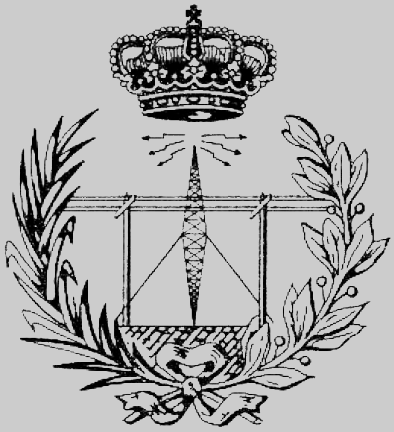
E.T.S de Ingeniería Industrial,
Informática y de Telecomunicación

Diseño de inversor con microcontrolador para control de motores



Grado en ingeniería eléctrica y electrónica

Trabajo Fin de Grado

Autor: Iñigo Huguet Embún

Director: Jesús Corrés Sanz

Pamplona, junio 2014

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Resumen

Diseño de tarjeta PCB, con entrada de tensión alterna de red, y salida en tensión alterna a partir de tensión de bus continua de 325V conmutada, para control de motor síncrono de imanes permanentes.

Desarrollo de software para microcontrolador, destinado a controlar el motor para el manejo completo de una puerta de ascensor, llevando a cabo un control de velocidad mediante control vectorial (en inglés FOC: field oriented control). Medición de la velocidad y la posición mediante encoder de cuadratura, detección de obstáculos por aumento de corriente y configuración de parámetros de control a través de un menú.

Palabras clave

Motor síncrono, imanes permanentes, control de velocidad, control vectorial, puerta de ascensor, PCB

Índice de contenidos

Índice de anexos

1. Objetivo

El presente Trabajo Fin de Grado se ha realizado como parte de las prácticas en empresa realizadas a lo largo de los últimos meses en la empresa Fanamoel, ubicada en Oteiza.

Quien redacta este Trabajo Fin de Grado ha formado parte, en dicha empresa, de un equipo de dos personas (incluyéndose a sí mismo) encargadas de llevar a cabo el proyecto que se describe en las siguientes lineas.

El proyecto desarrollado ha consistido en la realización del diseño completo de un controlador electrónico para un motor síncrono de imanes permanentes (PMSM), cuya función será la de operar puertas de ascensor.

La implementación física de dicho controlador debía ser realizada mediante una tarjeta de circuito impreso (PCB), la cual estaría gobernada por un microcontrolador. Así pues, las tareas realizadas comprenden tanto el diseño del hardware (el PCB) como el desarrollo del software para dicho microcontrolador.

Requisitos de partida:

- Motor a controlar
  - Motor síncrono de imanes permantentes (PMSM)
  - Alimentación 50V
  - 180 rpm
  - 8 pares de polos
- Control de posición de la puerta
  - Sin finales de carrera ni detectores de posición: debe calcularse la posición en función de la rotación del motor
  - Cálculo de la rotación mediante un encoder:
    - Encoder de cuadratura (para detectar el sentido de giro)
    - Hasta 4096 pulsos por vuelta (contando con flancos de subida y bajada)
    - Con detección de índice
  - Posibilidad de ser usado en puertas de distintas longitudes.

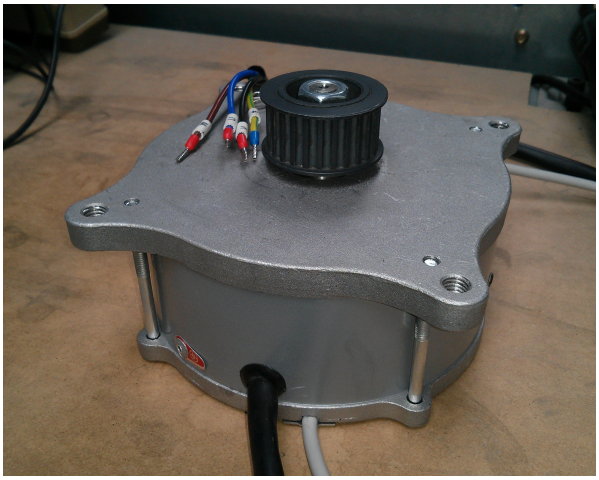


Imagen 1: motor síncrono de imanes permanentes con encoder

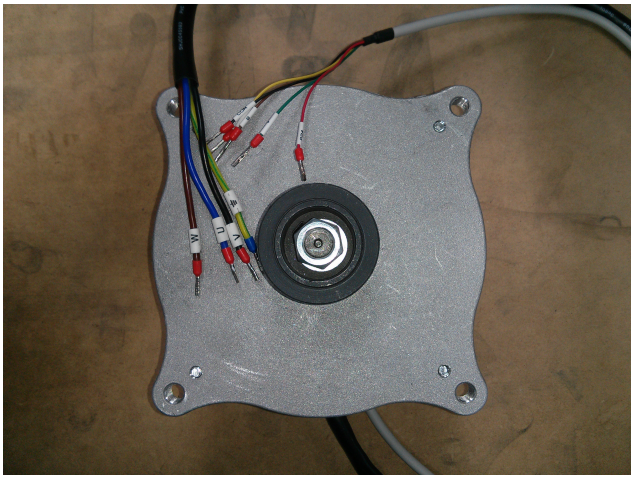


Imagen 2: motor síncrono de imanes permanentes con encoder

Además de los requisitos anteriores, que fueron impuestos por el cliente, por parte de la empresa se decidió hacer uso de las técnicas de control vectorial, en aras tanto de una mayor eficiencia como de una mayor precisión en el control.

2. La máquina síncrona de imanes permanentes

Como en toda máquina eléctrica rotativa, el funcionamiento de la máquina síncrona de imanes permanentes se fundamenta en la interacción de los campos magnéticos del rotor y del estator, la cual genera un par mecánico que hace girar al rotor (en el caso del funcionamiento como motor, que es el que se necesita en la presente aplicación).

En la máquina síncrona de imanes permanentes el rotor está constituido por imanes permanentes. Por ese motivo, el campo magnético del rotor es constante y alineado con la posición mecánica del rotor.

Este hecho tiene la ventaja de que, por tanto, el rotor no requiere de conexiones eléctricas con el exterior, evitando así contactos con escobillas que están sujetas a fuertes desgastes. Por contra, se pierde la posibilidad de controlar en gran medida la interacción entre ambos campos magnéticos, ya que no se puede atenuar o fortalecer el campo magnético del rotor, como en la máquina síncrona convencional.

El estator se compone de tres devanados que deben ser alimentados con un sistema trifásico de corrientes senoidales desfasadas 120º.

No es objeto del presente documento una descripción pormenorizada de la máquina síncrona, por lo que no se van a explicar ni tener en cuenta las distintas técnicas constructivas de la misma, tales como el uso de varios pares de polos por devanado, devanados distribuidos y/o de paso acortado, etc. Todos los cálculos, explicaciones teóricas, etc, que se lleven a cabo en todo el documento se realizarán en base al modelo simplificado de la máquina, con una sola espira por devanado, un solo par de polos* y campo magnético de distribución senoidal a lo largo del entrehierro.

**nota: el número de pares de polos sí deberá tenerse en cuenta en algunos casos, como por ejemplo para relacionar las magnitudes mecánicas con las magnitudes eléctricas*

3. Control vectorial

Como el tipo de control a llevar a cabo es el control vectorial, se hace indispensable abordar brevemente ese tema, haciendo una pequeña introducción a la teoría acerca del mismo. Así pues, en este apartado se verá de manera resumida en qué consiste el control vectorial y cuales son sus ventajas.

3.1. Necesidad del control vectorial

La técnica del control vectorial responde a la necesidad de controlar por separado la componente de la corriente que genera el par de la máquina de la componente que genera el flujo magnético. En el modelo matemático de la máquina estas variables no pueden controlarse de forma independiente y sencilla al mismo tiempo, obligando a realizar los cálculos necesarios para el control utilizando ecuaciones diferenciales.

El control vectorial consigue un tipo de control lineal respecto a dichas variables, siendo dicho control muy similar al que se puede utilizar con una máquina de corriente continua, en la que sí se pueden controlar estas corrientes de manera independiente.

3.2. Representación vectorial de las tensiones y corrientes: vectores espaciales

La técnica de control vectorial hace uso de los vectores para representar la orientación y magnitud de las variables que intervienen en el movimiento de la máquina, tales como corrientes, tensiones, campos magnéticos, etc.

Cabe aclarar que los vectores con los que se trabaja en el control vectorial son vectores espaciales, no fasores. Es decir, son vectores que representan una orientación espacial y una magnitud, a diferencia de los fasores que representan una magnitud y un desfase temporal.

Cabría la posibilidad de utilizar vectores en el sistema de referencia de 3 ejes que podemos imaginar sobre los ejes magnéticos de los 3 devanados del estator, que se encuentran a 120º de distancia entre sí (ejes a, b y c).

De esta forma, sería posible representar el sistema trifásico como un solo vector espacial, cuyas componentes serían vectores sobre los ejes a, b y c, cuyas magnitudes serían las magnitudes reales de tensiones y corrientes de cada uno de los devanados (véase figura 1).

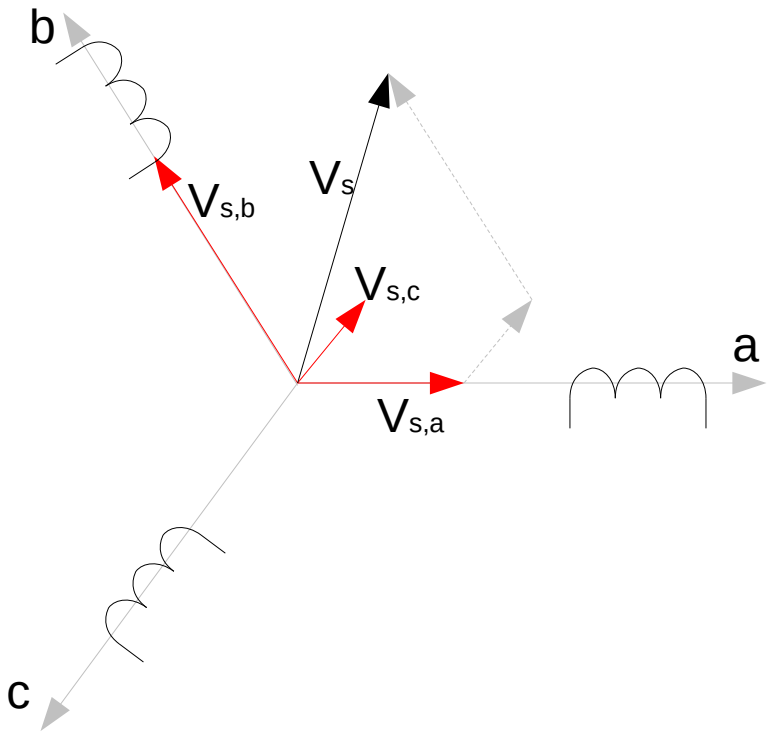


Figura 1: Vector de tensiones en el sistema de referencia a-b-c

En un sistema trifásico equilibrado, el vector resultante gira a una velocidad angular constante, que se corresponde con la velocidad de sincronismo.

No obstante, resultaría muy difícil operar utilizando un sistema de referencia de estas características.

3.3. Cambio de referencia a sistema de dos ejes perpendiculares: transformada de Clarke

Utilizando la transformada de Clarke nos es posible representar un sistema trifásico equilibrado de tensiones o corrientes senoidales en forma de vector espacial, referenciado respecto a un sistema de tan solo dos ejes perpendiculares entre sí.

En definitiva, mediante la transformada de Clarke podemos pasar del sistema de referencia a-b-c descrito un poco más arriba, a un sistema de dos ejes α-β perpendiculares entre sí, como puede verse en la figura 2.

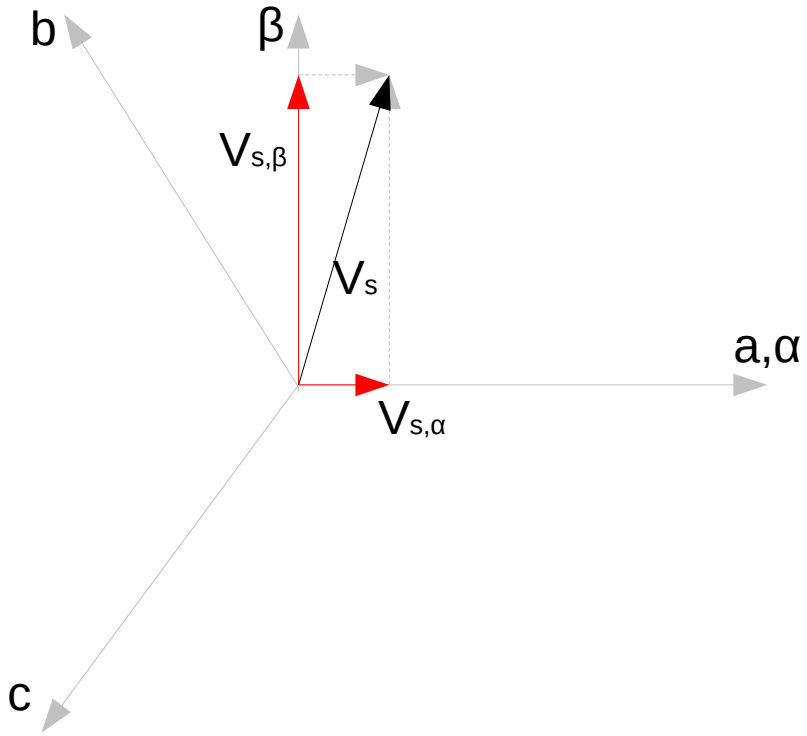


Figura 2: Vector de tensiones en el sistema de referencia α-β

Es importante recordar que esto solo es posible en el caso de un sistema trifásico equilibrado. Tampoco se debe olvidar que, en realidad, el vector no cambia en absoluto, ni en magnitud ni en dirección, sino que tan solo se expresa en base a un sistema de referencia distinto.

Transformada de Clarke (representación matricial):

$$\begin{bmatrix} v_\alpha \\ v_\beta \\ v_\gamma \end{bmatrix} = \frac{2}{3} \begin{bmatrix} 1 & -1/2 & -1/2 \\ 0 & \sqrt{3}/2 & -\sqrt{3}/2 \\ 1/2 & 1/2 & 1/2 \end{bmatrix} \begin{bmatrix} v_a \\ v_b \\ v_c \end{bmatrix} \quad (1)$$

Transformada inversa de Clarke (representación matricial):

$$\begin{bmatrix} v_a \\ v_b \\ v_c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ -1/2 & \sqrt{3}/2 & 1 \\ -1/2 & -\sqrt{3}/2 & 1 \end{bmatrix} \begin{bmatrix} v_\alpha \\ v_\beta \\ v_\gamma \end{bmatrix} \quad (2)$$

3.4. Cambio de referencia a sistema rotatorio: transformada de Park

No obstante, el cambio a un sistema de referencia α-β no soluciona el problema que se pretendía resolver, que es el de conseguir un control desacoplado de las corrientes de par y de magnetización. Esto se consigue con un nuevo cambio de referencia.

El nuevo sistema de referencia consta de dos ejes perpendiculares entre sí, los ejes d-q. El eje d se encuentra alineado con el campo magnético del rotor y el eje q en cuadratura con el mismo. Esto supone que los ejes d-q son ejes rotatorios, ya que giran de forma solidaria con el rotor.

Así pues el cambio de referencia del sistema α-β al sistema d-q consiste en girar los ejes un ángulo igual a la posición angular del rotor. Esto se conoce como transformada de Park.

Esto es importante debido a lo siguiente:

- La magnitud de la corriente proyectada sobre el eje d se corresponde con la corriente de magnetización, al estar alineada con el campo magnético del rotor
- La magnitud de la corriente proyectada sobre el eje q se corresponde con la corriente de par, al ser perpendicular al campo magnético del rotor.

**nota: el campo magnético del estator es proporcional a la corriente*

Esto es de vital importancia, y es la base sobre la que se fundamenta la utilidad del control vectorial, ya que es precisamente lo que permite controlar de forma desacoplada las corrientes de par y de magnetización, controlando por separado las componentes d y q del vector de tensiones del estator.

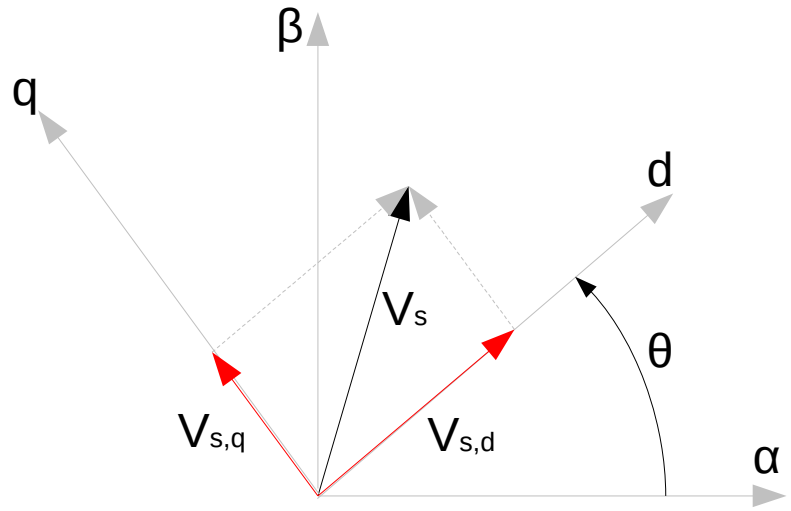


Figura 3: Vector de tensiones en el sistema de referencia d-q

Además, hay que tener en cuenta lo siguiente:

- El vector de tensiones del estator (Vs) es un vector que gira a la velocidad de sincronismo en el sistema de referencia α-β.
- Al ser una máquina síncrona, el rotor gira a la velocidad de sincronismo en el sistema de referencia α-β, por lo que los ejes d-q también giran a la velocidad de sincronismo respecto a los ejes α-β.
- Según los dos puntos anteriores, el vector Vs y los ejes d-q giran a la misma velocidad, la de sincronismo.

Según este último punto podemos concluír que el vector Vs no es un vector rotatorio en el sistema de referencia d-q, ya que no existe velocidad relativa entre dicho vector y los ejes d-q. Además, las proyecciones d y q del vector de tensiones se mantienen constantes mientras no se decida modificar dicho vector.

Gracias a esto resulta mucho más sencillo aplicar técnicas de control automático para controlar distintas variables del sistema (velocidad, corriente...). Esto se explica más adelante con más detalle.

Huelga decir que el vector Vs en realidad no ha cambiado, es el mismo vector rotatorio que representa las tensiones del estator, solo que proyectado sobre unos ejes distintos: los ejes d-q.

Transformada de Park:

$$\begin{bmatrix} v_d \\ v_q \\ v_0 \end{bmatrix} = \begin{bmatrix} \cos\theta & sen\,\theta & 0 \\ -sen\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_\alpha \\ v_\beta \\ v_\gamma \end{bmatrix} \tag{3}$$

Antitransformada de Park:

$$\begin{bmatrix} v_\alpha \\ v_\beta \\ v_\gamma \end{bmatrix} = \begin{bmatrix} \cos\theta & -sen\,\theta & 0 \\ sen\,\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_d \\ v_q \\ v_0 \end{bmatrix} \tag{4}$$

Siendo θ el giro angular de los ejes d-q respecto a los ejes α-β. Este valor irá variando con el giro del rotor.

3.5. Modulación de espacios vectoriales

Una vez que se conoce el vector de tensiones deseado, es necesario generar ese vector con el inversor trifásico, compuesto por tres células de conmutación PWM.

Para la modulación es necesario expresar el vector Vs en un sistema de referencia estático, como el α-β o el a-b-c, para lo cual es necesario calcularlo utilizando la antitransformada de Park.

Como se ha visto anteriormente, dicho cálculo depende del valor de θ, que es el giro angular entre los ejes d-q y α-β en cada instante de tiempo. Esto supone que, a lo largo de una revolución del rotor, como hay infinitos valores de θ, hay infinitos valores del vector Vs.

No obstante, el inversor no permite modular cualquier vector, sino que de hecho solo puede modular 8 vectores diferentes, que se corresponden con las 8 posibles combinaciones de estados de conmutación. A estos 8 vectores se los conoce como vectores directores.

Todas las combinaciones de conmutación se muestran en la figura 4.

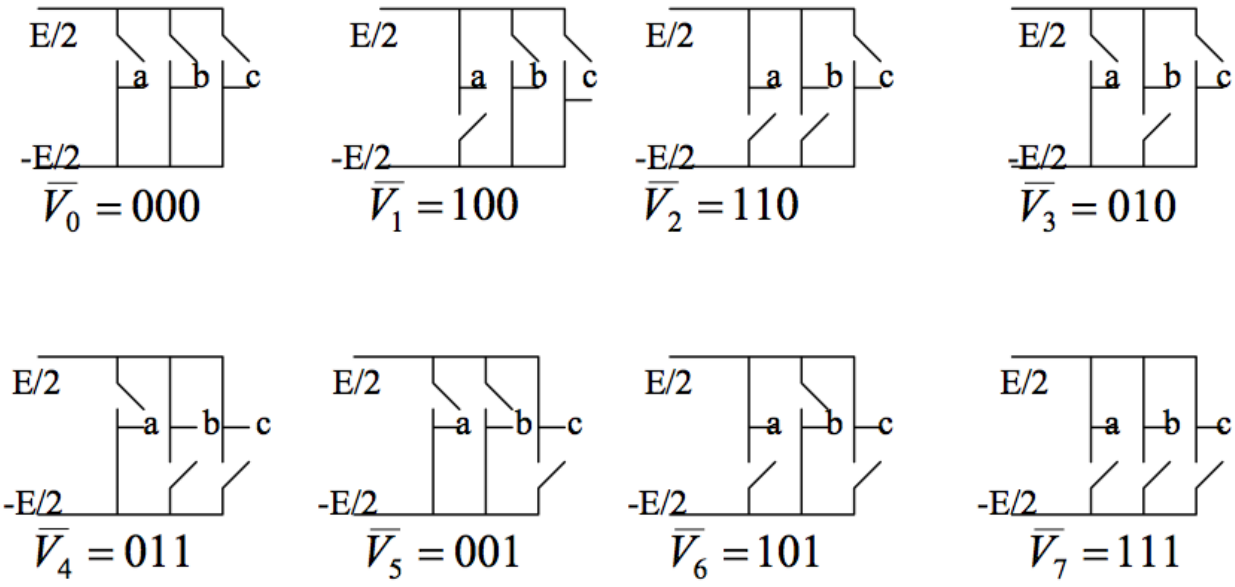


Figura 4: posibles combinaciones de conmutación (fuente: Control vectorial del motor asíncrono. Angel Vernavá, Roberto Gibbons, Antonio Nachez, Marcelo Arias y Armando Novello)

Como se puede comprobar, los vectores 000 y 111 son vectores nulos, ya que no generan diferencias de tensión entre las fases.

Utilizando estos vectores, es muy común el uso de la modulación “six steps” y otras, pero que normalmente consiguen unos resultados algo toscos, que no son válidos para muchas aplicaciones.

Sin embargo, con la modulación de espacios vectoriales (SVM) pueden conseguirse un mayor número de vectores utilizando los 8 que es capaz de entregar el inversor. Esta técnica se basa en

utilizar para cada valor de θ una combinación de los dos vectores adyacentes al vector deseado y un vector nulo, controlando los tiempos en que esta activo cada uno de esos 3 vectores.

El vector resultante se halla mediante la siguiente expresión:

$$\vec{V}=\frac{T_a}{T_s}\vec{V}_a+\frac{T_b}{T_s}\vec{V}_b+\frac{T_0}{T_s}\vec{V}_0 \qquad (5)$$

Siendo:

V_a = vector director anterior
V_b = vector director siguiente
V_0 = vector nulo
T_a = tiempo que permanece activo el vector V_a en un periodo de conmutación
T_b = tiempo que permanece activo el vector V_b en un periodo de conmutación
T_0 = tiempo que permanece activo el vector V_0 en un periodo de conmutación
T_s = periodo de conmutación

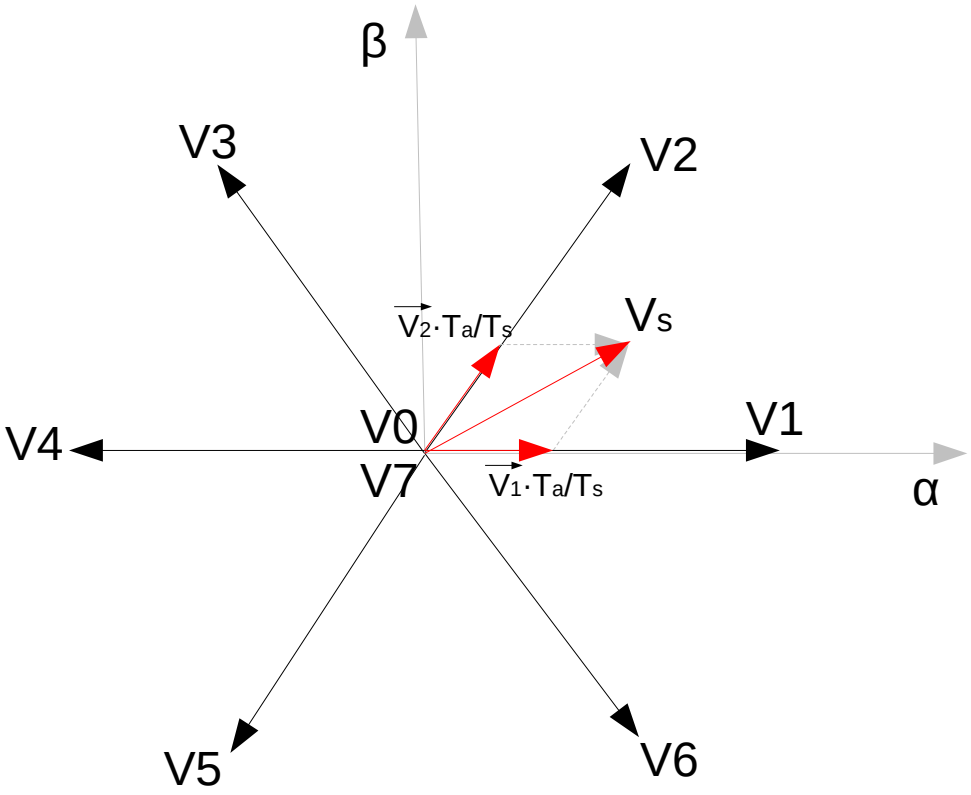


Figura 5: Vectores directores y conformación de un vector a partir de dos de ellos

Los tiempos se hallan de la siguiente manera

$$\begin{aligned} T_a &= \sqrt{3}\frac{V_{ref}}{E} T_s \cdot sen(\pi/3 - \phi_a) \\ T_b &= \sqrt{3}\frac{V_{ref}}{E} T_s \cdot sen(\phi_b) \\ T_0 &= T_s - T_a - T_b \end{aligned} \tag{6}$$

Siendo ϕ el ángulo que corresponde a cada vector director.

Con estos cálculos de tiempo, la conmutación de las 3 células se hace siguiendo un patrón de simetría respecto a Ts/2, lo que se conoce como “secuencia simétrica de los vectores nulos”. Aunque existen otros patrones que se podrían utilizar, son menos usados.

En la figura 6 se muestran las señales de activación de las llaves superiores de las células de conmutación correspondientes a este tipo de modulación.

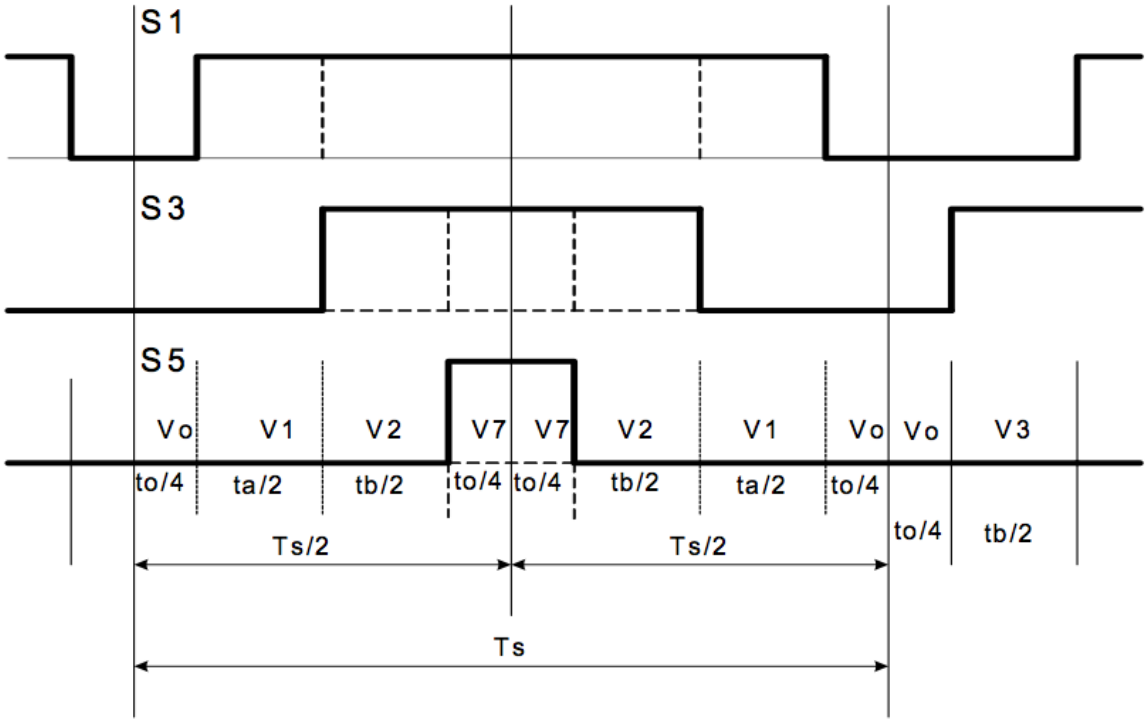


Figura 6: secuencia simétrica de los vectores nulos (fuente: Control vectorial del motor asíncrono. Angel Vernavá, Roberto Gibbons, Antonio Nachez, Marcelo Arias y Armando Novello)

En definitiva, la modulación de espacios vectoriales (SVM) permite controlar las conmutaciones del inversor para alimentar al motor con un sistema trifásico de tensiones, el cual se corresponde con el vector de tensiones Vs.

3.6. Controladores PI

Una vez se sabe como representar el vector de tensiones en distintos sistemas de referencia y como determinar los tiempos de conmutación de las células del inversor, queda implantar un método para seleccionar el vector de tensiones adecuado.

En la aplicación desarrollada en el presente trabajo, se pretende llevar a cabo un control sobre la velocidad de giro del motor. Para ello se implementarán 2 lazos de control, uno para controlar la velocidad, con un controlador de velocidad y otro interior para controlar el par*, y otro para controlar el flujo*. Esto hace necesario el uso de un total de 3 controladores PI.

**nota: tanto el flujo como el par se controlan de forma indirecta, midiendo las componentes d y q de la corriente*

La velocidad se controla en función de la referencia seleccionada. El flujo deseado es un flujo nulo, ya que se desea controlar una máquina síncrona, cuyo funcionamiento normal, al girar el rotor a velocidad de sincronismo, no debe generar flujo.

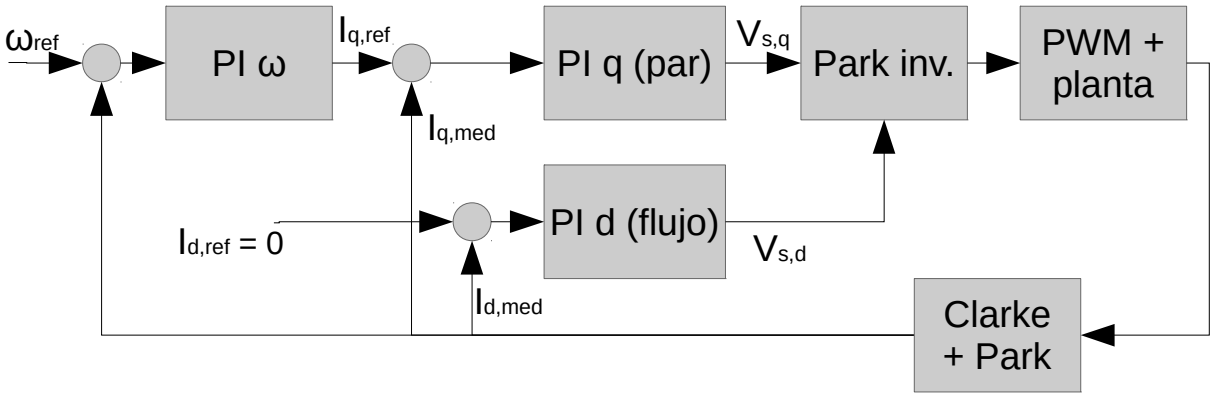


Figura 7: Esquema de control con controladores PI

3.7. Resumen control vectorial

Una vez explicados todos los conceptos que intervienen en el control vectorial, se puede entender el proceso completo del mismo:

1. Se miden las corrientes de fase del motor I_a, I_b e I_c, que conforman un vector de corrientes rotatorio. También se miden los valores de posición y velocidad angular del rotor.
2. Se calcula el valor del vector de corrientes I_{abc} en el sistema de referencia rotatorio d-q mediante las transformadas de Clarke y Park. Para ello es necesario conocer la posición angular del rotor.

En este sistema de referencia, el vector de corrientes es constante en magnitud y dirección (I_{dq}).

3. Se calcula el vector de tensiones $V_{s,dq}$ mediante los controladores PI. Los valores de realimentación son los siguientes:
   1. Controlador de velocidad: velocidad medida
   2. Controlador de par: componente de cuadratura del vector de corrientes (I_q)
   3. Controlador de flujo: componente directo del vector de corrientes (I_d)
4. Se calcula el vector de tensiones $V_{s,dq}$ referenciado a un sistema de referencia estático mediante las antitransformadas de Clarke y Park. Para ello es necesario conocer la posición angular del rotor.
5. Se calculan los tiempos de conmutación de las células del inversor a partir de dicho vector de tensiones.

La figura 8 muestra un esquema del sistema de control completo.

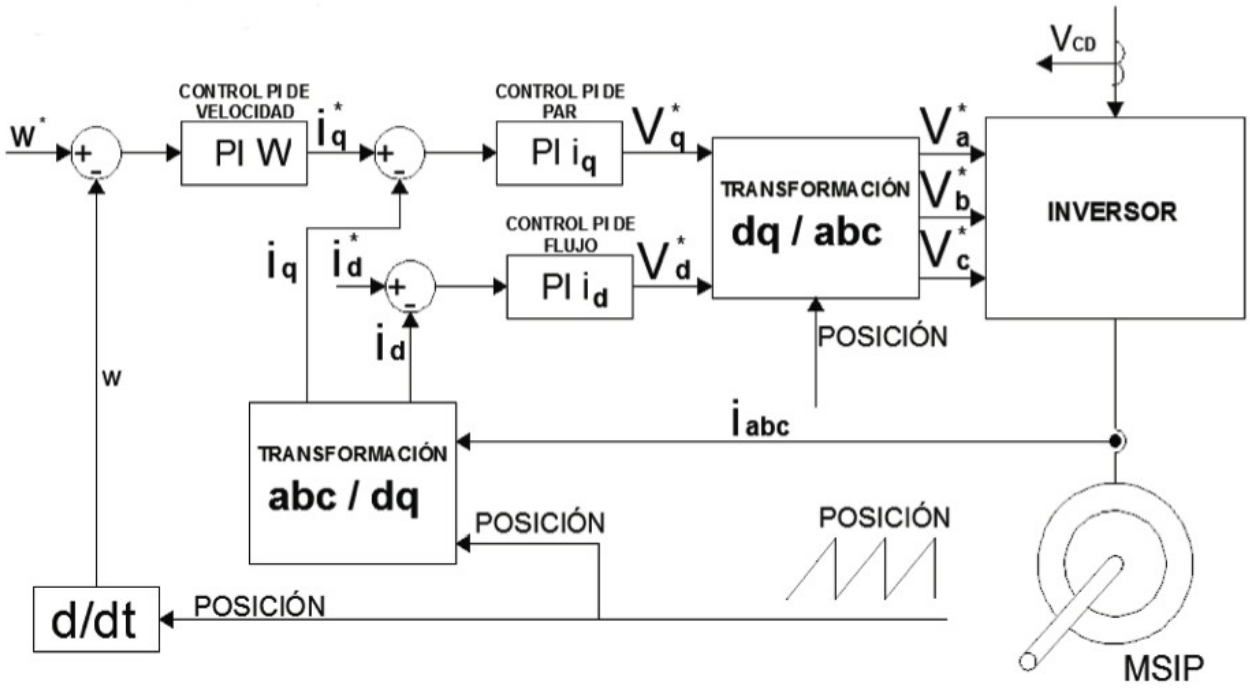


Figura 8: Control vectorial por control de velocidad (fuente:Control vectorial de la velocidad de una máquina síncrona de imanes permanentes. Pedro Celestino Castellano Morales)

3.8. Ventajas y desventajas del control vectorial

La principal ventaja del control vectorial es que permite desacoplar el control del par y el del flujo magnético, permitiendo un control mucho más preciso. Además, utilizando el sistema de referencia d-q, controlar la magnitud y dirección del vector de tensiones resulta mucho más sencillo, y se puede llevar a cabo de una forma muy similar a la de cualquier método de control tradicional (lazos de control anidados, etc).

De hecho, gracias al control vectorial, el control de la máquina se vuelve muy similar al de una máquina de corriente continua, que siempre ha resultado mucho más sencillo y preciso que el de las máquinas de corriente alterna.

Por contra, se hace necesario calcular la posición angular de manera precisa, o el cálculo del cambio de referencia al sistema d-q no se haría correctamente. Para esto se suele utilizar un encoder o sensores hall, que también sirven para la medición de la velocidad, aunque también existen algoritmos que estiman estos valores.

4. Estado inicial del proyecto

En el momento de la incorporación a la empresa, la realización del proyecto descrito en el presente informe se encontraba ya empezada, aunque en una etapa sumamente temprana. Es por eso que en este apartado se describirá el estado del mismo en dicho momento.

4.1. Hardware

En lo que al hardware se refiere, se había tomado la decisión de utilizar un microcontrolador de la familia dsPIC33E de Microchip, que gracias a su tecnología DSP (procesamiento digital de señal) cuenta con mejor soporte de hardware para operaciones matemáticas complejas.

También se habían adquirido dos tarjetas de evaluación que permitieran probar la viabilidad del proyecto, así como probar el microcontrolador, el software, el comportamiento del motor, etc. Además, permitieron avanzar con el desarrollo del software antes de avanzar demasiado con el diseño del hardware, ya que ofrecían la posibilidad de probar los cambios realizados de manera inmediata.

Las tarjetas adquiridas fueron las siguientes:

- dsPICDEM MCLV-2 de Microchip
- dsPICDEM MCHV-2 de Microchip

Imagen 3: MCHV-2

Imagen 4: MCLV-2

Por otra parte, en lo que al diseño de la tarjeta PCB a desarrollar se refiere, este se encontraba en una fase muy temprana. Se había decidido realizar dicho diseño basándose en el de la tarjeta de evaluación dsPICDEM MCHV-2, de la que se disponía de sus esquemáticos, los cuales se habían usado para evaluar qué partes eran necesarias y qué partes no lo eran. No obstante, no se había realizado hasta la fecha ningún otro avance en este campo.

4.2. Software

En cuanto al software, se había decidido partir de un programa escrito en C y ensamblador, facilitado por el propio fabricante del microcontrolador, y que está especialmente adaptado para ser utilizado con la familia de microcontroladores elegida para el proyecto.

Este código es el que acompaña a la nota de aplicación AN1078 de Microchip, disponible en la siguiente dirección de internet:

> http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en530042

El programa contiene el código necesario para realizar un control vectorial de un motor síncrono de imanes permanentes, utilizando las tarjetas de evaluación arriba descritas (o una tarjeta que haga uso de las mismas entradas y salidas del micro).

No obstante, el programa presentaba algunas carencias con respecto a la función para la que se pretendía utilizar:

- Al ser un programa con una función únicamente de evaluación, no está preparado para manejar las órdenes necesarias para operar una puerta de ascensor: apertura y cierre, ambos con sus respectivas curvas de velocidades, detección de obstáculos, etc. Solamente admite operaciones de arranque y paro, con una velocidad ajustada manualmente mediante un potenciómetro y en un solo sentido de giro.
- El programa está pensado para ser utilizado para el manejo de motores sin sensores de posición y velocidad (sensorless). Como el control vectorial necesita esta información, el programa realiza una estimación de estos valores en base a los valores medidos de fuerza contraelectromotriz en las fases del estátor.

Por lo tanto, es fácil deducir que las modificaciones a realizar sobre dicho código debían estar destinadas a lo siguiente:

- Permitir la recepción de órdenes de apertura, cierre, etc.
- Controlar la velocidad de referencia del motor de forma automática y en función de la orden recibida, la posición de la puerta, las configuraciones tales como curvas de velocidades, etc.
- Utilizar el encoder como sensor de posición y velocidad, y utilización de estos valores para los cálculos propios del control vectorial, en lugar de los valores estimados.
- Detección de obstáculos
- Etc

A continuación se detallarán las modificaciones realizadas al programa original en el momento de que el autor de este Trabajo Fin de Grado se incorporara al proyecto. Una descripción detallada del programa original se desarrolla en el capítulo “Descripción del programa original”.

4.2.1. Modificaciones para uso del encoder

La principal modificación llevada a cabo en el programa era la que permitía el uso del encoder de cuadratura para calcular la posición del rotor, configurando los registros apropiados para el uso de los puertos del microcontrolador, así como los parámetros de configuración para la comunicación con el encoder, para la que el microcontrolador viene especialmente preparado mediante el denominado *quadrature encoder interface* (QEI).

Mediante el encoder, y haciendo uso de las funciones para lectura, escritura y configuración del encoder y sus registros asociados (registros del *QEI*) se realizaba el cálculo de la posición angular y la velocidad angular.

También se había “puenteado” el uso del algoritmo de estimación de dichos parámetros, forzando al programa a utilizar los valores medidos en lugar de los estimados.

No obstante, aunque la posición angular se calculaba correctamente, no estaba correctamente implementado el cálculo de la posición de la puerta, el cual debe hacerse también mediante el encoder como requisito indispensable del proyecto. El fallo cometido era el siguiente: cuando el motor gira en sentido negativo, el contador de pulsos del encoder se va decrementando. Esto provocaba que al girar en este sentido, el contador de pulsos pasara de 0 a 0xFFFFFFFF (underflow), a pesar de que el rango esperado al medir el ángulo debería estar entre 0 y 4095, ya que el encoder tiene una resolución de 4096 pulsos por revolución.

4.2.2. Modificaciones para uso de velocidad de referencia calculada automáticamente

También se había programado un modo de funcionamiento en el que se establecía la velocidad de referencia mediante código, pero su finalidad era tan solo la de probar el movimiento de la puerta, y no estaba preparado para recibir órdenes de apertura, cierre, etc.

4.2.3. Modificaciones para uso de la memoria flash

Por último, se habían realizado algunas pruebas sobre cómo guardar datos en la memoria flash del microcontrolador, ya que se deben guardar ciertas configuraciones en la misma. No obstante, nuevamente estas modificaciones no pasaban de ser meras pruebas que no cubrían los objetivos del proyecto.

5. Descripción del programa original

En este apartado se detalla el funcionamiento del código original, tal y como es facilitado por parte de Microchip, y por tanto antes de cualquier posible modificación.

Como ya se ha comentado, el programa lleva a cabo un control vectorial, con la peculiaridad de que realiza una estimación de la posición y velocidad del rotor mediante un algoritmo especialmente pensado para ello.

El diagrama de control llevado a cabo por el programa es la que se muestra en la ***figura 9.***

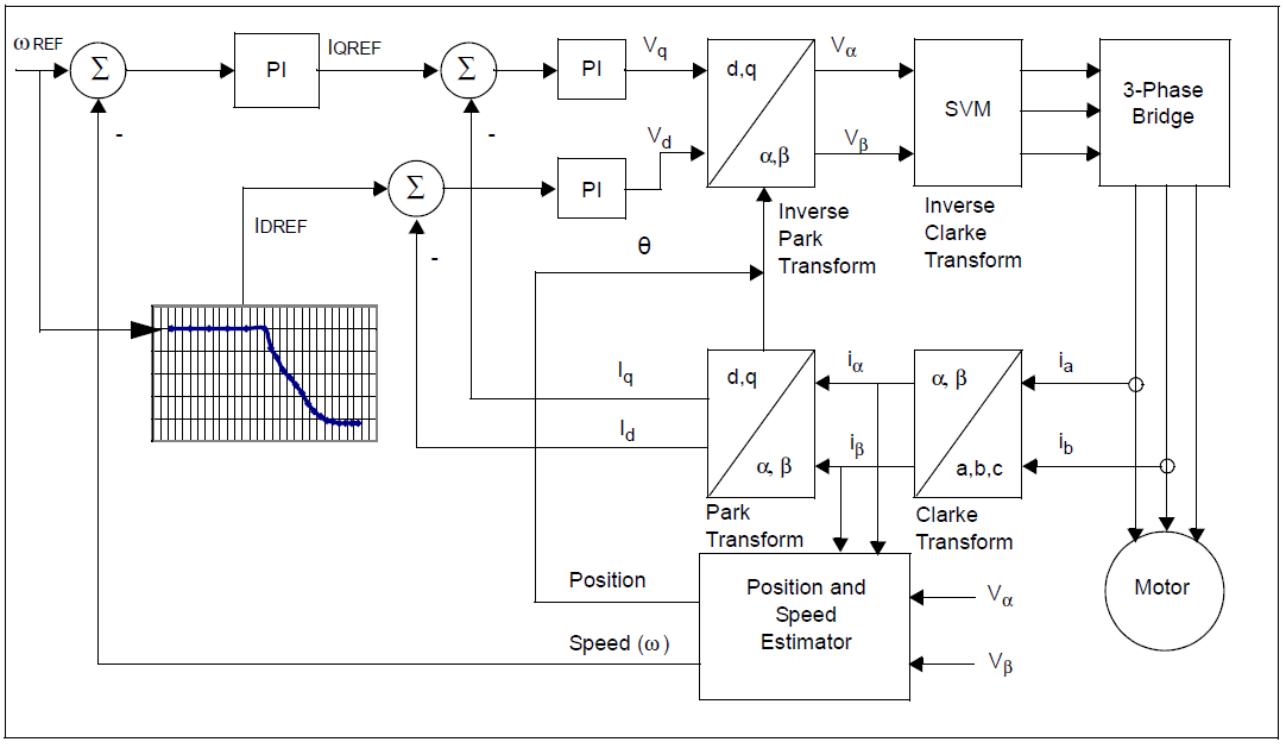


Figura 9: Diagrama de control (fuente: AN1078 - Microchip)

Una peculiaridad del programa es que, como necesita que el motor esté girando a una velocidad no demasiado baja para poder estimar la posición, en el arranque funciona en lazo abierto hasta que alcanza la velocidad necesaria para ello.

5.1. Variables principales

El programa lleva control del estado del sistema así como de distintos parámetros necesarios para la correcta realización de los cálculos mediante una serie de variables principales, las cuales se enumeran a continuación:

- **uGF**: variable de tipo struct cuyos bits son usados a modo de flags para controlar el modo de funcionamiento del programa. Flags más importantes que contiene:
  - **OpenLoop**: determina si se está funcionando en lazo abierto o en lazo cerrado (es necesario funcionar en lazo abierto hasta cierta velocidad para poder hacer uso del algoritmo de estimación)
  - **ChangeMode**: determina si es el primer bucle de control en lazo abierto o en lazo cerrado (dependiendo del flag OpenLoop será uno u otro). Necesario para inicializar otras variables a los valores adecuados en cada caso al entrar en cada uno de ellos.
  - **RunMotor**: determina si el motor debe estar o no parado.
  - **BtnXPressed**: determina si se ha pulsado determinado botón para llevar a cabo la acción asociada a dicha pulsación.
  - Otros
- **CtrlParm**: variable de tipo struct que contiene los valores de referencia de todos los controladores PI. Elementos que contiene:
  - **qVelRef**: referencia de velocidades
  - **qVdRef**: tensión de referencia de la componente D (flujo)
  - **qVqRef**: tensión de referencia de la componente Q (par)
- **PIParmD**, **PIParmQ** y **PIParmW**: variables del tipo **tPIParm**. Este tipo es en realidad una estructura (struct) que contiene los datos necesarios para llevar a cabo los cálculos propios de un controlador PI.

  Las tres variables se corresponden respectivamente con los controladores del lazo de corriente en el eje D (flujo), el lazo de corriente en el eje Q (par) y el lazo de velocidad angular.

  La estructura **tPIParm** contiene los siguientes elementos principales:
  - **qKp**: ganancia de la parte proporcional
  - **qKi**: ganancia de la parte integral
  - **qInRef**: valor instantáneo de la referencia
  - **qInMeas**: valor instantáneo de la rama de realimentación
  - **qOut**: valor instantáneo de la salida (resultado del cálculo del controlador)
  - **qOutMax**: valor máximo admisible a la salida

  - **qOutMin**: valor mínimo admisible a la salida
  - otros

- **smc1**: variable de tipo struct que contiene los parámetros y valores necesarios para calcular la estimación de posición y velocidad del rotor. Dicho cálculo entraña cierta complejidad, y dado que no se va a hacer uso del algoritmo de estimación, no merece la pena realizar una explicación detallada del mismo. No obstante, quien esté interesado en conocer su funcionamiento puede remitirse a la nota de aplicación AN1078 de Microchip.
- **AccumTheta**: variable utilizada como acumulador que representa el diferencial del ángulo de giro, utilizado para calcular la velocidad angular.
- **PrevTheta**: valor anterior del ángulo de giro, utilizado para conocer su incremento entre dos iteraciones del lazo de control y sumarlo a **AccumTheta**.
- **AccumThetaCnt**: contador del número de iteraciones durante las que se ha ido incrementando **AccumTheta.**
- **ParkParm**: variable de tipo struct que contiene los parámetros y valores necesarios para calcular las transformadas de Clarke y Park, así como sus inversas, propias del control vectorial.

  Sus elementos principales son los siguientes:

  - **qAngle**: posición angular del rotor (necesaria para calcular las transformadas)
  - **qIa**, **qIb**: corrientes medidas de fase (corrientes reales)
  - **qIalpha**, **qIbeta**, **qValpha**, **qVbeta**: valores en el “espacio de Clarke”. Es decir, tras aplicar la transformada de Clarke o antes de aplicar la transformada inversa de Clarke.
  - **qId**, **qIq**, **qVd**, **qVq**: valores en el “espacio de Park”. Es decir, tras aplicar la transformada de Park o antes de aplicar la transformada inversa de Park. Nota: estos son los valores que manejan los controladores PI.
  - **qV1**, **qV2**, **qV3**: valores reales de tensión que se utilizan en la modulación PWM, calculados mediante las transformadas inversas de Park y Clarke.
  - Otros

- **SVGenParm**: contiene los valores calculados para la modulación de espacios vectoriales, en los siguientes elementos:

  - **iPWMPeriod**: periodo de conmutación PWM (en número de instrucciones del procesador)

- **qVr1**, **qVr2**, **qVr3**: tensiones de referencia para calcular los ciclos de trabajo de las 3 células PWM.

Nota importante:

No se han especificado las unidades en que se almacena ninguna de las variables anteriores. Esto es debido a que la inmensa mayoría no se almacenan utilizando magnitudes físicas reconocibles.

En su lugar se almacenan como número binario, de tal forma que, aprovechando los 16 bits de los registros del microcontrolador, abarcan distintos rangos de valores, escalados para estar representados entre los valores 0x0000 y 0xFFFF.

De esta manera, el programa consigue que el microcontrolador pueda realizar las operaciones de forma mucho más eficiente, y evitando la necesidad de conversiones de tipo de variable (casting).

A esto Microchip le llama formato de coma fija 1.15, ya que conceptualmente puede entenderse como una representación del rango de números reales -1.0 a 0.9999, organizados de tal forma que el primer bit representa el signo, y los 15 bits restantes la parte fraccional, considerando siempre la parte entera igual a cero (excepto 0xFFFF que equivale a -1.0).

No obstante, lo anterior es, como se decía, una explicación conceptual, ya que en la práctica el tipo de dato utilizado es el de ***signed int****, lo que permite, como ya se ha mencionado, un mayor rendimiento en las operaciones matemáticas.*

5.2. Funciones principales

El funcionamiento del programa se vertebra principalmente alrededor de una serie de funciones, las cuales son sin lugar a duda las más importantes de dicho programa, ya que son las que realizan las tareas principales del mismo, o bien llaman a otras funciones secundarias para ello, controlando todo el proceso. Todas estas funciones principales se detallan a continuación, explicando brevemente cual es su cometido:

- **main**: inicializa algunas variables necesarias para el funcionamiento del programa y configura las interrupciones. Después, comprueba constantemente si se ha pulsado alguno de los botones.
- **_AD1Interrupt**: interrupción que se ejecuta una vez por cada conmutación del PWM. Esta función es la que realiza en orden las llamadas a las demás funciones que intervienen en el control, por lo que se podría decir que es la función principal que ejecuta el control.

- **CalculateParkAngle**: calcula el ángulo del rotor para poder hacer la transformada y la antitransformada de Park. Utiliza los valores obtenidos con el algoritmo de estimación.

  Función llamada desde _AD1Interrupt.

- **SMC_Position_Estimation_Inline**: ejecuta el algoritmo de estimación de posición y velocidad.

  Función llamada desde CalculateParkAngle.

- **ClarkePark**, **InvPark**: hacen el cálculo de las transformadas y antitransformadas de Clarke y Park.

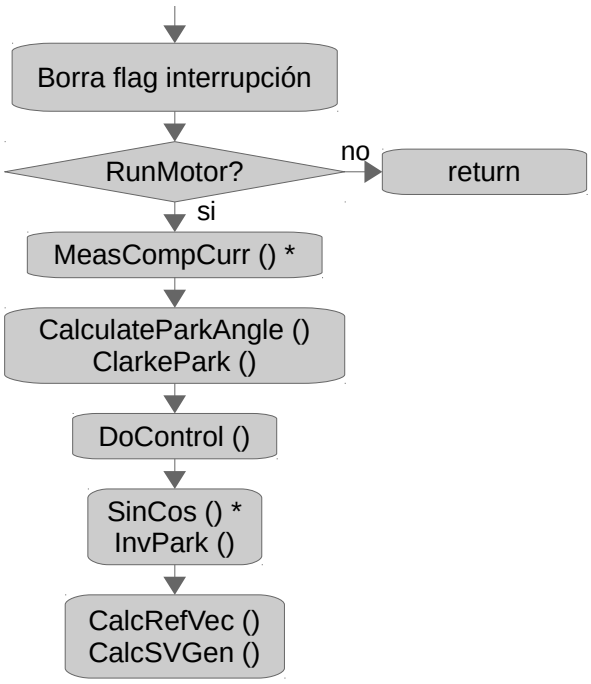
  Funciones llamadas desde _AD1Interrupt.

- **DoControl**: hace el cálculo de los valores obtenidos en los PI's. Tiene en cuenta si se está en lazo abierto o en lazo cerrado, ya que los valores de referencia son diferentes.

  Función llamada desde _AD1Interrupt.

- **CalcRefVec**, **CalcSVGen**: hace el cálculo de los ciclos de trabajo en función del vector de tensión hallado.

  Funciones llamadas desde _AD1Interrupt.



** MeasCompCurr: mide corrientes de fase*
SinCos: calcula el seno y el coseno del ángulo para calcular la transformada de Park

Figura 10: diagrama de flujo de la función _AD1Interrupt

6. Software desarrollado

6.1. Características a implementar

Partiendo del programa original de Microchip (AN1078) se deben realizar las modificaciones oportunas para poder operar una puerta de ascensor, para lo que debe incluir las siguientes características:

- **Control de posición angular del rotor**, necesario para el cálculo de la transformada de Park. Debe usarse el encoder del motor en lugar del algoritmo de estimación original del programa de Microchip, ya que este solo da valores fiables a altas velocidades.
- **Control de posición de la puerta**. Se debe utilizar para ello el encoder incorporado en el motor, no se disponen de señales externas como podrían ser las enviadas por sendos finales de carrera.
- **Proceso de aprendizaje de los distintos parámetros** necesarios para el correcto funcionamiento del motor y de la puerta: offset entre el ángulo mecánico y el eléctrico y longitud de la puerta.
- **Control de velocidad**. Se debe utilizar el encoder para controlar el giro del motor a las velocidades angulares deseadas. La apertura y el cierre de la puerta debe realizarse en base a una curva de velocidades cuya finalidad es la de conseguir un movimiento suave pero rápido, con baja velocidad al principio, aceleración hasta la velocidad máxima y posterior deceleración y cierre completo.
- **Respuesta del sistema rápida y sin excesivas sobreoscilaciones**
- **Rápida detección de obstáculos**, produciéndose ante un impacto una fuerza máxima de 150N (valor establecido por normativa). Esta detección es independiente de la que pueda haber mediante sensores de infrarrojos o similares.
- **Parámetros** de funcionamiento, velocidades, opciones, etc **configurables a través de un menú**, utilizando un display y una botonera.
- **Guardado en memoria permanente** (memoria flash del microcontrolador) de los parámetros obtenidos en el aprendizaje, así como los parámetros configurables vía menú. Esto permite cargarlos tras un corte de luz sin necesidad de repetir el proceso de aprendizaje ni reconfigurar los parámetros.
- **Recepción de órdenes desde el control de maniobra** del ascensor, para determinar cuál es la acción a realizar en cada momento: apertura, cierre, apertura de emergencia, etc.

A continuación se pasa a detallar cómo es el funcionamiento de cada una de las partes del programa desarrolladas por quien redacta el presente informe. El código propiamente se adjunta al presente informe en soporte digital, así como se adjuntan en el anexo 1, en soporte impreso, las partes más importantes desarrolladas por el alumno.

6.2. Relación del código desarrollado

El código desarrollado en el trabajo descrito en este documento se encuentra en los siguientes archivos:

- Archivos nuevos:
  - botonera.c + botonera.h
  - controlPuerta.c + controlPuerta.h
  - displays.c + displays.h
  - encoder.c + encoder.h
  - entradas.c + entradas.h
  - flash_handler.c + flash_handler.h
  - obstaculo.c + obstaculo.h
  - user_options_input.c + user_options_input.h
- Archivos modificados:
  - PMSM.c
  - general.h
  - UserParms.h
  - initdspic.c

6.3. Control de la posición angular del rotor mediante el encoder

Para poder utilizar el control vectorial sobre el motor, es necesario conocer en todo momento la posición angular del rotor, necesario para calcular la transformada de Park, tal y como se explica en el capítulo “Control vectorial”.

Como ya se ha explicado anteriormente, el programa original de Microchip, al estar pensado para aplicaciones *sensorless*, o sin sensores, no realizaba el cálculo de la posición angular del rotor de forma empírica, sino que la estimaba a partir de los valores medidos de fuerza contraelectromotriz.

No obstante, aunque esto supone un ahorro material, no es válido para cualquier aplicación, ya que el algoritmo de estimación solamente es eficaz a velocidades elevadas, lo que invalida su uso para determinadas aplicaciones.

Tal es el caso de la presente aplicación. Así pues, es necesario calcular la posición angular utilizando el encoder de cuadratura que viene incorporado junto al motor para tal fin.

Dicho encoder nos permite calcular la posición mecánica del rotor, pero para la aplicación se requiere el valor de la posición eléctrica del mismo. Para hallar la segunda a partir de la primera, es necesario tener en cuenta lo siguiente:

- Ambos valores se calculan y/o almacenan en memoria utilizando diferentes unidades:
  - La posición mecánica se calcula con el encoder, y la podemos obtener en número de pulsos del encoder. Este valor varía de 0 a 4095 a lo largo de una revolución completa del rotor.
  - La posición eléctrica se guarda en un rango de valores que aprovecha los 16bits de los registros del microcontrolador, en una variable del tipo **signed in**t, por lo que su valor varía entre -32768 a 32767. Este es un rango ficticio de valores, que representa el rango de ángulos comprendido entre -π y π radianes, pero aprovechando toda la resolución de los 16 bits.
- La posición eléctrica se halla multiplicando la posición mecánica por el número de pares de polos del motor (expresadas en las mismas unidades). En el motor de la presente aplicación, el número de pares de polos del motor es de 8.
- Los ejes eléctrico y mecánico es probable que no se encuentren alineados, por lo que es necesario calcular el offset que existe entre ellos. El eje mecánico se corresponde a la posición del rotor en la que el encoder mande la señal de índice detectado. El eje eléctrico se corresponde al vector director 100 (ver capítulo “Modulación de espacios vectoriales”).

6.3.1. Conversión de ángulo mecánico a eléctrico

Por lo tanto, para convertir el ángulo mecánico medido con el encoder a ángulo eléctrico en las unidades requeridas con el programa es necesario realizar la siguiente conversión:

$$X\ pulsos \cdot \frac{2^{16}\ uds./rev.}{4096\ pulsos/rev.} \cdot 8\ pp = 128 \cdot X \tag{7}$$

No obstante, probando la conversión anterior para un valor relativamente alto de pulsos, por ejemplo 320 pulsos, puede calcularse que el resultado obtenido es 40960. Este valor excede el rango máximo de una variable de tipo signed de 16 bits, que es 32767 (equivalente en este caso a π rad).

Además, si utilizamos un valor aún más alto, por ejemplo 1000 pulsos, puede calcularse que el resultado obtenido es de 128000, que excede con mucho el valor máximo de una variable de 16 bits, aun si esta fuera de tipo unsigned (valor máximo de unsigned int: 65536).

Sin embargo, sabemos que a los ángulos comprendidos fuera del rango -π a π se les puede restar 2π rad tantas veces como se desee hasta que se obtenga un valor dentro de dicho rango, que será totalmente equivalente al original. Lo mismo se puede hacer en este caso escalando los valores a las unidades utilizadas, por lo que podemos restar 2^{16} hasta obtener un valor comprendido entre -32768 y 32767. De esta forma, obtenemos el valor -3072, equivalente por tanto a 128000.

Este cálculo de la posición angular del rotor se ha añadido en la función “CalculateParkAngle” del software original, sustituyendo a la función “EstimatePosition”.

6.3.2. Cálculo del offset entre los ejes mecánicos y eléctricos

Por otro lado, a la conversión anterior habrá que sumarle el ángulo de offset existente entre los ejes mecánicos y eléctricos, expresados en ángulos eléctricos.

Este offset debe ser calculado de forma previa a la instalación del motor en la puerta, y debe hacerse con cada unidad del motor que se vaya a utilizar, ya que puede variar de uno a otro. Basta con calcularlo una vez, guardar su valor en la memoria flash del microcontrolador, y recuperar dicho valor cuando se necesite.

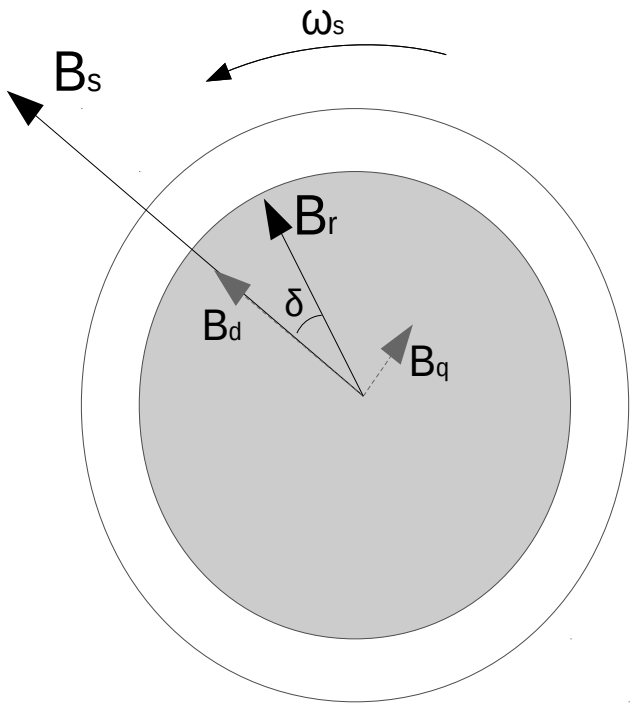


Figura 11: Interacción entre los campos magnéticos del rotor y el estator

El cálculo se realiza haciendo girar el motor forzándole a una posición (eléctrica) conocida en cada instante, y calculando la diferencia respecto al ángulo eléctrico calculado a partir de los valores del encoder. Dicha diferencia es el offset existente entre los ejes mecánicos y eléctricos.

Es importante destacar que este proceso debe llevarse a cabo sin carga en el eje del motor. El motivo de ello se explica a continuación.

Los motores síncronos giran a la velocidad de sincronismo, de tal forma que los campos magnéticos del estátor y del rótor giran a la misma velocidad. No obstante, ambos campos no están alineados, sino que se encuentran desfasados un ángulo δ. Este desfase es, precisamente, lo que permite al motor síncrono desarrollar el par motor, ya que solo dos campos magnéticos en cuadratura generan par. Véase la figura 11.

Así pues, para calcular el desfase, es necesario que no se esté desarrollando ningún par, o se cometería un error de valor δ.

Esta funcionalidad se ha introducido en la función “CalculateParkAngle” del software original, de forma que si se inicia el movimiento en lazo abierto, es decir, forzando el ángulo del rotor en cada momento, se puede calcular la diferencia entre el ángulo que se está forzando (por tanto es conocido) y el ángulo calculado a partir de las mediciones del encoder, obteniendo así el desfase entre ambos.

6.3.3. Otros problemas solucionados

El código necesario para implementar lo anterior se encontraba desarrollado en el momento de iniciar el presente Trabajo Fin de Grado, con un funcionamiento óptimo en casi todos los aspectos.

No obstante, hubo que resolver dos pequeños fallos, que se detallan a continuación:

- El sentido de giro considerado como positivo podría no coincidir en los campos mecánicos y eléctricos. Cambiando dos de las fases del motor, al aplicar un sentido de giro positivo, este giraría en sentido contrario al original, y el encoder mediría por tanto un giro mecánico negativo.
- Al pasar por el índice reinicia el contador de pulsos, quedando este a cero. El problema es que cuando el rotor gira en sentido negativo, tras reiniciar, cuando se resta un pulso se produce un underflow, es decir, pasa de 0x00000000 a 0xFFFFFFFF (valores en base hexadecimal). Esto puede equivaler a -1 o a un número altísimo, según si la variable es signed almacenada en complemento a dos o unsigned, pero este valor no es coherente, ya que debería estar entre 0 y 4095.

En cuanto al primer punto, deben tenerse en cuenta varios factores:

- El orden de fases que el sofware considera positivo. Como la modulación viene ya definida en el software original y su funcionamiento es óptimo, se optó por no modificarlo.

- El orden en el que se coloquen los conectores de las fases del motor en la tarjeta.
- El orden en que se conecten las fases del motor a los conectores anteriores
- El sentido de giro que el módulo QEI (quadrature encoder interface) considere positivo. Según la documentación de referencia de la familia dsPIC33E, este parámatro es configurable.

Se barajó la opción de implementar un código que detecte el sentido de giro correcto y actuara en consecuencia, pero se declinó para evitar posibles confusiones al instalador de la puerta, en caso de que se den situaciones no previstas por el programador.

La solución adoptada es la de exigir al usuario de la tarjeta que conecte las fases del motor en el orden adecuado (fases UVW conectadas en ese orden) y preconfigurar por software el sentido de giro considerado como positivo por el encoder. Se adopta esta solución por considerarse la más sencilla a la par que robusta.

En lo que se refiere a evitar el underflow durante el giro negativo del encoder, la solución resulta bastante sencilla. Es posible configurar una función de interrupción cuando el encoder detecte un índice. En dicha función basta con comprobar el sentido de giro y, si este es negativo, sustituir el valor del contador de pulsos del encoder, que acaba de ser reiniciado a 0, por 4096. De esta forma, cuando reste el siguiente pulso, el valor pasará a 4095, y no a -1.

El código desarrollado correspondiente al cálculo de la posición angular se implementó en los archivos “encoder.h” y “encoder.c”, y el código correspondiente al cálculo del desfase entre los ángulos mecánico y eléctrico en el archivo principal del programa, “PMSM.c”. Ambos se adjuntan en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.4. Control de posición de la puerta

Haciendo uso de las señales del encoder, el software debe ser capaz de determinar en todo momento la posición actual de la puerta, de forma que pueda saberse cuando está a punto de completar el cierre o la apertura, cuando debe acelerar o decelerar, etc.

Para ello, es necesario lo siguiente:

- Fijar una posición de referencia, respecto a la cual se mida la posición. Esta es la posición cero. La posición de referencia debe ser la de completamente cerrada y con el patín también cerrado (a veces puede ser necesario que una vez cerrada la puerta, el motor deba girar más para cerrar el patín).
- Conocer la longitud total de la puerta. Para hallar este dato, es necesario llevar a cabo el proceso de aprendizaje por parte del instalador. Este dato se almacena en la memoria flash, de forma que el proceso de aprendizaje solo deba realizarse una vez.

Por comodidad y rapidez de ejecución, tanto la longitud de la puerta como su posición se medirán en número de pulsos respecto a la posición cero.

Debe tenerse en cuenta que, según en que lado de la puerta se ponga el motor, el sentido de apertura será un giro positivo o negativo. En caso de que sea negativo, deberá configurarse de tal manera desde el menú de opciones, ya que el software no tiene forma de determinar en qué sentido se mueve la puerta para determinado sentido de giro, al disponer tan solo del encoder para determinar la posición.

A fin de calcular correctamente la posición angular del rotor, el contador de pulsos se reinicia cada vez que se detecta la señal de índice del encoder, de forma que dicho contador de pulsos marca el ángulo pero no la posición total.

Para calcular la posición total deben tenerse en cuenta los siguientes valores:

- Ángulo inicial. Es el ángulo del rotor, en número de pulsos, parado en la posición cero (posición puerta cerrada).
- Ángulo final. Es el ángulo del rotor, en número de pulsos, parado en la posición final (posición puerta abierta).
- Número de vueltas completas. Para hallar el número de pulsos que suponen, basta con multiplicar por 4096pul/rev.

Conociendo los valores anteriores, podemos calcular el número de pulsos totales, teniendo en cuenta los 3 distintos tramos que recorre el rotor:

- Tramo inicial: Nº de pulsos de la vuelta incompleta del principio del movimiento. El número de pulsos de este tramo se calcula tal y como se describe en la figura 12.

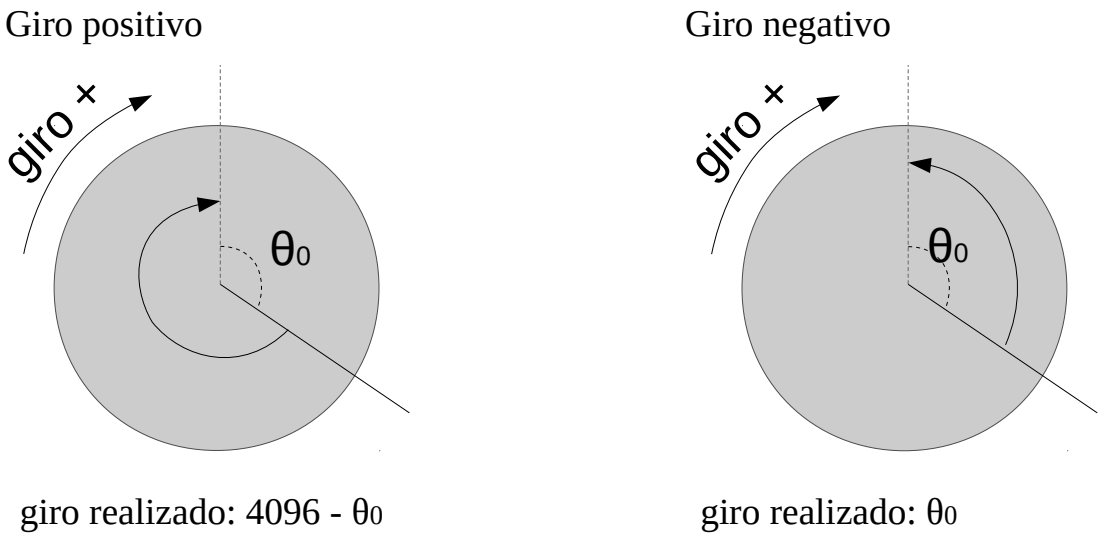


Figura 12: giro realizado durante la vuelta incompleta del principio del movimiento

- Tramo medio: Nº de vueltas completas realizadas multiplicado por 4096 pulsos/vuelta. Para calcular el número de vueltas completas, basta con contar el número de índices detectados desde la posición cero, y si es mayor o igual a 1, restarle 1 (se le resta 1 para restarle el primer índice detectado, que es una vuelta incompleta).
- Tramo final: Nº de pulsos de la vuelta incompleta del final del movimiento. El número de pulsos de este tramo se calcula tal y como se describe en la figura 13.

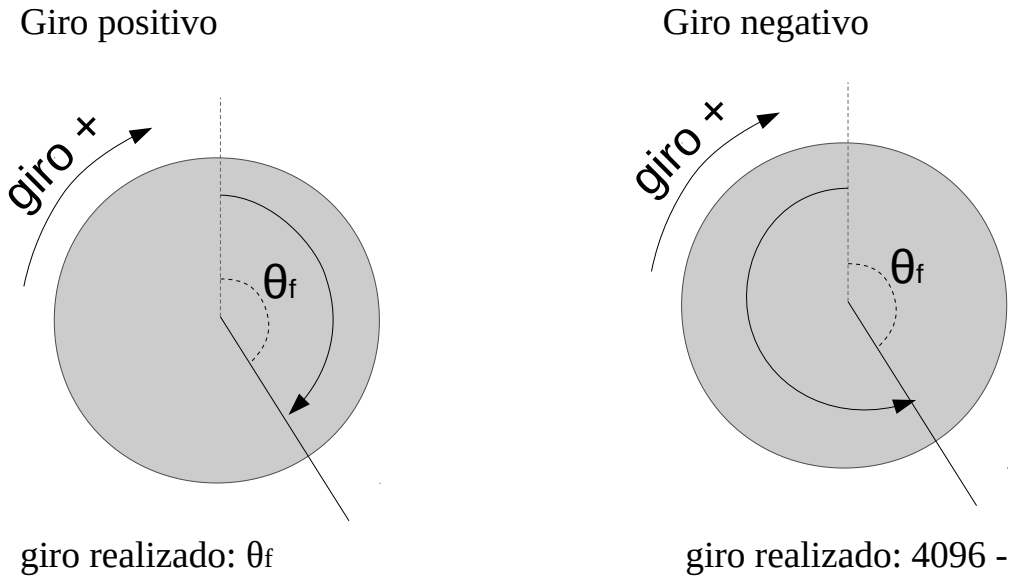


Figura 13: giro realizado durante la vuelta incompleta del final del movimiento

Según lo anterior se puede saber cómo calcular el número de pulsos de cada uno de los 3 tramos en función de si el sentido de apertura es positivo o negativo. Se puede calcular la posición de la puerta de la siguiente manera:

- Si el conteo de índices es igual a cero, para sentido de apertura tanto negativo como positivo:

$$posición = ángulofinal - ánguloinicial = \theta_f - \theta_0 \quad (8)$$

- Si el conteo de índices no es cero, para sentido de apertura positivo:

$$posición = tramo1 + tramo2 + tramo3 = (conteoíndices - 1)\cdot 4096 + (4096 - \theta_0) + \theta_f \quad (9)$$

- Si el conteo de índices no es cero, para sentido de apertura negativo:

$$posición = -tramo1 - tramo2 - tramo3 = (conteoíndices + 1)\cdot 4096 - \theta_0 - (4096 - \theta_f) \quad (10)$$

** nota: en apertura en sentido negativo, tanto posición como conteo de índices son valores negativos*

Simplificando las tres expresiones anteriores , vemos que se pueden expresar en una sola, que por tanto abarcaría todos los casos:

$$posición = conteodeíndices \cdot 4096 - \theta_0 + \theta_f \quad (11)$$

Esto tiene la ventaja de que no es necesario tener en cuenta si el sentido de apertura es negativo o positivo para calcular la posición (aunque sí hará falta en otras partes del código para determinar el sentido de la velocidad de referencia).

En el software se implementó la funcionalidad de cálculo de la posición en el archivo “encoder.c”, donde se escribió una función que hace el cálculo tal y como se ha descrito. Esta función necesitó de unas variables auxiliares para almacenar el ángulo inicial, así como el conteo de índices. El valor del ángulo inicial debe haberse almacenado de forma previa al cálculo de la posición, con la puerta completamente cerrada.

En lo que a precisión se refiere, el encoder utilizado tiene un desempeño realmente bueno. A modo de ejemplo, en la puerta en la que se realizaron las pruebas el recorrido total era de aproximadamente 71000 pulsos. En dicha puerta, tras un movimiento de apertura y uno de cierre, la posición final de la puerta es de ±10 pulsos en todos los casos. Esto supone una precisión mayor al 0,015%.

No obstante, a lo largo de un periodo largo de funcionamiento, los pequeños errores podrían acabar acumulándose, de forma que la posición calculada mediante software acabe siendo errónea. El riesgo es mayor cuando la puerta se topa con obstáculos, los cuales pueden provocar errores mayores.

Para solucionar este problema, cada vez que la puerta hace un movimiento de cierre completo (lo que significa que el error entra dentro del margen tolerable) se reinicia a cero el contador de pulsos que indica la posición. De igual manera, cada vez que la puerta hace un movimiento de apertura completo, el contador de pulsos que indica la posición se establece al valor de la longitud total de la puerta. De esta manera, nunca se acumulan errores entre movimientos sucesivos.

Las funciones y variables necesarias para la correcta lectura de los datos del encoder se implementaron en los archivos “encoder.h” y “encoder.c”, que se adjuntan en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.5. Aprendizaje de la longitud de la puerta

Para poder controlar en qué tramo del recorrido se encuentra la puerta en cada momento, resulta necesario, además de cuantificar cuánto se ha movido respecto a la posición de origen (posición de cierre), conocer la longitud total de dicho recorrido.

La posición y la longitud de la puerta es manejada por el software en número de pulsos del encoder.

Para poder calcular la longitud, se desarrolló el "proceso de aprendizaje de longitud", que el instalador de la puerta debe poner en marcha.

Este proceso es muy sencillo, y consiste en lo siguiente:

1. La puerta se cierra por completo, a una velocidad fija y baja, ya que no se conoce donde está el final del recorrido y por tanto no se sabe cuándo se debe decelerar.
2. Una vez alcanzada la posición de cierre, se fija dicha posición como posición de origen.
3. La puerta se abre por completo. Se considera que está abierta cuando se topa con un obstáculo que le impida avanzar.
4. Una vez abierta completamente la puerta, se guarda el valor de la posición actual. Si la puerta realmente se ha cerrado primero y abierto después por completo, este debería ser el valor de la longitud total de la puerta.
5. Se vuelve a cerrar por completo la puerta
6. Si la posición final es de aproximadamente cero, la puerta ha recorrido la misma distancia al abrir que al cerrar, y se da el valor medido de la longitud como bueno.

Como se puede deducir de la explicación del proceso de aprendizaje, no se puede distinguir si el obstáculo detectado es el final del recorrido o es una obstrucción en mitad del mismo. Por eso, el instalador debe asegurarse visualmente de que todo el proceso se ha realizado correctamente.

La implementación en software del proceso de aprendizaje se describe mediante el diagrama de flujo de la figura 14.

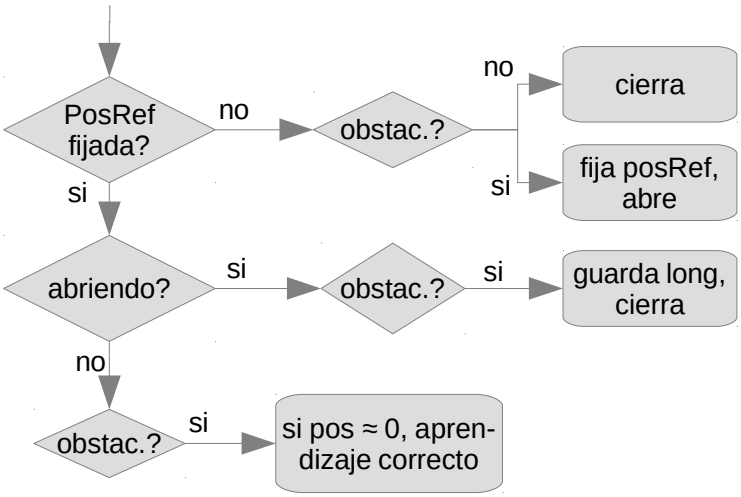


Figura 14: diagrama de flujo del proceso de aprendizaje

El código desarrollado correspondiente al proceso de aprendizaje se implementó en los archivos principal del programa, “controlPuerta.c” y “controlPuerta.h”, que se adjunta en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.6. Control de velocidad

El control de la velocidad se realiza mediante la técnica de control vectorial, tal y como se explica en el capítulo “Controladores PI”.

Estos controladores PI, tanto el de velocidad como los de corrientes, necesitan recibir una serie de parámetros de entrada, que son los siguientes:

- Velocidad de referencia
- Velocidad medida
- Corriente de cuadratura (corriente de par) medida
- Corriente directa (corriente de flujo) medida

L a **velocidad de referencia** se escoge de manera automática en función de los parámetros introducidos por el usuario, así como de la acción que se esté realizando, la posición de la puerta, etc.

No obstante, un factor a tener en cuenta a la hora de elegir la velocidad es cuál es el **sentido de giro** que realiza el cierre de la puerta y cual el que realiza la apertura. Esto varía en función del lado de la puerta en que se coloque el motor, el sistema de ruedas dentadas o poleas que utilice, etc.

Por ello, resulta necesario la inclusión de un parámetro configurable vía menú, según el cual se determine si el sentido de apertura es el que viene por defecto o no. Si se configura este parámetro como “sentido inverso”, la velocidad

L a **medición de velocidad** se realiza mediante el encoder. El módulo QEI del microcontrolador tiene un registro que permite medir la velocidad. Este registro es un contador que aumenta o disminuye en uno con cada pulso del encoder, y que se reinicia a cero automáticamente cada vez que se lee su valor.

De esta forma, es posible programar una interrupción de uno de los timers del microcontrolador para leer dicho valor, y a partir del mismo y de la frecuencia elegida para la interrupción se puede calcular la velocidad.

Las **corrientes directa y de cuadratura** no pueden calcularse directamente. Es necesario medir las corrientes de al menos dos de las tres fases del motor (la tercera se calcula a partir de la fórmula 12), y aplicarles las transformadas de Clarke y Park.

$$i_a + i_b + i_c = 0 \quad (12)$$

La medición de las corrientes de dos de las fases se ejecuta mediante dos resistencias shunt, cuya caida de tensión se filtra y amplifica con los amplificadores operacionales internos del propio microcontrolador.

El código desarrollado correspondiente al la medición de la velocidad se implementó en los archivos “encoder.h” y “encoder.c”. Ambos se adjuntan en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.6.1. Control de la puerta

La parte del software que controla las acciones de la puerta en cada momento, decidiendo en función de la orden recibida (aprendizaje, apertura, cierre...) se implementó en los archivos “controlPuerta.c” y “controlPuerta.h”, que se adjunta en el anexo 1 del presente documento, así como en soporte informático.

El diagrama de flujo que describe el funcionamiento del módulo de control de la puerta es el de la figura 15.

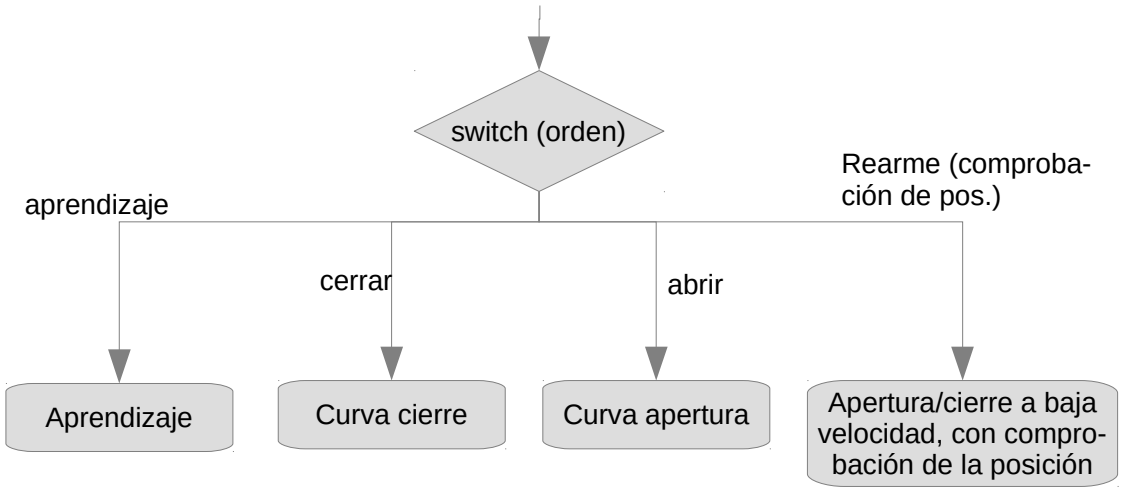


Figura 15: diagrama de flujo del control de la puerta

La orden de “rearme” se activa automáticamente cuando se recibe una orden de apertura o cierre, pero no se ha realizado previamente la comprobación de la posición, ya que si no se ha realizado no se pueden ejecutar los movimientos con curvas de velocidad. También se activa automáticamente si se detecta que la puerta ha realizado un movimiento de cierre más allá de la posición de referencia, o uno de apertura más allá de la longitud total de la puerta (ambos con un pequeño margen), ya que esto sería síntoma de que, por algún motivo, la posición de referencia no se encuentra fijada en el lugar correcto.

6.6.2. Curvas de velocidad en apertura y cierre

Durante los movimientos normales de apertura y cierre de la puerta se debe controlar la velocidad de la misma en función de la posición en que se encuentre. El objetivo de esto es poder fijar velocidades bajas en los tramos iniciales y final, y una velocidad alta durante el tramo medio, que normalmente es el más largo.

No obstante, tanto los valores de las velocidades como de la longitud de cada uno de los tramos deben ser configurables por parte del usuario, de forma que sea capaz de seleccionar la curva de velocidades que, a su juicio, más convenga para una determinada puerta, según la longitud de su recorrido, su peso, etc. La selección de estos parámetros se realiza vía menú. Todos los parámetros configurables de la curva pueden consultarse en los capítulos “Curva de velocidades en movimiento de apertura”, “Curva de velocidades en movimiento de cierre” y “Menú de opciones”.

La forma de conseguir que las velocidades de la puerta se ajusten a la curva de velocidades configurada consiste en fijar la velocidad de referencia en función de la posición actual de cada momento, la cual se mide en tiempo real.

Las curvas de velocidad son diferentes para los movimientos de apertura y cierre. Las curvas y sus respectivos parámetros configurables son las que se describen a continuación.

6.6.3. Curva de velocidades en movimiento de apertura

El movimiento de apertura tiene un total de cuatro tramos. Se inicia el movimiento a velocidad reducida durante una distancia relativamente pequeña, luego se acelera hasta la velocidad de referencia, que se mantiene durante la mayoría del recorrido. Finalmente decelera a una velocidad reducida, y cuando se completa la apertura, se mantiene un par constante para que la puerta permanezca en esa posición.

Este movimiento se describe más en detalle a continuación, y se muestra en la gráfica de la figura 16.

1. Tramo inicial:

- F1.01: longitud del tramo inicial
- F1.02: velocidad inicial

2. Segundo tramo:

- F1.03: tiempo que se tarda en acelerar de la velocidad inicial a la velocidad de referencia
- F1.04: velocidad de referencia
- F1.06: posición del punto final del segundo tramo

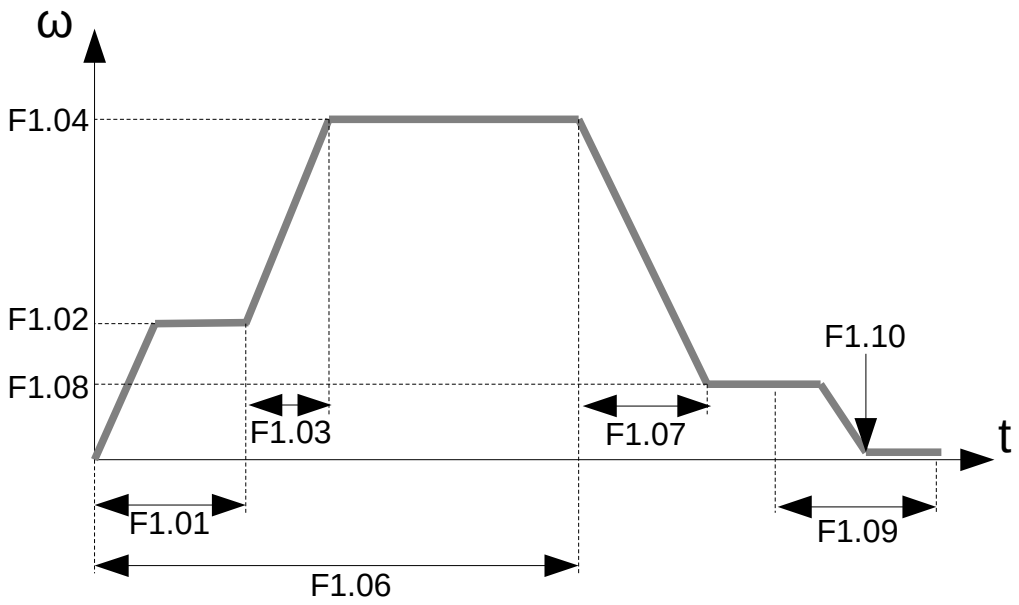


Figura 16: Curva de velocidades de apertura

3. Tramo final:

- F1.07: tiempo que se tarda en decelerar de la velocidad de referencia a la velocidad de finalización del movimiento
- F1.08: velocidad de finalización del movimiento
- F1.09: distancia respecto al final del recorrido por debajo de la cual se inicia el proceso de completado del movimiento de apertura. Durante ese proceso la obstrucción que se detecte se considerará como el final del recorrido, y se considerará que se ha completado la apertura.
- F1.10: límite en la corriente de par por encima de la cual se considerará que se ha detectado una obstrucción durante el proceso de completado de movimiento de apertura. Esta obstrucción se considerará como el final del recorrido.

4. Mantenimiento del par: una vez completado todo el movimiento de apertura, se debe aplicar un par constante, de valor igual al configurado en el parámetro F1.11 del menú de opciones.

La implementación en el software de la curva de apertura es la que se describe en la figura 17, y se encuentra en los archivos “controlPuerta.c” y “controlPuerta.h”. Dicho código se adjunta en el anexo 1, así como en los archivos entregados en soporte digital.

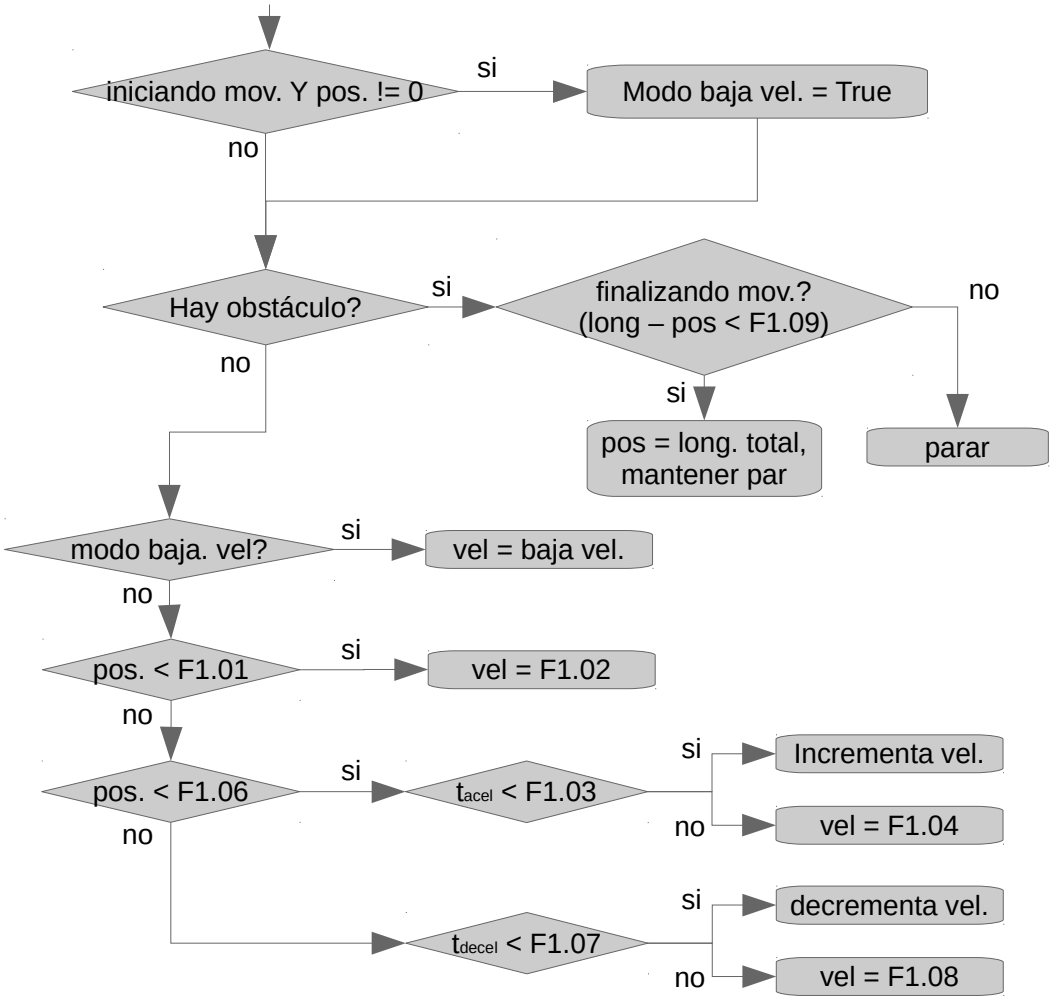


Figura 17: diagrama de flujo de la curva de apertura

6.6.4. Curva de velocidades en movimiento de cierre

El movimiento de cierre tiene un total de cinco tramos. Se inicia el movimiento a velocidad reducida durante una distancia relativamente pequeña, luego se acelera hasta la velocidad de referencia, que se mantiene durante la mayoría del recorrido. A continuación decelera a una velocidad reducida, y después se modifica la velocidad a la velocidad de bloqueo de la puerta. Finalmente, cuando se completa la apertura, se mantiene un par constante para que la puerta permanezca en esa posición.

Este movimiento se describe más en detalle a continuación, y se muestra en la gráfica de la figura 18.

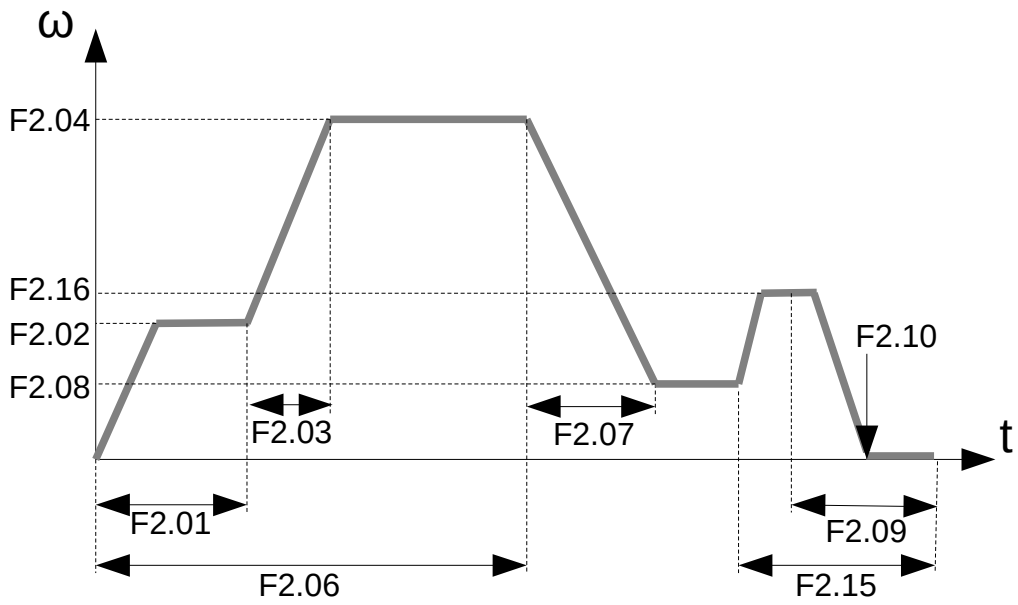


Figura 18: Curva de velocidades de cierre

1. Tramo inicial:

- F2.01: longitud del tramo inicial
- F2.02: velocidad inicial

2. Segundo tramo:

- F2.03: tiempo que se tarda en acelerar de la velocidad inicial a la velocidad de referencia
- F2.04: velocidad de referencia
- F2.06: posición del punto final del segundo tramo

3. Tramo final:

- F2.07: tiempo que se tarda en decelerar de la velocidad de referencia a la velocidad de finalización del movimiento
- F2.08: velocidad del tramo final del movimiento

4. Tramo de bloqueo de puerta: en este tramo el motor gira para bloquear la puerta, cerrando el patín. La puerta puede o no haber completado su recorrido.

- F2.15: distancia respecto al final del recorrido por debajo de la cual comienza el tramo de bloqueo de la puerta.
- F2.16: velocidad de bloqueo de la puerta

- F2.09: distancia respecto al final del recorrido por debajo de la cual se inicia el proceso de completado del movimiento de cierre. Durante ese proceso la obstrucción que se detecte se considerará como el final del recorrido, y se considerará que se ha completado el cierre.
- F2.10: límite en la corriente de par por encima de la cual se considerará que se ha detectado una obstrucción durante el proceso de completado de movimiento de cierre. Esta obstrucción se considerará como el final del recorrido.

5. Mantenimiento del par: una vez completado todo el movimiento de cierre, se debe aplicar un par constante, de valor igual al configurado en el parámetro F2.11 del menú de opciones.

La implementación en el software de la curva de apertura es la que se describe en la figura 19, y se encuentra en los archivos “controlPuerta.c” y “controlPuerta.h”. Dicho código se adjunta en el anexo 1, así como en los archivos en soporte digital.

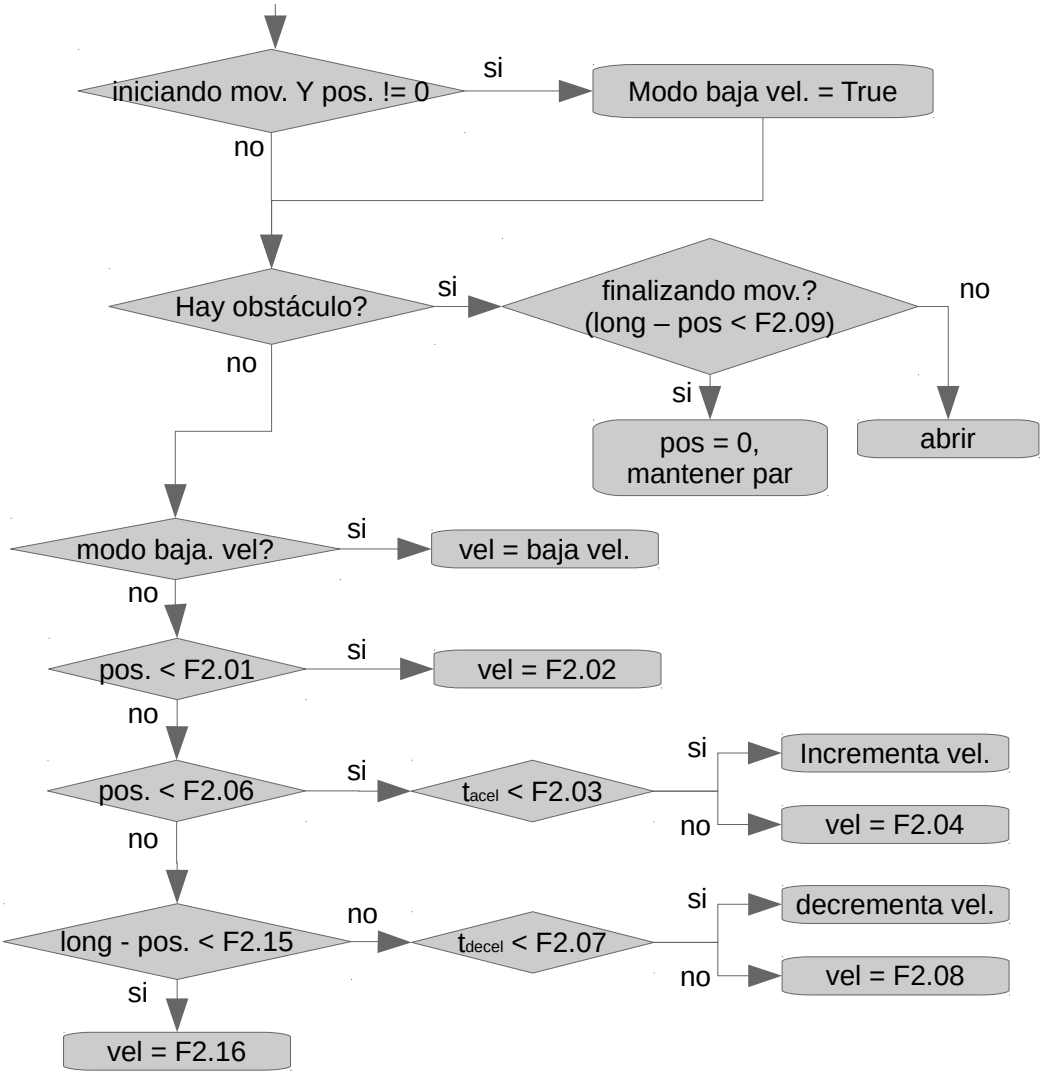


Figura 19: diagrama de flujo de la curva de cierre

Se realizaron ensayos para comprobar que la velocidad de la puerta se ajusta con suficiente precisión a la curva de velocidades arriba descritas. Pueden verse los resultados en el capítulo "Respuesta del sistema con curva de velocidades"

6.6.5. Movimiento de comprobación de la posición

Tras haberse producido un "reset" del microcontrolador, no se puede conocer la posición de la puerta en un primer instante, ya que como se explica en el capítulo "Control de posición de la puerta", esta se calcula en base al movimiento realizado, medido mediante el encoder, respecto a una posición de referencia. Como se ha producido un "reset", la información sobre la cantidad de movimiento realizada se ha perdido, y por lo tanto no puede saberse dónde está dicha posición de referencia.

Por este motivo, antes de poder realizar el control de velocidad en función de la posición (curvas de velocidad), es necesario encontrar la posición de referencia, para lo cual se implementa el modo "comprobación de la posición" en los procesos de apertura y cierre. Este proceso se activa automáticamente cuando no se conoce la ubicación de la posición de referencia, y se describe en la figura 20.

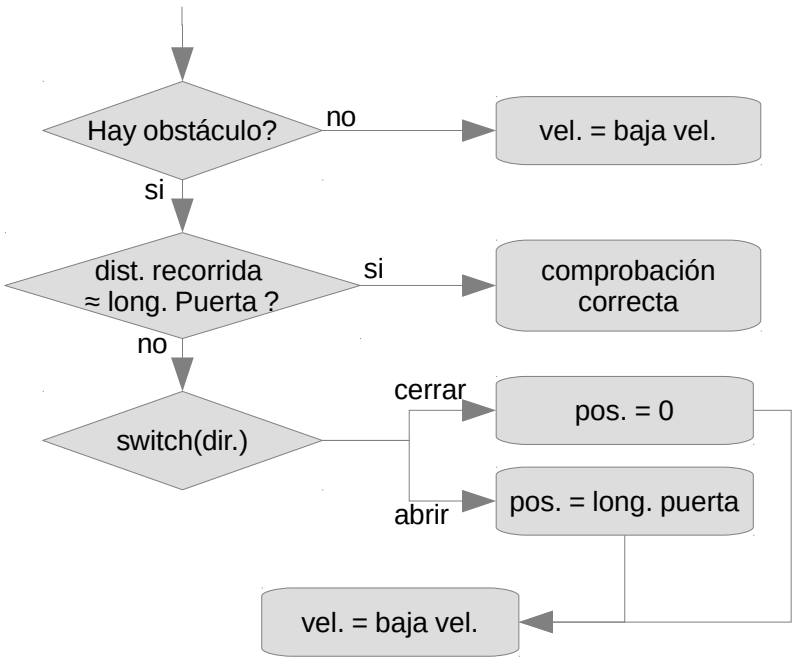


Figura 20: diagrama de flujo del movimiento de comprobación de la posición

6.7. Respuesta del sistema

El objetivo es conseguir una respuesta del sistema rápida, sin excesivas sobreoscilaciones y con un error en régimen permanente pequeño.

Para conseguir esto, es necesario ajustar correctamente los parámetros de los controladores PI.

En lugar de calcular esos parámetros, se puede realizar un ajuste manual a prueba y error, para lo cual hay que tener en cuenta lo siguiente:

- Un valor alto en el parámetro k_p supone una mejora en rapidez, pero empeora el error permanente.
- Un valor alto en el parámetro k_i supone una mejora en el error permanente, pero disminuye la rapidez del sistema y puede generar oscilaciones.
- El parámetro k_i debe ser, en la mayoría de los casos, notablemente más pequeño que el parámetro k_p, normalmente del orden de entre 5 y 10 veces menor.
- Los PI de lazos internos deben ser mucho más rápidos que los externos, al menos 10 veces más rápidos. De lo contrario no reaccionarían a tiempo ante los cambios en su valor de referencia. En este caso por tanto, los PI de corriente deben ser más rápidos que el PI de velocidad.

Sabiendo lo anterior, se pueden ajustar los parámetros de la siguiente manera, ajustando primero los internos, y a continuación los externos:

1. Se fija un valor de k_i nulo o muy bajo, y se ajusta el valor de kp más alto posible pero sin oscilaciones.
2. Se reduce ese valor de k_p a un 75% o un 50% del mismo, asegurándose de que la respuesta sea suficientemente rápida.
3. Se ajusta el valor de k_i para conseguir un valor bajo en el error permanente, intentando mantener dicho valor bajo en la medida de lo posible y sin que aparezcan excesivas oscilaciones

Los parámetros idóneos para los controladores PI varían en función de las características del sistema. Por ejemplo, en el caso de las puertas de ascensor, depende en gran medida del peso de la puerta, la fuerza ejercida por muelles o contrapesos, etc.

Por eso, resulta necesario ofrecer la posibilidad de configurar estos parámetros vía menú. No obstante, se dejarán configurados por defecto unos valores que resulten versátiles y válidos para la mayoría de los casos.

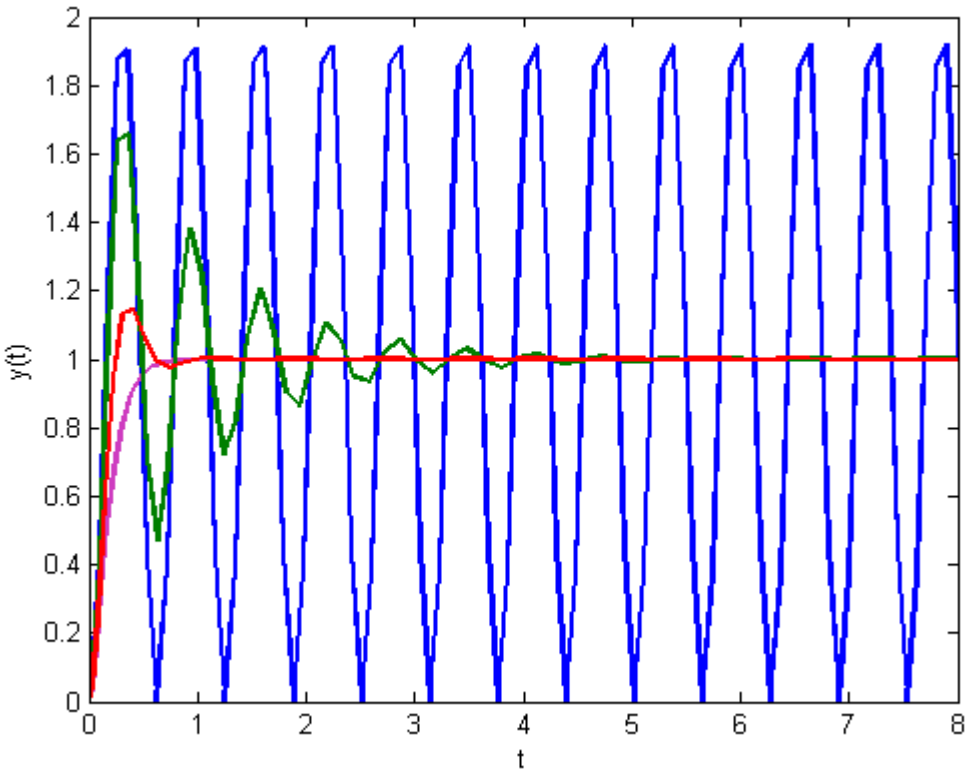


Figura 21: Distintos tipos de respuesta de sistemas

Las respuestas del sistema obtenidas en las pruebas con el motor y puertas reales de ascensor se recogen en el capítulo “Pruebas y ensayos realizados“.

6.8. Detección de obstáculos

Uno de los requerimientos que se deben cumplir para operar la puerta de ascensor es el de implementar un sistema que permita detectar obstrucciones que impidan o se opongan al movimiento de la puerta.

Esto es especialmente importante en el caso del cierre, por motivos de seguridad, ya que en este caso resulta probable que el obstáculo se trate de personas u objetos de valor.

La normativa sobre ascensores exige que la fuerza máxima que debe hacer la puerta durante el cierre es de 150N. Esto supone que la detección debe ser suficientemente rápida y que el golpe que realice antes de detenerse sea menor a ese valor.

El tiempo máximo de detección que se consideró óptimo es de aproximadamente 100ms.

También resulta necesario detectar la obstrucción en los últimos centímetros del recorrido de la puerta para determinar si la puerta está o no completamente abierta o cerrada, ya que la medición de la posición mediante el encoder es susceptible de pequeños errores.

Existen dos posibles métodos para detectar obstrucciones:

- Por variación de la velocidad
- Por variación del par

Ambos métodos fueron puestos a prueba para determinar cuál de los dos es el más óptimo, o bien una combinación de los dos.

6.8.1. Detección por variación de velocidad

La detección por variación de velocidad se basa en detectar si la velocidad medida baja de la velocidad de referencia. Sin embargo, esto supone algunas dificultades, que se describen a continuación.

La velocidad medida normalmente no se mantiene exactamente en la velocidad de referencia, debido al error en régimen permanente del control. Aunque este error sea pequeño, resulta difícil conseguir reducirlo a cero.

Además, en momentos en los que se producen pequeños cambios en la carga, circunstancia que se da por varios motivos a lo largo del recorrido de la puerta, se producen pequeños bajones en la velocidad durante breves intervalos de tiempo, hasta que el control reacciona a ese cambio.

Por otro lado, hay que tener en cuenta que el control es susceptible de producir oscilaciones al final del transitorio, lo que puede provocar que la velocidad baje en un momento puntual del límite de detección de obstáculo. Para solucionar este problema, es necesario ajustar correctamente los parámetros de los controladores PI. No obstante, esto no evita por completo que la velocidad caiga por debajo de la velocidad de referencia durante cortos periodos de tiempo.

Para solucionar estos inconvenientes, debe fijarse un margen de velocidad ligeramente por debajo de la velocidad de referencia, por debajo del cual se considera que la puerta está siendo ralentizada.

Tras varias pruebas sobre como fijar dicho margen, se llegó a la conclusión de que debía fijarse en base a varios parámetros:

- En base a la velocidad de referencia: cuando la velocidad de referencia es alta, resulta necesario oponer una fuerza mayor para ralentizarla, y cuando es baja, pequeñas fuerzas producen una ralentización considerable. Por ello, el margen de velocidad debe aumentarse de forma inversamente proporcional a la velocidad de referencia.
- En base a las oscilaciones del control: si las oscilaciones producidas por los controladores son grandes, debe darse un margen mayor. Por ello, el margen de velocidad debe aumentarse de forma directamente proporcional al parámetro Ki del controlador PI de velocidad, ya que cuanto mayor es este parámetro mayores son las oscilaciones.

- En base al error permanente: si el error permanente hace que el valor de la velocidad en régimen permanezca por debajo de la de referencia, el margen de velocidad que se había establecido queda notablemente reducido. Por eso, el margen de velocidad debe aumentarse de forma directamente proporcional al parámetro Kp del controlador PI de velocidad, ya que cuanto mayor es ese parámetro mayor es el error en régimen permanente.

En base a lo anterior, se determinó la siguiente ecuación para calcular el límite de detección de obstáculo:

$$VelLim = VelRef - margen = VelRef - \left(\frac{k_1}{VelRef} + k_2 \cdot k_{i,\omega} + k_3 \cdot k_{p,\omega}\right) \quad (13)$$

Siendo k_1, k_2 y k_3 constantes que se deben elegir para conseguir un ajuste óptimo, que consiga una sensibilidad suficiente para una rápida detección, pero suficientemente baja para evitar falsas detecciones.

En este tipo de detección merecen consideración aparte los tramos de aceleración. En estos tramos, se cambia la velocidad de referencia a una mayor que la actual, y a su vez el límite de detección varía a un valor ligeramente inferior a esta nueva velocidad de referencia. En un primer momento, la velocidad es inferior a dicho límite, y permanece por debajo hasta que el motor consigue acelerar. Véase la figura 22.

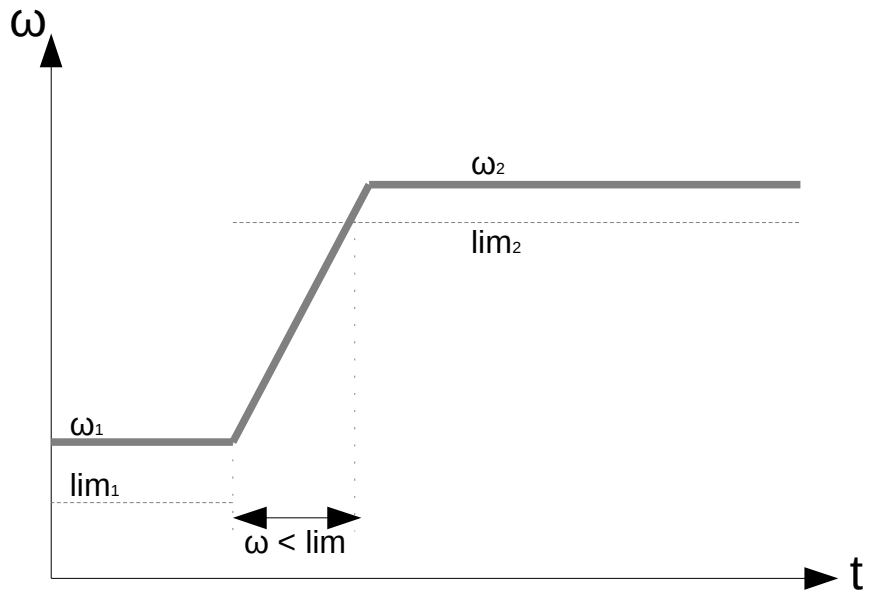


Figura 22: límite de detección de obstáculo durante la aceleración

Por eso, durante los tramos de aceleración, la detección debe realizarse de forma diferente, ya que de hacerse de la misma manera, se detectaría obstáculo nada más empezar a acelerar.

La solución para estos tramos podría ser una de las siguientes:

- Detectar variaciones bruscas en la aceleración, considerando la aceleración como variación de velocidad entre dos instantes muy próximos en el tiempo. Las pruebas con este método no dieron buenos resultados, ya que la aceleración depende directamente del control que en cada instante realicen los controladores PI, que no se puede prever de antemano de forma suficientemente precisa y sencilla.
- Fijar un valor de aceleración mínimo, considerando la aceleración como variación de velocidad entre dos instantes muy próximos en el tiempo, por debajo del cual se considera que se ha detectado un obstáculo. Este método tampoco dio buenos resultados, por el mismo motivo.
- Determinar que se ha encontrado un obstáculo si en cualquier momento del tramo de aceleración la velocidad baja respecto al instante anterior. Este es el método que dio mejores resultados, ya que fue el único que no detectaba obstáculos en momentos en que no los había. No obstante, debido a la gran fuerza ejercida por el motor para dar aceleración a la puerta, el golpe producido al obstáculo era notablemente mayor de lo permitido.

Así pues, se concluye que la detección de obstáculo durante los tramos de aceleración no es posible de realizar de manera óptima mediante el método de variación por velocidad. No obstante, durante los tramos de velocidad constante, su desempeño resulta excelente, por encima incluso de las expectativas iniciales.

6.8.2. Detección por variación de par

El par motor ejercido en cada momento resulta muy sencillo de cuantificar gracias al control vectorial, ya que tal y como se explica en el capítulo “Control vectorial“, dicho par es directamente proporcional a la componente de cuadratura del vector de corrientes, es decir, a I_q, valor calculado mediante la transformada de Park a partir de las corrientes medidas.

Para determinar la mejor manera de detectar obstáculos en función de la variación de la corriente de par, se realizaron una serie de pruebas en la puerta de prueba, midiendo los valores de dicha corriente durante un movimiento normal de cierre y durante un movimiento de cierre ante el que una persona se interpone en el trayecto, ejerciendo por tanto un par contrario al movimiento. Concretamente se midieron dichos valores durante la aceleración de la puerta, a fin de determinar si este método de detección sería capaz de solucionar las carencias del método de detección por variación de velocidad.

Los resultados obtenidos fueron los siguientes:

Prueba 1: puerta sin el peso que facilita el cierre

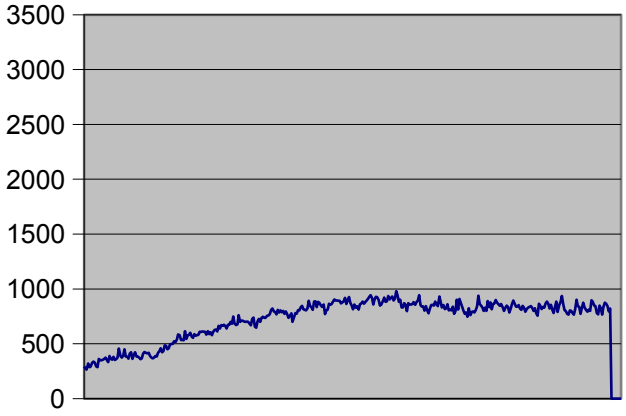


Figura 23: Iq sin obstáculo I

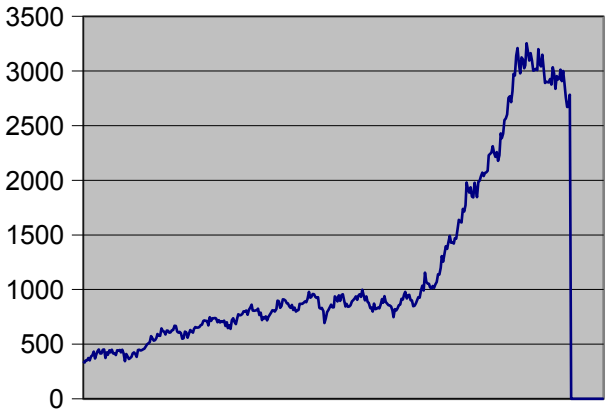


Figura 24: Iq con obstáculo I

Prueba 2: puerta con el peso que facilita el cierre

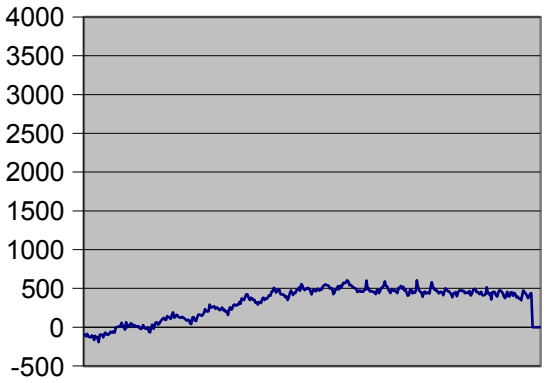


Figura 25: Iq sin obstáculo II

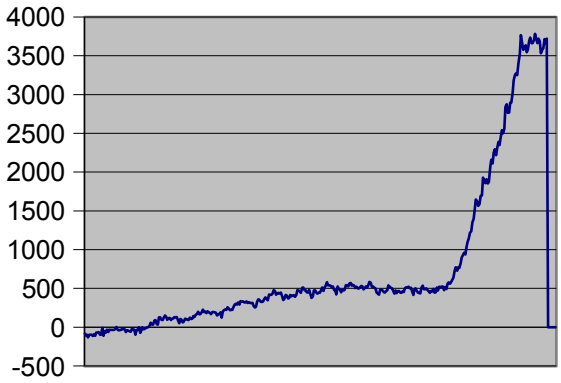


Figura 26: Iq con obstáculo II

Analizando los resultado, se puede observar que en el caso de toparse con un obstáculo, la corriente de par se incrementa a valores muy superiores a los que alcanza durante un cierre sin obstrucciones. Por ello, se decidió que la forma más fiable de detectar un obstáculo, a la par que la más sencilla, es la de fijar un valor máximo de la corriente de par, por encima de la cual se consideraría que se ha producido una obstrucción.

El valor límite de corriente de par no puede ser un valor fijo, ya que el valor que alcanza dicha corriente durante un movimiento libre depende del peso de la puerta, la velocidad máxima que se desee alcanzar, y una serie de circunstancias más. Por ese motivo, el citado valor límite de corriente de par debe poder fijarse vía menú de opciones, pudiendo adaptarse así a cualquier caso de uso que se le de.

6.8.3. Método de detección de obstáculos implementado

A la vista de los resultados de las pruebas realizadas con ambos métodos, se decidió implementar el método de detección por límite de corriente. El motivo de tomar esta decisión fueron:

- Eficacia muy superior durante los tramos de aceleración
- Eficacia muy similar durante los periodos de mantenimiento de velocidad
- Mayor simplicidad de implementación
- Mayor simplicidad de configuración para el instalador de la puerta

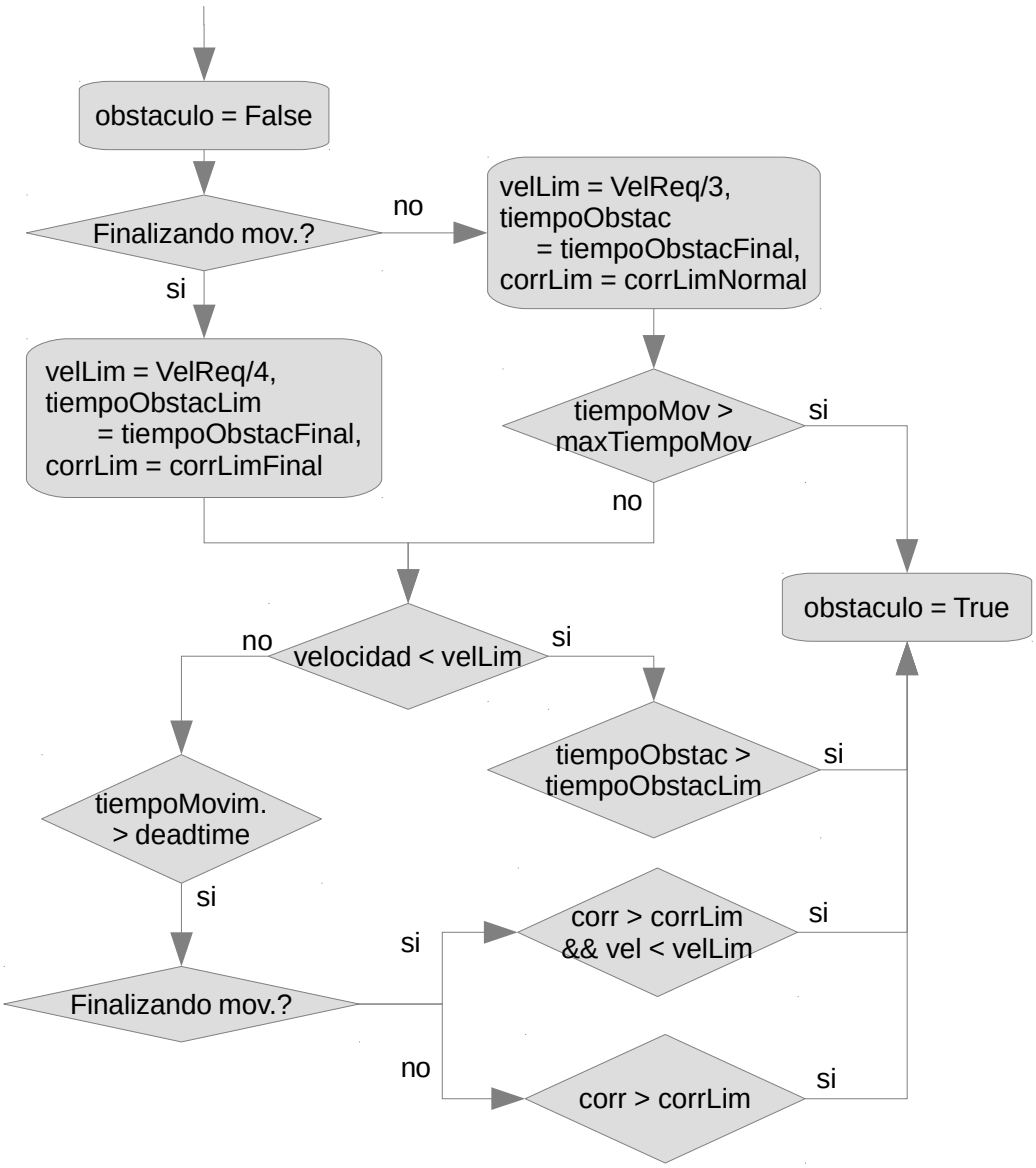


Figura 27: diagrama de flujo del módulo de detección de obstáculos

No obstante, se decidió implementar también un método de detección de obstáculo por velocidad, con motivo de asegurar que una mala configuración por parte del instalador de la puerta pueda derivar en hipotéticos daños materiales o personales. Esto podría pasar si, por ejemplo, el instalador configurara como corriente límite de detección una corriente muy alta.

Sin embargo, la detección por velocidad implementada es notablemente más sencilla que la descrita en el apartado “Detección por variación de velocidad”: se considera que hay una obstrucción si la velocidad medida permanece por debajo de una tercera parte de la velocidad de referencia durante más de 1,2s. Como ya se ha explicado, este no es el método de detección principal, sino uno adicional, de seguridad, de mucha menor sensibilidad.

El software implementado está representado, de manera simplificada, en el diagrama de flujo de la figura 27.

Se realizaron ensayos para comprobar el desempeño y la eficacia de la detección de obstáculos implementada. Sus resultados pueden verse en el capítulo “Respuesta del sistema con curva de velocidades”.

El código desarrollado correspondiente al sistema de detección de obstáculos se implementó en los archivos “obstaculo.c” y “obstaculo.h”, que se adjunta en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.9. Menú de opciones

Para permitir al usuario que configure los distintos parámetros ajustables, es necesario que pueda interactuar con un menú de opciones.

Como este menú no es previsible que se utilice muchas veces (solamente una al instalar la puerta), parece razonable utilizar un hardware muy básico a fin de ahorrar costes. Por este motivo, se decidió utilizar 4 displays “7 segmentos” y una serie de pulsadores.

El menú se compondrá de dos niveles. El primer nivel determina la categoría a la que pertenece cada opción, y el segundo nivel cada una de las opciones. Como con los displays “7 segmentos” no se pueden mostrar textos largos o complejos, se mostrará las opciones según su numeración.

Así pues, en el primer nivel del menú, se mostrará el número de la categoría en el que se encuentra el usuario. Por ejemplo, para la categoría nº2 se mostrará en los displays el texto “-F2-”. En el segundo nivel del menú, se mostrará el número de la categoría y el número de la opción. Por ejemplo, para la opción 3 de la categoría 2, se mostrará el texto “F2.03”. Una vez se selecciona una opción, se muestra su valor numérico y se permite editar dicho valor.

Para entrar al menú, se debe pulsar la tecla “mode”. Una vez dentro, la tecla “set” sirve para avanzar un nivel o aceptar, la tecla “mode” para bajar un nivel y las flechas para avanzar o retroceder en el nivel actual del menú. Al editar el valor de un parámetro, mantener una flecha

pulsada hace que el valor se incremente o decremente de forma continua y cada vez más rápido.

La implementación del software para controlar la botonera y los displays se explica en los capítulos “Botonera” y “Displays”.

Imagen 5: botonera y displays

Las siguientes tablas muestran todas las opciones disponibles para configurar.

F0: parámetros de operación básicos		
Parámetro	**Descripción**	**Valores**
F0.02	**Selección del sistema de control:** *Permite seleccionar la forma en que se reciben las órdenes de control de la puerta* *0 – botonera* *1 – señales externas* *2 – demostración automática*	0-2
F0.03	**Selección de la dirección de la marcha:** *0 – directa* *1 - inversa*	0-1

Tabla 1: parámetros de operación básicos (F0)

F1: parámetros de la curva de apertura		
Parámetro	**Descripción**	**Valores**
F1.01	**Distancia del tramo inicial de la curva de apertura:**	0-30%
F1.02	**Velocidad del tramo inicial de la curva de apertura**	1-25 hz
F1.03	**Tiempo de aceleración de la curva de apertura:** *Tiempo que tarda en acelerar de la velocidad inicial a la velocidad de referencia*	0.1-999.9 s
F1.04	**Velocidad de referencia de la curva de apertura**	0.01-20 hz
F1.06	**Posición de inicio de la deceleración en la curva de apertura:** *Distancia a la que comienza la deceleración desde la velocidad de referencia a la velocidad del tramo final*	60-95%
F1.07	**Tiempo de deceleración de la curva de apertura:** *Tiempo que tarda en decelerar de la velocidad de referencia a la velocidad del tramo final*	0.1-999.9 s
F1.08	**Velocidad del tramo final de la curva de apertura**	0-15 hz
F1.09	**Distancia de límite de posición en la curva de apertura** *Distancia respecto al final del recorrido por debajo de la cual se inicia el proceso de completado de la apertura.*	1-9999 pulsos
F1.10	**Límite de par durante el proceso de completado de apertura:** *Par realizado, en porcentaje respecto al par nominal, por encima del cual se considera que se ha completado la apertura durante el proceso de completado de apertura.*	0-100%
F1.11	**Par realizado durante el modo de mantenimiento del par:** *Par realizado durante el modo de mantenimiento del par, para completado del movimiento de apertura, en % del par nominal*	0-100%

Tabla 2: parámetros de la curva de apertura (F1)

Se recomienda ver el capítulo “Curva de velocidades en movimiento de apertura“, así como la figura 16 de dicho capítulo.

F2: parámetros de la curva de cierre		
Parámetro	**Descripción**	**Valores**
F2.01	**Distancia del tramo inicial de la curva de cierre:**	0-30%
F2.02	**Velocidad del tramo inicial de la curva de cierre**	1-25 hz
F2.03	**Tiempo de aceleración de la curva de cierre:** *Tiempo que tarda en acelerar de la velocidad inicial a la velocidad de referencia*	0.1-999.9 s
F2.04	**Velocidad de referencia de la curva de cierre**	0.01-20 hz
F2.06	**Posición de inicio de la deceleración en la curva de cierre:** *Distancia a la que comienza la deceleración desde la velocidad de referencia a la velocidad del tramo final*	60-95%
F2.07	**Tiempo de deceleración de la curva de cierre:** *Tiempo que tarda en decelerar de la velocidad de referencia a la velocidad del tramo final*	0.1-999.9 s
F2.08	**Velocidad del tramo final de la curva de cierre**	0-15 hz
F2.09	**Distancia de límite de posición en la curva de cierre** *Distancia respecto al final del recorrido por debajo de la cual se inicia el proceso de completado del cierre.*	1-9999 pulsos
F2.10	**Límite de par durante el proceso de completado de cierre:** *Par realizado, en porcentaje respecto al par nominal, por encima del cual se considera que se ha completado el cierre durante el proceso de completado de cierre.*	0-100%
F2.11	**Par realizado durante el modo de mantenimiento del par:** *Par realizado durante el modo de mantenimiento del par, para completado del movimiento de cierre, en % del par nominal*	0-100%
F2.15	**Posición de inicio del modo de bloqueo de la puerta:** *Distancia respecto al final del recorrido por debajo de la cual comienza el tramo de bloqueo de la puerta*	0-9999 pulsos
F2.16	**Velocidad del tramo de bloqueo de la puerta:** *Velocidad del tramo de bloqueo de la puerta*	0-20 hz

Tabla 3: parámetros de la curva de cierre (F2)

Se recomienda ver el capítulo “Curva de velocidades en movimiento de cierre“, así como la figura 18 de dicho capítulo.

F3: parámetros de funciones adicionales		
Parámetro	**Descripción**	**Valores**
F3.00	**Velocidad de movimiento en modo baja velocidad**	0-20 hz
F3.01	**Tiempo límite de apertura**	0-3600 s
F3.02	**Tiempo límite de cierre**	0-3600 s
F3.03	**Tiempo límite de movimiento en modo baja velocidad**	0-3600 s
F3.07	**Acción ante obstrucción durante el cierre:** *0 – parar* *1 - abrir*	0-1
F3.08	**Sensibilidad ante obstrucción durante el cierre:** *Límite de par, en porcentaje respecto al par nominal, por encima del cual se considera obstáculo durante el cierre*	0-150%
F3.10	**Sensibilidad ante obstrucción durante la apertura:** *Límite de par, en porcentaje respecto al par nominal, por encima del cual se considera obstáculo durante la apertura*	0-150%
F3.11	**Tiempo de espera tras apertura en modo demo:** *En modo de demostración automática, tiempo de espera antes de iniciar el cierre, tras haber completado una apertura. Durante este tiempo se debe ejecutar el modo "mantenimiento del par"*	1-3600 s
F3.12	**Tiempo de espera tras cierre en modo demo:** *En modo de demostración automática, tiempo de espera antes de iniciar la apertura, tras haber completado un cierre. Durante este tiempo se debe ejecutar el modo "mantenimiento del par"*	1-3600 s
F3.13	**Número máximo de movimientos en modo demo**	0-9999

Tabla 4: parámetros de funciones adicionales (F3)

F4: parámetros del proceso de aprendizaje		
Parámetro	**Descripción**	**Valores**
F4.04	**Velocidad de movimiento en el proceso de aprendizaje**	0-20 hz
F4.05	**Activación del proceso de aprendizaje**	0-1

Tabla 5: parámetros del proceso de aprendizaje (F4)

F7: parámetros de control vectorial		
Parámetro	**Descripción**	**Valores**
F7.00	**Constante proporcional del PI de velocidad (velocidades altas)**	1-9999
F7.01	**Constante integral del PI de velocidad (velocidades altas)**	1-9999
F7.02	**Constante proporcional del PI de velocidad (velocidades bajas)**	1-9999
F7.03	**Constante integral del PI de velocidad (velocidades bajas)**	1-9999
F7.04	**Velocidad de cambio entre altas y bajas velocidades:** *Velocidad por encima de la cual se utilizan los parámetros del PI de velocidad para velocidades altas, y por debajo de la cual se utilizan los parámetros del PI de velocidad para velocidades bajas*	0-50 hz
F7.06	**Máxima referencia de par en modo motor** *Máximo par de referencia cuando el motor consume potencia (modo motor), en porcentaje respecto al par nominal*	0-200%
F7.07	**Máxima referencia de par en modo generador** *Máximo par de referencia cuando el motor genera potencia (modo generador), en porcentaje respecto al par nominal*	0-200%
F7.08	**Constante proporcional de los PI de corriente**	1-9999
F7.09	**Constante integral de los PI de corriente**	1-9999

Tabla 6: parámetros de control vectorial (F7)

Las variables y funciones desarrolladas para implementar el manejo del menú se implementaron en los archivos “user_options_input.h” y “user_options_input.c”, que se adjuntan en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.10. Señales de entrada

La tarjeta cuenta con una serie de entradas para recibir señales externas. Las señales más importantes son las de apertura y cierre, que deben llegar desde el dispositivo de control de maniobra del ascensor, pero también están disponibles 5 entradas multifunción, cuya función se debe poder configurar desde el menú de opciones (pendiente de desarrollo).

Estas señales llegan directamente a diversos puertos del microcontrolador, por lo que basta con leer el valor digital de la tensión en dichos puertos, que deben estar configurados como entradas.

El código desarrollado correspondiente a la lectura de las entradas se implementó en los archivos “entradas.h” y “entradas.c”, que se adjuntan en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.11. Botonera

El número total de botones es de 6. Para utilizar un menor número de pines del microcontrolador, se decidió utilizar el circuito integrado 74HC165. Este componente almacena en un registro los datos en paralelo de 8 de sus pines, y a continuación los transmite en serie en sincronía con una señal de reloj. Son necesarios 3 pines del microcontrolador para leer los datos en paralelo. Una señal sirve para almacenar en el registro los datos leídos en paralelo, lo que hay que evitar mientras se transmiten los datos en serie. Otra señal es la de reloj, para controlar los tiempos de la transmisión en serie, y hace falta un tercer pin para leer el dato enviado en serie desde el 74HC165.

Según la hoja de características del 74HC165, la máxima frecuencia de la señal de reloj debe ser, para la temperatura y tensión de trabajos, de 4MHz (T=250ns). Además, deben tenerse en cuenta los tiempos de propagación (propagation delay, ts), que debe respetarse antes de leer un dato desde que este se transmite. Este tiempo de propagación es, para la temperatura y tensión de trabajos, de 60ns.

Como la cantidad de datos a transmitir es de tan solo 6 bits, se decidió que la rapidez en su transmisión no debía considerarse crítica. Por eso, se decidió dar un amplio margen, estableciendo un periodo de 400ns en la señal de reloj. Además, como aún así el tiempo de transmisión sigue sin ser largo, se decidió implementarlo en la rutina principal del programa, en lugar de utilizar interrupciones. Esto también permite priorizar la ejecución del código correspondiente al control, más crítico, que se ejecuta en interrupciones.

La función que se encarga de comprobar qué botón se ha pulsado se encarga de comprobar que se mantiene durante un breve periodo de tiempo, para evitar la detección de falsas pulsaciones.

En base a lo anterior, se decidió realizar la lectura de los datos serie tal y como se describe en las figuras 28 y 29.

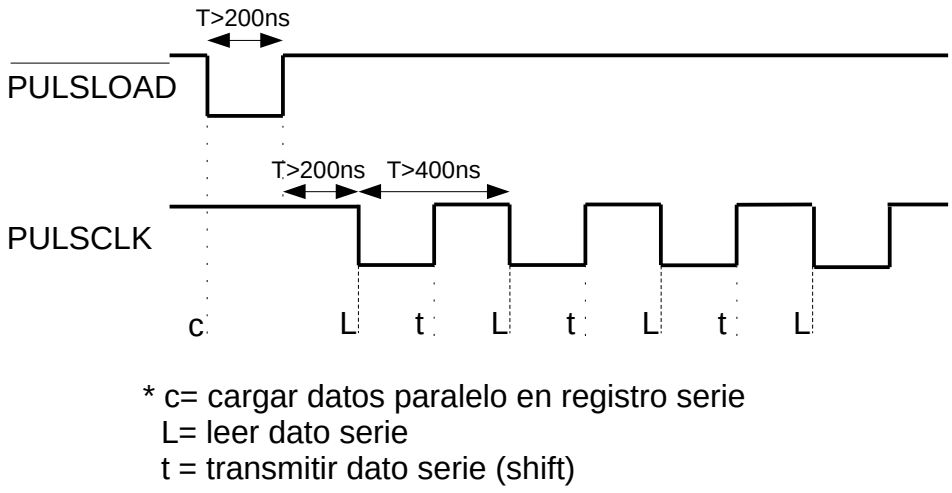


Figura 28: 74HC165: señales reloj y lectura paralelo (ej. de lectura de 4 bits)

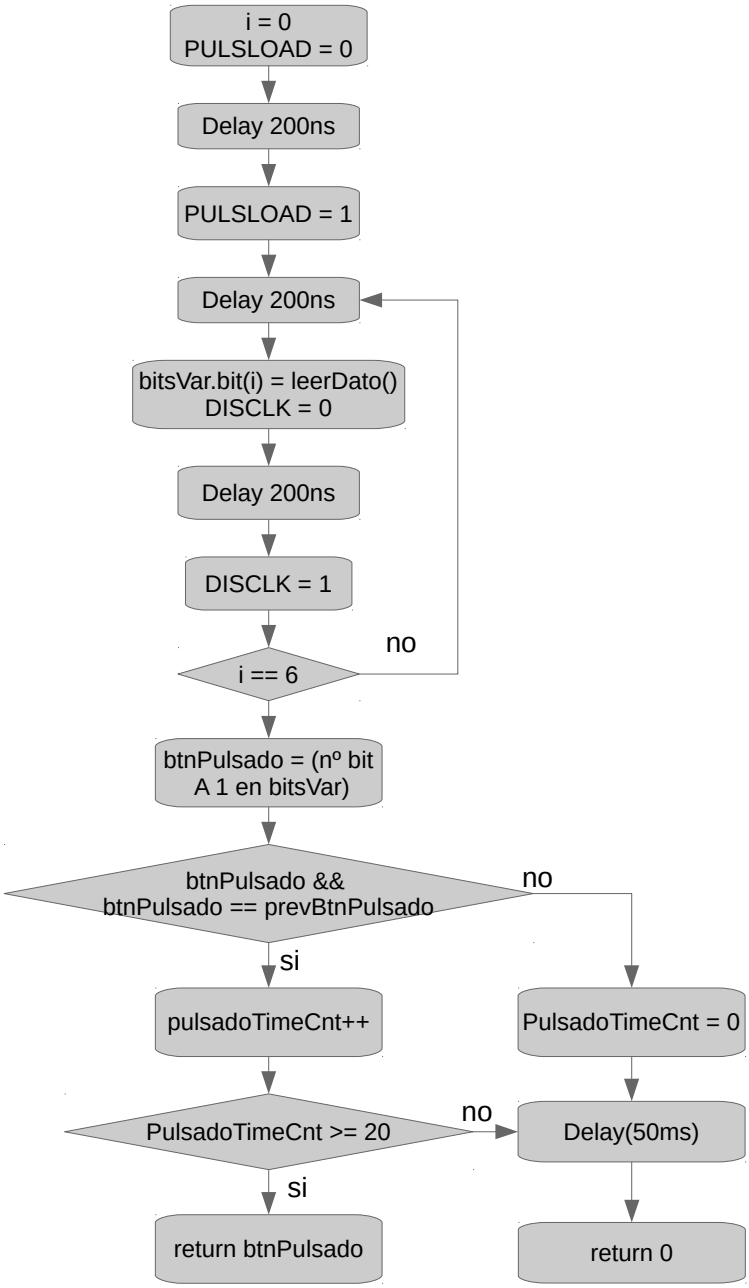


Figura 29: diagrama de flujo del envío de datos en serie a los displays

El código desarrollado correspondiente a la lectura de la pulsación de botones se implementó en los archivos “botonera.h” y “botonera.c”, que se adjuntan en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.12. Displays

Debido al gran número de pines necesarios para controlar los displays (4 displays de 8 pines cada uno), se decidió utilizar el circuito integrado 74HC595, 4 de ellos en total, uno por cada display.

El 74HC595 recibe 8 bits de datos en serie, sincronizados con una señal de reloj, los almacena en un registro, y pone esos datos en las salidas, en paralelo. Además, también cuenta con una salida serie, cuya función es la de poder encadenar varios 74HC595, lo que permite transmitir datos en serie a un número indefinido de 74HC595, sin necesidad de utilizar más de los dos pines ya citados del microcontrolador: uno para la señal de reloj, y otro para el dato a transmitir.

Una vez se ha terminado de transmitir todos los datos en serie, se debe mandar una tercera señal, cuya función es la de fijar esos datos en el registro de salida en paralelo. Así no se aprecian cambios en los displays mientras se están transmitiendo los datos en serie.

Según la hoja de características del 74HC595, la frecuencia máxima de la señal de reloj para la tensión y temperaturas de trabajo es de 4MHz (T = 250ns). La transmisón del dato se realiza en el flanco positivo de la señal de reloj, y debe tenerse en cuenta el tiempo de establecimiento del dato (set-up time, t_{su}) y el tiempo de mantenimiento del dato (holding time, t_h), tal y como se muestra en la figura 30.

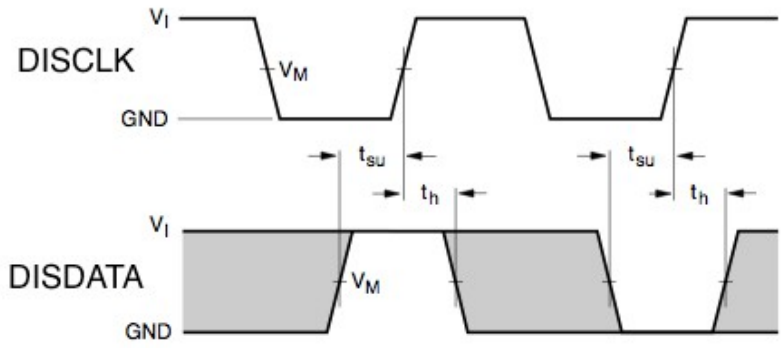
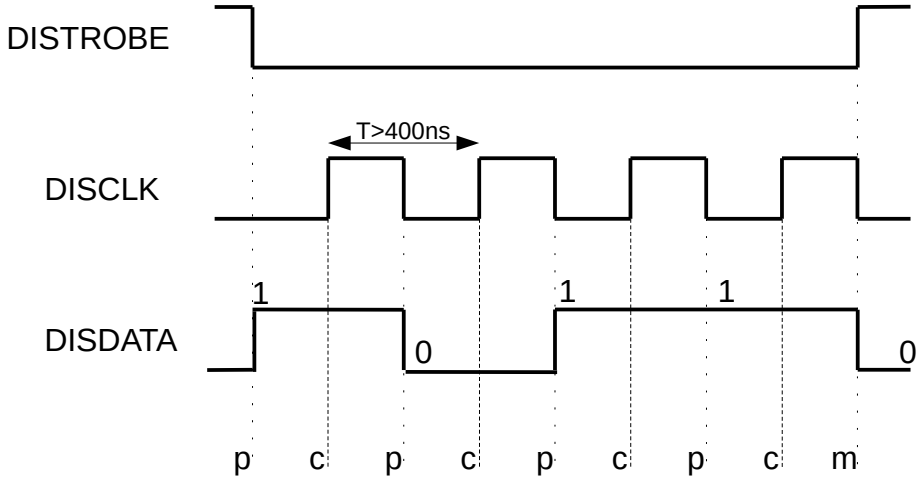


Figura 30: 74HC595: tiempos de establecimiento y mantenimiento del dato (fuente: hoja de características del 74HC595 de NXP)

Según la hoja de características t_{su}=110ns y t_h=3ns.

Como la cantidad de datos a transmitir es de tan solo 32 bits, se decidió que la rapidez en su transmisión no debía considerarse crítica. Por eso, se decidió dar un amplio margen, estableciendo un periodo de 400ns en la señal de reloj. Además, como aún así el tiempo de transmisión sigue sin ser largo, se decidió implementarlo en la rutina principal del programa, en lugar de utilizar interrupciones. Esto también permite priorizar la ejecución del código correspondiente al control, más crítico, que se ejecuta en interrupciones.

En base a lo anterior, se decidió realizar el envío de los datos serie tal y como se describe en las figuras 31 y 32.



* c = capturar dato en registro serie
p = preparar dato
m= mostrar datos en paralelo

Figura 31: 74HC595: señales de reloj y de dato (ejemplo de envío de 4 bits)

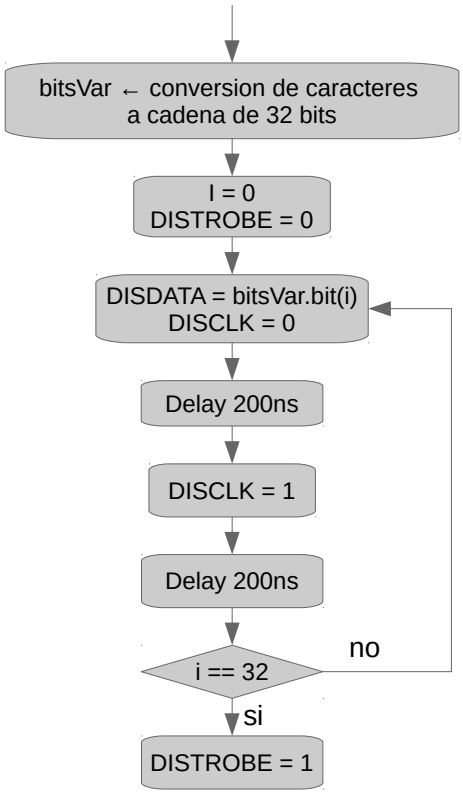


Figura 32: diagrama de flujo del envío de datos en serie a los displays

El código desarrollado correspondiente a la actualización de los displays se implementó en los archivos “displays.h” y “displays.c”, que se adjuntan en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

6.13. Recepción de órdenes

Las órdenes de apertura, cierre y parada deben llegar de manera externa, para ser ejecutadas una vez recibidas.

Estas órdenes pueden provenir de dos maneras diferentes:

- Desde la botonera
- Desde señales eléctricas externas, desde el sistema de control de maniobra del ascensor

La selección de una de las dos órdenes se realiza mediante la configuración de uno de los parámetros del menú de opciones.

La comprobación de si las señales están activas se realiza en el programa principal, en la función “main” del archivo “PMSM.c”.

6.14. Modificaciones en el software de control

Como se explica en el capítulo “Descripción del programa original”, en el programa original la función “main” detecta la pulsación del botón, activando y desactivando las conmutaciones PWM. Una vez activadas, todo el control se ejecuta desde la interrupción del módulo PWM, llamando en orden a las funciones oportunas.

Para adaptar el software original a las nuevas necesidades, fueron necesarias una serie de modificaciones, principalmente en la función “main” y en la función “DoControl”, que es llamada desde la interrupción del módulo PWM.

En las figuras se muestra la estructura de las funciones “main” y “DoControl” tras dichas modificaciones.

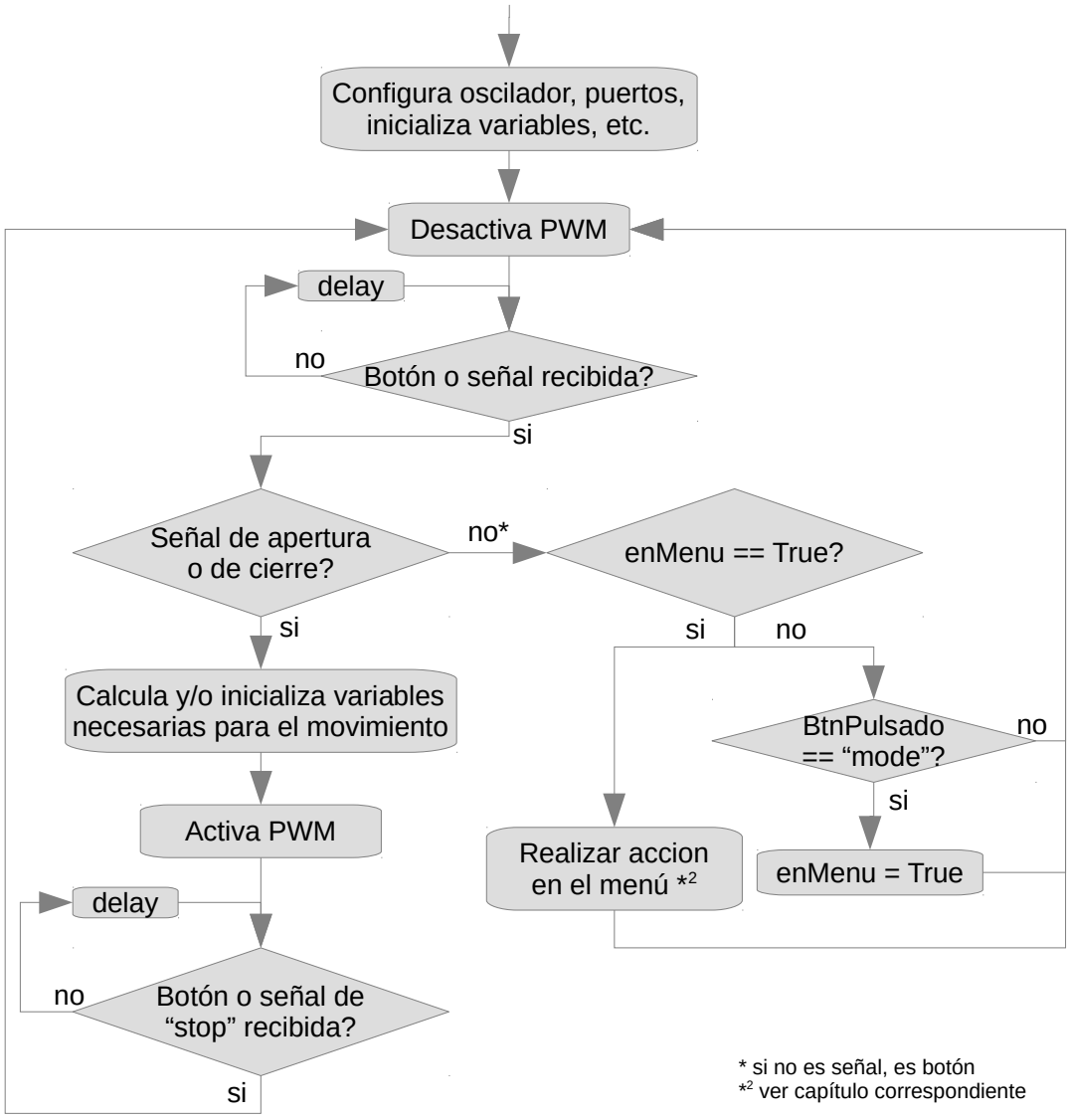


Figura 33: diagrama de flujo de la función "main" tras las modificaciones

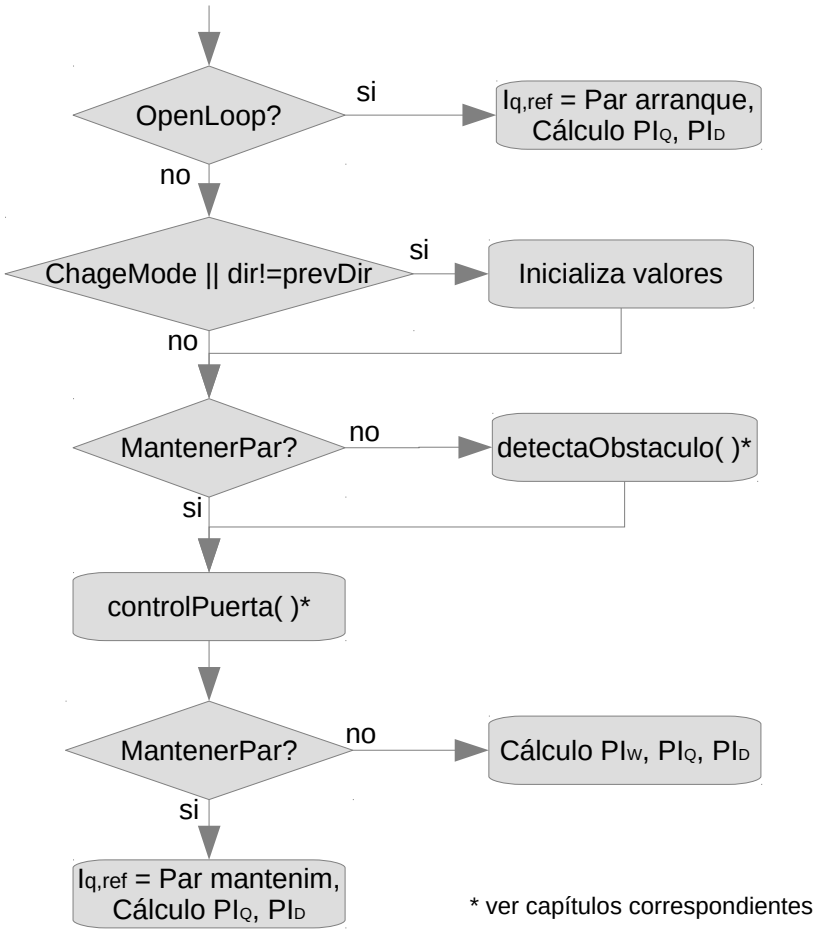


Figura 34: diagrama de flujo de la función "DoControl" tras las modificaciones

6.15. Guardado en memoria flash

Algunos de los valores calculados y parámetros configurados se perderían en caso de reinicio o corte de luz. Por ese motivo, resulta conveniente guardar dichos valores en la memoria flash del microcontrolador, de forma que se puedan recuperar en el siguiente encendido.

Estos valores que resulta necesario guardar son los siguientes:

- Desfase calculado entre ángulo mecánico y ángulo eléctrico (ver capítulo “Control de la posición angular del rotor mediante el encoder”)
- Longitud de la puerta obtenida durante el *aprendizaje* (ver capítulo “Aprendizaje de la longitud de la puerta”)
- Valores configurados por el usuario a través del menú de opciones (ver capítulo “Menú de opciones”)

Entre las características propias de las memorias flash, se encuentra la de que no es posible realizar el borrado o escritura de una sola posición de memoria. Necesario hacerlo en bloques de determinados tamaños. En el caso del microcontrolador utilizado, el borrado debe realizarse en bloques de 1024 bytes, y la escritura en bloques de 2 ó 4 bytes.

Otras de las características propias de las memorias flash es la de que resulta necesario borrar una posición de memoria antes de escribir en ella.

Por lo tanto, para escribir en una sola posición de memoria, se debe borrar primero todo el bloque al que pertenece (en este caso de 1024bytes). Por eso, hace falta guardar en memoria RAM todos los datos de ese bloque, borrarlo y volver a escribir esos datos. De lo contrario se perderían.

A fin de agilizar todos los procesos de lectura y escritura, se decidió implementar una serie de funciones encargadas de gestionar todo lo relativo al manejo de los datos guardados en la memoria flash. Estas funciones agilizan los procesos de lectura y escritura de la siguiente manera:

- Tras el encendido, se realiza una lectura de todo el bloque de 1024 palabras y se almacena en memoria RAM, que actúa como “copia” de los datos almacenados en dicho bloque.
- Escritura:
    - Primero se modifica los datos en la “copia” en memoria RAM. Puede ser un solo dato o varios.
    - Luego se borra el bloque de 1024 bytes de la memoria flash. Como sus datos estaban ya previamente copiados en memoria RAM, no es necesario leerlos previamente para salvarlos, evitando así el paso de la lectura de todo el bloque en cada operación de escritura.
    - Finalmente se escriben en memoria flash los datos, mediante iteraciones de 4 bytes cada una.
- Lectura: se puede realizar directamente de los datos de la memoria RAM, ya que son en todo momento los mismos que hay en la memoria flash.

Como puede deducirse, para que este sistema funcione correctamente, únicamente deben manipularse los datos utilizando las funciones que utilizan dicho método. De lo contrario, la “copia” en memoria RAM de los datos podría presentar diferencias respecto a los datos en memoria flash, quedando “desincronizada”.

No obstante, esto no supone ningún problema, ya que este juego de funciones, además de optimizar recursos del microcontrolador, hace que resulte más fácil el manejo de los datos almacenados, ya que permite abstraerse de la forma de leer y escribir en memoria flash, que ya está implementada en las propias funciones y que puede resultar un tanto tediosa.

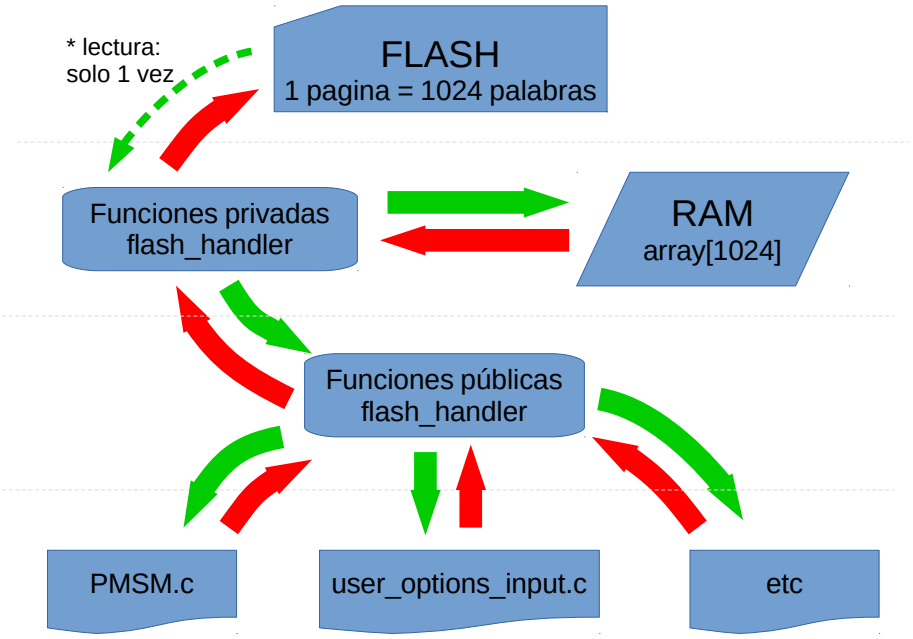


Figura 35: funcionamiento del módulo flash_handler

Todas las funciones y variables utilizadas para gestionar los datos almacenados en la memoria flash se implementaron en los archivos “flash_handler.h” y “flash_handler.c”, cuyo código se adjunta en el anexo 1 de este documento, así como en el código facilitado en soporte digital.

7. Hardware desarrollado

En este capítulo se pretende exponer una explicación acerca del diseño del hardware realizado. No obstante, el trabajo realizado por quien redacta este informe no ha estado enfocado en este campo, motivo por el cual se dará una explicación general y sin demasiado detalle.

Las únicas tareas relacionadas con el diseño del hardware en las que el autor del presente informe ha participado de forma activa fueron las de diseñar y fabricar varios prototipos de PCB, con la intención de probar el funcionamiento de la parte de la alimentación. Dicha alimentación de la tarjeta comprende dos partes: un rectificador de tensión de red de 230V de corriente alterna a 325V de corriente continua y un circuito regulador para proporcionar las tensiones de 24V, 15V y 3,3V para la electrónica.

Tanto los esquemáticos como el diseño del PCB completo se adjunta al presente documento en el anexo 2. Los esquemáticos y diseños de PCB de los prototipos de la parte de la alimentación se adjuntan al presente documento en el anexo 3.

7.1. Alimentación

La tarjeta debe alimentarse directamente de la red, que no obstante, debe ser convertida ya que en ninguna parte de la misma se necesita corriente alterna. La tarjeta necesita las siguientes tensiones:

- **325V DC**: para el bus de continua. Para obtener las tensiones de fase en el motor, se utilizan las secuencias de conmutación oportunas mediante el módulo de potencia, troceando esta tensión de bus.
- **24V DC**: la tarjeta debe ofrecer una salida para alimentar electrónica de 24V, hasta un máximo de 200mA (4,8VA). Esta alimentación se utiliza por ejemplo para el encoder.
- **15V DC**: esta es la tensión de alimentación del módulo de potencia. El consumo estimado es de 100mA (1,5VA)
- **3,3V DC**: esta es la tensión de alimentación del resto de la electrónica. El consumo estimado es de 250mA (0,825VA)

7.1.1. Rectificador

Para obtener los 325V DC, basta con rectificar la tensión de red. Para ello se utiliza el clásico puente de diodos seguido de un condensador de suficiente capacidad para estabilizar y eliminar el rizado. A esto se le añade un filtro de entrada para no transmitir ruidos a la red, así como una serie de condensadores que reduzcan el ruido de las conmutaciones de los diodos.

El diseño de esta parte de la tarjeta PCB, así como el esquemático correspondiente, puede verse en la documentación incluída en el anexo 2.

Quien redacta el presente informe elaboró el diseño de una tarjeta PCB prototipo para probar esta parte de la tarjeta, cuyo funcionamiento se pudo evaluar sustituyendo el rectificador de la tarjeta de evaluación MCHV-2 de Microchip, con la que se realizaban las pruebas del software, obteniendo resultados muy satisfactorios.

El diseño de dicho prototipo puede verse en el anexo 3. Pueden verse fotografías del prototipo fabricado en las imágenes 6 y 7.

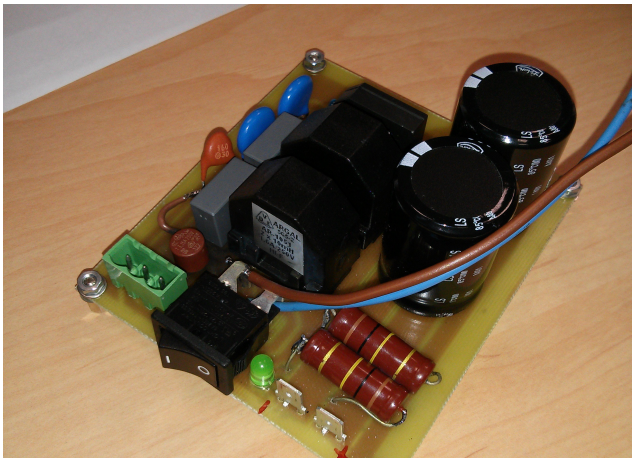


Imagen 6: vista superior del prototipo de rectificador

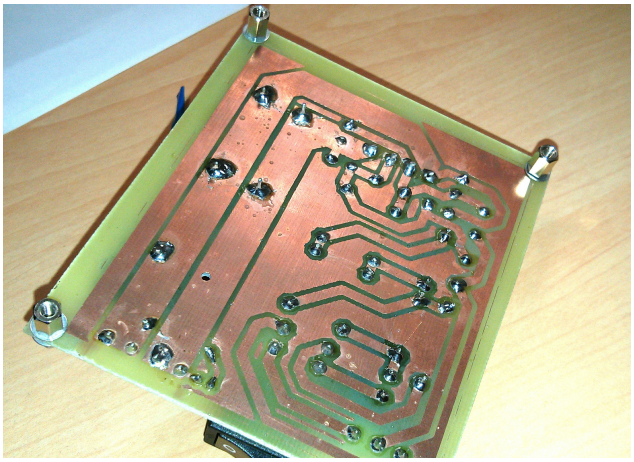


Imagen 7: vista inferior del prototipo de rectificador

7.1.2. Conversor reductor de tensión

Para obtener las distintas tensiones de continua de alimentación de la electrónica se elaboraron dos opciones de diseño diferentes:

- Con transformadores convencionales
- Con transformador de alta frecuencia con configuración flyback

La segunda opción se empezó a barajar al comprobar que la potencia demandada por todos los dispositivos electrónicos, en el caso de optar por la primera opción, derivaba en la necesidad de utilizar dos transformadores diferentes, uno de 2,5VA de potencia y el otro de 5VA, lo cual resulta en una ocupación excesiva del espacio disponible.

Se fabricó un prototipo de la parte del conversor con el transformador de 2,5VA, que alimenta los circuitos de 15 y 3,3V. Esas dos tensiones se obtienen rectificando la salida del secundario del transformador y utilizando los circuitos integrados 78M15 y LM2736, que son reguladores de

tensión. Los diseños del PCB y el esquemático de dicho prototipo pueden verse en el anexo 3 del presente documento.

No obstante, finalmente se optó por utilizar un conversor reductor con configuración flyback, con el que gracias a una conmutación a altas frecuencias (132kHz aprox.) se puede utilizar un solo transformador, de menor tamaño que en el caso anterior, capaz de transferir una potencia mucho mayor. De hecho, dicho transformador resultó suficiente para alimentar holgadamente los tres circuitos de alimentación de la electrónica, más un circuito auxiliar para controlar la conmutación del propio conversor.

Este conversor reductor funciona de la siguiente manera:

- La tensión de bus se trocea como onda cuadrada de alta frecuencia (132kHz aprox.) con el circuito integrado TOP232PN
- La onda cuadrada entra al primario del transformador, y gracias a la relación de transformación del mismo, se reduce su tensión a 3 valores diferentes para cada uno de los 4 secundarios. Estas tensiones se filtran y rectifican.

  Las tensiones, una vez rectificadas, de esos secundarios, son las siguientes:

  - 24V: circuito de salida 24V 200mA
  - 15V: circuito de 15V
  - 5V: esta tensión se reduce a 3,3V mediante el regulador LM1086
  - 5V: circuito de realimentación del TOP232PN
- Mediante un divisor de tensión se obtiene una tensión de 2,5V a partir de la linea de 15V. En caso de que la tensión baje de esos 2,5V, se envía la señal de control al TOP232 a través de un optoacoplador. Este optoacoplador activa la señal a través del circuito de realimentación del TOP232 de 5V. Este circuito es una realimentación permite mantener con precisión la tensión de salida.

El diseño de este circuito, así como el del transformador, puede verse en el anexo 2 del presente documento.

El diseño del PCB prototipo de este circuito puede verse en el anexo 3.

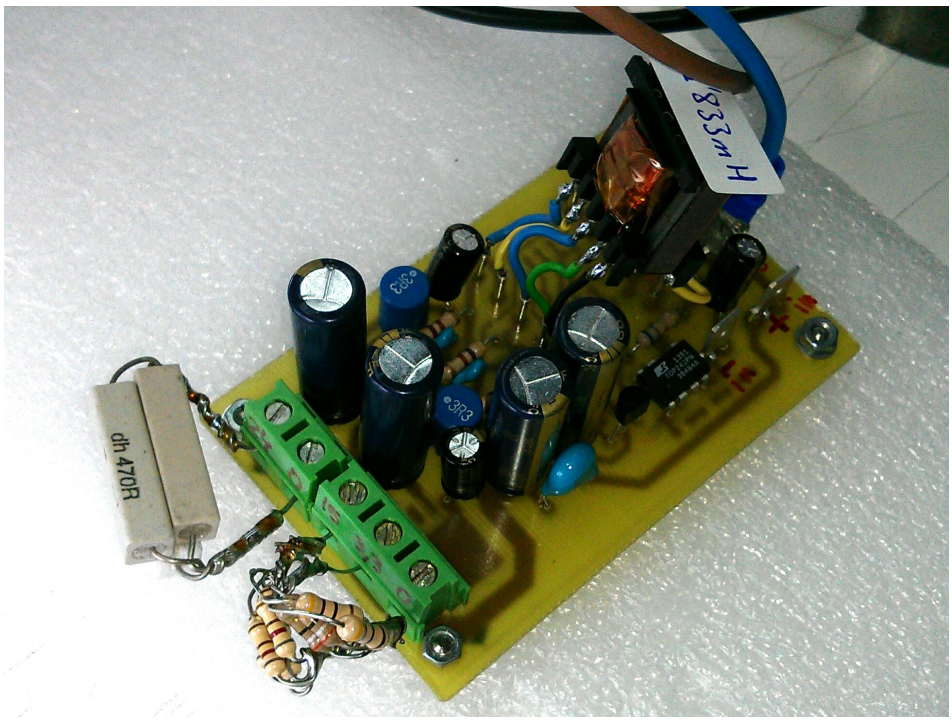


Imagen 8: prototipo de la fuente conmutada

7.2. Módulo de potencia

Las conmutaciones de la tensión de bus en cada una de las tres fases se realizan en función de las señales enviadas desde el microcontrolador, que indican los tiempos de encendido y apagado. Estas señales llegan al módulo de potencia FSBB15CH60C a través de un buffer.

El módulo de potencia incluye tanto los drivers como las células de conmutación, y requiere de una alimentación de 15V DC.

El diseño de esta parte del circuito puede verse en el anexo 2 del presente documento.

7.3. Recepción de las señales de entrada

La tarjeta debe ser capaz de recibir una serie de señales externas, para lo cual se han utilizado una serie de optoacopladores a fin de conseguir aislamiento galvánico entre los circuitos de entrada y el resto de la electrónica.

Las señales llegan desde los optoacopladores a los puertos del microcontrolador.

El diseño de esta parte del circuito puede verse en el anexo 2 del presente documento.

7.4. Botonera

Las señales de los pulsadores no se reciben en paralelo cada una en un puerto del microcontrolador, sino a través de un circuito integrado 74HC165, que es un shift register que recibe datos en paralelo y los transmite en serie de forma síncrona con una señal de reloj. De esta forma se necesita utilizar un menor número de pines del microcontrolador (véase el subcapítulo “Botonera” del capítulo “Software desarrollado”).

El diseño de esta parte del circuito puede verse en el anexo 2 del presente documento.

7.5. Displays

Las señales para controlar los displays 7 segmentos no se envían en paralelo desde el microcontrolador, sino a través de una serie de circuitos integrados 74HC595, que es un shift register que recibe datos en serie de forma síncrona con una señal de reloj, y los transmite a una salida de pines en paralelo.

El 74HC595 permite encadenar varios de estos componentes, de forma que se pueden utilizar un número indefinido de ellos sin necesidad de utilizar un mayor número de pines del microcontrolador. En el caso de la tarjeta descrita en el presente documento, se utilizan 4 74HC595, uno por cada display.

De esta forma se consigue lo siguiente:

- Uso de un número mucho menor de pines del microcontrolador (3 en lugar de 32)
- Se evita tener que suministrar la corriente desde los pines del microcontrolador
- Se puede alimentar a los 4 displays en todo momento, en lugar de tener que alternar entre los 4 displays como se hace en algunas ocasiones

Véase el subcapítulo “Displays” del capítulo “Software desarrollado”

El diseño de esta parte del circuito puede verse en el anexo 2 del presente documento.

7.6. Salidas

La tarjeta cuenta con 3 salidas para poder enviar señales al dispositivo de control de maniobra del ascensor. No obstante, la corriente necesaria para dichas señales no la suministra la propia tarjeta, sino que tan solo abre o cierra unos relés, y la corriente la suministra dicho dispositivo de control de maniobra. Los relés se abren o cierran a través de unos transistores, que reciben las señales de apertura y cierre en sus bases desde el microcontrolador.

El diseño de esta parte del circuito puede verse en el anexo 2 del presente documento.

8. Pruebas y ensayos realizados

Durante todo el proceso de desarrollo del software se fueron haciendo pruebas y comprobaciones de forma visual, evaluando el correcto funcionamiento del sistema a simple vista.

Una vez alcanzado cierto grado de madurez en dicho desarrollo del software, sabiendo por tanto que a rasgos generales ya realiza la tarea que debe desempeñar, se vuelve necesario hacer una comprobación empírica más fidedigna, para evaluar más detalladamente su comportamiento. Principalmente interesa comprobar hasta que punto la respuesta del sistema se ajusta a lo esperado (velocidad, par, etc), y ajustar los parámetros del control en consecuencia.

Con este fin se escribieron unas cuantas lineas de código que permiten monitorizar diferentes variables. Las características de este código son las siguientes:

- Entre 1 y 7 buffers de datos (arrays)
- Capacidad de cada buffer de más de 1024 muestras, ajustable
- Posibilidad de ajustar el intervalo de muestreo
- Posibilidad de elegir qué variables se desean monitorizar

Una vez almacenados los valores de las muestras en los buffers, estos pueden exportarse en formato CSV, compatible con hojas de cálculo, gracias al modo debug de la plataforma de desarrollo de Microchip. Asimismo, haciendo uso del modo debug permite modificar el intervalo de muestreo, así como qué variables se desean monitorizar.

El código desarrollado correspondiente a la monitorización de variables se implementó en el archivo principal del programa, “PMSM.c”, que se adjunta en el anexo 1 del presente documento, así como en el código facilitado en soporte digital.

En cuanto a los ensayos realizados haciendo uso del software mencionado, se siguió el siguiente proceso:

1. Ajuste de los parámetros de los controladores PI con carga en el eje del motor
    1. Ajuste de los PI de corriente ante entrada escalón, hasta conseguir una respuesta rápida, precisa y con baja oscilación
    2. Ajuste del PI de velocidad, más externo, ante entrada escalón, hasta conseguir una respuesta rápida, precisa y con baja oscilación.
2. Comprobación de la respuesta con los parámetros calculados sin carga en el motor
3. Comprobación de la respuesta del sistema, con los parámetros calculados, aplicando la curva de velocidades descrita en el capítulo “Error: Reference source not found“.

NOTA IMPORTANTE: debe saberse que los valores de los parámetros de los controladores PI utilizados en este software no coinciden con los que se podrían calcular aplicando la teoría de control. Esto se debe a que el algoritmo del controlador PI implementado está pensado para optimizar tiempo de cálculo, motivo por el cual podría decirse que los parámetros se miden en una escala diferente, para la que está optimizada también el resto de las partes del programa. Exceptuando lo anterior, el comportamiento del algoritmo es el que cabe esperar de un controlador PI.

8.1. Respuesta del sistema frente a entrada escalón, con carga en el eje

8.1.1. Ajuste del lazo interno: controladores PI de corriente

A continuación se irán probando distintos valores para los parámetros de los controladores PI de corriente.

Se decide empezar probando un valor alto de la constante proporcional y uno bajo de la integral, en busca de rapidez y estabilidad. El resultado puede verse en la figura 36.

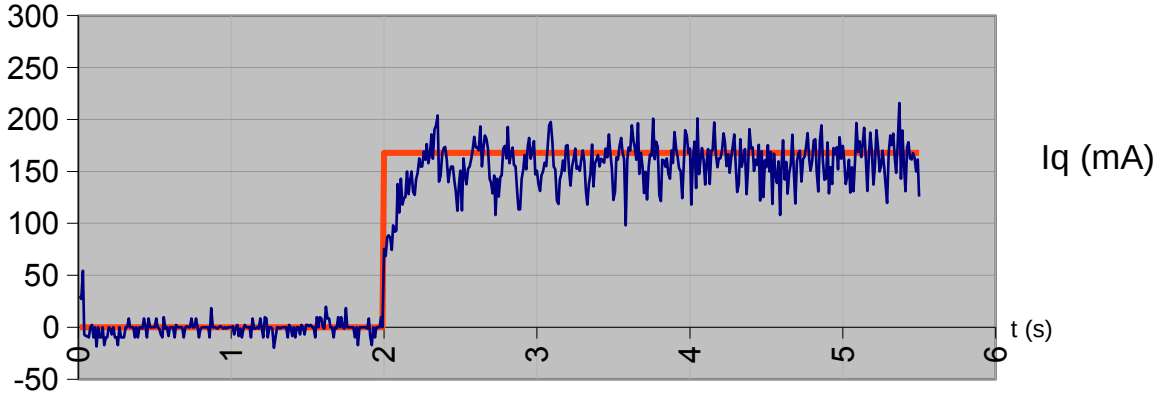


Figura 36: respuesta ante escalon de corriente. Kp=10000, Ki=75

Se comprueba como se obtiene una respuesta aceptable, pero se intenta mejorar la velocidad de respuesta, aumentando la constante proporcional, así como reducir la oscilación, aumentando la constante integral. Los resultados de las pruebas realizadas a tal efecto pueden verse en las figuras 37 a 40.

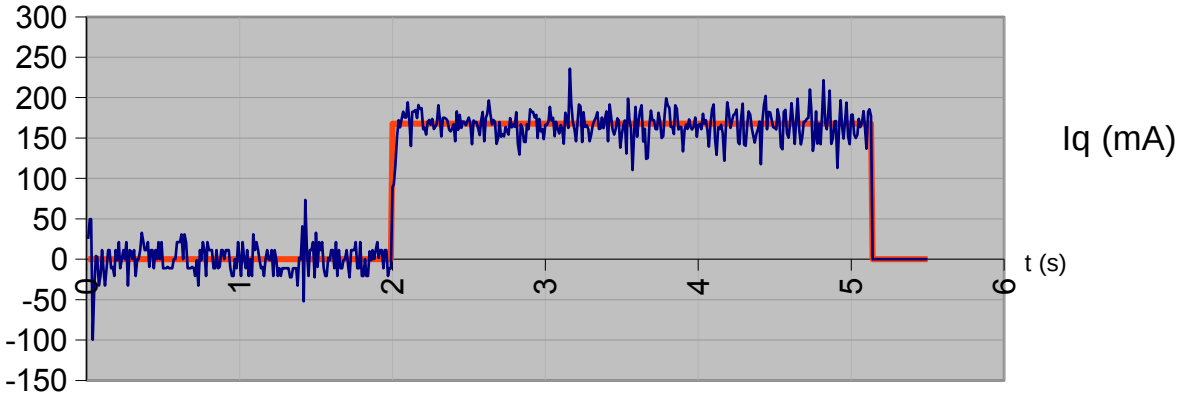


Figura 37: respuesta ante escalón de corriente. Kp=15000, Ki=750

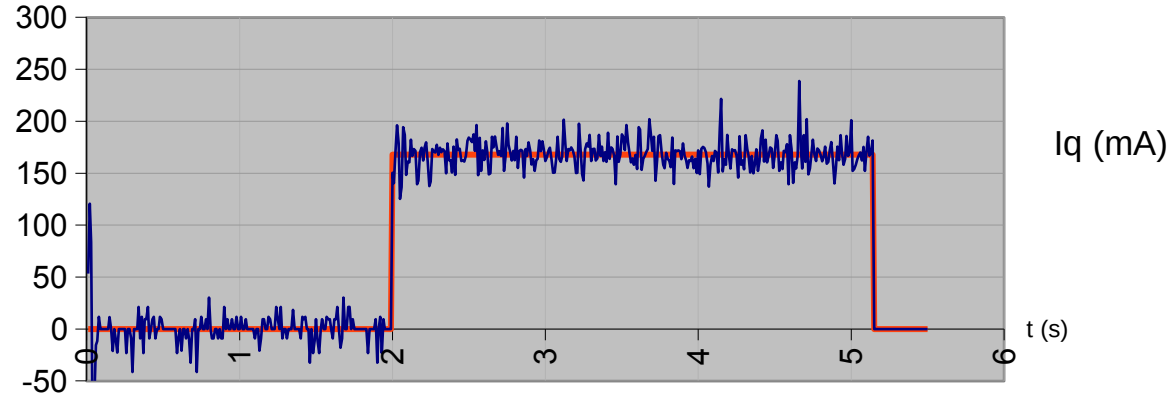


Figura 38: respuesta ante escalon de corriente. Kp=15000, Ki=2000

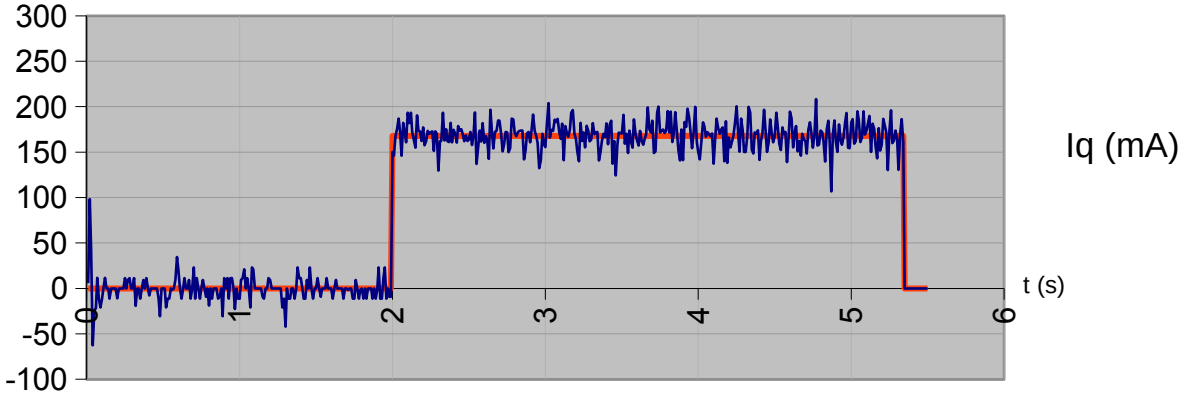


Figura 39: respuesta ante escalon de corriente. Kp=15000, Ki=3000

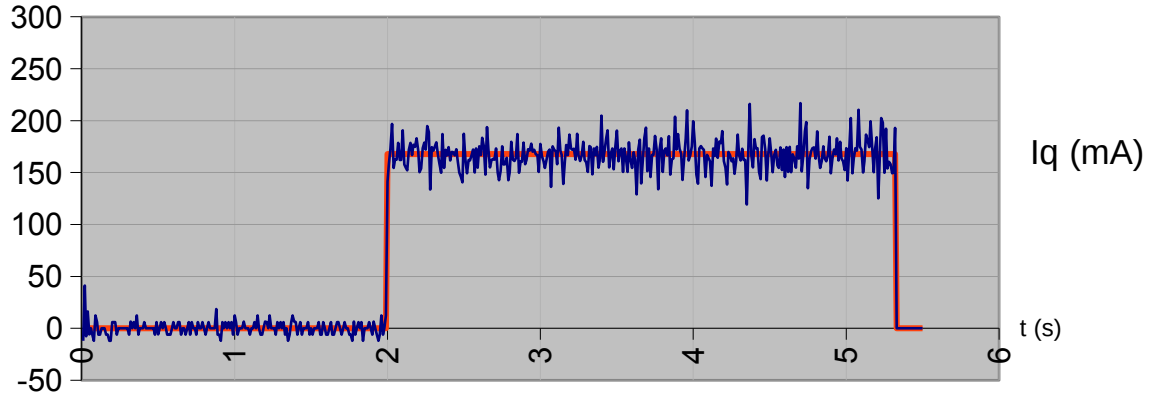


Figura 40: respuesta ante escalon de corriente. Kp=15000, Ki=4000

En vista de las gráficas obtenidas, se decidió utilizar los siguientes valores para los parámetros de los controladores PI de corriente:

- Kp: 15000
- Ki: 4000

8.1.2. Ajuste del lazo externo: controlador PI de velocidad

Una vez ajustados los PI internos, se procede a ajustar el PI externo de control de velocidad.

Al intentar realizar las primeras pruebas, se obtuvo un resultado realmente malo, con movimientos muy rápidos, oscilantes y muy bruscos, generando gran cantidad de ruido y un movimiento errático.

En vista de lo anterior, se realizaron nuevas pruebas ajustando unos valores más bajos. Los resultados de las pruebas pueden verse en las figuras 41 a 44.

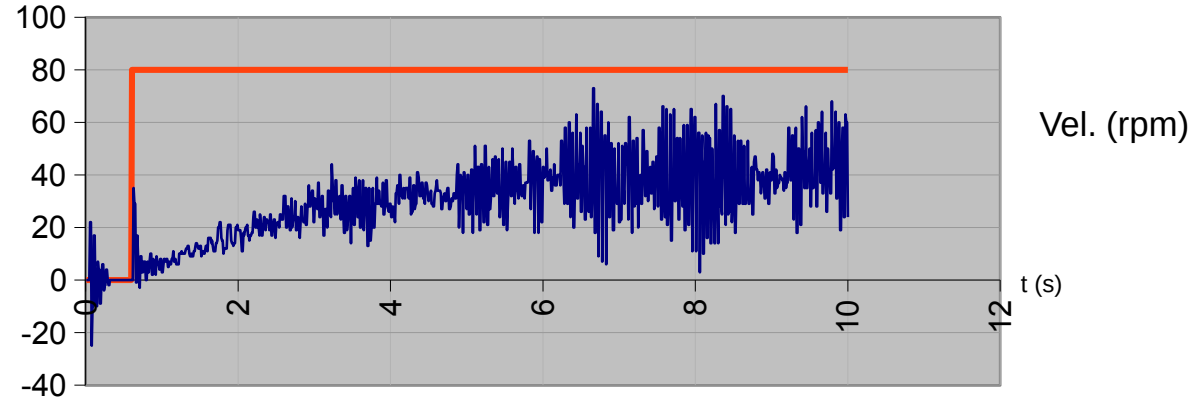


Figura 41: respuesta de la velocidad ante escalon de velocidad. $K_{p,\omega}$=1000, $K_{i,\omega}$=25

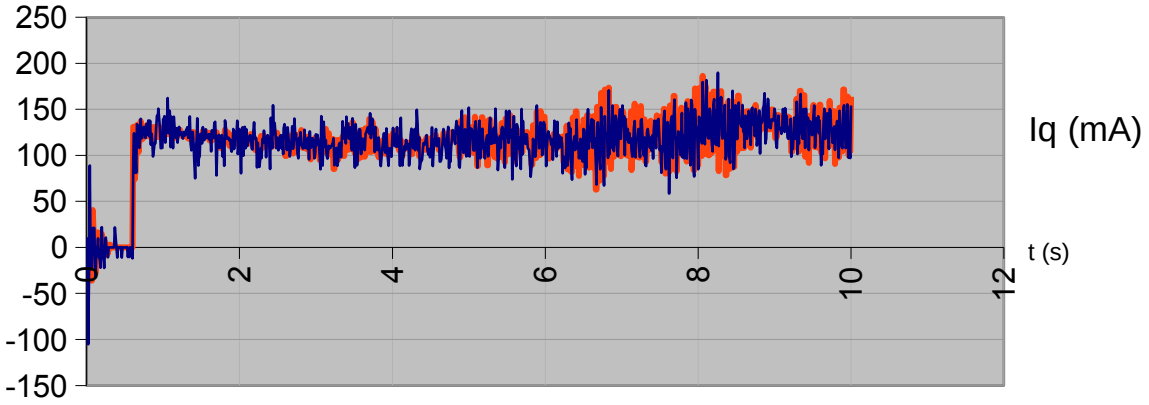


Figura 42: respuesta de la corriente de par ante escalon de velocidad. $K_{p,\omega}$=1000, $K_{i,\omega}$=25

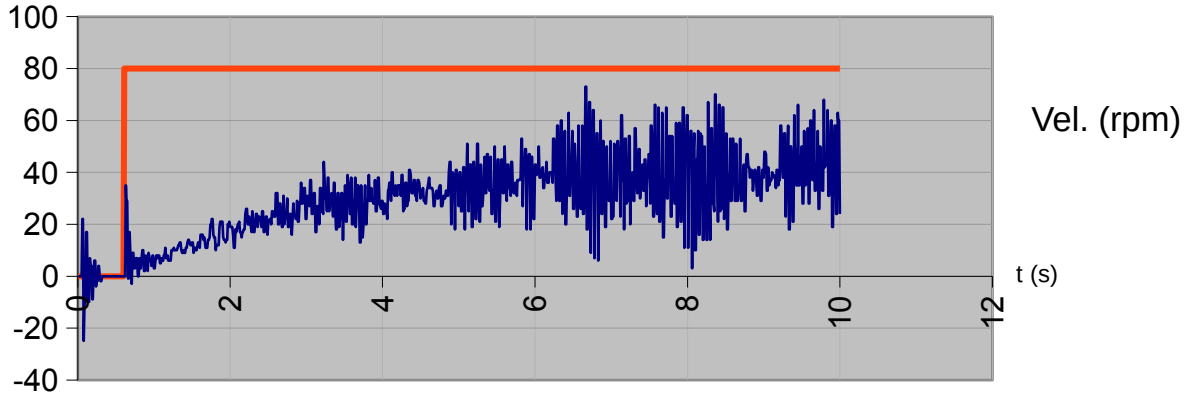


Figura 43: respuesta de la corriente de par ante escalon de velocidad. $K_{p,\omega}$=1000, $K_{i,\omega}$=500

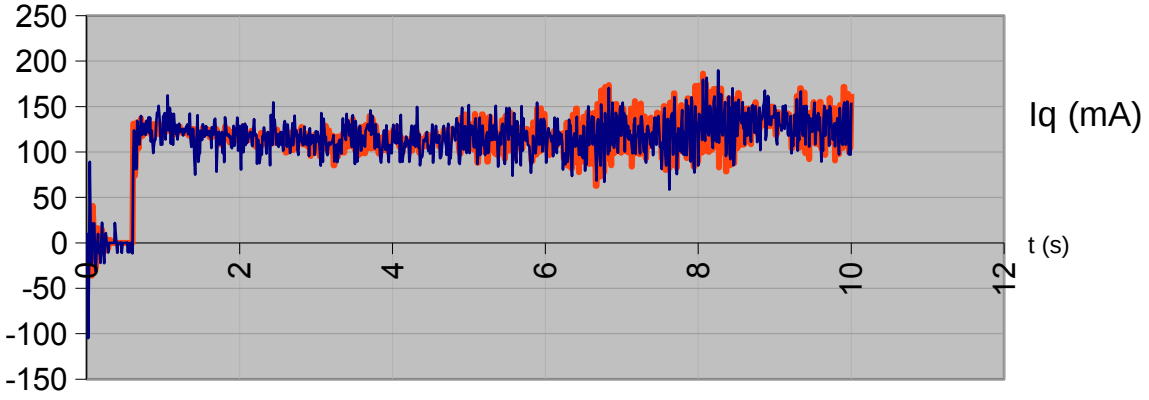


Figura 44: respuesta de la corriente de par ante escalon de velocidad. $K_{p,\omega}$=1000, $K_{i,\omega}$=500

Como puede observarse, la respuesta en corriente es altamente precisa, pero la respuesta en velocidad dista mucho de alcanzar siquiera la velocidad de referencia. Por ello, se prueba a ajustar

un valor mucho más alto de la componente integral. La respuesta a esta prueba puede verse en la figura 45.

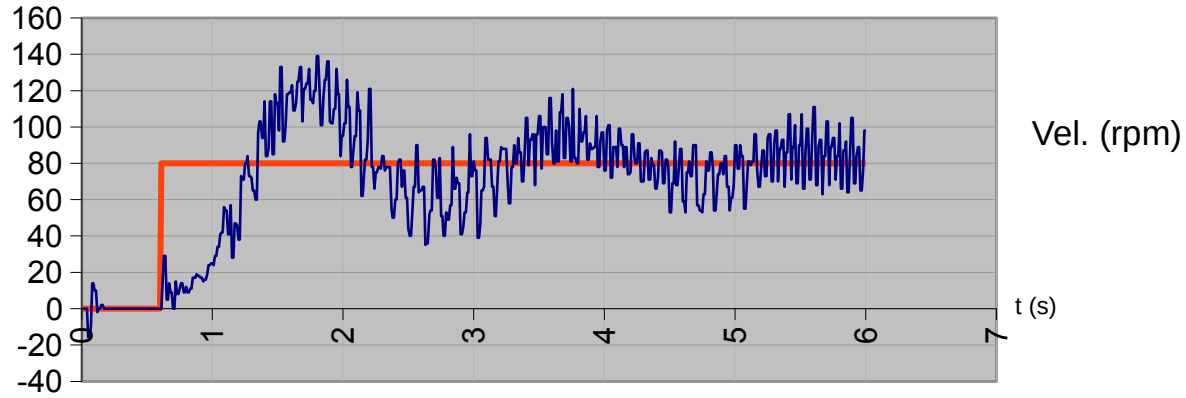


Figura 45: respuesta ante escalon de velocidad. Kp=500, Ki=1000

Aunque con estos parámetros sí se alcanza la velocidad de referencia, la respuesta es nuevamente oscilante y errática.

Considerando las fuertes y rápidas variaciones que se observan en todos los casos, parece claro que los cambios a la salida del PI de velocidad producen una variación excesiva en la corriente de par. Además, si aumentamos más la ganancia del PI de velocidad, la salida de corriente acaba saturando, como puede observarse en las figuras 46 y 47.

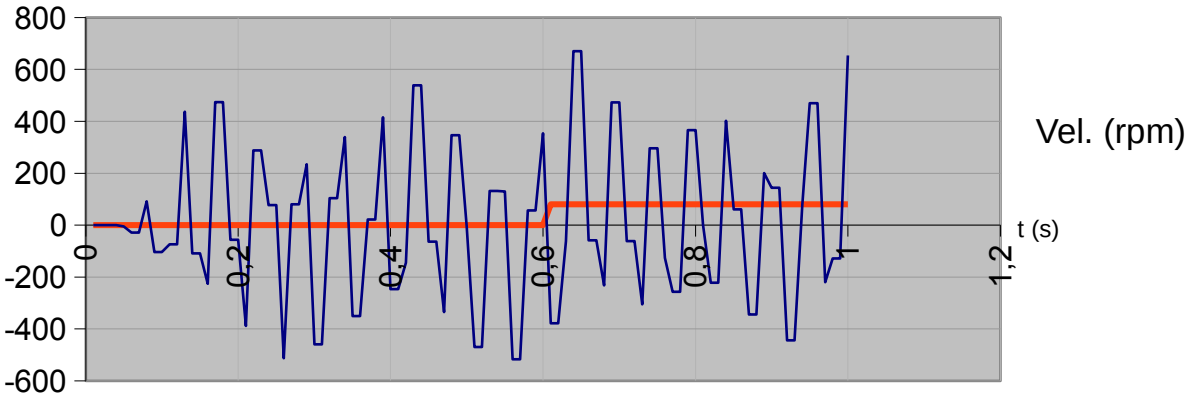


Figura 46: respuesta de la velocidad ante entrada escalón de velocidad. PI de velocidad: kp=5000, ki=225. PIs de corriente: kp=5000, ki=2500

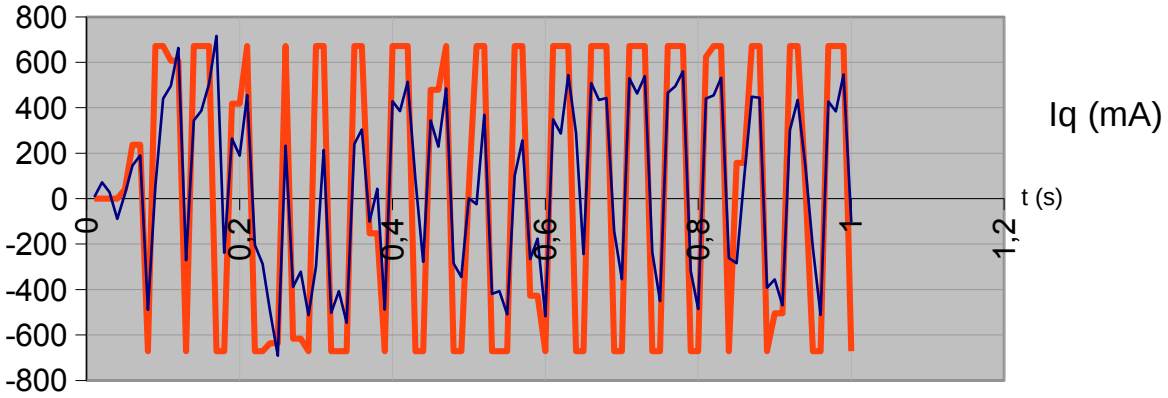


Figura 47: respuesta de la corriente de par ante entrada escalón de velocidad. PI de velocidad: kp=5000, ki=225. PIs de corriente: kp=5000, ki=2500

8.1.3. Reajuste de los controladores PI de corriente

Se decide por tanto reajustar los parámetros de los controladores PI de corriente, disminuyendo su ganancia. La respuesta ante entrada escalón de corriente en las nuevas pruebas realizadas puede verse en las figuras 48 y 49.

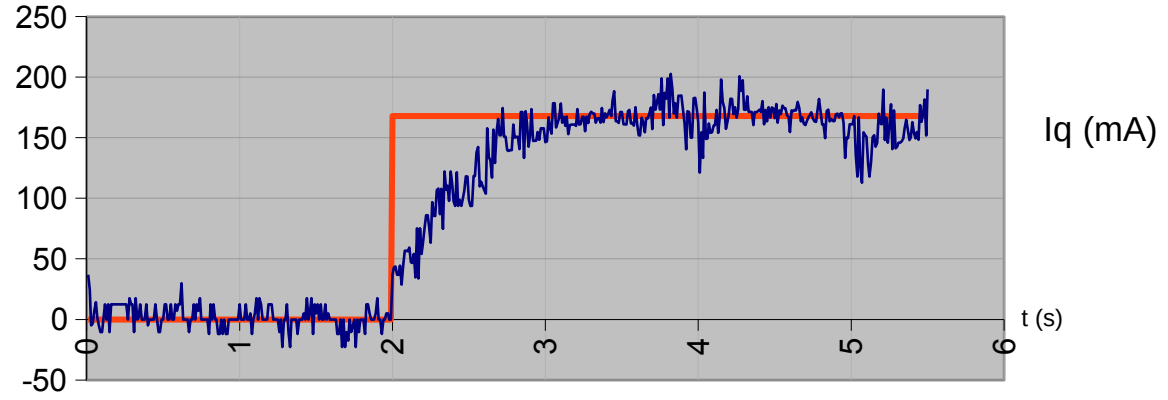


Figura 48: respuesta ante escalon de corriente. Kp=2000, Ki=100

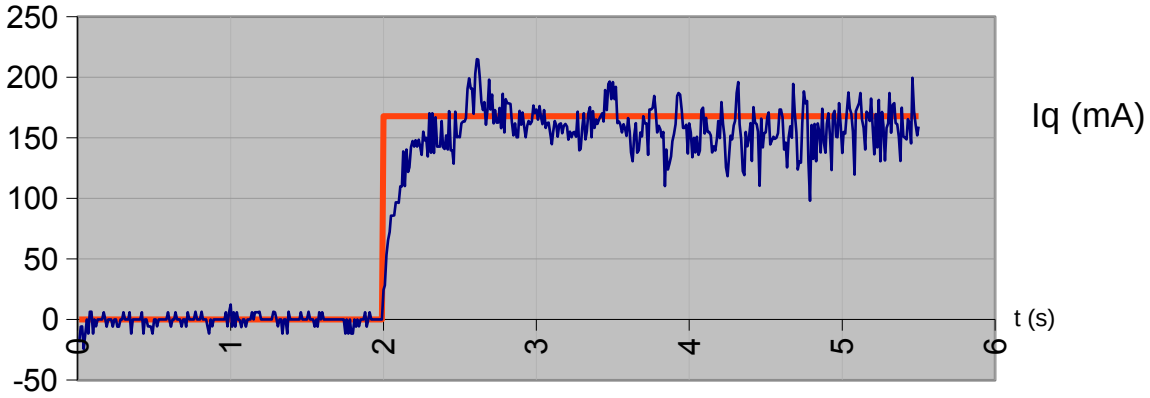


Figura 49: respuesta ante escalon de corriente. Kp=3000, Ki=50

Considerando los resultados obtenidos, se decide utilizar los siguientes parámetros para los controladores PI de corriente:

- Kp: 3000
- Ki: 50

Con estos valores, aunque la respuesta no es tan precisa como la que se había conseguido en un principio, es aún así suficientemente buena, tanto en rapidez como en precisión. Se espera que con estos valores, de menor ganancia, se obtenga una buena respuesta en velocidad, evitando la saturación en corriente.

8.1.4. Reajuste del controlador de velocidades

Una vez reajustados los parámetros de los controladores PI de corriente, se procede a ajustar los del controlador PI de velocidad, nuevamente mediante ensayos cuyos resultados pueden verse en las figuras 50 a 53.

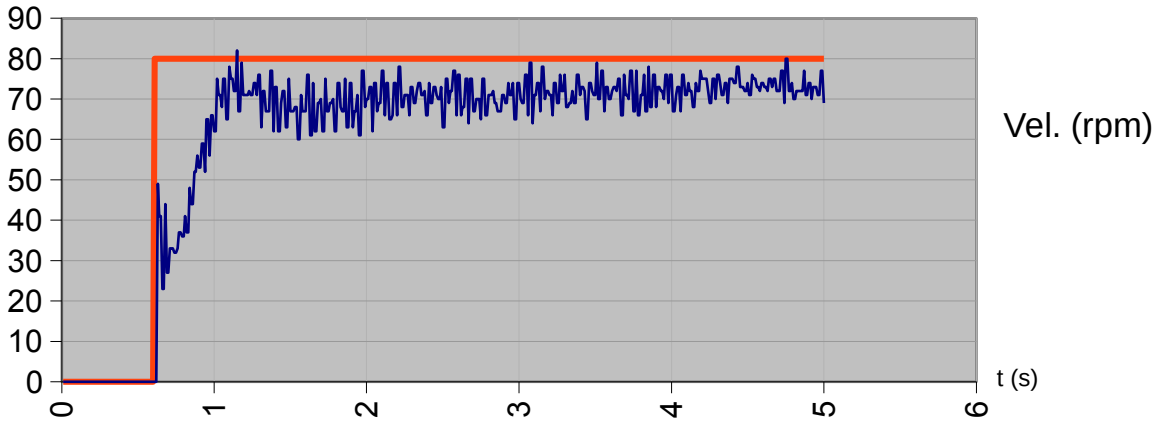


Figura 50: respuesta de la velocidad ante escalón de velocidad. $K_{p,\omega}$=5000, $K_{i,\omega}$=225

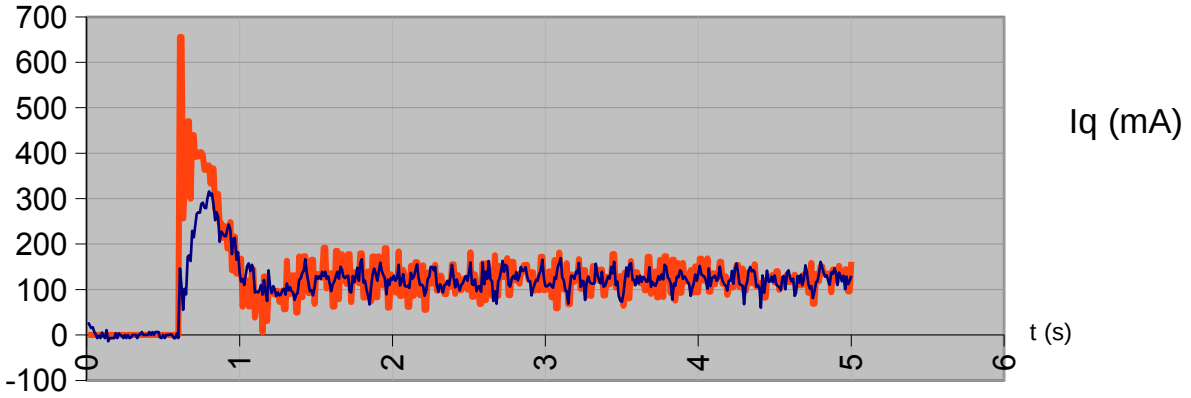


Figura 51: respuesta de la corriente de par ante escalón de velocidad. $K_{p,\omega}$=5000, $K_{i,\omega}$=225

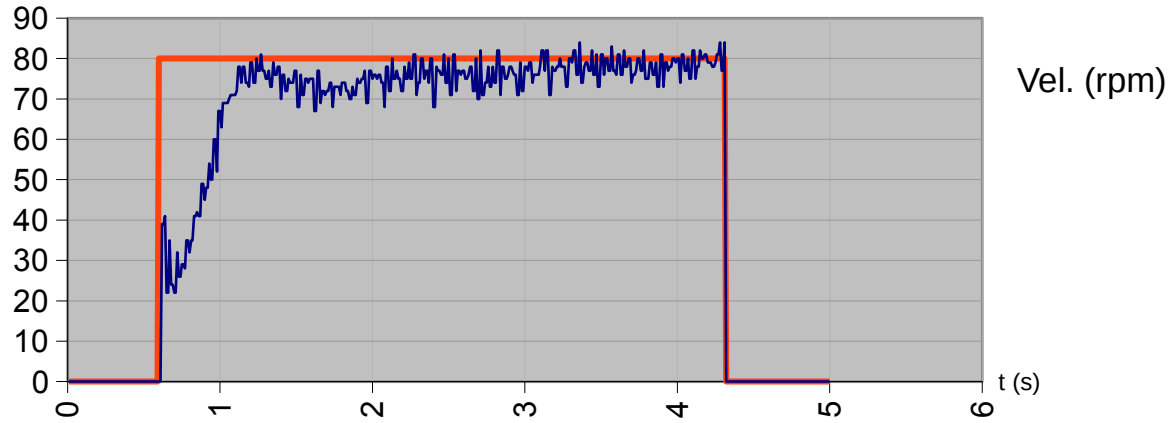


Figura 52: respuesta de la velocidad ante escalón de velocidad. $K_{p,\omega}$=4000, $K_{i,\omega}$=600

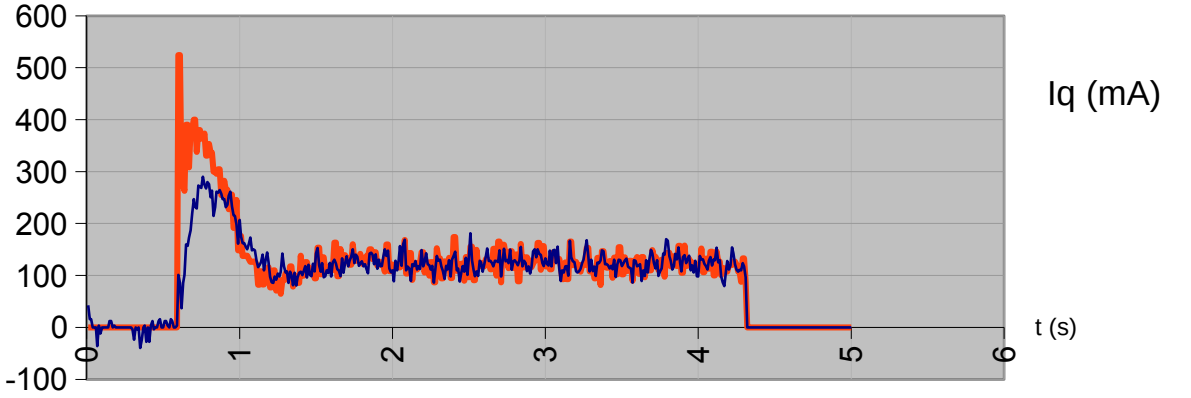


Figura 53: respuesta de la corriente de par ante escalón de velocidad. $K_{p,\omega}$=4000, $K_{i,\omega}$=600

En vista de los resultados, se deciden utilizar los siguientes parámetros para el controlador PI de velocidad:

- Kp: 4000
- Ki: 600

Una vez ajustados los parámetros, se lleva a cabo una nueva prueba para cerciorarnos de que la respuesta dl controlador PI de corriente de flujo (Id) es adecuada. El resultado puede verse en la figura 54.

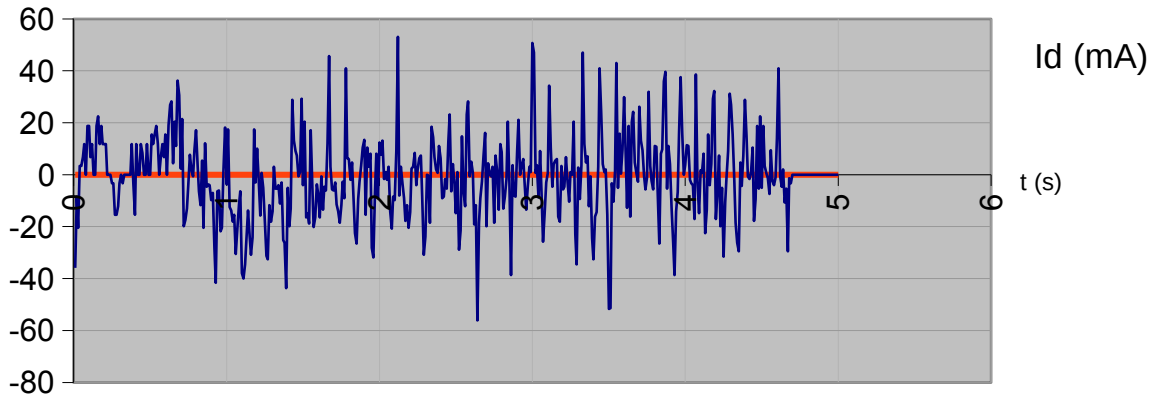


Figura 54: respuesta ante escalón de velocidad de la corriente de flujo (Id)

8.2. Comprobación de la respuesta sin carga en el eje

La respuesta del sistema frente a escalones de corriente y velocidad puede verse en las figuras 55 a 57.

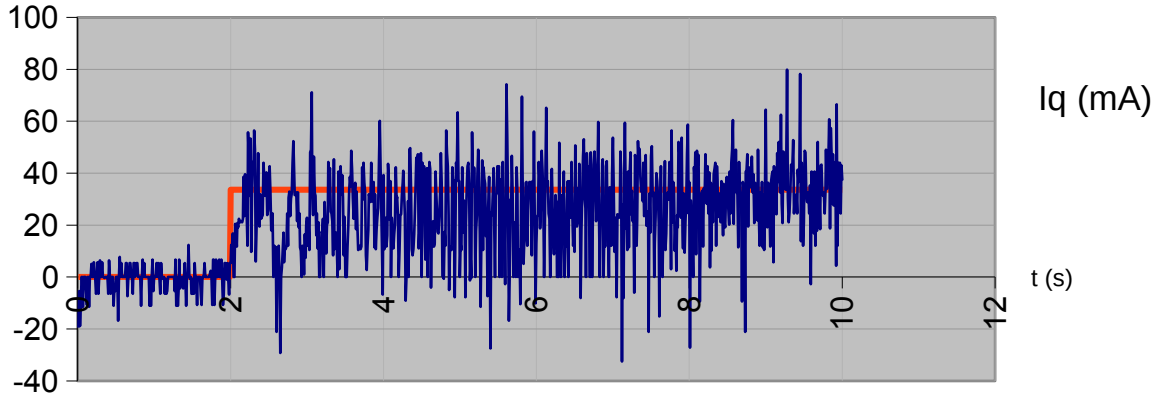


Figura 55: respuesta de la corriente de par ante escalón de corriente sin carga en el eje

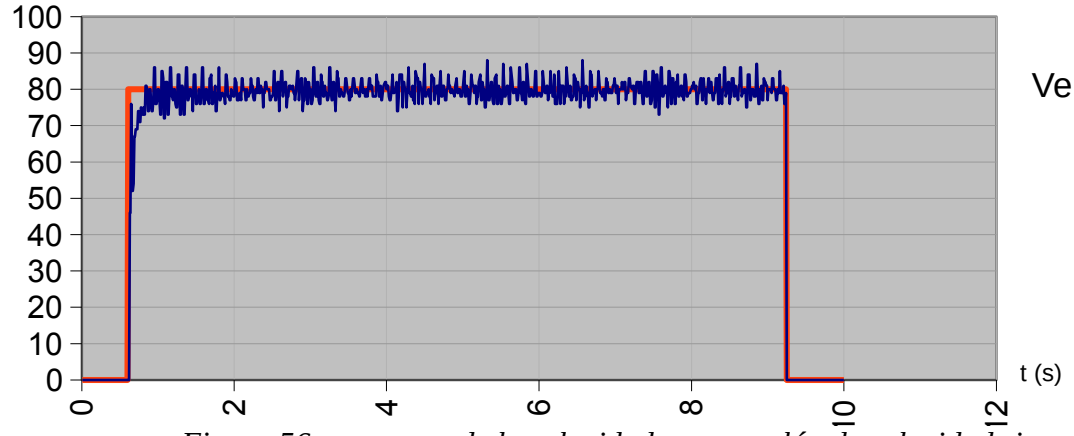


Figura 56: respuesta de la velocidad ante escalón de velocidad sin carga en el eje

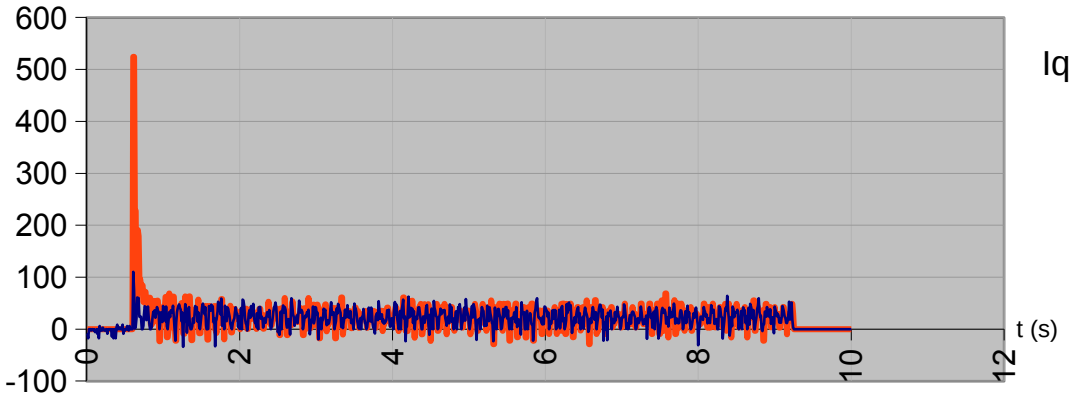


Figura 57: respuesta de la corriente de par ante escalón de velocidad sin carga en el eje

8.3. Respuesta del sistema con curva de velocidades

Se efectuaron pruebas de la respuesta del sistema aplicando las curvas de velocidad descritas en el capítulo “Error: Reference source not found“.

A la vista de los resultados del primer ensayo, que pueden verse en la figura 58, se consideró que el rizado en la velocidad era aún excesivo, y se decidió reducir aún más la ganancia de los controladores de corriente, evitando así cambios tan bruscos en la velocidad. El efecto de dicho cambio fue satisfactorio, tal y como puede comprobarse en las figuras .

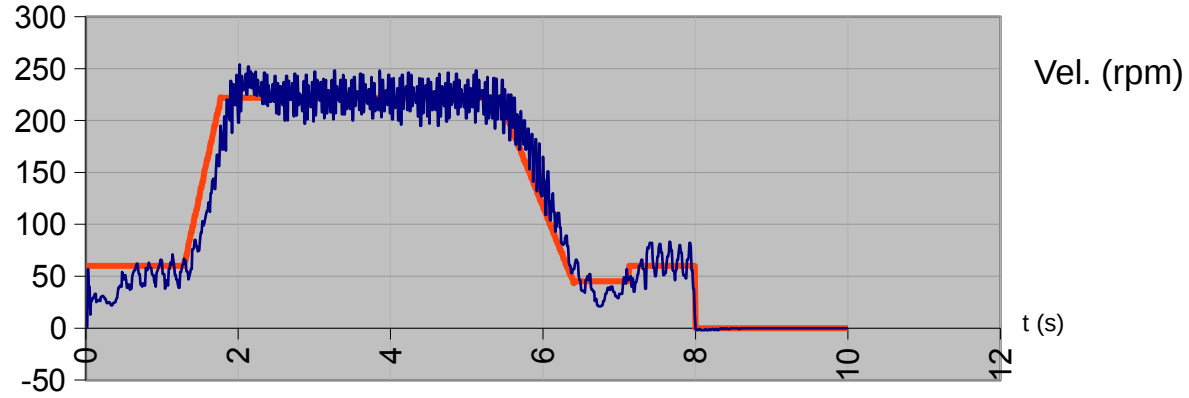


Figura 58: respuesta de la velocidad en movimiento de cierre

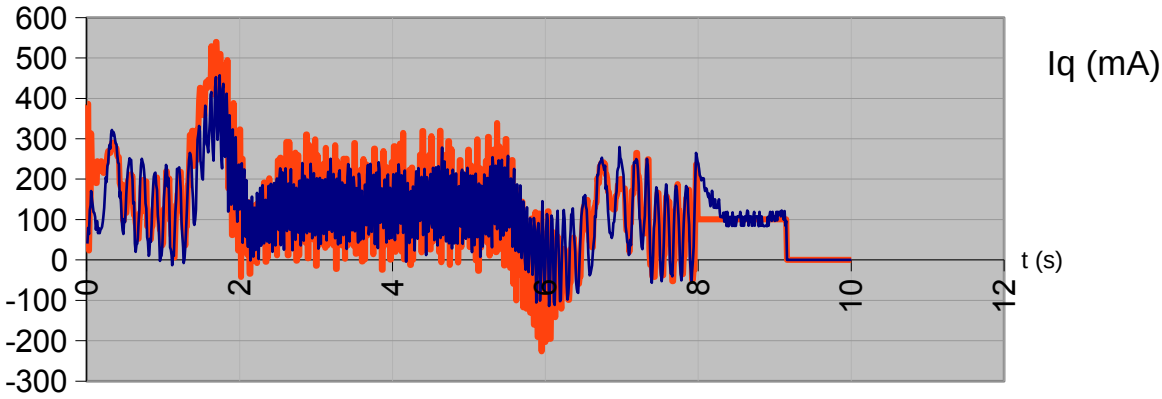


Figura 59: respuesta de la corriente de par en movimiento de cierre

Para intentar comprobar el buen funcionamiento del sistema ante cualquier situación posible, se decidió hacer ensayos que cubrieran las siguientes situaciones:

- Tipo de movimiento:
  - Cierre
  - Apertura
- Obstrucciones:
  - Sin obstrucción
  - Con obstrucción, apertura automática desactivada
  - Con obstrución, apertura automática activada
- Peso de la puerta:
  - Peso bajo
  - Peso alto

8.3.1. Cierre, sin obstrucción, peso bajo

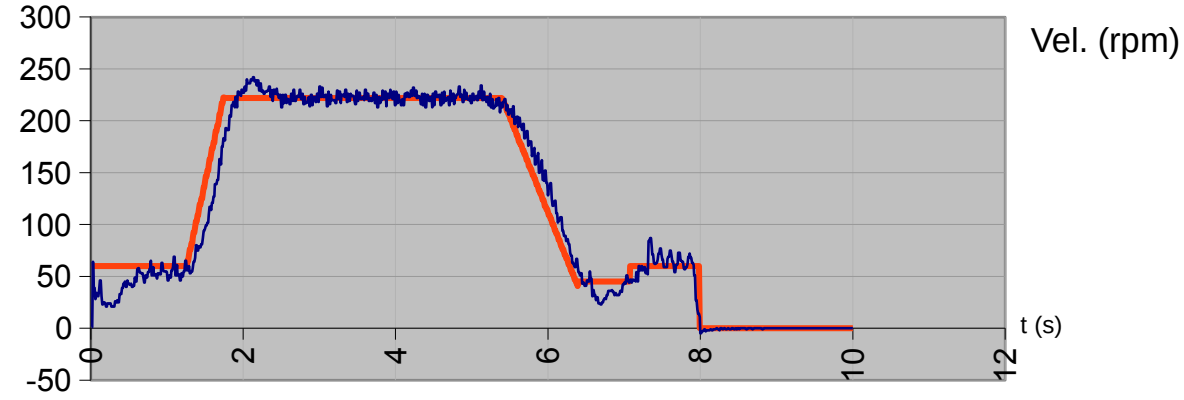


Figura 60: respuesta de la velocidad en movimiento de cierre, peso bajo

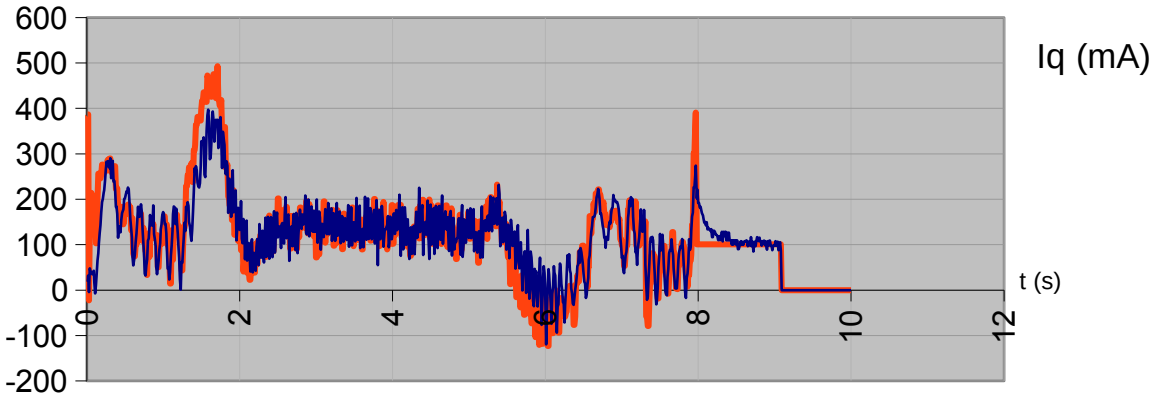


Figura 61: respuesta de la corriente de par en movimiento de cierre, peso bajo

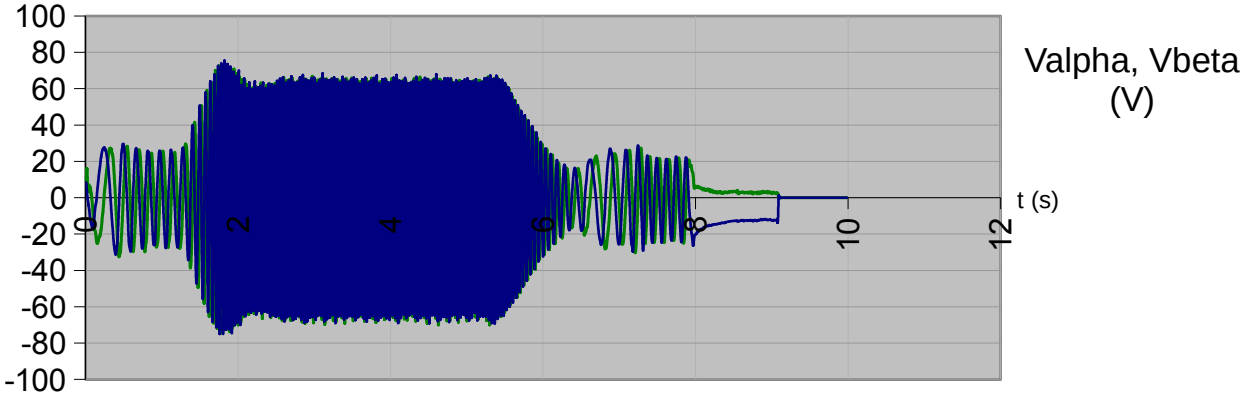


Figura 62: tensión en movimiento de cierre, peso bajo

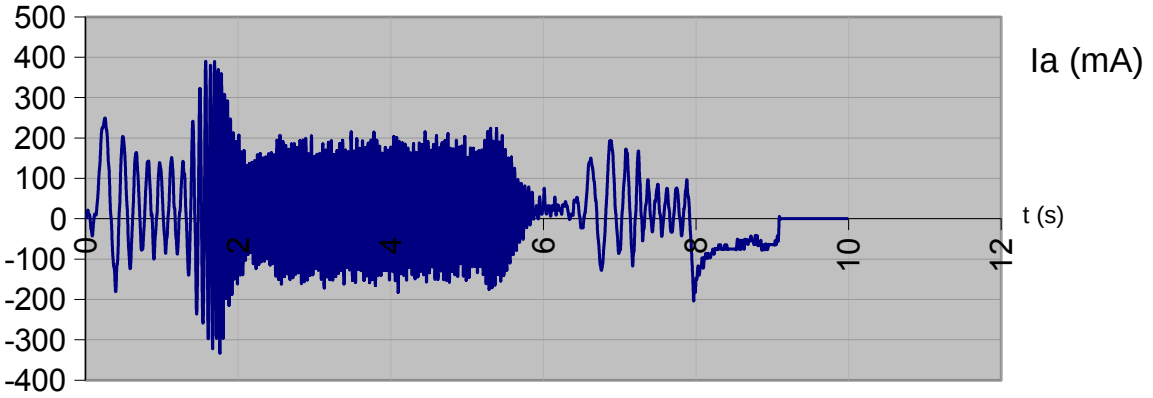


Figura 63: corriente de fase en movimiento de cierre, peso bajo

8.3.2. Cierre, con obstrucción, detección activada, peso bajo

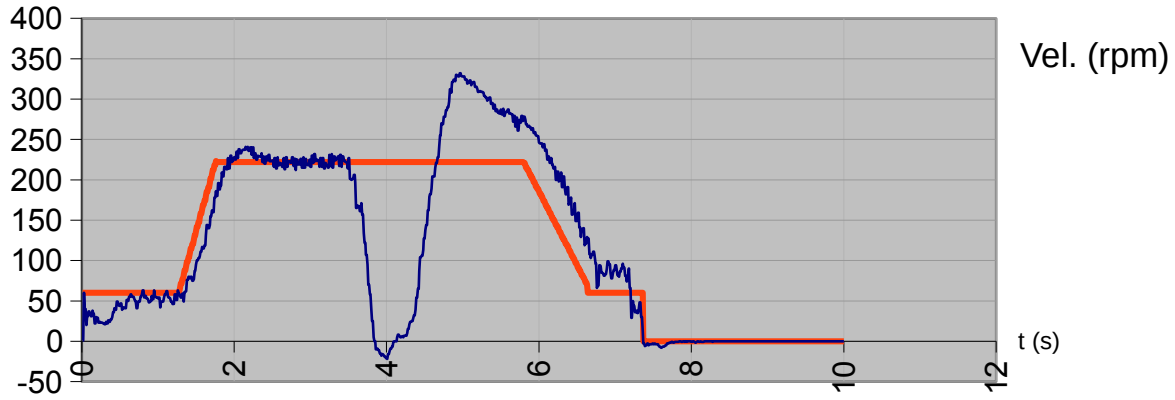


Figura 64: respuesta de velocidad en cierre, con obstáculo, detección desactivada, peso bajo

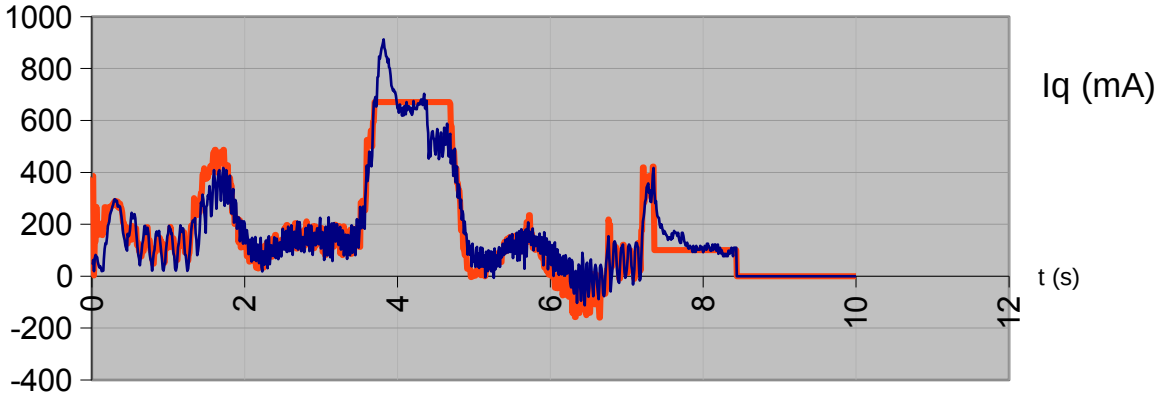


Figura 65: respuesta de corriente de par en cierre, con obstáculo, detección desactivada, peso bajo

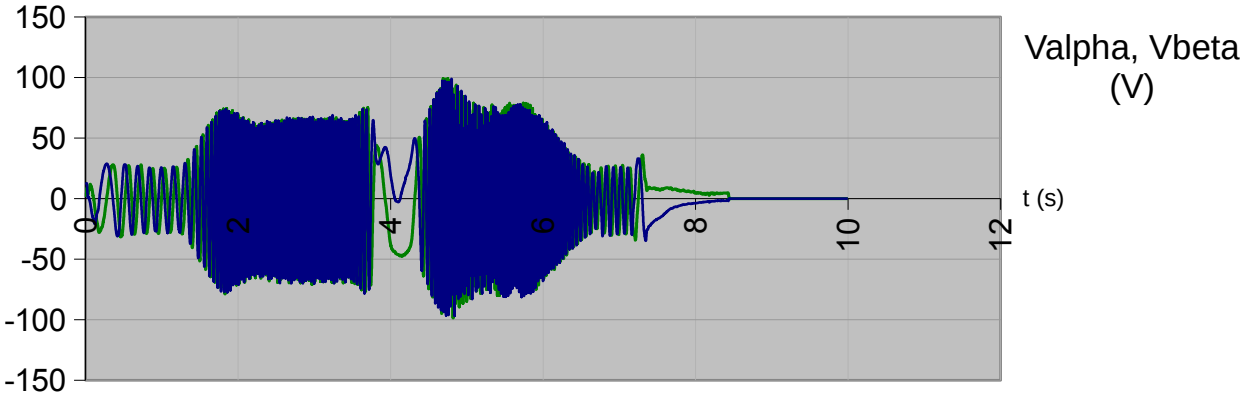


Figura 66: tensión en movimiento de cierre, con obstáculo, detección desactivada, peso bajo

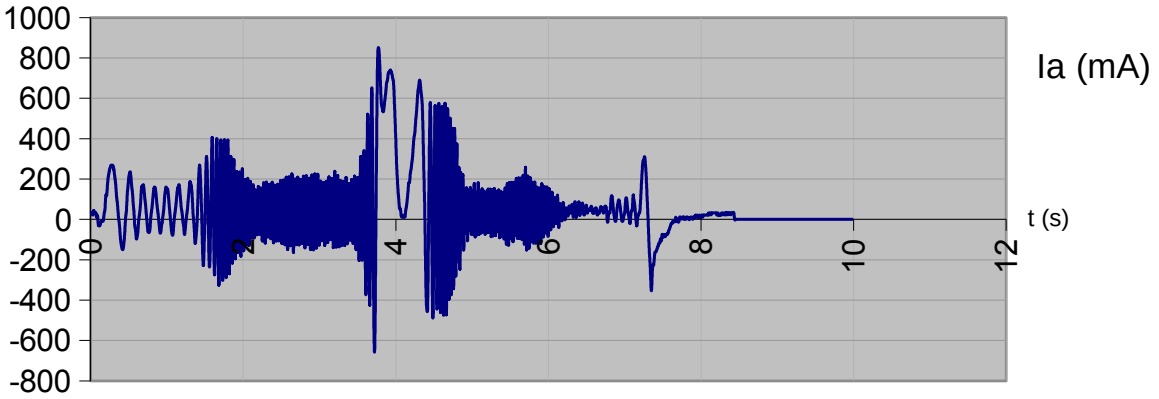


Figura 67: corriente de fase en cierre, con obstáculo, detección desactivada, peso bajo

8.3.3. Cierre, con obstrución, detección activada, peso bajo

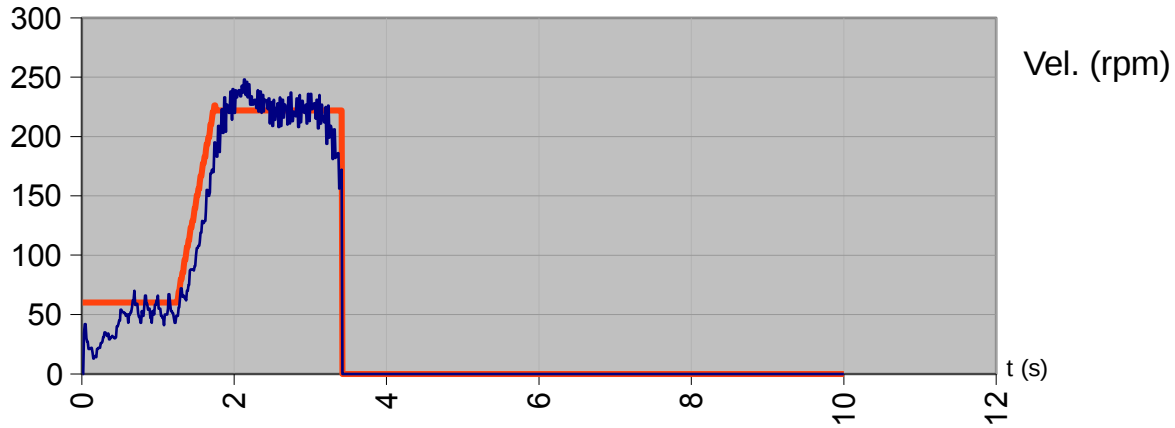


Figura 68: respuesta de la velocidad en cierre, con obstáculo, detección activada, peso bajo

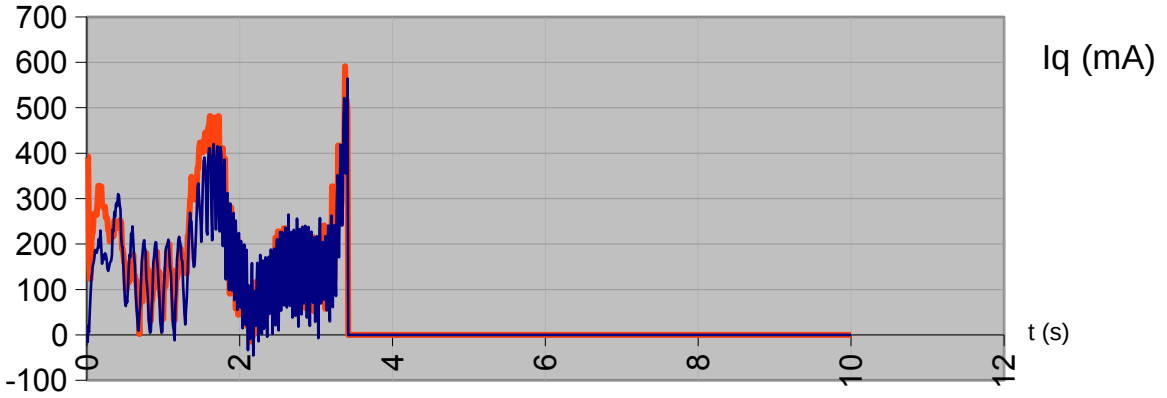


Figura 69: respuesta de la corriente de par en cierre, con obstáculo, detección activada, peso bajo

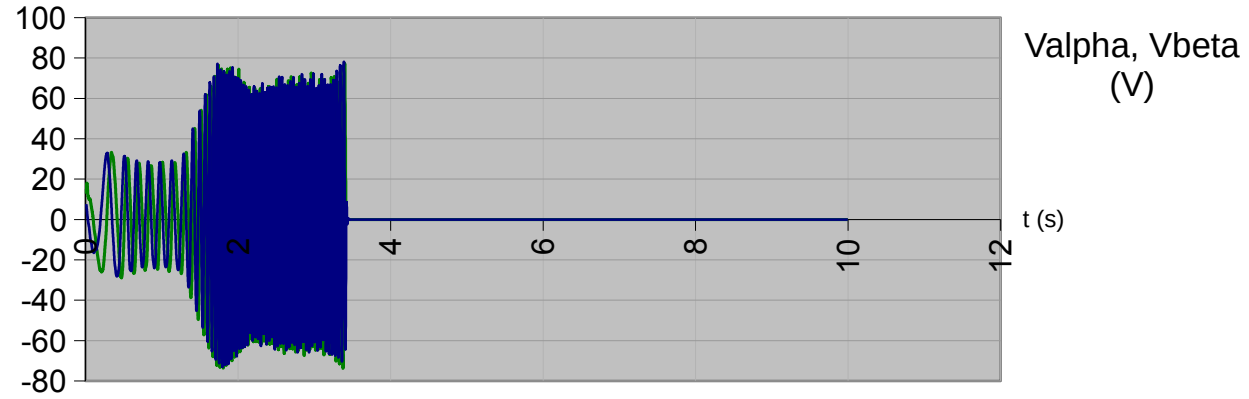


Figura 70: tensón en movimiento de cierre, con obstáculo, detección activada, peso bajo

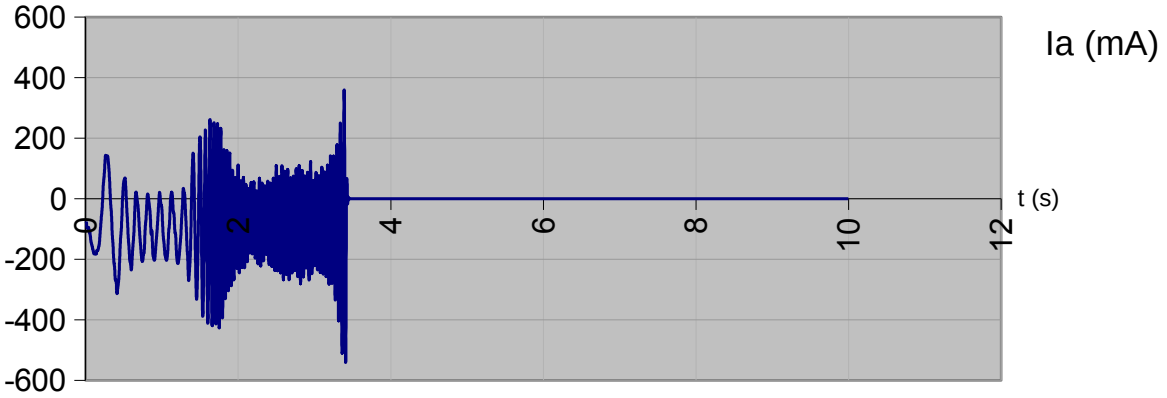


Figura 71: corriente de fase en movimiento de cierre, con obstáculo, detección activada, peso bajo

8.3.4. Cierre, sin obstucción, peso alto

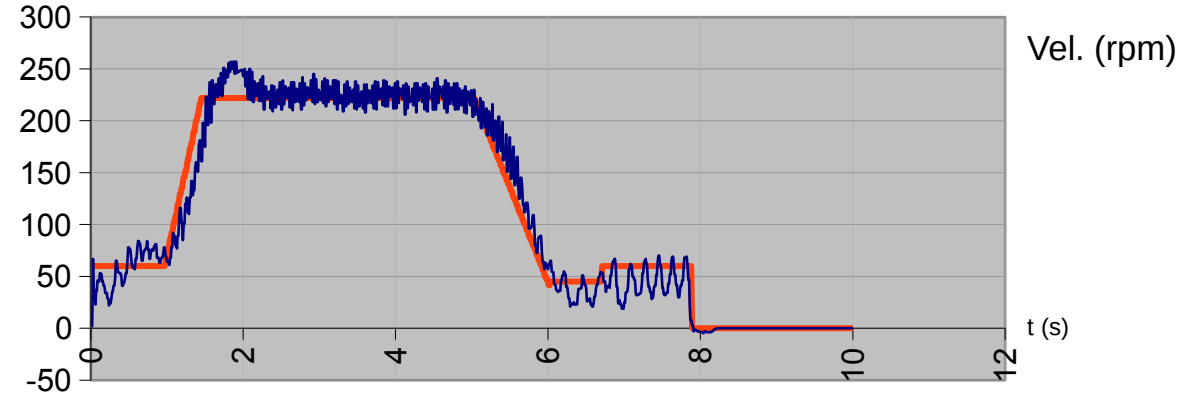


Figura 72: respuesta de la velocidad en movimiento de cierre, sin obstáculo, peso alto

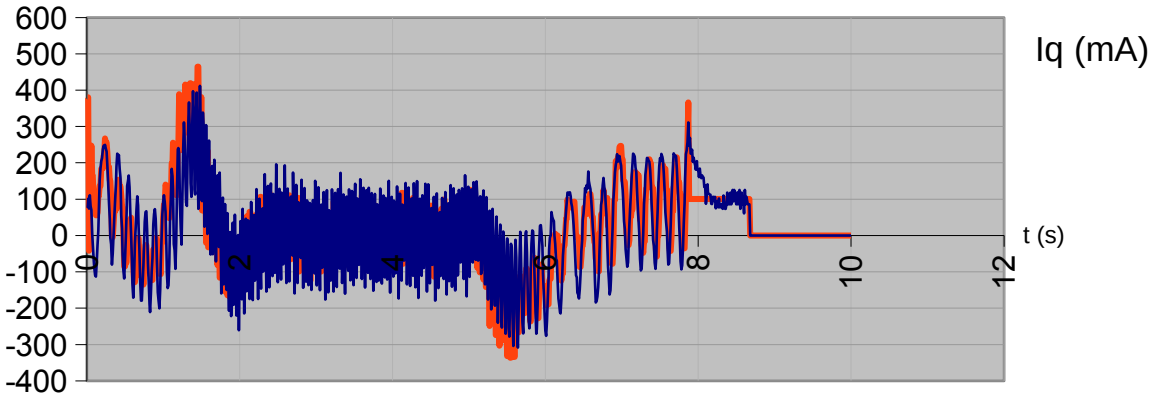


Figura 73: respuesta de la corriente de par en movimiento de cierre, sin obstáculo, peso alto

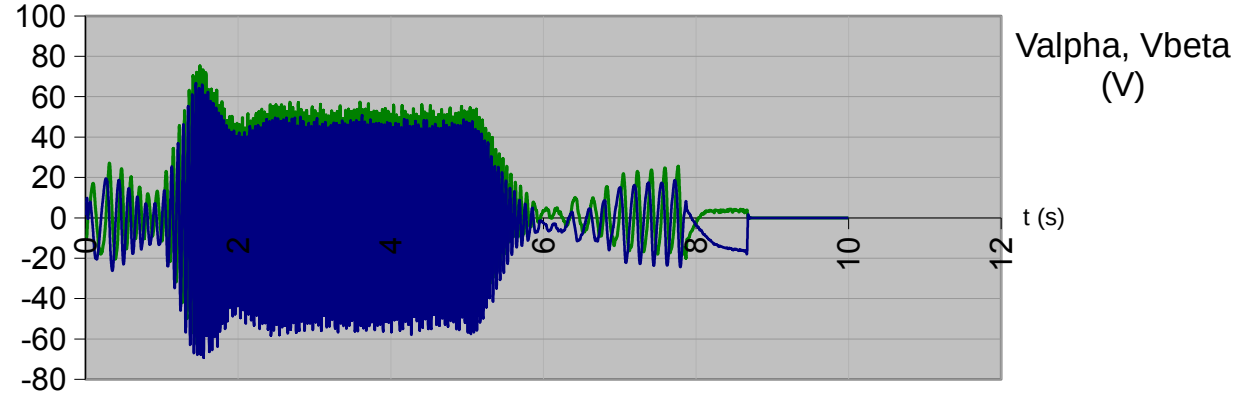


Figura 74: tensión en movimiento de cierre, sin obstáculo, peso alto

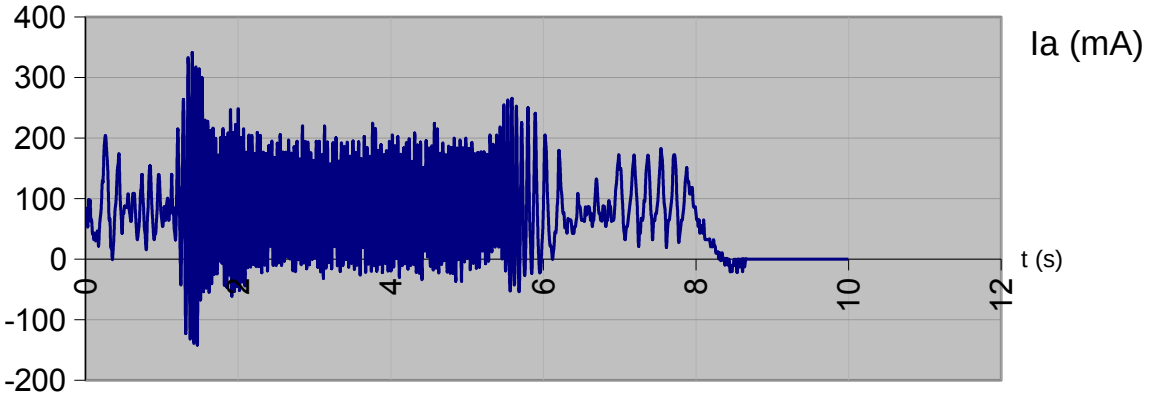


Figura 75: corriente de fase en movimiento de cierre, sin obstáculo, peso alto

8.3.5. Apertura, sin obstáculo, peso alto

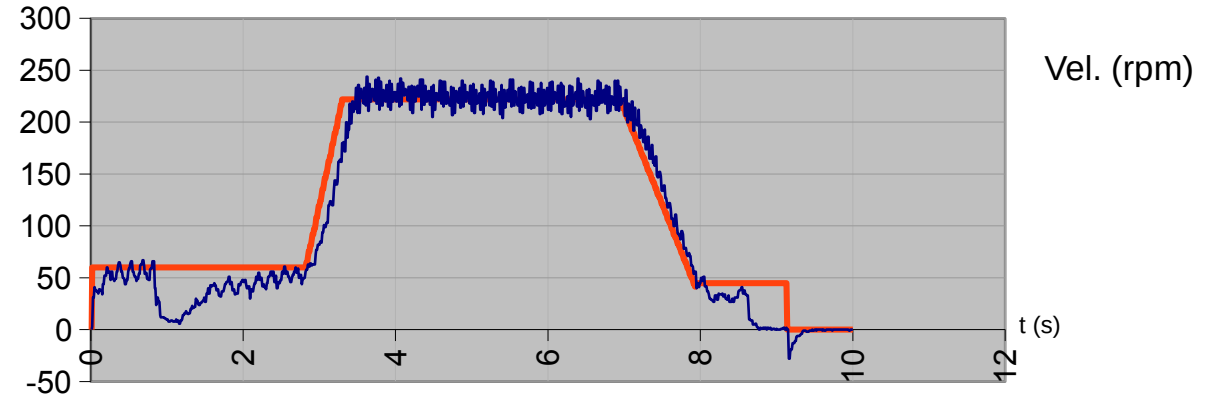


Figura 76: respuesta de la velocidad en movimiento de apertura, sin obstáculo, peso alto

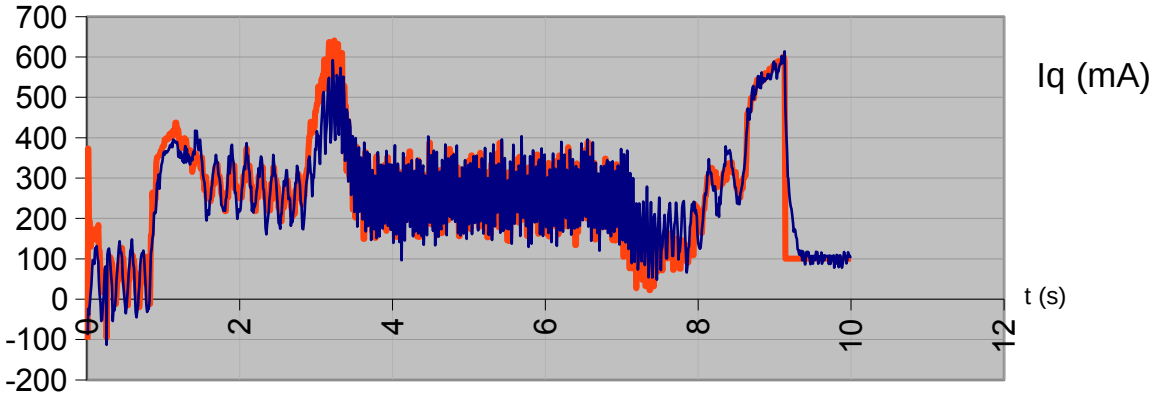


Figura 77: respuesta de la corriente de par en movimiento de apertura, sin obstáculo, peso alto

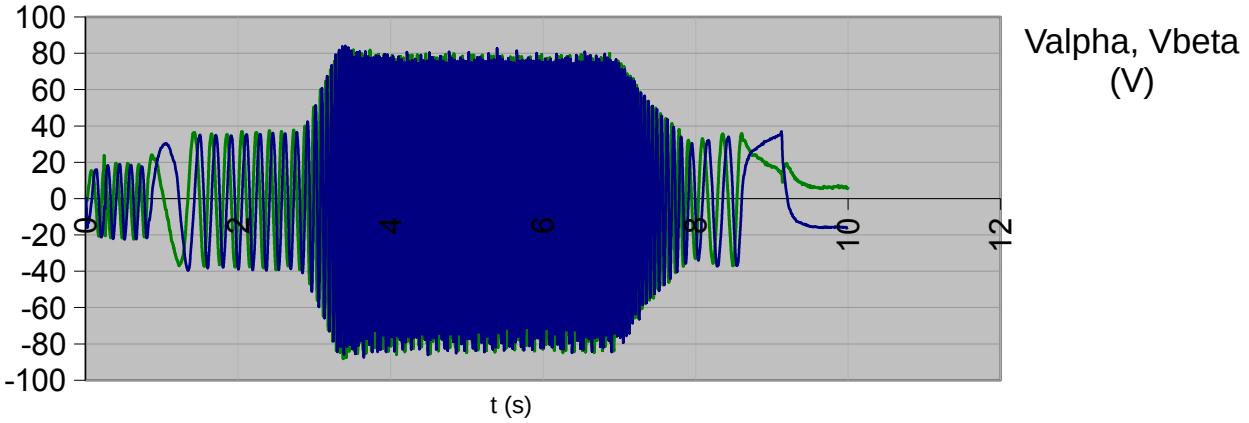


Figura 78: respuesta de la tensión en movimiento de apertura, sin obstáculo, peso alto

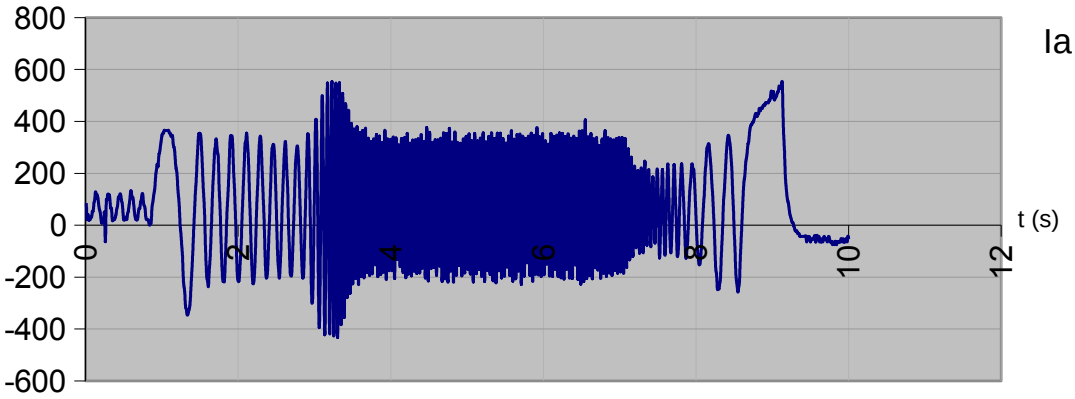


Figura 79: corriente de fase en movimiento de apertura, sin obstáculo, peso alto

9. Presupuesto

Por no haberse completado de forma definitiva la fase de diseño y fabricación del hardware, se detalla tan solo el presupuesto del desarrollo del software, parte de la que se ha encargado quien redacta este documento.

También se incluye el trabajo realizado en pruebas del hardware, en concreto de la parte de alimentación.

Concepto	**Precio unitario**	**Cantidad**	**Precio total**
Desarrollo del software	12€/h	350h	4200€
Tarjeta de evaluación MCHV-2	509,25€/ud	1	509,25€
Tarjeta de evaluación MCLV-2	145,50€/ud	2	191€
Pruebas de hardware	12€/h	105h	1260€
Material para pruebas del hardware	-	-	45€
TOTAL			6205,25€

10. Conclusiones

Tal y como se ha explicado en el presente documento, se han cubierto los siguientes objetivos, requisitos indispensables del proyecto:

- Controlar el motor síncrono de imanes permanentes, utilizando la información de velocidad y posición angular del encoder de cuadratura, utilizando la técnica del control vectorial.
- Ejecución de las órdenes de movimiento propias de una puerta de ascensor, implementando para ello las curvas de velocidad que crean un movimiento rápido y fluido, en función de la posición, así como la detección de obstrucciones. Las órdenes se reciben a través de botones del PCB o a través de señales externas.
- Procesos automáticos de aprendizaje y cálculo de parámetros (longitud de la puerta, offset del ángulo mecánico respecto al eléctrico...)
- Parámetros configurables a través de un menú, gestionable a través de una botonera y un display.
- Ajuste correcto de los parámetros de control, obteniendo una respuesta rápida y sin excesivas oscilaciones.
- Comprobado funcionamiento del circuito de alimentación completo, probado en un hardware prototipo.

En cuanto a los objetivos que no se pueden considerar como completados a la fecha de entrega de este documento, se encuentran en fase final de ejecución. En concreto, estos objetivos son los que se refieren al diseño del hardware que compone el PCB, y a la fecha de la entrega de este documento se ha completado dicho diseño, pero todavía no se han recibido los prototipos fabricados para poder probar su correcto funcionamiento.

11. Desarrollo futuro

Una vez completados los objetivos principales del proyecto, se han planteado diversas propuestas de mejora, con el objetivo principal de aportar mayor valor al producto, dándole una mejor salida comercial. Las mejoras propuestas están orientadas principalmente a añadir mayor variedad de usos al producto mediante mejoras del software.

- Control sin sensores
- Control con relés indicadores de posición de la puerta
- Nuevas funciones en función de señales recibidas a través de los terminales de entrada multifunción (señal de hora punta, de incendio, etc)
- Nuevas señales de salida para enviar a través de los terminales de salida multifunción.

Otras de las posibles mejoras que se barajan tienen el objetivo de conseguir un ahorro económico que permita una mayor competitividad en la venta del producto:

- Elección de un microcontrolador de la misma familia, compatible y equivalente, con menos memoria flash y menos memoria RAM.
- Elección de un microcontrolador de la misma familia, compatible y equivalente, con un menor número de pines (44 en lugar de 64)
- Supresión de componentes no estrictamente necesarios.
- Otros

12. Bibliografía

A continuación se detalla la bibliografía consultada para la realización del presente Trabajo Fin de Grado.

En caso de aquellos documentos a los que es posible acceder de manera pública y gratuita a través de internet se adjunta la URL a dicho documento.

12.1. Control vectorial

- Control vectorial del motor asíncrono. Angel Vernavá, Roberto Gibbons, Antonio Nachez, Marcelo Arias y Armando Novello.

  Facultad de Ciencias Exactas, Ingeniería y Agrimesura. Universidad Nacional de Rosario.

  http://potencia.eie.fceia.unr.edu.ar/CONTROL%20VECTORIAL.pdf

- Control vectorial de la velocidad de una máquina síncrona de imanes permanentes. Pedro Celestino Castellano Morales.

  Tesis, Instituto Politécnico Nacional (México)

  http://www.biblio-sepi.esimez.ipn.mx/electrica/2011/Control%20vectorial%20de%20la%20velocidad%20de%20una%20maquina%20sincrona.pdf

- Diseño de un controlador de velocidad para un motor de inducción utilizando control Vectorial. Diana Fernanda Morales Rincón y Jorge Olmedo Vanegas Serna.

  Trabajo fin de carrera, universidad de Cauca (Colombia)

  ftp://ftp.unicauca.edu.co/Facultades/FIET/DEIC/Materias/Control%20de%20maquinas/Tesis/Tesis%20JorgeDiana/Control%20Vectorial.pdf

- Nota de aplicación AN1078: “control PMSM sensorless FOC”. Microchip

  ref.: DS01078B – microchip.com

  http://ww1.microchip.com/downloads/en/AppNotes/01078B.pdf

12.2. Control de motores

- Apuntes de máquinas eléctricas

  Área ingeniería eléctrica, universidad de Alcalá

  http://delep.uah.es/antigua/PLANES%20ANTIGUOS/ITI/2/Maquinas%20Electricas/Apuntes/TME4.pdf

- Apuntes de clase de las asignaturas máquinas I, máquinas II y variación de velocidad
- Wikipedia

12.3. Microcontrolador PIC: especificaciones y programación

- Datasheet del dsPIC33EPxxxMC20x/50x. Microchip

  ref.: DS70000657H – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/70000657H.pdf
- 16-Bit Language Tool Libraries - Reference Manual. Microchip

  ref.: DS50001456H – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/50001456H.pdf
- 16-Bit MCU and DSC Programmer's Reference Manual. Michrochip

  ref.: DS70157F – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/70157F.pdf
- dsPIC33E/PIC24E Family Reference Manual, Section 5: “Flash Programming”. Microchip

  ref.: DS70609D – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/70609D.pdf
- dsPIC33E/PIC24E Family Reference Manual, Section 15: “Quadrature Encoder Interface (QEI)”. Microchip

  ref.: DS70601B – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/S15.pdf
- dsPIC33E/PIC24E Family Reference Manual, Section 10: “I/O Ports”. Microchip

  ref.: DS70000598 – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/70000598c.pdf
- dsPIC33E/PIC24E Family Reference Manual, Section 2: “CPU”. Microchip

  ref.: DS70359B – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/S2.pdf
- dsPIC33/PIC24 Family Reference Manual: “Interrupts”. Microchip

  ref.: DS70000600D – microchip.com

http://ww1.microchip.com/downloads/en/DeviceDoc/70000600d.pdf

- dsPIC33/PIC24 Family Reference Manual: “Oscillator”. Microchip

  ref.: DS70580C – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/70580C.pdf

- Infinidad de contribuciones desinteresadas de usuarios en multitud de webs, blogs, foros, etc.

12.4. Tarjetas de evaluación de Microchip

- dsPICDEM MCLV-2 Development Board User Guide. Microchip

  ref.: DS52080A – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/DS-52080a.pdf

- dsPICDEM MCHV-2 Development Board User Guide. Microchip

  ref.: DS52074A – microchip.com

  http://ww1.microchip.com/downloads/en/DeviceDoc/52074A.pdf

12.5. Software de control vectorial para dsPIC33E

- Nota de aplicación AN1078: “control PMSM sensorless FOC”. Microchip

  ref.: DS01078B – microchip.com

  http://ww1.microchip.com/downloads/en/AppNotes/01078B.pdf

- Documento “AN1078 Tuning Guide”. Microchip

  ref.: DS70638A – microchip.com

  http://ww1.microchip.com/downloads/en/AppNotes/70638A.pdf

- Contribuciones de los usuarios en los foros de Microchip

  http://www.microchip.com/forums/

Anexo 1

código desarrollado

```c
#include "general.h"
#include "botonera.h"

#define delay200ns() { Nop();Nop();Nop();Nop();Nop();Nop();Nop();Nop();Nop
#define COMPROB_PULS_DELAY 50

tPulsacion leerBotonPulsado (void) {
    static unsigned int prev = 0U, btnTimeCnt = 0U, btnMantCnt = 0U;
    unsigned int i, sum, bits;
    tPulsacion puls = { 0U, False, False };

    // recibe los bits del shift register
    PULSLOAD = 1;       // carga los datos paralelo en el registro serie
    delay200ns();
    PULSLOAD = 0;       // deja de cargar los datos paralelo en el registr
    bits = 0U;
    for ( i=0U; i<8U; i++ ) {   // transmite los datos
        delay200ns();           // espera 200ns
        bits = bits<<1;         // lee el dato
        bits |= PULSDATA & 0x0001;
        PULSCLK = 0;            // flanco negativo, que no tiene ningun ef
        delay200ns();           // espera
        PULSCLK = 1;            // flanco positivo: prepara el siguiente c
    }

    // determina que boton se ha pulsado
    sum = 0U;                       // cuenta cuantos botones hay pulsados
    for ( i=0U; i<6U; i++ )
        sum += bits>>i & 0x0001;
    if ( sum == 0 ) {               // si no es ninguno lo toma como 0
        puls.btn = 0U;
    }
    else if ( sum == 1 ) {          // si es solo uno, lo toma como pulsac
        for ( i=0U; !(bits>>i & 0x0001); i++ )
            ;
        puls.btn = i+1;
    }
    else if ( bits>>prev-1 & 0x0001 ) { // si es mas de uno, y prev sigue
        puls.btn = prev;
    }
    else {                          // si es mas de uno y prev ya no esta
        for ( i=0U; !(bits>>i & 0x0001); i++ )
            ;
        puls.btn = i+1;
    }

    // si hay alguno pulsado, determina cuanto tiempo lleva. Si es poco, n
    if ( puls.btn && puls.btn==prev ) {
```

```
49          if ( btnTimeCnt < 20U )     // para saber si ya se considera pulsa
50              btnTimeCnt++;
51          if ( btnMantCnt == 10U )    // para saber si se esta manteniendo p
52              btnMantCnt = 0U;
53          else
54              btnMantCnt++;
55      }
56      else {
57          btnTimeCnt = 0U;
58          btnMantCnt = 0U;
59      }
60      if ( btnTimeCnt < 3U )          // se considera pulsado?
61          puls.btn = 0;
62      else if ( btnTimeCnt == 20U )   // es pulsacion larga?
63          puls.pulsLarga = True;
64      if ( btnMantCnt == 10U )        // es pulsacion mantenida?
65          puls.pulsMantenida = True;
66
67      // Delay entre comprobaciones
68      Delay(COMPROB_PULS_DELAY);
69
70      prev = puls.btn;
71      return puls;
72  }
73
74  bool esperarPulsLarga (tPulsacion puls) {
75      while ( puls.btn ) {
76          if ( puls.pulsLarga )
77              return True;
78          puls = leerBotonPulsado();
79      }
80      return False;
81  }
```

```
/*
 * File:   botonera.h
 * Author: Iñigo Huguet (Fanamoel)
 *
 * Created on June 6, 2014, 9:31 AM
 */

#ifndef BOTONERA_H
#define BOTONERA_H

typedef enum {BTN_MODE=1, BTN_ARRIBA, BTN_ABAJO, BTN_RUN, BTN_SET, BTN_DOWN
typedef struct {
    tBoton btn;
    bool pulsLarga;
    bool pulsMantenida;     // si se mantiene una pulsacion, se pone a True
} tPulsacion;
tPulsacion leerBotonPulsado (void);   // devuelve el tBoton que este pulsad
bool esperarPulsLarga (tPulsacion puls);// esperar a que se compruebe si en

#endif  /* BOTONERA_H */
```

```
#include "general.h"
#include "encoder.h"
#include "user_options_input.h"
#include "controlPuerta.h"

// variables para control avanzado en modo debug
bool acelControlada = True;                    // activa o desactiva las
bool doEscalonVel = False, doEscalonIq = False; // activa o desactiva los
int escalonVel = 800, escalonIq = 500;         // valores de referencia p

// constantes
#define ESCALON_MANTPAR_SECS 2  // segundos con par 0 en modo escalon de c
#define ESCALON_MANTPAR_ITERATIONS ((unsigned long)((double)ESCALON_MANTPA
#define MARGEN_PULSOS 400   // margen de error tolerable en el aprendizaje

// macros
#define setVel(vel) { VelReq = (ctrlStatus.dir ? 1:-1) * usrOptions.revers
#define incVel(vel) { VelReq += (ctrlStatus.dir ? 1:-1) * usrOptions.rever
#define calcParCurr(curr) ({ (ctrlStatus.dir ? 1:-1) * usrOptions.reverse

// variables externas que modifica este archivo
extern tCodigoOrden orden;
extern tCtrlStatus ctrlStatus;
extern tUsrOptions usrOptions;
extern tUGF uGF;
extern int VelReq, prevDir, IqMantener;
extern unsigned long movingTimeIters;
//extern long longPuerta, posPuerta, difLongPuerta;

extern long longPuerta;         // longitud total de la puerta
                                    // unidades: numero de pulsos del enco
                                    // rango: capacidad de la variable SIG
long posPuerta;                 // posicion actual de la puerta
                                    // unidades: numero de pulsos del enco
                                    // rango: capacidad de la variable SIG
long difLongPuerta;             // diferencia de la posicion de la puerta
                                    // unidades: numero de pulsos del enco
                                    // rango: capacidad de la variable SIG


inline void setLongPuerta (unsigned long lng) {
    longPuerta = lng;
}

void controlPuerta (void) {

    static unsigned long acelIters,            // numero de iteraciones e
                         acelItersTotal,decelIters,decelItersTotal;
```

```
49      static unsigned long mantParIters;          // numero de iteraciones c
50      static int demoNumMovimientos;              // numero de procesos cier
51
52      if ( doEscalonVel )
53          orden = ORDEN_ESCALON_VEL;
54      else if ( doEscalonIq )
55          orden = ORDEN_ESCALON_PAR;
56
57      if ( uGF.bit.ChangeMode || ctrlStatus.dir != prevDir ) {
58          acelIters = 0UL;
59          acelItersTotal = 0UL;
60          decelIters = 0UL;
61          decelItersTotal = 0UL;
62          ctrlStatus.finalizando = False;
63          mantParIters = 0UL;
64          ctrlStatus.mantenerPar = False;
65          ctrlStatus.modoBajaVel = False;
66          demoNumMovimientos = -1;
67      }
68
69      // calculo posicion
70      if ( ctrlStatus.aprendizajeDone && ctrlStatus.posRefSet ) {
71          posPuerta = absVal(encoderGetPosPuerta());
72          if ( ctrlStatus.dir == OPEN )
73              difLongPuerta = absVal(posPuerta - absVal(longPuerta));
74          else
75              difLongPuerta = absVal(posPuerta);
76      }
77
78      // si la puerta se mueve mas alla de la posicion de cierre o de apertu
79      // haya perdido la posicion, y hay que volver a fijar posRef correctam
80      if (    ( absVal(longPuerta) + usrOptions.openFinalPulsos < absVal(pos
81          ||  ( absVal(longPuerta) + usrOptions.closeFinalPulsos < absVal(lo
82          ctrlStatus.posRefSet  = False;
83          ctrlStatus.posChecked = False;
84      }
85
86      // Si el modulo de potencia ha tenido un fallo, reabre lentamente
87      if ( modPotAlarma ) {
88          ctrlStatus.modoBajaVel = True;
89          orden = ORDEN_ABRIR;
90      }
91
92      // INICIO EJECUTAR ORDEN
93      if ( orden == ORDEN_APRENDIZAJE ) {
94
95          // primero cierra del todo y fija la posicion 0. Luego inicia la a
96          if ( !ctrlStatus.posRefSet ) {
```

```
97              // se mueve en sentido de cierre hasta topar con obstaculo
98              if ( !ctrlStatus.obstacle ) {
99                  ctrlStatus.dir = CLOSE;
100                 setVel( usrOptions.aprendizVel );
101             }
102
103             // al topar con obstaculo fija la posicion 0 e inicia la apert
104             else {
105                 ctrlStatus.obstacle = False;
106                 encoderSetPosRef();
107                 ctrlStatus.posRefSet = True;
108                 ctrlStatus.dir = OPEN;    // cambio de direccion, misma ve
109                 setVel( usrOptions.aprendizVel );
110             }
111         }
112
113         // cuando termina de abrir, mide la distancia e inicia cierre
114         else if ( ctrlStatus.obstacle && ctrlStatus.dir == OPEN ) {
115             ctrlStatus.obstacle = False;
116             longPuerta = encoderGetPosPuerta();
117             ctrlStatus.dir = CLOSE;   // cambio de direccion, misma veloci
118             setVel( usrOptions.aprendizVel );
119         }
120
121         // cuando termina de cerrar, comprueba que se ha recorrido la mism
122         else if ( ctrlStatus.obstacle ) { // && dir==CLOSE
123             ctrlStatus.obstacle = False;
124             difLongPuerta = encoderGetPosPuerta();
125             if ( difLongPuerta < MARGEN_PULSOS ) {   // IF posicionActual
126                 ctrlStatus.aprendizajeDone = True;
127                 setLongPuertaInFlash(longPuerta);
128             }
129             uGF.bit.RunMotor = 0;
130         }
131         // else: continua moviendose
132     } // fin orden aprendizaje
133
134
135 /*    else if ( orden == ORDEN_REARMAR ) {
136
137         if ( !ctrlStatus.posRefSet ) {
138             // se mueve en sentido de cierre hasta topar con obstaculo
139             if ( !ctrlStatus.obstacle ) {
140                 ctrlStatus.dir = CLOSE;
141                 setVel( usrOptions.lowVel );
142             }
143
144             // al topar con obstaculo fija la posicion inicial e inicia la
```

```
145             else {
146                 ctrlStatus.obstacle = False;
147                 encoderSetPosRef();
148                 ctrlStatus.posRefSet = True;
149                 ctrlStatus.dir = OPEN;    // cambio de direccion, misma ve
150                 setVel( usrOptions.lowVel );
151             }
152         }
153 
154         // al terminar de abrir comprueba que la distancia recorrida coinc
155         if ( ctrlStatus.obstacle ) { // && dir==OPEN
156             ctrlStatus.obstacle = False;
157             if ( difLongPuerta < MARGEN_PULSOS ) {
158                 encoderSetPosPuerta(longPuerta);
159                 ctrlStatus.posChecked = True;
160             }
161             uGF.bit.RunMotor = 0;
162         }
163         // else: continua moviendose
164     } // fin orden rearmar
165 */
166 
167     else if ( orden == ORDEN_ABRIR ) {
168 
169         // hay que fijar el modo baja velocidad?
170         if (  !ctrlStatus.posChecked || (movingTimeIters<2 && absVal(posPu
171             ctrlStatus.modoBajaVel = True;
172         }
173 
174         // fija la direccion de movimiento
175         ctrlStatus.dir = OPEN;
176 
177         // comprueba si esta finalizando el movimiento
178         if ( ctrlStatus.aprendizajeDone && ctrlStatus.posRefSet && difLong
179             ctrlStatus.finalizando = True;
180         else
181             ctrlStatus.finalizando = False;
182 
183 
184         // deteccion de obstaculo
185         if ( ctrlStatus.obstacle ) {
186             ctrlStatus.obstacle = False;
187 
188             // si no se ha hecho comprobacion de distancia aun, fija posRe
189             if ( !ctrlStatus.posChecked && !ctrlStatus.finalizando ) {
190                 encoderSetPosPuerta(longPuerta);   // seguramente posRef r
191                 ctrlStatus.posRefSet = True;
192                 ctrlStatus.mantenerPar = True;
```

```
193                 IqMantener = calcParCurr( usrOptions.openMantCurr );
194             }
195             // si esta en posicion final, apertura correcta: modo mantener
196             else if ( ctrlStatus.finalizando ){   // si se ha abierto corr
197                 encoderSetPosPuerta(longPuerta);
198                 ctrlStatus.mantenerPar = True;
199                 IqMantener = calcParCurr( usrOptions.openMantCurr );
200                 ctrlStatus.posChecked = True;       // comprobado que posP
201             }
202             // si no, es un obstaculo inesperado
203             else {
204                 uGF.bit.RunMotor = 0;
205             }
206         }
207 
208         // si es apertura lenta
209         else if ( ctrlStatus.modoBajaVel ) {
210             setVel ( usrOptions.lowVel );
211         }
212 
213         // tramo 1
214         else if ( posPuerta < usrOptions.openDist1 ) {
215             setVel( usrOptions.openVel1 );
216         }
217 
218         // tramo 2
219         else if ( posPuerta < usrOptions.openDist2 ) {
220             if ( acelControlada && acelItersTotal < usrOptions.openAcelTim
221                 acelIters++;
222                 acelItersTotal++;
223                 if ( acelIters >= usrOptions.openAcelLoopsStep ) {
224                     incVel( usrOptions.openAcelVelInc );
225                     acelIters = 0;
226                 }
227             }
228             else {  // fija velocidad tramo 2
229                 setVel( usrOptions.openVel2 );
230             }
231         }
232 
233         // tramo 3
234         else { // posPuerta > opendist2
235             if ( acelControlada && decelItersTotal<usrOptions.openDecelTim
236                 decelIters++;
237                 decelItersTotal++;
238                 if ( decelIters >= usrOptions.openDecelLoopsStep ) {
239                     incVel( usrOptions.openDecelVelInc );
240                     decelIters = 0;
```

```
241                 }
242             }
243             else {
244                 setVel( usrOptions.openVel3 );
245             }
246         }
247 
248     } // FIN if(ORDEN ABRIR)
249 
250     else if ( orden == ORDEN_CERRAR ) {
251 
252         // hay que fijar el modo baja velocidad?
253         if (  !ctrlStatus.posChecked || (movingTimeIters<2 && absVal(longP
254             ctrlStatus.modoBajaVel = True;
255         }
256 
257         // fija la direccion de movimiento
258         ctrlStatus.dir = CLOSE;
259 
260         // comprueba si esta finalizando el movimiento
261         if ( ctrlStatus.aprendizajeDone && ctrlStatus.posRefSet && difLong
262             ctrlStatus.finalizando = True;
263         else
264             ctrlStatus.finalizando = False;
265 
266 
267         // deteccion de obstaculo
268         if ( ctrlStatus.obstacle ) {
269             ctrlStatus.obstacle = False;
270 
271             // si aun no se ha comprobado la posicion, fija posRef para co
272             if ( !ctrlStatus.posChecked && !ctrlStatus.finalizando ){
273                 encoderSetPosRef();                 // seguramente posRef
274                 ctrlStatus.posRefSet = True;
275                 ctrlStatus.mantenerPar = True;
276                 IqMantener = calcParCurr( usrOptions.closeMantCurr );
277             }
278             // si esta en posicion final, apertura correcta: modo mantener
279             else if ( ctrlStatus.finalizando ) {    // si se ha abierto co
280                 encoderSetPosRef();                 // fija el 0 como posi
281                 ctrlStatus.mantenerPar = True;
282                 IqMantener = calcParCurr( usrOptions.closeMantCurr );
283                 ctrlStatus.posChecked = True;       // por si aun no se ha
284             }
285             // si no, es un obstaculo inesperado
286             else {
287                 if ( usrOptions.openIfObstacle ) {
288                     orden = ORDEN_ABRIR;
```

```
289                     ctrlStatus.dir = OPEN; // para que recalcule
290                 }
291                 else {
292                     uGF.bit.RunMotor = 0;
293                 }
294             }
295         }
296 
297         // si es cierre lento
298         if ( ctrlStatus.modoBajaVel ) {
299             setVel( usrOptions.lowVel );
300         }
301 
302         // tramo 1
303         else if ( posPuerta > usrOptions.closeDist1 ) {
304             setVel( usrOptions.closeVel1 );
305         }
306 
307         // tramo 2
308         else if ( posPuerta > usrOptions.closeDist2 ) {
309             if ( acelControlada && acelItersTotal < usrOptions.closeAcelT
310                 acelIters++;
311                 acelItersTotal++;
312                 if ( acelIters >= usrOptions.closeAcelLoopsStep ) {
313                     incVel( usrOptions.closeAcelVelInc );
314                     acelIters = 0;
315                 }
316             }
317             else {  // fija velocidad tramo 2
318                 setVel( usrOptions.closeVel2 );
319             }
320         }
321 
322         // tramo 3
323         else if ( posPuerta > usrOptions.closeDist3 ) {
324             if ( acelControlada && decelItersTotal < usrOptions.closeDece
325                 decelIters++;
326                 decelItersTotal++;
327                 if ( decelIters >= usrOptions.closeDecelLoopsStep ) {
328                     incVel( usrOptions.closeDecelVelInc );
329                     decelIters = 0;
330                 }
331             }
332             else {
333                 setVel( usrOptions.closeVel3 );
334             }
335         }
336 
```

```
337         // tramo 4
338         else { // if(posPuerta<closeDist3)
339             setVel( usrOptions.closeVel4 );
340         }
341 
342     } // FIN if(ORDEN CERRAR)
343 
344     else if ( orden == ORDEN_ESCALON_PAR ) {
345         ctrlStatus.mantenerPar = True;
346         if ( movingTimeIters < ESCALON_MANTPAR_ITERATIONS ) {
347             IqMantener = 0;
348         }
349         else {
350             ctrlStatus.dir = CLOSE;
351             IqMantener = calcParCurr( escalonIq );
352         }
353     }
354 
355     else if ( orden == ORDEN_ESCALON_VEL ) {
356         if ( movingTimeIters < ESCALON_MANTPAR_ITERATIONS ) {
357             setVel(0);
358         }
359         else {
360             ctrlStatus.dir = CLOSE;
361             setVel( escalonVel );
362         }
363     }
364 
365     else {  // no deberia entrar aqui
366         uGF.bit.RunMotor = 0;
367     }
368     // FIN EJECUTAR ORDEN
369 
370 
371     if ( usrOptions.modoControl == CONTROL_DEMO && (orden==ORDEN_ABRIR ||
372         // si esta en modo mantener par, cuenta el tiempo
373         if ( !ctrlStatus.mantenerPar ) {
374             mantParIters = 0UL; // pone a 0 el contador de tiempo en el c
375         }
376         else { // mantenerPar==True
377             if ( orden == ORDEN_ABRIR ) {
378                 if ( mantParIters++ >= usrOptions.demoOpenMantParTime ) {
379                     // comprueba si debe detener ya la demostracion
380                     if ( usrOptions.demoMaxNumMovimientos && ++demoNumMov
381                         uGF.bit.RunMotor = 0;
382                     if ( demoNumMovimientos > 9999 )
383                         demoNumMovimientos = 0;
384                     orden = ORDEN_CERRAR;
```

```
385                         ctrlStatus.dir = CLOSE; // para que recalcule
386                     }
387                 }
388                 else { // orden==ORDEN_CERRAR
389                     if ( mantParIters++ >= usrOptions.demoCloseMantParTime )
390                         if ( usrOptions.demoMaxNumMovimientos && ++demoNumMov
391                             uGF.bit.RunMotor = 0;
392                         if ( demoNumMovimientos > 9999 )
393                             demoNumMovimientos = 0;
394                         orden = ORDEN_ABRIR;
395                         ctrlStatus.dir = OPEN; // para que recalcule
396                     }
397                 }
398             }
399         }
400 
401 
402 }
403 
```

```
1  /*
2   * File:   controlPuerta.h
3   * Author: IÒigo Huguet - Fanamoel
4   *
5   * Created on June 12, 2014, 9:40 AM
6   */
7
8  #ifndef CONTROLPUERTA_H
9  #define CONTROLPUERTA_H
10
11
12 void controlPuerta (void);  // Realiza las acciones oportunas para controla
13                             // decidiendo en cada momento la velocidad, dir
14                             // movimiento, etc
15
16 inline void setLongPuerta (unsigned long lng);  // indica al control de la
17                                                 // de la puerta, en numero
18
19
20 #endif  /* CONTROLPUERTA_H */
21
22
```

```c
#include "general.h"
#include "displays.h"

/********************************** DISPLAYS 7 SEGMENTOS ********************


#define delay200ns() { Nop();Nop();Nop();Nop();Nop();Nop();Nop();Nop();Nop

char get7segBits (char chr) {
    switch (chr) {
        case '0': return 0b11111100;
        case '1': return 0b01100000;
        case '2': return 0b11011010;
        case '3': return 0b11110010;
        case '4': return 0b01100110;
        case '5': return 0b10110110;
        case '6': return 0b10111110;
        case '7': return 0b11100000;
        case '8': return 0b11111110;
        case '9': return 0b11110110;
        case 'F': return 0b10001100;
        case '-': return 0b00000010;
        default : return 0;
    }
}
void refreshDisplays ( char const* str, unsigned short posDecPoint ) {
    unsigned long bits;
    unsigned int i;

    DISTROBE = 0;   // flanco negativo: no hace nada. Solo para permitir l
    bits = 0UL;
    bits |= (unsigned long)(get7segBits(*str++) & posDecPoint==1) << 24;
    bits |= (unsigned long)(get7segBits(*str++) & posDecPoint==2) << 16;
    bits |= (unsigned long)(get7segBits(*str++) & posDecPoint==3) << 8;
    bits |= (unsigned long) get7segBits(*str);
    for ( i=0U; i<32U; i++ ) {      // transmite los datos de dcha a izda
        DISDATA = bits>>i & 0x0001; // prepara el dato
        DISCLK = 0;                 // flanco negativo del reloj
        delay200ns();               // espera 200ns
        DISCLK = 1;                 // flanco positivo del reloj: guarda e
        delay200ns();               // espera 200ns
    }
    DISTROBE = 1;   // flanco positivo: pasa los datos del registro serie
    DISON = 0;      // Enciende los displays (esta entrada es negada)
}
```

```
1  /*
2   * File:   displays.h
3   * Author: IÒigo Huguet (Fanamoel)
4   *
5   * Created on June 6, 2014, 8:33 AM
6   */
7
8  #ifndef DISPLAYS_H
9  #define DISPLAYS_H
10
11     void refreshDisplays ( char const* str, unsigned short posDecPoint );
12
13 #endif  /* DISPLAYS_H */
14
15
```

```
#include "encoder.h"
#include "general.h"
#include <qei32.h>

#define PULSOS_POR_VUELTA 4096  ////// cambiar por parametro del menu, per

// variables externas
extern int velocidad;       // variable de PMSM.c, se actualiza en la inte
                                    // velocidad en rp10m (rev/10min)
extern bool recalcularPIW;  // variable de PMSM.c, se actualiza en la inte

// variables privadas
bool primerIndexIgnorado = False;       // Indica si ya se ha ignorado una
bool indexDetectado = False;            // Indica si ya se ha detectado el
unsigned long prevIndexCounter = 0;     // Anterior valor de IndexCounter
signed long signedIndexCounter = 0;     // Valor de IndexCounter leido, pe
long pulsosSum = 0;                     // pulsosSum = signedIdxCounter *
long pulsosVuelta0 = 0;                 // PosiciÛn angular del motor cuar
qeiCounter val,val0,valTmp;             // Variables para pasar valores a


// funciones publicas

inline bool encoderIsIndexDetected (void) {
    return indexDetectado;
}

inline void encoderSetPosRef (void) {
    pulsosVuelta0 = (long)Read32bitQEI1PositionCounter();
    Write32bitQEI1IndexCounter(&val0);
    prevIndexCounter = 0UL;
    signedIndexCounter = 0L;
    pulsosSum = 0L;
}

inline long encoderGetPosPuerta (void) {
    /*long pulsosActual = (long)Read32bitQEI1PositionCounter();
    if ( signedIndexCounter == 0 )
        return pulsosActual - pulsosVuelta0;
    else if ( signedIndexCounter < 0 )   // si signedIdxCntr es negativo,
        return (signedIndexCounter+1)*(long)PULSOS_POR_VUELTA - pulsosVuel
    else
        return (signedIndexCounter-1)*(long)PULSOS_POR_VUELTA + (long)PULS
    // simplificando, todas las expresiones quedan asi: */

    return pulsosSum - pulsosVuelta0 + (long)Read32bitQEI1PositionCounter(
}

```

```
49 inline void encoderSetPosPuerta (long pos) {
50     // actualiza pulsosVuelta0 y el conteo de vueltas, manteniendo el Posi
51     long pulActual = Read32bitQEI1PositionCounter();
52     pulsosSum = 0L;
53     prevIndexCounter = 0UL;
54     signedIndexCounter = 0L;
55     pulsosVuelta0 = pulsosSum + pulActual - pos;
56     if ( pos>0 ) {
57         while ( pulsosVuelta0 < 0L ) {
58             pulsosSum += (long)PULSOS_POR_VUELTA;
59             prevIndexCounter++;
60             signedIndexCounter++;
61             pulsosVuelta0 = pulsosSum + pulActual - pos;
62         }
63     }
64     else {
65         while ( pulsosVuelta0 > (long)(PULSOS_POR_VUELTA-1) ) {
66             pulsosSum -= (long)PULSOS_POR_VUELTA;
67             prevIndexCounter--;
68             signedIndexCounter--;
69             pulsosVuelta0 = pulsosSum + pulActual - pos;
70         }
71     }
72     valTmp.l = prevIndexCounter;
73     Write32bitQEI1IndexCounter(&valTmp);
74 }
75
76 int encoderConfig(void) {
77     /* Configure QEICON, QEIIOC and QEISTAT register */
78
79     Open32bitQEI1( QEI_COUNTER_QEI_MODE &
80                 QEI_GATE_ENABLE &
81                 QEI_COUNT_NEGATIVE &
82                 QEI_INPUT_PRESCALE_8 &
83                 QEI_INDEX_MATCH_NO_EFFECT &
84                 QEI_POS_COUNT_INIT_EVERY_INDEX &
85                 //QEI_POS_COUNT_INIT_No_EFFECT &
86                 QEI_IDLE_CON &
87                 QEI_COUNTER_ENABLE,
88                 QEI_QEA_POL_NON_INVERTED &
89                 QEI_QEB_POL_NON_INVERTED &
90                 QEI_INDX_POL_NON_INVERTED &
91                 QEI_HOM_POL_NON_INVERTED &
92                 QEI_QEA_QEB_NOT_SWAPPED &
93                 QEI_COMPARE_HIGH_OUTPUT_DISABLE &
94                 QEI_DIF_FLTR_PRESCALE_8 &
95                 QEI_DIG_FLTR_ENABLE &
96                 QEI_POS_COUNT_TRIG_DISABLE,
```

```
97
98                  QEI_INDEX_INTERRUPT_ENABLE &
99                  QEI_HOME_INTERRUPT_DISABLE &
100                 QEI_VELO_OVERFLOW_INTERRUPT_DISABLE &
101                 QEI_POS_INIT_INTERRUPT_DISABLE &
102                 QEI_POS_OVERFLOW_INTERRUPT_DISABLE &
103                 QEI_POS_LESS_EQU_INTERRUPT_DISABLE &
104                 QEI_POS_GREAT_EQU_INTERRUPT_DISABLE );
105
106     ConfigInt32bitQEI1(QEI_INT_PRI_4 & QEI_INT_ENABLE); // Habilita las ir
107
108
109     // Conf TMR1
110     TMR1 = 0;           // Reset timer counter
111     T1CONbits.TON = 0;  // Turn off timer 1
112     T1CONbits.TSIDL = 0;// Continue operation during sleep
113     T1CONbits.TGATE = 0;// Gated timer accumulation disabled
114     T1CONbits.TCS = 0;  // use Tcy as source clock
115     T1CONbits.TCKPS = 2;// Tcy / 64 as input clock
116     PR1 = 17280;        // Interrupt period 17280. T = 1/(MIPS/prescaler)*
117     IFS0bits.T1IF = 0;  // Clear timer 1 interrupt flag
118     IEC0bits.T1IE = 1;  // Enable timer 1 interrupts
119     IPC0bits.T1IP = 1;  // Prioridad 1: ponemos prioridad muy baja para re
120     T1CONbits.TON = 1;  // Turn on timer 1
121
122     val.l = (unsigned long)PULSOS_POR_VUELTA; // Se usara este valor para
123     val0.l = 0UL;   // Se usara este valor para reiniciar los contadores
124 }
125
126
127
128 // funciones privadas
129
130 void __attribute__((interrupt, no_auto_psv)) _T1Interrupt (void) {
131     IFS0bits.T1IF = 0;   // Clear timer 1 interrupt flag
132     //////// hacer que se calcule en funcion del numero de pulsos del encc
133     velocidad = __builtin_mulss( (signed)Read32bitQEI1VelocityCounter(), 1
134     recalcularPIW = True;
135
136     // otros usos de la interrupcion no relacionados con el encoder
137     static unsigned int cnt = 0;
138     if ( cnt++ >= 34 ) {// 34 * 0.015s ~= 0.5s
139         SECLED ^= 1;    // cambia estado de SECLED (led para saber que el
140         cnt = 0;
141     }
142     ClrWdt();           // clear Watchdog
143 }
144
```

```
145 // Modifica los contadores de vueltas completas, marca indice como detecta
146 void __attribute__ (( interrupt, no_auto_psv )) _QEI1Interrupt(void) {
147     QEI1STATbits.IDXIRQ = 0;   // Borra flags interrupcion
148     IFS3bits.QEI1IF = 0;
149 
150     // suma o resta una vuelta, segun la direccion de giro
151     if ( Read32bitQEI1IndexCounter() - prevIndexCounter == 1 ) {   // incr
152         prevIndexCounter++;
153         signedIndexCounter++;
154         pulsosSum += (long)PULSOS_POR_VUELTA;   // pulsosSum = signedIdxCr
155     }
156     else {  // decrementando
157         prevIndexCounter--;
158         signedIndexCounter--;
159         pulsosSum -= (long)PULSOS_POR_VUELTA;   // pulsosSum = signedIdxCr
160         Write32bitQEI1PositionCounter( &val );  // escribe 4096 en el Posi
161     }
162 
163     // si es el segundo indice que se detecta (el primero siempre es falso
164     if ( !primerIndexIgnorado ) {
165         primerIndexIgnorado = True;
166         Write32bitQEI1IndexCounter(&val0);
167         prevIndexCounter = 0UL;
168         signedIndexCounter = 0L;
169         pulsosSum = 0L;
170     }
171     else {
172         indexDetectado = True;
173     }
174 
175 }
```

```
/*
 * File:   encoder.h
 * Author: IÒigo Huguet - Fanamoel
 *
 * Funciones para
 *
 * Created on April 28, 2014, 8:06 AM
 */

#ifndef ENCODER_H
#define ENCODER_H

#include "general.h"

int encoderConfig(void);                    // Configura el modulo QEI par
                                            // asi como la interrupcion de

inline bool encoderIsIndexDetected (void);  // Devuelve True si se ha dete
                                            // al menos una vez. De lo con
                                            // no se calcularia bien.

inline void encoderSetPosRef (void);        // Fijar la posicion actual co

inline void encoderSetPosPuerta (long pos); // Forzar un valor concreto co
                                            // Esto se usa, por ejemplo, p
                                            // la longitud total de la pue
                                            // la apertura, eliminando asi

inline long encoderGetPosPuerta (void);     // Devuelve la posiciÛn actual
                                            // respecto a la posicion cero

#endif  /* ENCODER_H */
```

```c
#include "general.h"
#include "entradas.h"

inline unsigned int leerEntrada (tEntrada entr) {
    switch (entr) {
        case ENTR_ABRIR: return ENTRADA_ABRIR;
        case ENTR_CERRAR: return ENTRADA_CERRAR;
        case ENTR_X1: return ENTRADA_X1;
        case ENTR_X2: return ENTRADA_X2;
        case ENTR_X3: return ENTRADA_X3;
        case ENTR_X4: return ENTRADA_X4;
        case ENTR_X5: return ENTRADA_X5;
        default: return 0;
    }
}
```

```
1  /*
2   * File:   entradas.h
3   * Author: IÒigo Huguet - Fanamoel
4   *
5   * Created on June 9, 2014, 9:08 AM
6   */
7
8  #ifndef ENTRADAS_H
9  #define ENTRADAS_H
10
11     typedef enum {ENTR_ABRIR,ENTR_CERRAR,ENTR_X1, ENTR_X2, ENTR_X3, ENTR_X4
12     inline unsigned int leerEntrada (tEntrada entr);    // devuelve el valo
13
14 #endif  /* ENTRADAS_H */
15
16
```

```c
#include <libpic30.h>
#include "general.h"
#include "flash_handler.h"

#define BITS_PER_REGISTER 16

#define FLASH_DIR_ANGLE 0
#define FLASH_DIR_ANGLEWASSAVED 1

#define FLASH_DIR_LENGTH 2
#define FLASH_LEN_LENGTH (sizeof(unsigned long) * 8 / BITS_PER_REGISTER)
#define FLASH_DIR_LENGTHWASSAVED (2 + FLASH_LEN_LENGTH)

#define FLASH_DIR_OPTIONSWERESAVED (FLASH_DIR_LENGTHWASSAVED + 1)
#define FLASH_DIR_OPTIONS (FLASH_DIR_OPTIONSWERESAVED + 1)

_prog_addressT flashAddress = 0; // variable para inicializar direccion de
int __attribute__((space(prog), aligned(_FLASH_PAGE*2))) flashPage[_FLASH_
int flashDataRamCopy[_FLASH_PAGE];  // copia en memoria RAM de la pagina d

// ver flash_handler.h para explicacion de como funciona esto

inline void loadDataFromFlash (void) {  // recupera los datos de la pagina
    if ( !flashAddress )
        _init_prog_address( flashAddress, flashPage );
    _memcpy_p2d16( flashDataRamCopy, flashAddress, _FLASH_PAGE*2 );
    Nop();
    Nop();
}
void flashDoSave (void) {           // graba permanentemente los datos de
    if ( ! flashAddress ) {
        return;                         // si no se han leido datos, no se
    }
    _erase_flash(flashAddress);         // borra la pagina. Necesario para
    while(NVMCONbits.WR){}              // espera a que termine el borrado
    int i;
    for ( i=0; i<_FLASH_PAGE; i+=2 ) {   // guarda todos los datos de la p
        _write_flash_word32( flashAddress+2*i, flashDataRamCopy[i], flashD
        while(NVMCONbits.WR){}
    }
}
void flashSetData ( int dirRelativa, int data ) {   // guarda un dato en u
    if ( !flashAddress ) {
        loadDataFromFlash();
    }
    flashDataRamCopy[dirRelativa] = data;
    flashDoSave();
}
```

```
49 void flashSetDataSkipSave ( int dirRelativa, int data ) {   // Guarda un d
50     if ( !flashAddress ) {                                  // Usar esta f
51         loadDataFromFlash();                                // hacer el gr
52     }                                                       // No obstante
53     flashDataRamCopy[dirRelativa] = data;
54 }
55 void flashSetDataArr ( int dirRelativa, int *data, int length ) {  // guar
56     if ( !flashAddress ) {
57         loadDataFromFlash();
58     }
59     unsigned int i = 0;
60     while ( i < length ) {
61         flashDataRamCopy[dirRelativa+i] = data[i];
62         i++;
63     }
64     flashDoSave();
65 }
66 int flashGetData ( int dirRelativa ) {      // devuelve el dato ubicado en
67     if ( !flashAddress ) {
68         loadDataFromFlash();
69     }
70     return flashDataRamCopy[dirRelativa];
71 }
72 void flashGetDataArr ( int dirRelativa, int *data, int length ) {   // car
73     if ( !flashAddress ) {
74         loadDataFromFlash();
75     }
76     unsigned int i = 0;
77     while ( i < length ) {
78         data[i] = flashDataRamCopy[dirRelativa+i];
79         i++;
80     }
81 }
82 void flashEraseData (void) {
83     if ( !flashAddress )        // si aun no se ha iniciado la direccion d
84         _init_prog_address( flashAddress, flashPage );
85     _erase_flash(flashAddress); // borra
86     while(NVMCONbits.WR){}      // espera a que termine el borrado
87     loadDataFromFlash();        // copia a la ram el contenido actual (deb
88 }
89
90
91 inline int getOffsetFromFlash (void) {          // Recupera de la memoria
92     return (int) flashGetData( FLASH_DIR_ANGLE );
93 }
94 inline bool isOffsetSavedInFlash (void) {
95     return flashGetData( FLASH_DIR_ANGLEWASSAVED ) == 0x1298;
96 }
```

```
97  inline void setOffsetInFlash ( int offsetTheta ) {
98      flashSetData( FLASH_DIR_ANGLE, offsetTheta );
99      flashSetData( FLASH_DIR_ANGLEWASSAVED, 0x1298 );        // Este dato e
100 }                                                           // ya que cero
101 inline void borraOffsetFromFlash (void) {
102     flashSetData( FLASH_DIR_ANGLEWASSAVED, 0 );
103 }
104
105 inline long getLongPuertaFromFlash (void) {    // Recupera de la memoria f
106     int data [FLASH_LEN_LENGTH];
107     long num = 0;
108     unsigned int i;
109     flashGetDataArr( FLASH_DIR_LENGTH, data, FLASH_LEN_LENGTH );
110     for ( i=0; i < FLASH_LEN_LENGTH; i++ ) {    // Como en la flash se gua
111         num = num << BITS_PER_REGISTER;
112         num = num | ( (long)data[i] & 0xFFFF );
113     }
114     return num;
115 }
116 inline bool isLongPuertaSavedInFlash (void) {
117     return flashGetData( FLASH_DIR_LENGTHWASSAVED ) == 0x1298;
118 }
119 inline void setLongPuertaInFlash ( long num ) {
120     int data [FLASH_LEN_LENGTH];
121     int i;
122     for ( i=FLASH_LEN_LENGTH-1; i>=0; i-- ) {
123         data[i] = num & 0xFFFF;
124         num = num >> BITS_PER_REGISTER;
125     }
126     flashSetDataArr( FLASH_DIR_LENGTH, data, FLASH_LEN_LENGTH );
127     flashSetData( FLASH_DIR_LENGTHWASSAVED, 0x1298 );
128 }
129 inline void borraLongPuertaFromFlash (void) {
130     flashSetData( FLASH_DIR_LENGTHWASSAVED, 0 );
131 }
132
133 inline int getOptionFromFlash (int pos) {
134     return flashGetData( FLASH_DIR_OPTIONS + pos );
135 }
136 inline bool areOptionsSavedInFlash (void) {
137     return flashGetData( FLASH_DIR_OPTIONSWERESAVED ) == 0x1298;
138 }
139 inline void setOptionInFlash ( int pos, int data ) {
140     flashSetData( FLASH_DIR_OPTIONS+pos, (int)(data) );
141 }
142 inline void setOptionInFlashBuffer (int dirRel, int data) {
143     flashSetDataSkipSave( FLASH_DIR_OPTIONS + dirRel, data );
144 }
```

```
145 inline void saveOptionsBufferInFlash (void) {
146     flashSetDataSkipSave( FLASH_DIR_OPTIONSWERESAVED, 0x1298 ); // Guarda
147     flashDoSave();              // Ejecuta de forma efectiva el guardado e
148 }
149
```

```
/*
 * File:   flash_handler.h
 * Author: IÒigo Huguet (fanamoel)
 *
 * Created on February 24, 2014, 10:19 AM
 */

#ifndef FLASH_HANDLER_H
#define FLASH_HANDLER_H

/*
 * Para minimizar la interacciÛn con la memoria flash, que es lenta y se c
 * existe una copia creada de los datos de toda la pagina (1024palabras) e
 * RAM.
 *
 * La lectura de datos desde otras partes del programa debe hacerse a esta
 * memoria RAM. Para ello se facilitan las funciones especificas de mas ab
 * en principio, EN EL CASO DE LA LECTURA, no es obligatorio usarlas).
 *
 * La escritura de datos debe hacerse unicamente a traves de las funciones
 * El motivo es que para escribir en flash primero hay que borrar toda la
 * (caracteristica de las memorias flash). Por eso, lo que se hace es borr
 * y guardar los datos que hay en la copia de la memoria RAM. Por tanto, l
 * RAM siempre debe estar sincronizada con los datos reales en memoria fla
 * consigue con las funciones de abajo.
 */


// dato offset en la flash
inline int getOffsetFromFlash (void);
inline bool isOffsetSavedInFlash (void);
inline void setOffsetInFlash (int offsetTheta);
inline void borraOffsetFromFlash (void);

// dato longitud de puerta
inline long getLongPuertaFromFlash (void);
inline bool isLongPuertaSavedInFlash (void);
inline void setLongPuertaInFlash (long num);
inline void borraLongPuertaFromFlash (void);

// datos de opciones de usuario
inline int getOptionFromFlash (int pos);
inline bool areOptionsSavedInFlash (void);
inline void setOptionInFlash ( int pos, int data );
inline void setOptionInFlashBuffer (int dirRel, int data);
inline void saveOptionsBufferInFlash (void);

 #endif /* FLASH_HANDLER_H */
```

```
49
50
```

```
1  /*********************************************************************
2   *      Code Description                                             *
3   *                                                                   *
4   *  Initialization for dsPIC's ports                                 *
5   *                                                                   *
6   *********************************************************************/
7
8  #include "general.h"
9
10 void SetupPorts( void )
11 {
12
13     // ============= Port A ==============
14     LATA  = 0x0000;             // 5432 1098 7654 3210
15     TRISA = 0x1903;             // 0001 1001 0000 0011  RA12-DCBUS; RA1-IA
16     ANSELA = 0X1003;            // 0001 0000 0000 0011  ANSA12 - DCBUS; AN
17     // ============= Port B ==============
18     LATB  = 0x0000;             // 5432 1098 7654 3210
19     TRISB = 0x010F;             // 0000 0001 0000 1111  RB15:RB10 - PWM1H:
20     ANSELB = 0X000F;            // 0000 0000 0000 1111  ANSB3 - IBUS- ANSE
21     // ============= Port C ==============
22     LATC  = 0x0000;             // 5432 1098 7654 3210
23     TRISC = 0x004f;             // 0000 0000 0100 1111
24     ANSELC = 0x0007;            // 0000 0000 0000 0111
25
26     // ============= Port D ==============
27     LATD  = 0x0000;             // 5432 1098 7654 3210
28     TRISD = 0x0000;          // 0000 0000 0000 0000
29     // ============= Port E ==============
30     LATE  = 0x0000;             // 5432 1098 7654 3210
31     TRISE = 0x3000;             // 0011 0000 0000 0000  RE12-OD, RE13-CD ;
32     ANSELE = 0X0000;            // 0000 0000 0000 0000
33     // ============= Port F ==============
34     LATF  = 0x0000;             // 5432 1098 7654 3210
35     TRISF = 0x0001;             // 0000 0000 0000 0001  RF0-IDX
36     // ============= Port G ==============
37     LATG  = 0x0000;             // 5432 1098 7654 3210
38     TRISG = 0x03C0;             // 0000 0011 1100 0000
39
40
41     __builtin_write_OSCCONL(OSCCON & (~(1<<6))); // clear bit 6 (OSCON.bit
42
43     // Peripheral Pin Select (PPS) - ver apdo 11.4 datasheet
44     RPINR14 = 0x3618;   // RPINR14 = mapeado perifericos: qeiB y qeiA
45                         // qeiA mapeado a pin RPI24 ( mask: 0x18 = 0011000
46                         // qeiB mapeado a pin RP54 ( mask: 0x36 = 0110110
47                         // RPINR14 = 0x7879 = 0b 00110110 00011000;
48
```

```
49      RPINR15 = 0x0060;   // RPINR15 = mapeado perifericos: home y index (de
50                          // qeiIdx mapeado a pin RPI96 ( mask: 0x60 = 11000
51                          // home mapeado a Vss ( mask: 0x00 )
52                          // RPINR15 = 0x0060 = 0b 00000000 01100000;
53
54      __builtin_write_OSCCONL(OSCCON | (1<<6));    // Set bit 6
55
56      return;
57  }
58
```

```c
#include "general.h"
#include "user_options_input.h"
#include "park.h"
#include "obstaculo.h"

// constantes
#define OBSTACLE_VEL_ITERATIONS  ((unsigned long)((double)OBSTACLE_VEL_DET
#define OBSTACLE_VEL_ITERATIONS_FINAL ((unsigned long)((double)OBSTACLE_VE
#define OBSTACLE_CURR_ITERATIONS 25
#define OBSTACLE_ITERATIONS_DEADTIME ((unsigned long)((double)OBSTACLE_DET

// variables externas que utiliza este archivo
extern int prevDir, VelReq, velocidad;
extern tCtrlStatus ctrlStatus;
extern tUsrOptions usrOptions;
extern tUGF uGF;
extern tParkParm ParkParm;
extern unsigned long movingTimeIters;

// variables
unsigned int openObstCurr,      // valores lÌmite de corriente en detecci
        closeObstCurr;              // unidades: porcentaje en SFRAC16 (fo
                                    // rango: -32768 a 32767 (32767 equiva

void setObstacCurrLims (int openCurr, int closeCurr) {
    openObstCurr = openCurr;
    closeObstCurr = closeCurr;
}

void detectaObstaculo (void) {

    int obstacVelLim, obstacCurrLim;            // valores limite para la
    unsigned long obstacVelItersLim;            // maximo numero de iterac
    static unsigned long obstacVelIters;        // numero de iteraciones c
    static unsigned int obstacCurrIters;        // numero de iteraciones c
    static unsigned long obstacDeadTimeIters;   // numero de iteraciones p

    if ( uGF.bit.ChangeMode || ctrlStatus.dir != prevDir ) {
        obstacDeadTimeIters = 0UL;
        obstacVelIters = 0UL;
        obstacCurrIters = 0U;
    }

    // Si ha superado el tiempo maximo de movimiento da obstaculo
    if ( !ctrlStatus.finalizando ) { // si esta finalizando ya no cuenta,
        if ( ctrlStatus.modoBajaVel ) {
            if ( movingTimeIters >= usrOptions.lowVelMaxTime )
                ctrlStatus.obstacle = True;
```

```
49          }
50          else if ( ctrlStatus.dir == OPEN ) {
51              if ( movingTimeIters >= usrOptions.openMaxTime )
52                  ctrlStatus.obstacle = True;
53          }
54          else { // dir==CLOSE
55              if ( movingTimeIters >= usrOptions.closeMaxTime )
56                  ctrlStatus.obstacle = True;
57          }
58      }
59
60      // recalcula limites de deteccion
61      if ( ctrlStatus.finalizando ) {
62          obstacVelLim = VelReq / 4;
63          obstacCurrLim = ctrlStatus.dir == OPEN ? usrOptions.openFinCurr :
64          obstacVelItersLim = OBSTACLE_VEL_ITERATIONS_FINAL;
65      }
66      else {
67          obstacVelLim = VelReq / 3;
68          obstacCurrLim = ctrlStatus.dir == OPEN ? openObstCurr : closeObstC
69          obstacVelItersLim = OBSTACLE_VEL_ITERATIONS;
70      }
71
72      // deteccion por velocidad
73      if ( VelReq>0 ? velocidad<obstacVelLim : velocidad>obstacVelLim ) {
74          if ( obstacVelIters++ > obstacVelItersLim  ) {
75              ctrlStatus.obstacle = True;
76              obstacVelIters = 0UL;
77          }
78      }
79      else {
80          obstacVelIters = 0UL;
81      }
82
83      // comprueba si ya ha pasado el deadtime
84      if ( obstacDeadTimeIters < OBSTACLE_ITERATIONS_DEADTIME )
85          obstacDeadTimeIters++;
86
87      // deteccion por corriente
88      if ( obstacDeadTimeIters == OBSTACLE_ITERATIONS_DEADTIME ) {    // if
89
90          // comprueba si se supera el limite
91          if ( obstacCurrLim>0 ? ParkParm.qIq>obstacCurrLim : ParkParm.qIq<o
92              if ( ctrlStatus.finalizando ) { // si esta finalizando, compru
93                  if ( velocidad < obstacVelLim )
94                      obstacCurrIters++;
95                  else
96                      obstacCurrIters = 0UL;
```

```
97              }
98              else {
99                  obstacCurrIters++;
100             }
101             if ( obstacCurrIters >= OBSTACLE_CURR_ITERATIONS )
102                 ctrlStatus.obstacle = True;
103         }
104         else {
105             obstacCurrIters = 0UL;
106         }
107     }
108
109 }
110
```

```
/*
 * File:   obstaculo.h
 * Author: IÒigo Huguet - Fanamoel
 *
 * Created on June 12, 2014, 9:38 AM
 */

#ifndef OBSTACULO_H
#define OBSTACULO_H

void detectaObstaculo (void);   // pone ctrlStatus.obstacle a True si detec
                                // se considera que hay obstaculo cuando ha
                                // o velocidad demasiado baja.


#endif  /* OBSTACULO_H */

```

```
/*****************************************************************************
 *
 * Desarrollado para el siguiente hardware:
 *      Microcontrolador: dsPIC33EP256MC506 (microchip)
 *      PCB: operador sincrono Fanamoel, version 1
 * Entorno de desarrollo: MPLABX v2.00 (microchip.com)
 * Compilador: XC16 v1.21 (microchip.com)
 * CÛmo compilar: desde MPLABX (o MPLAB8, reconfigurando el proyecto un po
 * DescripciÛn del producto:
 *      El software est· destinado a controlar un operador de puerta de as
 *      que maneje el movimiento de un motor elÈctrico. El motor a control
 *      en principio, un motor sÌncrono de imanes permanentes con encoder
 *      cuadratura, aunque no se descartan otros tipos en el futuro.
 *
 *      Para poder controlar el motor se implementa un control vectorial (
 *      oriented control). Ver documentaciÛn adjunta para m·s informaciÛn.
 *      disponible dicha documentaciÛn, ver la nota de aplicaciÛn AN1078 d
 *      que explica este tipo de control (adem·s, este software es una mod
 *      programa facilitado junto a dicha nota de aplicaciÛn).
 *
 *      Para un correcto funcionamiento del control vectorial es necesario
 *      offset que existe entre el ·ngulo elÈctrico y el ·ngulo mec·nico m
 *      Para conocer la posiciÛn de la puerta en cada momento, se calcula
 *      en funciÛn del movimiento realizado, medido con el encoder, desde
 *      de cierre, que es la posiciÛn de referencia.
 *
 *      El manejo de la puerta incluye las siguientes caracterÌsticas:
 *          - Proceso para c·lculo del offset entre los ·ngulos mec·nicos
 *          El resultado se guarda en memoria flash para no tener que repe
 *          - Proceso para c·lculo de la longitud total de la puerta, en n
 *          pulsos del encoder. El resultado se guarda en memoria flash pa
 *          que repetirlo.
 *          - Control de la velocidad seg˙n curvas de velocidad en base a
 *          de la puerta
 *          - Muchos par·metros son ajustables desde un men˙, que se contr
 *          pulsadores y se muestra en 4 displays 7 segmentos.
 *          - DetecciÛn r·pida de obstrucciones de la puerta
 *
 *      El comportamiento descrito es, a peticiÛn del cliente, lo m·s pare
 *      al del operador EV3200 de Emerson
 *
 *      Gran parte de este software se desarrollÛ como proyecto fin de car
 *      que la memoria de dicho proyecto deberÌa estar disponible en caso
 *      explicaciones m·s detalladas.
 *
 *
 * Breve explicacion del codigo:
 *      - En main se detecta la pulsacion de botones y recepcion de seÒale
```

```
 *      Si esas acciones deben arrancar el motor, calcula valores necesari
 *      movimiento y activa el control vectorial. Una vez activado, main c
 *      pulsacion de botones y recepcion de seÒales de entrada para saber
 *      - El control vectorial se ejecuta en la funcion _AD1Interrupt, que
 *      del PWM. Por tanto, se ejecuta 20000 veces por segundo. _AD1Interr
 *      orden las distintas funciones necesarias para el control (medicion
 *      medicion del angulo, tansformadas de Clarke y Park, controladores
 *      - El codigo aÒadido por Fanamoel se ejecuta sobre todo desde la fu
 *      y es el codigo de los archivos obstaculo.c y controlPuerta.c
 *      - Otras partes del codigo aÒadido son:
 *          + botones y displays: display.c, entradas.c, botonera.c
 *          + memoria flash: flash_handler.c
 *          + menu de opciones: user_options_input.c
 *          + medicion de posicion y velocidad con encoder: encoder.c
 *          + constantes a configurar: UserParms.h
 *
******************************************************************************

/*************** GLOBAL DEFINITIONS ***********/

#define INITIALIZE

// includes Microchip
#include "general.h"
#include "Parms.h"
#include "SVGen.h"
#include "ReadADC.h"
#include "MeasCurr.h"
#include "Control.h"
#include "PI.h"
#include "Park.h"
#include "UserParms.h"  // archivo de configuracion de parametros, etc
#include "smcpos.h"
#include "FdWeak.h"

// includes Fanamoel
#include "encoder.h"
#include "flash_handler.h"
#include "user_options_input.h"
#include "displays.h"
#include "botonera.h"
#include "entradas.h"
#include "obstaculo.h"
#include "controlPuerta.h"
// CONFIGURACION DE LOS PUERTOS: initdspic.c
```

```
97  /*************************************************************************
98  /* Configuration bits
99  /*************************************************************************
100 _FPOR(ALTI2C1_OFF & ALTI2C2_OFF);                // ?
101 _FWDT(PLLKEN_ON & FWDTEN_OFF);                  // watchdog /////// activa
102 _FOSCSEL(FNOSC_FRC & IESO_OFF & PWMLOCK_OFF);   // seleccion de oscilador
103 _FGS(GWRP_OFF & GCP_OFF);                       // ?
104
105 _FICD(ICS_PGD2 & JTAGEN_OFF);                  // canal para programador:
106 _FOSC(FCKSM_CSECMD & POSCMD_XT);        // configuracion del oscilador pri
107 //_FOSC(FCKSM_CSECMD & OSCIOFNC_OFF & POSCMD_NONE); // configuracion del o
108                                                  // para oscilador inte
109
110 /*************** END OF GLOBAL DEFINITIONS ***********/
111
112
113
114 /************************** VARIABLES MICROCHIP ************************
115
116 int VelReq = 0;             // Velocidad de referencia, en rp10m (rev/10mi
117 SMC smc1 = SMC_DEFAULTS;    // Estimador para modo sensorless (en principi
118 unsigned long Startup_Ramp = 0; /* Start up ramp in open loop. This variab
119                                 is incremented in CalculateParkAngle()
120                                 subroutine, and it is assigned to
121                                 ParkParm.qAngle as follows:
122                                 ParkParm.qAngle += (int)(Startup_Ramp >> 1
123 unsigned int Startup_Lock = 0;  /* This is a counter that is incremented i
124                                 CalculateParkAngle() every time it is call
125                                 Once this counter has a value of LOCK_TIME
126                                 then theta will start increasing moving th
127                                 motor in open loop. */
128 unsigned int  trans_counter = 0;
129 tUGF uGF;
130 tPIParm     PIParmD;    // Structure definition for Flux component of curr
131 tPIParm     PIParmQ;    // Structure definition for Torque component of cu
132 tPIParm     PIParmW;    // Structure definition for Speed, or Omega
133 tReadADCParm ReadADCParm;   // Struct used to read ADC values.
134
135 // Speed Calculation Variables
136 WORD iADCisrCnt = 0;    // This Counter is used as a timeout for polling t
137                         // in main() subroutine. It will be reset to zero
138                         // dButPolLoopCnt defined in UserParms.h
139 SFRAC16 PrevTheta = 0;  // Previous theta which is then substracted from T
140                         // delta theta. This delta will be accumulated in
141                         // after a number of accumulations Omega is calcul
142 SFRAC16 AccumTheta = 0; // Accumulates delta theta over a number of times
143 WORD AccumThetaCnt = 0; // Counter used to calculate motor speed. Is incre
144                         // in SMC_Position_Estimation() subroutine, and ac
```

```
145                         // delta Theta. After N number of accumulations, C
146                         // calculated. This N is diIrpPerCalc which is def
147                         // UserParms.h.
148
149 // Vd and Vq vector limitation variables
150 SFRAC16 qVdSquared = 0; // This variable is used to know what is left from
151                         // in order to have maximum output PWM without sat
152                         // done before executing Iq control loop at the en
153 SFRAC16 DCbus = 0;      // DC Bus measured continuously and stored in this
154                         // while motor is running. Will be compared with T
155                         // and Vd and Vq will be compensated depending on
156                         // between DCbus and TargetDCbus
157 SFRAC16 TargetDCbus = 0;// DC Bus is measured before running motor and sto
158                         // variable. Any variation on DC bus will be compa
159                         // and compensated linearly.
160 SFRAC16 Theta_error = 0;// This value is used to transition from open loop
161                         // At the end of open loop ramp, there is a differ
162                         // forced angle and estimated angle. This differen
163                         // Theta_error, and added to estimated theta (smc1
164                         // effective angle used for commutating the motor
165                         // the end of open loop, and at the begining of cl
166                         // This Theta_error is then substracted from estim
167                         // gradually in increments of 0.05 degrees until t
168                         // than 0.05 degrees.
169 // otros
170 tParkParm ParkParm;
171
172
173
174 /****************** VARIABLES Y FUNCIONES/MACROS FANAMOEL ***************
175
176 // macros y funciones inline
177 #define clearCtrlStatus() { asm("CLR _ctrlStatus"); }
178 inline long absVal(long num) { if(num>0) return num; else return -num; }
179 inline int signo(int num) { if(num<0) return -1; else if(num>0) return 1;
180
181 // variables para conocer angulo, velocidad, posicion, etc
182 SFRAC16 angulo, offsetAngulo=0; // medicion del angulo del rotor, para pas
183                                     // unidades: radianes * (32768/PI), pa
184                                     // rango de valores: -32768 a 32767 (r
185                                     // nota: es angulo electrico, no mecan
186                                     // nota2: offsetAngulo es el desfase e
187 int velocidad;                  // medicion de la velocidad para la realim
188                                     // unidades: rp10m (rev/10min)
189                                     // rango de valores: -MaxRp10mMotor a
190                                     // nota: es velocidad mecanica, no ele
191 long longPuerta;                // longitud total de la puerta, para el co
192                                     // unidades: numero de pulsos del enco
```

```
193                                     // rango: capacidad de la variable SIG
194
195 // otras variables
196 tCtrlStatus ctrlStatus;         // struct con los flags que indican el est
197 tCodigoOrden orden;              // variable tipo enum que indica que accio
198 extern tUsrOptions usrOptions;  // struct con todas las opciones de usuari
199 bool recalcularPIW = True;      // determina si debe recalcularse el PI de
200 unsigned int motorMaxCurr,      // valores lÌmite de la referencia de la
201         frenoMaxCurr;               // unidades: porcentaje en SFRAC16 (fo
202                                     // rango: -32768 a 32767 (32767 equiva
203 int prevDir = INDEF;            // variable que se usa para saber si se ac
204                                     // posibles valores: OPEN, CLOSE, INDE
205 int IqMantener;                 // valor de la corriente que se debe mante
206                                     // unidades: porcentaje en SFRAC16 (fo
207                                     // rango: -32768 a 32767 (32767 equiva
208 unsigned long movingTimeIters;  // numero de interrupciones PWM que ha dur
209
210 // data monitor
211 #ifdef DM_ON    // parametros y variables del data monitor
212     unsigned long DM_delay = DM_DELAY;  // cada cuantas interrupciones PWM
213     unsigned long DM_delayCnt;          // contador del numero de interrup
214     unsigned int DM_arrPos;             // posicion de la siguiente muestr
215     bool DM_arrOverflow;                // True si se han llenado los buff
216     int DM_var1 = &DM_VAR1;             // direcciones de memoria de las v
217     int DM_var2 = &DM_VAR2;
218     int DM_var3 = &DM_VAR3;
219     int DM_var4 = &DM_VAR4;
220     int DM_var5 = &DM_VAR5;
221     int DM_var6 = &DM_VAR6;
222     int DM_var7 = &DM_VAR7;
223     #if DM_NUMVARS >= 1                 // buffers (arrays)
224         int DM_buffer1[DM_ARRAYSIZE] __attribute__((far));
225     #endif
226     #if DM_NUMVARS >= 2
227         int DM_buffer2[DM_ARRAYSIZE] __attribute__((far));
228     #endif
229     #if DM_NUMVARS >= 3
230         int DM_buffer3[DM_ARRAYSIZE] __attribute__((far));
231     #endif
232     #if DM_NUMVARS >= 4
233         int DM_buffer4[DM_ARRAYSIZE] __attribute__((far));
234     #endif
235     #if DM_NUMVARS >= 5
236         int DM_buffer5[DM_ARRAYSIZE] __attribute__((far));
237     #endif
238     #if DM_NUMVARS >= 6
239         int DM_buffer6[DM_ARRAYSIZE] __attribute__((far));
240     #endif
```

```
241     #if DM_NUMVARS >= 7
242         int DM_buffer7[DM_ARRAYSIZE] __attribute__((far));
243     #endif
244 #endif
245 
246 
247 
248 int corrienteMax = 2000;    //////// mover, hacer que se calcule segun los
249 
250 
251 
252 
253 /************** START OF MAIN FUNCTION ***************/
254 int main ( void )
255 {
256   //The settings below set up the oscillator and PLL for x MIPS as
257     //follows:
258     //                  Fin  * M
259     // Fosc       =     ---------
260     //                   N2 * N1
261     //
262     // Fin        = 7.37 MHz (Internal Oscillator) or 8MHZ ( Crystal Oscil
263     // Fosc       = x MHz
264     // Fcy        = Fosc/2 =
265 
266     /******************* Clock definitions ********************************
267     // 70 MIPS (70.015 Mhz) (i.e 7.37 * (76/4))
268 /*
269     PLLFBD =  74;               // M = 76    74
270     CLKDIVbits.PLLPOST = 0;     // N2 = (2 x (PLLPOST + 1)) = 2   0
271     CLKDIVbits.PLLPRE = 0;      // N1 = (PLLPRE + 2) = 2        0
272 */
273     // 70 MIPS (70.00 Mhz)
274     PLLFBD = 68;                // M = 70
275     CLKDIVbits.PLLPOST = 0;     // N1=2
276     CLKDIVbits.PLLPRE = 0;      // N2=2
277 
278 
279     // Initiate Clock Switch to Primary Oscillator with PLL (NOSC=0b011)
280     __builtin_write_OSCCONH(0x03);
281     __builtin_write_OSCCONL(0x01);
282     while(OSCCONbits.COSC != 0x03);
283     while(OSCCONbits.LOCK != 1);
284 
285 
286     // Initiate Clock Switch to FRC with PLL (NOSC=0b001)
287     /////// para probar oscilador interno (descomentar tambien en los bits
288 /*    __builtin_write_OSCCONH(0x01);
```

```
289     __builtin_write_OSCCONL(0x01);
290     while(OSCCONbits.COSC != 0b001);
291     while(OSCCONbits.LOCK != 1);
292 */
293
294     SetupPorts();               // configura los puertos
295     encoderConfig();            // configura el encoder (QEI, interrupcio
296     refreshDisplays("    ",4);  // deja en blanco la pantalla
297     getUserOptionsValues();     // recupera de la flash las opciones defi
298     clearCtrlStatus();          // pone a 0 todos los flags de ctrlStatus
299     SMCInit(&smc1);
300     SetupControlParameters();
301     FWInit();
302     uGF.Word = 0;               // clear flags
303
304     #ifdef TORQUEMODE
305     uGF.bit.EnTorqueMod = 1;
306     #endif
307
308     #ifdef ENVOLTRIPPLE
309     uGF.bit.EnVoltRipCo = 1;
310     uGF.bit.ADCH0_PotOrVdc = 0;    // Selects POT as CH0 Input
311     #endif
312
313     while(1) {
314
315         IEC0bits.AD1IE = 0; // Make sure ADC does not generate interrupts
316
317         // DETECCION DE BOTON PULSADO O SE—AL RECIBIDA
318         tPulsacion btnPulsado;
319         tEntrada entrRecibida;
320
321         // espera a que se suelte el boton, por si estaba pulsado, a no s
322         // se este manteniendo pulsada una flecha al editar un valor del
323         while (1) {
324                     btnPulsado = leerBotonPulsado();
325                     if ( !btnPulsado.btn )
326                         break;
327                     else if ( menuEditandoOpcion() && (btnPulsado.btn==BT
328                         menuIncVel();   // un incrementa la velocidad a l
329                         break;
330                     }
331         }
332
333         // espera a que se pulse un boton. Si es modo control externo tam
334         while (1) {
335                     // si esta en modo control externo, y no esta en el m
336                     if ( usrOptions.modoControl==CONTROL_EXTERNO && !ctrl
```

```
337                         if ( leerEntrada(ENTR_ABRIR) && !leerEntrada(ENTR
338                             entrRecibida = ENTR_ABRIR;
339                             orden = ORDEN_ABRIR;
340                             uGF.bit.RunMotor = 1;
341                             break;
342                         }
343                         else if ( leerEntrada(ENTR_CERRAR) && !leerEntrac
344                             entrRecibida = ENTR_CERRAR;
345                             orden = ORDEN_CERRAR;
346                             uGF.bit.RunMotor = 1;
347                             break;
348                         }
349                     }
350 
351                     // comprueba la pulsacion de botones
352                     btnPulsado = leerBotonPulsado();
353                     if ( btnPulsado.btn )  // comprueba si se ha pulsado
354                         break;
355                     menuResetVel(); // si no se esta pulsando ningun boto
356         }
357 
358         // determina la accion a realizar en funcion de la tecla pulsada
359         if ( !ctrlStatus.enMenu ) { // si esta en el menu, delega la acci
360                     ctrlStatus.enMenu = accionMenu(btnPulsado.btn);
361         }
362         else {
363                     if ( btnPulsado.btn == BTN_MODE ) {        // btn moc
364                         ctrlStatus.enMenu = True;
365                         entrarMenu();   // inicia el menu
366                     }
367                     else if ( btnPulsado.btn == BTN_RUN && usrOptions.moc
368                         if ( esperarPulsLarga(btnPulsado) ) // espera a v
369                             orden = ORDEN_CERRAR;
370                         else                                    // si no es p
371                             orden = ORDEN_ABRIR;
372                         uGF.bit.RunMotor = 1;
373                     }
374         }
375 
376         // si como resultado de la accion no se ha ordenado arrancar, rei
377         if ( !uGF.bit.RunMotor )
378             continue;
379 
380         // Si se ha ordenado arrancar, ejecuta el codigo que viene a cont
381         // NOTA: todos los parametros y comprobaciones que hay a continua
382         // motor, en prevision de que desde la ultima vez que se movio se
383         // volver a calcular las distancias de las curvas de apertura y c
384         // de usuario o vuelto a hacer el aprendizaje
```

```
385
386
387         /////// quitar (esto hace que inicie con el offset y la longitud
388         setOffsetInFlash(3000); //setOffsetInFlash( 0x040C );
389         setLongPuertaInFlash( 0xFFFEEA23 );
390         ////////
391
392
393         // determina el estado actual (se ha hecho ya el aprendizaje? se
394         ctrlStatus.indexDetectado = encoderIsIndexDetected();   // compru
395         if ( isOffsetSavedInFlash() ) {         // comprueba si esta el c
396             offsetAngulo = getOffsetFromFlash();
397             ctrlStatus.offsetCalculado = True;
398         }
399         else { // si no, borra flags correspondientes
400             ctrlStatus.offsetCalculado = False;
401         }
402         if ( isLongPuertaSavedInFlash() ) {     // comprueba si se habia
403             longPuerta = getLongPuertaFromFlash();
404             ctrlStatus.aprendizajeDone = True;
405         }
406         else { // si no, borra flags correspondientes
407             ctrlStatus.aprendizajeDone = False;
408             ctrlStatus.posRefSet = False;
409         }
410         if ( !ctrlStatus.posRefSet )            // si no se ha fijado pos
411             ctrlStatus.posChecked = False;
412
413
414         // Determina si la accion a realizar es alguna accion especial (c
415         if ( !ctrlStatus.offsetCalculado ) {      ////// cambiar por " IF
416             orden = ORDEN_CALCULAR_OFFSET;
417         }
418         else if ( usrOptions.doAprendizaje ) {   // F4.05 del menu == 1?
419             orden = ORDEN_APRENDIZAJE;
420             if ( !ctrlStatus.offsetCalculado || !ctrlStatus.indexDetectad
421                 ////// error, no se puede
422             }
423         }
424 /*        else if ( !ctrlStatus.posChecked ) {      ////// intentar que e
425             orden = ORDEN_REARMAR;
426             if ( !ctrlStatus.aprendizajeDone ) {
427                 ////// error, no se puede
428             }
429         }
430         if ( !ctrlStatus.indexDetectado ) {  ///// intentar que esto se h
431             orden = ORDEN_DETECTAR_INDEX;
432         }*/
```

```
433
434
435         // calcula las corrientes de deteccion de obstaculo y las corrien
436         /////////// calcular corrienteMax segun parametro del menu
437         setObstacCurrLims( usrOptions.openObstacleSensitivity * corriente
438                            usrOptions.closeObstacleSensitivity * corr
439         motorMaxCurr = usrOptions.corrLimMotorScale * corrienteMax;
440         frenoMaxCurr = usrOptions.corrLimFrenoScale * corrienteMax;
441
442         // Fija los parametros de los PIs (el de velocidad varia para vel
443         PIParmQ.qKp = usrOptions.ikp;
444         PIParmQ.qKi = usrOptions.iki;
445         PIParmD.qKp = usrOptions.ikp;
446         PIParmD.qKi = usrOptions.iki;
447
448         // calcula los parametros de la curva en funcion de la longitud d
449         // y de los porcentajes definidos en las opciones de usuario, etc
450         if ( ctrlStatus.aprendizajeDone ) {
451
452             // calcula las distancias
453             usrOptions.openDist1  = absVal(usrOptions.openDist1scale * lo
454             usrOptions.openDist2  = absVal(usrOptions.openDist2scale * lo
455
456             usrOptions.closeDist1 = absVal(longPuerta - usrOptions.closeD
457             usrOptions.closeDist2 = absVal(longPuerta - usrOptions.closeD
458             usrOptions.closeDist3 = absVal(usrOptions.closeDist3scale * l
459
460             // calcula la deteccion de obstaculos en tramos de finalizaci
461             usrOptions.openFinCurr  = usrOptions.openCorrFinScale  * corr
462             usrOptions.closeFinCurr = usrOptions.closeCorrFinScale * corr
463
464             // calcula el par que hay que mantener tras finalizacion de m
465             usrOptions.openMantCurr  = usrOptions.openCorrMantenerScale
466             usrOptions.closeMantCurr = usrOptions.closeCorrMantenerScale
467
468             // calcula las aceleraciones y deceleraciones para la apertur
469             usrOptions.openAcelLoopsStep = usrOptions.openAcelTime / absV
470             if ( usrOptions.openAcelLoopsStep == 0 ) usrOptions.openAcelL
471             usrOptions.openAcelVelInc = (long)(usrOptions.openVel2 - usrO
472             if ( usrOptions.openAcelVelInc == 0 ) usrOptions.openAcelVelI
473
474             usrOptions.openDecelLoopsStep = usrOptions.openDecelTime / ab
475             if ( usrOptions.openDecelLoopsStep == 0 ) usrOptions.openDece
476             usrOptions.openDecelVelInc = (long)(usrOptions.openVel3 - usr
477             if ( usrOptions.openDecelVelInc == 0 ) usrOptions.openDecelVe
478
479             // calcula las aceleraciones y deceleraciones para el cierre
480             usrOptions.closeAcelLoopsStep = usrOptions.closeAcelTime / ab
```

```
481             if ( usrOptions.closeAcelLoopsStep == 0 ) usrOptions.closeAce
482             usrOptions.closeAcelVelInc = (long)(usrOptions.closeVel2 - us
483             if ( usrOptions.closeAcelVelInc == 0 ) usrOptions.closeAcelVe
484 
485             usrOptions.closeDecelLoopsStep = usrOptions.closeDecelTime /
486             if ( usrOptions.closeDecelLoopsStep == 0 ) usrOptions.closeDe
487             usrOptions.closeDecelVelInc = (long)(usrOptions.closeVel3 - u
488             if ( usrOptions.closeDecelVelInc == 0 ) usrOptions.closeDecel
489         }
490 
491 
492         // modo inicial: øopenLoop o closedLoop?
493         if ( !ctrlStatus.indexDetectado || !ctrlStatus.offsetCalculado )
494             uGF.bit.OpenLoop = 1;
495             ctrlStatus.usarEstimador = True;
496         }
497         else {   // resto de los casos: closed loop, con encoder
498             uGF.bit.OpenLoop = 0;
499             ctrlStatus.usarEstimador = False;
500         }
501         uGF.bit.ChangeMode = 1;
502 
503         // otras acciones necesarias antes de iniciar el movimiento
504         Startup_Lock = 0;
505         Startup_Ramp = 0;
506         trans_counter = 0;
507         uGF.bit.ChangeSpeed = 0;
508         if ( orden == ORDEN_CALCULAR_OFFSET ) {
509             ctrlStatus.offsetCalculado = False;
510         }
511         else if ( orden == ORDEN_APRENDIZAJE ) {
512             ctrlStatus.aprendizajeDone = False;   // borra flags
513             ctrlStatus.posChecked = False;
514             ctrlStatus.posRefSet = False;
515         }
516 /*        else if ( orden == ORDEN_REARMAR ) {
517             ctrlStatus.posChecked = False;
518             ctrlStatus.posRefSet = False;
519         }*/
520 
521 
522         // init user specified parms and stop on error
523         if( SetupParm() ) {
524             // Error
525             uGF.bit.RunMotor=0;
526             return False;
527         }
528 
```

```
529         // zero out i sums
530         PIParmD.qdSum = 0;
531         PIParmQ.qdSum = 0;
532         PIParmW.qdSum = 0;
533 
534         // Enable ADC interrupt and begin main loop timing
535         IFS0bits.AD1IF = 0;
536         IEC0bits.AD1IE = 1;
537 
538 
539     //Run Motor loop
540         /////// repetido, quitar? SetupParm();
541         while (uGF.bit.RunMotor) {
542             if ( usrOptions.modoControl == CONTROL_EXTERNO ) {
543                 if ( !leerEntrada(entrRecibida) || (leerEntrada(ENTR_ABRI
544                     uGF.bit.RunMotor = 0;
545             }
546             else {  // modo control keypad o demo
547                 btnPulsado = leerBotonPulsado();
548                 if ( btnPulsado.btn == BTN_STOP ) {
549                     uGF.bit.RunMotor = 0;
550                 }
551                 else if ( btnPulsado.btn == BTN_RUN ) {
552                     if ( esperarPulsLarga(btnPulsado) ) {
553                         if ( orden != ORDEN_CERRAR ) {
554                             orden = ORDEN_CERRAR;
555                             uGF.bit.ChangeMode = 1;
556                         }
557                     }
558                     else {
559                         if ( orden != ORDEN_ABRIR ) {
560                             orden = ORDEN_ABRIR;
561                             uGF.bit.ChangeMode = 1;
562                         }
563                     }
564                 }
565             } // fin if (modo keypad o demo)
566         } // fin while(run motor)
567 
568         // comienza de nuevo main loop (deteccion de pulsacion de boton,
569 
570     } // End of Main loop
571 
572     // should never get here
573     while(1){}
574 }
575 
576 //-----------------------------------------------------------------------
```

```
577 // Executes one PI itteration for each of the three loops Id,Iq,Speed,
578 inline void DoControl( void )
579 {
580     #ifndef ENVOLTRIPPLE
581             ReadSignedADC0( &ReadADCParm );
582     #endif
583 
584     #ifdef ENVOLTRIPPLE
585         if(uGF.bit.ADCH0_PotOrVdc == 0)
586         {
587             ReadSignedADC0( &ReadADCParm );
588             AD1CHS0bits.CH0SA = 10;         // Selects DCBUS as CH0 Input
589             uGF.bit.ADCH0_PotOrVdc = 1;
590         }
591         else if(uGF.bit.ADCH0_PotOrVdc == 1)
592         {
593             ReadADC0_VDC();
594             AD1CHS0bits.CH0SA = 13;         // Selects POT as CH0 Input f
595             uGF.bit.ADCH0_PotOrVdc = 0;
596         }
597     #endif
598 
599     if( uGF.bit.OpenLoop ) {
600 
601         // si acaba de entrar a open loop, inicializa variables necesaria
602         if( uGF.bit.ChangeMode )
603         {
604             // just changed to openloop
605             uGF.bit.ChangeMode = 0;
606             // synchronize angles
607 
608             // VqRef & VdRef not used
609             CtrlParm.qVqRef = 0;
610             CtrlParm.qVdRef = 0;
611             CtrlParm.qVelRef = 0;
612 
613             // Initialize SMC
614             smc1.Valpha = 0;
615             smc1.Ealpha = 0;
616             smc1.EalphaFinal = 0;
617             smc1.Zalpha = 0;
618             smc1.EstIalpha = 0;
619             smc1.Vbeta = 0;
620             smc1.Ebeta = 0;
621             smc1.EbetaFinal = 0;
622             smc1.Zbeta = 0;
623             smc1.EstIbeta = 0;
624             smc1.Ialpha = 0;
```

```
625             smc1.IalphaError = 0;
626             smc1.Ibeta = 0;
627             smc1.IbetaError = 0;
628             smc1.Theta = 0;
629             smc1.Omega = 0;
630         }
631 
632         // fija valor de tension para la modulacion SVM
633         CtrlParm.qVqRef = REFINAMPS(INITIALTORQUE);
634 
635         if( recalcularPIW )  {
636             PIParmW.qInMeas = velocidad;
637             recalcularPIW = False;
638     }
639 
640         // PI control for D
641         PIParmD.qInMeas = ParkParm.qId;
642         PIParmD.qInRef  = CtrlParm.qVdRef;
643         CalcPI(&PIParmD);
644         ParkParm.qVd    = PIParmD.qOut;
645 
646         // Vector limitation
647         // Vd is not limited
648         // Vq is limited so the vector Vs is less than a maximum of 95%.
649         // The 5% left is needed to be able to measure current through
650         // shunt resistors.
651         // Vs = SQRT(Vd^2 + Vq^2) < 0.95
652         // Vq = SQRT(0.95^2 - Vd^2)
653         qVdSquared = FracMpy(PIParmD.qOut, PIParmD.qOut);
654         PIParmQ.qOutMax = Q15SQRT(Q15(0.95*0.95) - qVdSquared);
655         PIParmQ.qOutMin = -PIParmQ.qOutMax;
656 
657         // PI control for Q
658         PIParmQ.qInMeas = ParkParm.qIq;
659         PIParmQ.qInRef  = CtrlParm.qVqRef;
660         CalcPI(&PIParmQ);
661         ParkParm.qVq    = PIParmQ.qOut;
662     }
663 
664     else { // Closed Loop Vector Control
665 
666         // variables locales para DoControl
667         static int wkpObjetivo, wkiObjetivo;        // valores a fijar de
668 
669         // Calcula los parametros de los PIs de velocidad (el de velocida
670         if ( absVal(velocidad) < usrOptions.altaBajaLim ) {
671             wkpObjetivo = usrOptions.wkpBaja;
672             wkiObjetivo = usrOptions.wkiBaja;
```

```
673         }
674         else {
675             wkpObjetivo = usrOptions.wkpAlta;
676             wkiObjetivo = usrOptions.wkiAlta;
677         }
678 
679         // reinicio parametros
680         if ( prevDir == INDEF )
681             prevDir = ctrlStatus.dir;
682         if ( uGF.bit.ChangeMode || ctrlStatus.dir != prevDir ) {
683             PIParmW.qdSum = 0;
684             PIParmD.qdSum = 0;
685             PIParmQ.qdSum = 0;
686             PIParmW.qKp = wkpObjetivo;
687             PIParmW.qKi = wkiObjetivo;
688             movingTimeIters = 0UL;
689         }
690 
691         // si no esta en modo mantener par, detecta obstaculo
692         if ( !ctrlStatus.mantenerPar )
693             detectaObstaculo();
694 
695         // Si los parametros del PI de velocidad han sido reajustado, cam
696         if ( wkpObjetivo > PIParmW.qKp )
697             PIParmW.qKp++;
698         else if ( wkpObjetivo < PIParmW.qKp )
699             PIParmW.qKp--;
700         if ( wkiObjetivo > PIParmW.qKi )
701             PIParmW.qKi++;
702         else if ( wkiObjetivo < PIParmW.qKi )
703             PIParmW.qKi--;
704 
705         // DATA_MONITOR
706         #ifdef DM_ON
707         if ( uGF.bit.ChangeMode || ctrlStatus.dir != prevDir ) {
708             DM_arrPos = 0U;
709             while ( DM_arrPos < (DM_ARRAYSIZE) ) { DM_buffer1[DM_arrPos]
710             DM_arrPos = 0U; DM_delayCnt = 0UL; DM_arrOverflow = False;
711         }
712         if ( DM_delayCnt++ >= DM_delay ) {
713             if ( DM_arrPos < (DM_ARRAYSIZE) ) { DM_buffer1[DM_arrPos] = *
714             else { DM_arrOverflow = True; }
715             DM_delayCnt = 0UL;
716         }
717         #endif
718 
719         controlPuerta();
720 
```

```
721         prevDir = ctrlStatus.dir;
722         uGF.bit.ChangeMode = 0;
723         movingTimeIters++;
724 
725         if ( ctrlStatus.mantenerPar )
726             VelReq = 0;  // para correcta visualizacion de las graficas
727         CtrlParm.qVelRef = VelReq;
728 
729         if( recalcularPIW ) {
730             // Execute the velocity control loop
731             PIParmW.qInMeas = velocidad;
732             PIParmW.qInRef  = CtrlParm.qVelRef;
733             CalcPI(&PIParmW);
734             CtrlParm.qVqRef = PIParmW.qOut;
735             recalcularPIW = False;
736         }
737 
738         // si esta en modo de mantener par, ignora la referencia de veloc
739         if ( ctrlStatus.mantenerPar ) {
740             CtrlParm.qVqRef = IqMantener;
741         }
742 
743         // limite de corriente de referencia
744         if ( signo(CtrlParm.qVqRef) == signo(velocidad) ) {   // mismo si
745             if ( absVal(CtrlParm.qVqRef) > motorMaxCurr )
746                 CtrlParm.qVqRef = CtrlParm.qVqRef > 0 ? motorMaxCurr : -m
747         }
748         else {  // distinto signo: frenada
749             if ( absVal(CtrlParm.qVqRef) > frenoMaxCurr )
750                 CtrlParm.qVqRef = CtrlParm.qVqRef > 0 ? frenoMaxCurr : -f
751         }
752 
753         // If the application is running in torque mode, the velocity cor
754         if (uGF.bit.EnTorqueMod)    ///////// que hacer con esto
755             CtrlParm.qVqRef = CtrlParm.qVelRef;
756 
757         // Fieldweakening
758         CtrlParm.qVdRef = FieldWeakening(_Q15abs(CtrlParm.qVelRef));
759 
760         // PI control for D
761         PIParmD.qInMeas = ParkParm.qId;
762         PIParmD.qInRef  = CtrlParm.qVdRef;
763         CalcPI(&PIParmD);
764         if(uGF.bit.EnVoltRipCo)
765             ParkParm.qVd = VoltRippleComp(PIParmD.qOut);
766         else
767             ParkParm.qVd = PIParmD.qOut;
768 
```

```
769         // Vector limitation
770         // Vd is not limited
771         // Vq is limited so the vector Vs is less than a maximum of 95%.
772         // Vs = SQRT(Vd^2 + Vq^2) < 0.95
773         // Vq = SQRT(0.95^2 - Vd^2)
774         qVdSquared = FracMpy(ParkParm.qVd, ParkParm.qVd);
775         PIParmQ.qOutMax = Q15SQRT(Q15(0.95*0.95) - qVdSquared);
776         PIParmQ.qOutMin = -PIParmQ.qOutMax;
777 
778         // PI control for Q
779         PIParmQ.qInMeas = ParkParm.qIq;
780         PIParmQ.qInRef  = CtrlParm.qVqRef;
781         CalcPI(&PIParmQ);
782         if(uGF.bit.EnVoltRipCo)
783             ParkParm.qVq = VoltRippleComp(PIParmQ.qOut);
784         else
785             ParkParm.qVq = PIParmQ.qOut;
786 
787         // Limit, if motor is stalled, stop motor commutation (solo si es
788         if (ctrlStatus.usarEstimador && smc1.OmegaFltred < 0) {
789             uGF.bit.RunMotor = 0;
790         }
791     }
792 }
793 
794 
795 inline void CalculateParkAngle(void)
796 {
797     // medicion de posicion con encoder
798     ////// modificar para que use el numero de pares de polos configurado
799     angulo = __builtin_mulss( 128, (int)Read32bitQEI1PositionCounter() );
800 
801     // si esta en open loop usa el estimador
802     // tambien si esta en closed loop y no ha calculado el offset o detec
803     // esto ultimo no deberia pasar con un funcionamiento normal, pero se
804     if ( ctrlStatus.usarEstimador ) {
805         smc1.Ialpha = ParkParm.qIalpha;
806         smc1.Ibeta = ParkParm.qIbeta;
807         smc1.Valpha = ParkParm.qValpha;
808         smc1.Vbeta = ParkParm.qVbeta;
809         SMC_Position_Estimation(&smc1);
810     }
811 
812     if( uGF.bit.OpenLoop) {
813 
814         // primero bloquea el rotor
815         if (Startup_Lock < MotorParm.LockTime) {
816             Startup_Lock += 1;
```

```
817         }
818
819         // deteccion del index
820         else if ( orden != ORDEN_CALCULAR_OFFSET && !ctrlStatus.indexDete
821
822             // comprueba si se ha detectado el indice
823             ctrlStatus.indexDetectado = encoderIsIndexDetected();
824             if ( ctrlStatus.indexDetectado ) {
825                 Theta_error = ParkParm.qAngle - angulo - offsetAngulo;  /
826                 uGF.bit.OpenLoop = 0;       // siguiente iteracion sera e
827                 uGF.bit.ChangeMode = 1;
828                 ctrlStatus.usarEstimador = False;
829                 return;
830             }
831
832             // determina la direccion del movimiento
833             ctrlStatus.dir = orden==ORDEN_ABRIR ? OPEN : CLOSE;
834             VelReq = 300; // para deteccion de obstaculo a velocidad<100
835
836             // comprueba si hay obstaculo
837             detectaObstaculo();
838
839             // realiza la accion oportuna
840             if ( ctrlStatus.obstacle )
841                 ctrlStatus.mantenerPar = True;
842             else
843                 Startup_Ramp += (ctrlStatus.dir ? 1:-1) * usrOptions.reve
844         }
845
846         // va acelerando hasta EndSpeed (usando el estimador)
847         else if (Startup_Ramp < MotorParm.EndSpeed) {
848             Startup_Ramp += DELTA_STARTUP_RAMP;
849         }
850
851         // si se esta calculando el offset, lo calcula y lo guarda en la
852         else if ( orden == ORDEN_CALCULAR_OFFSET ) {
853             if ( ctrlStatus.indexDetectado ) {  // el indice debe estar d
854                 offsetAngulo = ParkParm.qAngle - angulo + 16384;    // 16
855                 setOffsetInFlash(offsetAngulo);
856                 ctrlStatus.offsetCalculado = True;
857                 uGF.bit.RunMotor = 0;   // para el motor
858             }
859         }
860
861         // si esta haciendo una operacion distinta a detectar el indice o
862         // esto no deberia pasar si todo funciona normal, pero se hace as
863         else {  // else IF usarEstimador
864             uGF.bit.OpenLoop = 0;
```

```
865             uGF.bit.ChangeMode = 1;
866             Theta_error = ParkParm.qAngle - smc1.Theta;
867         }
868 
869         ParkParm.qAngle += (int)(Startup_Ramp >> 16);
870     }
871 
872 
873     else {  // CLOSED LOOP
874 
875         // si ha detectado el indice y calculado el offset ParkAngle = en
876         if ( !ctrlStatus.usarEstimador ) {
877             if ( Theta_error < 0 )
878                 Theta_error++;
879             else if ( Theta_error > 0 )
880                 Theta_error--;
881             ParkParm.qAngle = angulo + offsetAngulo + Theta_error;  // sc
882         }
883 
884         // si no, usa el estimador
885         // esto no deberia pasar si todo funciona normal, pero se hace as
886         else {
887             ParkParm.qAngle = smc1.Theta + Theta_error;
888             if((_Q15abs(Theta_error) > _0_05DEG)&&(trans_counter == 0)) {
889                 if (Theta_error < 0)
890                     Theta_error += _0_05DEG;
891                 else
892                     Theta_error -= _0_05DEG;
893             }
894         }
895     }
896 }
897 
898 
899 
900 //-----------------------------------------------------------------------
901 // The ADC ISR does speed calculation and executes the vector update loop
902 // The ADC sample and conversion is triggered by the PWM period.
903 // The speed calculation assumes a fixed time interval between calculatio
904 //-----------------------------------------------------------------------
905 
906 void __attribute__((interrupt, no_auto_psv)) _AD1Interrupt(void)
907 {
908     IFS0bits.AD1IF = 0;
909 
910     // Increment count variable that controls execution
911     // of display and button functions.
912     iADCisrCnt++;
```

```
913
914     if( uGF.bit.RunMotor )
915     {
916
917
918             // Calculate qIa,qIb
919             MeasCompCurr();
920
921             // Calculate commutation angle using estimator
922             CalculateParkAngle();
923
924             // Calculate qId,qIq from qSin,qCos,qIa,qIb
925             ClarkePark();
926
927             // Calculate control values
928             DoControl();
929
930             // Calculate qSin,qCos from qAngle
931             SinCos();
932
933             // Calculate qValpha, qVbeta from qSin,qCos,qVd,qVq
934             InvPark();
935
936             // Calculate Vr1,Vr2,Vr3 from qValpha, qVbeta
937             CalcRefVec();
938
939             // Calculate and set PWM duty cycles from Vr1,Vr2,Vr3
940             CalcSVGen();
941
942     }
943
944     return;
945 }
946
947 //-----------------------------------------------------------------------
948 bool SetupParm(void)
949 {
950     // Turn saturation on to insure that overflows will be handled smooth
951     CORCONbits.SATA  = 0;
952
953     // Setup required parameters
954
955 // ============= Open Loop ======================
956     // Motor End Speed Calculation
957     // MotorParm.EndSpeed = ENDSPEEDOPENLOOP * POLEPAIRS * LOOPTIMEINSEC
958     // Then, * 65536 which is a right shift done in "void CalculateParkAn
959     // ParkParm.qAngle += (int)(Startup_Ramp >> 16);
960     MotorParm.EndSpeed = ENDSPEEDOPENLOOP * POLEPAIRS * LOOPTIMEINSEC * 6
```

```
961     MotorParm.LockTime = LOCKTIME;
962
963 // ============= ADC - Measure Current & Pot =====================
964
965     // Scaling constants: Determined by calibration or hardware design.
966     ReadADCParm.qK      = DQK;
967
968     MeasCurrParm.qKa    = DQKA;
969     MeasCurrParm.qKb    = DQKB;
970
971 // ============= SVGen ===============
972     // Set PWM period to Loop Time
973     SVGenParm.iPWMPeriod = LOOPINTCY;
974
975    // Initialise OPAMP/Comparator
976     // OpAmp 1,2,3 for Signal Conditioning Currents & Comparator 4 for Fa
977
978                                 // C1IN1+ :IBUS+ ; C1IN1- : IBUS-
979     CM1CON = 0x8C00;            // OpAmp1- OA1OUT is connected to pin,ope
980                                 // C2IN1+ :IA+ ; C2IN1- : IBUS+
981     CM2CON = 0x8C00;            // OpAmp2- OA2OUT is connected to pin,ope
982                                 // C3IN1+ :IB+ ; C3IN1- : IBUS+
983     CM3CON = 0x8C00;            // OpAmp3- OA3OUT is connected to pin,ope
984
985     CM4CON = 0xA011;        // cmp output inverted
986                             // VIN+ input connects to internal CVREFIN vo
987                             // VIN- input connects to CMP1 (source Ibus)
988
989     CVRCON = 0x0088;        // CVREF = (0.1031)*CVR + 0.825, CVR = 15
990     CM4FLTR = 0x000F;       // max filtering
991
992    // ============= Motor PWM ======================
993
994     // Center aligned PWM.
995     // Note: The PWM period is set to dLoopInTcy/2 but since it counts up
996     // and then down => the interrupt flag is set to 1 at zero => actual
997     // interrupt period is dLoopInTcy
998
999     PHASE1 = LOOPINTCY;
1000    PHASE2 = LOOPINTCY;
1001    PHASE3 = LOOPINTCY;
1002    PTPER = 2*LOOPINTCY+1;
1003
1004    PWMCON1 = 0x0204;   // Enable PWM output pins and configure them as
1005    PWMCON2 = 0x0204;   // complementary mode
1006    PWMCON3 = 0x0204;
1007
1008    DTR1 = 0x0000;
```

```
1009    DTR2 = 0x0000;
1010    DTR3 = 0x0000;
1011
1012    ALTDTR1 = DDEADTIME;    // 700ns of dead time
1013    ALTDTR2 = DDEADTIME;    // 700ns of dead time
1014    ALTDTR3 = DDEADTIME;    // 700ns of dead time
1015
1016    //fault disabled
1017    FCLCON1 = 0x0003;
1018    FCLCON2 = 0x0003;
1019    FCLCON3 = 0x0003;
1020
1021    PTCON2 = 0x0000;    // Divide by 1 to generate PWM
1022
1023    /* low side turn on errate workaraund */
1024    PDC1 = MIN_DUTY;   // PDC cannot be init with 0, please check errata
1025    PDC2 = MIN_DUTY;
1026    PDC3 = MIN_DUTY;
1027
1028    IPC23bits.PWM1IP = 4;   // PWM Interrupt Priority 4
1029    IPC23bits.PWM2IP = 4;   // PWM Interrupt Priority 4
1030    IPC24bits.PWM3IP = 4;   // PWM Interrupt Priority 4
1031    IFS5bits.PWM1IF=0;      // Clearing the PWM Interrupt Flag
1032    IEC5bits.PWM1IE=0;      // Enabling the PWM interrupt
1033
1034    //I/O pins initially controlled by GPIO
1035    IOCON1 = 0;
1036    IOCON2 = 0;
1037    IOCON3 = 0;
1038
1039    PTCON = 0x8000;         // Enable PWM for center aligned operation
1040
1041    //I/O pins controlled by PWM
1042    IOCON1 = 0xC000;
1043    Delay(100);             // Adding this delay allows enough time for h
1044    IOCON2 = 0xC000;
1045    Delay(100);             // Adding this delay allows enough time for h
1046    IOCON3 = 0xC000;
1047
1048
1049
1050    // SEVTCMP: Special Event Compare Count Register
1051    // Phase of ADC capture set relative to PWM cycle: 0 offset and count
1052    SEVTCMP = 0;
1053
1054    // ============= ADC - Measure Current & Pot ======================
1055    // ADC setup for simultanous sampling on
1056    //     CH1 - CMP0, CH2- CMP1, CH3 - CMP2
```

```
1057    // Sampling triggered by PWM and stored in signed fractional form.
1058
1059    // Signed fractional (DOUT = sddd dddd dd00 0000)
1060    AD1CON1bits.FORM = 3;
1061    AD1CON1bits.SSRC = 3;
1062    AD1CON1bits.SSRCG = 0;
1063    // Simultaneous Sample Select bit (only applicable when CHPS = 01 or
1064    // Samples CH0, CH1, CH2, CH3 simultaneously (when CHPS = 1x)
1065    // Samples CH0 and CH1 simultaneously (when CHPS = 01)
1066    AD1CON1bits.SIMSAM = 1;
1067    // Sampling begins immediately after last conversion completes.
1068    // SAMP bit is auto set.
1069    AD1CON1bits.ASAM = 1;
1070
1071
1072    AD1CON2 = 0;
1073    // Samples CH0, CH1, CH2, CH3 simultaneously (when CHPS = 1x)
1074    AD1CON2bits.CHPS = 2;
1075
1076    AD1CON3 = 0;
1077    // A/D Conversion Clock Select bits = 7 * Tcy
1078    AD1CON3bits.ADCS = 6;
1079
1080
1081    /* ADCHS: ADC Input Channel Select Register */
1082    AD1CHS0 = 0;
1083    // CH0 is AN13 for POT
1084    AD1CHS0bits.CH0SA = 13;
1085
1086   // CH1 positive input is CMP0, CH2 positive input is CMP1, CH3 positiv
1087    AD1CHS123bits.CH123SA = 1;
1088
1089 /* ADCSSL: ADC Input Scan Select Register */
1090    AD1CSSL = 0;
1091
1092    // Turn on A/D module
1093    AD1CON1bits.ADON = 1;
1094
1095    #ifdef ENVOLTRIPPLE
1096
1097        // CH0 is AN10 for VDC
1098        AD1CHS0bits.CH0SA = 10;
1099        // Wait until first conversion takes place to measure offsets.
1100        Delay(50);
1101        // Target DC Bus, without sign.
1102        TargetDCbus = ((SFRAC16)ADC1BUF0 >> 1) + Q15(0.5);
1103
1104        // CH0 is AN13 for POT
```

```
1105            AD1CHS0bits.CH0SA = 13;
1106            uGF.bit.ADCH0_PotOrVdc = 0;
1107
1108        #endif
1109
1110        // Wait until first conversion takes place to measure offsets.
1111        Delay(100);
1112
1113        //fault enabled
1114        FCLCON1 = 0x0058; //Fault enabled Fault SRC - Comparator 4 O/P
1115        FCLCON2 = 0x0058; //Fault enabled Fault SRC - Comparator 4 O/P
1116        FCLCON3 = 0x0058; //Fault enabled Fault SRC - Comparator 4 O
1117
1118        // Initial Current offsets
1119        InitMeasCompCurr( ADC1BUF2, ADC1BUF3 );
1120
1121        return False;
1122    }
1123
1124    void SetupControlParameters(void)
1125    {
1126
1127    // ============= PI D Term ===============
1128        PIParmD.qKp = DKP;
1129        PIParmD.qKi = DKI;
1130        PIParmD.qKc = DKC;
1131        PIParmD.qOutMax = DOUTMAX;
1132        PIParmD.qOutMin = -PIParmD.qOutMax;
1133
1134        InitPI(&PIParmD);
1135
1136    // ============= PI Q Term ===============
1137        PIParmQ.qKp = QKP;
1138        PIParmQ.qKi = QKI;
1139        PIParmQ.qKc = QKC;
1140        PIParmQ.qOutMax = QOUTMAX;
1141        PIParmQ.qOutMin = -PIParmQ.qOutMax;
1142
1143        InitPI(&PIParmQ);
1144
1145    // ============= PI W Term ===============
1146        PIParmW.qKp = WKP;
1147        PIParmW.qKi = WKI;
1148        PIParmW.qKc = WKC;
1149        PIParmW.qOutMax = WOUTMAX;
1150        PIParmW.qOutMin = -PIParmW.qOutMax;
1151
1152        InitPI(&PIParmW);
```

```
1153     return;
1154 }
1155 
1156 /*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
1157 /*****************************************************************************
1158 /* Function name: Delay
1159 /* Function parameters: Delay_Count
1160 /* Function return: None
1161 /* Description: Delay Routine - Can generate Delay in multiple of 100us
1162 /*****************************************************************************
1163 /*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
1164 void Delay(unsigned int Delay_Count)
1165 {
1166     int i;
1167     while (Delay_Count-- > 0)
1168     {
1169         for (i = 0;i < 1000;i++);
1170     }
1171     return;
1172 }
1173 
1174 
1175 // NOTE:
1176 //
1177 // If Input power supply has switching frequency noise, for example if a
1178 // switch mode power supply is used, Voltage Ripple Compensation is not
1179 // recommended, since it will generate spikes on Vd and Vq, which can
1180 // potentially make the controllers unstable.
1181 
1182 SFRAC16 VoltRippleComp(SFRAC16 Vdq)
1183 {
1184     SFRAC16 CompVdq;
1185     // DCbus is already updated with new DC Bus measurement
1186     // in ReadSignedADC0 subroutine.
1187     //
1188     // If target DC Bus is greater than what we measured last sample, adj
1189     // output as follows:
1190     //
1191     //                   TargetDCbus - DCbus
1192     // CompVdq = Vdq + ---------------------- * Vdq
1193     //                          DCbus
1194     //
1195     // If Measured DCbus is greater than target, then the following compe
1196     // is implemented:
1197     //
1198     //             TargetDCbus
1199     // CompVdq = ------------- * Vdq
1200     //                DCbus
```

```
1201     //
1202     // If target and measured are equal, no operation is made.
1203     //
1204     if (TargetDCbus > DCbus)
1205         CompVdq = Vdq + FracMpy(FracDiv(TargetDCbus - DCbus, DCbus), Vdq)
1206     else if (DCbus > TargetDCbus)
1207         CompVdq = FracMpy(FracDiv(TargetDCbus, DCbus), Vdq);
1208     else
1209         CompVdq = Vdq;
1210 
1211     return CompVdq;
1212 }
1213 
```

```c
#include "general.h"
#include <string.h>
#include "flash_handler.h"
#include "user_options_input.h"
#include "displays.h"
#include "botonera.h"


/***************************** DEFINICION DE LOS ELEMENTOS DEL MENU *******


#define hz2velRef(hz) ({ (int)((float)(hz)*60*10); })    // scale hertzs t
#define segs2time(segs)  ({ (unsigned long)( (float)(segs)*(unsigned long)


tUsrOptions usrOptions = {1,-1};  /* Objeto necesario para el control. Des
                                     * sus valores segun lo que elija



typedef  void (*fnPointer)(float);  /* typedef de puntero a funcion para h
                                     * el usuario elija un valor para algu
                                     * una funcion distinta, porque las ac
                                     * la necesidad de usar un puntero a f
                                     * en cada variable de tipo tOptionEle
                                     * El typedef es "fnPointer" (vaya sir

// struct que define las propiedades de cada una de las opciones que se pu
typedef struct {            // Informacion sobre como mostrar y manejar un
    char num [2];           // String con el numero identificativo. Por ej
    char numDigits;         // Numero de digitos, 1-4
    char posDecPoint;       // Posicion de la coma, 1-4 (4 = sin coma)
    float minVal;           // Valor minimo admisible, 0-9999
    float maxVal;           // Valor maximo admisible, 0-9999
    float defVal;           // Valor por defecto, 0-9999
    float currVal;          // Valor actual, 0-9999
    bool lock;              // Es una opcion reservada?, True/False
    fnPointer const setValFn;// Funcion que recibira el valor que elija el
} tOptionElem;

// struct que define las propiedades de los elementos del menu que te perm
typedef struct {            // Guarda informacion sobre un elemento del me
    char num;               // Numero identificativo. Por ejemplo, para F1
    tOptionElem* const * const children; // Elementos de tipo "opcion" que
    char numChildren;       // Numero de elementos de tipo "opcion" que co
} tMenuElem;


```

```
49
50
51  /****************************** MODIFICAR MENU ***************************
52
53  // INICIALIZACION F0
54
55  void fnF002 (float val) { usrOptions.modoControl = (bool) val; }
56  tOptionElem f002 = { "02", 1, 4, CONTROL_KEYPAD, CONTROL_DEMO, CONTROL_DEM
57
58  void fnF003 (float val) { usrOptions.reverse = val ? usrOptions.REVERSE :
59  tOptionElem f003 = { "03", 1, 4, 0, 1, 0, 1, False, &fnF003 };
60
61  tOptionElem* const f0children [] = {&f002, &f003};     // const indica que
62  tMenuElem f0 = { '0', f0children, sizeof(f0children)/sizeof(tOptionElem*)
63
64
65  // INICIALIZACION F1
66
67  void fnF101 (float val) { usrOptions.openDist1scale = val / 100; }
68  tOptionElem f101 = { "01", 3, 3, 0, 30, 15, 10, False, &fnF101 };
69
70  void fnF102 (float val) { usrOptions.openVel1 = hz2velRef ( val ); }
71  tOptionElem f102 = { "02", 4, 2, 1, 25, 7, 1, False, &fnF102 };
72
73  void fnF103 (float val) { usrOptions.openAcelTime = segs2time( val ); }
74  tOptionElem f103 = { "03", 4, 3, 0.1, 999.9, 2, 1, False, &fnF103 };
75
76  void fnF104 (float val) { usrOptions.openVel2 = hz2velRef( val ); }
77  tOptionElem f104 = { "04", 4, 2, 0, 128, 20, 3.7, False, &fnF104 };
78
79  void fnF106 (float val) { usrOptions.openDist2scale = val / 100; }
80  tOptionElem f106 = { "06", 3, 3, 60, 95, 70, 85, False, &fnF106 };
81
82  void fnF107 (float val) { usrOptions.openDecelTime = segs2time( val ); }
83  tOptionElem f107 = { "07", 4, 3, 0.1, 999.9, 2, 1, False, &fnF107 };
84
85  void fnF108 (float val) { usrOptions.openVel3 = hz2velRef( val ); }
86  tOptionElem f108 = { "08", 4, 2, 0, 15, 3, 0.5, False, &fnF108 };
87
88  void fnF109 (float val) { usrOptions.openFinalPulsos = val; }
89  tOptionElem f109 = { "09", 4, 4, 1, 9999, 600, 400, False, &fnF109 };
90
91  void fnF110 (float val) { usrOptions.openCorrFinScale = val / 100; }
92  tOptionElem f110 = { "10", 4, 3, 0, 100, 50, 70, False, &fnF110 };
93
94  void fnF111 (float val) { usrOptions.openCorrMantenerScale = val / 100; }
95  tOptionElem f111 = { "11", 4, 3, 0, 100, 50, 15, False, &fnF111 };
96
```

```
97  tOptionElem* const f1children [] = {&f101,&f102,&f103,&f104,&f106,&f107,&f
98  tMenuElem f1 = { '1', f1children, sizeof(f1children)/sizeof(tOptionElem*)
99
100
101 // INICIALIZACION F2
102
103 void fnF201 (float val) { usrOptions.closeDist1scale = val / 100; }
104 tOptionElem f201 = { "01", 3, 3, 0, 30, 15, 10, False, &fnF201 };
105
106 void fnF202 (float val) { usrOptions.closeVel1 = hz2velRef ( val ); }
107 tOptionElem f202 = { "02", 4, 2, 1, 25, 6, 1, False, &fnF202 };
108
109 void fnF203 (float val) { usrOptions.closeAcelTime = segs2time( val ); }
110 tOptionElem f203 = { "03", 4, 3, 0.1, 999.9, 2, 1, False, &fnF203 };
111
112 void fnF204 (float val) { usrOptions.closeVel2 = hz2velRef( val ); }
113 tOptionElem f204 = { "04", 4, 2, 0, 128, 15, 3.7, False, &fnF204 };
114
115 void fnF206 (float val) { usrOptions.closeDist2scale = val / 100; }
116 tOptionElem f206 = { "06", 3, 3, 60, 95, 70, 80, False, &fnF206 };
117
118 void fnF207 (float val) { usrOptions.closeDecelTime = segs2time( val ); }
119 tOptionElem f207 = { "07", 4, 3, 0.1, 999.9, 2, 1, False, &fnF207 };
120
121 void fnF208 (float val) { usrOptions.closeVel3 = hz2velRef( val ); }
122 tOptionElem f208 = { "08", 4, 2, 0, 15, 2, 0.5, False, &fnF208 };
123
124 void fnF209 (float val) { usrOptions.closeFinalPulsos = val; }
125 tOptionElem f209 = { "09", 4, 4, 1, 9999, 600, 400, False, &fnF209 };
126
127 void fnF210 (float val) { usrOptions.closeCorrFinScale = val / 100; }
128 tOptionElem f210 = { "10", 4, 3, 0, 100, 50, 40, False, &fnF210 };
129
130 void fnF211 (float val) { usrOptions.closeCorrMantenerScale = val / 100; }
131 tOptionElem f211 = { "11", 4, 3, 0, 100, 50, 15, False, &fnF211 };
132
133 void fnF215 (float val) { usrOptions.closeDist3scale = val / 100; }
134 tOptionElem f215 = { "15", 3, 3, 3, 15, 10, 10, False, &fnF215 };
135
136 void fnF216 (float val) { usrOptions.closeVel4 = hz2velRef(val); }
137 tOptionElem f216 = { "16", 4, 2, 0, 20, 5, 1, False, &fnF216 };
138
139 tOptionElem* const f2children [] = {&f201,&f202,&f203,&f204,&f206,&f207,&f
140 tMenuElem f2 = { '2', f2children, sizeof(f2children)/sizeof(tOptionElem*)
141
142
143 // INICIALIZACION F3
144
```

```
145 void fnF300 (float val) { usrOptions.lowVel = hz2velRef( val ); }
146 tOptionElem f300 = { "00", 4, 2, 0, 20, 1, 1, False, &fnF300 };
147
148 void fnF301 (float val) { usrOptions.openMaxTime = segs2time(val); }
149 tOptionElem f301 = { "01", 4, 4, 0, 3600, 300, 300, False, &fnF301 };
150
151 void fnF302 (float val) { usrOptions.closeMaxTime = segs2time(val); }
152 tOptionElem f302 = { "02", 4, 4, 0, 3600, 300, 300, False, &fnF302 };
153
154 void fnF303 (float val) { usrOptions.lowVelMaxTime = segs2time(val); }
155 tOptionElem f303 = { "03", 4, 4, 0, 3600, 400, 400, False, &fnF303 };
156
157 void fnF307 (float val) { usrOptions.openIfObstacle = (bool)val; }
158 tOptionElem f307 = { "07", 1, 4, 0, 1, 1, 1, False, &fnF307 };
159
160 void fnF308 (float val) { usrOptions.closeObstacleSensitivity = val / 100;
161 tOptionElem f308 = { "08", 4, 3, 0, 150, 50, 70, False, &fnF308 };
162
163 void fnF310 (float val) { usrOptions.openObstacleSensitivity = val / 100;
164 tOptionElem f310 = { "10", 4, 3, 0, 150, 50, 80, False, &fnF310 };
165
166 void fnF311 (float val) { usrOptions.demoOpenMantParTime = segs2time(val);
167 tOptionElem f311 = { "11", 4, 4, 1, 3600, 2, 2, False, &fnF311 };
168
169 void fnF312 (float val) { usrOptions.demoCloseMantParTime = segs2time(val)
170 tOptionElem f312 = { "12", 4, 4, 1, 3600, 2, 2, False, &fnF312 };
171
172 void fnF313 (float val) { usrOptions.demoMaxNumMovimientos = (int)val; }
173 tOptionElem f313 = { "13", 4, 4, 0, 9999, 0, 0, False, &fnF313 };
174
175 tOptionElem* const f3children[] = {&f300, &f301, &f302, &f303, &f307, &f30
176 tMenuElem f3 = { '3', f3children, sizeof(f3children)/sizeof(tOptionElem*)
177
178
179 // INICIALIZACION F4
180
181 void fnF404 (float val) { usrOptions.aprendizVel = hz2velRef(val); }
182 tOptionElem f404 = { "04", 4, 2, 0, 20, 5, 0.5, False, &fnF404 };
183
184 void fnF405 (float val) { usrOptions.doAprendizaje = (bool)val; }
185 tOptionElem f405 = { "05", 1, 4, 0, 1, 0, 0, False, &fnF405 };
186
187 tOptionElem* const f4children[] = {&f404, &f405};
188 tMenuElem f4 = { '4', f4children, sizeof(f4children)/sizeof(tOptionElem*)
189
190
191 // INICIALIZACION F7
192
```

```
193 void fnF700 (float val) { usrOptions.wkpAlta = val; }
194 tOptionElem f700 = { "00", 4, 4, 1, 9999, 4500, 4000, False, &fnF700 };
195
196 void fnF701 (float val) { usrOptions.wkiAlta = val; }
197 tOptionElem f701 = { "01", 4, 4, 1, 9999, 150, 600, False, &fnF701 };
198
199 void fnF702 ( float val) { usrOptions.wkpBaja = val; }
200 tOptionElem f702 = { "02", 4, 4, 1, 9999, 4500, 4000, False, &fnF702 };
201
202 void fnF703 (float val) { usrOptions.wkiBaja = val; }
203 tOptionElem f703 = { "03", 4, 4, 1, 9999, 150, 600, False, &fnF703 };
204
205 void fnF704 (float val) { usrOptions.altaBajaLim = hz2velRef(val); }
206 tOptionElem f704 = { "04", 4, 2, 0, 50, 5, 2, False, &fnF704 };
207
208 void fnF706 (float val) { usrOptions.corrLimMotorScale = val / 100; }
209 tOptionElem f706 = { "06", 4, 3, 0, 200, 100, 100, False, &fnF706 };
210
211 void fnF707 (float val) { usrOptions.corrLimFrenoScale = val / 100; }
212 tOptionElem f707 = { "07", 4, 3, 0, 200, 100, 100, False, &fnF707 };
213
214 void fnF708 (float val) { usrOptions.ikp = val; }
215 tOptionElem f708 = { "08", 4, 4, 1, 9999, 2000, 3000, False, &fnF708 };
216
217 void fnF709 (float val) { usrOptions.iki = val; }
218 tOptionElem f709 = { "09", 4, 4, 1, 9999, 20, 50, False, &fnF709 };
219
220 tOptionElem* const f7children[] = {&f700,&f701,&f702,&f703,&f704,&f706,&f7
221 tMenuElem f7 = { '7', f7children, sizeof(f7children)/sizeof(tOptionElem*)
222
223
224 // INICIALIZACION MENU RAIZ
225
226 tMenuElem* const menuRaiz [] = { &f0, &f1, &f2, &f3, &f4, &f7 }; // array
227
228 /****************************** FIN MODIFICAR MENU ************************
229
230
231
232
233
234
235
236
237
238
239
240
```

```
241 /*************************** MANEJO DEL MENU ****************************
242
243
244 // Variables que determinan el estado del menu (nivel del menu, elemento a
245 #define LEVEL1 0
246 #define LEVEL2 1
247 #define OPTIONLEVEL 2
248
249 char chars7seg[5];              // Cadena de 4 caracteres (mas el final de
250 int incremento = 1;
251 struct {
252     unsigned short currElemLvl1:4;  // Posicion del elemento actual (del t
253     unsigned short currElemLvl2:4;  // Posicion del elemento actual (del t
254     unsigned short currLvl:2;       // LEVEL1 / LEVEL2 / OPTIONLEVEL
255     unsigned short elemLvl1Count:4; // Total numero de elementos del menu
256     bool optionsLoadedFromFlash:1;  // Indica si ya se han recuperado o no
257
258     unsigned unused:1;
259 } menuState = { 0, 0, LEVEL1, sizeof(menuRaiz)/sizeof(tMenuElem*), False }
260
261
262 // Macros
263 #define currMenuElem ( *menuRaiz[menuState.currElemLvl1] )
264 #define currOptionElem ( *currMenuElem.children[menuState.currElemLvl2] )
265 #define my10pow(ex) ({ int  _ex = (ex), num = 1; /* devuelve 10^ex */  \
266                             while ( _ex-- ) num *= 10;      \
267                             num; })
268
269 // Funciones para mostrar el menu raiz, y operar con el
270 inline void showMenuLevel1(void) { // Cambia texto de los displays a -FX-
271     strcpy( chars7seg, "-FX-" );
272     chars7seg[2] = currMenuElem.num;
273     menuState.currLvl = LEVEL1;
274     refreshDisplays( chars7seg, 4 );
275 }
276 inline void lvl1Up(void) {
277     if ( menuState.currElemLvl1  == menuState.elemLvl1Count - 1 )
278         menuState.currElemLvl1 = 0;
279     else
280         menuState.currElemLvl1++;
281     showMenuLevel1();
282 }
283 inline void lvl1Down(void) {
284     if ( menuState.currElemLvl1 == 0 )
285         menuState.currElemLvl1 = menuState.elemLvl1Count - 1;
286     else
287         menuState.currElemLvl1--;
288     showMenuLevel1();
```

```
289 }
290
291
292 // Funciones para mostrar el nivel 2 del menu, y operar con el
293 inline void showMenuLevel2(void) { // Cambia texto de los displays a FX.X
294     chars7seg[0] = 'F';
295     chars7seg[1] = currMenuElem.num;
296     chars7seg[2] = currOptionElem.num[0];
297     chars7seg[3] = currOptionElem.num[1];
298     menuState.currLvl = LEVEL2;
299     refreshDisplays( chars7seg, 2 );
300 }
301 inline void lvl2Up(void) {
302     if ( menuState.currElemLvl2 == currMenuElem.numChildren - 1 )
303         menuState.currElemLvl2 = 0;
304     else
305         menuState.currElemLvl2++;
306     showMenuLevel2();
307 }
308 inline void lvl2Down(void) {
309     if ( menuState.currElemLvl2 == 0 )
310         menuState.currElemLvl2 = currMenuElem.numChildren - 1;
311     else
312         menuState.currElemLvl2--;
313     showMenuLevel2();
314 }
315
316
317 // Funciones para mostrar la opcion del menu actualmente seleccionada, y
318 inline void showOption(val) {
319     int i;
320     strcpy( chars7seg, "    ");                    // resetea texto, por
321     for ( i=0; i<currOptionElem.numDigits; i++ ) {  // extrae los digitos
322         val /= i==0? 1 : 10;
323         chars7seg[3-i] = val % 10 + '0';           // se le suma el cara
324     }
325     menuState.currLvl = OPTIONLEVEL;
326     refreshDisplays( chars7seg, currOptionElem.posDecPoint );
327 }
328 inline int optionUp(int val) {
329     int max = currOptionElem.maxVal * my10pow( 4 - currOptionElem.posDecP
330     if ( (int)(currOptionElem.maxVal*my10pow(4-currOptionElem.posDecPoint
331         max++;
332     if ( val < max ) {
333         val += incremento;  // val += my10pow( 3 - menuState.blinkingDigi
334         if ( val > max )
335             val = max;
336     }
```

```
337     showOption(val);
338     return val;
339 }
340 inline int optionDown(int val) {
341     int min = currOptionElem.minVal * my10pow( 4 - currOptionElem.posDecP
342     if ( (int)(currOptionElem.minVal*my10pow(4-currOptionElem.posDecPoint
343         min++;
344     if ( val > min ) {
345         val -= incremento; // val -= my10pow( 3 - menuState.blinkingDigit
346         if ( val < min )
347             val = min;
348     }
349     showOption(val);
350     return val;
351 }
352 inline void saveToFlash(int val) { // si el usuario actualiza un valor, l
353     char i, pos=0; // pos es la direccion de memoria donde guardar el dat
354     for ( i=0; i<menuState.currElemLvl1; i++ ) {    // va incrementando p
355         pos += menuRaiz[i]->numChildren;            // el incremento de p
356     }
357     pos += menuState.currElemLvl2;                  // incrementamos pos
358     setOptionInFlash( pos, (int)val );              // graba el dato
359 }
360
361 void entrarMenu (void) {
362     menuState.currElemLvl1 = 0;
363     menuState.currElemLvl2 = 0;
364     menuState.currLvl = LEVEL1;
365     showMenuLevel1();
366 }
367 bool menuEditandoOpcion(void) {
368     return menuState.currLvl == OPTIONLEVEL;
369 }
370 void menuIncVel (void) {
371     static unsigned int cnt = 0;
372     if ( cnt++ >= 5 )
373         cnt = 0;
374     if ( cnt = 0 ) {
375         if ( incremento < 2 )
376             incremento = 2;
377         else if ( incremento < 5 )
378             incremento = 5;
379         else if ( incremento < 25 )
380             incremento = 25;
381         else
382             incremento = 50;
383     }
384 }
```

```
385 void menuResetVel (void) {
386     incremento = 1;
387 }
388 bool accionMenu (tBoton btn) {
389     static int currShownVal;
390     if ( btn == BTN_ARRIBA ) { // UP
391         if ( menuState.currLvl == LEVEL1 )
392             lvl1Up();
393         else if ( menuState.currLvl == LEVEL2 )
394             lvl2Up();
395         else if ( menuState.currLvl == OPTIONLEVEL )
396             currShownVal = optionUp(currShownVal);
397     }
398     else if ( btn == BTN_ABAJO ) { // DOWN
399         if ( menuState.currLvl == LEVEL1 )
400             lvl1Down();
401         else if ( menuState.currLvl == LEVEL2 )
402             lvl2Down();
403         else if ( menuState.currLvl == OPTIONLEVEL )
404             currShownVal = optionDown(currShownVal);
405     }
406     else if ( btn == BTN_SET ) { // OK
407         if ( menuState.currLvl == LEVEL1 ) {
408             showMenuLevel2();
409         }
410         else if ( menuState.currLvl == LEVEL2 ) {
411             currShownVal = currOptionElem.currVal * my10pow(4-currOptionE
412             showOption(currShownVal);
413         }
414         else if ( menuState.currLvl == OPTIONLEVEL ) {
415             currOptionElem.currVal = (float)currShownVal / (float)my10pow
416             saveToFlash(currShownVal);
417             if ( currOptionElem.setValFn != NULL )
418                 (*currOptionElem.setValFn)( currOptionElem.currVal );
419             lvl2Up();
420         }
421     }
422     else if ( btn == BTN_MODE ) { // BACK
423         if ( menuState.currLvl == LEVEL1 ) {
424             refreshDisplays( "    ", 4 );   // borra los displays
425             return False;                   // sale del menu
426         }
427         else if ( menuState.currLvl == LEVEL2 )
428             showMenuLevel1();
429         else if ( menuState.currLvl == OPTIONLEVEL )
430             showMenuLevel2();
431     }
432
```

```
433     return True;    // devuelve True, que significa que aun se permanece
434 }
435
436
437
438
439
440
441 /****************************** CARGAR LOS VALORES GUARDADOS ************
442
443
444 inline void getAllOptionsFromFlash (void) {  // inicia todos los valores
445     unsigned int i,j,pos=0;
446     tOptionElem* p;
447     bool optionsWereSaved = areOptionsSavedInFlash();   /* Comprueba si y
448
449     if ( optionsWereSaved ) {
450         for( i=0; i<menuState.elemLvl1Count; i++ ) {    // se itera por t
451             for ( j=0; j<menuRaiz[i]->numChildren; j++ ) {      // se ite
452                 p = menuRaiz[i]->children[j];
453                 p->currVal = (float)getOptionFromFlash(pos) / (float)my10
454                 pos++;  // j se reinicia en cada iteracion de i, pero pos
455             }
456         }
457     }
458     else {
459         for( i=0; i<menuState.elemLvl1Count; i++ ) {    // se itera por t
460             for ( j=0; j<menuRaiz[i]->numChildren; j++ ) {      // se ite
461                 p = menuRaiz[i]->children[j];
462                 setOptionInFlashBuffer( pos, p->currVal * my10pow(4-p->po
463                 pos++;  // j se reinicia en cada iteracion de i, pero pos
464             }
465         }
466     }
467     if ( !optionsWereSaved ) {      // Si ha sido la primera vez que se l
468         saveOptionsBufferInFlash();             // Ejecuta de forma efecti
469     }
470     menuState.optionsLoadedFromFlash = True;
471 }
472
473 void getDefaultOptionsValues(void) {
474     unsigned int i, j;
475     if ( !menuState.optionsLoadedFromFlash ) {
476        getAllOptionsFromFlash();
477     }
478     for ( i=0; i<menuState.elemLvl1Count; i++ ) {
479         for ( j=0; j<menuRaiz[i]->numChildren; j++ ) {
480             (*menuRaiz[i]->children[j]->setValFn)( menuRaiz[i]->children[
```

```
481         }
482     }
483     return;
484 }
485 void getUserOptionsValues(void) {
486     unsigned int i, j;
487     if ( !menuState.optionsLoadedFromFlash ) {
488        getAllOptionsFromFlash();
489     }
490     for ( i=0; i<menuState.elemLvl1Count; i++ ) {
491         for ( j=0; j<menuRaiz[i]->numChildren; j++ ) {
492             (*menuRaiz[i]->children[j]->setValFn)( menuRaiz[i]->children[
493         }
494     }
495     return;
496 }
497
```

```
/*
 * File:   user_options_input.h
 * Author: IÒigo Huguet - Fanamoel
 *
 * Created on February 25, 2014, 1:01 PM
 */

#ifndef USER_OPTIONS_INPUT_H
#define USER_OPTIONS_INPUT_H

    void getDefaultOptionsValues (void);    // Carga los valores de fabric
    void getUserOptionsValues (void);       // Carga los valores definidos
    bool accionMenu (unsigned int btn);     // Ejecuta la accion del menu
    void entrarMenu (void);                 // Entra al menu
    bool menuEditandoOpcion(void);
    void menuIncVel (void);
    void menuResetVel (void);

typedef enum { CONTROL_KEYPAD, CONTROL_EXTERNO, CONTROL_DEMO} tModoControl
typedef struct {
    const short DIRECT;
    const short REVERSE;

    // modo de control
    tModoControl modoControl;

    // direccion
    int reverse;                    // direccion: DIRECT / REVERSE

    // parametros de distancias, velocidad y tiempos de apertura
    unsigned long openMaxTime;      // Tiempo maximo de apertura
    float openDist1scale;           // Distancia primer tramo (baja veloci
    int openVel1;                   // Velocidad de referencia para el pri
    unsigned long openAcelTime;     // Tiempo de aceleracion EN NUMERO DE
    int openVel2;                   // Velocidad de referencia en el segur
    float openDist2scale;           // Distancia absoluta (desde pos 0) ha
    unsigned long openDecelTime;    // Tiempo de deceleracion EN NUMERO DE
    int openVel3;                   // Velocidad de referencia en el tramo
    unsigned int openFinalPulsos;   // Numero de pulsos al final del recor
    float openCorrFinScale;         // Corriente con la que se considera c
    float openCorrMantenerScale;    // Corriente que hay que mantener cuar

    // parametros de distancias, velocidad y tiempos de cierre
    unsigned long closeMaxTime;     // Tiempo maximo de cierre
    float closeDist1scale;          // Distancia primer tramo (baja veloci
    int closeVel1;                  // Velocidad de referencia para el pri
    unsigned long closeAcelTime;    // Tiempo de aceleracion EN NUMERO DE
    int closeVel2;                  // Velocidad de referencia en el segur
```

```
49      float closeDist2scale;          // Distancia absoluta (desde pos 0) ha
50      unsigned long closeDecelTime;   // Tiempo de deceleracion EN NUMERO DE
51      int closeVel3;                  // Velocidad de referencia en el tramo
52      float closeDist3scale;          // Distancia absoluta (desde pos final
53      int closeVel4;                  // Velocidad de referencia para el tra
54      unsigned int closeFinalPulsos;  // Numero de pulsos al final del recor
55      float closeCorrFinScale;        // Corriente con la que se considera c
56      float closeCorrMantenerScale;   // Corriente que hay que mantener cuan
57
58      // modo baja velocidad
59      int lowVel;                     // Velocidad de referencia en modo baj
60      unsigned long lowVelMaxTime;    // Tiempo maximo de apertura o cierre
61
62      // modo demo
63      unsigned long demoOpenMantParTime;  // Tiempo que hay que mantener el
64      unsigned long demoCloseMantParTime; // Tiempo que hay que mantener el
65      int demoMaxNumMovimientos;          // Numero de procesos de cierre-ap
66
67      // aprendizaje
68      int aprendizVel;                // Velocidad de referencia para el apr
69      bool doAprendizaje;             // Iniciar aprendizaje
70
71      // parametros para cuando hay obstaculos
72      bool openIfObstacle;            // Determina si, al encontrar un obsta
73      float openObstacleSensitivity;  // Sensibilidad a cambios de velocidad
74      float closeObstacleSensitivity; // Sensibilidad a cambios de velocidad
75
76      // limites de corriente
77      float corrLimFrenoScale;        // Limite de corriente para cuando act
78      float corrLimMotorScale;        // Limite de corriente para cuando act
79
80      // PIs
81      int wkpAlta;                    // PIs velocidad
82      int wkiAlta;
83      int wkpBaja;
84      int wkiBaja;
85      int altaBajaLim;                // transicion entre alta y baja veloci
86      int ikp;                        // PIs de corriente
87      int iki;
88
89
90      // parametros calculados a partir de los anteriores
91      long openDist1;         // Distancia primer tramo EN NUMERO DE PULSOS
92      long openDist2;         // Distancia absoluta hasta fin de tramo 2 EN
93      long closeDist1;        // Distancia primer tramo EN NUMERO DE PULSOS
94      long closeDist2;        // Distancia absoluta hasta fin de tramo 2 EN
95      long closeDist3;        // Distancia absoluta desde el final del recor
96
```

```
97      unsigned long openAcelLoopsStep;    // numero de loops que hay que esp
98      int openAcelVelInc;                 // incremento de VelReq durante la
99      unsigned long openDecelLoopsStep;   // numero de loops que hay que esp
100     int openDecelVelInc;                // incremento de VelReq durante la
101
102     unsigned long closeAcelLoopsStep;   // idem para el cierre
103     int closeAcelVelInc;
104     unsigned long closeDecelLoopsStep;
105     int closeDecelVelInc;
106
107     unsigned int closeFinCurr;          // corrientes para deteccion de ob
108     unsigned int openFinCurr;
109
110     unsigned int openMantCurr;          // corrientes que hay que mantener
111     unsigned int closeMantCurr;
112 } tUsrOptions;
113
114
115
116 #endif  /* USER_OPTIONS_INPUT_H */
117
118
```

```
1  #ifndef UserParms_H
2  #define UserParms_H
3
4  #include "pi.h"
5
6
7  /***************** puertos *****************/
8  // si se cambian los puertos, se debe editar tambien la configuracion de l
9
10 // botonera
11 #define PULSDATA    (PORTCbits.RC3)
12 #define PULSCLK     (PORTAbits.RA9)
13 #define PULSLOAD    (PORTAbits.RA4)
14 #define COMPROB_PULS_DELAY 50
15
16 // entradas
17 #define ENTRADA_ABRIR (PORTEbits.RE12)
18 #define ENTRADA_CERRAR (PORTEbits.RE13)
19 #define ENTRADA_X1  (PORTGbits.RG6)
20 #define ENTRADA_X2  (PORTGbits.RG7)
21 #define ENTRADA_X3  (PORTGbits.RG8)
22 #define ENTRADA_X4  (PORTGbits.RG9)
23 #define ENTRADA_X5  (PORTAbits.RA11)
24
25 // displays
26 #define DISDATA     (PORTCbits.RC4)
27 #define DISCLK      (PORTCbits.RC13)
28 #define DISTROBE    (PORTBbits.RB7)
29 #define DISON       (PORTCbits.RC5)
30
31 // encoder: configurado en initdspic.c, mediante PPS (Peripheral Pin Selec
32
33 // otros
34 #define ALARMA_MOD_POT (PORTBbits.RB8)
35 #define SECLED      (PORTEbits.RE15)
36
37
38 //****** Parametros para deteccion de obstaculo ******
39 #define OBSTACLE_VEL_DETECTION_SECS 1.5
40 #define OBSTACLE_VEL_DETECTION_SECS_FINAL 0.5
41 #define OBSTACLE_DETECTION_SECS_DEADTIME 1.2
42
43
44 //*************** Start-Up Parameters **************
45 // parametros para arranque en lazo abierto: tras las modificaciones reali
46 // usara para detectar el indice
47 #define PWM_DT_ERRATA
48
```

```
49 #define DEADTIMESEC    0.000002 // Dead time in seconds - 2us
50
51 #define LOCKTIMEINSEC  1.0      // Initial rotor lock time in seconds
52                                 // Make sure LOCKTIMEINSEC*(1.0/LOOPTIMEIN
53                                 // is less than 65535.
54 #define OPENLOOPTIMEINSEC 5.0   // Open loop time in seconds. This is the
55                                 // will take from stand still to closed lo
56                                 // Optimized to overcome the brake inertia
57                                 // (Magtrol AHB-3 brake inertia = 6.89 kg
58 #define INITIALTORQUE   0.8     // Initial Torque demand in Amps.
59                                 // Enter initial torque demand in Amps usi
60                                 // macro. Maximum Value for reference is c
61                                 // shunt resistor value and differential a
62                                 // Use this equation to calculate maximum
63                                 // Amperes:
64                                 //
65                                 // Max REFINAMPS = (VDD/2)/(RSHUNT*DIFFAMP
66                                 //
67                                 // For example:
68                                 //
69                                 // RSHUNT = 0.005
70                                 // VDD = 3.3
71                                 // DIFFAMPGAIN = 75
72                                 //
73                                 // Maximum torque reference in Amps is:
74                                 //
75                                 // (3.3/2)/(.005*75) = 4.4 Amperes, or REF
76 #define ENDSPEEDOPENLOOP MINSPEEDINRPM
77
78
79
80 //*************** Motor Parameters **************
81 ///////// esto sera sustituido por los configurados via menu
82 #define POLEPAIRS       8       // Number of pole pairs
83 #define PHASERES        ((float)11.25)  // Phase resistance in Ohms.
84 #define PHASEIND        ((float)0.0275)// Phase inductance in Henrys.
85 #define NOMINALSPEEDINRPM 180   // Make sure NOMINALSPEEDINRPM generates a
86                                 // Use this formula:
87                                 // MAXOMEGA = NOMINALSPEEDINRPM*SPEEDLOOPT
88                                 // If MAXOMEGA > 1.0, reduce NOMINALSPEEDI
89                                 // speed loop faster by reducing SpeedLoop
90                                 // Maximum position of POT will set a refe
91                                 // NOMINALSPEEDINRPM.
92 #define MINSPEEDINRPM   100 // Minimum speed in RPM. Closed loop will oper
93                                 // speed. Open loop will transition to clo
94                                 // this minimum speed. Minimum POT positio
95                                 // a speed reference of MINSPEEDINRPM
96 #define FIELDWEAKSPEEDRPM 180   // Make sure FIELDWEAKSPEEDRPM generates a
```

```
97                                  // Use this formula:
98                                  // MAXOMEGA = FIELDWEAKSPEEDRPM*SPEEDLOOPT
99                                  // If MAXOMEGA > 1.0, reduce FIELDWEAKSPEE
100                                 // speed loop faster by reducing SpeedLoop
101                                 // Maximum position of POT will set a refe
102                                 // FIELDWEAKSPEEDRPM.
103 #define SPEEDDELAY 50           // Delay for the speed ramp.
104                       // Necessary for the PI control to work properly at
105
106
107 //*************** Oscillator Parameters **************
108
109 #define PLLIN       8000000     // External Crystal or Clock Frequency (Hz
110 #define DESIREDMIPS 70000000    // Enter desired MIPS.
111
112
113 //*************** PWM and Control Timing Parameters **********
114
115 #define PWMFREQUENCY    20000       // PWM Frequency in Hertz
116
117 // ya no se usan:
118 #define BUTPOLLOOPTIME  0.100       // Button polling loop period in sec
119 #define SPEEDLOOPFREQ   1000        // Speed loop Frequency in Hertz. This
120 // fin de "ya no se usan"
121
122 //*************** Slide Mode Controller Parameters **********
123 // valores del estimador, que ya casi no se usa
124 #define SMCGAIN         0.85        // Slide Mode Controller Gain (0.0 to
125 #define MAXLINEARSMC    0.005       // If measured current - estimated cur
126                                 // is less than MAXLINEARSMC, the slide mo
127                                 // Controller will have a linear behavior
128                                 // instead of ON/OFF. Value from (0.0 to 0
129 //*************** Hardware Parameters ****************
130
131 #define RSHUNT          0.01    // Value in Ohms of shunt resistors used.
132 #define DIFFAMPGAIN     15      // Gain of differential amplifier.
133 #define VDD         3.3 // VDD voltage, only used to convert torque
134                         // reference from Amps to internal variables
135
136 #define UMAX 340          // Maximum DC link voltage on inverter
137 #define IPEAK 8.8          // maximum current (peak to peak) that can be r
138
139
140 //*************** Generador de graficas para modo debug ******************
141 // acumula valores de las variables deseadas en arrays
142 // usando el modo debug se pueden exportar los valores de estos arrays a f
143
144 //#define DM_ON   // descomentar para activar data monitor
```

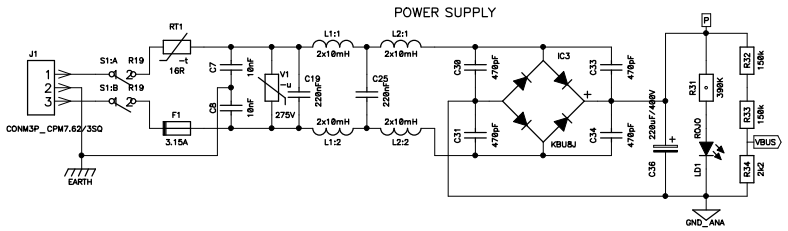
```
145 #define DM_NUMVARS 7                        // 1 a 7 - se puede poner por
146 #define DM_ARRAYSIZE (unsigned int) 1024    // numero total de muestras
147 #define DM_DELAY 200                        // cada cuantas interrupciones
148                                                 // se puede cambiar este v
149 // variables a monitorizar. Se pueden cambiar en modo debug, editando las
150 #define DM_VAR1 PIParmW.qInRef
151 #define DM_VAR2 velocidad
152 #define DM_VAR3 PIParmQ.qInRef
153 #define DM_VAR4 ParkParm.qIq
154 #define DM_VAR5 ParkParm.qValpha
155 #define DM_VAR6 ParkParm.qId
156 #define DM_VAR7 ParkParm.qIa
157
158
159
160 //*************** Optional Modes **************
161 // mejor no usar, no probados con el software tan modificado. øEliminar?
162 //#define TORQUEMODE
163 //#define ENVOLTRIPPLE
164
165 //*************** PI Coefficients **************
166 // estos valores ya no se usan. Se usan los configurados via menu de opcio
167 //******** D Control Loop Coefficients *******
168 #define     DKP         2000 //6000 //Q15(0.0005)
169 #define     DKI         20 //50 //1000//Q15(0.08)
170 #define     DKC         Q15(0.99999)
171 #define     DOUTMAX     Q15(0.35) //Q15(0.99999)
172
173 //******** Q Control Loop Coefficients *******
174 #define     QKP         2000 //6000 //Q15(0.0005)
175 #define     QKI         20 //50 //1000//Q15(0.08)
176 #define     QKC         Q15(0.99999)
177 #define     QOUTMAX     Q15(0.35) //Q15(0.99999)
178
179 //*** Velocity Control Loop Coefficients *****
180 #define     WKP         4500 //2500 //6000 //10000 //1500//Q15(0.025)
181 #define     WKI         150 //450 //250 //300 //200 //1500//Q15(0.005)
182 #define     WKC         Q15(0.99999)
183 #define     WOUTMAX     Q15(0.9)
184
185 //*************** ADC Scaling **************
186 // Scaling constants: Determined by calibration or hardware design.
187 #define     DQK         Q15(0.99999) // POT Scaling
188 #define     DQKA        Q15(0.5) // Current feedback software gain
189 #define     DQKB        Q15(0.5) // Current feedback software gain
190
191 //*************** Field Weakening **************
192 // Enter flux demand Amperes using REFINAMPS() macro. Maximum Value for
```

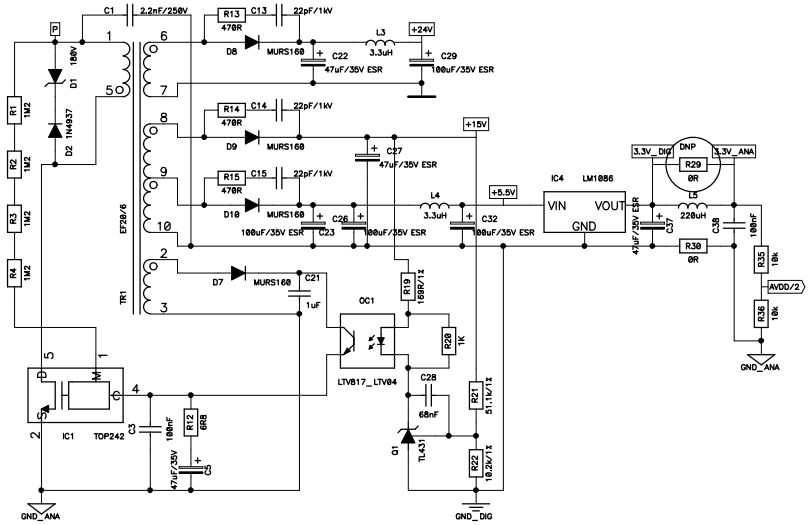
```
193 // reference is defined by shunt resistor value and differential amplifier
194 // Use this equation to calculate maximum torque in Amperes:
195 //
196 // Max REFINAMPS = (VDD/2)/(RSHUNT*DIFFAMPGAIN)
197 //
198 // For example:
199 //
200 // RSHUNT = 0.005
201 // VDD = 3.3
202 // DIFFAMPGAIN = 75
203 //
204 // Maximum torque reference in Amps is:
205 //
206 // (3.3/2)/(.005*75) = 4.4 Amperes, or REFINAMPS(4.4)
207 //
208 // in order to have field weakening, this reference value should be negati
209 // so maximum value in this example is -4.4, or REFINAMPS(-4.4)
210
211 //****Values for Field weakening used to test Sander Motor at 160VDC input
212 // For other motors, FW was not used to reach the rated speed
213 #define dqKFw0 REFINAMPS(0) //3000
214 #define dqKFw1 REFINAMPS(-0.650) //3166
215 #define dqKFw2 REFINAMPS(-0.915) //3333
216 #define dqKFw3 REFINAMPS(-1.075) //3500
217 #define dqKFw4 REFINAMPS(-1.453) //3666
218 #define dqKFw5 REFINAMPS(-1.732) //3833
219 #define dqKFw6 REFINAMPS(-1.970) //4000
220 #define dqKFw7 REFINAMPS(-2.038) //4166
221 #define dqKFw8 REFINAMPS(-2.152) //4333
222 #define dqKFw9 REFINAMPS(-2.242) //4500
223 #define dqKFw10 REFINAMPS(-2.364) //4666
224 #define dqKFw11 REFINAMPS(-2.40) //4833
225 #define dqKFw12 REFINAMPS(-2.425) //5000
226 #define dqKFw13 REFINAMPS(-2.45) //5166
227 #define dqKFw14 REFINAMPS(-2.475) //5333
228 #define dqKFw15 REFINAMPS(-2.5) //5500
229
230
231
232
233
234 //*************** Derived Parameters ****************
235 // parametros "autocalculados", no editar
236
237 #define DPLL        (unsigned int)(2.0*DESIREDMIPS/PLLIN)   // PLL ratio
238 #define FOSC        (PLLIN*DPLL)    // Clock frequency (Hz)
239 #define DFCY        (FOSC/2)        // Instruction cycle frequency (Hz)
240 #define DTCY        (1.0/DFCY)      // Instruction cycle period (sec)
```

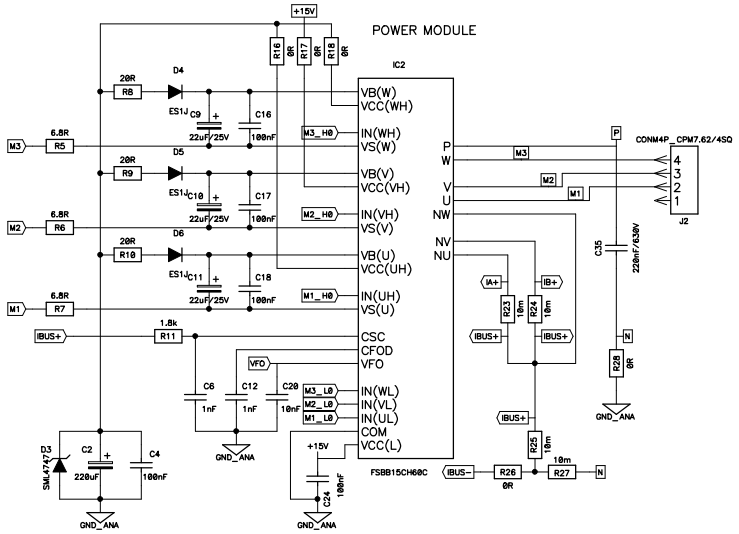
```
241 #define DDEADTIME   (unsigned int)(DEADTIMESEC*DFCY)    // Dead time in dT
242 #define LOOPTIMEINSEC (1.0/PWMFREQUENCY) // PWM Period = 1.0 / PWMFREQUENC
243 #define IRP_PERCALC (unsigned int)(SPEEDLOOPTIME/LOOPTIMEINSEC) // PWM loo
244 // PWM loops necessary for transitioning from open loop to closed loop
245 #define TRANSITION_STEPS   IRP_PERCALC/4
246
247 #define SPEEDLOOPTIME (float)(1.0/SPEEDLOOPFREQ) // Speed Control Period
248 #define LOOPINTCY   (unsigned int)(LOOPTIMEINSEC/DTCY)   // Basic loop per
249 #define LOCKTIME    (unsigned int)(LOCKTIMEINSEC*(1.0/LOOPTIMEINSEC))
250 // Time it takes to ramp from zero to MINSPEEDINRPM. Time represented in s
251 #define DELTA_STARTUP_RAMP  (unsigned int)(MINSPEEDINRPM*POLEPAIRS*LOOPTIM
252                             LOOPTIMEINSEC*65536*65536/(60*OPENLOOPTIMEINSE
253 // Number of control loops that must execute before the button routine is
254 #define BUTPOLLOOPCNT   (unsigned int)(BUTPOLLOOPTIME/LOOPTIMEINSEC)
255
256 // This pre-processor condition will generate an error if maximum speed is
257 // range on Q15 when calculating Omega.
258 #if (FIELDWEAKSPEEDRPM < NOMINALSPEEDINRPM)
259     #error FIELDWEAKSPEEDRPM must be greater than NOMINALSPEEDINRPM for fi
260     #error if application does not require Field Weakening, set FIELDWEAKS
261     #error equal to NOMINALSPEEDINRPM
262 #else
263     #if ((FIELDWEAKSPEEDRPM*POLEPAIRS*2/(60*SPEEDLOOPFREQ)) >= 1)
264         #error FIELDWEAKSPEEDRPM will generate an Omega value greater thar
265         #error maximum in Q15 format. Reduce FIELDWEAKSPEEDRPM value, or i
266         #error control loop frequency, SPEEDLOOPFREQ
267     #endif
268 #endif
269
270 #ifdef PWM_DT_ERRATA
271     #define MIN_DUTY  (unsigned int)(DDEADTIME/2 + 1)        // Should be
272 #else
273     #define MIN_DUTY  0x00
274 #endif
275
276
277 #endif
278
```

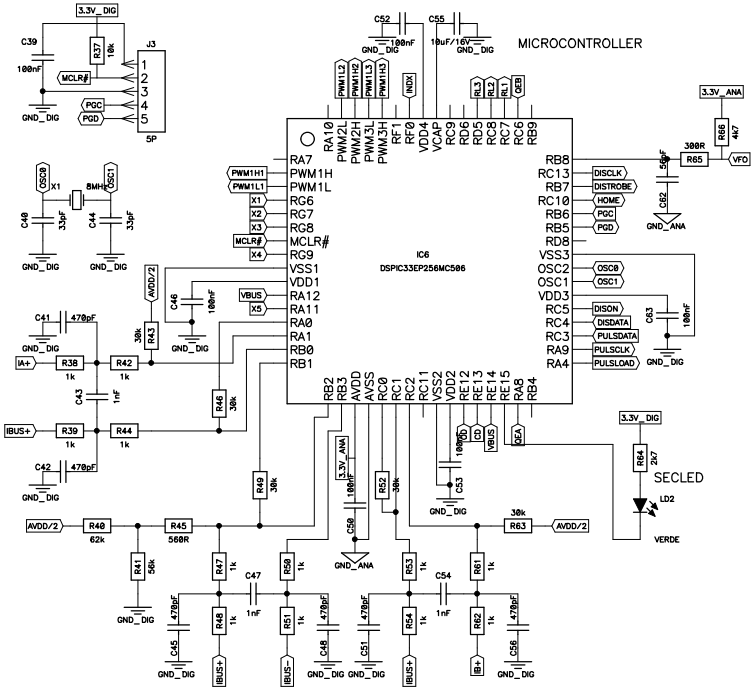
Anexo 2

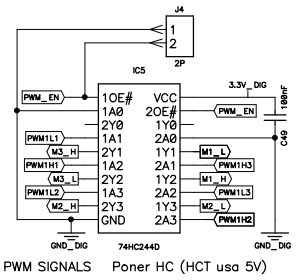
diseño del hardware



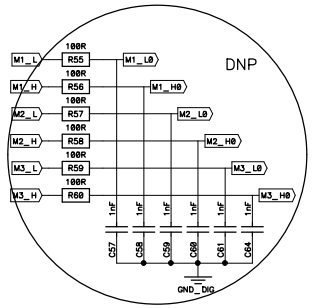




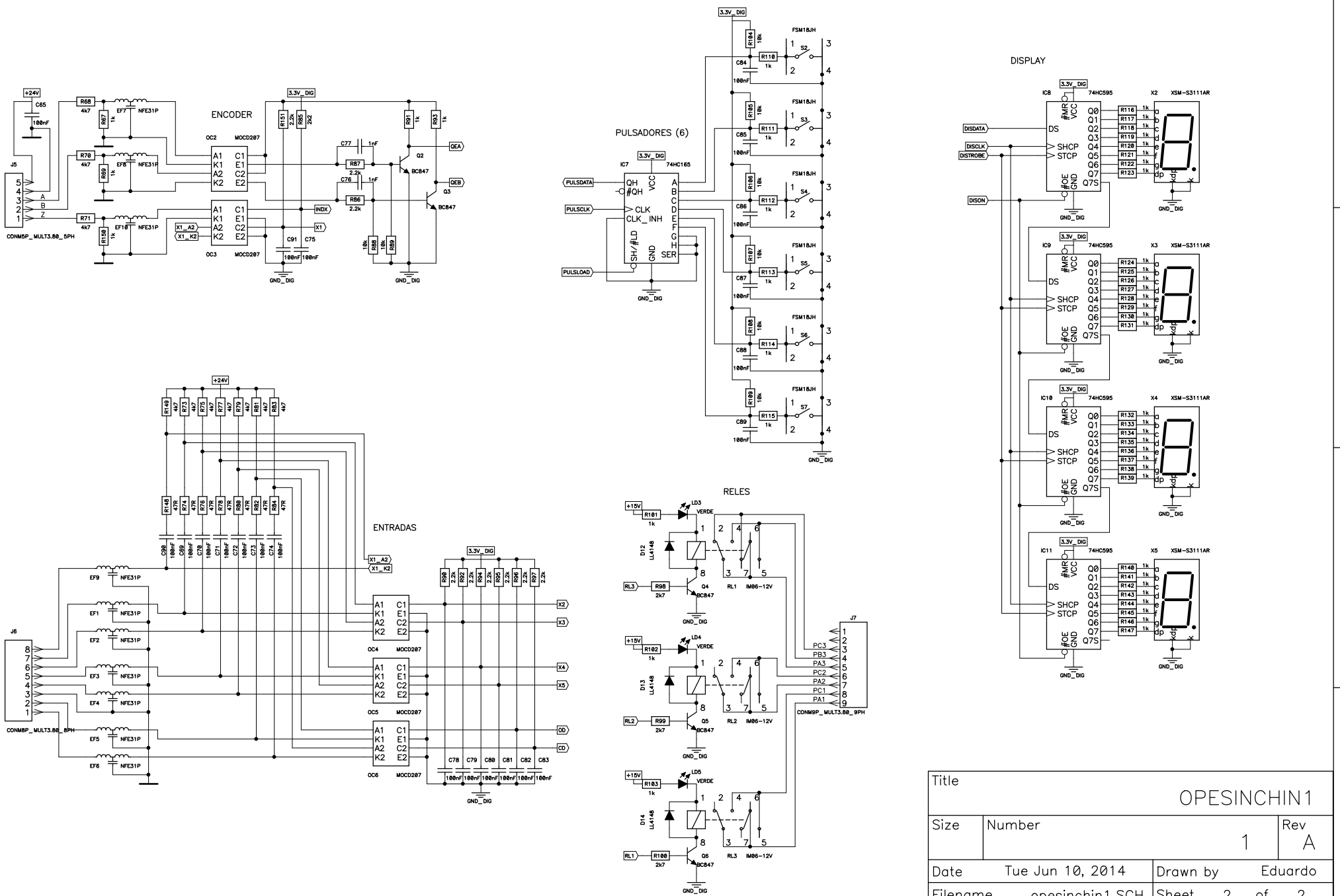




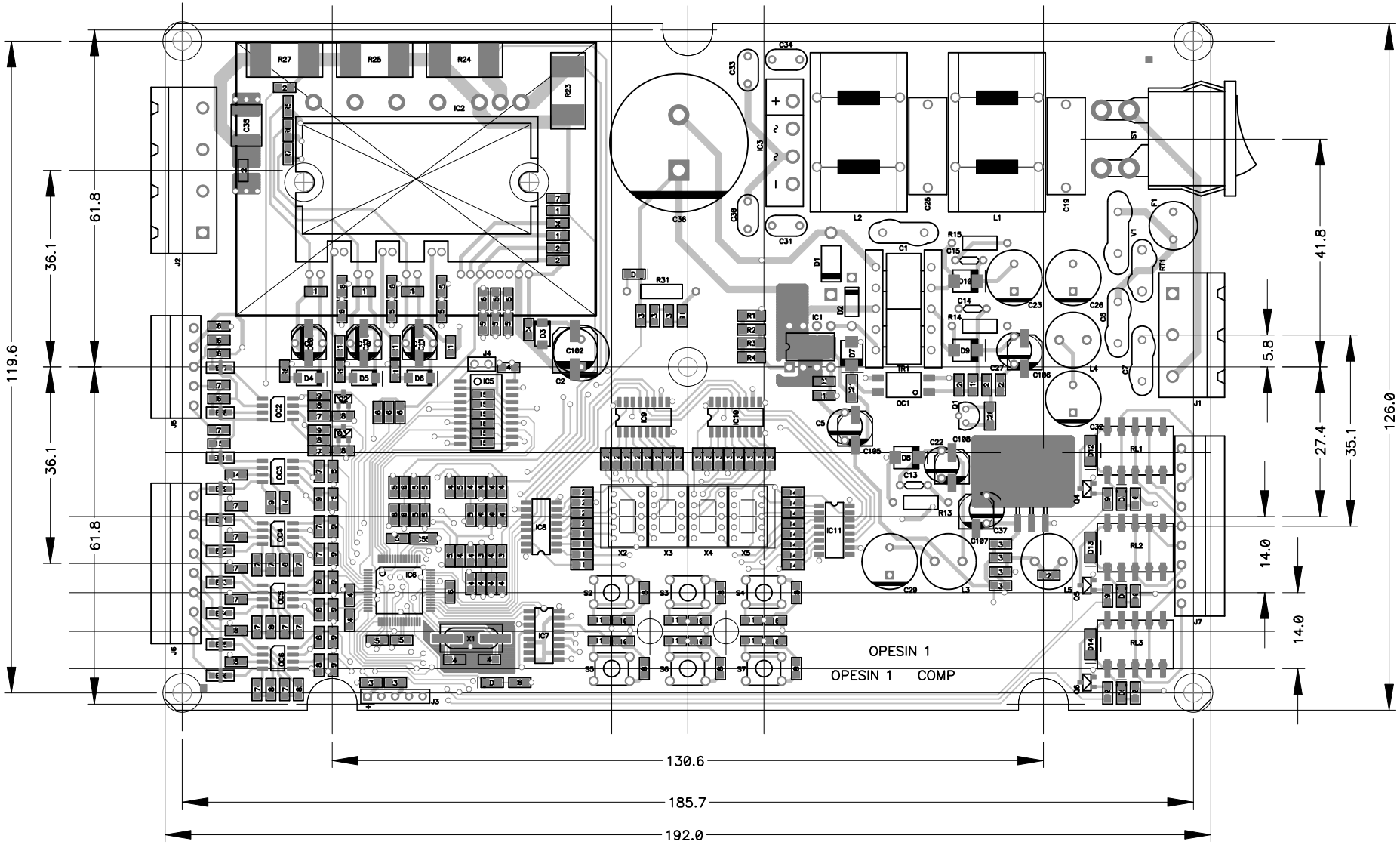
PWM SIGNALS Poner HC (HCT usa 5V)



Title	OPESINCHIN1		
Size	Number 1		Rev A
Date	Tue Jun 10, 2014	Drawn by	Eduardo
Filename	opesinchin1.SCH	Sheet 1 of 2	

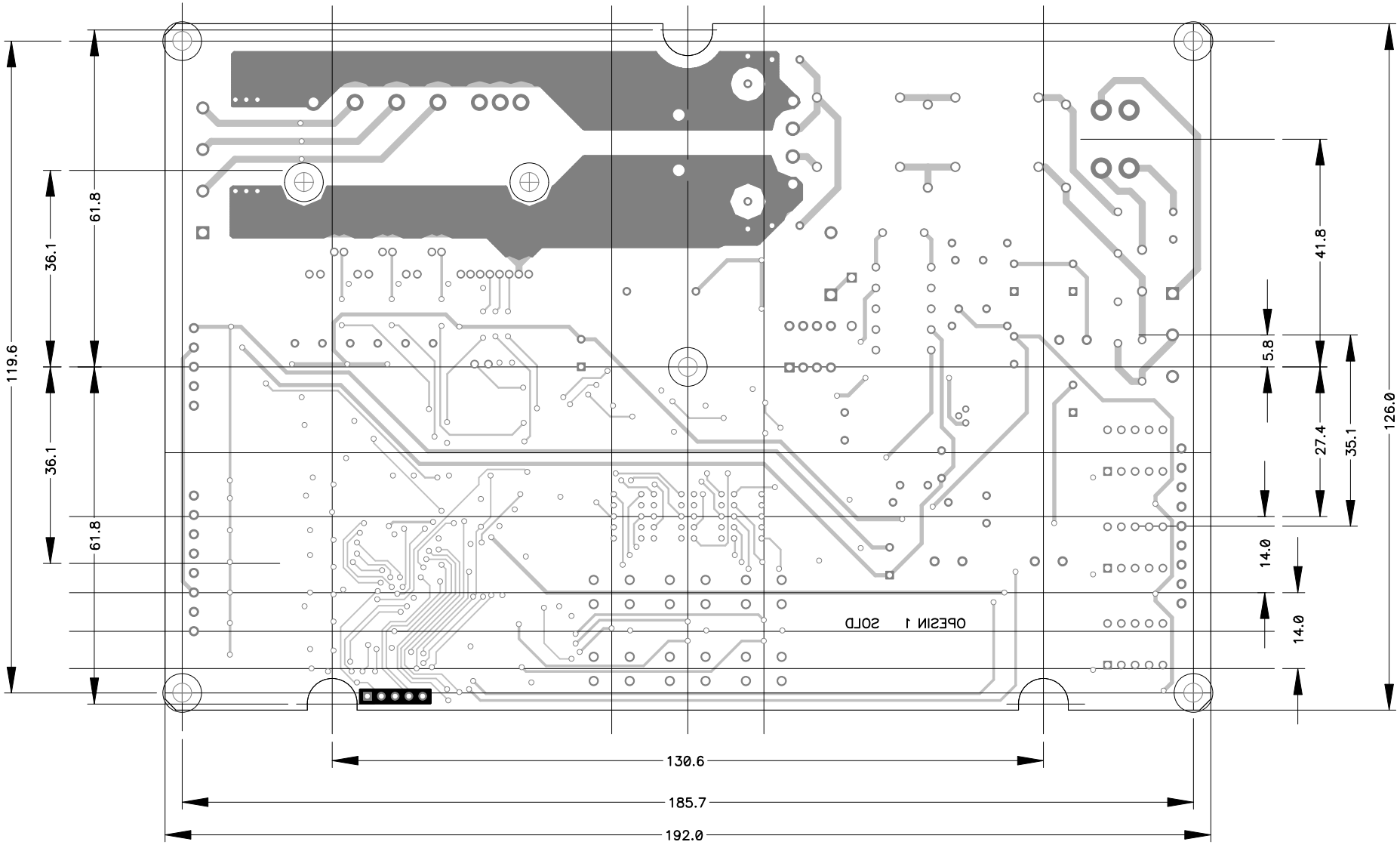


Title	OPESINCHIN1		
Size	Number 1		Rev A
Date	Tue Jun 10, 2014	Drawn by	Eduardo
Filename	opesinchin1.SCH	Sheet	2 of 2



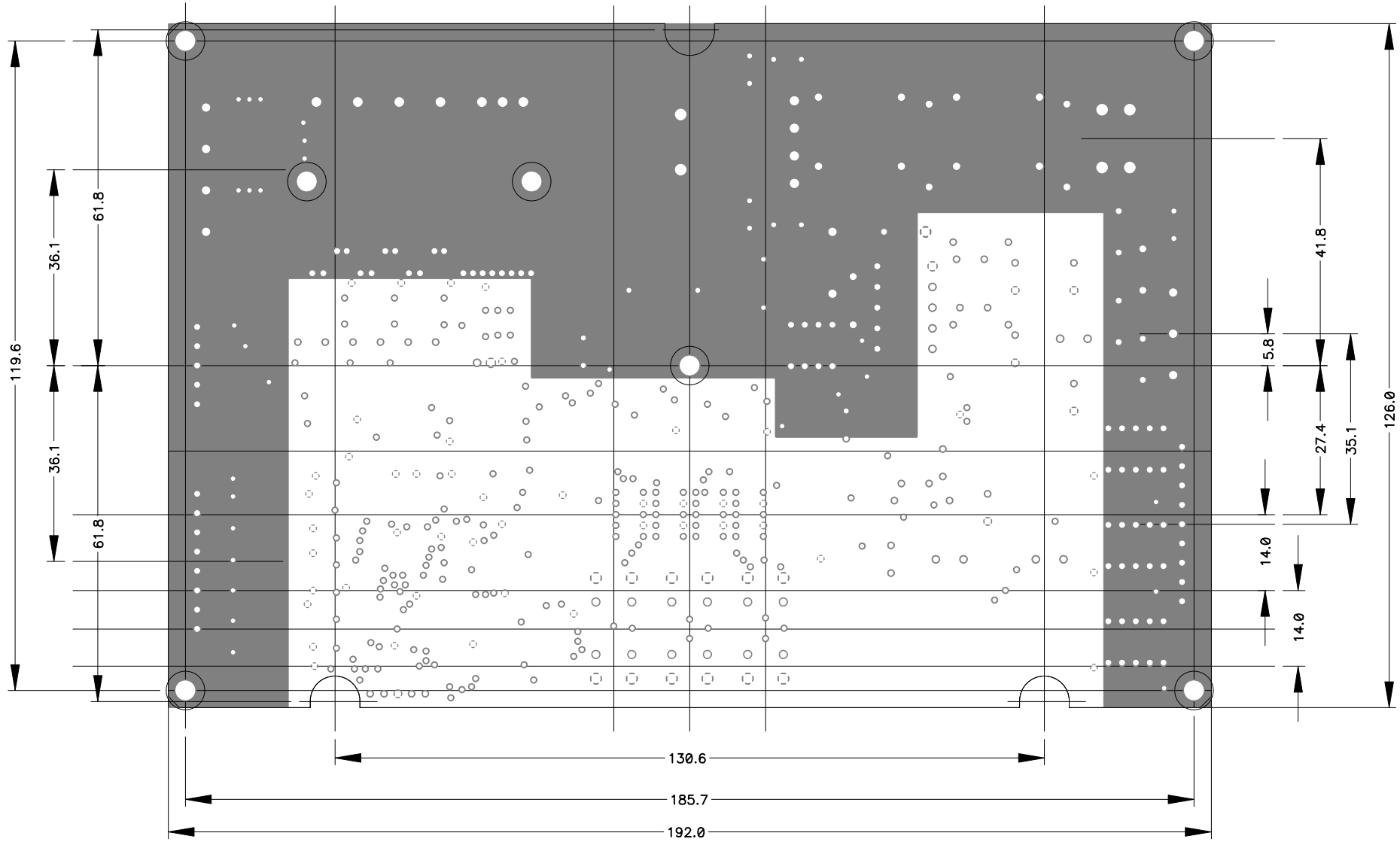
TOP LAYER

Title	PCB OPESINCHIN1		
Size	Number 1		Rev A
Date	Mon Jun 16, 2014	Drawn by	Eduardo
Filename	opesinchin1.PCB	Sheet	of



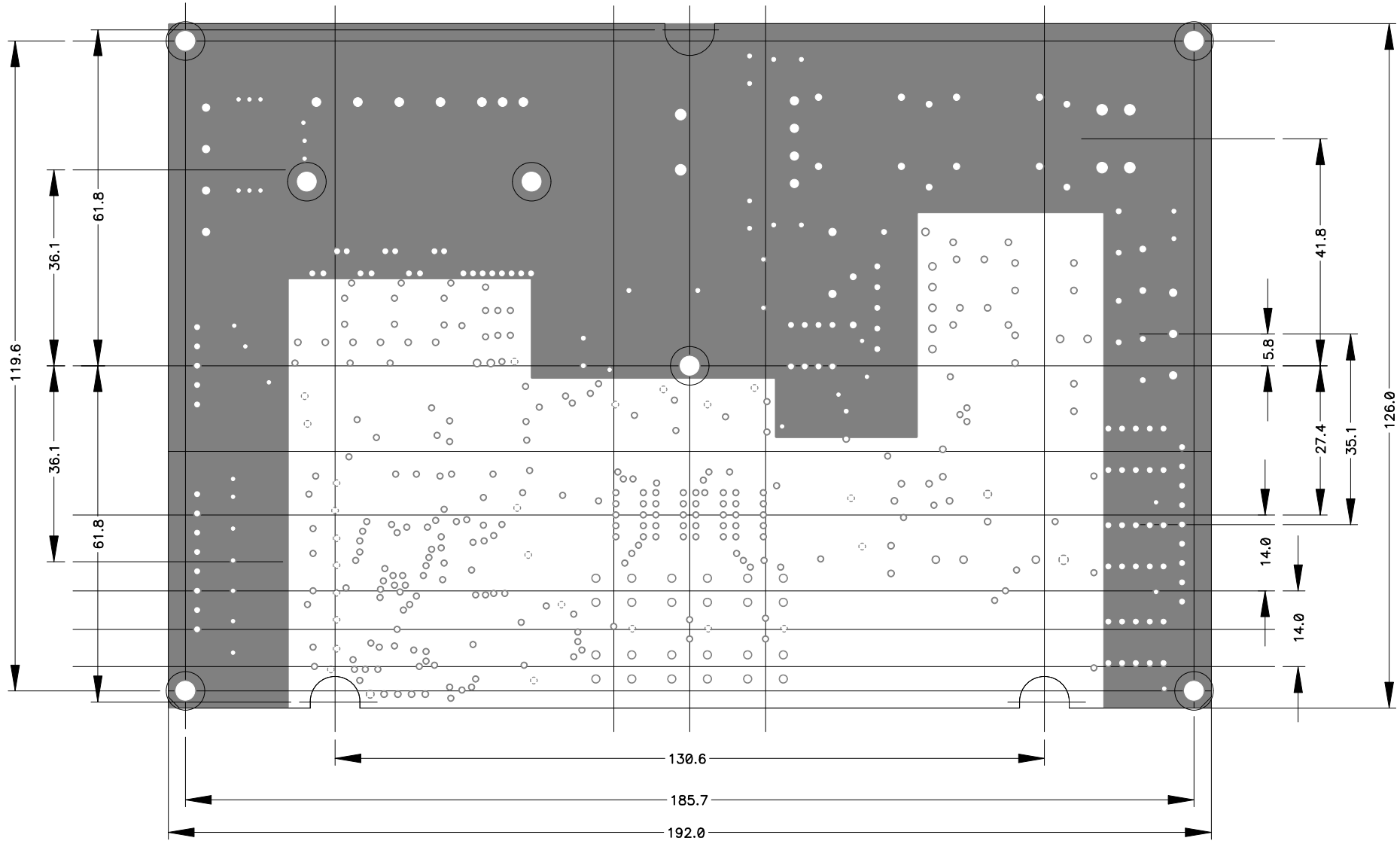
BOTTOM LAYER

Title	PCB OPESINCHIN1		
Size	Number 1		Rev A
Date	Mon Jun 16, 2014	Drawn by	Eduardo
Filename	opesinchin1.PCB	Sheet	of



GND LAYER

Title	PCB OPESINCHIN1		
Size	Number 1		Rev A
Date	Mon Jun 16, 2014	Drawn by	Eduardo
Filename	opesinchin1.PCB	Sheet	of



POWER LAYER

Title	PCB OPESINCHIN1	
Size	Number 1	Rev A
Date Mon Jun 16, 2014	Drawn by Eduardo	
Filename opesinchin1.PCB	Sheet of	

Nº: I0000023

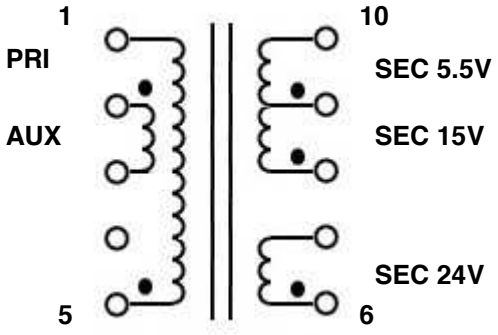
FECHA: 03-06-14

PRODUCTO: Operador síncrono.

TÍTULO: Transformador de la fuente conmutada. Versión 3.

DESCRIPCIÓN:

Visto desde arriba (Top View)



Notas:

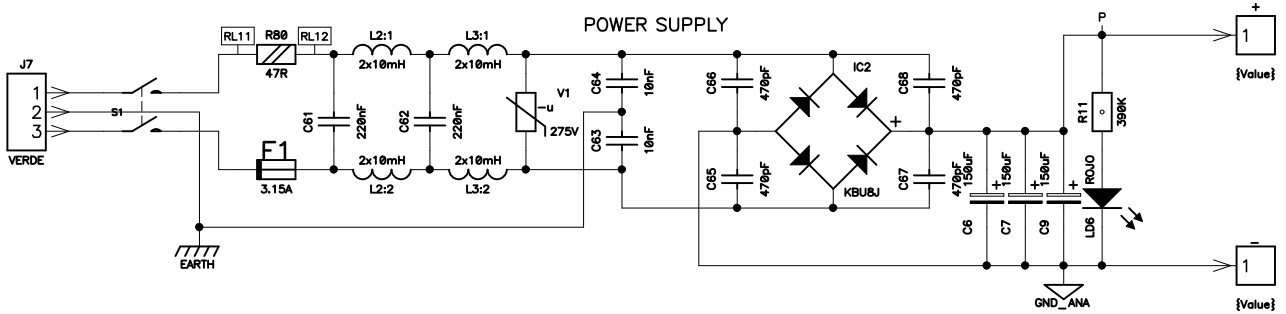
- Pin 4 cortado.
- Núcleo EF20.
- Material núcleo 3C85, 3C94, N27 o similar.
- Carrete 10 pines vertical.
- $L_{PRIMARIA}$ = 1.8mH
- Rigidez dieléctrica 3038 VAC, entre cualquier pin del primario, y cualquier pin del secundario.

Orden de bobinados:

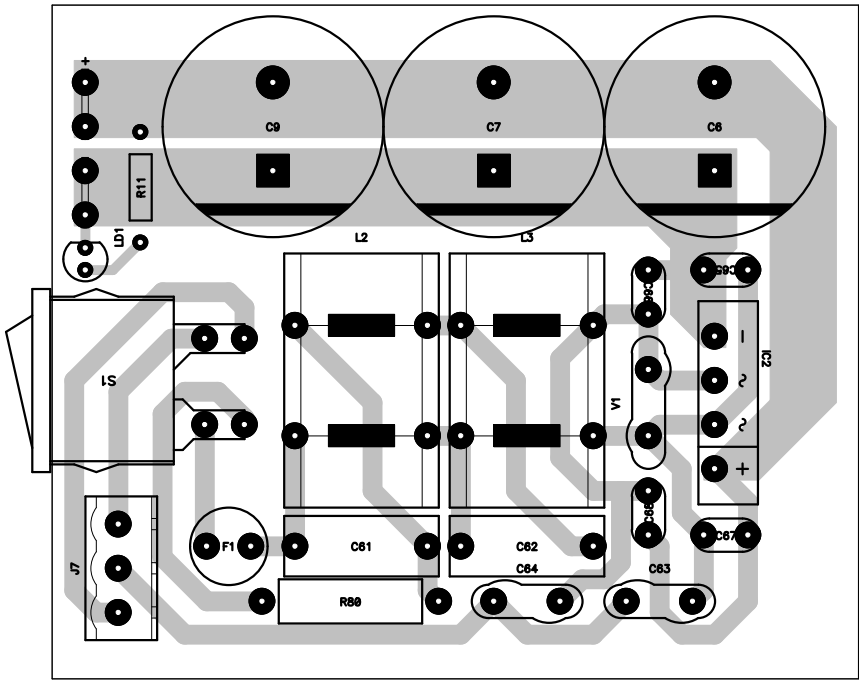
ORDEN		
5º	$N_{6\text{-}7}$=19 espiras; 1 hilo de Ø 0.25mm	SEC 24V
4º	$N_{8\text{-}9}$=7 espiras; 1 hilo de Ø 0.25mm	SEC 15V
3º	$N_{9\text{-}10}$=5 espiras; 1 hilo de Ø 0.25mm	SEC 5.5V
2º	$N_{2\text{-}3}$=10 espiras; 1 hilo de Ø 0.25mm	AUX
1º	$N_{5\text{-}1}$=83 espiras; 1 hilo de Ø 0.25mm	PR1

Anexo 3

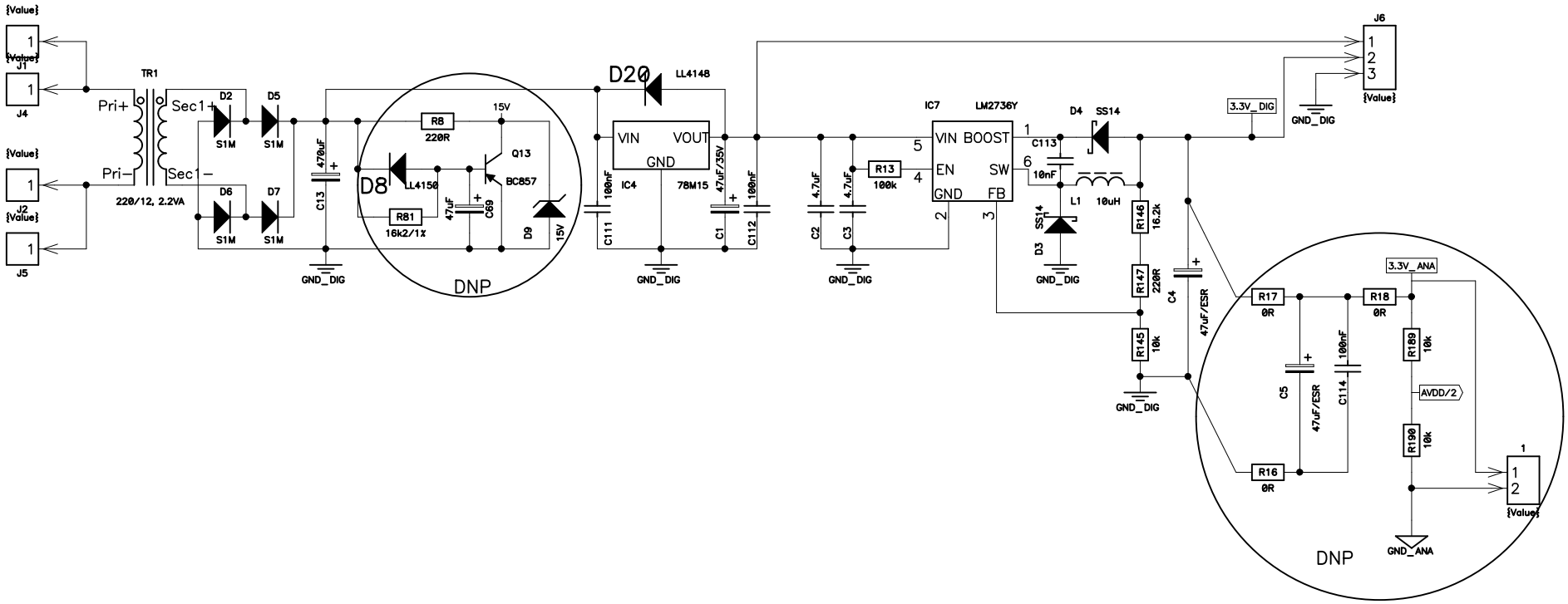
prototipos del hardware



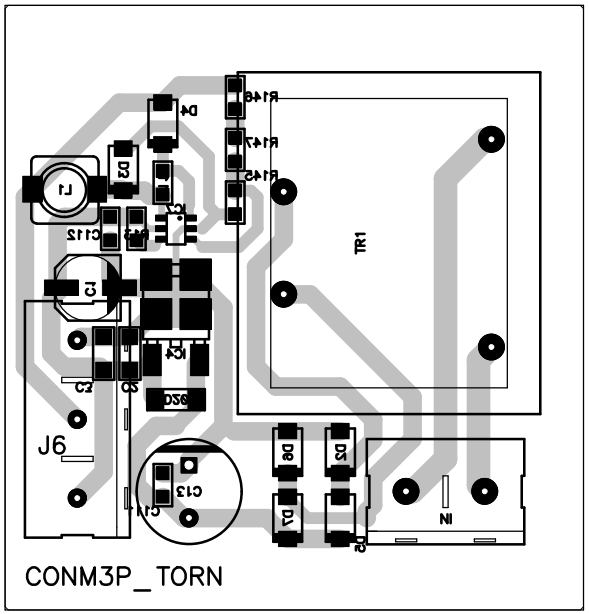
Title	Rectif. 230AC – 325DC		
Size	Number 1		Rev 1
Date	Mon Jun 09, 2014	Drawn by	Inigo
Filename	rectificador.SCH	Sheet	1 of 1



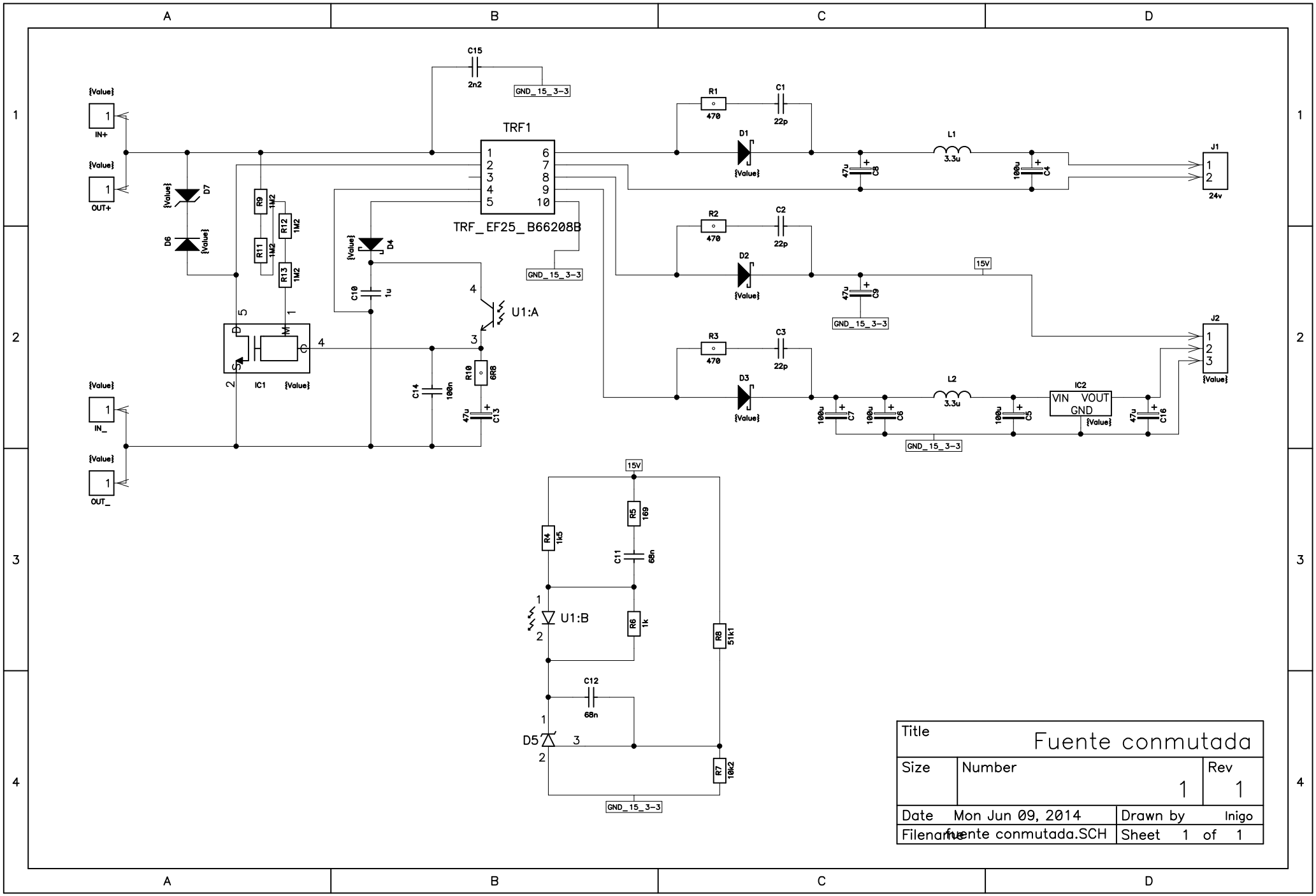
Title: top Rectif. 230AC – 325DC		
Size	Number: 1	Rev: 1
Date: Tue Jun 10, 2014		Drawn by: Inigo
Filename: rectificador.PCB		Sheet: of



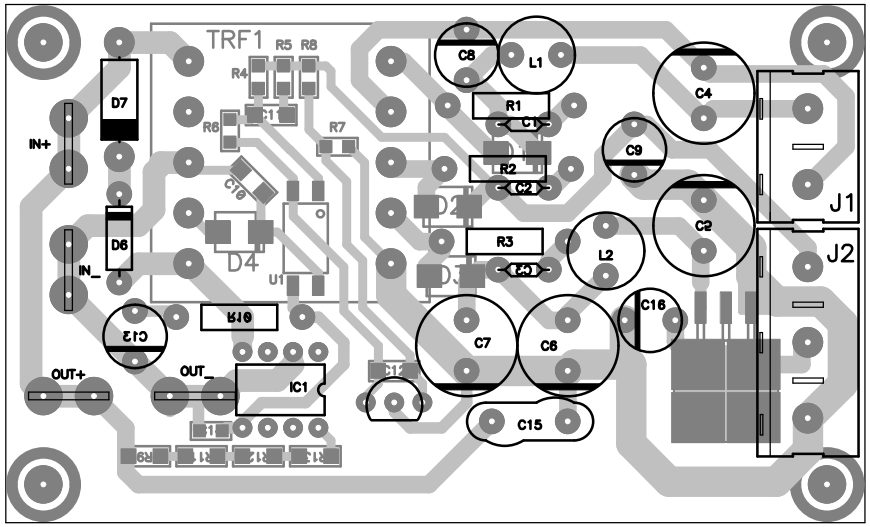
Title	Reg. 15V – 3.3V	
Size	Number 1	Rev 1
Date Mon Jun 09, 2014	Drawn by Inigo	
Filename reguladores.SCH	Sheet 1 of 1	



Title	Reg. 15V – 3.3V		
Size	Number 1	Rev 1	
Date	Tue Jun 10, 2014	Drawn by	Inigo
File	reguladores imprimir.PCB	Sheet	of



Title	Fuente conmutada	
Size	Number 1	Rev 1
Date	Mon Jun 09, 2014	Drawn by Inigo
Filename	fuente conmutada.SCH	Sheet 1 of 1



Title	Fuente conmutada		
Size	Number 1		Rev 1
Date	Tue Jun 10, 2014	Drawn by	Inigo
File	fuente conmutada 2.PCB	Sheet	of