estética como práctica. Crear una PCB reduciría el tamaño de la placa y el peso que tiene que sostener el globo del dirigible, reduciendo la cantidad de helio que necesita para flotar.

Por último, sólo quedaría buscar reemplazos que pudiesen mejorar el rendimiento del zeppelín para el resto de componentes: una batería de mayor capacidad, motores más potentes o que admitan velocidades bajas, un servo que gire en menos tiempo, y así con cada uno de los elementos que forman el dirigible.

BIBLIOGRAFÍA

[1] Podila, P.; Hoffman, K., *WPF control development unleashed: building advanced user experiences,* Pearson Education, (2009).

[2] Nathan, A., *WPF 4 unleashed,* Pearson Education, (2010).

[3] http://www.ghielectronics.com/
Página web oficial del fabricante de la placa FEZ Mini, donde se pueden encontrar documentos, software, foros de consulta, etc.

[4] http://www.microsoft.com/latam/expression/solutions/Default.aspx
Página web oficial de Microsoft Expression Studio con información general sobre el peograma.

[5] http://www.tinyclr.com/
Comunidad de usuarios de .NET Micro Framework donde recopilar información, tutoriales y opiniones sobre productos de GHI, y más concretamente sobre s línea FEZ.

[6] http://msdn.microsoft.com/es-es/
Página oficial de Microsoft Developer Network donde se puede encontrar información extensa de todas las clases y librerías utilizadas en Visual Studio.

ANEXOS

Código del Net Zeppelin:

Program.cs

```
using System;
using System.Threading;
using System.Resources;
using System.Text;
using System.IO.Ports;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;
using Microsoft.SPOT.Presentation.Controls;
using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.Hardware;

namespace NetZeppelin
{
   public class Program
   {
      public static SerialPort serial1;
      public static Servo servo = new Servo(FEZ_Pin.Digital.Di9);
      public static Motores motor = new Motores(FEZ_Pin.PWM.Di10, FEZ_Pin.PWM.Di3,
FEZ_Pin.PWM.Di5, FEZ_Pin.PWM.Di6);
      public static int currentA = 32767;
      public static int currentB = 32767;
      public static int cont = 0;
      public static byte[] readBuffer;

      //serialdata received
      public static string datosRecibidos;
      public static int readed;
      public static char[] datosR;

      public static string[] coms;
      public static int pos;
      public static int ang;
      public static string mot;
      public static int pot;

      public static void Main()
      {
         Debug.EnableGCMessages(true);

         Ultrasonidos ultrasonidos = new Ultrasonidos(FEZ_Pin.AnalogIn.An3);
         Acelerometro acelerometro = new Acelerometro(FEZ_Pin.AnalogIn.An0, FEZ_Pin.AnalogIn.An1,
FEZ_Pin.AnalogIn.An2, FEZ_Pin.Digital.An4, FEZ_Pin.Digital.An5, FEZ_Pin.Digital.An6);
         Bateria bateria = new Bateria(FEZ_Pin.AnalogIn.An7);
         OutputPort led = new OutputPort((Cpu.Pin)FEZ_Pin.Digital.LED, true);
         byte[] bts;
         int bat;
         int acX;
         int acY;
         int acZ;
         int ult;


         readBuffer = new byte[100];

         int cont = 0;
```

```
serial1 = new SerialPort("COM3", 9600, Parity.None, 8, StopBits.One);
serial1.Open();
serial1.DataReceived += new SerialDataReceivedEventHandler(serial1_DataReceived);

while (true)
{
   if (cont == 0)
   {
      bat = bateria.Medir();
      if (bat < 10)
      {
         bts = Encoding.UTF8.GetBytes("00" + bat + "B!");
      }
      else if (bat < 100)
      {
         bts = Encoding.UTF8.GetBytes("0" + bat + "B!");
      }
      else
      {
         bts = Encoding.UTF8.GetBytes(bat + "B!");
      }
      cont++;
   }
   else if (cont == 1)
   {
      acX = acelerometro.MedirX();
      if (acX < 10)
      {
         bts = Encoding.UTF8.GetBytes("00" + acX + "X!");
      }
      else if (acX < 100)
      {
         bts = Encoding.UTF8.GetBytes("0" + acX + "X!");
      }
      else
      {
         bts = Encoding.UTF8.GetBytes(acX + "X!");
      }
      cont++;
   }
   else if (cont == 2)
   {
      acY = acelerometro.MedirY();
      if (acY < 10)
      {
         bts = Encoding.UTF8.GetBytes("00" + acY + "Y!");
      }
      else if (acY < 100)
      {
         bts = Encoding.UTF8.GetBytes("0" + acY + "Y!");
      }
      else
      {
         bts = Encoding.UTF8.GetBytes(acY + "Y!");
      }
      cont++;
   }

   else if (cont == 3)
   {
      acZ = acelerometro.MedirZ();
      if (acZ < 10)
      {
         bts = Encoding.UTF8.GetBytes("00" + acZ + "Z!");
      }
      else if (acZ < 100)
      {
```

```
                bts = Encoding.UTF8.GetBytes("0" + acZ + "Z!");
            }
            else
            {
                bts = Encoding.UTF8.GetBytes(acZ + "Z!");
            }
            cont++;
        }

        else
        {
            ult = ultrasonidos.Medir();
            if (ult < 10)
            {
                bts = Encoding.UTF8.GetBytes("00" + ult + "U!");
            }
            else if (ult < 100)
            {
                bts = Encoding.UTF8.GetBytes("0" + ult + "U!");
            }
            else
            {
                bts = Encoding.UTF8.GetBytes(ult + "U!");
            }
            cont = 0;
        }
        Debug.Print(cont.ToString()); ;
        serial1.Write(bts, 0, bts.Length);
        bts = null;
        Thread.Sleep(50);
    }
}

static void serial1_DataReceived(object sender, SerialDataReceivedEventArgs e)
{
    // Check if Chars are received
    if (e.EventType == SerialData.Chars)
    {
        lock (readBuffer)
        {
            readed = serial1.Read(readBuffer, 0, System.Math.Min(serial1.BytesToRead, 99));
            readBuffer[readed] = 0;
            datosRecibidos = datosRecibidos + new string(Encoding.UTF8.GetChars(readBuffer));

            int position;

            while ((position = datosRecibidos.IndexOf('!')) != -1)
            {
                String mensaje = datosRecibidos.Substring(0, position + 1);
                if (mensaje.Length > 10)
                {
                    Debug.Print("KK");
                }
                else
                {
                    ProcesarComando(mensaje);
                }
                datosRecibidos = datosRecibidos.Substring(position + 1);
            }
        }
    }
}

public static void ProcesarComando(string com)
{
    coms = com.Split('#');
```

```
            ang = Convert.ToInt32(coms[0]);
            mot = coms[1];
            motor.potencia = Convert.ToInt32(coms[2].Substring(0, coms[2].IndexOf('!')));

            servo.Mover(ang);
            motor.Mover(mot);
        }
    }
}
```

Servo.cs

```
using System;
using System.Threading;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.Hardware;

namespace NetZeppelin
{
    public class Servo
    {
        OutputCompare oc;

        public Servo() { }
        public Servo(FEZ_Pin.Digital digi)
        {
            this.oc = new OutputCompare((Cpu.Pin)digi, true, 2);
        }

        public void Mover(int angulo)
        {
            uint microsec = (uint)((angulo * 5) + 800);
            oc.Set(true, new uint[] { microsec, 20000 - microsec }, 0, 2, true);
        }
    }
}
```

Motores.cs

```
using System;
using System.Threading;
using Microsoft.SPOT;
using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.Hardware;
using Microsoft.SPOT.Hardware;


namespace NetZeppelin
{
    public class Motores
    {
        PWM motor11;
        PWM motor12;
        PWM motor21;
        PWM motor22;
        public int potencia;
        const uint periodoTotal = 50000000;
        uint periodoAlto;

        public Motores() { }
```

```
        public Motores(FEZ_Pin.PWM pwm11, FEZ_Pin.PWM pwm12, FEZ_Pin.PWM pwm21,
FEZ_Pin.PWM pwm22)
        {
            this.motor11 = new PWM((PWM.Pin)pwm11);
            this.motor12 = new PWM((PWM.Pin)pwm12);
            this.motor21 = new PWM((PWM.Pin)pwm21);
            this.motor22 = new PWM((PWM.Pin)pwm22);
        }

        public void EstablecerPotencia()
        {
            periodoAlto = (uint)((double)(periodoTotal / 5) * ((double)potencia /
100));
        }

        public void Parar()
        {
            motor11.Set(false);
            motor12.Set(false);
            motor21.Set(false);
            motor22.Set(false);
        }

        public void MoverAdelante()
        {
            motor12.Set(false);
            motor22.Set(false);
            motor11.SetPulse(periodoTotal, periodoAlto);
            motor21.SetPulse(periodoTotal, periodoAlto);
        }

        public void MoverAtras()
        {
            motor11.Set(false);
            motor21.Set(false);
            motor12.SetPulse(periodoTotal, periodoAlto);
            motor22.SetPulse(periodoTotal, periodoAlto);
        }

        public void GirarIzquierdaAdelante()
        {
            motor12.Set(false);
            motor21.Set(false);
            motor22.Set(false);
            motor11.SetPulse(periodoTotal, periodoAlto);
        }

        public void GirarIzquierdaAtras()
        {
            motor11.Set(false);
            motor21.Set(false);
            motor22.Set(false);
            motor12.SetPulse(periodoTotal, periodoAlto);
        }

        public void GirarDerechaAdelante()
        {
            motor11.Set(false);
            motor12.Set(false);
            motor22.Set(false);
            motor21.SetPulse(periodoTotal, periodoAlto);
```

```
        }

        public void GirarDerechaAtras()
        {
            motor11.Set(false);
            motor12.Set(false);
            motor21.Set(false);
            motor22.SetPulse(periodoTotal, periodoAlto);
        }

        public void Mover(string com)
        {
            EstablecerPotencia();

            if (com == "F")
                MoverAdelante();
            else if (com == "I")
                GirarIzquierdaAdelante();
            else if (com == "D")
                GirarDerechaAdelante();
            else if (com == "B")
                MoverAtras();
            else if (com == "L")
                GirarIzquierdaAtras();
            else if (com == "R")
                GirarDerechaAtras();
            else if (com == "S")
                Parar();
        }
    }
}
```

Ultrasonidos.cs

```
using System;
using Microsoft.SPOT;
using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.Hardware;

namespace NetZeppelin
{
    class Ultrasonidos
    {
        AnalogIn ai;

        public Ultrasonidos() { }
        public Ultrasonidos(FEZ_Pin.AnalogIn analog)
        {
            this.ai = new AnalogIn((AnalogIn.Pin)analog);
        }

        public int Medir()
        {
            return ai.Read();
        }
    }
}
```

Bateria.cs

```
using System;
using Microsoft.SPOT;
using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.Hardware;

namespace NetZeppelin
{
    public class Bateria
    {
        AnalogIn ai;

        public Bateria() { }
        public Bateria(FEZ_Pin.AnalogIn analog)
        {
            this.ai = new AnalogIn((AnalogIn.Pin)analog);
        }

        public int Medir()
        {
            return ai.Read();
        }
    }
}
```

Acelerometro.cs

```
using System;
using Microsoft.SPOT;
using Microsoft.SPOT.Hardware;
using Microsoft.SPOT.Presentation;
using Microsoft.SPOT.Presentation.Media;
using Microsoft.SPOT.Presentation.Controls;
using GHIElectronics.NETMF.FEZ;
using GHIElectronics.NETMF.Hardware;

namespace NetZeppelin
{
    class Acelerometro
    {
        AnalogIn entradaX;
        AnalogIn entradaY;
        AnalogIn entradaZ;
        OutputPort gs1;
        OutputPort gs2;
        OutputPort sleep;

        public Acelerometro() { }

        public Acelerometro(FEZ_Pin.AnalogIn analogPinX, FEZ_Pin.AnalogIn
analogPinY, FEZ_Pin.AnalogIn analogPinZ, FEZ_Pin.Digital pinGs1, FEZ_Pin.Digital
pinGs2, FEZ_Pin.Digital digitalPinSleep)
        {
            this.entradaX = new AnalogIn((AnalogIn.Pin)analogPinX);
            this.entradaY = new AnalogIn((AnalogIn.Pin)analogPinY);
            this.entradaZ = new AnalogIn((AnalogIn.Pin)analogPinZ);
            this.gs1 = new OutputPort((Cpu.Pin)pinGs1, false);
            this.gs2 = new OutputPort((Cpu.Pin)pinGs2, false);
            this.sleep = new OutputPort((Cpu.Pin)digitalPinSleep, true);
        }
```

```
        public int MedirX()
        {
            return entradaX.Read();
        }

        public int MedirY()
        {
            return entradaY.Read();
        }

        public int MedirZ()
        {
            return entradaZ.Read();
        }
    }
}
```

Código de la estación base

MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Text;
using System.Diagnostics;
using System.IO.Ports;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Threading;
using System.Drawing;

namespace Groundstation
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        bool haEmpezado = false;
        bool storyBatOn = false;
        public bool openCOM = false;
        Storyboard storyBat = new Storyboard();
        Storyboard storyRCOn = new Storyboard();
        Storyboard storyRCOff = new Storyboard();
        public SerialPort serial1;
        ConfiguracionGroundstation ventanaConfig;

        // Variables del evento de recepción de datos por el puerto serie

        String datosRecibidos;
        byte[] readBuffer;
        int readed;
```

```
        int lecturaBat = 0;
        int lecturaUlt = 0;
        int[] acs = new int[3];
        public JoystickInterface.Joystick joystick;
        bool isLoaded = false;

        public MainWindow()
        {
            this.InitializeComponent();
            this.Loaded += new RoutedEventHandler(MainWindow_Loaded);

        }

        void MainWindow_Loaded(object sender, RoutedEventArgs e)
        {
            if (!isLoaded)
            {
                isLoaded = true;
                ventanaConfig = new ConfiguracionGroundstation(this);
                ventanaConfig.WindowStartupLocation =
System.Windows.WindowStartupLocation.Manual;
                ventanaConfig.Top = this.Top;
                ventanaConfig.Left = this.Left + this.ActualWidth;
                ventanaConfig.Show();

                // Buscar el storyboard de la batería
                storyBat = TryFindResource("storyBajaBateria") as Storyboard;
                storyRCOn = TryFindResource("storyRCOn") as Storyboard;
                storyRCOff = TryFindResource("storyRCOff") as Storyboard;

                // Detectar y seleccionar un joystick
                joystick = new JoystickInterface.Joystick();
                string[] listaJoysticks = joystick.FindJoysticks();
                readBuffer = new byte[100];

                if (listaJoysticks != null)
                {
                    joystick.AcquireJoystick(listaJoysticks[0]);
                    textoModo.Text = "Control manual: activado";
                    bloqueText.Text = "Se han detectado joysticks,
seleccionando...\r\n";
                    bloqueText.AppendText("Joystick seleccionado: " +
listaJoysticks[0]);
                    storyRCOn.Begin();
                }
                else
                {
                    textoModo.Text = "Control manual: desactivado";
                    bloqueText.Text = "No se ha detectado ningún
joystick...\r\n";
                    bloqueText.AppendText("Conecte uno y reinicie la
aplicación.");
                    storyRCOff.Begin();
                }

                // Actualización inicial de la
interfaz.....................................................
                ActualizarBateria(barraBateria, textoBateria);

                textoAltitud.Text = "Altitud: " + medidorAltitud.Value.ToString()
+ " cm";
```

```
                textoServo.Text = "Ángulo servo: " + ((int)((posicionServo.Value
* 2 / 100) * 90) - 90).ToString() + "°";

                // Iniciar conexión XBee
                serial1 = new SerialPort("COM7", 9600, Parity.None, 8,
StopBits.One);

                try
                {
                    serial1.Open();
                    openCOM = true;
                }
                catch
                {
                    bloqueText.AppendText("\r\nNo se ha podido abrir el puerto
serie...");
                    openCOM = false;
                }
                serial1.DataReceived += new
SerialDataReceivedEventHandler(serial1_DataReceived);

                if ((listaJoysticks != null) && (openCOM == true))
                {
                    DispatcherTimer timer1 = new DispatcherTimer(new TimeSpan(0,
0, 0, 0, 20), DispatcherPriority.Normal,
                        enviarOrdenesJoystick, this.Dispatcher);
                    timer1.Start();
                }
            }

        }

        private void medidorAltitud_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
        {
            textoAltitud.Text = "Altitud: " +
Convert.ToInt32(medidorAltitud.Value).ToString() + " cm";
        }

        private void barraBateria_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
        {
            if (haEmpezado == true) { ActualizarBateria(barraBateria,
textoBateria); }
        }

        private void posicionServo_ValueChanged(object sender,
RoutedPropertyChangedEventArgs<double> e)
        {
            if (haEmpezado == true) { textoServo.Text = "Ángulo servo: " +
(posicionServo.Value - 90) + "°"; }
            haEmpezado = true;
        }

        private void ActualizarBateria(ProgressBar bBat, TextBlock tBat)
        {
            if (bBat.Value > 60)
            {
                bBat.Template = TryFindResource("ControlTemplateNivelNormal") as
ControlTemplate;
                tBat.Style = TryFindResource("StyleTextoVerde") as Style;
                if (storyBatOn == true)
```

```
                {
                    storyBat.Stop();
                    storyBatOn = false;
                }
                else { }
            }
            if ((bBat.Value < 61) && (bBat.Value > 25))
            {
                bBat.Template =
TryFindResource("ControlTemplateBateriaNivelMedio") as ControlTemplate;
                tBat.Style = TryFindResource("StyleTextoAmarillo") as Style;

                if (storyBatOn == true)
                {
                    storyBat.Stop();
                    storyBatOn = false;
                }
                else { }
            }
            if (bBat.Value <= 25)
            {
                bBat.Template =
TryFindResource("ControlTemplateBateriaNivelBajo") as ControlTemplate;
                tBat.Style = TryFindResource("StyleTextoRojo") as Style;

                if (storyBatOn == false)
                {
                    storyBat.Begin();
                    storyBatOn = true;
                }
                else { }
            }
            tBat.Text = "Batería: " + bBat.Value.ToString() + "%";
        }

        public void serial1_DataReceived(object sender,
SerialDataReceivedEventArgs e)
        {
            // Check if Chars are received
            if (e.EventType == SerialData.Chars)
            {
                lock (readBuffer)
                {
                    int position;
                    readed = serial1.Read(readBuffer, 0,
System.Math.Min(serial1.BytesToRead, 99));
                    datosRecibidos = datosRecibidos + new
string(Encoding.UTF8.GetChars(readBuffer, 0, readed));

                    while ((position = datosRecibidos.IndexOf('!')) != -1)
                    {
                        string mensaje = datosRecibidos.Substring(0, position);

                        if (mensaje.Length != 4)
                        {
                            Debug.Print("Mensaje cortado!");
                        }

                        if (mensaje.IndexOf('B') != -1)
                        {
                            try
                            {
```

```
                                lecturaBat = Convert.ToInt32(mensaje.Substring(0,
mensaje.IndexOf('B')));
                            }
                            catch (Exception ex)
                            {
                                Debug.Print(ex.ToString());
                            }
                        }
                        else if (mensaje.IndexOf('X') != -1)
                        {
                            try
                            {
                                acs[0] = Convert.ToInt32(mensaje.Substring(0,
mensaje.IndexOf('X')));
                            }
                            catch (Exception ex)
                            {
                                Debug.Print(ex.ToString());
                            }
                        }
                        else if (mensaje.IndexOf('Y') != -1)
                        {
                            try
                            {
                                acs[1] = Convert.ToInt32(mensaje.Substring(0,
mensaje.IndexOf('Y')));
                            }
                            catch (Exception ex)
                            {
                                Debug.Print(ex.ToString());
                            }
                        }
                        else if (mensaje.IndexOf('Z') != -1)
                        {
                            try
                            {
                                acs[2] = Convert.ToInt32(mensaje.Substring(0,
mensaje.IndexOf('Z')));
                            }
                            catch (Exception ex)
                            {
                                Debug.Print(ex.ToString());
                            }
                        }
                        else if (mensaje.IndexOf('U') != -1)
                        {
                            try
                            {
                                lecturaUlt = Convert.ToInt32(mensaje.Substring(0,
mensaje.IndexOf('U')));
                            }
                            catch (Exception ex)
                            {
                                Debug.Print(ex.ToString());
                            }
                        }
                        else
                        {
                        }
                        datosRecibidos = datosRecibidos.Substring(position + 1);
                    }
```

```
                    this.Dispatcher.Invoke(DispatcherPriority.Normal, (Action)(()
=>
                    {
                        barraBateria.Value = lecturaBat / 10;
                        cuadroAceleracion.Text = "Eje X: " + acs[0] + "\r\n" +
"Eje Y: " + acs[1] + "\r\n" + "Eje Z: " + acs[2];
                        medidorAltitud.Value = lecturaUlt;
                    }));
                }
            }
        }

        public void enviarOrdenesJoystick(object sender, EventArgs e)
        {
            joystick.UpdateStatus();

            if (joystick.AxisB < 32768)
            {
                ProgressBarJoystickArriba.Value = (int)((float)(joystick.AxisB /
-327.67)) + 100;
                ProgressBarJoystickAbajo.Value = 0;
            }
            else
            {
                ProgressBarJoystickAbajo.Value = (int)((float)(joystick.AxisB /
327.67)) - 100;
                ProgressBarJoystickArriba.Value = 0;
            }
            if (joystick.AxisA < 32768)
            {
                ProgressBarJoystickIzquierda.Value = (int)((float)(joystick.AxisA
/ -327.67)) + 100;
                ProgressBarJoystickDerecha.Value = 0;

            }
            else
            {
                ProgressBarJoystickDerecha.Value = (int)((float)(joystick.AxisA /
327.67)) - 100;
                ProgressBarJoystickIzquierda.Value = 0;
            }
            string angulo = calculoServo(joystick.Slider);
            posicionServo.Value = Convert.ToInt32(angulo);
            string com = calculoMotores(joystick.AxisA, joystick.AxisB);
            int pot = calculoPotenciaMotores(com);
            serial1.Write(angulo + "#" + com + "#" + pot + "!");
            this.Dispatcher.Invoke(DispatcherPriority.Normal, (Action)(() =>
            {
                        if (com =="F"||com=="B")
                        {
                            textoPotenciaI.Text = "Potencia del\r\nmotor
izquierdo: " + pot + "%";
                            barraPotenciaI.Value = pot;
                            textoPotenciaD.Text = "Potencia del\r\nmotor
derecho: " + pot + "%";
                            barraPotenciaD.Value = pot;
                        }
                        else if(com=="I"||com=="L")
                        {
                            textoPotenciaI.Text = "Potencia del\r\nmotor
izquierdo: " + pot + "%";
                            barraPotenciaI.Value = pot;
```

```
                                textoPotenciaD.Text = "Potencia del\r\nmotor
derecho: " + 0 + "%";
                                barraPotenciaD.Value = 0;
                            }
                            else if(com=="D"||com=="R")
                            {
                                textoPotenciaD.Text = "Potencia del\r\nmotor
derecho: " + pot + "%";
                                barraPotenciaD.Value = pot;
                                textoPotenciaI.Text = "Potencia del\r\nmotor
izquierdo: " + 0 + "%";
                                barraPotenciaI.Value = 0;
                            }
                            else{}

                if (pot > 66)
                {
                    barraPotenciaI.Template =
TryFindResource("barraPotenciaMaxTemplate") as ControlTemplate;
                                barraPotenciaD.Template =
TryFindResource("barraPotenciaMaxTemplate") as ControlTemplate;
                }

                else if (pot > 33)
                {
                    barraPotenciaI.Template =
TryFindResource("barraPotenciaMedTemplate") as ControlTemplate;
                                barraPotenciaD.Template =
TryFindResource("barraPotenciaMedTemplate") as ControlTemplate;
                }

                else
                {
                    barraPotenciaI.Template =
TryFindResource("barraPotenciaTemplate") as ControlTemplate;
                                barraPotenciaD.Template =
TryFindResource("barraPotenciaTemplate") as ControlTemplate;
                }
            }));
        }

        public string calculoServo(int slider)
        {
            double angulo = (65535 - slider) / 364.08333333333;
            int ang = (int)angulo;
            string res;
            if (ang < 10)
                res = "00" + ang.ToString();
            else if (ang < 100)
                res = "0" + ang.ToString();
            else
                res = ang.ToString();

            return res;
        }

        public string calculoMotores(int ejeX, int ejeY)
        {
            string com = "S";
            int currentA = joystick.AxisA;
            int currentB = joystick.AxisB;
            if (currentB < 32767 && currentA > 16384 && currentA < 49152)
```

```
            { com = "F"; }      // Acelerar (Forward)
            else if (currentB < 32768 && currentA < 32767) //16384
            { com = "I"; }      // Giro izquierda hacia adelante
            else if (currentB < 32768 && currentA > 32768) //49152
            { com = "D"; }     // Giro derecha hacia adelante
            else if (currentB > 32767 && currentA > 16384 && currentA < 49152)
            { com = "B"; }      // Marcha atrás (Backwards)
            else if (currentB > 32767 && currentA < 16384)
            { com = "L"; }    // Giro izquierda hacia atrás
            else if (currentB > 32767 && currentA > 49152)
            { com = "R"; }      // Giro derecha hacia atrás
            else if (currentA == 32767 && currentB == 32767)
            { com = "S"; }       // Parar (Stop)

            return com;
        }

        public int calculoPotenciaMotores(string orden)
        {
            int pot = 0;
            int currentA = joystick.AxisA;
            int currentB = joystick.AxisB;
            if (orden == "F")
            {
                pot = 100 - (int)(currentB / 327.66D);
            }
            else if (orden == "I" || orden == "L")
            {
                pot = 100 - (int)(currentA / 327.66D);
            }
            else if (orden == "B")
            {
                pot = (int)(currentB / 327.66D) - 100;
            }
            else if (orden == "D" || orden == "R")
            {
                pot = (int)(currentA / 327.66D) - 100;
            }
            return pot;
        }
    }
}
```