Proyecto Final de Carrera

Implementación de un Regulador PID en un dsPIC33F

Alumno: Francisco Salavert Torres

Director: Pascual Pérez Blasco

Ingeniería Técnica en Informática de sistemas

15 de diciembre de 2010

Índice general

1. Introducción **7**

1.1. Visión general 7

1.2. Motivación 9

1.3. Objetivos 12

1.4. Organización de los contenidos 13

2. Regulador PID **15**

2.1. Término Proporcional 17

2.2. Término Integral 18

2.3. Término Derivativo 19

2.4. Ecuación del regulador 19

2.5. Ecuación del regulador discreta 20

3. Motor de corriente continua **21**

3.1. Modificar la velocidad de giro, PWM 22

3.2. Posicion y velocidad, encoders 23

3.2.1. Control de velocidad 24

3.2.2. Control de posición 24

3.3. Sentido de giro, puente H 25

4. Implementación **27**

4.1. Microcontroladores 28

4.2. dsPIC 29
4.3. Esquema del montaje y simulación 30
4.4. Programación en C del algoritmo PID 32
4.4.1. Regulador PID de Posición en C 32
4.4.2. Regulador PID de Velocidad en C 34
4.5. Programación usando las librerías del dsPIC 36
4.5.1. Funciones PID 36
4.5.2. Funciones implementadas en ensamblador 39
4.5.3. Ejemplo de uso de las funciones PID 43
4.5.4. Regulador PID de Posición con librerias 45
4.5.5. Regulador PID de Velocidad con librerias 46

5. Tareas, FreeRTOS **49**
5.1. Implementación de tareas 50
5.2. Tiempo absoluto 52
5.3. Ejemplo de implementación de una tarea 53

Apéndices **55**

A. Configuración del módulo PWM **57**
A.1. Registros de configuración 57
A.1.1. P1TCON 58
A.1.2. P1TPER 58
A.1.3. PWM1CON1 58
A.1.4. P1DTCON1 y P1DTCON2 58
A.1.5. P1DC1 58
A.1.6. P1OVDCON 59
A.2. Tiempo muerto 59
A.2.1. Ejemplo de inicialización 60

B. Configuración de módulo QEI **61**

B.1. Registros de configuración 62

B.1.1. QEI1CON 62

B.1.2. DFLT1CON 63

B.1.3. POS1CNT 63

B.1.4. MAX1CNT 63

B.1.5. Ejemplo de inicialización 64

C. Configuración de módulo ADC **65**

C.1. Funcionamiento 65

C.2. Registros de configuración 66

C.2.1. AD1CON1 66

C.2.2. AD1CON2 66

C.2.3. AD1CON3 66

C.2.4. AD1PCFGL 67

C.2.5. AD1CHS0 y AD1CHS123 67

C.3. Ejemplo de inicialización 68

I. main.c **71**

Capítulo 1

Introducción

Hoy en día todo es automático, por ejemplo un calefactor o aire acondicionado, sólo con elegir la temperatura, enfría o calienta una habitación. Los vehículos actuales son capaces de mantener una velocidad establecida por el conductor sin tener que presionar el pedal del acelerador, todo ello es posible con la ayuda de reguladores y la teoría de control.

1.1. Visión general

Los reguladores permiten respuestas muy rápidas ante cambios en el sistema, lo cual aumenta la precisión de éstos. Algunos sistemas no necesitan tal velocidad como por ejemplo el termostato de un aire acondicionado, ya que transcurren varios segundos hasta que la temperatura varía, una persona seria capaz de encender o apagar el aire acondicionado a gusto, como si se tratase de un simple ventilador, que se enciende si hace calor y se apaga cuando ya no lo hace, es mucho más cómodo que se encienda y se apague

automáticamente para no tener que estar pendiente de la temperatura. Estos sistemas no necesitan altos tiempos de respuesta sin embargo otros muchos sistemas no podrían ser controlados si no es mediante reguladores, la velocidad de respuesta es crítica y serían imposibles de controlar, por ejemplo el antibloqueo de frenos en un coche ABS, que necesita ser activado en el momento preciso en que las ruedas se bloquean para evitar que el coche se deslice sobre el asfalto.

La teoría de control permite estudiar y tratar sistemas dinámicos. El control digital es la implementación de un regulador mediante un algoritmo que se ejecuta en un computador. Este regulador proporciona las señales de control necesarias para que el sistema o proceso tenga un comportamiento determinado.

Esta clase de sistemas están gobernados por un regulador que decide como actuar. Para ello deben conocer a donde quieren llegar, por ejemplo una temperatura o una velocidad, y donde están, es decir, cual es la temperatura o la velocidad actual.

Un sistema controlado está en bucle cerrado, de forma que la salida del sistema se compara con la referencia y cuando la diferencia entre la referencia y la salida es cero el sistema está en el estado deseado. El diagrama de bloques correspondiente a este sistema de control se muestra en la figura 1.1.

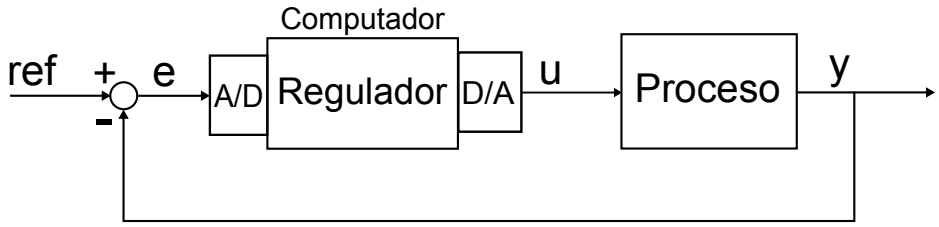


Figura 1.1: Sistema controlado por un regulador discreto.

La implementación en un computador necesita que las señales sean discretas o digitales, para ello se usan convertidores analógico/digital, los cuales muestrean la señal y toman valores para distintos instantes de tiempo. Estos valores discretos son procesados por el computador uno a uno, a continuación a partir de los nuevos valores crea la señal de control que le debe llegar al proceso y se conviertiendola en continua o analógica mediante convertidores digital/analógico. Esto se debe a que los computadores deben operar con valores concretos y de una precisión determinada.

Aunque este proceso sea mas complejo que el procesamiento analógico y tenga un rango de valores discretos, el procesamiento digital ofrece detección y corrección de errores y es menos vulnerable a ruido, lo cual es una ventaja en muchas, aunque no todas, las aplicaciones.

1.2. Motivación

Los principales motivos que han llevado a la realización de este proyecto consisten en el interés por conocer mejor los sistemas empotrados y sus aplicaciones. Durante la carrera de se ha tocado este tema principalmente de

forma teórica, ya entonces aparece cierta curiosidad acerca de como trabajar y programar esta clase de procesadores, que al parecer están en toda clase dispositivos actuales como reproductores MP3 ó teléfonos móviles.

Las primeras características conocidas acerca de estos microcontroladores son su reducido tamaño y gran capacidad de integración, su potencia parece menor con respecto a los procesadores de propósito general sin embargo lo que en principio parece negativo es visto desde otro modo incluso una ventaja. Los procesadores convencionales que se encuentra en cualquier ordenador de uso general están destinados a realizar una gran cantidad de variadas tareas sin embargo los microcontroladores son usados para aplicaciones específicas y la elección entre los distintos tipos de microcontroladores es importante.

Si con un procesador lento y barato se puede realizar cierta tarea, no es necesario usar otros más potentes que consumen más energía. Éste factor es clave a la hora de trabajar con microcontroladores.

Dentro de los distintos tipos de microcontroladores destacan los procesadores digitales de señal DSP, específicamente diseñados para el procesado digital de señal, su meta es generalmente medir o filtrar señales analógicas del mundo real, para ello disponen de instrucciones especificas para el muestreo y cálculo de éstas. La arquitectura de los DSP les permite ejecutar instrucciones multiplicación y acumulación en un sólo flanco de reloj, lo cual con menos frecuencia de trabajo son capaces de realizar muchísimas mas operaciones de este tipo que un procesador de propósito general con mucha mas frecuencia de trabajo, ya que, necesitan varios flancos de reloj para completar este tipo de operaciones.

En una asignatura del último curso de la ingeniería, aparece la posibilidad de realizar pequeños programas con estos dispositivos y es cuando

continuando con la materia introductoria en la programación de estos microcontroladores aumenta el interés en la implementación de aplicaciones algo más grandes.

Inicialmente la propuesta de realizar un péndulo invertido parece seductora simplemente por el hecho de conocer y entender éste sistema ya implementado y comercializado por una empresa que fabrica un vehículo basado en este ejercicio de control.

Una parte de esta aplicación es el control y regulación de un motor de corriente continua. El control incluye establecer una velocidad y desplazarse cierta distancia mediante un controlador PID.

Es por eso que resulta interesante implementar el algoritmo de control en un microcontrolador actual, en concreto la serie dsPIC33 de microchip el cual aporta una gran cantidad de prestaciones destinadas al control de motores. El integrado posee módulos como el QEI que es capaz de leer las señales procedentes de los encoders conectados al motor los cuales permiten obtener la posición, velocidad en incluso la aceleración. También contiene un modulo para generar PWM el cual con ayuda de un puente H permite controlar la velocidad y el sentido de giro.

Poner en marcha todo esto implica estudiar y entender la forma de trabajar de los microcontroladores además de aprender a manejar las herramientas de desarrollo de aplicaciones correspondientes. Al ser un sistema empotrado también se estudiará un kernel de tiempo real especialmente diseñado para estos procesadores, en concreto FreeRTOS, sobre el cual se implementarán las tareas de control necesarias facilitando así la gestión de tiempos.

1.3. Objetivos

Se pretende implementar un algoritmo de control PID en un microcontrolador con el fin de gobernar un motor de corriente continua, el trabajo se desarrollará en MPLAB IDE para programar en C el algoritmo de control PID, lo cual implica configurar y poner en marcha el módulo PWM y el módulo QEI. Implantar los algoritmos de control en tareas del kernel FreeRTOS y poner en marcha el sistema.

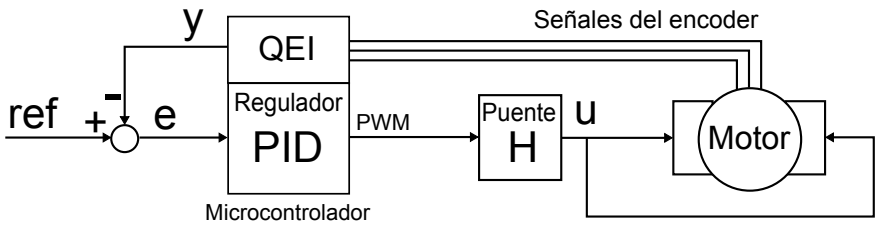


Figura 1.2: Sistema de control de un motor de continua.

Para implementar el control del sistema que se pretende realizar, se usará como computador un microcontrolador, en concreto un DSP y como sistema o proceso, un motor de corriente continua con un encoder. La referencia es una posición a la cual se quiere llegar o la velocidad que se quiere alcanzar. La salida y es la posición la velocidad obtenida y calculada mediante el módulo QEI que posee el DSP usando de las señales proporcionadas por el encoder del motor. La acción de control u proporcionada por el regulador al proceso es el ciclo de trabajo de la señal PWM y el sentido, mediante el puente H, en que debe de accionarse el motor. El error e se obtiene de la resta de la referencia ref con la salida y, este error es proporcionado al regulador el cual calculará la acción de control correspondiente. El diagrama de bloques resultante se observa en la figura 1.2.

1.4. Organización de los contenidos

En el **capítulo 1**, se realiza una introducción a las tareas desarrolladas en el proyecto fin de carrera. En el **capítulo 2**, se explica que es y como funciona un regulador PID para poder ser implementado en un computador. En el **capítulo 3**, se explica brevemente que es un motor de corriente continua y como controlar su funcionamiento. En el **capítulo 4**, se explican algunos detalles de la implementación en microcontroladores, además de la implementación del regulador PID de dos formas distintas: en C y usando librerías específicas para un dsPIC.

En los apéndices se explica más concretamente la implementación del regulador PID usando el kernel freeRTOS, también se explica como inicializar los módulos del dsPIC usados para mover y obtener los valores de posición y velocidad del motor de corriente continua.

Capítulo 2

Regulador PID

Un regulador o controlador PID (proporcional, integral, derivativo) es un mecanismo de control retroalimentado en bucle cerrado ampliamente usado en sistemas de control industrial. Un PID calcula un error como la diferencia entre el valor actual del sistema y el valor al que se desea llegar. Además intenta minimizar el error mediante el ajuste de las entradas del proceso.

En el calculo del regulador PID intervienen tres parámetros distintos. Éstos valores se pueden interpretar en función del tiempo. El proporcional P depende del error actual, el integral I depende de la suma de todos los errores pasados, y el derivativo D es la predicción de errores futuros, basándose en la tasa de cambio actual. La suma ponderada de los tres términos permiten ajustar un proceso mediante un elemento de control como por ejemplo la velocidad que debe alcanzar un motor de corriente continua.

$$U = P + I + D \tag{2.1}$$

Mediante el ajuste de estas tres constantes en el algoritmo de control PID, el regulador es capaz de proporcionar acciones de control específicas a los requerimientos de un sistema. La respuesta del regulador se puede describir como la capacidad de respuesta ante un error, el grado en que el regulador llega más allá del punto de ajuste, y la oscilación del sistema.

El ajuste de las constantes se logra mediante el análisis del sistema se puede llegar a ajustar el tiempo de respuesta, que es cuanto tiempo tarda el regulador en llevar al sistema a cierto estado. También se puede ajustar que el sistema llegue sin errores de posición y minimizar las sobreoscilaciones alrededor de la referencia o estado deseado.

Sin embargo esto no siempre se puede ajustar perfectamente ya que si se quiere una respuesta muy rápida las oscilaciones son inevitables ya que el sistema no puede responder correctamente en tan poco tiempo, un ejemplo de esto es querer que un motor alcanze una determinada posición y que debido a la inercia se pase de la referencia y oscile por arriba y por debajo hasta llegar a la posición determinada por la referencia. Es claramente un efecto negativo ya que por intentar aumentar el tiempo de respuesta se ha provocado una sobreoscilación y como consecuencia se ha relentizado la respuesta.

Es importante mencionar que el uso de un regulador PID no garantiza un control óptimo ni estabilidad al sistema. Hay sistemas que son inestables. Un sistema es inestable si el valor regulado oscila indefinidamente, o sí dicho valor diverge sin limite del estado estable.

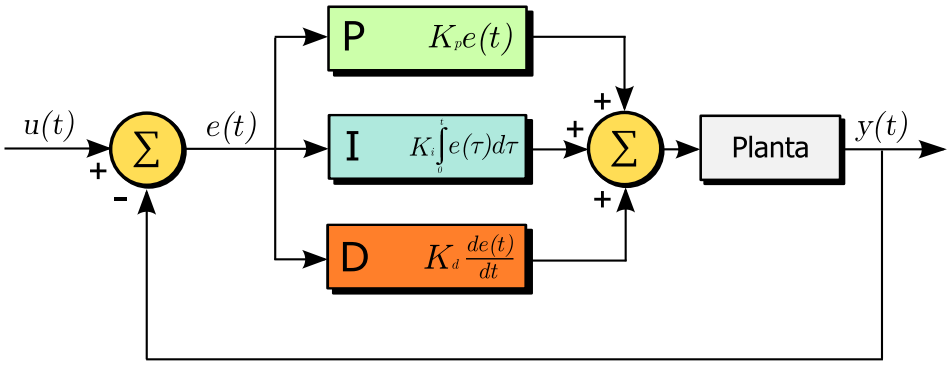


Figura 2.1: Diagrama de bloques de un Regulador PID.

Algunas aplicaciones sólo requieren el uso de uno o dos parámetros para proporcionar un control apropiado al sistema, para no usar un parámetro simplemente hay que poner la ganancia del parámetro a cero. Dependiendo de los parámetros activos, los reguladores se pueden llamar PI, PD, P o I. Los reguladores PI son muy comunes ya que la accion derivativa D es sensible al ruido. Por otro lado la ausencia de la acción integral puede evitar que el sistema llegue al valor deseado.

2.1. Término Proporcional

El término proporcional modifica la salida proporcionalmente con el error actual. La respuesta proporcional se puede ajustar multiplicando el error por una constante K_p, conocida como ganancia proporcional.

$$P = K_p e_{(t)} \tag{2.2}$$

Donde u es la salida y e el error en un instante de tiempo, de este modo cuanto más grande sea el error más rápida será la respuesta, y conforme el

error se reduzca, la respuesta será más pequeña, hasta que finalmente el error sea cero y la respuesta nula ya que el valor deseado es el mismo que el valor actual.

Un controlador proporcional no siempre alcanzará su valor objetivo, conservando un error de estado estacionario. Para eliminar este error hay que usar el término integral.

2.2. Término Integral

El término integral es proporcional a la magnitud del error y a la duración del error. Es decir, la suma de todos los errores en cada instante de tiempo o como su nombre indica la integración de los errores. Ésta suma compensa la diferencia que debería haber sido corregida anteriormente. El error acumulado se multiplica por la ganancia integral K_i que indicará la cantidad de acción integral respecto de toda la acción de control.

$$I = K_i \int_0^t e_{(\tau)} d\tau \tag{2.3}$$

Al usar el integral junto al proporcional se acelera el movimiento del sistema llegando antes al valor deseado, es decir, menor tiempo de respuesta. Además elimina el error estacionario que se produce al usar un regulador proporcional. Sin embargo el término integral tiene en cuenta los errores acumulados en el pasado y puede hacer que el valor actual sobrepase el valor deseado lo cual creará un error en el otro sentido, creando oscilaciones alrededor del valor deseado.

2.3. Término Derivativo

El término derivativo calcula la variación del error mediante la pendiente del error en cada instante de tiempo, es decir, la primera derivada con respecto al tiempo, y multiplica esa variación del error con la ganancia derivativa K_d, la cual indica la cantidad de acción derivativa respecto de toda la acción de control.

$$D = K_d \frac{d}{dt} e_{(t)} \tag{2.4}$$

El derivativo ralentiza la tasa de cambio de la salida del regulador y cuanto más cerca esta del valor deseado, se hace aún mas lento. Por tanto el uso del derivativo sirve para disminuir las oscilaciones producidas por el control integral y así mejorar la estabilidad del proceso de regulación. Uno de los inconvenientes es que al hacer la derivada, el ruido es amplificado, y hace al regulador más sensible a las interferencias llevando al sistema a la inestabilidad en el peor de lo casos.

2.4. Ecuación del regulador

Los términos proporcional, integral y derivativo son sumandos para hallar la salida del regulador PID, sabiendo que u(t) es tal salida, la ecuación que define al algoritmo es la siguiente:

$$u_{(t)} = K_p e_{(t)} + K_i \int_0^t e_{(\tau)} d\tau + K_d \frac{d}{dt} e_{(t)} \tag{2.5}$$

Donde hay que tener en cuenta las siguientes consecuencias: La ganancia proporcional K_p. Cuanto mayor es, más rápida es la respuesta ya que

cuanto más grande es el error, más grande es la acción proporcional. Valores muy grandes pueden ocasionar oscilaciones o incluso la inestabilidad. La ganancia integral K_i. Valores grandes eliminan más rápidamente los errores estacionarios. Sin embargo puede provocar mayor sobreoscilación. La ganancia derivativa K_d. Cuanto más grande es, más se reduce la sobreoscilación, pero ralentizan el tiempo de respuesta, y pueden llevar al sistema a la inestabilidad debido a la amplificación del ruido en el calculo diferencial del error.

2.5. Ecuación del regulador discreta

Un computador solo es capaz de procesar valores discretos y finitos, así que para poder implementar el algoritmo de control de un regulador PID, es necesario discretizar la ecuación del regulador PID, una vez discretizada se calcula para cada uno de los instantes de tiempo muestreados el valor de la acción de control. Para cada instante de tiempo se vuelve a calcular una nueva acción de control utilizando la nueva salida del sistema, esto permitirá al sistema avanzar hacia el estado deseado marcado por la referencia con cada nuevo cálculo. La ecuación discreta del regulador para poder ser implementada en un computador se puede expresar mediante la siguiente ecuación.

$$u_{(t)} = K_p e_{(t)} + K_i T_s \sum_{k=0}^{t} e_k + K_d \frac{e_{(t)} - e_{(t-1)}}{T_s} \tag{2.6}$$

Donde $e_{(t)}$ es el error de la respuesta del sistema en el instante t, T es periodo de muestreo de la señal y K_p , K_i, y K_d son la ganancia proporcional, integral y derivativa del regulador, respectivamente.

Capítulo 3

Motor de corriente continua

El uso de motores de CC (corriente continua) está muy extendido, hoy en día se pueden encontrar en cualquier dispositivo electromecánico. Controlar la posición y la velocidad de estos motores es muy sencillo convirtiéndose en una buena opción a la hora de crear aplicaciones de automatización y control.

Los motores de corriente continua simplemente necesitan ser conectados a una fuente de corriente continua o una batería compatible para que se pongan en marcha. Es posible modificar la velocidad bajando o subiendo la alimentación, girando más lento o más rápido según el voltaje. Al cambiar la polaridad el motor gira en sentido contrario y si lo desconectas, por inercia sigue girando hasta que se para.

3.1. Modificar la velocidad de giro, PWM

Para regular la velocidad de un motor de continua se usa la modulación por ancho de pulsos (pulse-width modulation PWM). El PWM consiste en modificar la cantidad de energía que recibe un motor. Si se conecta un motor a una fuente directamente el motor comienza a acelerar hasta que alcanza cierta velocidad, si en ese momento se desconecta, el motor decelera hasta que se para. Así pues al conectar y desconectar repetidamente el motor de la fuente en intervalos pequeños el motor no llega a pararse y gira mas despacio. De este modo dependiendo del tiempo que se mantenga conectado y desconectado se pueden obtener distintas velocidades, esto se conoce como ciclo de trabajo D.

$$D = \frac{\tau}{T} \tag{3.1}$$

Donde τ representa el tiempo que está conectado y T el periodo. Por ejemplo, con un periodo de un segundo, se deja conectado 0.7 segundos, se dice que el ciclo de trabajo es del 70 %. Si el ciclo es 0 % se considera apagado y si es 100 % se considera siempre conectado.

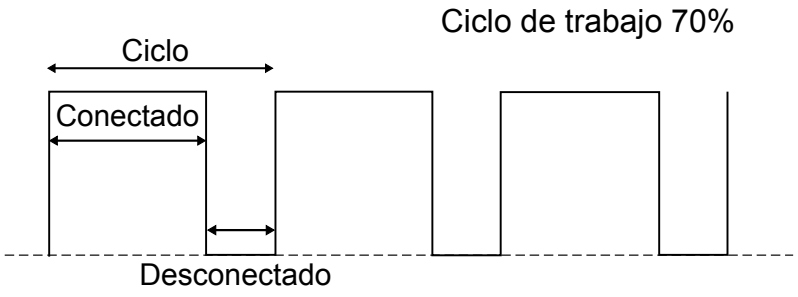


Figura 3.1: Ciclo de trabajo.

Ya que el sistema que se pretende controlar es un motor de continua, se puede realizar tanto un control de posición como un control de velocidad. El control de posición mueve el motor una distancia determinada para después detenerse mientras que el control de velocidad establece una velocidad a la cual debe moverse. Con esta técnica generando una señal con forma de pulso periódico, se puede regular la velocidad y la posición de un motor de corriente continua.

3.2. Posicion y velocidad, encoders

Para medir la velocidad, posición o aceleración de estos motores se suele usar un encoder digital, un encoder digital consta de un disco codificado con ranuras y enganchado al eje de tal manera que el disco gira con el eje. Un sensor detecta el paso de las ranuras y de esta forma sabiendo cuantas ranuras tiene el disco se puede deducir cuantas ranuras son una vuelta, por tanto conociendo el numero de vueltas por unidad de tiempo se puede deducir la velocidad o simplemente cuanto se ha movido.

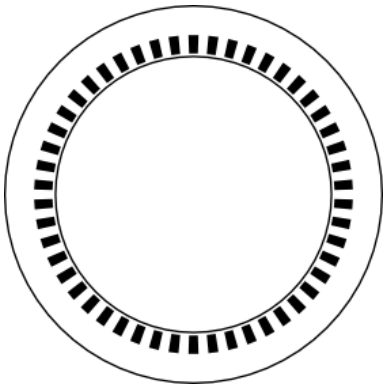


Figura 3.2: Disco encoder incremental.

Normalmente el eje del motor no está conectado directamente al sistema, por ejemplo una rueda, suele utilizar una reductora. Una reductora es un mecanismo compuesto por un conjunto de engranajes que modifica la velocidad del motor, además el eje del motor no debe soportar el peso del sistema y es la reductora junto con un sistema de transmisión el que soporta toda la carga. De este modo conociendo la relación de la reductora y contando las ranuras que se detectan por unidad de tiempo se puede deducir la posición o la velocidad.

3.2.1. Control de velocidad

El control de velocidad implica obtener el numero de pulsos detectados por el paso de las ranuras en cada instante de tiempo. Es decir, pulsos por unidad de tiempo, el valor obtenido será la salida del sistema, en concreto la velocidad actual del motor, que se restará a la referencia para obtener la siguiente acción de control. Así que es importante conocer el tiempo entre acciones de control ya que este periodo servirá para conocer la velocidad del motor entre instantes de tiempo.

3.2.2. Control de posición

El control de posición mira los pulsos detectados desde que el sistema se puso en funcionamiento, y los resta con la referencia a medida que avance el tiempo y se ejecuten acciones de control, el error se irá reduciendo pues la cuenta total de pulsos se irá acercando al valor de posición establecido por la referencia conforme gire el motor, y cuando la resta sea cero, el motor se parará indicando que ha llegado a la posición deseada.

3.3. Sentido de giro, puente H

El sentido de giro de un motor de corriente continua está relacionado con la polaridad de la corriente que se aplica en los bornes. Un puente H consiste de cuatro transistores y un motor en medio, este diseño se asimila a la letra H y de ahí su nombre.

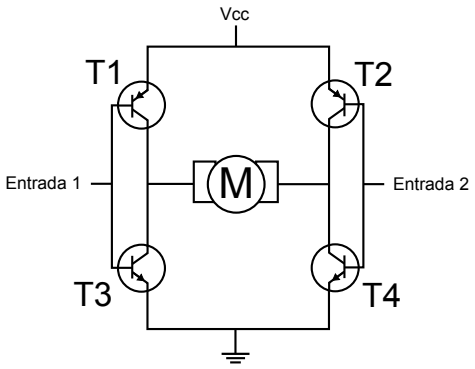


Figura 3.3: Esquema de un puente H.

Al aplicar un *cero* lógico en la entrada 1 y un *uno* lógico en la entrada 2, la corriente irá desde el transistor T1 hacia el T4, lo cual polarizará el motor haciéndolo girar en un sentido, si cambiamos las entradas, un *uno* lógico en la entrada 1 y un *cero* lógico en la entrada 2, la corriente circulará desde el transistor T2 hacia el T3 polarizando el motor tal que girará en sentido contrario.

Si se aplica *cero* lógico en ambas entradas el motor no estará polarizado quedando *liberado*, esto quiere decir que se puede hacer girar con la mano en ambos sentidos. Del mismo modo al poner *uno* lógico en las dos entradas se quedará $fijo$ y no se podrá girar manualmente en ningún sentido.

Capítulo 4

Implementación

Se pretende implementar un Regulador PID en un microcontrolador dsPIC, para ello se explicarán algunas características de la programación de microcontroladores y como interactuar con el y sus módulos. La aplicación se ha desarrollado en el entorno de trabajo MPLAB IDE, proporcionado por la empresa fabricante de los dsPIC, el compilador utilizado es el C30 que posee distintas librerías especializadas en tareas de procesamiento digital de señales.

Se explicará como implementar el algoritmo de control de forma más general en C y, como implementarlo usando las librerías especializadas en cálculo PID. La implementación en C realiza un control e posición y el otro un control de velocidad.

Para probar el funcionamiento se usará un simulador de circuitos llamado Proteus que permite añadir los distintos componentes que forman parte del sistema, el dsPIC y el motor con encoder, y conectarlos entre sí para que interactúen.

4.1. Microcontroladores

Para programar el algoritmo hay que entender la forma de trabajar con estos procesadores, normalmente un microcontrolador, DSP o dsPIC tiene distintos modos de funcionamiento y distintos módulos que pueden ser activados o no a elección del usuario, a su vez los módulos que incorpora se configuran según las necesidades de cada aplicación.

Tal configuración se realiza mediante registros. En los registros se escriben *unos* y *ceros*, dependiendo de lo que se escriba y en que registro se escriba el dispositivo activará o no los módulos que trabajarán según la configuración elegida.

Para saber que valores poner en los registros, los fabricantes ponen a disposición el *DataSheet*, que explica con detalle los módulos del controlador y los registros de configuración de cada módulo. Leer este documento es crítico ya que se explican con detalle las características del dispositivo, lo cual permite ver si sirve para la aplicación a desarrollar. Así pues a la hora de realizar un programa lo primero que se debe hacer es una inicialización de registros con el fin de configurar el procesador.

La comunicación con el procesador se realiza mediante las patillas de éste, de hecho la típica aplicación básica que consiste en imprimir por pantalla *Hello World!*, en microcontroladores lo primero y más básico es encender un LED conocido como *Bliking LED*. Cada patilla se puede configurar como entrada o salida, si se configura como salida, activará físicamente esa patilla con un nivel alto o uno bajo. Al configurarlo como entrada, se realizará una conversión analógica/digital del valor de tensión en la patilla y se guardará en un registro el cual podrá ser leído por un programa.

4.2. dsPIC

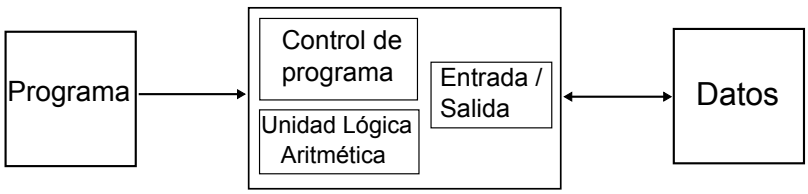
Básicamente un dsPIC es un potente microcontrolador de 16 bit al que se le han añadido las principales características de los DSP. Poseen todos los recursos de los mejores microcontroladores embebidos junto con las principales características de los DSP.

Ofrecen todo lo que se puede esperar de los microcontroladores: velocidad, potencia, manejo flexible de interrupciones, un amplio campo de funciones periféricas analógicas y digitales, opciones de reloj, simulación en tiempo real, etc. La filosofía de Microchip con los modelos de la serie dsPIC ha sido aprovechar que los PIC de 16 bit ya están en el mercado.

Así los diseñadores que han estado utilizando y conocen estos microcontroladores se pueden introducir en el mundo de procesamiento digital de señal con más facilidad. Además, su precio es similar al de los microcontroladores. Son capaces de procesar señales digitales rápidamente, por ejemplo trabajar con audio y vídeo en tiempo real, siendo muy útiles en aplicaciones en las que no se permitan retrasos.

Los dsPIC, al igual que otros DSP usan la arquitectura Harvard que se caracteriza por la separación entre zonas de memoria y datos.

Figura 4.1: Arquitectura Harvard.



Hay distintos modelos de dsPICs, algunos son de propósito general y otros están pensados para controlar motores. Por ejemplo, si se desea usar un solo motor quizás con uno de propósito general sería suficiente sin embargo, para controlar varios motores sería interesante utilizar aquellos especializados en ello. Es importante saber elegir el dispositivo que mejor se adapta a las necesidades de la aplicación, además del precio si se pretende llegar a fabricar en masa.

La empresa que fabrica estos dispositivos pone al alcance de forma gratuita el entorno de desarrollo MPLAB IDE, esta herramienta proporciona un entorno sencillo y potente. Gestiona todas las partes en las que se descompone un proyecto, proporciona un completo editor y un potente depurador a nivel del lenguaje fuente ASM, C y otros.

El integrado posee distintos módulo que permiten el control de motores, uno de ellos el módulo QEI que se encarga de interactuar con el encoder del motor de continua. Otro módulo usado es el generador de señales PWM que se encargará de alimentar a motor con un ciclo de trabajo determinado por el regulador PID.

4.3. Esquema del montaje y simulación

Antes de usar componentes reales es interesante realizar las primeras pruebas de programación mediante simulación, el Proteus permite añadir microcontroladores como componentes, entre ellos el dsPIC, de este modo, teniendo un código ejecutable generado por el MPLAB IDE, se puede especificar en el Proteus donde esta ese fichero y que lo use a la hora de la simulación, ejecutando cualquier programa que queramos y ver el funcionamiento. Esto es

muy útil a la hora de realizar pruebas. Este tipo de herramientas ayuda, pero no tiene en cuenta todos los problemas reales a los que te puedes enfrentar con un montaje físico.

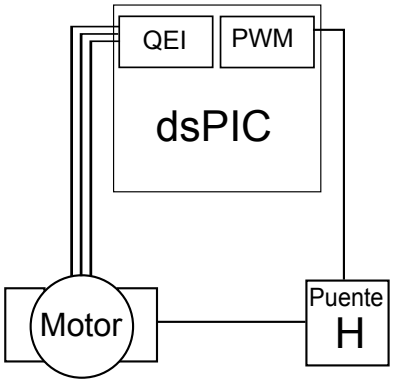


Figura 4.2: Esquema del montaje.

El esquema de montaje de la figura 4.2 pretende reflejar la conexión de los componentes. Teniendo un motor de corriente continua con encoder y un dsPIC, se conecta el encoder del motor al dsPIC con las patillas donde esté configurado el QEI, por otro lado se conectan las patillas donde esté configurado el PWM al puente H, y de ahí al motor. Con el dsPIC programado y conectado al motor, se obtienen los valores de posición o velocidad y el PID calculará la acción de control que generará la señal PWM necesaria para hacer girar el motor.

4.4. Programación en C del algoritmo PID

Como se ha explicado en el capítulo 2 para implementar el algoritmo de control hay que programar la siguiente ecuación:

$$u_{(t)} = K_p e_{(t)} + K_i T_s \sum_{k=0}^{t} e_k + K_d \frac{e_{(t)} - e_{(t-1)}}{T_s} \tag{4.1}$$

4.4.1. Regulador PID de Posición en C

Para programar el PID, se usará un sencillo ejemplo en lenguage C. El ejemplo consiste en controlar la posición de un motor de corriente continua mediante un regulador PID.

La salida del sistema será la posición del motor, el dsPIC calcula la posición a partir de los valores almacenados en un registro del módulo QEI, el módulo QEI lee los pulsos proporcionados por el encoder del motor e incrementa este registro con cada pulso, los pulsos indican cuanto a avanzado el motor. Así, en cada iteración se sabe cuanto ha avanzado, ésto causará que el error disminuya conforme avance el motor. La salida del regulador se escribirá en un registro del módulo PWM, este registro indica el ciclo de trabajo de la señal PWM generada. Al disminuir el error la acción de control también lo hará y por lo tanto en cada iteración la señal PWM disminuirá hasta que la salida del regulador sea cero y por tanto, el registro contendrá un cero, lo cual significará que el ciclo de trabajo es del 0 % y el motor se parará.

```
float ref;//referencia
float y;//salida del sistema
float e;//error
float ek=0;//suma de todos los errores
float e_1=0;//error del instante de tiempo anterior
float Ts=0.1;//periodo de muestreo de la senyal
float u;//salida del controlador
float Kp=3;//ganancia proporcional
float Ki=0.1;//ganancia integral
float Kd=1;//ganancia derivativa

while(1)
  {
  //posicion actual
  y = POS1CNT;/* Registro donde se encuentra la salida
     del sistema, es decir el avance total del motor */
  //regulador PID
  e = ref-y;//error actual
  ek+=e;//acumulo todos los errores para el integral
  u=Kp*e+Ki*Ts*ek+kd*(e-e_1)/Ts;//accion de control
  P1DC1=u;//Registro donde se escribe el ciclo de
     trabajo
  e_1=e;// error para el instante anterior e(t-1)
  }
```

En el ejemplo se ha usado un bucle `while`, calculando la acción de control tan rápido como puede la máquina, sin embargo la acción de control debe ejecutarse cada cierto periodo según el sistema que se esté regulando, por ello cada acción de control ha de ejecutarse activando alguna interrupción, utilizando tareas, o simplemente colocando una espera en el bucle, el tiempo entre cada ejecución de las acciones de control marcará el periodo.

4.4.2. Regulador PID de Velocidad en C

Con el mismo montaje que el ejemplo anterior se pretende controlar la velocidad de un motor de corriente continua mediante un regulador PID. La principal diferencia frente al control de posición es que se debe calcular el número de pulsos detectados en cada ciclo del bucle, esta cantidad de pulsos dividida por el tiempo que tarda en ejecutarse un ciclo, permite obtener la velocidad del motor que es la salida del sistema. En el momento que alcance la velocidad deseada, el error será cero, esto provocará que la señal PWM sea cero y el motor se pare durante en esa acción de control provocando nuevamente un error que intentará regular la acción de control del instante de tiempo siguiente. Así pues el motor alcanzará y mantendrá la velocidad establecida por la referencia.

```
float ref;//referencia
float y;//salida del sistema
float e;//error
float ek=0;//suma de todos los errores
float e_1=0;//error del instante de tiempo anterior
float Ts=0.1;//periodo de muestreo de la senyal
float u;//salida del controlador
float Kp=3;//ganancia proporcional
float Ki=0.1;//ganancia integral
float Kd=1;//ganancia derivativa

float ultimoQEItick;
float cicloQEIticks;
float actualQEItick;

ultimoQEItick = POS1CNT;//se almacena el ultimo tick
```

```
while(1)
  {

  /* velocidad = cicloQEIticks / tiempo de ciclo */

  actualQEItick = POS1CNT; /* Guardar la cuenta de
     pulsos QEI */
  cicloQEIticks = actualQEItick - ultimoQEItick; /*
     Restar con el instante anterior */
  ultimoQEItick = actualQEItick; /* Actualizar la cuenta
      de pulsos anterior */

  //Velocidad actual
  y = cicloQEIticks;/* Registro donde se encuentra la
     salida del sistema, es decir la velocidad del motor
      en este instante de tiempo */


  //regulador PID
  e = ref-y;//error actual
  ek+=e;//acumulo todos los errores para el integral
  u=Kp*e+Ki*Ts*ek+kd*(e-e_1)/Ts;//accion de control

  P1DC1=u;//Registro donde se escribe el ciclo de
     trabajo

  e_1=e;// error para el instante anterior e(t-1)
  }
```

4.5. Programación usando las librerías del dsPIC

En el compilador C30 de microchip se incluyen distintas librerías las cuales incluyen: librería C estándar, librería DSP y librería de funciones matemáticas, tanto en como fija como coma flotante.

La librería para DSP proporciona un conjunto de funciones rápidas y optimizadas para aplicaciones de procesamiento digital de señales, incrementa significativamente el rendimiento frente a las equivalentes en C. Se puede utilizar con cualquier dispositivo dsPIC. Está escrita principalmente en lenguaje ensamblador y hace un uso extensivo del juego de instrucciones y recursos hardware del dsPIC.

Proporciona funciones para operar con vectores, matrices, filtros, transformadas y manejo de ventanas. En concreto se usarán las funciones que permiten calcular la acción de control de un regulador PID en vez de implementarlo en C directamente como se ha hecho en el apartado anterior.

4.5.1. Funciones PID

Para el cálculo del PID lo primero es declarar una estructura tPID. La estructura contiene distintas variables que almacenarán los datos usados por las funciones de cálculo PID encargadas recibir la referencia y obtener la acción de control.

```
typedef struct {

  fractional* abcCoefficients;
  /* Puntero a los coeficientes que se encuentran en el
     espacio de memoria X, estos coeficientes se deriva
     de los valores de las ganancias PID: Kp, Ki y Kd */

  fractional* controlHistory;
  /* Puntero a las 3 ultimas muestras alamcenadas en el
     espacio de memoria Y, la primera es la mas reciente
      */


  fractional controlOutput;
  /* Salida del regulador PID  */

  fractional measuredOutput;
  /* Valor medido de la respuesta del sistema */

  fractional controlReference;
  /* Referencia del sistema */

} tPID;
```

A los espacios de memoria se acceden mediante diferentes unidades de direccionamiento y distintas rutas de datos. Las instrucciones DSP con dos operandos como las instrucciones MAC, acceden a los espacios de memoria X e Y de forma separada para poder hacer lecturas simultáneas de ambos operandos y así poder completarse en un ciclo de reloj.

Las funciones encargadas del cálculo de la acción de control están implementadas en ensamblador lo cual agiliza el tiempo de cálculo y permite controlar sistemas que requieren una respuesta muy rápida. En el fichero `pid.s` se encuentra la implementación de dichas funciones. Este fichero se proporciona con el compilador C30.

```
extern void PIDCoeffCalc( fractional* kCoeffs, tPID*
   controller );
  /* Deriva los coeficientes ABC usando las ganacias del
      PID: Kp, Ki y Kd. Necesita un array que contenga
     los valores Kp, Ki y Kd en secuencia y el puntero a
      la estructura de datos tPID */
extern void PIDInit ( tPID* controller );
  /* Limpia las variables de estado del PID y la salida
     de las muestras. Necesita el puntero a la
     estructura de datos tPID */
extern fractional* PID ( tPID* controller );
  /* Calcula la accion de control PID. Necesita el
     puntero a la estructura de datos tPID */
```

4.5.2. Funciones implementadas en ensamblador

Las siguientes funciones se encuentran implementadas en ensamblador, y son proporcionadas por el compilador C30 en concreto en el fichero `pid.s`, dichas funciones son llamadas desde el programa en C. En los siguientes apartados hay ejemplos de su uso.

```
.nolist
.include "dspcommon.inc" ; fractsetup
.list

.equ    offsetabcCoefficients, 0
.equ    offsetcontrolHistory, 2
.equ    offsetcontrolOutput, 4
.equ    offsetmeasuredOutput, 6
.equ    offsetcontrolReference, 8

.section .libdsp, code
```

4.5.2.1. PID

```
; _PID:
; tPID PID ( tPID *fooPIDStruct )

        .global _PID  ; provide global scope to routine
_PID:

; Save working registers.
push    w8
push    w10
push    CORCON ; Prepare CORCON for fractional computation.

fractsetup      w8

mov [w0 + #offsetabcCoefficients], w8
; w8 = Base Address of _abcCoefficients array [(Kp+Ki+Kd), -(Kp+2Kd), Kd]
mov [w0 + #offsetcontrolHistory], w10
; w10 = Address of _ControlHistory array (state/delay line)
```

```
mov [w0 + #offsetcontrolOutput], w1
mov [w0 + #offsetmeasuredOutput], w2
mov [w0 + #offsetcontrolReference], w3

; Calculate most recent error with saturation, no limit checking required
lac    w3, a    ; A = tPID.controlReference
lac    w2, b    ; B = tPID.MeasuredOutput
sub    a        ; A = tPID.controlReference - tPID.measuredOutput
sac.r  a, [w10] ; tPID.ControlHistory[n] = Sat(Rnd(A))

; Calculate PID Control Output
clr a, [w8]+=2, w4, [w10]+=2, w5  ; w4 = (Kp+Ki+Kd),w5=_ControlHistory[n]
lac w1, a                         ; A = ControlOutput[n-1]
mac w4*w5,a,[w8]+=2,w4,[w10]+=2,w5; A += (Kp+Ki+Kd)*_ControlHistory[n]
                                  ; w4 = -(Kp+2Kd),w5=_ControlHistory[n-1]
mac w4*w5, a, [w8], w4,[w10]-=2,w5; A += -(Kp+2Kd) * _ControlHistory[n-1]
                                  ; w4 = Kd, w5 = _ControlHistory[n-2]
mac w4*w5, a, [w10]+=2, w5        ; A += Kd * _ControlHistory[n-2]
                                  ; w5 = _ControlHistory[n-1]
                                  ; w10 = &_ControlHistory[n-2]
sac.r   a, w1                     ; ControlOutput[n] = Sat(Rnd(A))
mov     w1, [w0 + #offsetcontrolOutput]

;Update the error history on the delay line
mov     w5, [w10]       ; _ControlHistory[n-2] = _ControlHistory[n-1]
mov     [w10 + #-4], w5 ; _ControlHistory[n-1] = ControlHistory[n]
mov     w5, [--w10]

pop     CORCON          ; restore CORCON.
pop     w10             ; Restore working registers.
pop     w8
return
```

4.5.2.2. PIDInit

```
; _PIDInit:
; void PIDInit ( tPID *fooPIDStruct )

        .global _PIDInit                ; provide global scope to routine
_PIDInit:
        push    w0
        add  #offsetcontrolOutput, w0   ; Set up pointer for controlOutput
     clr  [w0]                       ; Clear controlOutput
     pop  w0
     push w0
                                        ;Set up pointer to the base of
                                        ;controlHistory variables within
                                            tPID
        mov     [w0 + #offsetcontrolHistory], w0
                                        ; Clear controlHistory variables
                                        ; within tPID
        clr     [w0++]                  ; ControlHistory[n]=0
        clr     [w0++]                  ; ControlHistory[n-1] = 0
        clr     [w0]                    ; ControlHistory[n-2] = 0
        pop     w0                      ;Restore pointer to base of tPID
        return
```

4.5.2.3. PID

```
; _PIDCoeffCalc:
; void PIDCoeffCalc ( fractional *fooPIDGainCoeff, tPID *fooPIDStruct )

        .global _PIDCoeffCalc
_PIDCoeffCalc:
        push    CORCON                  ; Prepare CORCON for fractional
            computation.
        fractsetup      w2
                                        ;Calculate Coefficients from Kp, Ki
                                            and Kd
        mov     [w1], w1
        lac     [w0++], a               ; ACCA = Kp
        lac     [w0++], b               ; ACCB = Ki
        add     a                       ; ACCA = Kp + Ki
        lac     [w0--], b               ; ACCB = Kd
        add     a                       ; ACCA = Kp + Ki + Kd
        sac.r   a, [w1++]               ; _abcCoefficients[0] =  Sat(Rnd(Kp
            + Ki + Kd))
        lac     [--w0], a               ; ACCA = Kp
        add     a                       ; ACCA = Kp + Kd
        add     a                       ; ACCA = Kp + 2Kd
        neg     a                       ; ACCA = -(Kp + 2Kd)
        sac.r   a, [w1++]               ; _abcCoefficients[1] =  Sat(Rnd(-Kp
             - 2Kd))
        sac     b, [w1]                 ; _abcCoefficients[2] =  Kd

        pop     CORCON
        return
```

```
        .end
```

4.5.3. Ejemplo de uso de las funciones PID

El siguiente ejemplo no pretende implementar ninguna aplicación, simplemente muestra la forma de trabajar con las funciones proporcionadas por las librerías.

```
/* Se declara una una estructura de datos PID, llamada
   fooPID */
tPID fooPID;
/* La estructura fooPID contiene un puntero a los
   coeficientes en el espacio de memoria X y un puntero
   a las muestras del estado del controlador (historial)
    en el espacio de memoria Y. Por tanto hay que
   declarar las variables para los coeficientes y las
   muestras del historial del regulador.*/
fractional abcCoefficient[3] __attribute__
((section (".xbss, bss, xmemory")));

fractional controlHistory[3] __attribute__
((section (".ybss, bss, ymemory")));
/* Los coeficientes ABC que se referencian en la
   estructura fooPID se derivan de los coeficientes de
   ganancia Kp, Ki y Kd, que deben estar en un array */
fractional kCoeffs[] = {0,0,0};
```

```
/* La funcion main llama a las funciones PID(), PIDInit
   () and PIDCoeffCalc() */
int main (void)
{
/* Primer paso, inicializar la estructura de datos PID,
   fooPID */
/* Apuntar a los coeficientes */
  fooPID.abcCoefficients = &abcCoefficient[0];
/* Apuntar al historial de muestras del regulador */
  fooPID.controlHistory = &controlHistory[0];
/* Limpiar el historial y la salida del controlador */
  PIDInit(&fooPID);

  kCoeffs[0] = Q15(0.7); //Kp
  kCoeffs[1] = Q15(0.2); //Ki
  kCoeffs[2] = Q15(0.07);//Kd
/* Derivar los coeficientes a,b, & c a partir de Kp, Ki
   & Kd */
  PIDCoeffCalc(&kCoeffs[0], &fooPID);

/* Segundo paso: Usar el regualdor PID */
  while(1)
  /* Se usa un bucle sin embargo se debe ejecutar
     mediante una tarea, la activacion de un timer o la
     interrupcion de una conversion A/D*/
  {
    /* La refencia del sistema. No tiene por que ser
       fija aunque en este ejemplo lo es */
    fooPID.controlReference = Q15(0.74);
    /* Normalmente, la variable measuredOutput es la
       salida del sistema medida a partir de una entrada
```

```
        A/D o un sensor. En este ejemplo se establece
        manualmente un valor de demostracion, pero se
        debe tener en cuenta que este valor va a seguir
        cambiando en una aplicacion real */
    fooPID.measuredOutput = Q15(0.453);
    /* Se llama al regulador PID con la nueva medida de
        la salida */
    PID(&fooPID);
    fooPID.controlOutput /* La salida del regulador o
        accion de control*/
  }
}
```

4.5.4. Regulador PID de Posición con librerias

El siguiente ejemplo usa las librerías DSP y las funciones PID para implementar un regulador PID.

```
float refpos;/* Referencia */
float pos;/* almacenar la poscion actual */



fooPID.abcCoefficients = &abcCoefficient[0];
fooPID.controlHistory = &controlHistory[0];
PIDInit(&fooPID);
kCoeffs[0] = Q15(0.99999999);
kCoeffs[1] = Q15(0.0);
kCoeffs[2] = Q15(0.0);
PIDCoeffCalc(&kCoeffs[0], &fooPID);
```

```
while(1)
  {

  /* velocidad = cicloQEIticks / tiempo de ciclo */

  pos = POS1CNT; /* Leer la cuenta de pulsos */

  fooPID.controlReference = Q15(refpos);   /* la
     referencia de posicion */
    fooPID.measuredOutput = Q15(pos);   /* */

  PID(&fooPID);

  //15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
  P1DC1 = fooPID.controlOutput&0x3FF;
  }
```

4.5.5. Regulador PID de Velocidad con librerias

El siguiente ejemplo calcula un regulador de velocidad PID usando las funciones de librería.

```
float refvel;
float ultimoQEItick;
float cicloQEIticks;
float actualQEItick;

ultimoQEItick = POS1CNT;
```

```
fooPID.abcCoefficients = &abcCoefficient[0];
fooPID.controlHistory = &controlHistory[0];
PIDInit(&fooPID);
kCoeffs[0] = Q15(0.99999999);
kCoeffs[1] = Q15(0.0);
kCoeffs[2] = Q15(0.0);
PIDCoeffCalc(&kCoeffs[0], &fooPID);



while(1)
  {

  /* velocidad = cicloQEIticks / tiempo de ciclo */

  actualQEItick = POS1CNT; /* Guardar la cuenta de
     pulsos QEI */
  cicloQEIticks = actualQEItick - ultimoQEItick; /*
     Restar con el instante anterior */
  ultimoQEItick = actualQEItick; /* Actualizar la cuenta
      de pulsos anterior */

  fooPID.controlReference = Q15(refvel);   /* la
     referencia de velocidad */
    fooPID.measuredOutput = Q15(cicloQEIticks);

  PID(&fooPID);

  //15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
  P1DC1 = fooPID.controlOutput&0x3FF;
  }
```

Capítulo 5

Tareas, FreeRTOS

Un sistema empotrado, normalmente necesita implementar distintas tareas para que se ejecuten paralelamente, es por eso que usando un mini Kernel de tiempo real ayuda en la creación y gestión de éstas. FreeRTOS ofrece todas estas soluciones además de incluir código de ejemplo para distintos modelos y marcas de microcontroladores.

Está distribuido bajo la licencia GPL, aunque se pueden adquirir otras versiones como OpenRTOS que incluye mejoras profesionales y SafeRTOS que está certificado. Estas otras versiones no son gratuitas.

Es simple y pequeño, muy útil para principiantes y aficionados. El planificador se puede configurar para que las tareas se ejecuten conjuntamente o para que se ejecute solo una dependiendo de la prioridad.

5.1. Implementación de tareas

Para implementar una tarea en FreeRTOS lo primero es establecer una priorirad, esto se hace mediante un `#define`

```
/* Prioridades de las tareas. */
#define mainBLOCK_Q_PRIORITY    ( tskIDLE_PRIORITY + 2 )
#define mainCHECK_TASK_PRIORITY ( tskIDLE_PRIORITY + 3 )
#define mainCOM_TEST_PRIORITY   ( 2 )
```

A continuación hay que establecer el periodo:

```
/* Periodos de las tareas. */
#define mainCHECK_TASK_PERIOD      ( ( portTickType )
   3000 / portTICK_RATE_MS )
#define mainREFERENCIA_TASK_PERIOD ( ( portTickType )
   300 / portTICK_RATE_MS )
#define mainCONTROL_TASK_PERIOD    ( ( portTickType ) 50
    / portTICK_RATE_MS )
```

Las cabeceras de las funciones:

```
static void vReferenciaTask(void *pvParameter);
static void vControlVelTask(void *pvParameter);

void inicializarQEI(void);
void inicializarPWM(void);

int refpos=256;
```

Y las variables globales, que usarán distintas tareas para comunicarse. En la función `main` se llamará a las funciones de inicialización, y se crearán las tareas, por último solo hay que llamar al planificador.

```
int main( void )
{
  /* Desde aqui se llama a las funciones de
     incializacion. */
  prvSetupHardware();

  xTaskCreate(vReferenciaTask, ( signed char * ) "
     Referencia", configMINIMAL_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY, NULL);
  xTaskCreate(vControlVelTask, ( signed char * ) "
     ControlVel", configMINIMAL_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY + 1, NULL);

  /* Finally start the scheduler. */
  vTaskStartScheduler();

  /* Will only reach here if there is insufficient heap
     available to start
  the scheduler. */
  return 0;
}
```

```
static void prvSetupHardware( void )
{   /*Inicializacion de los modulos del dsPIC*/
  inicializarQEI();
  inicializarPWM();
}
```

5.2. Tiempo absoluto

Para que las tareas se ejecuten en tiempos absolutos, hay que usar las siguientes instrucciones dentro del cuerpo de la función que implementa la tarea, de este modo la tarea se ejecutará según el periodo establecido.

```
static void vMiTareaTask(void *pvParameter) {
/* Se usa para despertar la tarea a una frecuencia
   determinada */
portTickType xLastExecutionTime;
/* Inicializar xLastExecutionTime la primera vez que se
   llama a vTaskDelayUntil(). */
xLastExecutionTime = xTaskGetTickCount();
  while(1)
  {
    /* Esperar a que llegue la hora del siguiente ciclo.
        */
    vTaskDelayUntil( &xLastExecutionTime,
       mainCONTROL_TASK_PERIOD );
    .
    .
    .

  }
}
```

5.3. Ejemplo de implementación de una tarea

La siguiente tarea implementa el control de velocidad de un motor de corriente continua, usando para ello las funciones de librería de un regulador PID.

```
static void vControlVelTask(void *pvParameter) {
/* Used to wake the task at the correct frequency. */
portTickType xLastExecutionTime;
int ultimoQEItick;
int cicloQEIticks;
int actualQEItick;
/* Initialise xLastExecutionTime so the first call to
   vTaskDelayUntil()
   works correctly. */
xLastExecutionTime = xTaskGetTickCount();
ultimoQEItick = POS1CNT;
/*  POS1CNT - Registro donde esta la cuenta de pulsos
   del encoder.
    Cuidado porque cambia a 0 al llegar al MAX1CNT. */

fooPID.abcCoefficients = &abcCoefficient[0];
fooPID.controlHistory = &controlHistory[0];
PIDInit(&fooPID);
kCoeffs[0] = Q15(0.99999999);
kCoeffs[1] = Q15(0.0);
kCoeffs[2] = Q15(0.0);
PIDCoeffCalc(&kCoeffs[0], &fooPID);
```

```
while(1)
  {
  /* Wait until it is time for the next cycle. */
  vTaskDelayUntil( &xLastExecutionTime,
     mainCONTROL_TASK_PERIOD );

  //velocidad = cycleQEIticks / ciclo
  actualQEItick=POS1CNT;
  cicloQEIticks = ultimoQEItick - actualQEItick;//cuenta
      hacia abajo
  ultimoQEItick = actualQEItick;
  //motor 256 ranuras en una vuelta

  fooPID.controlReference = Q15(refvel);
    fooPID.measuredOutput = Q15(cicloQEIticks/0x3E);

  PID(&fooPID);

  //15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
  //putc(fooPID.controlOutput);

/* Se escribe la accion de control en el registro que
   establece el PWM donde 0x3ff es el 100% del ciclo de
   trabajo */
  P1DC1 = fooPID.controlOutput&0x3FF;
  //sentido=(fooPID.controlOutput&0x200)==0x200;//para
     el puente h
  }
}
```

Apéndices

Apéndice A

Configuración del módulo PWM

El MCPWM (Motor Control PWM) se encarga de generar una señal con forma de pulso periódico. El periódo y el ciclo de trabajo de los pulsos son programables. Dependiendo del microcontrolador puede tener varios módulos MCPWM, en cualquier caso la configuración que se explica sólo usará uno ya que solo se va a controlar un motor.

A.1. Registros de configuración

El microcontrolador posee distintos registros con distintos bits para controlar el modo de operación del MCPWM. La información de todos los registros se encuentra en el *datasheet* del microcontrolador. Se explicarán los utilizados, el resto al no modificarse quedarán en su valor por defecto (cero) y por tanto en el modo de funcionamiento correspontiente al cero.

A.1.1. P1TCON

En el registro P1TCON se selecciona el modo de funcionamiento y se controla el reloj del PWM.

A.1.2. P1TPER

En el registro P1TPER se escribe el valor del tiempo base, el cual determina la frecuencia de funcionamiento del PWM.

A.1.3. PWM1CON1

En el registro PWM1CON1 se seleccionan los pines por los cuales saldrá el PWM, se pueden elegir dos pines y hacer que funcionen en modo independiente o complementario.

A.1.4. P1DTCON1 y P1DTCON2

En el registro P1DTCON1 se configura el periodo de reloj del tiempo muerto y en el registro P1DTCON2 se eligen los pines en los cuales la señal de tiempo muerto está activa.

A.1.5. P1DC1

En el registro P1DC1 se escribe el valor, en 16 bits, del ciclo de trabajo de la salida PWM para el par de pines 1.

A.1.6. P1OVDCON

En el registro P1OVDCON se seleccionan los pines que serán controlados por el generador PWM .

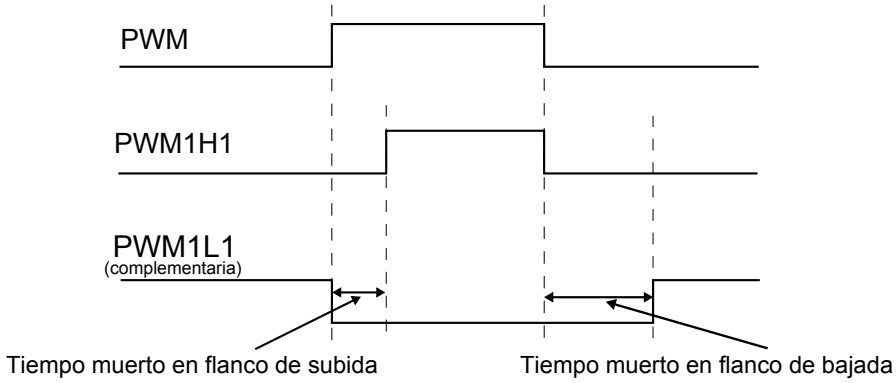
A.2. Tiempo muerto

Debido a que los dispositivos de alimentación no pueden cambiar instantáneamente de valor, se requiere cierto tiempo entre el evento de desconexión y el evento de conexión de una salida PWM en un par complementario. La generación del tiempo muerto es activada automáticamente cuando un par de pines está funcionando en modo complementario.

Cada par de salidas complementaria del PWM tiene un contador descendente de 6 bits que se usa para producir la inscerción del tiempo muerto.

El tiempo muerto se carga en el contador al detectar el evento de flanco del PWM. Dependiendo del flanco, si es de subida o de bajada, una de las transiciones de la salida complementaria es retrasada hasta que el contador llegue a cero.

Figura A.1: Tiempo muerto en un par complementario.



A.2.1. Ejemplo de inicialización

```
P1TCON = 0;
P1TMR = 0;
P1TPER = 0x1FE; //pagina 34 Section 14 PWM
P1SECMP = 0;
PWM1CON1 = 0x0E11; // Enable PWM output pins and
   configure them as complementary mode
PWM1CON2 = 0x0000;
P1DTCON1 = 0x0000;
P1DTCON2 = 0x0000;
P1FLTACON = 0x0000;//PxFLTACON: Fault A Control Register
//bit 15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
P1DC1 = 0x0;
P1OVDCON=0Xff00;
P1TCONbits.PTEN = 1;
```

Apéndice B

Configuración de módulo QEI

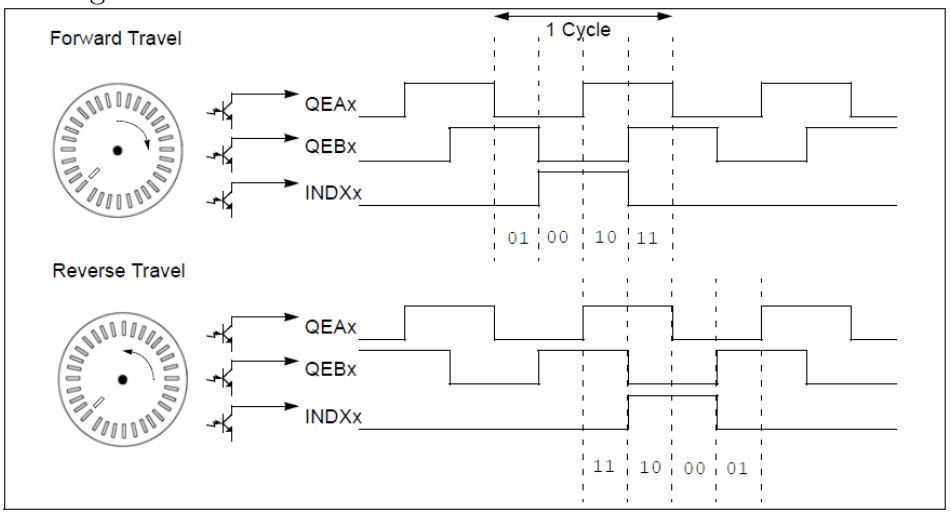
El módulo QEI proporciona una interfaz para encoders incrementales que permite obtener datos de posición mecánicos. Los encoders incrementales o de cuadratura detectan la posición y la velocidad de sistemas de movimiento de rotación.

Un típco encoder de cuadratura incluye una rueda ranurada unida al eje del motor y un módulo que detecta las ranuras de la rueda. La información se emite normalmente por tres canales de salida, Fase A (QEAx), Fase B (QEBx) y Índice (INDXx), obteniendo el movimiento del eje del motor, la distancia y la dirección. Las señales proporcionados son digitales.

Los canales Fase A y Fase B tienen una relación única. Si la Fase A va por delante de la Fase B, la dirección del motor se considera positiva o hacia delante. Si la Fase B va por delante de la Fase A, se considera negativa o hacia atrás. El pulso que incrementa el Índice ocurre una vez por revolución y es usado como referencia para indicar la posición absoluta. Se observa un diagrama de tiempos relativo a éstas tres señales en la figura B.1.

El QEI consta de un decodificador lógico que interpreta las señales de la Fase A (QEAx) y la Fase B (QEBx), además tiene un contador up/down para acumular la cuenta. Las señales de entrada son filtradas mediante un filtro de ruido digital.

Figura B.1: Señales de la interfaz del encoder de cuadratura.



B.1. Registros de configuración

El módulo QEI tiene cuatro registros accesibles para el usuario. Para poner en marcha este módulo, se deberán configurar estos registros adecuadamente segun las necesidades. Además se deben establezer que pines recibirán las señales procedientes del encoder.

B.1.1. QEI1CON

El QEI Control Register, controla las operaciones QEI y proporciona flags de estado del modulo.

B.1.2. DFLT1CON

El Digital Filter Control Register, controla las operaciones del filtro digital de entrada.

B.1.3. POS1CNT

El Position Count Register, permite la lectura y escritura del contador de posición de 16 bits. Cuando este registro llega a su valor máximo, vuelve a valer cero y sigue incrementándose, esto puede provocar problemas a la hora de calcular el valor de pulsos transcurrido en un intervalo de tiempo, al hacer la resta entre el valor del inicio y del final del intervalo el registro puede reiniciarse provocando un resta errónea desde el punto de vista del algoritmo de control, para solucionarlo se puede usar una variable de programa adicional para detectar el valor máximo del registro e incrementar en uno la variable, de este modo la cuenta está almacenada en el registro(parte baja) y la variable (parte alta) lo cual proporciona una mayor capacidad de cuenta.

B.1.4. MAX1CNT

El Maximum Count Register, Mantiene un valor que es comparado con el contador POS1CNT en algunas operaciones.

B.1.5. Ejemplo de inicialización

```
//POS1CNT = 0xFFFF;
//Se incrementa con la cuenta de pulsos del encoder

QEI1CONbits.QEIM = 0b101;
//resolucion 2x, Reset al coincidir con MAXCNT
QEI1CONbits.QEISIDL = 0;
// continuar en modo idle (0)
QEI1CONbits.SWPAB = 0;
// Phase-A y Phase-B no intercambiados
QEI1CONbits.PCDOUT = 1;
// activado el Position Counter Direction Status
QEI1CONbits.TQGATE = 0;
// Timer gate apagado
QEI1CONbits.TQCKPS = 0;
// Prescaler 1:1
QEI1CONbits.POSRES = 1;
// Un pulso en INDEX hace un reset
QEI1CONbits.TQCS =0;
// Usamos clock interno para el timer
QEI1CONbits.UPDN_SRC=1;
// Phase-B indica direccion
MAX1CNT=23;
//POS1CNT se pone a 0 cuando POS1CNT es igual que
   MAX1CNT
DFLT1CON = 0x00F0;
//filtro de ruido habilitado
```

Apéndice C

Configuración de módulo ADC

C.1. Funcionamiento

La conversión analógica-digital (ADC) consiste en transformar señales analógicas en señales digitales, para poder procesarlas de forma más sencilla. Las señales digitales se pueden procesar en computadores, lo cual implica la posibilida de aplicar filtros para mejorar la señal o eliminar el ruido.

Convertir una señal analógica en digital consiste en obtener un valor de la señal para cada instante de tiempo, a esto sele llama muestreo, y al tiempo entre la toma de muestras, periodo de muestreo.

C.2. Registros de configuración

A continuación se describen los distintos registros que se han de configurar para poner en funcionamiento y ejemplo sencillo de conversión. El módulo ADC tiene dos modos de funcionamiento, 4 canales de 10 bits o un canal de 12 bits.

C.2.1. AD1CON1

En el modo de 10 bits se puede seleccionar usar un canal, dos canales o cuatro canales y no todos lo pines del microcontrolador pueden usarse para cualquier canal. También se puede seleccionar el modo de autoconversión. Además este registro inicia o para la conversión modificando el bit AD1CON1bits.ADON.

C.2.2. AD1CON2

En este registro se pueden seleccionar los voltages de referencia VREFH y VREFL para las conversiones A/D ya sea mediante valores externos conectados a los pines o usando los valores internos del microcontrolador AVDD y AVSS.

C.2.3. AD1CON3

En este registro se configura el reloj de conversión analógica para que conincida la velocidad de muestreo de los datos con el reloj del procesador.

C.2.4. AD1PCFGL

En este registro se seleccionan los pines de un puerto como entradas analógicas. Cuando se configura un pin como entrada analógica, tambien hay que configuralo como entrada usando TRIS.

C.2.5. AD1CHS0 y AD1CHS123

El módulo puede escanear distintos canales o no, esto se configura en AD1CON2bits.CSCNA, si se usa el modo de no escanear se usará el canal cero y la entrada será el pin seleccionado en AD1CHS0bits.CH0SA, a su vez el canal puede muestrear dos entradas, primero una y luego la otra, A y B seleccionando el pin en AD1CHS0bits.CH0SB . Para muestrear dos entradas hay que activar el bit ALTS en el registro AD1CON2, si no siempre muestreará la A. En este caso para simplificar se seleccionara un canal, el cero, y solo una entrada.

C.3. Ejemplo de inicialización

```
AD1CON1bits.AD12B = 0; //10-bit ADC operation
AD1CON1bits.FORM = 0;
// 00 = Integer (DOUT= 0000 00dd dddd dddd)
AD1CON1bits.SSRC = 7;
// 111 = Internal counter ends sampling and starts
   conversion (auto-convert)
AD1CON1bits.ASAM = 1;
// 1 = Sampling begins immediately after last conversion
AD1CON1bits.SIMSAM = 0;
// 0 = Samples multiple channels individually in
   sequence
AD1CON2bits.CHPS = 0; //Converts channel CH0
AD1CON2bits.SMPI = 0;
// 0000 = ADC interrupt is generated at the completion
// of every sample/conversion operation
AD1CON3bits.ADRC = 0;
// ADC Clock is derived from Systems Clock
AD1CON3bits.SAMC = 0; //Auto Sample Time = 0 * TAD
AD1CON3bits.ADCS = 2;
// ADC Conversion Clock
// TAD = TCY * (ADCS + 1) = (1/40M) * 3 =
AD1CHS0bits.CH0SA = 0;
//MUXA +ve input selection (AIN0) for CH0
AD1CHS0bits.CH0NA = 0;
//MUXA -ve input selection (VREF-) for CH0
AD1CHS123bits.CH123SA = 0;
//MUXA +ve input selection (AIN0) for CH1
AD1CHS123bits.CH123NA = 0;
//MUXA -ve input selection (VREF-) for CH1
```

```
AD1PCFGL = 0xFFFF;
//AD1PCFGH/AD1PCFGL: Port Configuration Register
AD1PCFGLbits.PCFG0 = 0; //AN0 as Analog Input
IFS0bits.AD1IF = 0; //Clear the A/D interrupt flag bit
IEC0bits.AD1IE = 0; //Do Not Enable A/D interrupt
AD1CON1bits.ADON = 1; //Turn on the A/D converter
TRISAbits.TRISA0 =1; //pin configured as input
```

Anexo i

main.c

```
/* Standard includes. */
#include <stdio.h>
#include <dsp.h>

/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "croutine.h"

/* Demo application includes. */
#include "BlockQ.h"
#include "crflash.h"
#include "blocktim.h"
#include "integer.h"
#include "comtest2.h"
#include "partest.h"
```

```
#include "lcd.h"
#include "timertest.h"

/* Demo task priorities. */
#define mainBLOCK_Q_PRIORITY        ( tskIDLE_PRIORITY +
    2 )
#define mainCHECK_TASK_PRIORITY       ( tskIDLE_PRIORITY
    + 3 )
#define mainCOM_TEST_PRIORITY       ( 2 )

/* The check task may require a bit more stack as it
   calls sprintf(). */
#define mainCHECK_TAKS_STACK_SIZE     (
   configMINIMAL_STACK_SIZE * 2 )

/* The execution period of the check task. */
#define mainCHECK_TASK_PERIOD       ( ( portTickType )
   3000 / portTICK_RATE_MS )
#define mainREFERENCIA_TASK_PERIOD      ( ( portTickType
    ) 300 / portTICK_RATE_MS )
#define mainCONTROL_TASK_PERIOD       ( ( portTickType )
    50 / portTICK_RATE_MS )


/* The number of flash co-routines to create. */
#define mainNUM_FLASH_COROUTINES      ( 5 )

/* Baud rate used by the comtest tasks. */
#define mainCOM_TEST_BAUD_RATE        ( 19200 )
```

```
/* The LED used by the comtest tasks.  mainCOM_TEST_LED
   + 1 is also used.
See the comtest.c file for more information. */
#define mainCOM_TEST_LED          ( 6 )

/* The frequency at which the "fast interrupt test"
   interrupt will occur. */
#define mainTEST_INTERRUPT_FREQUENCY    ( 20000 )

/* The number of processor clocks we expect to occur
   between each "fast
interrupt test" interrupt. */
#define mainEXPECTED_CLOCKS_BETWEEN_INTERRUPTS (
   configCPU_CLOCK_HZ / mainTEST_INTERRUPT_FREQUENCY )

/* The number of nano seconds between each processor
   clock. */
#define mainNS_PER_CLOCK ( ( unsigned short ) ( ( 1.0 /
   ( double ) configCPU_CLOCK_HZ ) * 1000000000.0 ) )

/* Dimension the buffer used to hold the value of the
   maximum jitter time when
it is converted to a string. */
#define mainMAX_STRING_LENGTH       ( 20 )

//paco
#define mainLED_DELAY ((portTickType) 1000/
   portTICK_RATE_MS)

//static void vUserLED1Task(void *pvParameter);
```

```
//static void vUserLED2Task(void *pvParameter);

static void vReferenciaTask(void *pvParameter);
static void vControlVelTask(void *pvParameter);
static void vControlPosTask_P(void *pvParameter);
static void vControlPosTask_PI(void *pvParameter);


/*en ref meto el valor de 15 bits correspontiente al
   ciclo de trabajo P1DC1<15:0>*/
float refvel=0.3;
int refpos=256;

//El motor con 256 ranuras en el encoder me de 32 ticks
   por ciclo. Al 100 me da 0x2D~~~0x2E


tPID fooPID;
fractional abcCoefficient[3] __attribute__ ((section (".
   xbss, bss, xmemory")));
fractional controlHistory[3] __attribute__ ((section (".
   ybss, bss, ymemory")));
fractional kCoeffs[] = {0,0,0};


void inicializarQEI(void);
void inicializarPWM(void);
//paco

/*
   ------------------------------------------------------------
```

```
   */

/*
 * The check task as described at the top of this file.
 */
//static void vCheckTask( void *pvParameters );

/*
 * Setup the processor ready for the demo.
 */
static void prvSetupHardware( void );

/*
   ----------------------------------------------------------
   */

/* The queue used to send messages to the LCD task. */
static xQueueHandle xLCDQueue;

/*
   ----------------------------------------------------------
   */

/*
 * Create the demo tasks then start the scheduler.
 */
int main( void )
{
  /* Configure any hardware required for this demo. */
  prvSetupHardware();
```

```
  /* Create the standard demo tasks. */
//  vStartBlockingQueueTasks( mainBLOCK_Q_PRIORITY );
//  vStartIntegerMathTasks( tskIDLE_PRIORITY );
//  vStartFlashCoRoutines( mainNUM_FLASH_COROUTINES );
//  vAltStartComTestTasks( mainCOM_TEST_PRIORITY,
   mainCOM_TEST_BAUD_RATE, mainCOM_TEST_LED );
//  vCreateBlockTimeTasks();

  /* Create the test tasks defined within this file. */
  //xTaskCreate( vCheckTask, ( signed char * ) "Check",
     mainCHECK_TAKS_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY, NULL );

  //paco
  //xTaskCreate(vUserTask, ( signed char * ) "UserLED1",
      configMINIMAL_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY, NULL);
  //xTaskCreate(vUserLED2Task, ( signed char * ) "
     UserLED2", configMINIMAL_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY - 1, NULL);
  xTaskCreate(vReferenciaTask, ( signed char * ) "
     Referencia", configMINIMAL_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY, NULL);
  //xTaskCreate(vControlVelTask, ( signed char * ) "
     ControlVel", configMINIMAL_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY + 1, NULL);
  //xTaskCreate(vControlPosTask_P, ( signed char * ) "
     ControlPos_P", configMINIMAL_STACK_SIZE, NULL,
     mainCHECK_TASK_PRIORITY + 1, NULL);
  xTaskCreate(vControlPosTask_PI, ( signed char * ) "
     ControlPos_PI", configMINIMAL_STACK_SIZE, NULL,
```

```
      mainCHECK_TASK_PRIORITY + 1, NULL);
  //paco

  /* Start the task that will control the LCD.  This
     returns the handle
  to the queue used to write text out to the task. */
//  xLCDQueue = xStartLCDTask();

  /* Start the high frequency interrupt test. */
//  vSetupTimerTest( mainTEST_INTERRUPT_FREQUENCY );

  /* Finally start the scheduler. */
  vTaskStartScheduler();

  /* Will only reach here if there is insufficient heap
     available to start
  the scheduler. */
  return 0;
}
//paco
static void vReferenciaTask(void *pvParameter) {
/* Used to wake the task at the correct frequency. */
portTickType xLastExecutionTime;
/* Initialise xLastExecutionTime so the first call to
   vTaskDelayUntil()
  works correctly. */
xLastExecutionTime = xTaskGetTickCount();
int a=1;
while(1)
  {
  /* Wait until it is time for the next cycle. */
```

```
  vTaskDelayUntil( &xLastExecutionTime,
     mainREFERENCIA_TASK_PERIOD );

  //PORTB=POS1CNT; // muestro por el puerto b el valor
     del registro de la cuenta
    switch (a)
    {
      case 1:   refvel=0.1; a=2;  break;
      case 2:   refvel=0.5; a=3;  break;
      case 3:   refvel=0.9; a=1;  break;
    }

  }
}
static void vControlVelTask(void *pvParameter) {
/* Used to wake the task at the correct frequency. */
portTickType xLastExecutionTime;
int ultimoQEItick;
int cicloQEIticks;
int actualQEItick;
/* Initialise xLastExecutionTime so the first call to
   vTaskDelayUntil()
  works correctly. */
xLastExecutionTime = xTaskGetTickCount();
ultimoQEItick = POS1CNT;
/*  POS1CNT - Registro donde esta la cuenta de pulsos
   del encoder.
  Cuidado porque cambia a 0 al llegar al MAX1CNT. */


/*
```

```
Step 1: Initialize the PID data structure, fooPID
*/
        fooPID.abcCoefficients = &abcCoefficient[0];
           /*Set up pointer to derived coefficients */
        fooPID.controlHistory = &controlHistory[0];
           /*Set up pointer to controller history
           samples */
        PIDInit(&fooPID);
           /*Clear the controler history and the
           controller output */
    kCoeffs[0] = Q15(0.99999999);
    kCoeffs[1] = Q15(0.0);
    kCoeffs[2] = Q15(0.0);
        PIDCoeffCalc(&kCoeffs[0], &fooPID);
           /*Derive the a,b, & c coefficients from the
           Kp, Ki & Kd */


while(1)
  {
  /* Wait until it is time for the next cycle. */
  vTaskDelayUntil( &xLastExecutionTime,
     mainCONTROL_TASK_PERIOD );

  //velocidad = cicloQEIticks / ciclo
  actualQEItick=POS1CNT;
  cicloQEIticks = ultimoQEItick - actualQEItick;//cuenta
      hacia abajo
  ultimoQEItick = actualQEItick;
  //motor 256 ranuras en 360 grados
  //PORTB=cicloQEIticks;
```

```
fooPID.controlReference = Q15(refvel);    /*Set the
   Reference Input for your controller */
  fooPID.measuredOutput = Q15(cicloQEIticks/0x3E);
     /*Typically the measuredOutput variable is a
     plant response*/
                    /*measured from an A/D input or a
                       sensor. */
                    /*In this example we manually set
                       it to some value for */
                    /*demonstration but the user
                       should note that this value
                       will */
                    /*keep changing in a real
                       application*/
//fooPID.controlOutput;
//Me falta escribir en P1DC1 el ciclo de trabajo.

PID(&fooPID);

//15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
//putc(fooPID.controlOutput);
PORTB = fooPID.controlOutput&0x3FF;

P1DC1 = fooPID.controlOutput&0x3FF;
//P1DC1 = fooPID.controlOutput&0x1FF;//pascual
//sentido=(fooPID.controlOutput&0x200)==0x200;//para
   el puente h
```

```
  }
}

static void vControlPosTask_P(void *pvParameter) {
int pos;
int error;
int u;
int k=3;
/* Used to wake the task at the correct frequency. */
portTickType xLastExecutionTime;
/* Initialise xLastExecutionTime so the first call to
   vTaskDelayUntil()
  works correctly. */
xLastExecutionTime = xTaskGetTickCount();
/*  POS1CNT - Registro donde esta la cuenta de pulsos
   del encoder.
  Cuidado porque cambia a 0 al llegar al MAX1CNT. */

while(1)
  {
  /* Wait until it is time for the next cycle. */
  vTaskDelayUntil( &xLastExecutionTime,
     mainCONTROL_TASK_PERIOD );

  //posicion
  pos = POS1CNT;//porque cuenta hacia abajo.
  PORTB=pos;
  //regulador P
  error = refpos-pos;
  if(error<0)error=0;
```

```
  u=k*error;

  //15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
  P1DC1=u&0x3FF;
  //P1DC1=u&0x3FF;
  //PORTB=u&0x3FF;
  }
}

static void vControlPosTask_PI(void *pvParameter) {
float pos;
float e;//error
float ek=0;//suma de todos los errores
float e_1=0;//error del instante de tiempo anterior
float Ts=0.1;//periodo de muestreo de la senyal
float u;//salida del controlador
float Kp=3;//ganancia proporcional
float Ki=0.1;//ganancia integral
float Kd=1;//ganancia derivativa


/* Used to wake the task at the correct frequency. */
portTickType xLastExecutionTime;
/* Initialise xLastExecutionTime so the first call to
   vTaskDelayUntil()
  works correctly. */
xLastExecutionTime = xTaskGetTickCount();
/*  POS1CNT - Registro donde esta la cuenta de pulsos
   del encoder.
  Cuidado porque cambia a 0 al llegar al MAX1CNT. */
```

```
while(1)
  {
  /* Wait until it is time for the next cycle. */
  vTaskDelayUntil( &xLastExecutionTime,
     mainCONTROL_TASK_PERIOD );

  //posicion
  pos = POS1CNT;//porque cuenta hacia abajo.
  PORTB=ek;

  //regulador PI
  e = refpos-pos;//error actual
  if(e<0)e=0;
  ek+=e;//acumulo todos los errores para el integral
  u=Kp*e+Ki*Ts*ek;
  //e_1=e;//me guardo el error para e(t-1)

  //15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
  P1DC1=u;
  //P1DC1=u&0x3FF;
  //PORTB=u&0x3FF;

  }
}
/*
   ------------------------------------------------------------
   */
/*
static void vUserLED1Task(void *pvParameter) {
while(1)
  {
```

```
    PORTCbits.RC1 = 1; // let it shine
    vTaskDelay(mainLED_DELAY);
    PORTCbits.RC1 = 0; // take a break
    vTaskDelay(mainLED_DELAY);
  }
}



/*-----------------------------------------------------------
   */
/*
static void vUserLED2Task(void *pvParameter) {
while(1)
  {
    PORTCbits.RC2 = 1; // let it shine
    vTaskDelay(mainLED_DELAY / 2);
    PORTCbits.RC2 = 0; // take a break
    vTaskDelay(mainLED_DELAY / 2);
  }
}

//paco
/*-----------------------------------------------------------
   */

static void prvSetupHardware( void )
{
  vParTestInitialise();

//paco
```

```
  inicializarQEI();
  inicializarPWM();

/*
  TRISCbits.TRISC1 = 0;//OUTPUT
  PORTCbits.RC1 = 0;//ALL OFF 0

  TRISCbits.TRISC2 = 0;//OUTPUT
  PORTCbits.RC2 = 0;//ALL OFF 0


*/
//paco
}
/*
   -----------------------------------------------------------
   */

static void vCheckTask( void *pvParameters )
{
/* Used to wake the task at the correct frequency. */
portTickType xLastExecutionTime;

/* The maximum jitter time measured by the fast
   interrupt test. */
extern unsigned short usMaxJitter ;

/* Buffer into which the maximum jitter time is written
   as a string. */
static char cStringBuffer[ mainMAX_STRING_LENGTH ];
```

```
/* The message that is sent on the queue to the LCD task
   .  The first
parameter is the minimum time (in ticks) that the
   message should be
left on the LCD without being overwritten.  The second
   parameter is a pointer
to the message to display itself. */
xLCDMessage xMessage = { 0, cStringBuffer };

/* Set to pdTRUE should an error be detected in any of
   the standard demo tasks. */
unsigned short usErrorDetected = pdFALSE;

  /* Initialise xLastExecutionTime so the first call to
     vTaskDelayUntil()
  works correctly. */
  xLastExecutionTime = xTaskGetTickCount();

  for( ;; )
  {
    /* Wait until it is time for the next cycle. */
    vTaskDelayUntil( &xLastExecutionTime,
       mainCHECK_TASK_PERIOD );

    /* Has an error been found in any of the standard
       demo tasks? */

    if( xAreIntegerMathsTaskStillRunning() != pdTRUE )
    {
      usErrorDetected = pdTRUE;
      sprintf( cStringBuffer, "FAIL #1" );
```

```
    }

    if( xAreComTestTasksStillRunning() != pdTRUE )
    {
      usErrorDetected = pdTRUE;
      sprintf( cStringBuffer, "FAIL #2" );
    }

    if( xAreBlockTimeTestTasksStillRunning() != pdTRUE )
    {
      usErrorDetected = pdTRUE;
      sprintf( cStringBuffer, "FAIL #3" );
    }

    if( xAreBlockingQueuesStillRunning() != pdTRUE )
    {
      usErrorDetected = pdTRUE;
      sprintf( cStringBuffer, "FAIL #4" );
    }

    if( usErrorDetected == pdFALSE )
    {
      /* No errors have been discovered, so display the
         maximum jitter
      timer discovered by the "fast interrupt test". */
      sprintf( cStringBuffer, "%dns max jitter", ( short
          ) ( usMaxJitter -
         mainEXPECTED_CLOCKS_BETWEEN_INTERRUPTS ) *
         mainNS_PER_CLOCK );
    }
```

```
    /* Send the message to the LCD gatekeeper for
       display. */
    xQueueSend( xLCDQueue, &xMessage, portMAX_DELAY );
  }
}
/*
   -----------------------------------------------------------
   */

void vApplicationIdleHook( void )
{
  /* Schedule the co-routines from within the idle task
     hook. */
  vCoRoutineSchedule();
}
/*
   -----------------------------------------------------------
   */
//paco
void inicializarQEI(void)
{
/*
The QEI module has the following four user-accessible
   registers. The
registers are accessible either in Byte mode or Word
   mode.
- QEIxCON: QEI Control Register - Controls QEI operation
    and provides status flags for
the state of the module.
- DFLTxCON: Digital Filter Control Register - Controls
   digital input filter operation.
```

```
- Position Count Register (POSxCNT) - Allows reading and
    writing of the 16-bit position
counter.
- Maximum Count Register (MAXxCNT) - Holds a value that
   is compared to the POSxCNT
counter in some operations.
*/
  /*
  QEI Phase A QEA RPINR14 QEAR[4:0]
  QEI Phase B QEB RPINR14 QEBR[4:0]
  QEI Index INDX RPINR15 INDXR[4:0]
  */

  //Seleccion de pines como entradas de la Fase A, Fase
     B e Indice
  RPINR14=0x1513; // RP19 QEA     RP21 QEB
  RPINR15=0x0014; // RP20 INDX

  TRISCbits.TRISC3 = 1;
    TRISCbits.TRISC4 = 1;
  TRISCbits.TRISC5 = 1;


  //POS1CNT = 0xFFFF;
  QEI1CONbits.QEIM = 0b101; // resolucion 4x, modo de
     Reset por MAXCNT
  QEI1CONbits.QEISIDL = 0; // continuar en modo idle (0)
  QEI1CONbits.SWPAB = 0; // Phase-A y Phase-B no
     intercambiados
  QEI1CONbits.PCDOUT = 1; // activado el pin de Position
      Counter Status
```

```
  QEI1CONbits.TQGATE = 0; // Timer gate apagado
  QEI1CONbits.TQCKPS = 0; // Prescaler 1:1
  QEI1CONbits.POSRES = 1; // Un pulso en INDEX hace un
     reset
  QEI1CONbits.TQCS =0; // Usamos clock interno para el
     timer
  QEI1CONbits.UPDN_SRC=1; // Phase-B indica direccion
  MAX1CNT=0xFFFF;//24

  DFLT1CON = 0x00F0; //filtro enabled

  TRISB=0x0000;
  TRISCbits.TRISC0 = 0;
  TRISCbits.TRISC1 = 0;

}
void inicializarPWM(void)
{
  P1TCON = 0;
  P1TMR = 0;
  // Compute Period based on CPU speed
  //P1TPER = 999; //(FCY/FPWM -1 ) ;
  P1TPER = 0x1FE; //pagina 34 Section 14 PWM

  P1SECMP = 0;

   PWM1CON1 = 0x0E11; // Enable PWM output pins and
      configure them as complementary mode
   PWM1CON2 = 0x0000;
   P1DTCON1 = 0x0000;
   P1DTCON2 = 0x0000;
```

```
    P1FLTACON = 0x0000;//PxFLTACON: Fault A Control
       Register

    //bit 15-0 PxDC1<15:0>: PWM Duty Cycle 1 Value bits
    P1DC1 = 0x0;
    //P1DC1 = 0x3DE;
    //P1DC1 = 0x3FE;//pagina 34 Section 14 PWM
    //P1DC1 = 1000

    //LA BUENA
    P1OVDCON=0Xff00;
    P1TCONbits.PTEN = 1;

  TRISBbits.TRISB8 = 0;
}
//paco
```

Bibliografía

[1] Wikipedia, *PID controller*.
http://en.wikipedia.org/wiki/PID_controller

[2] *Discretised PID Controllers*.
http://lorien.ncl.ac.uk/ming/digicont/digimath/dpid1.htm

[3] *Introduction to dsPIC33 programming*.
http://dangerousprototypes.com/docs/Introduction_to_dsPIC33_programming

[4] Microchip, *dsPIC 33F Ref Manual*.
http://ww1.microchip.com/downloads/en/DeviceDoc/70157D.pdf

[5] Microchip MPLAB C30, *C30 Compiler docs*. En los archivos del compilador C30.

[6] Microchip MPLAB C30, *C30 examples*. En los archivos del compilador C30.