Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Registro de datos masivo en plataformas SoC para aplicaciones de potencia

Autor: Ramón Heredero Bermejo

Tutor: Raúl Mateos Gil

2019

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería en Electrónica y Automática Industrial

Trabajo Fin de Grado

Registro de datos masivo en plataformas SoC para aplicaciones de potencia

Autor: Ramón Heredero Bermejo

Tutor: Raúl Mateos Gil

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

CALIFICACIÓN:

FECHA:

A mis padres, a mis hermanos y a mis abuelos…

AGRADECIMIENTOS

Para empezar, quiero dar las gracias a mi tutor Raúl por toda su ayuda, por el tiempo que me ha dedicado, por sus consejos y por sus ideas. Ha sido un placer poder trabajar junto a él y por ello le estoy enormemente agradecido.

No puedo olvidar una mención a mis compañeros de clase por todas las horas de estudio juntos en la biblioteca, los trayectos en el tren, las charlas y las risas en el comedor, pero sobre todo agradecerles su ayuda y apoyo durante estos años, sin ellos estoy seguro de que habría sido todo mucho más complicado, gracias amigos.

Para finalizar, un especial agradecimiento a mis padres por darme la oportunidad de realizar estos estudios, a mis hermanos por ser para mí una referencia a seguir de esfuerzo y dedicación y, por último, a mis abuelos por tener el privilegio de seguir disfrutando de ellos a tan avanzada edad.

RESUMEN

En este Trabajo de Fin de Grado (TFG) se presenta el planteamiento, el desarrollo y la validación de un sistema orientado a aplicaciones de potencia que permite el registro de datos de forma masiva, empleando una plataforma Zynq - 7000 (tarjeta de desarrollo ZedBoard de Xilinx).

El proyecto implementa un SoC que está dividido en dos grandes bloques complementarios entre sí y que forman el sistema completo final.

El primer bloque consiste en un sistema cuya finalidad es la generación de datos de forma dinámica. Representa un posible equipo de potencia del que se desea realizar un registro de datos. Se trata de un control de velocidad sobre un motor DC que aporta datos como, por ejemplo, la referencia de velocidad deseada, la velocidad real del motor, el error soportado, el número de muestra, etc. Para implementar este primer bloque se desarrollan dos periféricos HW a medida empleando el lenguaje VHDL.

El segundo bloque del proyecto consiste en un sistema capaz de almacenar dichos datos en una memoria no volátil, en este caso se ha empleado una memoria SD card. Para ello se crea un documento de texto con extensión TXT que se guarda en la memoria SD y sobre el cuál se escriben los datos que desean ser registrados. Finalmente, para gobernar el controlador de memoria se hace uso de las funciones de la librería FatFs proporcionada por el fabricante del dispositivo.

Palabras Claves: Registro de datos, SoC, ZedBoard, equipos de potencia, memoria SD card, control de velocidad.

ABSTRACT

The purpose of this Project is the development and the validation of a data log system to store data massively for power system applications. The system created is based in a Zynq – 7000 device (ZedBoard, Xilinx).

The Project implements a SoC divided into two large defined and complementary parts of a whole that creates the complete final system.

The first section involves a system created to generate data dynamically. It represents a possible power system to be datalogged. It is about a speed control on a DC motor that generates data, such as speed refereance, error or sample number. To implement this first section, VHDL lenguaje is used to develop two customized HW peripherals.

The second section it is about a system that is able to store the datastream in a non-volatile memory. In this case, a SD memory card has been used. The system uses this memory to create a text file with TXT extension where the datastream is written. Finally, the system resort to FatFs library proportioned by Xilinx to manage the SD host controller.

Keywords: Data logging, SoC, ZedBoard, power systems, SD memory card, speed control.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

GLOSARIO DE ACRÓNIMOS Y ABREVIATURAS

SoC: System on Chip

FPGA: Field Programmable Gate Array

SW: Software

HW: Hardware

VHDL: proviene de la combinación de VHSIC y HDL

VHSIC: Very High Speed Integrated Circuit

HDL: Hardware Description Lenguage

DC: Direct Current

API: Application Programming Interface

TXT: Extensión de un archivo de texto formado por caracteres legibles para el ser humano

SD: Secure Digital

SDIO: Secure Digital Input Output

PC: Personal Computer

PS: Processing System

PWM: Pulse-Width Modulation

ARM: Arquitectura RISC para procesadores

RISC: Reduced Instruction Set Computer

UART: Universal Asynchronous Receiver-Transmitter

GPIO: General Purpose Input/Output

LED: Light-Emitting Diode

AXI: Advanced eXtensible Interface. Se trata de uno de los tipos de bus de la especificación AMBA de procesadores ARM.

AMBA: Advanced Microcontroller Bus Architecture

FSM: Finite State Machine

GIC: Generic Interrupt Controller

PL: Programmable Logic (También denominado Fabric)

IRQ_F2P: Interrupt Request_Fabric to Processing System

SPI: Shared Peripheral Interrupts

IER: Interrupt Enable Register

ISR: Interrupt Status Register

SDK: Software Development Kit

FatFs: File Allocation Table File System

WP: Write Protect Pin

CD: Card Detected

MIO: Multiplexed Input Output

EOF: End Of File

BSP: Board Support Package

CAPÍTULO 1: INTRODUCCIÓN

El uso de SoC en aplicaciones de potencia requiere frecuentemente el registro de los datos correspondientes a la evolución del equipo controlado. La solución tradicional empleada en los sistemas de data logging consiste en enviar dichos datos a través de una conexión Ethernet o similar hacia un computador que sirve de lugar de almacenamiento. Pueden darse ocasiones en las cuáles esta solución no es la óptima o simplemente no es posible llevarla a cabo.

En este proyecto se propone y se desarrolla una alternativa que consiste en implementar un sistema capaz de realizar el registro de datos de forma embebida empleando una memoria SD como lugar de almacenamiento.

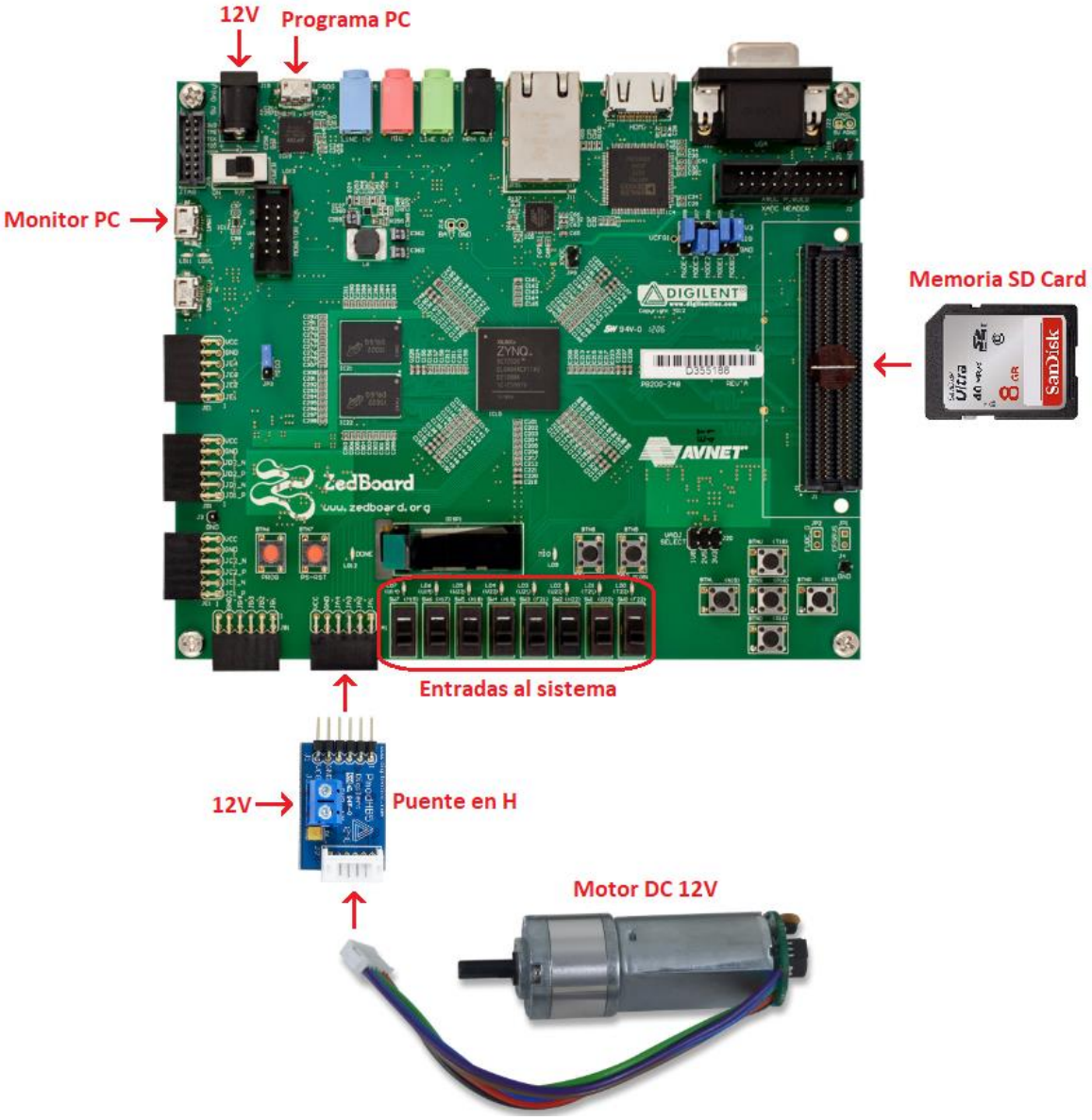


Figura 1. Sistema completo basado en ZedBoard

El sistema propuesto puede dividirse en dos grandes bloques: el control de velocidad sobre el motor DC y el registro de datos en la memoria SD. Ambos bloques tienen funcionalidades muy distintas pero que se complementan entre sí para formar el sistema final. Además, el sistema se rige por una interrupción periódica configurable que une el funcionamiento de ambos bloques y que es la base para el cálculo de la velocidad, el muestreo para el control PI y el registro de datos en la memoria SD.

El primero de estos dos grandes bloques consiste en la implementación de un control de velocidad sobre un motor DC siendo necesario el desarrollo de dos periféricos a medida empleando el lenguaje VHDL: PWMcore y ENCcore.

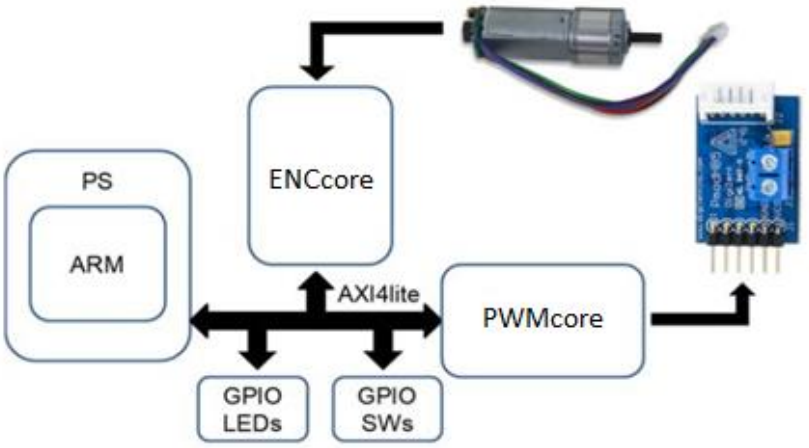


Figura 2. Diagrama de bloques para el control de velocidad [SEDA, 2018]

El periférico PWMcore está destinado a la generación de una señal PWM para el control de un puente en H que alimenta a un motor DC, mientras que el periférico ENCcore tiene como función la lectura de las señales digitales del encoder magnético montado sobre el rotor con el objetivo de calcular la velocidad de giro del motor DC en el microprocesador. Ambos periféricos tienen asociados una serie de registros de configuración y de control que permiten al microprocesador interactuar con el HW para el correcto funcionamiento del sistema.

Una vez finalizado todo el desarrollo del sistema a nivel HW se empieza a trabajar en el código C que ejecutará el microprocesador y en el cuál, entre otras muchas cosas, se incluye el algoritmo de control en lazo cerrado, tal y como se indica en la *Figura 3.* Este lazo consiste en un control PI de velocidad cuyo objetivo es

conseguir que el motor DC acabe finalmente girando a la velocidad que se establece como referencia.

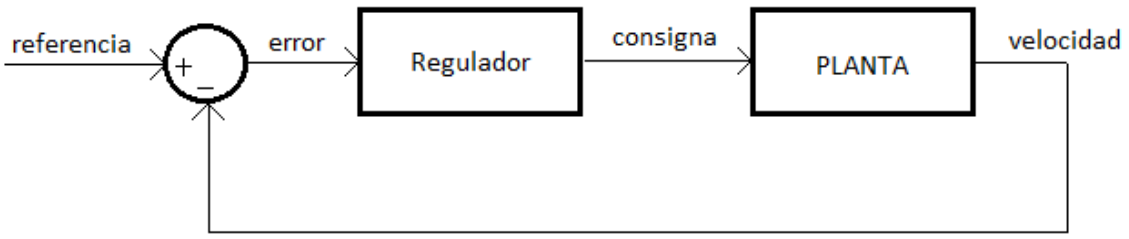


Figura 3. Esquema de control en lazo cerrado

En la *Figura 1* se pueden observar 8 microinterruptores (con el nombre de [SW7…SW0]) que sirven como entradas hacia el sistema. El microinterruptor SW1 determina el tipo de control que se realiza sobre el motor, es decir, si el sistema se encuentra operando en lazo abierto o en lazo cerrado.

Cuando el sistema opera en lazo abierto, el código introducido en los microinterruptores [SW7…SW2] se interpreta como valor de tensión a aplicar al motor en el rango 0-12 V y se envía la velocidad del motor a la pantalla del PC a través de la UART1.

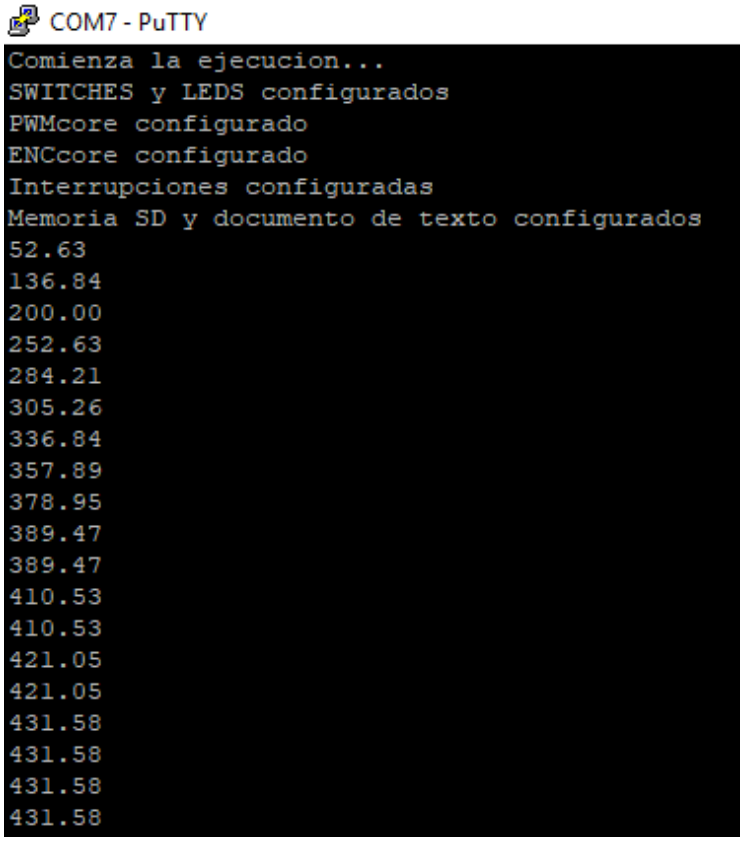
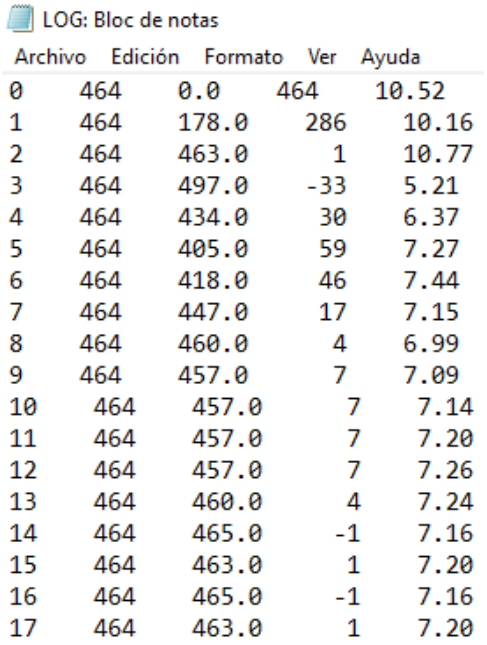


Figura 4. Información enviada al terminal del PC

Si por el contrario el sistema opera en lazo cerrado, el código introducido en los microinterruptores es interpretado como valor de referencia que debe emplear el algoritmo de control para actuar sobre el motor con la consigna apropiada. En este caso, se realiza el registro en la memoria SD de los siguientes datos: número de muestra, referencia (rpm), velocidad real (rpm), error (rpm) y consigna (voltios). Del registro en la memoria SD se encarga el segundo de los dos grandes bloques que forman el sistema y del que se habla a continuación.


LOG: Bloc de notas

Archivo Edición Formato Ver Ayuda

0	464	0.0	464	10.52
1	464	178.0	286	10.16
2	464	463.0	1	10.77
3	464	497.0	-33	5.21
4	464	434.0	30	6.37
5	464	405.0	59	7.27
6	464	418.0	46	7.44
7	464	447.0	17	7.15
8	464	460.0	4	6.99
9	464	457.0	7	7.09
10	464	457.0	7	7.14
11	464	457.0	7	7.20
12	464	457.0	7	7.26
13	464	460.0	4	7.24
14	464	465.0	-1	7.16
15	464	463.0	1	7.20
16	464	465.0	-1	7.16
17	464	463.0	1	7.20

Figura 5. Datos almacenados en la memoria SD

El segundo de los dos grandes bloques tiene la función de almacenar en la memoria SD los datos que se desean registrar. Para ello, el sistema crea un archivo de texto con extensión TXT que se almacena en la memoria SD y sobre el que se van a escribir todos los datos que desean ser guardados, como se puede apreciar en la *Figura 5.*

Para este proceso se recurre a las funciones API que nos aporta el controlador de memoria SD que posee el PS. Las funciones empleadas son las necesarias para configurar el controlador y la memoria SD, crear un archivo de texto TXT, abrir el archivo, cerrar el archivo, moverse a lo largo de éste, etc. Como ejemplo de estas funciones tenemos:

- f_open()
- f_close()

- f_write()
- f_mount()
- f_lseek()

Para acabar, una vez finalizado un registro, el usuario dispone de un conjunto de datos que, entre otras posibles acciones, pueden ser analizados de forma analítica o, si el registro no es muy extenso, realizar un análisis gráfico que permita visualizar el conjunto de los datos de forma global e instantánea.

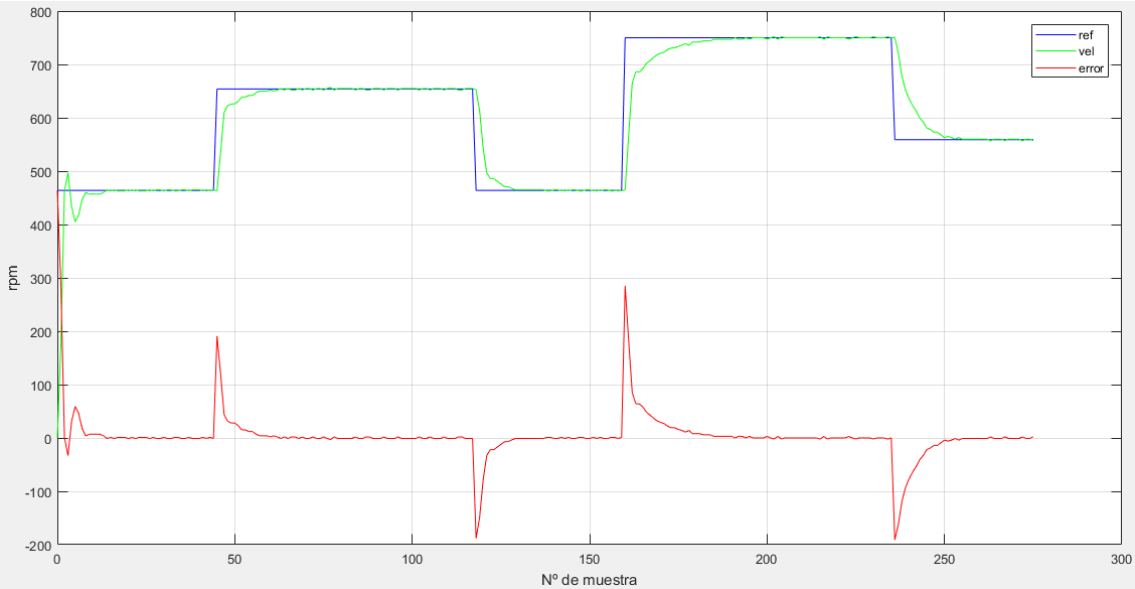


Figura 6. Representación gráfica en Matlab de los datos de un registro

La *Figura 6* muestra el resultado de representar gráficamente mediante un script en Matlab los datos almacenados en el archivo de texto que aparece en la *Figura 5.* Para generar el eje de abscisas se emplea el número de muestra que corresponde con la primera columna del archivo de texto, dando así lugar a la base de tiempos. Por otro lado, en el eje de ordenadas se representan las tres variables principales del control PI: la referencia en color azul, la velocidad real en verde y el error en rojo, que se corresponden con la segunda, tercera y cuarta columna respectivamente.

CAPÍTULO 2: CONTROL DE VELOCIDAD SOBRE MOTOR DC

Este capítulo tiene como finalidad realizar un control PI sobre un motor DC. Para ello se implementa un sistema SoC formado principalmente por dos periféricos diseñados a medida: PWMcore y ENCcore (ver *Figura 2*). El primero de ellos está destinado a la generación de una señal PWM que controla un puente en H para alimentar al motor DC y el segundo para la lectura de las señales digitales del encoder magnético con el objetivo de calcular la velocidad de giro del motor en el microprocesador.

El desarrollo del sistema de control está dividido en dos etapas: control en lazo abierto y control PI en lazo cerrado. Cuando el sistema se encuentra operando en lazo abierto, el microprocesador envía la velocidad de giro al monitor del PC empleando la UART1. Si por el contrario está activado el control en lazo cerrado, se crea un archivo de texto en la memoria SD donde se almacenan las variables asociadas al control (número de muestra, referencia, velocidad, error y consigna).

Para el diseño del sistema se ha elegido una tarjeta de desarrollo Zedboard, basada en la arquitectura Zynq - 7000 de Xilinx, como se puede apreciar en la *Figura 1.*

Además de los 2 periféricos ya mencionados, se van a emplean dos módulos GPIO para la lectura de los 8 microinterruptores y su posterior escritura en los 8 LEDs. El microinterruptor SW0 determina el sentido de giro del motor, el SW1 el tipo de control aplicado y los 6 microinterruptores de mayor peso (desde el SW2 al SW7) van a determinar el ancho de pulso de la señal PWM en lazo abierto y la referencia de velocidad en lazo cerrado.

2.1 Control de velocidad en lazo abierto

El primer paso en el diseño del sistema SoC consiste en la realización del conexionado de los distintos componentes hardware que se van a utilizar para definir el sistema completo, empleando para ello la herramienta Vivado 2016.4.

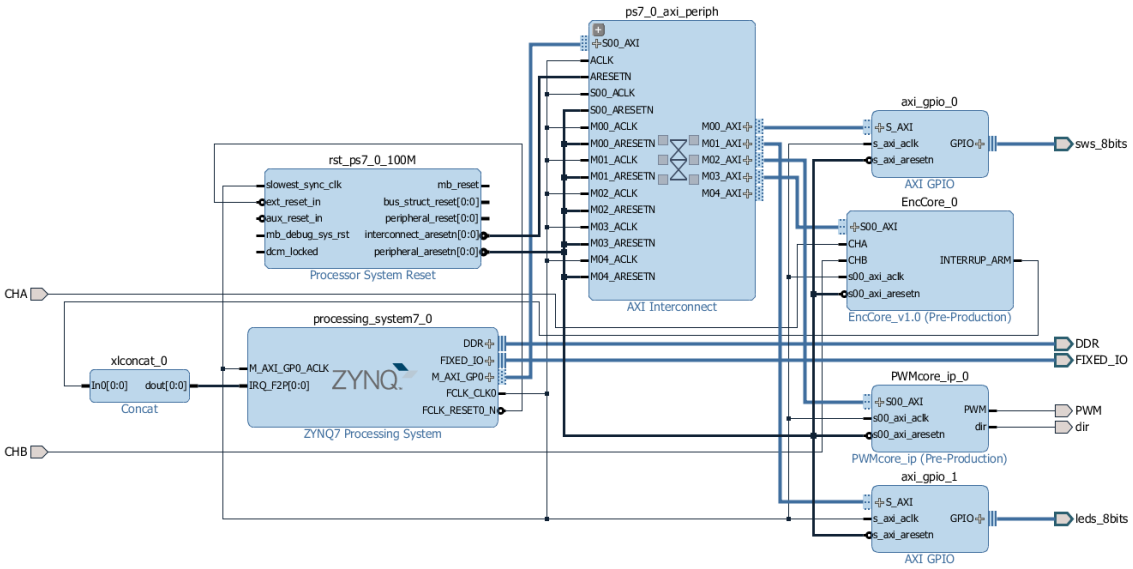


Figura 7. Diagrama de bloques del SoC en Vivado 2016.4

En el diagrama de la *Figura 7* se pueden apreciar los siguientes bloques: un procesador Zynq, los dos módulos GPIO (GPIO0 destinado a leer los microinterruptores y GPIO1 a escribir sobre los LEDs), el periférico PWMcore (envía las señales *PWM* y *dir* al puente en H para controlar la velocidad y el sentido de giro del motor), el periférico ENCcore (lee las dos señales digitales del encoder y desencadena una interrupción periódica para que el microprocesador calcule la velocidad) y un módulo de bus AXI (encargado de comunicar el microprocesador con el resto de los bloques).

El sistema consta de:

- 3 entradas: *sws_8bits* (los 8 microinterruptores de la Zedboard) y *CHA* y *CHB* (señales digitales que genera el encoder magnético del motor DC).
- 3 salidas: *leds_8bits* (los 8 LEDs de la Zedboard), *PWM* (señal PWM con tiempos muertos incluidos dirigida al puente en H) y *dir* (señal de sentido de giro dirigida al puente en H).

Como ya se ha comentado con anterioridad, cada vez que el microprocesador calcula la velocidad de giro tiene que enviarla al monitor del PC empleando para ello la UART1. Por lo tanto, se debe abrir la configuración de los parámetros del microprocesador Zynq para activar los periféricos entrada/salida que se van a necesitar en el diseño. En concreto, se va a activar la UART1 para que la Zedboard se comunique con el monitor del PC y GPIO MIO para leer los microinterruptores y activar los correspondientes LEDs.

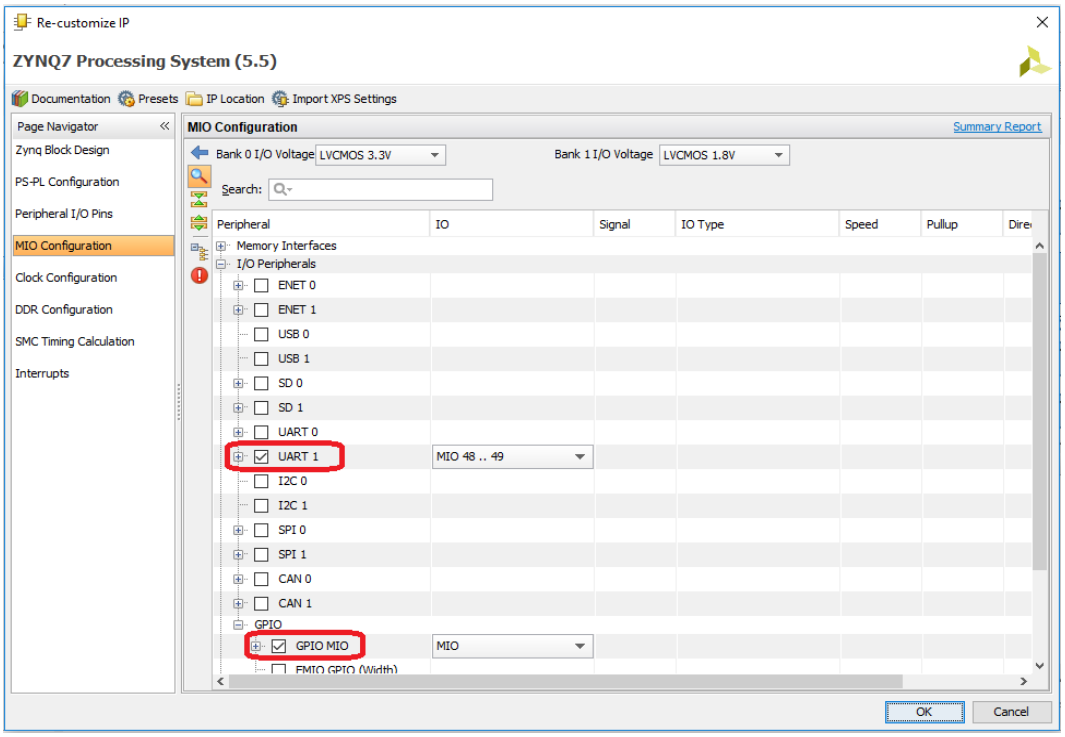


Figura 8. Activación de UART1 y GPIO MIO

En cuanto a HW externo se refiere, el puente en H implementado en el PmodHB5 de Digilent se va a conectar en la parte superior del conector Pmod JA1 (situado justo a la izquierda de los microinterruptores).

Pmod	Signal Name	Zynq pin
JA1	JA1	Y11
	JA2	AA11
	JA3	Y10
	JA4	AA9
	JA7	AB11
	JA8	AB10
	JA9	AB9
	JA10	AA8

Figura 9. Pines I/O para el puente en H [ZedBoard, 2014]

Estos cuatro pines se van a utilizar para conectar las señales de entrada (*CHA* y *CHB*) y las señales de salida (*PWM* y *dir*). Esta acción queda configurada mediante el archivo *conexiones.xdc* mostrado a continuación:

```
1 set_property PACKAGE_PIN Y11     [get_ports dir]
2 set_property IOSTANDARD LVCMOS33 [get_ports dir]
3
4 set_property PACKAGE_PIN AA11    [get_ports PWM]
5 set_property IOSTANDARD LVCMOS33 [get_ports PWM]
6
7 set_property PACKAGE_PIN Y10     [get_ports CHA]
8 set_property IOSTANDARD LVCMOS33 [get_ports CHA]
9
10 set_property PACKAGE_PIN AA9     [get_ports CHB]
11 set_property IOSTANDARD LVCMOS33 [get_ports CHB]
```

Figura 10. Señal I/O y pin correspondiente del dispositivo Zynq

Se ha determinado que las señales de salida *PWM* y *dir* se conectan a los pines AA11 y Y11, y las señales de entrada *CHA* y *CHB* a los pines Y10 y AA9, respectivamente.

Una vez llegados a este punto, se debe desarrollar la estructura y definir el funcionamiento de los dos periféricos a medida basándonos en las especificaciones del proyecto. En primer lugar, se procede a desarrollar el periférico PWMcore y validar su funcionamiento mediante simulación. Después, se hace lo mismo con el segundo periférico ENCcore. Finalmente, se desarrolla un código en lenguaje C para controlar el sistema completo y darle funcionalidad a nuestro SoC.

2.1.1 Desarrollo del periférico PWMcore

La función de este periférico es determinar la tensión que recibe el motor DC en sus terminales de alimentación mediante una señal PWM, controlando no solamente el valor de la componente continua de la tensión sino también su polaridad, de esta manera, podemos modificar la velocidad y el sentido de giro del motor.

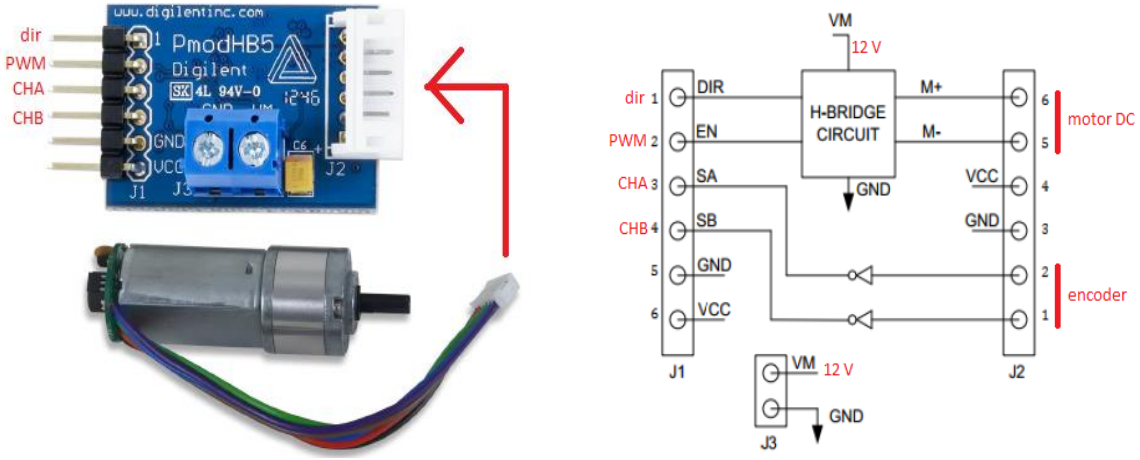


Figura 11. Conexionado motor DC, encoder, PmodHB5 y ZedBoard

El control del puente en H se reduce por tanto a una señal de sentido de giro (*dir*) y una señal de enable (*PWM*).

El motor empleado es un motor con tensión nominal de 12 V, por lo que se necesita utilizar una fuente de alimentación de 12 V para alimentar el puente en H y disponer así de todo su rango de funcionamiento.

Una característica muy importante del puente en H y que se debe de tener en cuenta a la hora del diseño del periférico, es la inclusión de tiempos muertos cuando se produce una conmutación en el sentido de giro para evitar que los 2 transistores de una misma rama se encuentren en conducción momentánea y se produzca un cortocircuito que pueda destruir el puente. Esto sucede porque el tiempo de apertura t_{OFF} (tiempo de paso de conducción a corte) de los transistores que forman el puente en H es mayor que su tiempo de cierre t_{ON} (tiempo de paso de corte a conducción) debido a sus características eléctricas.

La solución empleada para afrontar este problema es la siguiente: cuando se detecta que se desea cambiar el sentido de giro se forzará a los 4 transistores a estar en corte durante un tiempo muerto (mediante *PWM* = '0') para después conmutar la señal *dir* que recibe el puente en H y dotar a la señal *PWM* el valor que le corresponde.

Este periférico será gestionado desde el microprocesador mediante una serie de registros, en concreto, serán necesarios 4 registros: 2 de configuración y 2 de control.

Registros de configuración:

- *carrier_period*[31:0]: permite establecer el periodo de la señal portadora de la modulación PWM mediante un número de cuentas de la señal CLK de 100 MHz. Determina la frecuencia de la señal *PWM*.
- *deadtime*[15:0]: permite establecer el valor de los tiempos muertos mediante un número de cuentas de la señal CLK de 100 MHz. Determina el valor de los tiempos muertos.

Registros de control:

- *direction*[0]: permite establecer el sentido de giro mediante el microinterruptor SW0, es el valor que adoptará la señal de salida *dir*. Determina el sentido de giro del motor DC.
- *mod_value*[31:0]: permite establecer el valor de la señal moduladora mediante los microinterruptores SW2 a SW7. Determina el ancho de pulso de la señal *PWM* y, por tanto, la tensión aplicada sobre el motor DC.

2.1.1.1 Desarrollo HW de PWMcore

El periférico se va a dividir en varios bloques para simplificar su diseño:

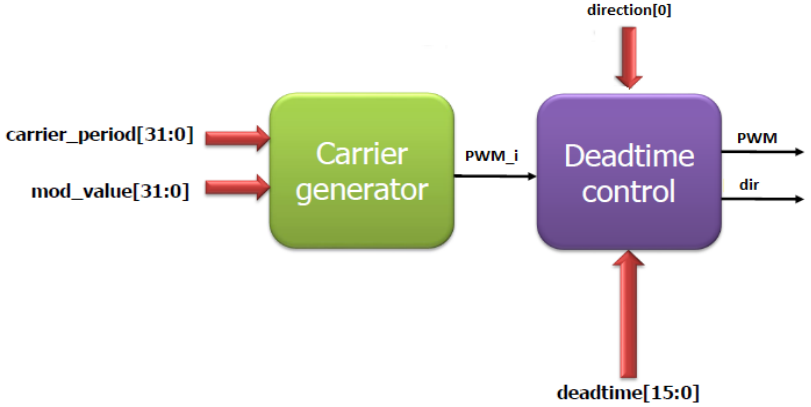


Figura 12. Diagrama de bloques de PWMcore [SEDA, 2018]

El periférico consta de 2 bloques:

- *Carrier generator*: a partir de los valores de *carrier_period* (determinado por el microprocesador) y *mod_value* (determinado por los microinterruptores SW7…SW2) se realiza una modulación PWM y se obtiene la señal *PWM_i*, es decir, la señal *PWM* deseada a la que se le deben añadir los tiempos muertos.
- *Deadtime control*: a partir de la señal *PWM_i*, *deadtime* (determinado por el microprocesador) y *direction* (determinado por el microinterruptor SW0) se genera una señal *PWM* con tiempos muertos incluidos y la señal *dir*, ambas conectadas al puente en H. Cuando se detecta un cambio en *direction*, *PWM* se fuerza a nivel bajo, se espera el tiempo *deadtime*, se conmuta *dir* y se actualiza el valor de *PWM*, todo ello para evitar un mal funcionamiento o incluso la destrucción del puente en H, como se explicó anteriormente.

El desarrollo HW se encuentra comentado al detalle en el archivo *PWMcore.vhd*.

Finalmente, se analiza la comunicación del periférico con el microprocesador a través del bus AXI en el archivo *PWMcore_ip_v1_0_S00_AXI.vhd*:

```
-- Add user logic here
PWMcore_1: entity work.PWMcore
  port map (
    mclk           => S_AXI_ACLK,      --señal de reloj bus AXI
    mrst           => S_AXI_ARESETN,   --señal reset bus AXI, activa a LOW
    direction      => slv_reg0(0),
    carrier_period => slv_reg1,
    mod_value      => slv_reg2,
    deadtime       => slv_reg3(15 downto 0),
    dir            => dir,
    PWM            => PWM);
-- User logic ends
```

Figura 13. Comunicación con PWMcore mediante bus AXI

El microprocesador se comunica con el periférico PWMcore a través de 4 registros del bus AXI, todos ellos de escritura, es decir, solamente el microprocesador escribe sobre ellos, nunca el periférico. Estos registros son: *slv_reg0*, *slv_reg1*, *slv_reg2* y *slv_reg3*.

Se ha determinado que *slv_reg0* corresponde al registro *direction*, *slv_reg1* al registro *carrier_period*, *slv_reg2* al registro *mod_value* y *slv_reg3* al registro *deadtime*. De esta manera, por ejemplo, si se escribe en el registro *slv_reg3* un valor de 5000, el periférico interpreta que debe generar tiempos muertos de 50 microsegundos (5000 x 10 ns).

Para acabar, el archivo que contiene todo el desarrollo HW del periférico es *PWMcore_ip_v1_0.vhd*, de manera que el encapsulado del periférico resulta así:

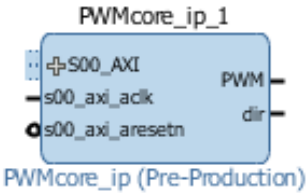


Figura 14. Bloque PWMcore

Nota: todos los archivos fuente utilizados se incluyen en la documentación del proyecto.

2.1.1.2 Simulación y validación de PWMcore

Para simular el comportamiento del periférico creado y validar su funcionamiento se ejecuta el testbench *PWMcore_tb* con la herramienta Model SIM.

En el testbench se realizan las siguientes acciones:

- Generar una señal *mclk* de 10 ns de periodo (de 100 MHz)
- Reset al inicio de la simulación
- Producir un cambio en *direction* en el instante 600 ns
- Asignar valor 16 a *carrier_period*, 7 a *mod_value* y 10 a *deadtime*

El resultado que se obtiene de la simulación es el siguiente:

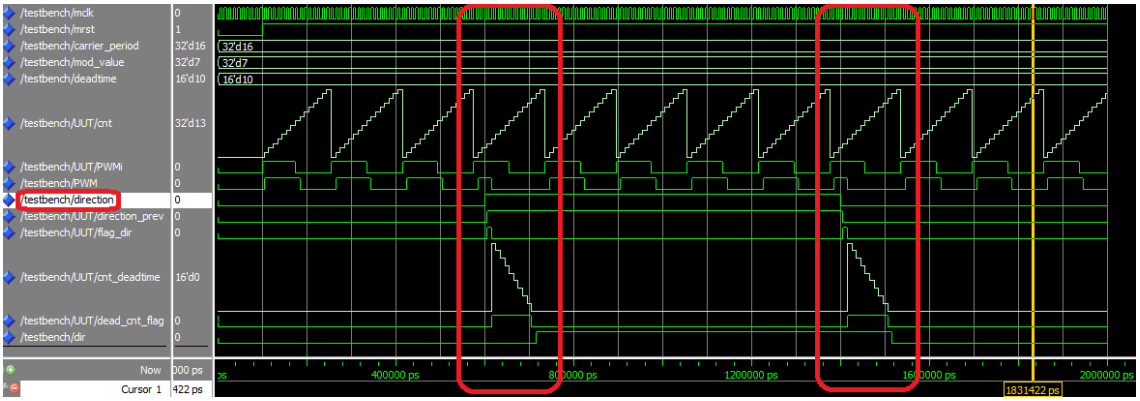


Figura 15. Resultado de la simulación de PWMcore

Se observa lo siguiente:

Al producirse un cambio en *direction*, en el siguiente flanco de subida de *mclk* se actualiza *direction_prev* y se detecta el cambio en *direction* activando *flag_dir*. Al detectar que *flag_dir* se ha activado, *PWM* adquiere valor 0 y se activa *dead_cnt_flag* comenzando a descontar el tiempo muerto. Al finalizar este tiempo muerto, *dead_cnt_flag* se desactiva, por lo que ya ha pasado el *deadtime* y se puede fijar *PWM* al valor que le corresponde y conmutar *dir*.

Por lo tanto, se puede afirmar que el periférico funciona correctamente y cumple con los requerimientos de diseño.

2.1.2 Desarrollo del periférico ENCcore

La función de este segundo periférico es obtener la información necesaria del encoder magnético para que el microprocesador pueda calcular la velocidad de giro del motor.

Figura 16. Encoder magnético acoplado al eje del motor DC

El encoder proporciona dos señales digitales que el periférico debe interpretar de tal manera que el microprocesador sea capaz de calcular la velocidad del motor.

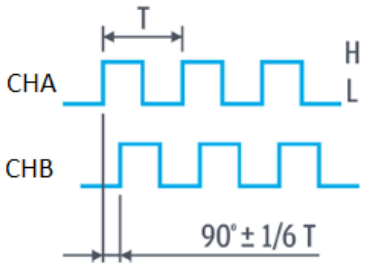


Figura 17. Señales digitales del encoder magnético

Si retrocedemos a la *Figura 11* se puede ver que estas dos señales digitales están conectadas a *CHA* y *CHB*, puertos de entrada del periférico ENCcore.

Generación de las señales: cada vuelta que realiza el encoder, genera un periodo T de la señal digital. El encoder proporciona 2 señales iguales pero desfasadas (el dispositivo dispone de 2 sensores magnéticos colocados a 90 grados) para poder conocer el sentido de giro del motor, es decir, si el flanco de subida de la señal *CHA* se produce antes que el de la señal *CHB* girará en un sentido y si es al revés, girará en el sentido contrario.

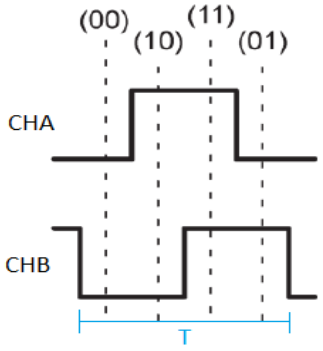


Figura 18. Combinaciones posibles en un periodo T

Dentro de un periodo de las señales *CHA* y *CHB* existen 4 posibles combinaciones de valores: 00, 01, 10 y 11.

En el periférico se va a implementar una máquina de estados que va a ir detectando cada uno de los posibles valores en los que se encuentran las señales *CHA* y *CHB*.

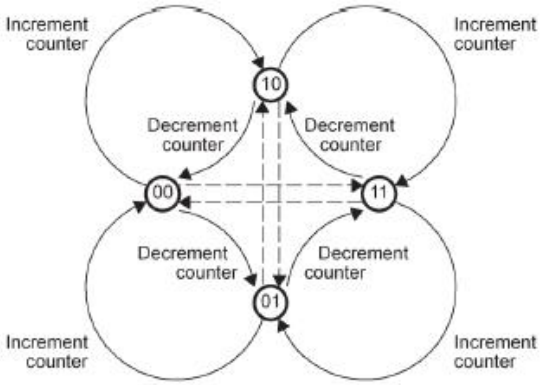


Figura 19. Máquina de estados de ENCcore [SEDA, 2018]

De esta manera, cada vez que se pasa de un estado a otro de las señales *CHA* y *CHB* se va a incrementar o decrementar un contador denominado *cnt_FSM*, en función de qué señal está retrasada frente a la otra, es decir, en función del sentido de giro del motor.

En resumen: el periférico va a contar el número de transiciones de estado durante un intervalo de tiempo predeterminado por un registro de configuración (por ejemplo, 1 segundo). Al finalizar este tiempo, el periférico activa la señal de alerta *INTERRUP_ARM_i* que desencadena una interrupción en el microprocesador para que este lea el valor del contador *cnt_FSM*.

Nota: en cada periodo T de las señales *CHA* y *CHB* se producen 4 transiciones de estado, es decir, cada vuelta del motor corresponde a 4 cuentas. Además, el motor tiene una reductora 1:19, de manera que cada 19 vueltas del motor la salida de la reductora da solamente 1 vuelta.

De este modo, si al finalizar el periodo de cuenta (que era 1 segundo) el valor de *cnt_FSM* que lee el microprocesador es, por ejemplo 634 cuentas, quiere decir que el encoder ha girado $\frac{634\ cuentas}{4\ cuentas/vuelta}$ = 158 vueltas, por lo tanto, el eje del motor gira a 158 revoluciones por segundo, o lo que es lo mismo, a 9500 rpm.

Aplicando el coeficiente de reducción 1:19, tenemos que la salida de la reductora gira a 500 rpm.

Este periférico va a estar organizado en torno a 4 registros: uno de configuración, uno de propósito general, uno de control y uno de estado.

Registro de configuración:

- *obs_period*[31:0]: registro de escritura que permite establecer el periodo de observación del contador *cnt_FSM*. Determina la frecuencia con la que el microprocesador es interrumpido. Es un valor conocido y necesario para calcular la velocidad.

Registro de propósito general:

- *enc_cnt*[23:0]: registro de lectura que permite al microprocesador conocer el valor del contador *cnt_FSM*. Es necesario para calcular la velocidad de giro del motor.

Registro de control:

- *IER*[0]: registro de escritura que permite habilitar/deshabilitar la fuente de interrupción *INTERRUP_ARM*, es decir, se trata del enable de la interrupción.

Registro de estado:

- *ISR*[0]: registro de lectura/escritura que permite conocer el estado de la interrupción y, además, permite borrar el flag de esta para indicar que ya ha sido atendida.

2.1.2.1 Desarrollo HW de ENCcore

El periférico se va a dividir en varios bloques para simplificar su diseño:

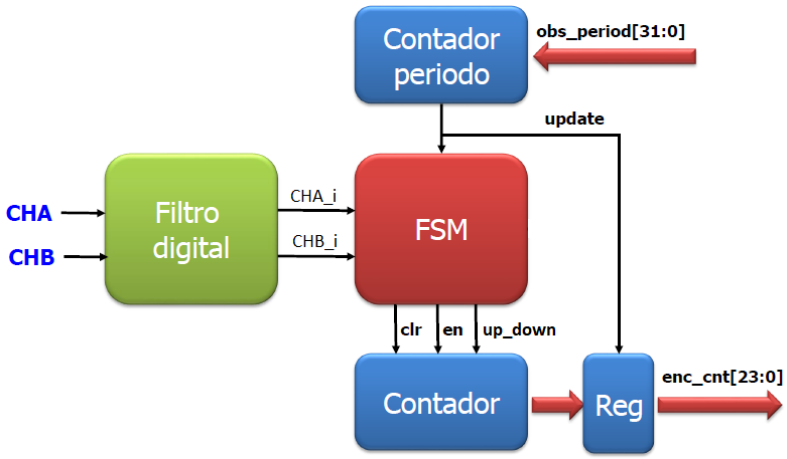


Figura 20. Diagrama de bloques de ENCcore [SEDA, 2018]

El periférico consta de 5 bloques:

- *Filtro digital*: se encarga de eliminar los posibles rebotes que presentan las señales que genera el encoder magnético. A partir de *CHA* y *CHB* se obtienen dos señales libres de rebotes: *CHA_i* y *CHB_i*.

  Su funcionamiento es sencillo: cada periodo de la señal *CLK* se muestrea el valor de la señal digital de entrada y hasta que este valor no permanece constante durante 4 periodos de *CLK* no se cambia la salida.

  Se encuentra implementado en el archivo *FILTRO_DIGITAL.vhd*.

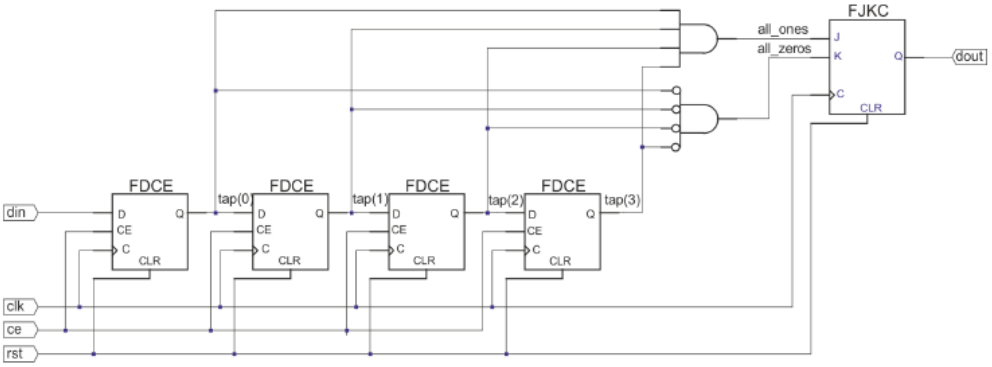


Figura 21. Estructura interna del filtro digital [SEDA, 2018]

- *Contador periodo*: su función es determinar el periodo de observación del periférico y la frecuencia de la interrupción periódica. A partir del valor de *obs_period* (determinado por el microprocesador) se activa la señal *update* de forma periódica transcurrido el tiempo de observación indicado.

- *FSM*: se trata de la máquina de estados que gobierna el funcionamiento del siguiente contador. A partir de las señales filtradas *CHA_i* y *CHB_i*, cada vez que se produce una transición de estado (00, 10, 11, 01) se activa la señal *en* y se asigna el valor correspondiente a la señal *up_down* (1 = up; 0 = down). Además, cada vez que se activa la señal *update*, es activada la señal *clr* y el contador sufre un reset.
- *Contador*: cada vez que se activa la señal *en* se incrementa o decrementa su valor en una unidad en función del valor de la señal *up_down*. Al activarse *clr* se realiza un reset y su valor vuelve a cero.
- *Reg*: cada vez que se activa la señal *update* registra el valor del contador antes de que este sea puesto a cero para que el microprocesador tenga disponible el valor de cuenta de la señal *enc_cnt*.

El desarrollo HW se encuentra comentado al detalle en el archivo *ENCODERcore.vhd*.

Finalmente, se analiza la comunicación del periférico con el microprocesador a través del bus AXI en el archivo *EncCore_v1_0_S00_AXI.vhd*:

```
signal INTERRUP_ARM_i    : std_logic;
signal enable_interrup   : std_logic;
signal reg_wr_addr       : std_logic_vector(1 downto 0);
signal int_clr           : std_logic;

    -- Add user logic here
    ENCODERcore_1: entity work.ENCODERcore
  port map (
    mclk           => S_AXI_ACLK,
    mrst           => S_AXI_ARESETN,
    obs_period     => slv_reg0,
    CHA            => CHA,
    CHB            => CHB,
    enc_cnt        => slv_reg1(23 downto 0),
    INTERRUP_ARM_i => INTERRUP_ARM_i);

slv_reg1(31 downto 24)<= (others => slv_reg1(23));

enable_interrup <= slv_reg2(0);  --IER es bit 0 de slv_reg2

reg_wr_addr <= axi_awaddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
int_clr <= (slv_reg_wren and S_AXI_WSTRB(0) and S_AXI_WDATA(0)) when (reg_wr_addr = "11") else '0';


process (S_AXI_ACLK, S_AXI_ARESETN) is  --incluimos efecto de IER en update
begin
  if (S_AXI_ARESETN = '0') then
    INTERRUP_ARM <= '0';
    slv_reg3 <= (others =>'0');
  elsif (S_AXI_ACLK'event and S_AXI_ACLK = '1') then
    if (INTERRUP_ARM_i = '1' and enable_interrup = '1') then
      INTERRUP_ARM <= '1';
      slv_reg3(0) <= '1';       --ISR se pone a 1
    elsif (int_clr = '1') then
      INTERRUP_ARM <= '0';      --Hemos escrito en ISR
      slv_reg3(0) <= '0';
    end if;
  end if;
end process;
    -- User logic ends
```

Figura 22. Comunicación con ENCcore mediante bus AXI

El microprocesador se comunica con el periférico ENCcore a través de cuatro registros del bus AXI, dos de escritura (*slv_reg0* y *slv_reg2*), uno de lectura (*slv_reg1*) y uno de lectura/escritura (*slv_reg3*).

Se ha determinado que *slv_reg0* corresponde al registro *obs_period*, *slv_reg1* al registro *enc_cnt*, *slv_reg2* al registro *IER* y *slv_reg3* al registro *ISR*.

El microprocesador debe escribir en el registro *slv_reg0* el valor deseado de *obs_period* y habilitar *IER* en el registro *slv_reg2*. Además, debe atender la interrupción desencadenada por la señal *update* para borrar el flag, es decir, dar al registro *ISR* valor cero y leer el valor de *enc_cnt* del registro *slv_reg1*.

Para acabar, el archivo que contiene todo el desarrollo HW del periférico es *EncCore_v1_0.vhd*, de manera que el periférico resulta así:

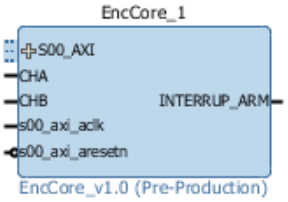


Figura 23. Bloque ENCcore

Nota: todos los archivos fuente utilizados se incluyen en la documentación del proyecto.

2.1.2.2 Simulación y validación de ENCcore

Para simular el comportamiento del periférico creado y validar su funcionamiento se ejecuta el testbench *ENCODERCore_tb* con la herramienta Model SIM.

En el testbench se realizan las siguientes acciones:

- Generar una señal *mclk* de 10 ns de periodo (de 100 MHz)
- Reset al inicio de la simulación
- Producir un cambio en la dirección de giro en el instante 900 ns
- Dar valor 128 a *obs_period*

A continuación, se muestran y analizan una serie de capturas de pantalla que han sido tomadas durante el proceso de simulación con el objetivo de comprobar que el periférico creado funciona correctamente.

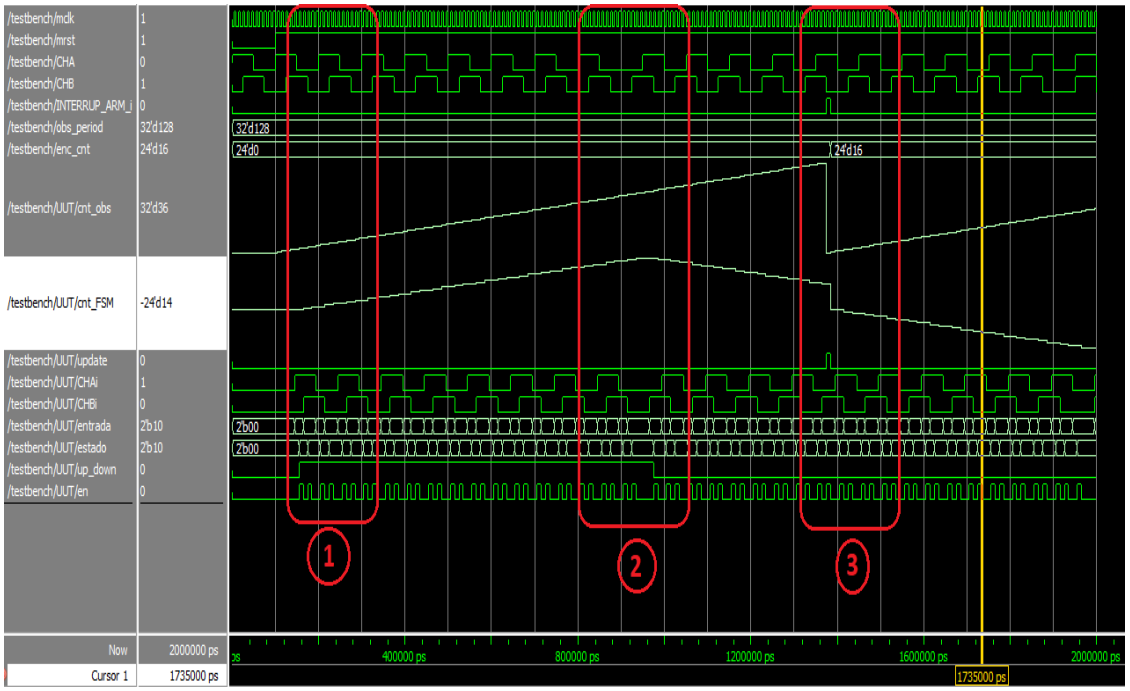


Figura 24. Resultado general de la simulación de ENCcore

Para analizar la simulación en detalle se muestran ampliadas las tres zonas señalizadas.

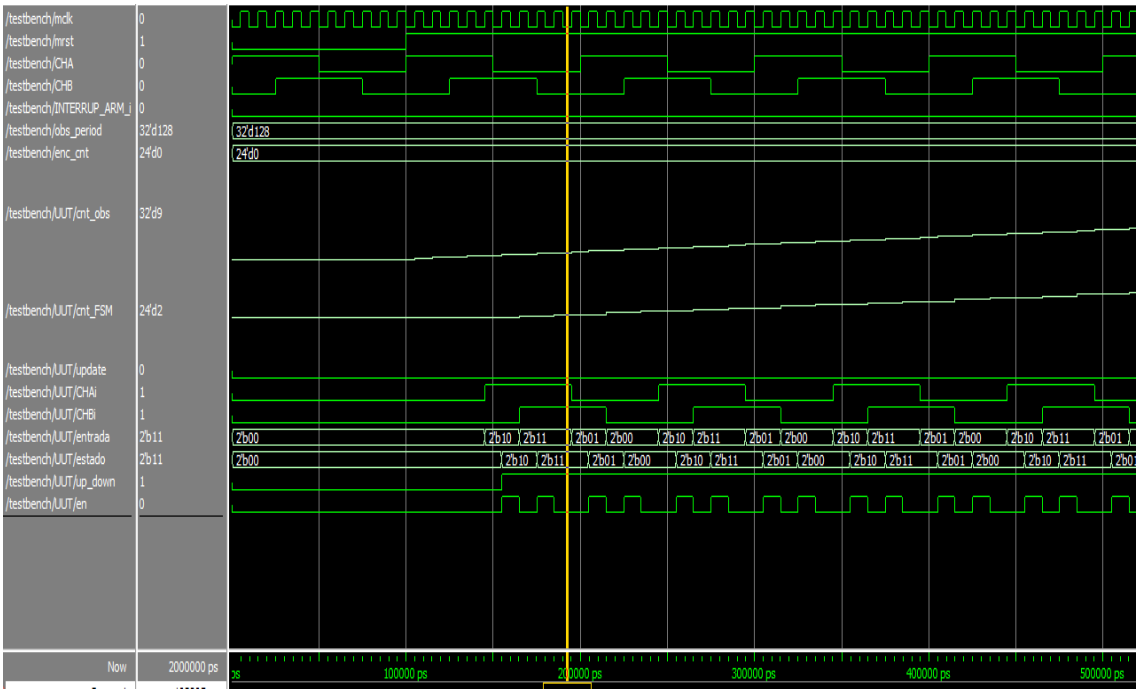


Figura 25. Funcionamiento FSM (zona 1)

Se puede observar que la FSM funciona correctamente: según el desfase de las señales *CHA_i* y *CHB_i*, el contador *cnt_FSM* debe incrementarse y la máquina de estados seguir la secuencia ascendente (00, 10, 11, 01, 00, 10, 11, 01…). Además,

cada vez que se produce una transición de estado se activa la señal *en* y *up_down* toma el valor 1, ya que el contador se debe incrementar.

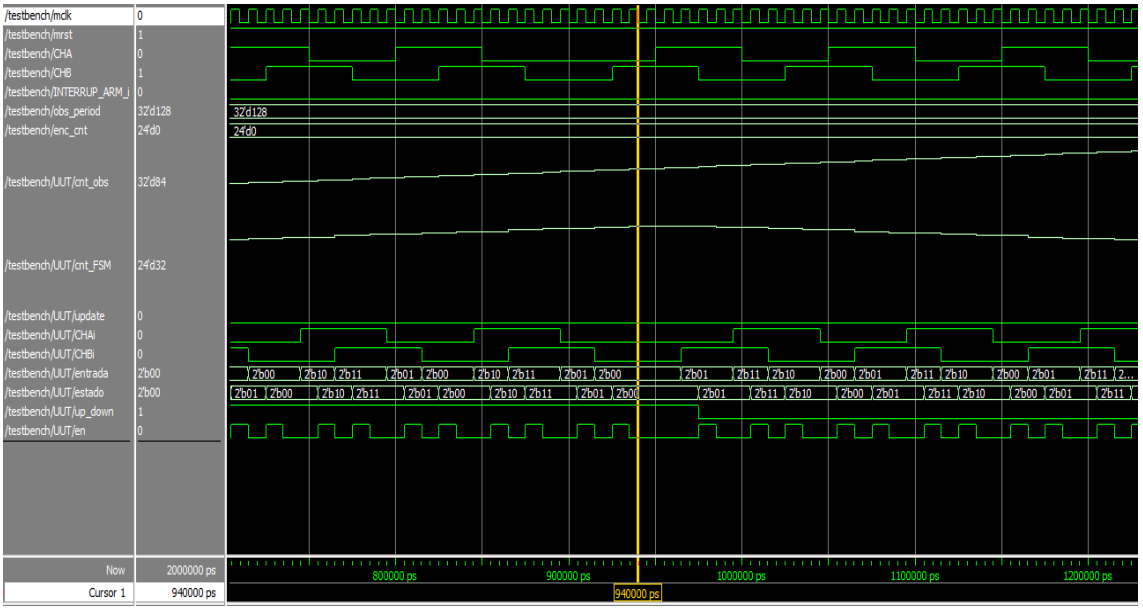


Figura 26. Cambio del sentido de giro (zona 2)

En instantes previos al cambio de sentido de giro (inversión del desfase entre *CHA_i* y *CHB_i*) la secuencia de estados era ascendente (00, 10, 11, 01, 00, 10, 11, 01…) pero tras el cambio pasa a ser descendente (00, 01, 11, 10, 00, 01, 11, 10…). Además, la señal *up_down* pasa a tomar el valor 0 indicando que el contador *cnt_FSM* debe decrementarse.

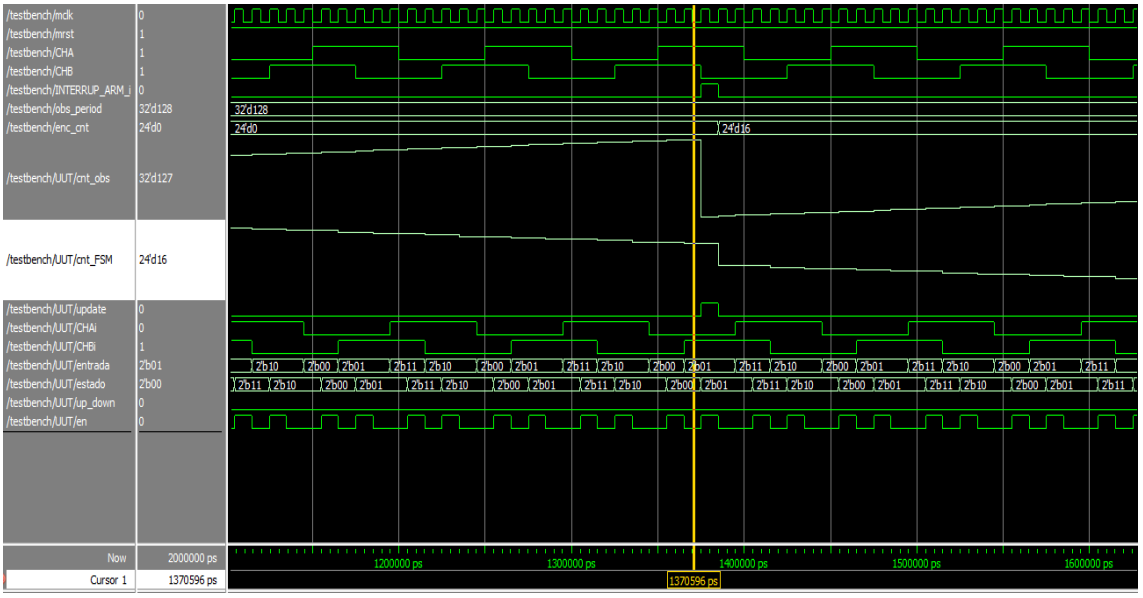


Figura 27. Fin de tiempo de observación

Nos encontramos en sentido descendente por lo que *up_down* vale 0 y el contador

cnt_FSM se decrementa. Al finalizar el tiempo de observación se activa la señal *update*, se registra en *enc_cnt* el valor de *cnt_FSM* que en ese momento vale 16 y se realiza un reset a *cnt_FSM*. A partir de este instante, *cnt_FSM* va tomando valores negativos (0x000000, 0xFFFFFF, 0xFFFFFE, 0xFFFFFD…, es decir, 0, -1, -2, -3…).

Por lo tanto, podemos afirmar que el periférico funciona correctamente y cumple con los requerimientos de diseño.

2.1.3 Desarrollo de la aplicación de usuario

Una vez finalizado todo el desarrollo hardware de nuestro SoC, se procede a crear con la herramienta SDK un proyecto en lenguaje C para que el microprocesador configure los periféricos y se comunique con ellos a través del bus AXI.

El código fuente desarrollado se incluye en los documentos del proyecto con el siguiente nombre: *P_FINAL.c*.

El código se encuentra completamente comentado, pero a continuación, se explican los apartados más relevantes con mayor detalle.

```
25  //REGISTROS DEL PERIFÉRICO PWMcore
26  #define XPAR_AXI_PWMcore_BASEADDR          0x43c00000
27  #define PWMCORE_IP_S00_AXI_SLV_REG0_OFFSET 0    //direction (1 bit)
28  #define PWMCORE_IP_S00_AXI_SLV_REG1_OFFSET 4    //carrier_period (32 bits)
29  #define PWMCORE_IP_S00_AXI_SLV_REG2_OFFSET 8    //mod_value (32 bits)
30  #define PWMCORE_IP_S00_AXI_SLV_REG3_OFFSET 12   //deadtime (16 bits)
31
32  //PARÁMETROS DE CONFIGURACIÓN DEL PERIFÉRICO PWMcore
33  #define CARRIER_PERIOD 0x0000c350    //50000 Tclks para obtener 2KHz en PWM
34  #define DEADTIME       0x0000ffff    //0.65ms
```

Figura 28. Definición de parámetros de PWMcore

Se define la dirección base donde se encuentran los registros del periférico PWMcore en el mapa de memoria del microprocesador, los 4 registros asociados al periférico, el valor 50000 en CARRIER_PERIOD necesario para conseguir generar una señal PWM de 2 KHz y un valor de DEADTIME de aproximadamente 0.65 ms.

```
36  //REGISTROS DEL PERIFÉRICO ENCcore
37  #define XPAR_AXI_ENCcore_BASEADDR       0x43c10000
38  #define ENCCORE_S00_AXI_SLV_REG0_OFFSET 0    //obs_period(32 bits)
39  #define ENCCORE_S00_AXI_SLV_REG1_OFFSET 4    //enc_cnt(24 bits)
40  #define ENCCORE_S00_AXI_SLV_REG2_OFFSET 8    //IER(1 bit)
41  #define ENCCORE_S00_AXI_SLV_REG3_OFFSET 12   //ISR(1 bit)
42
43  //PARÁMETROS DE CONFIGURACIÓN DEL PERIFÉRICO ENCcore
44  #define OBS_PERIOD 2.5e6    //1e6 Tclks para obtener un obs_period de 10ms
45  #define IER        0x00000001    //slv_reg2(0) habilitado
```

Figura 29. Definición de parámetros de ENCcore

Se define la dirección base donde se encuentran los registros del periférico ENCcore en el mapa de memoria del microprocesador, los 4 registros asociados al periférico, el valor de OBS_PERIOD necesario para determinar el periodo de interrupción (en este caso está configurado a 25 ms) y el valor de habilitación de IER.

A continuación, se analiza el contenido de las funciones de configuración de los periféricos y de la interrupción periódica del sistema.

```
 95   void Config_GPIO(void){
 96       //SW0...SW7 SON ENTRADAS
 97       XGpio_WriteReg(XPAR_AXI_GPIO_0_BASEADDR, XGPIO_TRI_OFFSET, 0xFF);
 98       //LD0...LD7 SON SALIDAS
 99       XGpio_WriteReg(XPAR_AXI_GPIO_1_BASEADDR, XGPIO_TRI_OFFSET, 0x00);
100       printf("SWITCHES y LEDS configurados\r\n");
101   }
```

Figura 30. Función de configuración de los GPIOs

Se determina que los puertos del periférico GPIO_0 operan como entradas para leer los microinterruptores SW7…SW0 mientras que los puertos del periférico GPIO_1 operan como salidas para activar/desactivar los LEDs LD7…LD0.

```
103   void Config_PWMcore(void){
104       //PERIODO DE LA SEÑAL PWM
105       PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
          PWMCORE_IP_S00_AXI_SLV_REG1_OFFSET, CARRIER_PERIOD);
106       //TIEMPOS MUERTOS AL CONMUTAR DIRECCIÓN DE GIRO
107       PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
          PWMCORE_IP_S00_AXI_SLV_REG3_OFFSET, DEADTIME);
108       printf("PWMcore configurado\r\n");
109   }
```

Figura 31. Función de configuración de PWMcore

Se escriben los parámetros definidos en la *Figura 28* en los registros de configuración para determinar la frecuencia de la señal PWM y el valor de sus tiempos muertos: el valor CARRIER_PERIOD se escribe en el registro *slv_reg1* que se corresponde con el registro *carrier_period*[31:0] y el valor DEADTIME se escribe en el registro *slv_reg3* que se corresponde con el registro *deadtime*[15:0].

```
111   void Config_ENCcore(void){
112       //TIEMPO DE CUENTA
113       ENCCORE_mWriteReg(XPAR_AXI_ENCcore_BASEADDR, ENCCORE_S00_AXI_SLV_REG0_OFFSET,
          OBS_PERIOD);
114       printf("ENCcore configurado\r\n");
115   }
```

Figura 32. Función de configuración de ENCcore

Se escribe el parámetro OBS_PERIOD definido en la *Figura 29* en el registro de configuración *slv_reg0* para determinar el tiempo de observación en 25 ms, este registro se corresponde con el registro *obs_period*[31:0].

```
125  int Config_Interrupt(void){
126      Xil_ExceptionDisable();  //Deshabilitación de interrupciones
127      SCUGIC_cfg= XScuGic_LookupConfig (XPAR_SCUGIC_0_DEVICE_ID);
128      if (SCUGIC_cfg == NULL)
129      {
130          xil_printf ("Error en la configuracion de Interrupciones\r\n");
131          return -1;
132      }
133
134      status= XScuGic_CfgInitialize(&SCUGIC_obj, SCUGIC_cfg,
         SCUGIC_cfg->CpuBaseAddress);
135      if (status != XST_SUCCESS)
136      {
137          xil_printf ("Error en la configuracion de Interrupciones\r\n");
138          return -1;
139      }
140
141      Xil_ExceptionInit();

142      Xil_ExceptionRegisterHandler (XIL_EXCEPTION_ID_IRQ_INT, (Xil_ExceptionHandler)
         XScuGic_InterruptHandler, &SCUGIC_obj);
143
144      //Habilitación de interrupción de FPGA Fabric (Encoder)
145      status= XScuGic_Connect (&SCUGIC_obj, XPS_FPGA0_INT_ID, (Xil_ExceptionHandler)
         ENCcore_ISR, NULL);
146      if (status != XST_SUCCESS)
147      {
148          xil_printf ("Error en la configuracion de Interrupción ENCcore\r\n");
149          return -1;
150      }
151
152      XScuGic_Enable(&SCUGIC_obj, XPS_FPGA0_INT_ID);
153
154      //Habilitación de interrupciones
155      Xil_ExceptionEnable();
156      xil_printf ("Interrupciones configuradas\r\n");
157  }
```

Figura 33. Función de configuración de la interrupción [Taylor, 2014]

Se configura el GIC (Generic Interrupt Controller) de tal manera que la fuente de interrupción es la señal periódica *INTERRUP_ARM* la cual se activa a nivel alto cada vez que concluye el tiempo de cuenta determinado por el parámetro OBS_PERIOD. Como esta señal proviene del apartado PL del SoC, se debe conectar al GIC a través del puerto IRQ_F2P[15:0] perteneciente al grupo de interrupciones SPI [Crocket, 2014].

```
117  void ENCcore_ISR (void){
118      //Flag de interrupción a nivel SW
119      ENCcore_flag = 1;
120
121      //Flag de interrupción a nivel HW sobre ENCcore
122      ENCCORE_mWriteReg(XPAR_AXI_ENCcore_BASEADDR, ENCCORE_S00_AXI_SLV_REG2_OFFSET,
         0x00000001);
123  }
```

Figura 34. Función de atención a la interrupción

En esta función simplemente se activan dos flags: uno a nivel SW para que el microprocesador pueda calcular la velocidad en el bucle infinito *while(1)* y otro flag a nivel HW para que el periférico ENCcore desactive la señal *INTERRUP_ARM* y dejarla así lista para la próxima conmutación a nivel alto.

Finalmente, se procede a analizar el bucle infinito de la función *main()* donde constantemente se observa el valor de los microinterruptores para actualizar cualquier cambio en las entradas del sistema y el flag de activación de la interrupción para proceder a calcular la velocidad.

```
258         //Lectura de los 8switches
259         value = XGpio_ReadReg(XPAR_AXI_GPIO_0_BASEADDR, XGPIO_DATA_OFFSET);
260
261         if (value_anterior |= value)  //Se ha modificado algún switch
262         {
263             value_anterior = value;
264
265             //Escritura en 8leds
266             XGpio_WriteReg(XPAR_AXI_GPIO_1_BASEADDR, XGPIO_DATA_OFFSET, value);
267
268             //Escritura en registro direction (bit 0 de value)
269             PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
                PWMCORE_IP_S00_AXI_SLV_REG0_OFFSET, value);
270         }
```

Figura 35. Actualización de entradas al sistema

Se lee continuamente el valor de los microinterruptores SW7…SW0 y se almacena en su variable asociada *value*. En el momento en que se modifica alguno de los 8 microinterruptores la condición se vuelve TRUE por lo que se ejecutan las instrucciones de la función *if()* de manera que: se copia el nuevo valor de *value* en los leds para actualizar su representación visual y se escribe en el registro *direction* [0] el valor actualizado, es decir, actualizamos el sentido de giro.

```
323                 //Escritura en registro mod_value (bits 7-2 de value)
324                 mod_value = (value >> 2) * 793;  //50000/63=793.65
325
326                 //Actuación sobre el motor DC
327                 PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
                    PWMCORE_IP_S00_AXI_SLV_REG2_OFFSET, mod_value);
```

Figura 36. Actualización de la señal PWM

De igual manera, se actualiza la señal *PWM* para determinar el nuevo valor de tensión que recibe el motor DC en sus terminales de alimentación.

Nota: Como el sistema se encuentra operando en lazo abierto implica que el microinterruptor SW1 y el LED LD1 están desactivados, es decir, *value*[1] es 0.

Para finalizar el control en lazo abierto, se analiza el momento en que el microprocesador calcula la velocidad de giro del motor:

```
330             if (ENCcore_flag & (1<<0))    //slv_reg3(0) = 1, se ha llegado al fin de
                obs_period
331             {
332                 ENCcore_flag = 0;  //Se borra el falg a nivel SW
333                 //Lectura del registro enc_cnt (24 bits)
334                 cuentas = ENCCORE_mReadReg(XPAR_AXI_ENCcore_BASEADDR,
                    ENCCORE_S00_AXI_SLV_REG1_OFFSET);
335                 cuentas = cuentas & 0x00FFFFFF;
336
337                 if ((cuentas & (1<<23)) == 0)  //Número positivo, giro en sentido
                    antihorario '0', cnt_FSM se incrementa
338                 {
339                     //rpm float de la reductora
340                     rpm_f = ((cuentas*(60*(CLK/OBS_PERIOD)))/(4*19))/3;  //rpm float de
                        la reductora
341                 }
342                 else   //Número negativo, giro en sentido horario '1', cnt_FSM se
                    decrementa
343                 {
344                     //rpm float de la reductora
345                     rpm_f = ((((0x00FFFFFF - cuentas) +
                        1)*(60*(CLK/OBS_PERIOD)))/(4*19))/3;  //rpm float de la reductora
346                 }
347
348                 //Mostramos la velocidad del motor en el monitor del PC
349                 printf("%3.2f\r\n", rpm_f);
350             }
```

Figura 37. Cálculo y envío de la velocidad al monitor del PC

En el momento en que se finaliza el tiempo de observación, el microprocesador es interrumpido y el flag SW es activado en la función de atención a la interrupción. La condición se vuelve TRUE ejecutándose las instrucciones de la función *if().* Lo que se hace es borrar el flag de la interrupción para indicar que ya está siendo atendida, se asigna a la variable *cuentas* el valor del contador *cnt_FSM* almacenado en el registro *enc_cnt*[23:0], quedándose solamente con los 24 bits de menor peso mediante una operación &. A continuación, se detecta si el número almacenado en la variable *cuentas* es positivo o negativo para su posterior uso. Si *cuentas*[23] es 1 quiere decir que el número es negativo, por ejemplo, el 0x00FFFFFD, que se corresponde con el -3, sino el número será positivo. Una vez calculada la velocidad en la variable float *rpm_f* se realiza un *printf()* para mostrar su valor en la pantalla del PC empleando para ello la UART1.

Llegados a este punto del desarrollo del proyecto el sistema se encuentra listo para analizar su funcionamiento dinámico en lazo abierto y poder validar el trabajo realizado hasta el momento. Aunque ya se hizo referencia a ello en el *CAPÍTULO 1: INTRODUCCIÓN*, concretamente en la *Figura 4*, se volverá a analizar con mayor detalle en el *CAPÍTULO 4: PRUEBAS EXPERIMENTALES Y CONCLUSIONES*, por lo que se retomará este apartado más adelante.

2.2 Control de velocidad en lazo cerrado

Se toma como punto de partida el trabajo realizado en el control en lazo abierto detallado en los puntos anteriores, por lo que ya se dispone de la mayor parte del material necesario para implementar el control en lazo cerrado.

Lo único que se debe añadir al sistema actual es el algoritmo de control PI que modela el comportamiento del diagrama de bloques de la *Figura 3* y quedará listo para este nuevo modo de funcionamiento.

Nota: se recuerda que para que el sistema opere en lazo cerrado el microinterruptor SW1 debe estar activado, es decir, el LED LD1 estará encendido y el valor de *value*[1] será 1.

2.2.1 Algoritmo de control PI

Para implementar el esquema típico de un sistema de control en lazo cerrado (ver *Figura 3*) es necesario emplear un regulador que controle el funcionamiento de nuestra planta a partir de un valor de referencia. En el campo analógico, estos reguladores se implementan mediante elementos discretos como, por ejemplo, amplificadores operacionales, resistores y condensadores, pero en el mundo digital se trata de un algoritmo encargado de calcular una consigna para cada valor muestreado.

El punto de partida para diseñar un regulador que controle correctamente el sistema de acuerdo a una serie de necesidades es disponer de un modelo matemático de la planta. A continuación, se muestra la función de transferencia típica de un motor DC, donde las constantes de K y τ tienen un valor determinado para el motor empleado en este proyecto:

$$\frac{K}{\tau s + 1}$$

Este modelo refleja su modo de funcionamiento y se obtiene mediante pruebas experimentales que consisten en someter a la planta a determinadas señales de entrada y analizar las salidas que se producen, con el objetivo de determinar el valor de las dos constantes del modelo. Este trabajo no es inmediato, por lo que vamos a optar por una solución menos precisa pero más rápida, ya que el objetivo

no es obtener un regulador óptimo sino conseguir que el sistema opere en lazo cerrado sin ningún requerimiento específico, es decir, no se busca una respuesta extremadamente rápida ni un error en régimen permanente que tienda a cero, simplemente que el sistema funcione correctamente sin necesidad de rapidez ni precisión.

Como regulador se utiliza el algoritmo de control diseñado cuidadosamente en un proyecto anterior, pero ligeramente modificado para adaptarse a esta nueva planta. Consiste en un algoritmo de control PI cuyo objetivo es hacer girar al motor a la velocidad que se indica en la referencia en un tiempo finito.

```
296                 //ALGORITMO DE CONTROL DE VELOCIDAD
297                 error = referencia - rpm_f;
298                 consigna = consigna_NM1 + K*error - CERO*K*error_NM1;
299
300                 if (consigna > 500)
301                 consigna = 500;
302                 if (consigna < 0)
303                 consigna = 0;
304
305                 error_NM1 = error;
306                 consigna_NM1 = consigna;
```

Figura 38. Algoritmo de control PI

El modo de operación de un regulador en el dominio discreto consiste en muestrear la variable a controlar a una determinada frecuencia y calcular un valor de consigna para cada una de las muestras. A continuación, se explica cómo se consigue esto en nuestro sistema: para empezar, la interrupción periódica es la encargada de determinar la frecuencia a la que se realiza el muestreo, en este caso está configurada para conseguir un periodo de muestreo de 25 ms. Cada vez que trascurre este tiempo se muestrea la variable bajo control, es decir, se calcula la velocidad del motor. Seguidamente, se determina el valor del error y se calcula la consigna correspondiente teniendo en cuenta el valor que tuvieron el error y la consigna en la muestra anterior. Finalmente, se registra el valor actual del error y de la consigna para utilizarlo en la próxima muestra (ver *Figura 38*).

Llegados a este punto, el sistema se encuentra listo para poder operar en lazo cerrado. A continuación, se muestra de forma breve el resultado obtenido, ya que consiste en la misma prueba que se realiza en lazo abierto y que se explica en detalle en el *CAPÍTULO 4: PRUEBAS EXPERIMENTALES Y CONCLUSIONES.*

PuTTY (inactive)

```
Comienza la ejecucion...
SWITCHES y LEDS configurados
PWMcore configurado
ENCcore configurado
Interrupciones configuradas
Memoria SD y documento de texto configurados
357    10.53    346.47    7.85
357    0.00    357.00    9.67
357    10.53    346.47    11.05
357    242.11    114.89    7.38
357    410.53    -53.53    4.10
357    421.05    -64.05    3.61
357    389.47    -32.47    4.04
357    368.42    -11.42    4.36
357    357.89    -0.89    4.54
357    336.84    20.16    4.99
357    347.37    9.63    4.85
357    326.32    30.68    5.38
357    336.84    20.16    5.29
357    347.37    9.63    5.15
357    347.37    9.63    5.20
357    357.89    -0.89    5.02
357    347.37    9.63    5.24
357    347.37    9.63    5.29
357    357.89    -0.89    5.11
357    357.89    -0.89    5.09
357    357.89    -0.89    5.07
```

Figura 39. Resultados del algoritmo de control PI

En cada una de las cuatro columnas se encuentran: la referencia, la velocidad, el error y la consigna, respectivamente. Según transcurre el tiempo de ejecución, el motor acelera hasta alcanzar la velocidad de referencia intentando que el error sea nulo, por lo que podemos confirmar que el algoritmo de control es efectivo y se puede continuar con la siguiente etapa del proyecto, es decir, la escritura de los datos en la memoria SD.

CAPÍTULO 3: REGISTRO DE DATOS EN LA MEMORIA SD

En este capítulo se explica al detalle cómo se consigue finalmente escribir datos en una memoria SD insertada en la tarjeta ZedBoard. Para ello, previamente se analiza el funcionamiento de este tipo de memorias, el modo de operación del controlador SD presente en la arquitectura Zynq y el conjunto de funciones que forman el driver de este dispositivo, sin olvidar además la librería FatFs que aporta al usuario funciones de alto nivel que facilitan el manejo del controlador SD. Para acabar, se comentan los cambios que se deben realizar sobre el sistema actual tanto a nivel HW como a nivel SW con el objetivo de añadir esta nueva funcionalidad y que el sistema quede operativo para poder trabajar en lazo cerrado y realizar el almacenamiento de las variables de control en la memoria SD.

3.1 Memoria SD Card

Se trata de un dispositivo basado en memorias flash ampliamente utilizado en sistemas embebidos para realizar un almacenamiento no volátil. En el caso de la tarjeta ZedBoard puede ser utilizada para dos propósitos: inicialización del sistema y almacenamiento de datos. Por una parte, puede servir para configurar el subsistema PL mediante el Bitstream, implementando así todos los módulos HW necesarios en el diseño e inicializar el subsistema PS, dejando el sistema listo para entrar en modo de ejecución. Por otra parte, puede ser utilizada como lugar de almacenamiento para realizar un registro de datos, esta es la funcionalidad que se pretende conseguir en este capítulo.

La necesidad de optimizar el espacio en los dispositivos portátiles y el desarrollo de la electrónica ha dado lugar a que estas memorias se comercialicen en 3 diferentes formatos con tamaños cada vez más reducidos: SD (empleado en la tarjeta ZedBoard), miniSD y microSD (ambos formatos utilizados en dispositivos de pequeño tamaño como smartphones y cámaras digitales).

Todas las memorias SD poseen un controlador en el interior de su encapsulado encargado, por ejemplo, de interactuar con las celdas de memoria a bajo nivel o de la corrección de errores [Henao, 2010]. A continuación, se muestra el diagrama de bloques de una memoria SD:

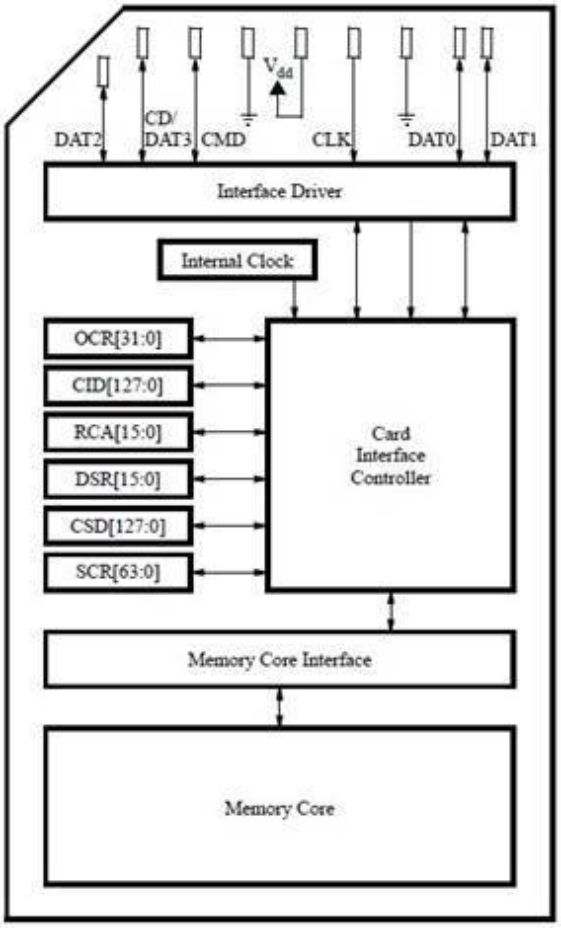


Figura 40. Diagrama de bloques de una memoria SD [Ababei, 2013]

En la figura anterior se pueden destacar las señales involucradas en la comunicación con la memoria SD y el controlador interno, elementos que se analizarán en detalle en los próximos apartados.

Para que el sistema pueda enviar datos a la memoria SD es necesario manejar el controlador situado en el subsistema PS (denominado SD host controller o controlador SD/SDIO). De esta manera, el controlador SD/SDIO interactúa con el controlador interno de la memoria SD empleando un protocolo de comunicación conocido como Especificación SDIO 2.0, donde quedan definidas las señales de la interfaz SDIO entre la memoria SD y el conector SD. El Protocolo SDIO está definido por la Asociación de memorias SD Card [SD Card Association, 2018].

NAME	DESCRIPTION
WP	Write protect pin. Pin low when write is enabled
CD	Card detect pin. Pin low when card detected

Figura 41. Señales exclusivas del conector SD

Hay dos señales que son exclusivas del conector SD y que por tanto no aparecen en la memoria SD. Se trata de las señales WP (informa de que la memoria SD se encuentra habilitada para escribir sobre ella) y CD (avisa de que hay una memoria SD insertada en el conector).

A continuación, se muestran las señales de la Especificación SDIO 2.0 comunes entre la memoria SD y el conector SD:

NR	SDIO INTERFACE	
	NAME	DESCRIPTION
1	CD/DAT3	Connector data line 3
2	CMD	Command/Response line
3	VSS1	GND
4	VDD	3.3V Power supply
5	CLK	Clock
6	VSS2	GND
7	DAT0	Connector data line 0
8	DAT1	Connector data line 1
9	DAT2	Connector data line 2

Figura 42. Señales comunes entre memoria SD y conector SD

A modo de resumen, la Especificación SDIO 2.0 trata con las siguientes señales:

- Cuatro señales para la transferencia de datos (DAT0-DAT3).
- Una señal para el reloj de sincronización (CLK).
- Una señal para la línea de comando (CMD).
- Una señal para detectar que existe una memoria SD insertada en el conector (CD).
- Una señal para indicar que la escritura en memoria está habilitada/bloqueada (WP).
- Una señal de alimentación (VDD).
- Dos señales de tierra (GND).

Las señales de datos (DAT0-DAT3) y las señales de control (CD, WP, CMD y CLK) se conectan al subsistema PS del dispositivo Zynq en los pines MIO40-MIO47 (Multiplexed Input Output) como se puede apreciar en la *Figura 43.*

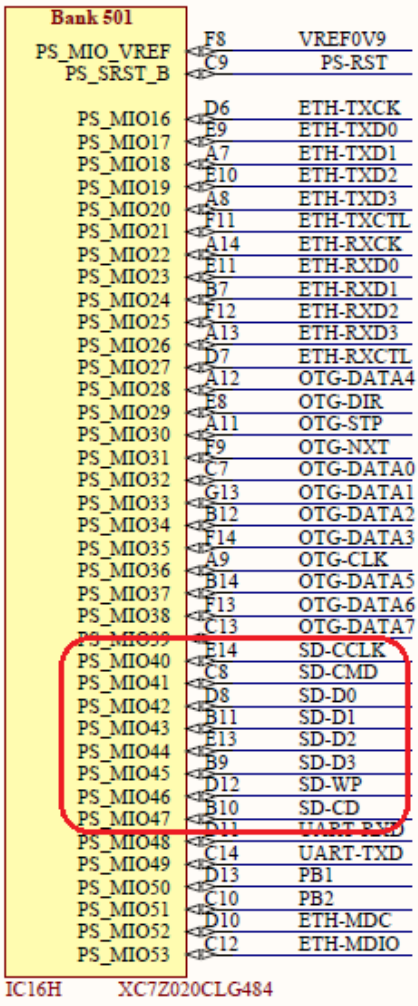


Figura 43. Conexión al PS de las señales de la memoria SD [ZedBoard, 2013]

Para poder realizar la conexión entre el PS y el conector de memoria SD (J12), el circuito integrado IC8 debe adaptar los niveles de tensión de las señales digitales, ya que el PS trabaja con el estándar de tensión LVCMOS1.8 mientras que la memoria SD trabaja con el estándar LVCMOS3.3, es decir, para el PS el nivel alto de las señales digitales es representado con 1.8V y para la memoria con 3.3V.

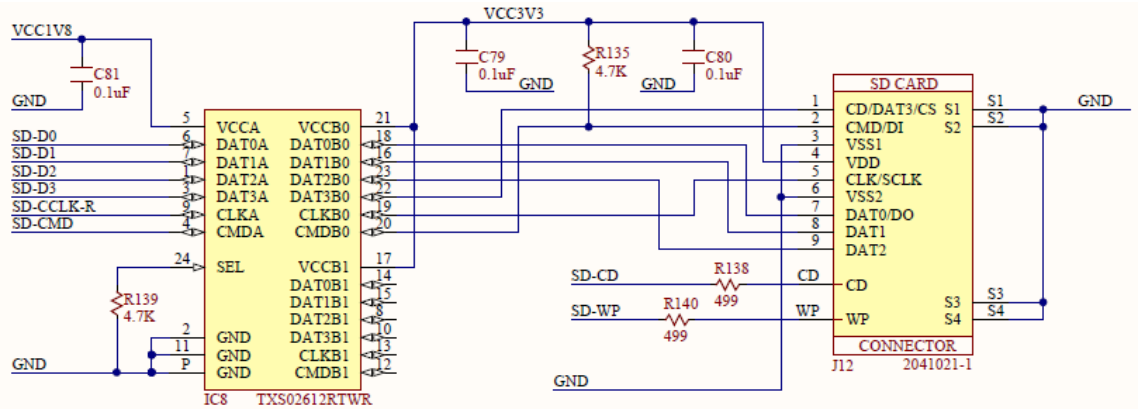


Figura 44. Adaptación de niveles de tensión entre PS y memoria SD [ZedBoard, 2013]

3.2 SD host controller

En los últimos tiempos las memorias SD se han convertido en un componente esencial en los sistemas embebidos, ya que son elementos de almacenamiento muy flexibles y tienen una capacidad razonable y adecuada para el uso en este tipo de sistemas. Esta es la razón por la cual el dispositivo Zynq incorpora un módulo que nos facilita el diseño a la hora de trabajar con una memoria SD, hablamos del SD host controller.

Para el desarrollo de este proyecto se ha utilizado el primero de los módulos, es decir, el SD0 host controller. Se debe habilitar de la misma forma que se habilitó UART1 y GPIO (ver *Figura 8*).

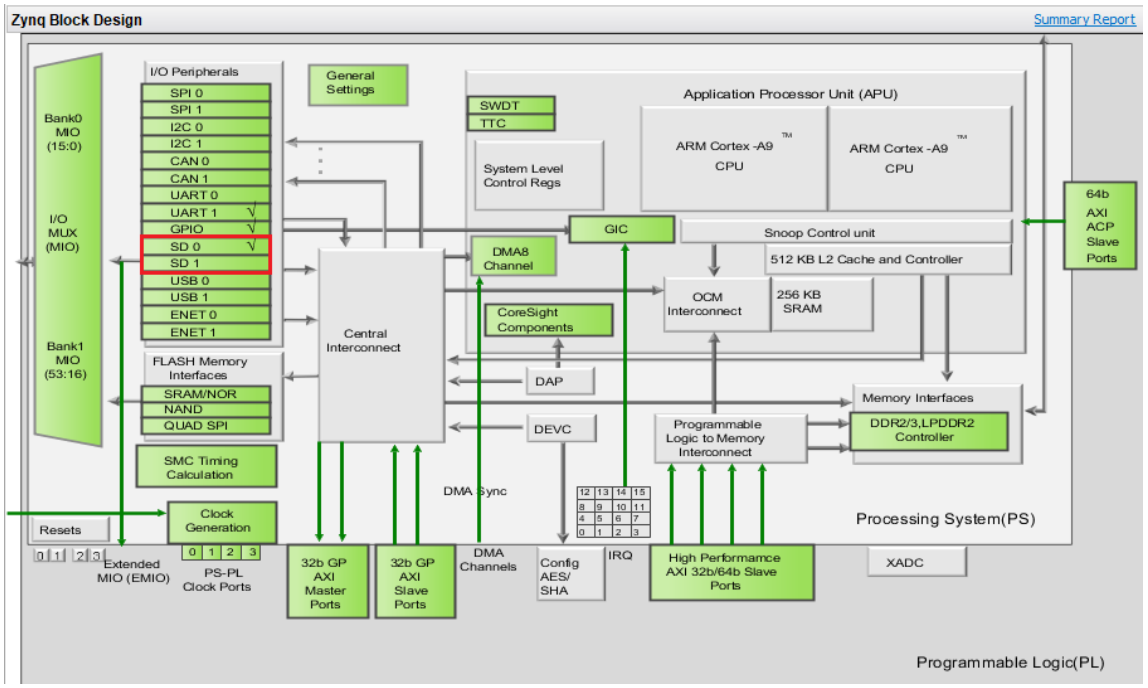


Figura 45. Diseño de bloques del dispositivo Zynq

El controlador SD0 es el encargado de realizar la comunicación con el controlador interno en la memoria SD empleando el protocolo SDIO comentado en el anterior apartado. Al tratarse de una memoria flash, la transferencia de datos entre los dos controladores se realiza en bloques de 512 bytes ya que la información es almacenada en la memoria SD en bloques de 512 bytes (valor por defecto pero que puede ser configurado en función de la necesidad del usuario). Por ejemplo, un archivo de un kilobyte ocupa dos bloques, pero un archivo de 200 bytes ocupa un bloque completo dejando los 312 bytes restantes inutilizados.

3.3 SD host controller driver (XSdPs driver)

Las funciones a bajo nivel del driver del controlador SD0 pueden ser utilizadas para configurar, inicializar, escribir y leer la memoria SD.

```
/************************** Function Prototypes ******************************/
XSdPs_Config *XSdPs_LookupConfig(u16 DeviceId);
s32 XSdPs_CfgInitialize(XSdPs *InstancePtr, XSdPs_Config *ConfigPtr,
              u32 EffectiveAddr);
s32 XSdPs_SdCardInitialize(XSdPs *InstancePtr);
s32 XSdPs_ReadPolled(XSdPs *InstancePtr, u32 Arg, u32 BlkCnt, u8 *Buff);
s32 XSdPs_WritePolled(XSdPs *InstancePtr, u32 Arg, u32 BlkCnt, const u8 *Buff);
s32 XSdPs_SetBlkSize(XSdPs *InstancePtr, u16 BlkSize);
s32 XSdPs_Select_Card (XSdPs *InstancePtr);
s32 XSdPs_Change_ClkFreq(XSdPs *InstancePtr, u32 SelFreq);
s32 XSdPs_Change_BusWidth(XSdPs *InstancePtr);
s32 XSdPs_Change_BusSpeed(XSdPs *InstancePtr);
s32 XSdPs_Get_BusWidth(XSdPs *InstancePtr, u8 *SCR);
s32 XSdPs_Get_BusSpeed(XSdPs *InstancePtr, u8 *ReadBuff);
s32 XSdPs_Pullup(XSdPs *InstancePtr);
s32 XSdPs_MmcCardInitialize(XSdPs *InstancePtr);
s32 XSdPs_CardInitialize(XSdPs *InstancePtr);
s32 XSdPs_Get_Mmc_ExtCsd(XSdPs *InstancePtr, u8 *ReadBuff);
s32 XSdPs_Set_Mmc_ExtCsd(XSdPs *InstancePtr, u32 Arg);
```

Figura 46. Funciones de bajo nivel del driver del controlador SD0

Estas funciones a bajo nivel pueden ser suficiente para el propósito de este capítulo, ya que solamente se va a tratar con un único archivo denominado *LOG.TXT*. Sin embargo, emplear estas funciones para tratar con múltiples archivos puede resultar muy laborioso, consumir mucho tiempo de trabajo y que aparezcan errores con gran facilidad. La razón es que se necesita registrar la dirección de inicio de todos los bloques de memoria de todos los archivos con los que se esté trabajando.

Para ilustrar este problema se plantea la siguiente situación: imaginemos que se está trabajando con los archivos *FILE1.TXT*, *FILE2.TXT* y *FILE3.TXT*, con un tamaño de 1536 bytes, 1536 bytes y 512 bytes, respectivamente. Como se puede ver en la *Figura 47*, para poder manejar estos tres archivos es necesario registrar la dirección de inicio de 7 bloques. Si el tamaño y el número de archivos aumenta, la situación puede llegar a ser insostenible. Por ejemplo, imaginemos ahora que se necesita trabajar con 10 archivos de 100Mbytes cada uno, ¿Cuántas direcciones de inicio sería necesario registrar? Además, la cosa se puede complicar todavía más ya que no necesariamente los bloques de un archivo se almacenan de manera consecutiva, como es el caso del archivo *FILE2.TXT*.

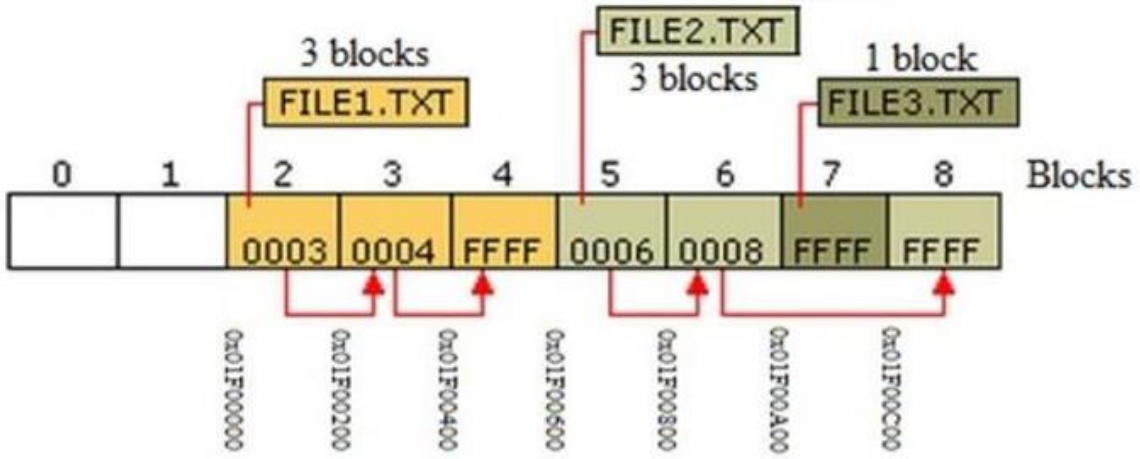


Figura 47. Registro de la dirección de inicio de los bloques de memoria de 3 archivos

La solución a este problema reside en utilizar las funciones de alto nivel que nos proporciona la librería FatFs y que nos facilita enormemente el trabajo.

3.4 Librería FatFs (File Allocation Table File System)

Esta librería proporciona funciones de alto nivel que son de fácil manejo para el usuario, es decir, forman un interfaz de programación denominada API (Application Programming Interface) que elimina en gran medida la dificultad de trabajar con la dirección de inicio de los bloques de los archivos que aparece al emplear las funciones de bajo nivel del driver, como se ha explicado al detalle en el apartado anterior.

A continuación, se muestra la lista de las funciones APIs de la librería FatFs:

```
/* FatFs module application interface*/
FRESULT f_open (FIL* fp, const TCHAR* path, BYTE mode);                /* Open or create a file */
FRESULT f_close (FIL* fp);                                             /* Close an open file object */
FRESULT f_read (FIL* fp, void* buff, UINT btr, UINT* br);              /* Read data from a file */
FRESULT f_write (FIL* fp, const void* buff, UINT btw, UINT* bw);       /* Write data to a file */
FRESULT f_lseek (FIL* fp, DWORD ofs);                                  /* Move file pointer of a file object */
FRESULT f_sync (FIL* fp);                                              /* Flush cached data of a writing file */
FRESULT f_opendir (DIR* dp, const TCHAR* path);                        /* Open a directory */
FRESULT f_closedir (DIR* dp);                                          /* Close an open directory */
FRESULT f_readdir (DIR* dp, FILINFO* fno);                             /* Read a directory item */
FRESULT f_stat (const TCHAR* path, FILINFO* fno);                      /* Get file status */
FRESULT f_truncate (FIL* fp);                                          /* Truncate file */
FRESULT f_unlink (const TCHAR* path);                                  /* Delete an existing file or directory */
FRESULT f_mkdir (const TCHAR* path);                                   /* Create a sub directory */
FRESULT f_rename (const TCHAR* path_old, const TCHAR* path_new);       /* Rename/Move a file or directory */
FRESULT f_chmod (const TCHAR* path, BYTE value, BYTE mask);            /* Change attribute of the file/dir */
FRESULT f_utime (const TCHAR* path, const FILINFO* fno);               /* Change times-tamp of the file/dir */
FRESULT f_mount (FATFS* fs, const TCHAR* path, BYTE opt);              /* Mount/Unmount a logical drive */
FRESULT f_mkfs (const TCHAR* path, BYTE sfd, UINT au);                 /* Create a file system on the volume */
```

Figura 48. Funciones APIs de alto nivel de la librería FatFs

La librería FatFs esconde la complejidad de trabajar directamente con las funciones del driver del controlador. Hay que resaltar que las funciones de esta librería llaman a las funciones del driver del controlador para implementar su funcionalidad. En este proyecto, la librería es empleada en un sistema standalone aunque también puede ser utilizada en proyectos que se rigen por un sistema operativo.

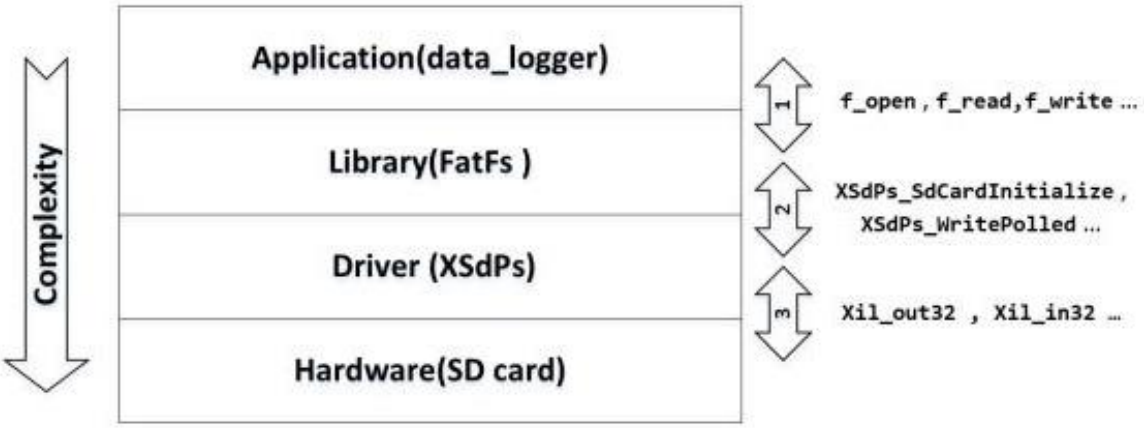


Figura 49. Capas SW entre el usuario y el HW

La figura anterior muestra como el procesador Zynq ARM se comunica con la memoria SD realizando una interacción entre la aplicación de usuario, la librería, el driver y el hardware de la memoria SD. De esta manera, la dificultad de tratar directamente con el hardware o el driver ha sido simplificada con la librería FatFs, es decir, se consigue que la aplicación de usuario interactúe de forma eficiente con el hardware de la memoria SD. Las APIs se traducen en llamadas a funciones del driver XSdPs, que se encargan de escribir en la región del mapa de memoria dedicada al SD host controller para que éste se comunique con el hardware.

3.5 APIs de la librería FatFs

En este apartado se habla en detalle de las APIs empleadas en el proyecto para gobernar el controlador SD0 y escribir sobre la memoria SD las variables del control de velocidad PI del motor DC.

Antes de poder utilizar cualquiera de las APIs, es necesario declarar 3 variables imprescindibles para el correcto funcionamiento del sistema:

- En primer lugar, una variable tipo FATFS que estará asociada con la

memoria SD insertada en la tarjeta ZedBoard.

- De igual manera, se necesita declarar una variable tipo FIL para el archivo usado en este proyecto y sobre el que se van a escribir los datos.
- En tercer y último lugar, una variable FRESULT en la que se almacenará el valor retornado por las APIs. De esta forma, se podrá conocer si las APIs se ejecutan con éxito o si ha surgido algún error.

```
48  //VARIABLES ASOCIADAS A LA MEMORIA SD
49  //Variable tipo FATFS asociada a la memoria SD
50  FATFS FS_instance;
51
52  //Variable tipo FIL asociada al archivo de texto en la memoria
53  FIL file1;
54
55  //Variable tipo FRESULT para almacenar el valor que retornan las funciones API*/
56  FRESULT result;
```

Figura 50. Variables necesarias para el uso de las APIs

Merece la pena analizar en mayor detalle la variable FRESULT ya que no solamente puede indicar al usuario la aparición de un error en alguna de las APIs sino que además, puede determinar el error concreto en función del valor que retorna la API utilizada. La siguiente figura muestra la lista de los posibles errores:

Value	Code	Meaning
0	FR_OK	Operation succeeded.
1	FR_DISK_ERR	A hard error occurred in the low level disk I/O layer.
2	FR_INT_ERR	Assertion failed.
3	FR_NOT_READY	The physical drive cannot work.
4	FR_NO_FILE	Could not find the file.
5	FR_NO_PATH	Could not find the path.
6	FR_INVALID_NAME	The path name format is invalid.
7	FR_DENIED	Access denied due to prohibited access or directory full.
8	FR_EXIST	Access denied due to prohibited access.
9	FR_INVALID_OBJECT	The file/directory object is invalid.
10	FR_WRITE_PROTECTED	The physical drive is write protected.
11	FR_INVALID_DRIVE	The logical drive number is invalid.
12	FR_NOT_ENABLED	The volume has no work area.
13	FR_NO_FILESYSTEM	There is no valid FAT volume on the physical drive.
14	FR_MKFS_ABORTED	The f_mkfs() aborted due to any parameter error.
15	FR_TIMEOUT	Could not get a grant to access the volume within defined period.
16	FR_LOCKED	The operation is rejected according to the file. sharing policy
17	FR_NOT_ENOUGH_CORE	LFN working buffer could not be allocated
18	FR_TOO_MANY_OPEN_FILES	Number of open files > _FS_SHARE

Figura 51. Valores de retorno de las APIs [Chan, 2014]

La variable FRESULT puede tomar 19 valores diferentes de manera que cualquier valor distinto de 0 implica un error de ejecución en alguna de las APIs.

A continuación, se analizan dos ejemplos de errores reales que aparecen durante la ejecución del sistema para demostrar la gran utilidad que tiene el valor de retorno de las APIs en la variable FRESULT.

Ejemplo 1: se inicia la ejecución de la aplicación de usuario sin disponer de una memoria SD insertada en el conector. Al ejecutar la función *f_mount()* para asociar la variable FATFS con la memoria SD, aparece el error número 3 con código FR_NOT_READY, es decir, el dispositivo físico de la memoria no se encuentra operativo.

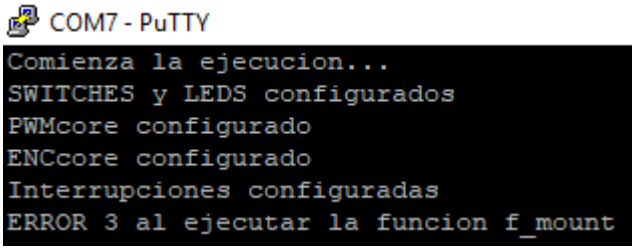


Figura 52. Error 3: memoria SD no insertada en el conector

Ejemplo 2: se inicia la ejecución de la aplicación de usuario con la memoria SD insertada en el conector, pero en este caso con el bloqueo de escritura activado. Al ejecutar la función *f_open()* con el objetivo de crear un archivo en la memoria aparece el error número 10 con código FR_WRITE_PROTECTED, es decir, la acción de escritura sobre la memoria se encuentra deshabilitada.

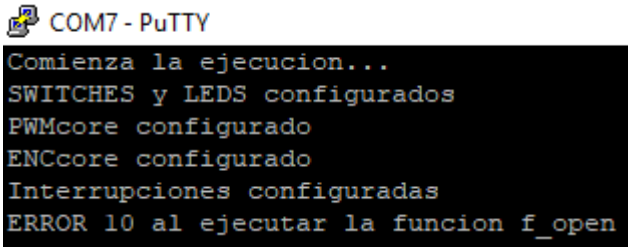


Figura 53. Error 10: memoria SD insertada con la escritura deshabilitada

Para finalizar este apartado, se procede a explicar con detalle las APIs de la librería FatFs que se han empleado en este proyecto para conseguir realizar el data logging y registrar así la información deseada en la memoria SD. En concreto han sido las siguientes funciones: f_mount(), f_open(), f_write(), f_lseek() y f_close().

La primera API que debe estar presente en el código es la función *f_mount()*.

```
FRESULT f_mount (
    FATFS* fs,          /* Pointer to the file system object (NULL:unmount)*/
    const TCHAR* path,  /* Logical drive number to be mounted/unmounted */
    BYTE opt            /* 0:Do not mount (delayed mount), 1:Mount immediately */
)
```

Figura 54. Formato de la función f_mount()

Esta función inicializa la memoria SD y la asocia con la variable de tipo FATFS. Para ello recibe tres argumentos: el puntero a la variable FATFS, el número de dispositivo de memoria (si solo hay una memoria SD en el sistema este valor debe ser *0:/*) y un parámetro de configuración.

Una vez que la función *f_mount()* se ha ejecutado con éxito, la memoria SD está inicializada y lista para ser utilizada. Los archivos en la memoria se manejan de igual manera que los archivos de cualquier computador, es decir, se deben abrir, realizar las acciones oportunas de escritura/lectura y finalmente, deben ser cerrados para guardar su contenido.

Dicho esto, la siguiente función que debe ser utilizada es *f_open()*, ya que no solamente permite abrir un archivo sino que también es la encargada de crearlos.

```
FRESULT f_open (
    FIL* fp,            /* Pointer to the blank file object */
    const TCHAR* path,  /* Pointer to the file name */
    BYTE mode           /* Access mode and file open mode flags */
)
```

Figura 55. Formato de la función f_open()

La función *f_open()* recibe tres argumentos: el puntero a la variable asociada al archivo, el nombre y ruta del archivo y el modo de apertura. Este modo de apertura determina si debe crearse un nuevo archivo o abrir uno ya existente, además de controlar la acción de sobreescritura.

Value	Code	Meaning
0x01	FA_READ	Specifies read access to the file. Data can be read from the file. Combine with FA_WRITE for read-write access. You can combine by using "|"
0x02	FA_WRITE	Specifies write access to the file. Data can be written to the file. Combine with FA_READ for read-write access. You can combine by using "|"
0x00	FA_OPEN_EXISTING	Opens the file. The function fails if the file is not existing.
0x10	FA_OPEN_ALWAYS	Opens the file if it is existing. If not, a new file is created.
0x04	FA_CREATE_NEW	Creates a new file. The function fails with FR_EXIST if the file is existing.
0x08	FA_CREATE_ALWAYS	Creates a new file. If the file is existing, it will be truncated and overwritten.

Figura 56. Modos de apertura de un archivo [ChaN, 2014]

Una vez creado y abierto el archivo, el siguiente paso lógico consiste en escribir sobre éste los datos que el usuario necesita, para ello nos ayudamos de la función *f_write()*.

```
FRESULT f_write (
    FIL* fp,            /* Pointer to the file object */
    const void *buff,   /* Pointer to the data to be written */
    UINT btw,           /* Number of bytes to write */
    UINT* bw            /* Pointer to number of bytes written */
)
```

Figura 57. Format de la función f_write()

La función *f_write()* recibe en este caso 4 argumentos: el puntero a la variable asociada al archivo, el puntero al buffer que contiene la cadena de los datos a escribir, el número de bytes de la cadena de datos y el puntero a la variable que almacena el número de bytes que forman el archivo.

El proceso de escritura es secuencial, es decir, cada cadena de datos se escribe a continuación de la cadena de datos anterior (de la misma forma que se escribe con bolígrafo sobre un papel). Para conseguir esto y que la escritura se realice correctamente se necesita emplear la función *f_lseek().*

```
FRESULT f_lseek (
    FIL* fp,        /* Pointer to the file object */
    DWORD ofs       /* File pointer from top of file */
)
```

Figura 58. Formato de la función f_lseek()

La función *f_lseek()* recibe solamente 2 argumentos: el puntero de la variable asociada al archivo y la cantidad de bytes que éste contiene, para de esta manera determinar la posición donde debe comenzar a escribirse la siguiente cadena de datos. Se puede decir que esta función "sitúa el cursor" al final del texto del archivo.

Tras realizar todas las tareas necesarias dentro del archivo, el último paso que se debe realizar es su cierre mediante la función *f_close().*

```
FRESULT f_close (
    FIL *fp     /* Pointer to the file object to be closed */
)
```

Figura 59. Formato de la función f_close()

Esta función recibe únicamente un argumento: el puntero a la variable asociada al archivo.

Es importante recordar la necesidad de analizar el valor que retorna una API tras su ejecución para de esta manera conocer si se ha producido un error. A continuación, se muestra uno de los muchos casos que aparecen en el código C:

```
226         // Cerrar archivo de texto
227         result = f_close(&file1);
228
229             if (result!=0) {
230                 printf("ERROR al ejecutar la función f_close\r\n");
231                 return XST_FAILURE;
232             }
```

Figura 60. Análisis del valor retornado por las APIs

Si una API retorna un valor diferente de 0 quiere decir que ha ocurrido algún error, por lo tanto, se debe informar al usuario mediante un mensaje de aviso, es la única manera de conocer el correcto funcionamiento de estas funciones.

3.6 Modificación de la aplicación de usuario

Para concluir con el capítulo, en este apartado se detallan los cambios realizados sobre el código C de la aplicación de usuario que aportan al sistema la opción de almacenar datos en la memoria SD y conseguir así realizar el data logging del sistema operando en lazo cerrado.

Como ya se comentó en el apartado anterior, el primer paso para poder trabajar con las funciones de la librería FatFs consiste en declarar tres variables, como se muestra en la *Figura 50.*

Una vez declaradas las variables FATFS *FS_instance*, FIL *file1* y FRESULT *result*, el siguiente paso será configurar la memoria SD y el archivo de texto.

```
160   void Config_fileSD(void){
161       // Inicialización de la memoria SD
162       result = f_mount(&FS_instance,Path, 1);
163
164           if (result != 0){
165               printf("ERROR %d al ejecutar la funcion f_mount\r\n", result);
166               return XST_FAILURE;
167           }
168
169       // Se crea un nuevo documento de texto con permisos de lectura/escritura
170       Log_File = (char *)FileName;
171
172       result = f_open(&file1, Log_File, FA_CREATE_ALWAYS | FA_WRITE | FA_READ);
173
174           if (result!= 0){
175               printf("ERROR %d al ejecutar la funcion f_open\r\n", result);
176               return XST_FAILURE;
177           }
178
179           printf("Memoria SD y documento de texto configurados\r\n");
180   }
```

Figura 61. Función de configuración de la memoria SD y del archivo

En esta función *Config_fileSD()*, se emplea la API *f_mount()* para asociar la memoria SD a la variable *FS_instance* y la API *f_open()* para crear el archivo de texto *LOG.TXT* donde se escribirán todos los datos a registrar.

Tras ejecutarse la función *Config_fileSD()*, la memoria SD y el archivo de texto se encuentran listos para recibir los datos. Se crea la función *WriteSD()* encargada de escribir los datos en cada periodo de interrupción.

```
182  void WriteSD(void){
183      // Apertura del documento de texto en modo escritura
184      result = f_open(&file1, Log_File,FA_WRITE);
185
186          if (result!=0) {
187              printf("ERROR al ejecutar la función f_open\r\n");
188              return XST_FAILURE;
189          }
190
191      // Situar nuevos datos a partir del último dato escrito
192      result = f_lseek(&file1,accum);
193
194          if (result!=0) {
195              printf("ERROR al ejecutar la función f_lseek\r\n");
196              return XST_FAILURE;
197          }
198
199      // Dar forma al paquete de informacion a guardar
200      if (cnt<1) // PRIMERA ESCRITURA EN MEMORIA SD
201      {
202          sprintf(Buffer_logger, "%d    %3d    %3.2f    %3.2f    %3.2f    %d/%d/%d
          %d:%d:%d\r\n", cnt, referencia, rpm_f, error, consigna_V, DAY, MONTH, YEAR,
          HOUR, MINUTE, SECOND);
203      }
204      else  // RESTO DE ESCRITURAS POSTERIORES
205      {
206          sprintf(Buffer_logger, "%d    %3d    %3.2f    %3.2f    %3.2f\r\n", cnt,
          referencia, rpm_f, error, consigna_V);
207      }
208
209      // CÁLCULO DEL TAMAÑO DEL PAQUETE DE DATOS
210      len = strlen(Buffer_logger);
211      Buffer_size=len;
212
213      cnt=cnt+1;  // número de renglón en el ARCHIVO DE TEXTO
214
215      // Escritura en el documento de texto
216      result = f_write(&file1, (const void*)Buffer_logger, Buffer_size,&BytesWr);
217
218          if (result!=0) {
219              printf("ERROR al ejecutar la función f_write\r\n");
220              return XST_FAILURE;
221          }
222
223      // Incremetar EOF
224      accum=accum+len;
225
226      // Cerrar archivo de texto
227      result = f_close(&file1);
228
229          if (result!=0) {
230              printf("ERROR al ejecutar la función f_close\r\n");
231              return XST_FAILURE;
232          }
233      }
```

Figura 62. Función de escritura WriteSD()

La función *WriteSD()* es llamada cada vez que el microprocesador es interrumpido, es decir, en cada periodo de interrupción, en este caso configurado en 25 milisegundos. Esta función realiza las siguientes tareas:

- Se llama a la función *f_open()* para abrir el archivo en modo escritura y que éste quede listo para recibir datos. Se comprueba el resultado que devuelve esta API para descartar un error en su ejecución.
- Se emplea la función *f_lseek()* para situar el siguiente dato a escribir a continuación del último dato escrito. Se vuelve a comprobar la variable *result* para descartar cualquier error de ejecución.
- Para enviar los datos a la memoria SD se crea una cadena de texto que contiene: el número de muestra, la referencia, la velocidad, el error y la consigna. La cadena de datos de la primera muestra lleva además información sobre la fecha, es decir, se le añade día, mes, año, hora, minuto y segundo. De esta manera, se puede conocer el momento en el que se inicia el registro de datos y utilizarla como referencia temporal para las demás muestras del registro.
- A continuación, se calcula la longitud de la cadena de datos, ya que es necesario conocer el número de bytes a escribir a la hora de utilizar la función *f_write()*.
- Una vez se tiene la cadena de datos formada y el número de bytes a almacenar, se utiliza la función *f_write()* para escribir en la memoria SD. Como es habitual, se comprueba si ha surgido algún error de ejecución.
- Tras haber completado la escritura con éxito se actualiza la variable *accum*, encargada de almacenar el número total de bytes escritos en el archivo, ya que es un valor necesario para la función *f_lseek()*.
- Para acabar con el proceso de escritura, se emplea la función *f_close()* encargada de cerrar el archivo de texto, sin olvidar comprobar el valor de retorno para asegurar que no hay error en su ejecución.

Llegados a este punto del proyecto, el sistema se encuentra completamente finalizado y listo para validar su funcionamiento en lazo cerrado, modo en el cuál se realiza el data logging de las variables asociadas al control PI de velocidad. Ya se hizo referencia a ello en el *CAPÍTULO 1: INTRODUCCIÓN*, pero se volverá a ver con más detalle en el siguiente capítulo dedicado a realizar pruebas de validación.

CAPÍTULO 4: PRUEBAS EXPERIMENTALES Y CONCLUSIONES

En este capítulo se procede a analizar el funcionamiento del sistema en sus dos modos de operación: modo de operación en lazo abierto y modo de operación en lazo cerrado implementando un control PI de velocidad.

4.1 Funcionamiento en lazo abierto

El objetivo de este primer apartado del capítulo es mostrar al detalle el procedimiento y los resultados de cada uno de los pasos realizados para validar el correcto funcionamiento del sistema en lazo abierto.

El funcionamiento general es sencillo: se realizan una serie de cambios en los microinterruptores SW7…SW2 para modificar el tiempo en alto de la señal *PWM* y como resultado se obtiene la velocidad del motor en la pantalla del PC.

Recordatorio: para que el sistema opere en lazo abierto el microinterruptor SW1 debe estar desactivado, es decir, debe tomar el valor cero o lo que es lo mismo, la variable *value*[1] tiene que ser 0.

Según la información reflejada en la hoja de características del motor, si éste se pone en funcionamiento sin carga en su eje y es alimentado a 12 V se consigue que la reductora gire a 789 rpm. Mediante una sencilla prueba experimental que consiste en alimentar el motor de forma directa a 12 V exactos, obtenemos una velocidad de 750 rpm, algo menor de lo indicado debido a las tolerancias presentes en el equipo.

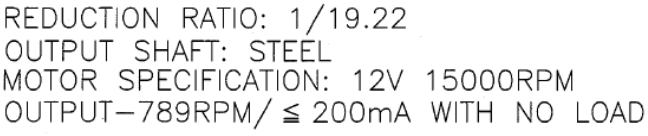


Figura 63. Información de la hoja de características del motor DC

De esta manera y para nuestro caso en particular, se puede determinar que el rango de tensión 0…12 V corresponde con el rango de velocidad 0…750 rpm. Para determinar el tiempo en alto de la señal PWM disponemos de 6 microinterruptores (SW7…SW2), es decir, de $2^6 = 64$ posibles valores de tensión, por lo que el rango de 0…12 V queda dividido en escalones de aproximadamente 0,1905 V. Para conocer con detalle la relación entre tensión aplicada y velocidad de giro se somete al motor a valores concretos a lo largo de su rango de funcionamiento con el objetivo de conocer la velocidad que le corresponde:

CÓDIGO DECIMAL	CÓDIGO BINARIO	TENSIÓN (V)	VELOCIDAD (RPM)
0	000000	0	0
1	000001	0,1905	0
3	000011	0,5715	0
5	000101	0,9525	0
7	000111	1,3335	0
9	001001	1,7145	0
11	001011	2,0955	0
13	001101	2,4765	0
15	001111	2,8575	76
17	010001	3,2385	150
19	010011	3,6195	221
21	010101	4,0005	284
23	010111	4,3815	336
25	011001	4,7625	378
27	011011	5,1435	421
29	011101	5,5245	453
31	011111	5,9055	494
33	100001	6,2865	536
35	100011	6,6675	557
37	100101	7,0485	578
39	100111	7,4295	600
41	101001	7,8105	621
43	101011	8,1915	631
45	101101	8,5725	652
47	101111	8,9535	663
49	110001	9,3345	673
51	110011	9,7155	684
53	110101	10,0965	694
55	110111	10,4775	700
57	111001	10,8585	705
59	111011	11,2395	715
61	111101	11,6205	726
63	111111	12	750

Tabla 1. Relación tensión-velocidad en motor DC

Para visualizar el conjunto de los datos y entender de forma global el comportamiento del motor en todo su rango de funcionamiento se procede a representar de forma gráfica los datos obtenidos con anterioridad. Para el eje de abscisas se emplean los datos de la columna TENSIÓN, es decir, el rango 0…12 V escalonado en intervalos de 0.1905 V, mientras que en el eje de coordenadas se representa la columna VELOCIDAD, datos de la velocidad obtenida para cada valor de entrada. La gráfica que se obtiene es la siguiente:

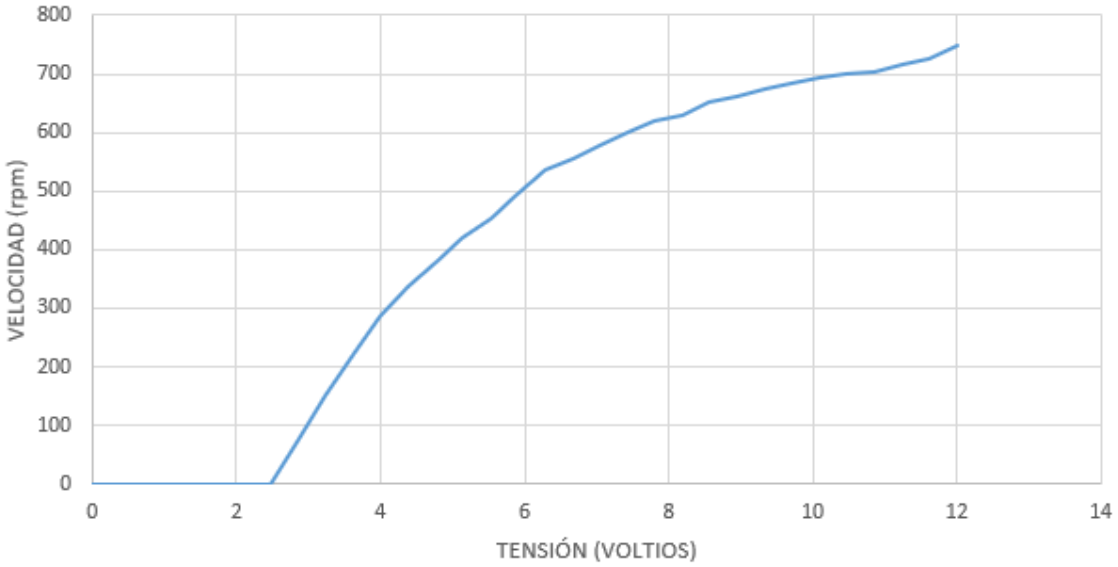


Figura 64. Representación gráfica de VELOCIDAD = f(TENSIÓN)

Se puede observar que la relación no es lineal y que existe una zona muerta en la que el motor permanece en reposo hasta que no se sobrepasan los 2.5 V aproximadamente.

Para comprobar el funcionamiento del sistema y validar que funciona correctamente, se va a introducir el código "100111" en los microinterruptores SW7…SW2 y a verificar que la velocidad mostrada en la pantalla del PC es cercana a 600 rpm. Se siguen los siguientes pasos:

1. Conectar todos los componentes que forman el sistema según la *Figura 1.*
2. Activar la alimentación de la ZedBoard y del módulo del puente en H.
3. Desactivar el SW1 para que el sistema trabaje en lazo abierto.
4. Configurar los microinterruptores SW7…SW2 para establecer el código "100111".
5. Abrir Vivado 2016.4
6. Abrir SDK.
7. Programar FPGA haciendo clic en el botón  .
8. Abrir el terminal PuTTY.
9. Configurar el terminal PuTTY según se muestra en la siguiente figura:

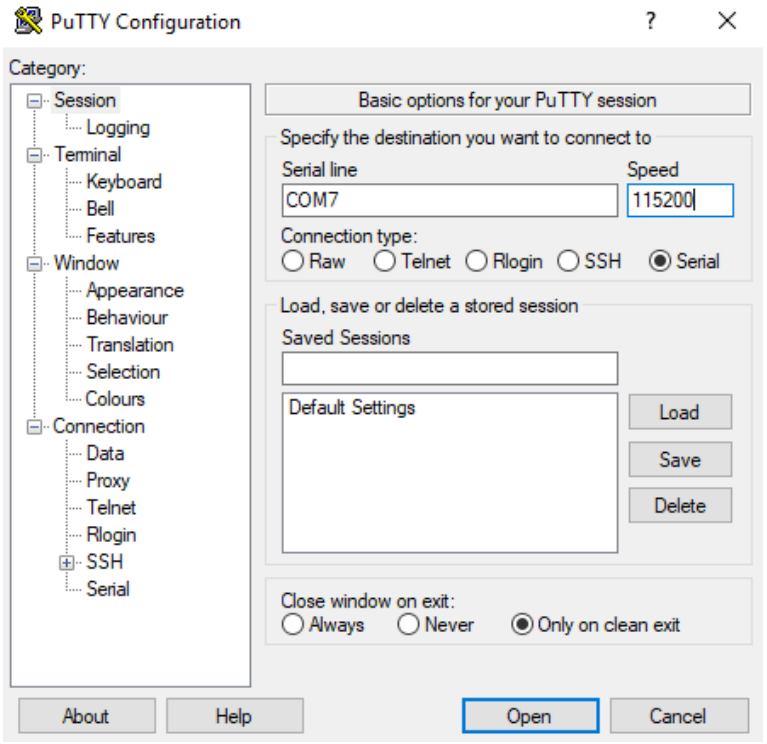


Figura 65. Configuración del terminal PuTTY

Se selecciona la opción "Serial", se determina el COMX que corresponde con el puerto USB empleado y se fija la velocidad de comunicación en 115200 baudios.

10. Finalmente, se inicia la ejecución haciendo clic en el botón  , de manera que tras unos segundos se obtienen los datos en la pantalla del PC.

```
Comienza la ejecucion...
SWITCHES y LEDS configurados
PWMcore configurado
ENCcore configurado
Interrupciones configuradas
Memoria SD y documento de texto configurados
94.74
221.05
315.79
378.95
431.58
473.68
505.26
515.79
547.37
547.37
568.42
568.42
578.95
589.47
589.47
589.47
589.47
600.00
```

Figura 66. Resultado del funcionamiento en lazo abierto

Como se observa en la anterior figura, la velocidad alcanzada por el motor es de 600 rpm, que se corresponde con el valor esperado según la relación presentada en la *Tabla 1.* Una vez realizadas estas pruebas se puede confirmar que el funcionamiento del sistema en lazo abierto es correcto.

4.2 Funcionamiento en lazo cerrado

El objetivo de este segundo apartado del capítulo es mostrar al detalle el procedimiento y los resultados de cada uno de los pasos realizados para validar el correcto funcionamiento del sistema en lazo cerrado, modo de operación en el que se realiza el registro de datos en la memoria SD.

El funcionamiento general es sencillo: se realizan una serie de cambios en los microinterruptores SW7…SW2 para determinar el valor de la referencia de velocidad. Como resultado, el regulador se encarga de controlar la planta para llevar al motor a la velocidad deseada, además de registrar en la memoria SD las variables asociadas al control PI.

Recordatorio: para que el sistema opere en lazo cerrado, el microinterruptor SW1 debe estar activado, es decir, debe tomar el valor uno o lo que es lo mismo, la variable *value*[1] tiene que ser 1.

Como el valor de la referencia de velocidad se introduce al sistema a través de los microinterruptores SW7…SW2, solamente se dispone de 64 códigos diferentes para determinar la velocidad deseada del motor. A continuación, se completa una tabla que recoge todas las posibles opciones:

CÓDIGO DECIMAL	CÓDIGO BINARIO	VELOCIDAD (RPM)
0	000000	0
1	000001	11,905
3	000011	35,715
5	000101	59,525
7	000111	83,335
9	001001	107,145
11	001011	130,955
13	001101	154,765
15	001111	178,575
17	010001	202,385
19	010011	226,195
21	010101	250,005
23	010111	273,815
25	011001	297,625
27	011011	321,435
29	011101	345,245
31	011111	369,055
33	100001	392,865
35	100011	416,675
37	100101	440,485
39	100111	464,295
41	101001	488,105
43	101011	511,915
45	101101	535,725
47	101111	559,535
49	110001	583,345
51	110011	607,155
53	110101	630,965
55	110111	654,775
57	111001	678,585
59	111011	702,395
61	111101	726,205
63	111111	750,015

Tabla 2. Relación velocidad-código de referencia

Como el rango de velocidad de 0…750 rpm se divide en 64 códigos, cada uno de los escalones que componen el rango equivale a 11,905 rpm. De esta manera, si por ejemplo se desea situar al motor en 250 rpm, se debe introducir en los microinterruptores el código “010101”

Para comprobar el funcionamiento del sistema y validar que funciona correctamente, se va a introducir el código “010101” en los microinterruptores

SW7…SW2 y a verificar que la velocidad mostrada en la pantalla del PC es cercana a 250 rpm. Se siguen los siguientes pasos:

1. Conectar todos los componentes que forman el sistema según la *Figura 1.* Se debe comprobar que la memoria SD se ha insertado con la escritura habilitada.
2. Activar la alimentación de la ZedBoard y del módulo del puente en H.
3. Activar el SW1 para que el sistema trabaje en lazo cerrado.
4. Configurar los microinterruptores SW7…SW2 para establecer el código "010101".
5. Abrir Vivado 2016.4
6. Abrir SDK.
7. Incluir la librería FatFs en el BSP (Board Support Package) del proyecto. Para ello se abre el archivo *system.mss*, se elige la opción *Modify this BSP´s Settings* y se selecciona la librería *xilffs 3.5 Generic Fat File System Library*, como queda indicado en la siguiente figura.

	Name	Version	Description
☐	libmetal	1.1	Libmetal Library
☐	lwip141	1.7	lwIP TCP/IP Stack library: lwIP v1.4.1
☐	openamp	1.2	OpenAmp Library
☑	xilffs	3.5	Generic Fat File System Library
☐	xilflash	4.2	Xilinx Flash library for Intel/AMD CFI compliant parallel flash
☐	xilisf	5.7	Xilinx In-system and Serial Flash Library
☐	xilmfs	2.2	Xilinx Memory File System
☐	xilpm	2.0	Power Management API Library for ZynqMP
☐	xilrsa	1.2	Xilinx RSA Library
☐	xilskey	6.1	Xilinx Secure Key Library

Figura 67. Librería FatFs es incluida en el BSP

8. Programar FPGA haciendo clic en el botón  .
9. Abrir el terminal PuTTY para recibir cualquier notificación de error en la ejecución.
10. Configurar el terminal PuTTY como se indica en la ***¡Error! No se encuentra el origen de la referencia.***
11. Finalmente, se inicia la ejecución haciendo clic en el botón  , de manera que tras unos segundos se almacenan los datos en la memoria SD.
12. Se desconecta la alimentación de la tarjeta ZedBoard para retirar la

memoria SD e insertarla en el PC.

13. Se abre el archivo de texto *LOG.TXT* para analizar su contenido y validar el funcionamiento del sistema implementado.

LOG: Bloc de notas

Archivo Edición Formato Ver Ayuda

0	250	0.00	250.00	5.67	24/5/2019	20:30:25
1	250	0.00	250.00	6.80		
2	250	0.00	250.00	7.93		
3	250	136.84	113.16	5.97		
4	250	242.11	7.89	4.10		
5	250	263.16	-13.16	3.67		
6	250	252.63	-2.63	3.85		
7	250	252.63	-2.63	3.82		
8	250	242.11	7.89	4.04		
9	250	242.11	7.89	4.09		
10	250	231.58	18.42	4.38		
11	250	252.63	-2.63	3.99		
12	250	242.11	7.89	4.21		
13	250	242.11	7.89	4.26		
14	250	252.63	-2.63	4.08		
15	250	242.11	7.89	4.29		
16	250	252.63	-2.63	4.10		
17	250	252.63	-2.63	4.08		

Figura 68. Archivo de texto de la prueba en lazo cerrado

En el archivo *LOG.TXT* se almacenan 6 datos: número de muestra, referencia, velocidad, error, consigna y fecha de inicio. Según los datos, el motor tarda 17x25 ms = 0.425 s en estabilizarse en la velocidad deseada partiendo del reposo.

14. Como ya se comentó en el primer capítulo, en ocasiones resulta muy útil representar los datos de forma gráfica. Para ello se emplea el script *Datosmotor.m* en Matlab.

Datosmotor.m

```
1 -    datos=load('LOG.txt');
2 -    datos=datos';
3 -    t=datos(1,:);        %Vector de tiempo fila 1
4 -    ref=datos(2,:);      %Vector referencia fila 2
5 -    vel=datos(3,:);      %Vector velocidad fila 3
6 -    error=datos(4,:);    %Vector error fila 4
7 -    plot(t,ref,'b')
8 -    hold on
9 -    plot(t,vel,'g')
10 -   plot(t,error,'r')
11 -   grid
12 -   length(t)
```

Figura 69. Script Datosmotor.m de Matlab

En el script se realizan las siguientes acciones: se convierten las columnas de datos de *LOG.TXT* en filas, es decir, se obtiene la matriz traspuesta. A continuación, se emplea el número de muestra como base de tiempo en el eje de abscisas y se representa la referencia, la velocidad, y el error en el eje de coordenadas. Por último, se calcula el número de muestras almacenadas para comprobar si alguna de ellas no se ha escrito correctamente en la memoria SD.

Tras ejecutar el script aparece en la pantalla la gráfica correspondiente:

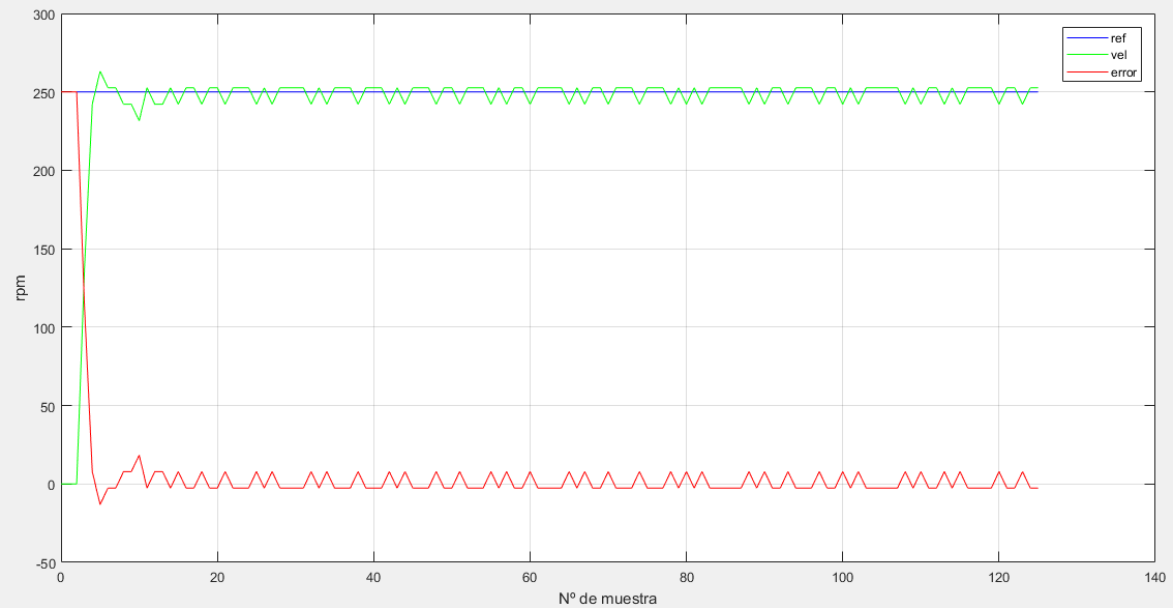


Figura 70. Representación gráfica del funcionamiento en lazo cerrado

Una característica que se desea conocer de este sistema es el ancho de banda de escritura que puede aportar al usuario, es decir, la cantidad de datos que pueden ser escritos en la memoria por unidad de tiempo. Para determinar su valor, se somete al sistema a pruebas experimentales que consisten en analizar su ejecución para valores de frecuencia de muestro cada vez mayores. De esta forma, se llega a la conclusión de que la frecuencia de muestreo máxima queda determinada en 1 KHz, ya que para frecuencias mayores aparecen errores como, por ejemplo, muestras vacías almacenadas en la memoria o un mal funcionamiento en el control PI. Para estas pruebas, en cada muestra se almacena una cadena de texto de 36 bytes, por lo que el ancho de banda máximo son 36 KB/s.

Se trata de una velocidad de escritura considerable y suficiente para este tipo de aplicaciones. Para hacernos una idea se plantea el siguiente ejemplo: si una hoja de texto tiene en torno a 30 renglones y 70 caracteres por renglón, se puede

aproximar a que ocupa 2 KB, por lo tanto, con este ancho de banda el sistema es capaz de rellenar 18 hojas de texto completas en un segundo.

Otro aspecto importante es la capacidad de la memoria SD. En este proyecto se ha empleado una memoria de 8 GB, por lo que si el sistema opera a su máxima capacidad de escritura la memoria tarda 62 horas en completarse. No obstante, hay memorias bastante asequibles de 256 GB que tardarían 83 días.

CAPÍTULO 5: PRESUPUESTO

En este capítulo se exponen todas las partidas presupuestarias necesarias para la realización del proyecto. En él se desglosarán tanto el coste del material empleado como el coste del software necesario para el desarrollo de los diferentes apartados del proyecto.

5.1 Coste del material empleado

En la *Tabla 3* se refleja el listado del coste asociado a cada uno de los elementos materiales que son utilizados en este proyecto.

Material	Precio unitario (€/ud)	Unidades	Periodo de amortización (años)	Uso (meses)	Coste de amortización (€)
ZedBoard Development Kit	475	1	3	6	79,17
Puente en H PmodHB5	20	1	2	6	5
Motor DC	16	1	2	6	4
Fuente de alimentación	150	1	4	6	18,75
Memoria SD	15	1	3	6	2,5
PC de desarrollo	750	1	3	6	125
Total	**1426**				**234,42**

Tabla 3. Coste del material empleado

De esta manera, es necesario realizar un desembolso de 1426 € de los que solamente se amortizan 234.42 €.

A continuación, se procede a analizar el coste asociado con las licencias software utilizadas para desarrollar los diferentes apartados del proyecto.

5.2 Coste del software empleado

En la *Tabla 4* se detalla el coste que implican las licencias software utilizadas en el proyecto.

Licencia Software	Precio (€)	Periodo de amortización (años)	Uso (meses)	Coste de amortización (€)
Vivado Design Suite 2016.4	0	2	6	0
MATLAB R2017a	2000	2	6	500
Microsoft Office 2016	450	2	6	112,5
Total	**2450**			**612,5**

Tabla 4. Coste del software empleado

Como caso particular, a la licencia *Vivado Design Suite 2016.4* se le ha asociado un coste de 0 € ya que forma parte del material *ZedBoard Development Kit*.

De esta manera, es necesario realizar un desembolso de 2450 € de los que solamente se amortizan 612.5 €.

5.3 Coste del proyecto

En la *Tabla 5* se realiza el resumen presupuestario del proyecto.

Material	Coste íntegro del proyecto (€)	Coste de amortización (€)
Coste del material empleado	1426	234,42
Coste del software empleado	2450	612,5
Total (sin IVA)	3876	846,92
Total (21% IVA incluido)	**4689,96**	**1024,77**

Tabla 5. Coste del proyecto

Para acabar, el importe total estimado del proyecto presente en este documento asciende a la cantidad de:

Mil veinticuatro euros con setenta y siete céntimos.

ANEXO 1: CÓDIGO VHDL DE PWMcore

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.std_logic_unsigned.all;
4   use ieee.std_logic_arith.all;
5
6   entity PWMcore is
7     port(
8       mclk               : in  std_logic;
9       mrst               : in  std_logic;
10      direction          : in  std_logic;
11      carrier_period     : in  std_logic_vector(31 downto 0);
12      mod_value          : in  std_logic_vector(31 downto 0);
13      deadtime           : in  std_logic_vector(15 downto 0);
14      dir                : out std_logic;
15      PWM                : out std_logic);
16  end PWMcore;
17
18  architecture RTL of PWMcore is
19    signal cnt              : std_logic_vector(31 downto 0);       --señal portadora
      diente de sierra
20    signal cnt_deadtime     : std_logic_vector(15 downto 0);       --contador del
      deadtime
21    signal PWMi             : std_logic;     --PWM intermedia, sin tiempos muertos
22    signal direction_prev   : std_logic;     --valor anterior de direction
23    signal flag_dir         : std_logic;     --flag de cambio en direction
24    signal dead_cnt_flag    : std_logic;     --flag de inicio y fin deadtime
25  begin
26
27  ----Inicio: Carrier generator----
28    --Inicio: Generación de la portadora--
29    CNT_carrier : process (mclk, mrst) is
30    begin
31      if (mrst = '0') then                  --rst activo a L
32        cnt <= (others => '0');
33      elsif (mclk'event and mclk = '1') then
34        if (cnt = carrier_period - 1) then  --limsup de cnt
35          cnt <= (others => '0');
36        else
37          cnt <= cnt + 1;
38        end if;
39      end if;
40    end process;
41    --Fin: Generación de la portadora--
42
43    --Inicio: Modulación PWM--
44    COMP : process (mrst, cnt, mod_value) is
45    begin
46      if (mrst = '0') then                --rst activo a L
47        PWMi <= '0';
48      elsif (cnt <= mod_value) then       --comparación
49        PWMi <= '1';
50      else
51        PWMi <= '0';
52      end if;
53    end process;
54    --Fin: Modulación PWM--
55  ----Fin: Carrier generator----
56
57  ----Inicio: Deadtime control----
58    --Inicio: Registro de direction--
59    process (mclk, mrst)
60    begin
61      if (mrst = '0') then
62        direction_prev <= '0';
63      elsif (mclk'event and mclk = '1') then
64        direction_prev <= direction;
65      end if;
66    end process;
67    --Fin: Registro de direction--
68
69    --Inicio: Detección cambio en direction--
70    process (mclk, mrst)
71    begin
```

```
72      if (mrst = '0') then
73        flag_dir <= '0';
74      elsif (mclk'event and mclk = '1') then
75        if (direction_prev /= direction) then
76          flag_dir <= '1';      --Detectado un cambio en direction
77        else
78          flag_dir <= '0';
79        end if;
80      end if;
81    end process;
82    --Fin: Detección cambio en direction--
83
84    --Inicio: Contador del deadtime--
85    CNT_dead_time : process (mrst, mclk) is
86    begin
87      if (mrst = '0') then
88        cnt_deadtime <= (others => '0');
89      elsif (mclk'event and mclk = '1') then
90        if (flag_dir = '1') then          --si se detecta cambio en direction
91          cnt_deadtime <= deadtime - 1;      --cargamos valor sup de cuenta
92        elsif (cnt_deadtime > 0) then
93          cnt_deadtime <= cnt_deadtime - 1;  --decrementamos
94        end if;
95      end if;
96    end process;
97
98    dead_cnt_flag <= '1' when (cnt_deadtime /= 0) else '0'; --flag vale 0 al acabar
      cuenta
99    --Fin: Contador del deadtime--
100
101   --Inicio: Generación de señal PWM--
102   process (mclk, mrst)
103   begin
104     if (mrst = '0') then
105       PWM <= '0';
106     elsif (mclk'event and mclk = '1') then
107       if (dead_cnt_flag = '0' and flag_dir = '0') then
108         PWM <= PWMi;
109       else
110         PWM <= '0';    --PWM vale 0 si se ha detectado cambio en direction y durante
            el tiempo de deadtime
111       end if;
112     end if;
113   end process;
114   --Fin: Generación de señal PWM--
115
116   --Inicio: Generación señal dir--
117   process (mclk, mrst)
118   begin
119     if (mrst = '0') then
120       dir <= '0';
121     elsif (mclk'event and mclk = '1') then
122       if (dead_cnt_flag = '0' and flag_dir = '0' and direction=direction_prev) then
123         dir <= direction;    --se conmuta dir una vez transcurrido deadtime
124       end if;
125     end if;
126   end process;
127   --Fin: Generación de señal dir--
128 ----Fin: Deadtime control----
129
130 end RTL;
131
```

ANEXO 2: CÓDIGO VHDL DE ENCcore

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.std_logic_arith.all;
5
6  entity ENCODERcore is
7    port(
8      mclk              : in  std_logic;
9      mrst              : in  std_logic;
10     obs_period        : in  std_logic_vector (31 downto 0);
11     CHA               : in  std_logic;
12     CHB               : in  std_logic;
13     enc_cnt           : out std_logic_vector (23 downto 0);
14     INTERRUP_ARM_i    : out std_logic);
15 end ENCODERcore;
16
17 architecture RTL of ENCODERcore is
18
19   signal cnt_obs           : std_logic_vector(31 downto 0);  --contador tiempo de
     observación
20   signal cnt_FSM           : std_logic_vector(23 downto 0);  --contador transiciones
     estados
21   signal update            : std_logic;  --señal de fin de tiempo de observación
22   signal CHAi, CHBi        : std_logic;  --señales del encoder filtradas
23   signal entrada, estado   : std_logic_vector(1 downto 0);  --entrada a FSM y estado
24   signal up_down, en       : std_logic;
25
26   --Declaracion de FILTRO_DIGITAL
27   component FILTRO_DIGITAL is
28     port (
29       rst  : in  std_logic;
30       clk  : in  std_logic;
31       din  : in  std_logic;
32       dout : out std_logic);
33   end component FILTRO_DIGITAL;
34
35
36 begin
37
38 ----Inicio: Filtro digital----
39   FILTRO_DIGITAL_CHA: entity work.FILTRO_DIGITAL
40   port map (
41     rst  => mrst,
42     clk  => mclk,
43     din  => CHA,     --entrada
44     dout => CHAi);   --salida
45
46   FILTRO_DIGITAL_CHB: entity work.FILTRO_DIGITAL
47   port map (
48     rst  => mrst,
49     clk  => mclk,
50     din  => CHB,     --entrada
51     dout => CHBi);   --salida
52 ----Fin: Filtro digital----
53
54 ----Inicio: Contador periodo----
55   --Inicio: contador de tiempo de observación--
56   CNT_obs_period : process (mclk, mrst) is
57   begin
58     if (mrst = '0') then
59       cnt_obs <= (others => '0');
60     elsif (mclk'event and mclk = '1') then
61       if (cnt_obs = obs_period - 1) then
62         cnt_obs <= (others => '0');
63       else
64         cnt_obs <= cnt_obs + 1;
65       end if;
66     end if;
67   end process;
68   --Fin: contador de tiempo de observación--
69
70   --Inicio: activación señal update--
71   process (mclk, mrst) is
```

```
72      begin
73        if (mrst = '0') then
74          update <= '0';
75        elsif (mclk'event and mclk = '1') then
76          if (cnt_obs = obs_period - 1) then
77            update <= '1';
78          else
79            update <= '0';
80          end if;
81        end if;
82      end process;
83      --Fin: activación señal update--
84
85      INTERRUP_ARM_i <= update;
86    ----Fin: Contador periodo----
87
88    ----Inicio: FSM----
89      entrada <= CHAi & CHBi;       --entrada <= "CHAi CHBi", p.e, "10"
90
91      FSM : process (mclk, mrst) is
92      begin
93        if (mrst = '0') then
94          estado <= "00";
95          en <= '0';
96          up_down <= '0';     --"0" down, "1" up
97        elsif (mclk'event and mclk = '1') then
98          en <= '0';
99          case estado is
100           when "00" =>
101             if (entrada = "10") then
102               estado  <= "10";
103               up_down <= '1';  --cnt_FSM se incrementará
104               en      <= '1';
105             elsif (entrada = "01") then
106               estado  <= "01";
107               up_down <= '0';  --cnt_FSM se decrementará
108               en      <= '1';
109             end if;
110
111           when "10" =>
112             if (entrada = "11") then
113               estado  <= "11";
114               up_down <= '1';  --cnt_FSM se incrementará
115               en      <= '1';
116             elsif (entrada = "00") then
117               estado  <= "00";
118               up_down <= '0';  --cnt_FSM se decrementará
119               en      <= '1';
120             end if;
121
122           when "11" =>
123             if (entrada = "01") then
124               estado  <= "01";
125               up_down <= '1';  --cnt_FSM se incrementará
126               en      <= '1';
127             elsif (entrada = "10") then
128               estado  <= "10";
129               up_down <= '0';  --cnt_FSM se decrementará
130               en      <= '1';
131             end if;
132
133           when others =>                  --caso para estado 01
134             if (entrada = "00") then
135               estado  <= "00";
136               up_down <= '1';  --cnt_FSM se incrementará
137               en      <= '1';
138             elsif (entrada = "11") then
139               estado  <= "11";
140               up_down <= '0';  --cnt_FSM se decrementará
141               en      <= '1';
142             end if;
143         end case;
144       end if;
```

```
145     end process;
146   ----Fin: FSM----
147
148   ----Inicio: Contador----
149     contador_FSM : process (mclk, mrst) is
150     begin
151       if (mrst = '0') then
152         cnt_FSM <= (others => '0');
153       elsif (mclk'event and mclk = '1') then
154         if (update = '1') then
155           cnt_FSM <= (others => '0');
156         elsif (en = '1') then
157           if (up_down = '1') then  --"0" down, "1" up
158             cnt_FSM <= cnt_FSM + 1;
159           else
160             cnt_FSM <= cnt_FSM - 1;
161           end if;
162         end if;
163       end if;
164     end process;
165   ----Fin: Contador----
166
167   ----Inicio: Reg----
168     cnt_FSM_reg : process (mclk, mrst) is
169     begin
170       if (mrst = '0') then
171         enc_cnt <= (others => '0');
172       elsif (mclk'event and mclk = '1') then
173         if (update = '1') then
174         enc_cnt <= cnt_FSM;    --se almacena cnt_FSM en enc_cnt
175         end if;
176       end if;
177     end process;
178   ----Fin: Reg----
179   end architecture RTL;
180
```

ANEXO 3: CÓGIDO VHDL DE FILTRO DIGITAL

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity FILTRO_DIGITAL is
6    port (
7      rst  : in  std_logic;
8      clk  : in  std_logic;
9      din  : in  std_logic;
10     dout : out std_logic);
11 end entity;
12
13 architecture rtl of FILTRO_DIGITAL is
14  signal tap       : std_logic_vector (3 downto 0);
15  signal all_ones  : std_logic;
16  signal all_zeros : std_logic;
17
18 begin
19   process (clk, rst)
20     begin
21       if (rst = '0') then
22         tap <= (others => '0');
23       elsif (clk'event and clk='1') then
24         tap <= tap (2 downto 0) & din;
25       end if;
26     end process;
27
28     all_ones  <= '1' when (tap ="1111") else '0';
29     all_zeros <= '1' when (tap ="0000") else '0';
30
31   process (clk, rst)
32     begin
33       if (rst = '0') then
34         dout <= '0';
35       elsif (clk'event and clk='1') then
36         if (all_ones = '1') then
37           dout <= '1';
38         elsif (all_zeros = '1') then
39           dout <= '0';
40         end if;
41       end if;
42   end process;
43 end rtl;
44
```

ANEXO 4: CÓDIGO EN C DEL PROYECTO

```
// ----------------------------------------------------------------------
// REGISTRO DE DATOS MASIVO EN PLATAFORMAS SOC PARA APLICACONES DE POTENCIA
// RAMÓN HEREDERO BERMEJO
// UNIVERSIDAD DE ALCALÁ DE HENARES
// 24/05/2019
// ----------------------------------------------------------------------

#include <stdio.h>
#include "xil_printf.h"
#include "platform.h"
#include "xparameters.h"
#include "xgpio_l.h"
#include "PWMcore_ip.h"
#include "EncCore.h"
#include "ff.h"
#include "xscugic.h"

//FRECUENCIA DE RELOJ PRINCIPAL
#define CLK 100e6

//PARÁMETROS DEL CONTROL DE VELOCIDAD
#define K    0.94458
#define CERO 0.8

//REGISTROS DEL PERIFÉRICO PWMcore
#define XPAR_AXI_PWMcore_BASEADDR          0x43c00000
#define PWMCORE_IP_S00_AXI_SLV_REG0_OFFSET 0    //direction (1 bit)
#define PWMCORE_IP_S00_AXI_SLV_REG1_OFFSET 4    //carrier_period (32 bits)
#define PWMCORE_IP_S00_AXI_SLV_REG2_OFFSET 8    //mod_value (32 bits)
#define PWMCORE_IP_S00_AXI_SLV_REG3_OFFSET 12   //deadtime (16 bits)

//PARÁMETROS DE CONFIGURACIÓN DEL PERIFÉRICO PWMcore
#define CARRIER_PERIOD 0x0000c350    //50000 Tclks para obtener 2KHz en PWM
#define DEADTIME       0x0000ffff    //0.65ms

//REGISTROS DEL PERIFÉRICO ENCcore
#define XPAR_AXI_ENCcore_BASEADDR       0x43c10000
#define ENCCORE_S00_AXI_SLV_REG0_OFFSET 0    //obs_period(32 bits)
#define ENCCORE_S00_AXI_SLV_REG1_OFFSET 4    //enc_cnt(24 bits)
#define ENCCORE_S00_AXI_SLV_REG2_OFFSET 8    //IER(1 bit)
#define ENCCORE_S00_AXI_SLV_REG3_OFFSET 12   //ISR(1 bit)

//PARÁMETROS DE CONFIGURACIÓN DEL PERIFÉRICO ENCcore
#define OBS_PERIOD 2.5e6    //1e6 Tclks para obtener un obs_period de 10ms
#define IER        0x00000001    //slv_reg2(0) habilitado


//VARIABLES ASOCIADAS A LA MEMORIA SD
//Variable tipo FATFS asociada a la memoria SD
FATFS FS_instance;

//Variable tipo FIL asociada al archivo de texto en la memoria
FIL file1;

//Variable tipo FRESULT para almacenar el valor que retornan las funciones API*/
FRESULT result;

//Argumento de entrada a la función f_mount
TCHAR *Path = "0:/";


//VARIABLES GLOBALES
static char FileName[32] = "Log.txt"; //Nombre del archivo de texto
static char *Log_File; //Puntero al archivo de texto
unsigned int BytesWr; //Número de bytes escritos en el archivo de texto
int len=0;  //Longitud de la cadena a escribir en el archivo de texto
int accum=0;  //Variable para almacenar el EOF
u32 Buffer_size;
char Buffer_logger[64];  //Cadena a escribir en el archivo de texto

float rpm_f;  //Valor de rpm de la reductora
int cnt=0;  //Número de línea en archivo de texto
uint32_t referencia;  //Valor de referencia de velocidad
```

```
74  float error;  //Error en el control de velocidad
75  float consigna;  //Valor de salida del controlador
76  float consigna_V;  //Valor de consigna en voltios
77  uint32_t ENCcore_flag;  //Flag de la interrupción que desencadena ENCcore
78
79
80  //FECHA DE COMIENZO DEL REGISTRO DE DATOS
81  uint32_t YEAR = 2019;
82  uint32_t MONTH = 05;
83  uint32_t DAY = 24;
84  uint32_t HOUR = 20;
85  uint32_t MINUTE = 30;
86  uint32_t SECOND = 25;
87
88
89  //VARIABLES ASOCIADAS AL CONTROLADOR DE INTERRUPCIONES
90  XScuGic_Config* SCUGIC_cfg;
91  XScuGic SCUGIC_obj;
92  int status;  //Retorno de las funciones de configuración de la interrupción
93
94
95
96  void Config_GPIO(void){
97      //SW0...SW7 SON ENTRADAS
98      XGpio_WriteReg(XPAR_AXI_GPIO_0_BASEADDR, XGPIO_TRI_OFFSET, 0xFF);
99      //LD0...LD7 SON SALIDAS
100     XGpio_WriteReg(XPAR_AXI_GPIO_1_BASEADDR, XGPIO_TRI_OFFSET, 0x00);
101     printf("SWITCHES y LEDS configurados\r\n");
102 }
103
104 void Config_PWMcore(void){
105     //PERIODO DE LA SEÑAL PWM
106     PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
        PWMCORE_IP_S00_AXI_SLV_REG1_OFFSET, CARRIER_PERIOD);
107     //TIEMPOS MUERTOS AL CONMUTAR DIRECCIÓN DE GIRO
108     PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
        PWMCORE_IP_S00_AXI_SLV_REG3_OFFSET, DEADTIME);
109     printf("PWMcore configurado\r\n");
110 }
111
112 void Config_ENCcore(void){
113     //TIEMPO DE CUENTA
114     ENCCORE_mWriteReg(XPAR_AXI_ENCcore_BASEADDR, ENCCORE_S00_AXI_SLV_REG0_OFFSET,
        OBS_PERIOD);
115     printf("ENCcore configurado\r\n");
116 }
117
118 void ENCcore_ISR (void){
119     //Flag de interrupción a nivel SW
120     ENCcore_flag = 1;
121
122     //Flag de interrupción a nivel HW sobre ENCcore
123     ENCCORE_mWriteReg(XPAR_AXI_ENCcore_BASEADDR, ENCCORE_S00_AXI_SLV_REG2_OFFSET,
        0x00000001);
124 }
125
126 int Config_Interrupt(void){
127     Xil_ExceptionDisable();  //Deshabilitación de interrupciones
128     SCUGIC_cfg= XScuGic_LookupConfig (XPAR_SCUGIC_0_DEVICE_ID);
129     if (SCUGIC_cfg == NULL)
130     {
131         xil_printf ("Error en la configuracion de Interrupciones\r\n");
132         return -1;
133     }
134
135     status= XScuGic_CfgInitialize(&SCUGIC_obj, SCUGIC_cfg,
        SCUGIC_cfg->CpuBaseAddress);
136     if (status != XST_SUCCESS)
137     {
138         xil_printf ("Error en la configuracion de Interrupciones\r\n");
139         return -1;
140     }
141
```

```
142     Xil_ExceptionInit();
143     Xil_ExceptionRegisterHandler (XIL_EXCEPTION_ID_IRQ_INT, (Xil_ExceptionHandler)
        XScuGic_InterruptHandler, &SCUGIC_obj);
144
145     //Habilitación de interrupción de FPGA Fabric (Encoder)
146     status= XScuGic_Connect (&SCUGIC_obj, XPS_FPGA0_INT_ID, (Xil_ExceptionHandler)
        ENCcore_ISR, NULL);
147     if (status != XST_SUCCESS)
148     {
149         xil_printf ("Error en la configuracion de Interrupción ENCcore\r\n");
150         return -1;
151     }
152
153     XScuGic_Enable(&SCUGIC_obj, XPS_FPGA0_INT_ID);
154
155     //Habilitación de interrupciones
156     Xil_ExceptionEnable();
157     xil_printf ("Interrupciones configuradas\r\n");
158 }
159
160 void Config_fileSD(void){
161     // Inicialización de la memoria SD
162     result = f_mount(&FS_instance,Path, 1);
163
164         if (result != 0){
165             printf("ERROR %d al ejecutar la funcion f_mount\r\n", result);
166             return XST_FAILURE;
167         }
168
169     // Se crea un nuevo documento de texto con permisos de lectura/escritura
170     Log_File = (char *)FileName;
171
172     result = f_open(&file1, Log_File, FA_CREATE_ALWAYS | FA_WRITE | FA_READ);
173
174         if (result!= 0){
175             printf("ERROR %d al ejecutar la funcion f_open\r\n", result);
176             return XST_FAILURE;
177         }
178
179         printf("Memoria SD y documento de texto configurados\r\n");
180 }
181
182 void WriteSD(void){
183     // Apertura del documento de texto en modo escritura
184     result = f_open(&file1, Log_File,FA_WRITE);
185
186         if (result!=0) {
187             printf("ERROR al ejecutar la función f_open\r\n");
188             return XST_FAILURE;
189         }
190
191     // Situar nuevos datos a partir del último dato escrito
192     result = f_lseek(&file1,accum);
193
194         if (result!=0) {
195             printf("ERROR al ejecutar la función f_lseek\r\n");
196             return XST_FAILURE;
197         }
198
199     // Dar forma al paquete de informacion a guardar
200     if (cnt<1) // PRIMERA ESCRITURA EN MEMORIA SD
201     {
202         sprintf(Buffer_logger, "%d    %3d    %3.2f    %3.2f    %3.2f    %d/%d/%d
            %d:%d:%d\r\n", cnt, referencia, rpm_f, error, consigna_V, DAY, MONTH, YEAR,
            HOUR, MINUTE, SECOND);
203     }
204     else  // RESTO DE ESCRITURAS POSTERIORES
205     {
206         sprintf(Buffer_logger, "%d    %3d    %3.2f    %3.2f    %3.2f\r\n", cnt,
            referencia, rpm_f, error, consigna_V);
207     }
208
209     // CÁLCULO DEL TAMAÑO DEL PAQUETE DE DATOS
```

```
210         len = strlen(Buffer_logger);
211         Buffer_size=len;
212
213         cnt=cnt+1;  // número de renglón en el ARCHIVO DE TEXTO
214
215         // Escritura en el documento de texto
216         result = f_write(&file1, (const void*)Buffer_logger, Buffer_size,&BytesWr);
217
218             if (result!=0) {
219                 printf("ERROR al ejecutar la función f_write\r\n");
220                 return XST_FAILURE;
221             }
222
223         // Incremetar EOF
224         accum=accum+len;
225
226         // Cerrar archivo de texto
227         result = f_close(&file1);
228
229             if (result!=0) {
230                 printf("ERROR al ejecutar la función f_close\r\n");
231                 return XST_FAILURE;
232             }
233         }
234
235     int main()
236     {
237         uint32_t value;           //Valor de SW0...SW7
238         uint32_t value_anterior;  //Detección de cambios en SW0...SW7
239         uint32_t mod_value;       //Valor que escribiremos en el registro mod_value (32
        bits)
240         uint32_t cuentas;         //Valor leído del registro enc_cnt (24 bits)
241         int error_NM1=0;          //Error en la muestra anterior
242         float consigna_NM1=0;     //Consigna en la muestra anterior
243
244         //INICIO DEL SISTEMA
245         init_platform();
246         printf("Comienza la ejecucion...\r\n");
247
248         //FUNCIONES DE CONFIGURACIÓN
249         Config_GPIO();
250         Config_PWMcore();
251         Config_ENCcore();
252         Config_Interrupt();
253         Config_fileSD();
254
255
256       while(1)  //Funcionamiento continuo del sistema
257       {
258          //Lectura de los 8switches
259          value = XGpio_ReadReg(XPAR_AXI_GPIO_0_BASEADDR, XGPIO_DATA_OFFSET);
260
261          if (value_anterior != value)  //Se ha modificado algún switch
262          {
263              value_anterior = value;
264
265              //Escritura en 8leds
266              XGpio_WriteReg(XPAR_AXI_GPIO_1_BASEADDR, XGPIO_DATA_OFFSET, value);
267
268              //Escritura en registro direction (bit 0 de value)
269              PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
                 PWMCORE_IP_S00_AXI_SLV_REG0_OFFSET, value);
270          }
271
272          // Se borra el flag a nivel HW en ENCcore de forma recurrente
273          ENCCORE_mWriteReg(XPAR_AXI_ENCcore_BASEADDR, ENCCORE_S00_AXI_SLV_REG2_OFFSET,
             0x00000000);
274
275          if (value & (1<<1))     //Activado control en lazo cerrado
276          {
277              if (ENCcore_flag & (1<<0))     //slv_reg3(0) = 1, se ha llegado al fin de
                 obs_period
278              {
```

```
279                 ENCcore_flag = 0;  //Se borra el falg a nivel SW
280                 referencia = (value >> 2) * 11.905;    //750 rpm máximas/63 valores
                    posibles=11.9047
281                 //Lectura del registro enc_cnt (24 bits)
282                 cuentas = ENCCORE_mReadReg(XPAR_AXI_ENCcore_BASEADDR,
                    ENCCORE_S00_AXI_SLV_REG1_OFFSET);
283                 cuentas = cuentas & 0x00FFFFFF;
284
285                 if ((cuentas & (1<<23)) == 0)  //Número positivo '0', giro en sentido
                    antihorario, cnt_FSM se incrementa
286                 {
287                     //rpm float de la reductora
288                     rpm_f = ((cuentas*(60*(CLK/OBS_PERIOD)))/(4*19))/3;
289                 }
290                 else   //Número negativo '1', giro en sentido horario, cnt_FSM se
                    decrementa
291                 {
292                     //rpm float de la reductora
293                     rpm_f = ((((0x00FFFFFF - cuentas) +
                        1)*(60*(CLK/OBS_PERIOD)))/(4*19))/3;
294                 }
295
296                 //ALGORITMO DE CONTROL DE VELOCIDAD
297                 error = referencia - rpm_f;
298                 consigna = consigna_NM1 + K*error - CERO*K*error_NM1;
299
300                 if (consigna > 500)
301                 consigna = 500;
302                 if (consigna < 0)
303                 consigna = 0;
304
305                 //Cálculo de la consigna en voltios (0...12V)
306                 consigna_V = (consigna*12)/500;
307
308                 error_NM1 = error;
309                 consigna_NM1 = consigna;
310
311                 //Prueba lazo cerrado control PI
312 //              printf("%3d    %3.2f    %3.2f    %3.2f\r\n", referencia, rpm_f, error,
    consigna_V);
313
314                 //Escritura en memoria SD
315                 WriteSD();
316
317                 //Actuación sobre el motor DC
318                 mod_value = consigna * 100;
319                 PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
                    PWMCORE_IP_S00_AXI_SLV_REG2_OFFSET, mod_value);
320             }
321         }
322         else     //Activado control en lazo abierto
323         {
324
325             if (value_anterior != value)
326             {
327                 value_anterior = value;
328
329                 //Escritura en registro mod_value (bits 7-2 de value)
330                 mod_value = (value >> 2) * 793;  //50000/63=793.65
331
332                 //Actuación sobre el motor DC
333                 PWMCORE_IP_mWriteReg(XPAR_AXI_PWMcore_BASEADDR,
                    PWMCORE_IP_S00_AXI_SLV_REG2_OFFSET, mod_value);
334             }
335
336             if (ENCcore_flag & (1<<0))    //slv_reg3(0) = 1, se ha llegado al fin de
                obs_period
337             {
338                 ENCcore_flag = 0;  //Se borra el falg a nivel SW
339                 //Lectura del registro enc_cnt (24 bits)
340                 cuentas = ENCCORE_mReadReg(XPAR_AXI_ENCcore_BASEADDR,
                    ENCCORE_S00_AXI_SLV_REG1_OFFSET);
341                 cuentas = cuentas & 0x00FFFFFF;
```

```
342
343                 if ((cuentas & (1<<23)) == 0)  //Número positivo, giro en sentido
                    antihorario '0', cnt_FSM se incrementa
344                 {
345                     //rpm float de la reductora
346                     rpm_f = ((cuentas*(60*(CLK/OBS_PERIOD)))/(4*19))/3;  //rpm float de
                        la reductora
347                 }
348                 else   //Número negativo, giro en sentido horario '1', cnt_FSM se
                    decrementa
349                 {
350                     //rpm float de la reductora
351                     rpm_f = ((((0x00FFFFFF - cuentas) +
                        1)*(60*(CLK/OBS_PERIOD)))/(4*19))/3;  //rpm float de la reductora
352                 }
353
354                 //Mostramos la velocidad del motor en el monitor del PC
355                 printf("%3.2f\r\n", rpm_f);
356             }
357         }
358     }
359
360       cleanup_platform();
361       return 0;
362   }
363
```

BIBLIOGRAFÍA

[Ababei, 2013] C. Ababei, "Lecture 12: SPI and SD cards", Electrical Engineering Department, University of Buffalo, USA, 2013.

[ChaN, 2014] E. ChaN, "FatFs – Fat File system module", The Electronic Lives Manufacturing, 2014.

[Crocket, 2014] L. H. Crockett, R. A. Elliot, M. A. Enderwitz and R. W. Stewart, "The Zynq Book, Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC". First Edition, Glasgow: University of Strathclyde, 2014.

[Henao, 2010] C. A. Henao, E. D. Cardona, "How to use a SD/MMC memory with PIC16F87x", Departamento de Ingeniería Eléctrica, Universidad Tecnológica de Pereira, Pereira, Colombia, 2010.

[SD Card Association, 2018] "SD Specifications Part 1: Physical Layer Simplified Specification". Version 6.0, Technical Committee SD Card Association, 2018.

[SEDA, 2018] "Sistemas Electrónicos Digitales Avanzados", apuntes de clase de la asignatura 600032, Departamento de Electrónica, Universidad de Alcalá, 2018.

[Taylor, 2014] A. P. Taylor, "How to use Interrupts on the Zynq SoC", *Xcell Journal*, Second Quarter, 2014.

[ZedBoard, 2013] ZedBoard Schematics, Second edition, http://www.zedboard.org/support/documentation/1521, último acceso agosto 2019.

[ZedBoard, 2014] ZedBoard Hardware User´s Guide, First edition, http://www.zedboard.org/support/documentation/1521, último acceso agosto 2019.

Universidad de Alcalá

Escuela Politécnica Superior

ESCUELA POLITECNICA SUPERIOR