



Proyecto de SED: Comunicación entre placas.

Comunicación serie entre STM32 y FPGA.

INTEGRANTES DEL GRUPO:

FRANCISCO JAVIER VALERO CUADRADO: 54245

ADRIÁN SÁNCHEZ ATIENZA: 53430

JORGE VARA AGUSTÍN: 53443

INDICE

1- Introducción del proyecto.

1.1- Componentes necesarios

1.2- Objetivos del proyecto

2- Comunicación

2.1- Comunicación I2C

2.1.1- Funcionalidad e interfaz del Módulo I2C

2.1.2- Simulaciones del módulo I2C

2.1.3- Máquina de estados del módulo I2C

2.1.4- Testeo sencillo en ARDUINO + Saleae Logic8

2.1.5- Testeo sencillo con microprocesador STM32F407

2.2- Errores en el desarrollo.

2.2.1- I2C.

2.2.2- Comunicación SPI.

2.2.2.1- SPI Full-Duplex.

2.2.2.2- SPI Half-Duplex.

3- Control PWM del servo implementado en VHDL.

3.1- Marco teórico

3.2- Caracterización del servo SG90

3.3- Diseño del control PWM en Lenguaje Descripción Hardware. VHDL.

3.3.1- Cálculos teóricos e Implementación.

3.3.1.1- Divisor de frecuencia.

3.3.1.2- Generador señal PWM.

3.3.2- Simulaciones en VIVADO.

3.4- Esquema de conexión y Montaje

4- MPU6050.

4.1- Configuración inicial del dispositivo.

4.1.1- WHO_AM_I

4.1.2- PWR_MGMT_1_REG

4.1.3- SMPLRT_DIV_REG

4.1.4- ACCEL_CONFIG_REG y GYRO_CONFIG_REG

4.1.5- Implementación en STMCubeIDE.

4.2- Lectura de aceleración. Aceleración angular.

4.2.1 Obtención del ángulo de inclinación.

4.2.2 Implementación en código.

4.3 Lectura del giroscopio. Velocidad angular.

4.3.1 Obtención del giro de dispositivo.

4.3.2 Errores en las medidas.

4.3.3 Implementación en el código.

4.3.4 Prueba y demostración de uso.

5- Pantalla LCD 16x2 con bus I2C conectado.

5.1 Librerías y configuración

5.2 Funciones y uso

6- Prueba final y conexiones.

6.1- Esquema de conexión.

7- Conclusiones del proyecto.

8- Bibliografía.

1. Introducción del proyecto.

En esta memoria se explicará el desarrollo del trabajo de la asignatura de Sistemas Electrónicos Digitales, tanto los errores que se produjeron en él como los avances. Se abordará así mismo un marco teórico sobre protocolos utilizar para la conexión, inicialización de dispositivos o formas de onda.

A la hora de la elección del proyecto se decidió realizar el proyecto conjunto STM32 y Nexus4DDR, en el que se propuso conectar ambas placas mediante un protocolo de comunicación y que trabajaran en conjunto para un fin común.

1.1. Componentes necesarios.

Los componentes que se utilizarán para el desarrollo del trabajo serán los siguientes:

- STM32F407 Discovery.
- FPGA Nexys 4DDR.
- Módulo MPU6050 con giróscopo y acelerómetro.
- Pantalla LCD 16X2 con modulo I2C conectado.
- Servomotor SG90.
- Fuente de alimentación 5V de protoboard.
- Convertidor de nivel lógico 3.3V a 5V.
- Cables de conexión.
- Analizador de ondas Saleae Logic8 para la depuración de la comunicación.
- Vivado y STM32CubeIDE para la programación.

Adicionalmente, el equipo adquirió una FPGA “**Basys 3**” como placa de test en remoto.

1.2. Objetivo del proyecto.

El proyecto centrará su funcionamiento en un mecanismo de balanceo, es decir, según una determinada inclinación de la MPU poder corregir esa inclinación moviendo un servo. Así mismo se habilitará una opción asíncrona en la que mediante un KeyPad pueda seleccionarse el grado de inclinación.

Por lo tanto, los objetivos de este proyecto serán:

- Comunicación de la placa Nexus 4DDR y STM32F407G mediante protocolo de comunicación I2C en la que la STM32 actuará como maestro iniciando y terminando toda comunicación y la Nexus como esclavo.
- STM32: Lectura de los registros de giro del dispositivo MPU6050, se trata de un chip con un acelerómetro y un giróscopo integrado. La STM32 deberá ser capaz de mediante tecnología I2C, comunicarse con el dispositivo, inicializarlo y leer sus valores.
- STM32: Lectura de los botones marcados por el usuario sobre la inclinación del dispositivo, se realizará mediante interrupciones.
- FPGA: Obtención de los valores de inclinación de la STM32 en un número de 1 byte.
- FPGA: Interpretación de estos valores para generar un pulso PWM que sea capaz de mover un servo hacia el ángulo de estabilización.
- FPGA: Una vez leídos los valores y generado el PWM, mandar un código a la placa con el ángulo al que me mueve el servo y su estado (Espera, Movimiento o Parado).
- STM32: Obtención de los valores de vuelta y presentación en una pantalla LCD comunicado por tecnología I2C.

2. Comunicación.

2.1 Comunicación I2C.

Para la implementar la comunicación I2C entre FPGA y microprocesador, se hizo uso del siguiente módulo IP de acceso libre a través de GitHub.

[Link al repositorio en GitHub del modulo I2C](#)

***Cabe destacar que no hay gran variedad de opciones disponibles en la web para implementar una comunicación I2C en VHDL siendo complicado encontrar una versión actualizada y totalmente funcional de forma gratuita.

De hecho, se tuvieron numerosos problemas para llegar implementar la comunicación serie, barajando diferentes posibilidades SPI e I2C. Estas adversidades se encuentran detalladas más adelante en el apartado 2.3.

2.1.1 Funcionalidad e Interfaz del Módulo I2C.

El módulo se implementa a través de una interfaz sencilla:

```
read_req      : out std_logic;
data_to_master : in  std_logic_vector(7 downto 0);
data_valid    : out std_logic;
data_from_master : out std_logic_vector(7 downto 0));
```

Cuando [read_req] está a nivel alto, el esclavo envía los datos [data_to_master] en el mismo ciclo de reloj, por lo que se deben tener los datos a enviar disponibles con anterioridad en el registro correspondiente.

Cuando [data_valid] está a nivel alto se registran los datos recibidos del master [data_from_master] en el mismo ciclo.

Por lo tanto, se completa la interfaz con un registro para trabajar con el flujo de datos:

```
signal data_reg : std_logic_vector(7 downto 0);
```

En último lugar, el esclavo ignorará todas las órdenes del maestro que no concuerden con la dirección [SLAVE_ADDR] genérico modificable en el .vhdl principal del módulo.

[SLAVE_ADDR] es una dirección de 7 bits en el formato **MSB** donde el primer bit es el más significativo en el lado del máster. Esto quiere decir que, por ejemplo, una dirección de 7 bits “0000011” que se corresponde con un 3 en el esclavo, es en realidad un 6 en el master “00000110”, puesto que el **LSB** nunca se llega a enviar.

El módulo cuenta, además, con una función anti-rebotes para optimizar el funcionamiento de las líneas SDA y SCL.

2.1.2 Simulaciones del Módulo I2C.

El módulo incluye los ficheros de test necesarios para simular el funcionamiento de la comunicación serie.

En la versión más moderna del módulo, se incluyen dos simulaciones.

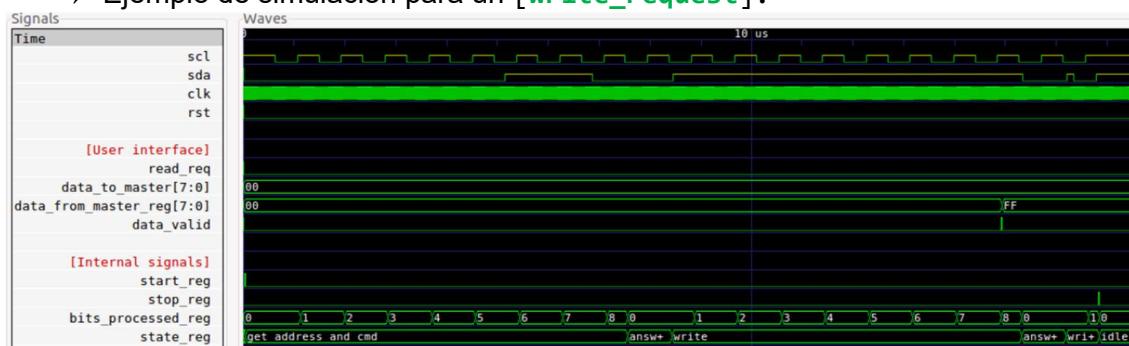
- Una primera simulación bajo condiciones ideales.
 - Una segunda simulación bajo línea SCL ruidosa.

Versión más reciente: [Link a la version mas reciente del modulo I2C](#)

Sin embargo, esta última versión no es totalmente funcional (explicación sección 2.3) por lo que se implementó un parche anterior en la que únicamente se incluye la simulación bajo condiciones iniciales.

Versión utilizada: [Link al repositorio en GitHub del modulo I2C](#)

→ Ejemplo de simulación para un [write_request].

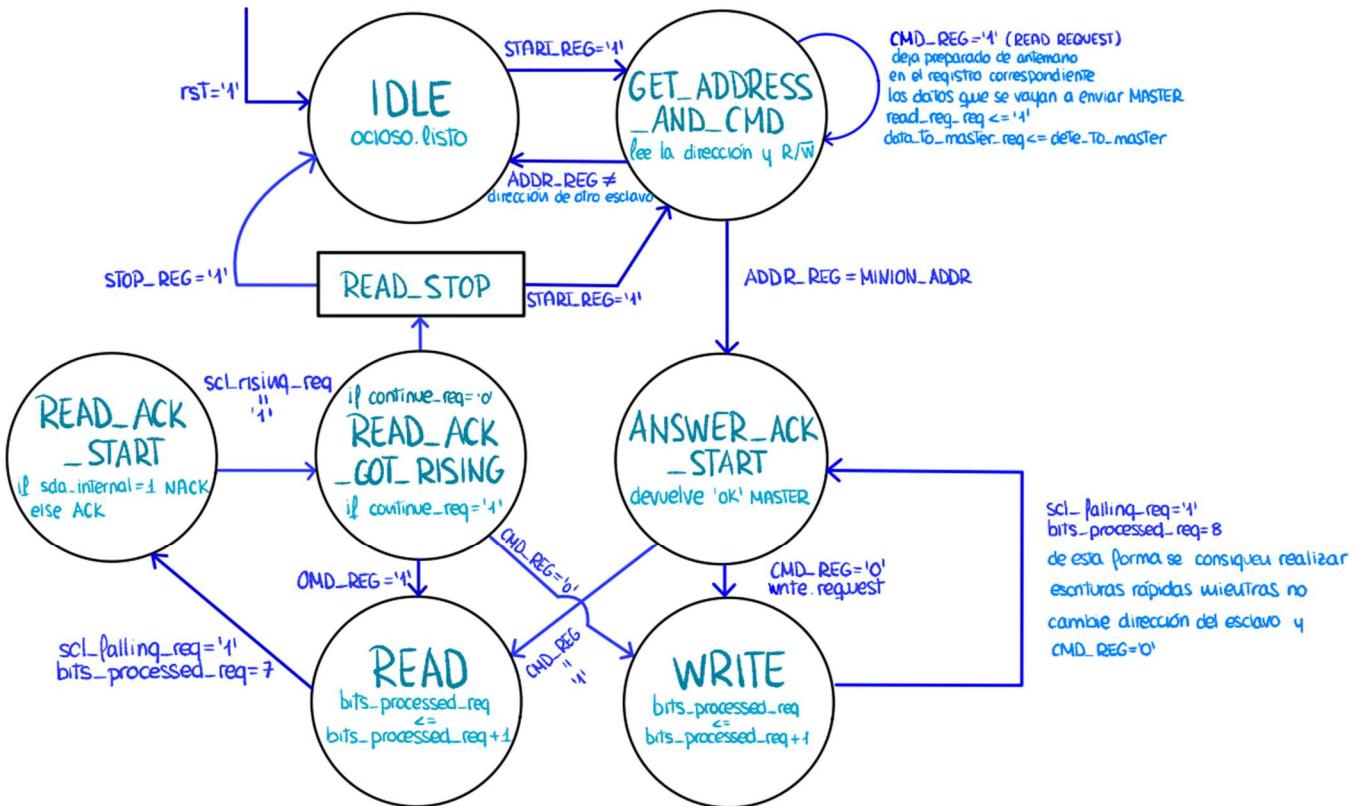


→ Ejemplo de simulación para 3 [write_request] seguido de un [read request].



Las simulaciones se encuentran disponibles tanto en los archivos de la memoria, como en el sitio web del módulo por si se desea observar los resultados con mayor detalle.

2.1.3 Máquina de estados del módulo I2C.



2.1. 4 Testeo sencillo en ARDUINO + Saleae Logic8

Para poder comprobar de manera sencilla y rápida el correcto funcionamiento de la comunicación, se encontró un tutorial en la web que fue de gran utilidad al tratarse de un experimento fácilmente replicable.

Todo el código en Arduino y fichero top para el módulo se encuentran en:

[Link al tutorial I2C ARDUINO-FPGA](#)

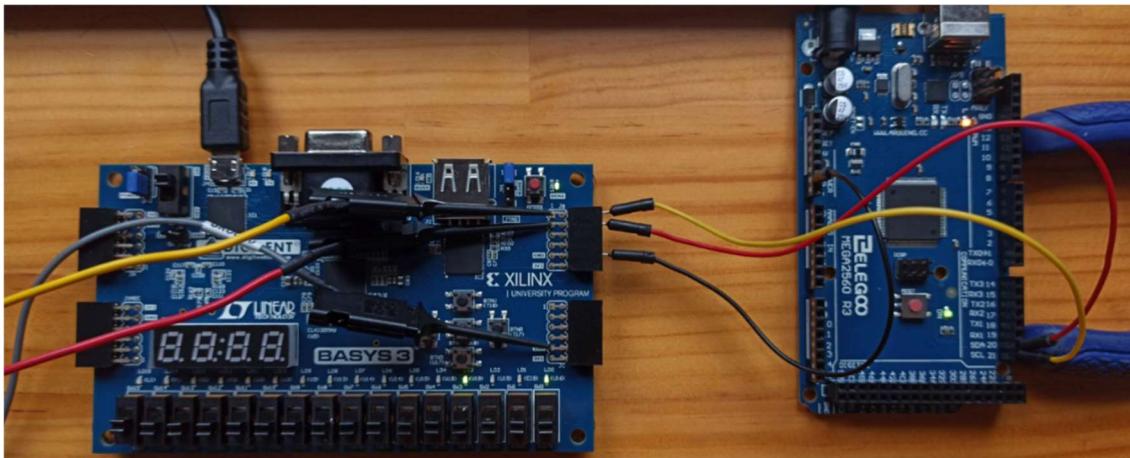
En el tutorial se propone conectar un Arduino, que actuará como maestro, a una FPGA, que actuará como esclava, usando el mismo módulo I2C que se seleccionó para el trabajo.

- Los componentes empleados para realizar el siguiente test, son:



De izquierda a derecha: FPGA Basys3, Saleae8logic, Microprocesador ARDUINO o equivalente.

Se requieren únicamente tres cables para el conexionado:



La idea es sencilla. Se va a enviar un valor del Arduino a la FPGA y esta va a devolver ese mismo valor + 5. Además, se encenderán los leds de la FPGA con el valor recibido del Arduino.

```
COM7
```

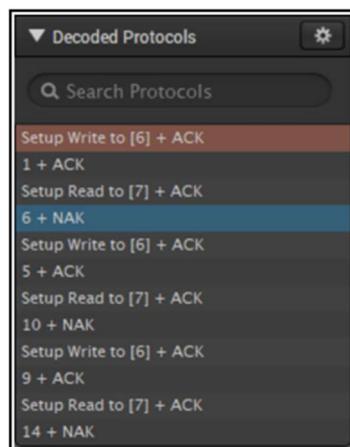
```
Enviando a la FPGA 1
Recibido de la FPGA 6
Enviando a la FPGA 5
Recibido de la FPGA 10
Enviando a la FPGA 9
Recibido de la FPGA 14
```

```
data_to_master <= data_reg + 5;

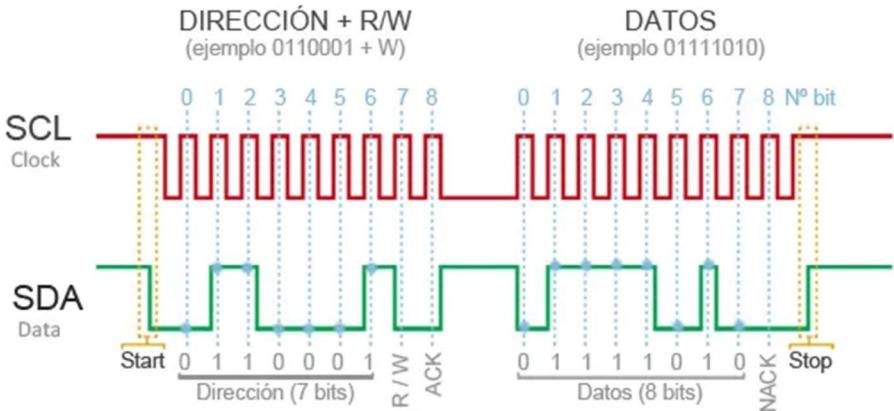
process(clk)
begin
  if(clk'event and clk='1') then
    if(data_valid='1') then
      data_reg <= data_from_master;
    end if;
  end if;
end process;

end architecture;
```

Se puede observar el resultado exitoso de la comunicación, tanto desde el COM7 en Arduino, como desde la interfaz que nos proporciona el Saleae8logic:



Donde, además se pueden observar los valores de SCL y SDA en tiempo de simulación. Además, el software es capaz de identificar con precisión en todo momento en que punto se encuentra la comunicación en base al protocolo estándar I2C:



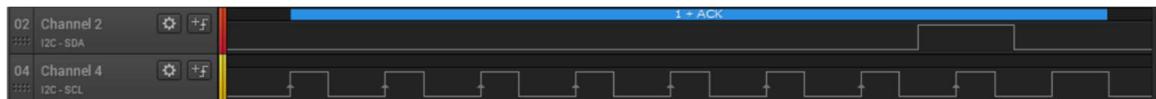
De esta forma:

Por ejemplo, cuando se envía un “1” desde el COM7 en Arduino:

- Se ordena a [SLAVE_ADDR] un `write_request` “0000 011” & “0”



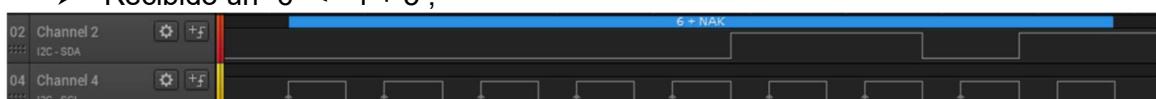
- Enviando un “1”



- Se ordena a [SLAVE_ADDR] un `read_request` “0000 011” & “1”

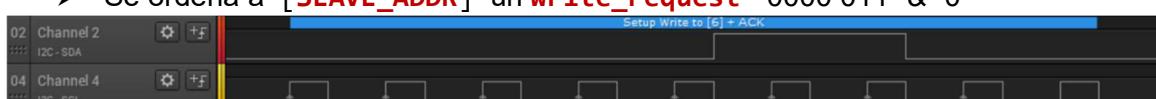


- Recibido un “6” <= 1 + 5 ;

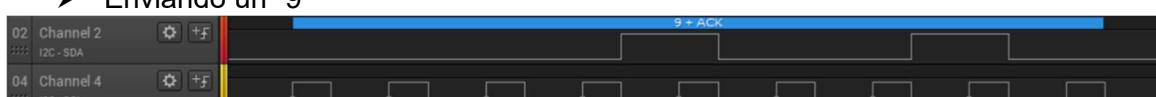


O, por otro lado, cuando se envía un “9” desde el COM7 en Arduino:

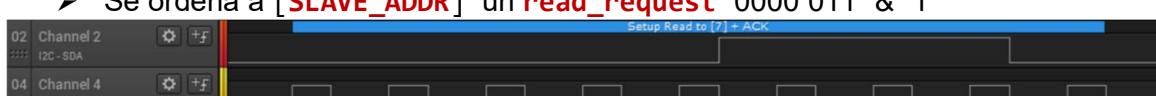
- Se ordena a [SLAVE_ADDR] un `write_request` “0000 011” & “0”



- Enviando un “9”



- Se ordena a [SLAVE_ADDR] un `read_request` “0000 011” & “1”



- Recibido un “14” <= 9 + 5 ;



Cómo se puede comprobar en las figuras, se satisfacen en todo momento las especificaciones deseadas para la comunicación entre FPGA y microprocesador.

2.1.5 Testeo sencillo con microprocesador STM32F407

Ahora que ya se ha comprobado de una forma clara y visual el correcto funcionamiento del módulo, se procede a realizar un test con el microprocesador STM32 usado para el desarrollo del trabajo.

- Los componentes empleados para realizar el siguiente test son:



De izquierda a derecha: FPGA Basys3, Saleae8logic, microprocesador STM32F407.

Se quiere comprobar el funcionamiento de la comunicación entre ambos dispositivos. La idea para el experimento es la siguiente:

- Se va solicitar un **write_request** al esclavo, seguido de un **read_request**, con la peculiaridad de que se envían y registran los datos en el mismo buffer.
- Este buffer [**i2cRxBuf**] está inicializado a X“00”.
- El código de la FPGA en VHDL es el mismo del test anterior, en el que se devuelve al maestro el buffer que recibe + 5.
- Los leds de la FPGA están asociados al registro de almacenamiento de datos, por lo que, en cada iteración entre las placas, se irán iluminando los leds correspondientes al estado actual del buffer [**i2cRxBuf**]. Esto es: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, etc.

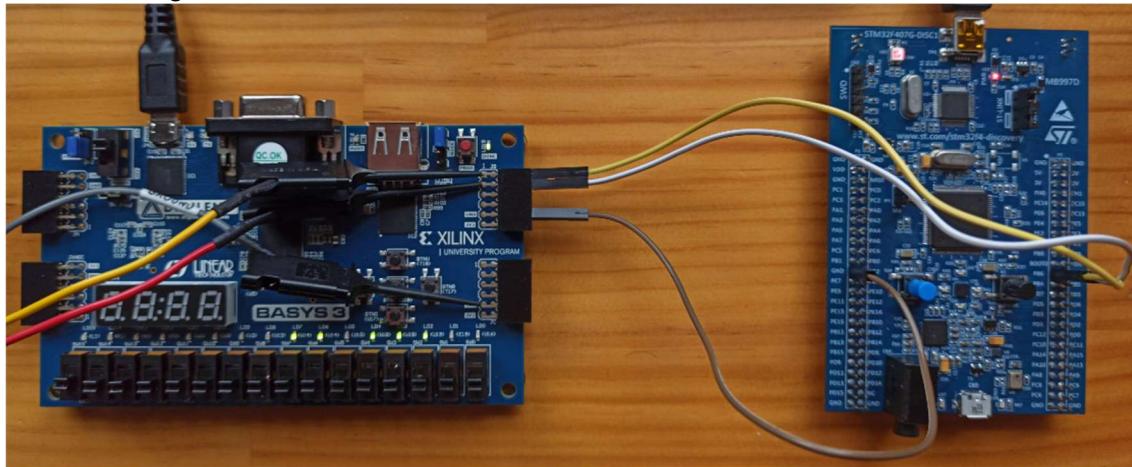
Código en la STM32:

```
if (HAL_I2C_Master_Transmit(&hi2c1, slave_addr_write, &i2cRxBuf, 1, HAL_MAX_DELAY) != HAL_OK){  
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, 1); //si se enciende el led D12, error de transmisión  
}  
  
if (HAL_I2C_Master_Receive(&hi2c1, slave_addr_read, &i2cRxBuf, 1, HAL_MAX_DELAY) != HAL_OK){  
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_14, 1); //si se enciende el led D14, error de recepción  
}
```

El funcionamiento en tiempo real se encuentra disponible en los archivos de la memoria.

Archivo correspondiente: [funcionamientoI2CSTM.mp4](#)

El montaje de nuevo solo requiere 3 cables + captación de las señales SDA y SCL con el Saleae8logic:



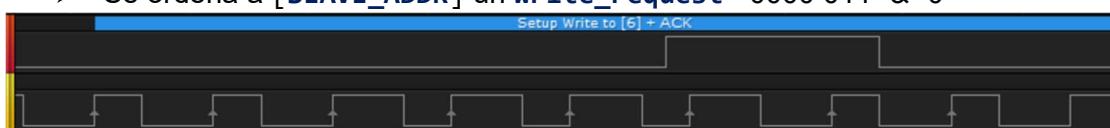
Se muestra de nuevo el código de la FPGA en VHDL + captación de la comunicación desde la interfaz del software del Saleae8logic en un intervalo finito de tiempo.

<pre> data_to_master <= data_reg + 5; process(clk) begin if(clk'event and clk='1') then if(data_valid='1') then data_reg <= data_from_master; end if; end if; end process; end architecture; </pre>	<p>Setup Write to [6] + ACK 25 + ACK Setup Read to [7] + ACK 30 + NAK Setup Write to [6] + ACK 30 + ACK Setup Read to [7] + ACK 35 + NAK Setup Write to [6] + ACK 35 + ACK Setup Read to [7] + ACK 40 + NAK</p>
---	---

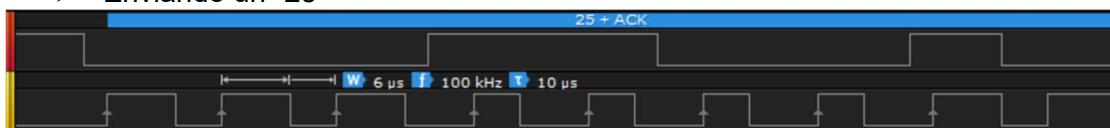
De esta forma:

Por ejemplo, cuando se envía un “25” desde la STM32:

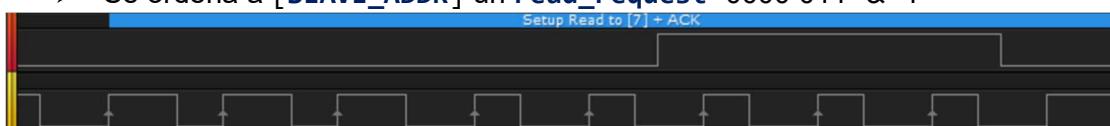
- Se ordena a [SLAVE_ADDR] un **write_request** “0000 011” & “0”



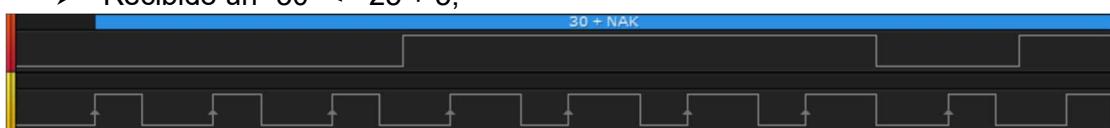
- Enviando un “25”



- Se ordena a [SLAVE_ADDR] un **read_request** “0000 011” & “1”



- Recibido un “30” <= 25 + 5;



Donde se puede observar de nuevo que se satisfacen en todo momento las especificaciones deseadas para la comunicación entre FPGA y microprocesador.

Una vez llegados a este punto, se puede afirmar que el módulo I2C elegido funciona con eficacia y robustez. Sin embargo, esto no se consiguió con facilidad. En el siguiente apartado se explica en detalle los diferentes obstáculos en el desarrollo de la comunicación, así como diversas aproximaciones al problema.

2.2. Errores en el desarrollo.

2.2.1. I2C.

Durante la implementación del I2C se tuvieron serios problemas a la hora de conseguir una comunicación bidireccional.

En un principio se consiguió el envío de datos hacia la FPGA, pero no la vuelta. Tras una ardua labor de investigación se llegó a entender que un reciente “**bugfix**” en el código del desarrollador provocaba un cortocircuito en la señal SDA a tierra, provocando que siempre se mandara “0” aunque especificáramos lo contrario.

Esto provocaba que, si el Maestro solicitaba información al Esclavo, este último no era capaz de poner la línea SDA “1” independientemente del valor almacenado en su registro de envío de datos.

Se pudo detectar este error gracias al analizador de ondas **Saleae Logic8** en combinación con el sencillo montaje en **ARDUINO** detallado anteriormente.

Versión más reciente: [Link a la version mas reciente del modulo I2C](#)

Versión utilizada: [Link al repositorio en GitHub del modulo I2C](#)

Para solucionar el problema, se encontró una versión anterior del módulo IP en el que aún no se había implementado el antes mencionado “**bugfix**”. Se implementó esta anterior versión y la comunicación funcionaba de manera rápida y eficaz.

Se tardó mucho tiempo en detectar este error puesto que se supuso que el error residía en la implementación de la interfaz del módulo y no en código a bajo nivel.

2.2.2. Comunicación SPI.

Mientras se trataba de averiguar el problema del I2C se propuso elaborar un código de comunicación SPI como alternativa a la comunicación.

Se buscaron módulos IP para esto, pero ninguno que fuera gratuito funcionaba bien o estaban desarrollados en otros lenguajes, por lo que si surgía algún problema no se podría depurar.

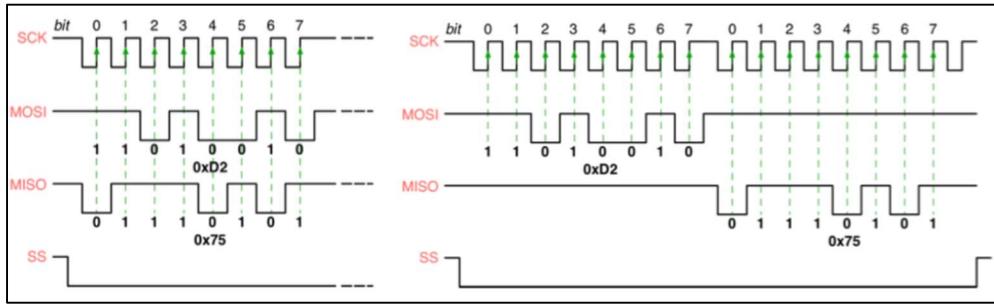
Por tanto, se decidió elaborar dos códigos completos mediante máquinas de estado:

- SPI Full-Duplex.
- SPI Half-Duplex.

Aunque no se llegarán a introducir finalmente, debido a que la STM32 no era capaz de sincronizarse completamente con el código, el código merece la pena explicarlo.

La comunicación SPI se basa en el envío por uno de los cables de los datos (MOSI) y la recepción por otro (MISO), todo ello sincronizado a la frecuencia de reloj de la

comunicación. Dependiendo si el envío y recibo de datos se hace en el mismo instante o tenemos primero una fase envío y otra de recepción se llamarán respectivamente Full-Duplex o Half-Duplex. El esquema de la comunicación es el siguiente:



2.2.2.1. SPI Full-Duplex.

Para el desarrollo de ambos códigos se utilizó una máquina de estados de Mealy. La entidad de dicho componente es la siguiente:

```

8 entity SPI is
9 generic(
10   WIDTH:positive:=8 --No cambiabile
11 );
12 port(
13   CE: in std_logic; --Communication Enable
14   RESET: in std_logic;
15   CLK: in std_logic; --clk fpga 100mhz
16   data_to_send:in std_logic_vector(WIDTH-1 downto 0);
17   MISO: in std_logic; --Master Input, Slave Output
18
19   MOSI: out std_logic; --Master Output, Slave Input
20   data_received:out std_logic_vector(WIDTH-1 downto 0);
21   SS: out std_logic; --Slave Select
22   CLK_OUT: out std_logic --SCK
23 );
24 end entity;

```

Donde los únicos que no se corresponden con el esquema son:

- CE: “Communication Enable” iniciando este bit a ‘1’ comienza la comunicación.
- RESET: Reinicia el estado de la comunicación pasando al estado de espera.
- Data_to_send: se trata de un vector de 8 bits que se rellenaría con los datos a mandar a la STM32.
- Data_received: Datos recibidos por la STM32.

La máquina de estados tendrá 4 estados que va recorriendo de forma secuencial:

- Espera: El módulo SPI se mantiene esperando a que el bit de CE se ponga a ‘1’. En el caso de que este se ponga a 1 pasaremos al estado de “Encendiendo”.

```

elsif rising_edge(CLK) then
  if state=ESPERA then
    SS<='1';
    MOSI<='1';
    if CE='1' then
      nxt_state<=ENCENDIENDO;
    end if;
  end if;

```

- Encendiendo: Se modifican los pines de salida del Master para preparar todo e iniciar la comunicación. El registro “data” se sobrescribe con lo que le hayamos introducido a la entrada del módulo para mandar. Acto seguido se pasa al estado “Comunicando” para comenzar la transferencia de datos.

```
elsif state=ENCENDIENDO then
    SS<='0';
    data<=data_to_send;
    nxt_state<=COMUNICANDO;
    bit_sent<=(others=>'0');
```

- Comunicando: En este estado es donde verdaderamente ocurre la transferencia de datos. Cuando detectemos que el SCK está a uno y que se produce un flanco mandaremos el LSB que tengamos almacenado en data por el MOSI y llenaremos el buffer “data_read” con los valores que nos vayan llegando por el MISO. Acto seguido se aumentará en uno el número de bits leídos y se girará el vector data para que el nuevo LSB sea el siguiente por mandar. Cuando los bits madados y leídos (que corresponderán a los mismo por ser full-duplex lleguen a 1 byte pasaremos al estado de “Apagando”.

```
elsif state=COMUNICANDO then
    if SCK='1' and flanco_SCK=1 then
        flanco_SCK:=0;
        if bit_sent<8 then
            MOSI<=data(0);
            data_read(to_integer(bit_sent))<=MISO;
            end if;
            data<=data(0) & data(WIDTH-1 downto 1);
            bit_sent<=bit_sent+1;
            if bit_sent=8 then
                nxt_state<=APAGANDO;
                data<=data(0) & data(WIDTH-1 downto 1);
            end if;
        end if;
```

- Apagando: Finalizamos la comunicación volviendo al estado de “Esperando” subiendo el SS y el MOSI de manera que el esclavo entienda que tiene que dejar de leer.

```
elsif state=APAGANDO then
    SS<='1';
    MOSI<='1';
    bit_sent<=(others=>'0');
    nxt_state<=ESPERA;
end if;
end process;
```

Aunque no pertenezca al comportamiento íntegramente definido por la máquina de estados, los códigos de RESET y SCK son los siguientes:

```

26 architecture behavioral of SPI is
27   signal data:std_logic_vector(WIDTH-1 downto 0); --170
28   signal data_read:std_logic_vector(WIDTH-1 downto 0); --datos que leemos
29   type state_t is (ESPERA, ENCENDIENDO ,COMUNICANDO, APAGANDO);
30   signal state:state_t:=ESPERA;
31   signal nxt_state:state_t:=ESPERA;
32   signal SCK:std_logic; --registro de reloj de salida
33   signal bit_sent:unsigned(3 downto 0); --contador de los bits enviados
34 begin
35   clk_gen:process(RESET,CLK)
36     --Proceso que únicamente se dedica a generar el SCK mientras se comunica
37     subtype natural_t is natural range 0 to 120;
38     variable counter:natural_t:=0;
39 begin
40   if RESET='1' then
41     SCK<='1';
42     counter:=9;
43   elsif rising_edge(CLK) and state=COMUNICANDO then
44     counter:=counter+1;
45     if counter=100_000_000 then --Para bajar o subir la frecuencia variar
46       counter:=0;
47       case SCK is
48         when '0'=> SCK<='1';
49         when others=> SCK<='0';
50       end case;
51     end if;
52   end if;
53 end process clk_gen;

```

Dicho proceso contiene un contador el cual permuta el valor del SCK entre 0 y 1 cuando se llega a un valor definido, en este caso 100.000 ciclos de reloj de la FPGA. Al variar este valor, modificamos la frecuencia del SCK, en este caso al utilizar un reloj de la FPGA que funciona a 100 MHz, nuestro SCK tendría una frecuencia de 1KHz.

El testbench realizado queda de la siguiente forma:

```

entity SPI_tb is
end entity;

architecture test of SPI_tb is
  constant size:positive:=8;
  --input
  signal CE,RESET: std_logic;
  signal data_to_send:std_logic_vector(size-1 downto 0);
  --output
  signal MOSI,MISO,SS,CLK_OUT:std_logic;
  signal data_received:std_logic_vector(size-1 downto 0);

  component SPI is
    generic(
      WIDTH:positive:=8 --No cambiable
    );
    port(
      CE: in std_logic; --Communication Enable
      RESET: in std_logic;
      CLK: in std_logic; --clk fpga 100mhz
      CLK_OUT: out std_logic; --SCK
      data_to_send:in std_logic_vector(WIDTH-1 downto 0);
      data_received:out std_logic_vector(WIDTH-1 downto 0);
      MOSI: out std_logic; --Master Output, Slave Input
      MISO: in std_logic; --Master Input, Slave Output
      SS: out std_logic --Slave Select
    );
  end component;
  constant CLK_PERIOD:time:=1 sec/100_000_000;
  signal data:std_logic_vector(size-1 downto 0):="11100111";
begin
  uut: SPI
  port map(
    CE=>CE,
    RESET=>RESET,
    CLK=>CLK,
    CLK_OUT=>CLK_OUT,
    data_to_send=>data_to_send,
    data_received=>data_received,
    MOSI=>MOSI,
    MISO=>MISO,
    SS=>SS
  );
  process
  begin
    CLK<='1';
    wait for 0.5*CLK_PERIOD;
    CLK<='0';
    wait for 0.5*CLK_PERIOD;
  end process;

  mis:process
  variable i:integer:=0;
  begin
    wait until CLK_OUT='0';
    wait for 5*CLK_PERIOD;
    MISO<=data(0);
    data<=data(0)&data(size-1 downto 1);
    wait until CLK_OUT='1';
  end process;

```

```

testing:process
begin
    data_to_send<="10101010";
    RESET<='1';
    wait for 1.5*CLK_PERIOD;
    RESET<='0';
    wait for 10*CLK_PERIOD;
    CE<='1';
    wait until CLK='1';
    CE<'0';
    for i in 0 to 250 loop
    wait until CLK='1';
    end loop;

    data_to_send<="11101011";
    RESET<='1';
    wait for 1.5*CLK_PERIOD;
    RESET<='0';
    wait for 10*CLK_PERIOD;
    CE<='1';
    wait until CLK='1';
    CE<'0';
    for i in 0 to 250 loop
    wait until CLK='1';
    end loop;

    assert false
    report "[SUCESS] Simulation finished"
    severity failure;
end process;
end architecture;

```

Tras realizar la simulación obtenemos las siguientes formas de onda, se ha reducido el contador del SCK a 10 para que pueda verse en la simulación:



Como puede observarse en la imagen superior en el momento que CE se pone a nivel alto comienzan a variar las señales, puede verse que al principio en data_to_send estamos introduciendo "10101010" que se corresponde con lo que sale por el MOSI.

El circuito es redispersable sin afectar al valor que tuvieran antes los registros de data_read y data_to_send.

2.2.2.2. SPI Half-Duplex.

Como adición al módulo anterior se decidió desarrollar otro Half-Duplex por si la STM no era capaz de leer a la vez los datos de entrada y salida sin desincronizarse demasiado.

Este módulo es igual al anterior salvo 3 estados más:

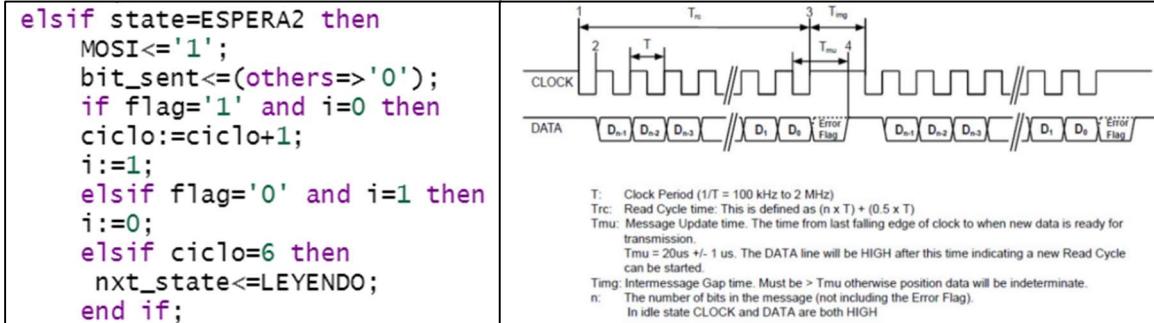
- Escribiendo: Este estado se habilita después de “Encendiendo” sería similar al “Comunicando” del Full-Duplex pero sin contar con la lectura del MISO.

```

elsif state=ESCRIBIENDO then
    if SCK='1' and flanco_SCK=1 then
        flanco_SCK:=0;
        if bit_sent<8 then
            MOSI<=data(0);
            data<=data(0) & data(WIDTH-1 downto 1);
        end if;
        bit_sent<=bit_sent+1;
        if bit_sent=8 then
            nxt_state<=ESPERA2;
            data<=data(0) & data(WIDTH-1 downto 1);
        end if;
    end if;

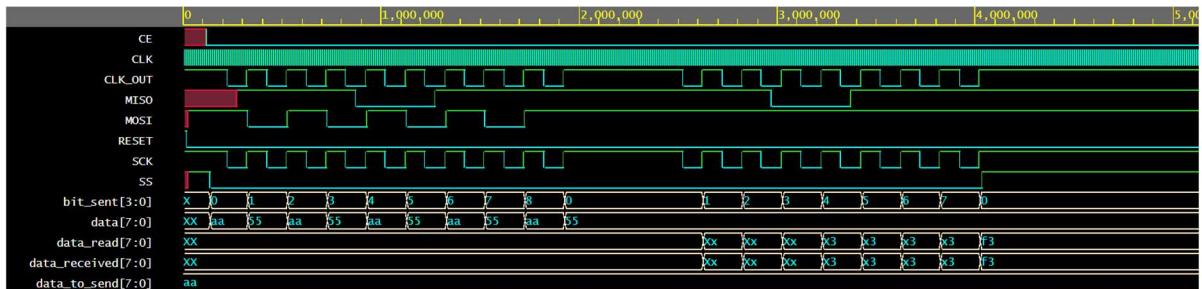
```

- ESPERA2: Correspondiente a la fase de transición entre la escritura y la lectura del dispositivo. Según la documentación oficial de STM32, librería HAL y documentación sobre comunicaciones SPI, todos concuerdan en que el tiempo entre la escritura y la lectura debe de ser al menos de 4 veces el periodo del reloj SCK, por lo que para asegurar este tiempo se ha decidido aumentarlo a 6.

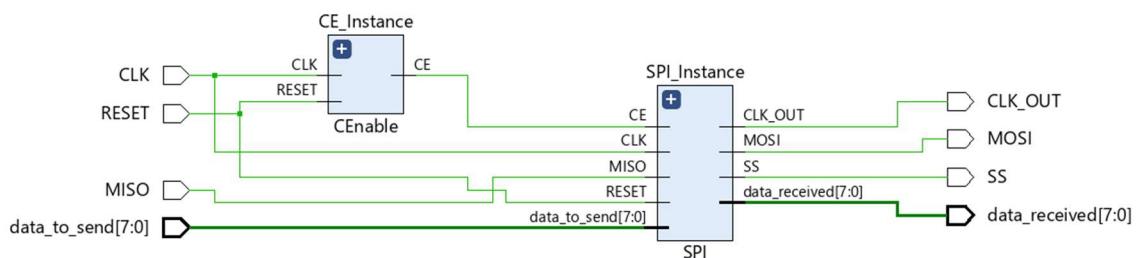


- Leyendo: Fase de lectura donde se rellena el buffer data_read.

Tras su simulación con el test-bench obtenemos las siguientes formas de onda:



Se creó otra entidad que generaba un pulso en CE cada cierto tiempo, de manera que pudieramos ver la comunicación de manera periódica con el Salae en el laboratorio. El esquemático RTL queda finalmente de la manera siguiente:



Cabe repetir que todo el código realizado para la comunicación SPI ha sido elaborado por nosotros sin ningún tipo de ayuda salvo la documentación oficial.

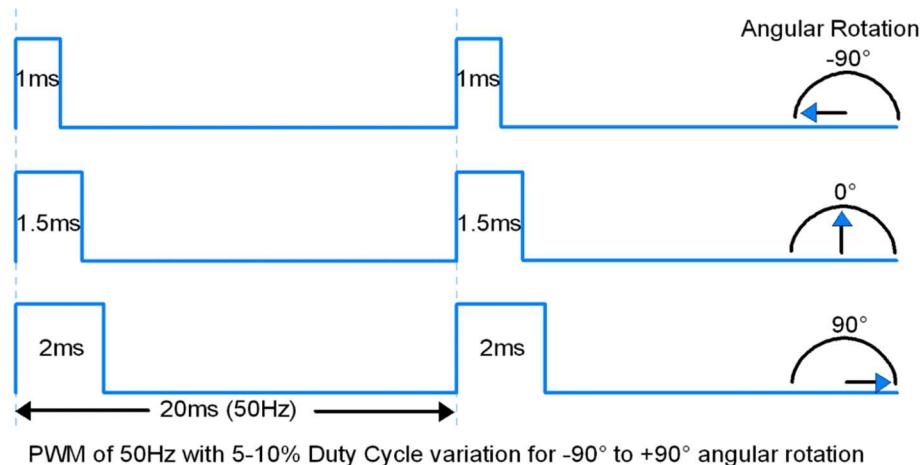
3. Control PWM del servo implementado en VHDL.

3.1. Marco teórico

El objetivo es controlar el servo a través de un pin I/O de la FPGA.

Para ello, se precisa obtener una señal con periodo de 20-25 ms y un ancho de pulso que varíe entre 0.5 ms y 2.5 ms dependiendo de la hoja de características del servo.

En el caso del servomotor SG90 se debe generar una señal PWM del tipo:



PWM of 50Hz with 5-10% Duty Cycle variation for -90° to +90° angular rotation

La idea para poder generar dicha señal mediante lenguaje de descripción hardware es la siguiente:

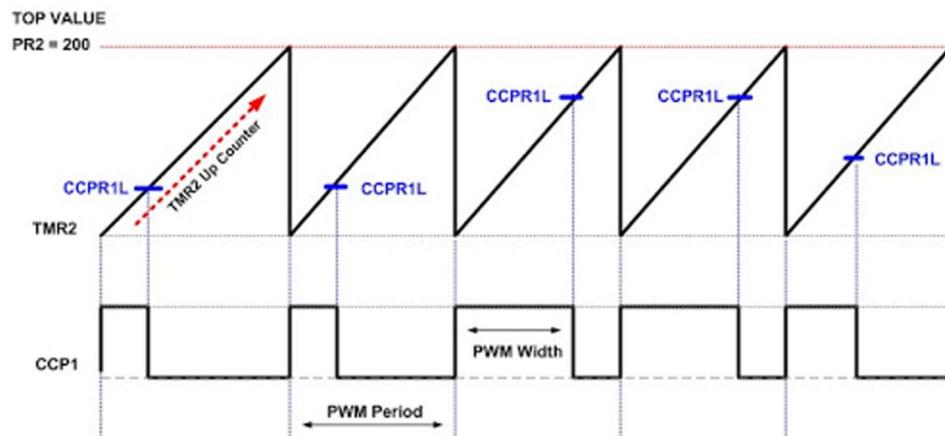
- Se tiene un contador a una frecuencia tal, que transcurre un periodo de 20ms desde su valor inicial hasta alcanzar su valor máximo (**TOP_VALUE**)
- Cuando alcanza **TOP_VALUE** el contador se resetea.
- Se tiene, además, un comparador en el cual se compara constantemente el valor del contador (que se incrementa en uno a flanco de reloj) con **CCPR1L**.

Donde **CCPR1L** tendrá un valor comprendido entre 0 y el valor máximo del contador.

Teniendo claro lo anterior, se puede generar la señal de salida (CCP1) cómo:

- CCP1=("1") cuando el valor del contador sea menor que **CCPR1L**.
- CCP1=("0") cuando el valor del contador sea mayor que **CCPR1L**.

Como se puede observar en la figura:



3.2. Caracterización del servo SG90

En una primera aproximación, se realizaron los cálculos del apartado 3.3 y su consecuente implementación en VHDL para un ancho de impulso comprendido entre 1ms y 2ms y un periodo de 20 ms cómo se indicaba en el **datasheet**.

Sin embargo, una vez implementado se pudo observar que el rango máximo del servo apenas era de 90°.

Llegados a este punto se encontró un datasheet más fiable disponible a través del siguiente link:

<https://robokits.co.in/motors/micro-servo-9g-with-accessories-sg90-premium>

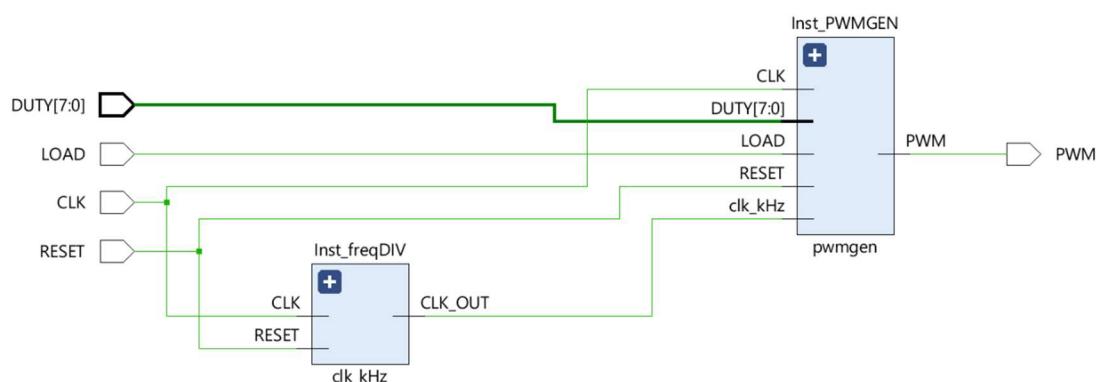
Datasheet que concordaba más con los resultados empíricos que se estaban obteniendo al implementar el control del servo.

Una vez comprendido esta pequeña complicación, se pasa a explicar el procedimiento para el diseño del control en VHDL.

3.3. Diseño del control PWM en Lenguaje Descripción Hardware. VHDL.

El diseño consta de dos módulos.

- Divisor de frecuencia (**Inst_freq_DIV**)
 - Cuya salida dictará el reloj del módulo PWM_GEN
 - Contador-Comparador (**PWM_GEN**)
 - Módulo encargado de generar la señal de control PWM de salida
- Además, cuenta con las siguientes funcionalidades:
- RESET -- reset asíncrono con prioridad.
 - LOAD -- carga síncrona del ciclo de trabajo **DUTY [7:0]**



3.3.1. Cálculos teóricos e Implementación.

Una vez caracterizado el servo con el que se está trabajando, se decidió implementar una señal cuyo ancho de impulso estuviera contenido entre 0.55 ms y 2.45 ms lo que permitía un rango de uso comprendido entre 0° y 150° aproximadamente.

3.3.1.1. Divisor de frecuencia.

En primer lugar, es necesario implementar un divisor de frecuencia puesto que el reloj interno de la FPGA es de 100Mhz. La frecuencia necesaria de este divisor para una resolución de 8 bits y un rango de operación de 1.9 ms = $t_{max} - t_{min}$ es la siguiente:

$$F_{necesaria} = \left(\frac{rango}{resolución} \right)^{-1} = 134.737 \text{ kHz}$$

$$\frac{100\text{Mhz}}{F_{necesaria}} = n_{iteraciones}^{\circ} \text{ del contador} = 742 \text{ iteraciones}$$

Dado que dicha señal de reloj debe tener un ciclo útil del 50%

$$n_{iteraciones}^{\circ} = \frac{742}{2} = 371 \text{ iteraciones}$$

empíricamente: $n = 373$ iteraciones.

Con esto se consigue una señal de reloj de aproximadamente 133.69kHz.

Ya se tiene todo lo necesario para implementar en VHDL el divisor de frecuencia.

```
cntr : process(RESET, CLK)
begin
    if RESET = '1' then
        pulse <= '0';
        cnt   <= (others => '0');
    elsif rising_edge(CLK) then
        -- Toggle signal pulse each 373 clock cycles
        -- Restart the counter
        if(cnt = 373) then
            pulse <= NOT(pulse);
            cnt   <= (others => '0');
        else
            cnt <= cnt + 1;
        end if;
    end if;
end process;
CLK_OUT <= pulse;
end architecture;
```

No es más complicado que un contador hasta 373 (mejores resultados empíricos) realizando un “**Toggle**” a la señal de salida cada vez que se alcanza este valor.

3.3.1.2. Generador señal PWM.

Instanciado por tres componentes:

- Contador:
 - Se incrementa en uno a flanco de subida de **133.69 kHz**).
 - Registro del ciclo de trabajo:
 - Actualiza el nuevo valor de DUTY[7:0] con carga síncrona LOAD = “1”.
 - Comparador:
 - Compara el valor del contador con el valor de [**DUTY[7:0] + offset**].
 - Mientras el valor del contador sea menor -----> PWM_OUT = “1”.
 - Cualquier otro valor -----> PWM_OUT = “0”.
- Ajuste de offset

No olvidemos que el objetivo era conseguir un ancho de impulso comprendido entre 0.55 ms y 2.45 ms.

$$\frac{\text{offset}}{133.69 \text{ kHz}} = 0.55 \text{ ms}$$
$$\rightarrow \text{offset} = 74 \text{ iteraciones}$$

De esta forma nos aseguramos de que cuando DUTY [7:0] = X“00” el valor mínimo de ancho de pulso sea de 0.55 ms como se desea.

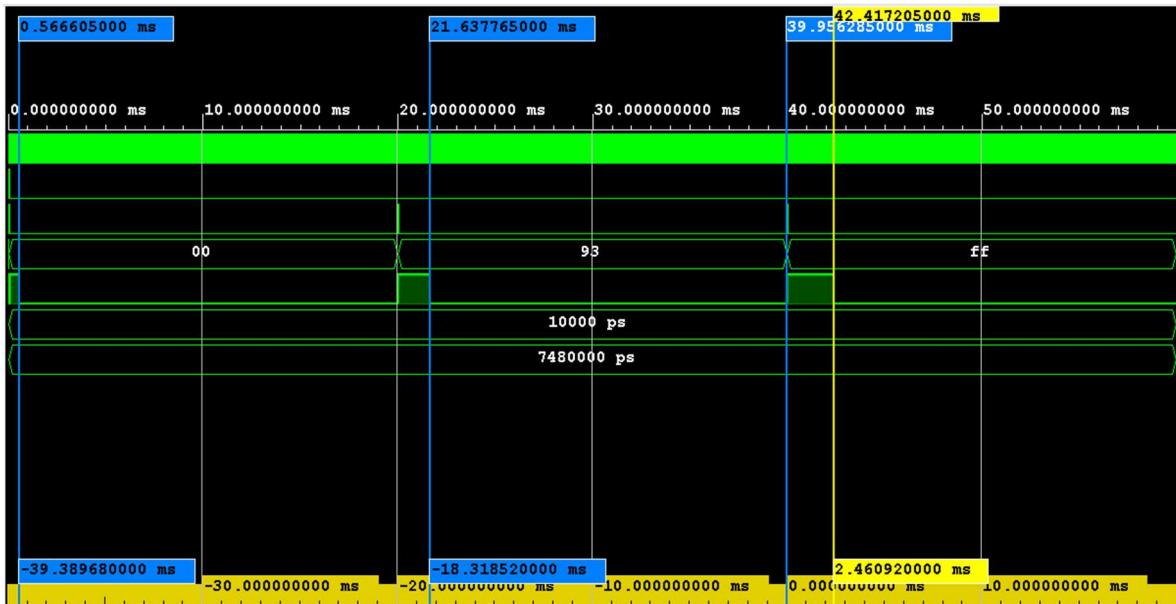
Finalmente, la implementación en VHDL:

```
-- PWM offset: 74/133.69kHz = 0.55ms
compare_value : cmp_i <= ("0" & duty_i) + 74;

-- OUTPUT PWM signal
COMPARATOR : PWM <= '1' when cntr_i < cmp_i else
              '0';
```

3.3.2. Simulaciones en VIVADO.

Donde se puede observar la modulación del ancho de pulso para valor mínimo, intermedio y máximo de DUTY[7:0] satisfaciendo en todo momento las especificaciones buscadas.



3.4 Esquema de conexión y Montaje

Será necesario para el montaje conectar un desplazador lógico de 3.3V a 5V a la señal de control de la FPGA puesto que la amplitud de la señal de control ronda los 3.3V cuando se requieren 5V de amplitud para controlar el servo.

Además, es necesario alimentar tanto el servo como el convertidor con 5V y GND para su funcionamiento. Para esto se usa una fuente de alimentación sencilla de protoboard.

[Enlace a la fuente de alimentacion](#)

[Enlace al convertidor logico](#)

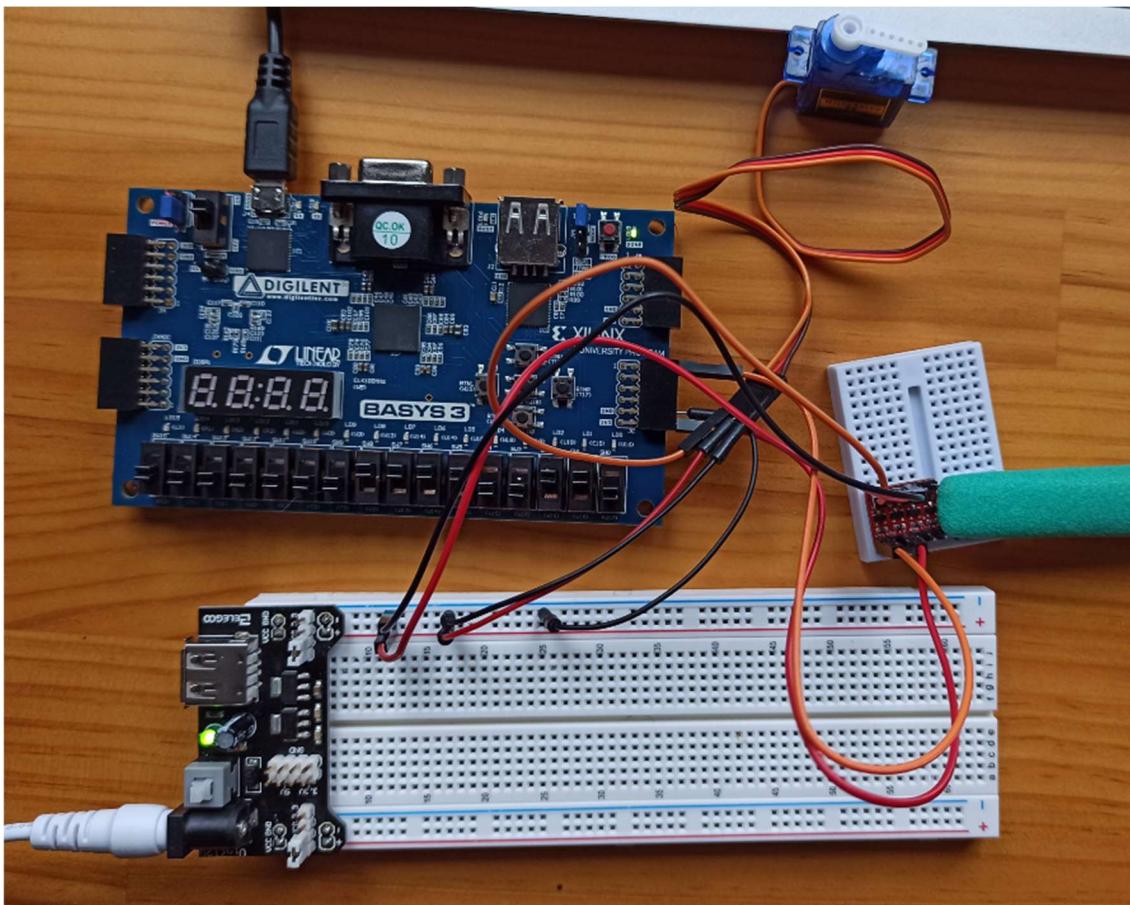
El utilizado para el montaje es el siguiente:

- Componentes:



De izquierda a derecha: FPGA Basys3, Convertidor lógico, Servo, Fuente alimentación ext. 5V

- Montaje:



Donde se puede observar el funcionamiento del sistema en el video adjunto en los archivos de la memoria: [funcionamientoPWM.mp4](#)

4. MPU6050.

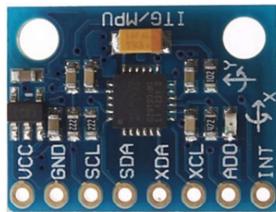
Como información útil que pasarle a la FPGA por I2C se ha decidido obtener el grado de inclinación de una MPU para que la FPGA mueva el servo acorde al ángulo que le esté llegando.

Este tipo de funcionalidad sería útil de en el caso de desarrollar un mecanismo de control de estabilidad donde una de las placas mide la inclinación de un objeto y el otro trata de equilibrarlo mediante la fuerza del servo.

El módulo que se ha usado es una MPU 6050, con buena resolución y muy usada ampliamente en la construcción de drones para sus sensores de localización. El dispositivo contiene en su interior:

- Acelerómetro: Nos permitirá obtener la aceleración angular e integrándolo podremos conocer su grado de inclinación.
- Giróscopo: Podremos obtener la velocidad angular que integrándolo nos dará información sobre la inclinación del dispositivo.
- Sensor de Temperatura: Aunque no se usara para el trabajo puede medirse la temperatura exterior.

El dispositivo se comunica gracias a comunicación I2C, la cual utilizaremos para configurar el dispositivo y posteriormente leer sus valores.



4.1 Configuración inicial del dispositivo.

4.1.1 WHO_AM_I

Lo primero será verificar que el dispositivo se encuentra conectado y se puede establecer una conexión fiable con el sensor. Para ello leeremos el registro WHO_AM_I que en el caso de estar conectado nos devolverá el valor 104.

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
75	117	-				WHO_AM_I[6:1]			-

The default value of the register is 0x68.

4.1.2 PWR_MGMT_1_REG

Tras haber verificado que estamos conectados, deberemos iniciar el sensor para que tome valores de forma continua, por defecto el sensor está en “Sleep Mode” para consumir poco, nosotros necesitamos que lea de forma continua.

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
6B	107	DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS			CLKSEL[2:0]

Este registro además no permite la opción de seleccionar el ciclo de reloj al que se encontrará trabajando el dispositivo.

CLKSEL	Clock Source
0	Internal 8MHz oscillator
1	PLL with X axis gyroscope reference
2	PLL with Y axis gyroscope reference
3	PLL with Z axis gyroscope reference
4	PLL with external 32.768kHz reference
5	PLL with external 19.2MHz reference
6	Reserved
7	Stops the clock and keeps the timing generator in reset

Para configurarlo correctamente introduciremos el byte 0x00, de esta manera despertaremos al sensor y lo configuraremos para una frecuencia de muestreo de 8MHz.

4.1.3 SMPLRT_DIV_REG

Una vez encendido el sensor y habilitada la frecuencia de trabajo de este deberemos seleccionar la frecuencia de actualización de las medidas a la salida.

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
19	25								SMPLRT_DIV[7:0]

Para ello escribiremos en el registro SMPLRT_DIV y dividiéndolo entre la frecuencia de trabajo obtendremos la de muestreo con la siguiente formula:

$$sample_rate = Gyroscope_output_rate / (1 + SMPLRT_DIV)$$

Introduciendo "7" obtendremos una frecuencia de muestreo de 1MHz.

4.1.4 ACCEL_CONFIG_REG y GYRO_CONFIG_REG

Por defecto, la MPU se configura para realizar test internos sobre el estado de los sensores, esto puede desactivarse fácilmente accediendo directamente a los registros y deshabilitando estas opciones.

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1C	28	XA_ST	YA_ST	ZA_ST	AFS_SEL[1:0]			-	

Register (Hex)	Register (Decimal)	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1B	27	XG_ST	YG_ST	ZG_ST	FS_SEL[1:0]	-	-	-	

Así mismo estos registros nos permiten configurar la sensibilidad y el rango de la medida de salida, a más sensibilidad o rango mayor tiempo de medición y conversión por lo que tardaremos más en comunicarnos, puesto que simplemente vamos a querer medir grados en la franja de 0-180° podremos el valor más bajo del rango.

AFS_SEL	Full Scale Range	LSB Sensitivity
0	±2g	16384 LSB/g
1	±4g	8192 LSB/g
2	±8g	4096 LSB/g
3	±16g	2048 LSB/g

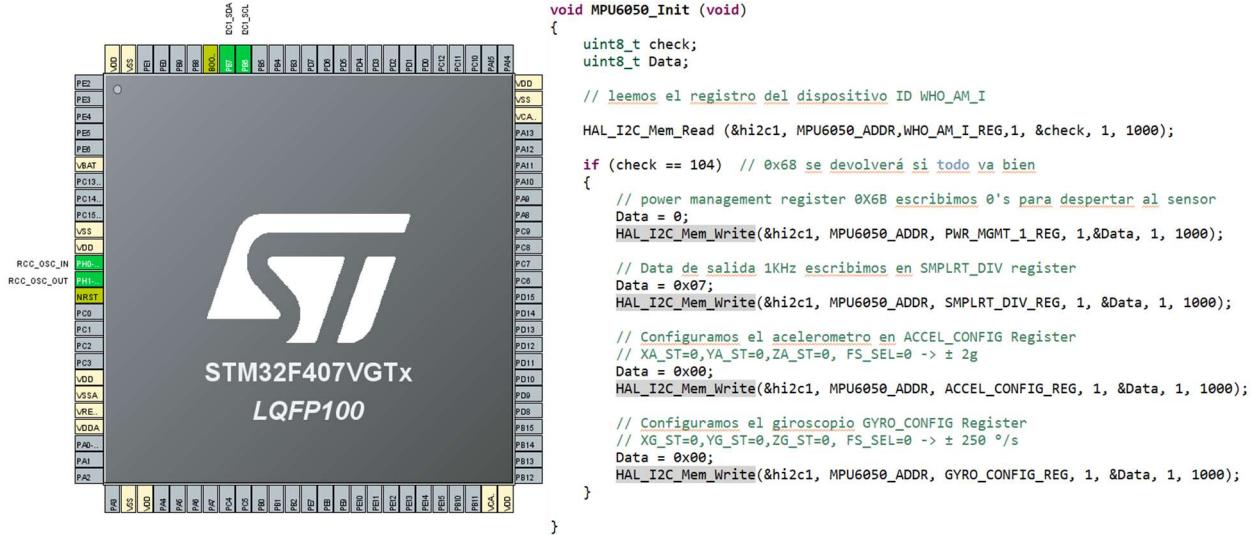
FS_SEL	Full Scale Range	LSB Sensitivity
0	± 250 °/s	131 LSB/°/s
1	± 500 °/s	65.5 LSB/°/s
2	± 1000 °/s	32.8 LSB/°/s
3	± 2000 °/s	16.4 LSB/°/s

En este caso el rango de la aceleración será de 4 veces el de la gravedad y el rango de giro será de 500 grados, más que suficiente para nuestra aplicación. En cuanto al rango de la sensibilidad cuando obtengamos el valor que mida el sensor deberemos dividirlo por estos valores para obtener la aceleración y los grados/seg respectivamente.

4.1.5 Implementación en STMCubeIDE.

En cuanto al CubeMx la configuración inicial será muy sencilla, habilitaremos un reloj externo RCC y un puerto I2C de comunicación. Se modificará la frecuencia de reloj para obtener un HCLK a 72MHz.

La configuración inicial de la MPU vendrá dada de la siguiente forma:



4.2 Lectura de aceleración. Aceleración angular.

Con la configuración inicial hecha, estamos preparados para comenzar a leer valores de la MPU, comenzaremos con la aceleración debido a que sus valores no harán falta para el cálculo del ángulo girado.

Cada eje de aceleración tendrá dos registros asociados a su medición HIGH y LOW, usualmente solo escogíramos el registro High puesto que representan una medida más estable y menos susceptible al ruido, pero puesto que tenemos la sensibilidad asociada al LSB recogeremos el valor en un entero de 16 bits.

La ventaja de estos es que al estar concatenados uno después de otro, si leyéramos 6 bytes podríamos tener en una lectura todos los valores de aceleraciones de los tres ejes, HIGH y LOW.

Addr (Hex)	Addr (Dec.)	Register Name	Serial I/F	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
3B	59	ACCEL_XOUT_H	R								ACCEL_XOUT[15:8]
3C	60	ACCEL_XOUT_L	R								ACCEL_XOUT[7:0]
3D	61	ACCEL_YOUT_H	R								ACCEL_YOUT[15:8]
3E	62	ACCEL_YOUT_L	R								ACCEL_YOUT[7:0]
3F	63	ACCEL_ZOUT_H	R								ACCEL_ZOUT[15:8]
40	64	ACCEL_ZOUT_L	R								ACCEL_ZOUT[7:0]

Por eso mismo comenzaremos desde la dirección 0x3B leyendo 3 bits.

Una vez guardado en un `int16_t` deberemos dividirlo entre la sensibilidad para obtener el valor de la aceleración angular.

4.2.1 Obtención del ángulo de inclinación.

Los acelerómetros actuales han pasado de contener una bola de metal que rebotaba en su interior a cristales piezoelectrinos que se deforman ante la acción de una fuerza, provocando corrientes y variaciones de una conductividad que de medirse obtendríamos la aceleración del dispositivo:

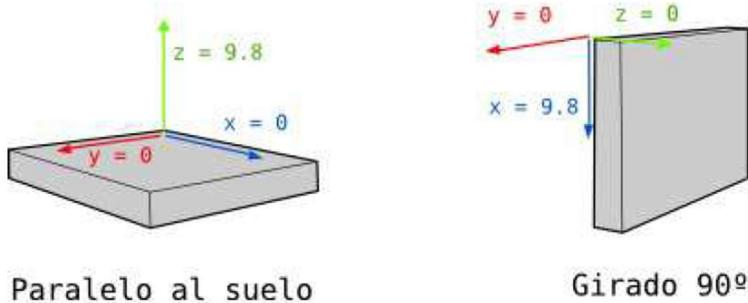


Figura: Representación de dos cristales piezoelectrinos sufriendo la acción de la gravedad.

Mediante trigonometría no es posible conocer el ángulo de inclinación del dispositivo:

$$\begin{aligned}\varnothing_x &= \tan^{-1}(y/\sqrt{x^2 + z^2}) \quad [\text{rad}] \\ \varnothing_y &= \tan^{-1}(x/\sqrt{y^2 + z^2}) \quad [\text{rad}]\end{aligned}$$

Donde:

- \varnothing_{xy} : Ángulo de inclinación del dispositivo.
- x, y, z : Aceleraciones angulares.

4.2.2 Implementación en código.

```
void MPU6050_Read_Accel (void)
{
    uint8_t Rec_Data[6];

    // Leemos 6 BYTES de datos comenzando por ACCEL_XOUT_H register

    HAL_I2C_Mem_Read (&hi2c1, MPU6050_ADDR, ACCEL_XOUT_H_REG, 1, Rec_Data, 6, 1000);
    Accel_X_RAW = (int16_t)(Rec_Data[0] << 8 | Rec_Data [1]);
    Accel_Y_RAW = (int16_t)(Rec_Data[2] << 8 | Rec_Data [3]);
    Accel_Z_RAW = (int16_t)(Rec_Data[4] << 8 | Rec_Data [5]);
    /*Convertimos los valores crudos o RAW en aceleración en 'g'
     * Tenemos que dividir de acuerdo a la escala seleccionada en FS_SEL
     * Como hemos configurado FS_SEL=0. Lo dividiremos entre 16384.0
    */
    Ax = Accel_X_RAW/16384.0;
    Ay = Accel_Y_RAW/16384.0;
    Az = Accel_Z_RAW/16384.0;

    Acc[1]=atan(-Ax/sqrt(pow(Ay,2)+pow(Az,2)))*RAD_A_DEG;
    Acc[0]=atan(-Ay/sqrt(pow(Ax,2)+pow(Az,2)))*RAD_A_DEG;
}
```

4.3 Lectura del giróscopo. Velocidad angular.

De forma análoga al acelerómetro mediremos la variación de grados partido del tiempo leyendo 6 bytes al mismo tiempo para juntarlos en enteros de 16 bits.

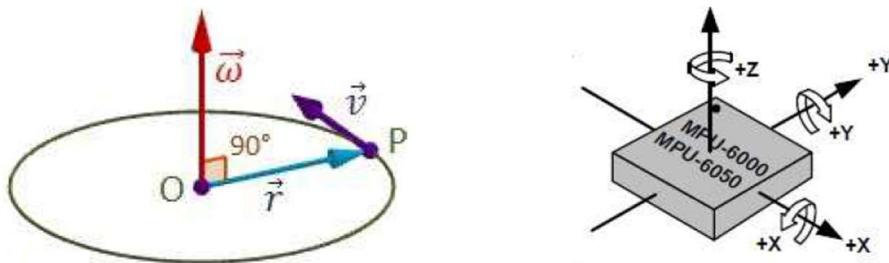
44	68	GYRO_XOUT_L	R	GYRO_XOUT[7:0]
45	69	GYRO_YOUT_H	R	GYRO_YOUT[15:8]
46	70	GYRO_YOUT_L	R	GYRO_YOUT[7:0]
47	71	GYRO_ZOUT_H	R	GYRO_ZOUT[15:8]
48	72	GYRO_ZOUT_L	R	GYRO_ZOUT[7:0]

Comenzando por la dirección 0x44 leeremos hasta abajo.

Una vez obtenidos los valores crudos de la velocidad angular, será necesario dividirlos por la sensibilidad para obtener el valor en dps(°/s) para nuestro caso dividiremos entre 131.

4.3.1 Obtención del giro de dispositivo.

Los giróscopos detectan la rotación del sensor, a través de las fuerzas centrifugas son capaces de hallar en dps(°/s) su velocidad angular, devolviendo un cero sino se encuentras sometidos a ninguna fuerza.



Integrando la medida que obtengamos del sensor con el tiempo somos capaces de detectar el ángulo de giro del sensor:

$$\theta_t = \int_{t_0}^t w(t) dw = \theta_{t_0} + w * \Delta t$$

Siendo:

- θ_t el ángulo de giro a calcular.
- θ_{t_0} el ángulo de giro de la muestra anterior.
- w la velocidad angular, es decir, la medida del dispositivo.
- Δt el tiempo que pasa entre muestra y muestra, en nuestro caso tomaremos 10 ms.

4.3.2 Errores en las medidas.

Tras tomar medidas de cualquier sensor siempre encontramos diferencias entre el valor medido y el valor real, estos se deben básicamente a los errores y al ruido.

En el caso del acelerómetro sus medidas suelen ser bastante precisas, permitiendo medir cualquier ángulo de manera fiable, pero su ruido asociado es muy alto, cualquier giro en uno de los ejes provocara una fuerza centrífuga en el otro, aunque no lo estemos moviendo.

Por el otro lado, el giróscopo nos permite obtener medidas muy estables al ruido, pero al tener una sensibilidad no nula, siempre contendrá un error asociado, que integrándolo comenzará a alejarse del valor real cada vez más, este error asociado es conocido como *drift*.

De esta manera se nos presenta la tesisura de reducir el ruido asociado a la medida, el cual por teoría sabemos que vendrá asociado a alta frecuencia y el error de *drift* del osciloscopio, el cual vendrá por un error de baja frecuencia.

Se decide por tanto la aplicación de un Filtro Complementario. Este filtro surge de la unión de dos filtros más: un filtro paso alto o High-Pass Filter para el giróscopo y un filtro paso bajo o Low-Pass Filter para el acelerómetro. De esta forma conseguiremos eliminar mayormente el error en nuestra medida.

La fórmula resultante de combinar estos filtros es la siguiente:

$$\alpha_{t+1} = 0.98 * (\alpha_{t_0} + \varphi_{giro} * \Delta t) + 0.02 * \varphi_{accel}$$

Aplicando esto a cada uno de los ejes se puede obtener una medida fiable del giro.

4.3.3 Implementación en el código.

```

void MPU6050_Read_Gyro (void)
{
    uint8_t Rec_Data[6];
    // Leemos 6 BYTES de datos comenzando por GYRO_XOUT_H register
    HAL_I2C_Mem_Read (&hi2c1, MPU6050_ADDR, GYRO_XOUT_H_REG, 1, Rec_Data, 6, HAL_MAX_DELAY);

    Gyro_X_RAW = (int16_t)(Rec_Data[0] << 8 | Rec_Data [1]);
    Gyro_Y_RAW = (int16_t)(Rec_Data[2] << 8 | Rec_Data [3]);
    Gyro_Z_RAW = (int16_t)(Rec_Data[4] << 8 | Rec_Data [5]);

    /***Convertimos los valores crudos o RAW en dps (°/s)
     * Tenemos que dividir de acuerdo a la escala seleccionada en FS_SEL
     * Como hemos configurado FS_SEL=0. Lo dividiremos entre 131.0
     ***/ 

    Gx = Gyro_X_RAW/131.0;
    Gy = Gyro_Y_RAW/131.0;
    Gz = Gyro_Z_RAW/131.0;
    //Filtro paso Banda, 2% aceleración y 98% angulo, debería dar el giro
    Angle[0]=0.98*(Angle[0]+Gx*0.01)+0.02*Acc[0];
    Angle[1]=0.98*(Angle[1]+Gy*0.01)+0.02*Acc[1];
    //Convertimos los valores en enteros de 8 bits acotados a 255
    dato2send[0]=((Angle[0]+90)*255/180);
    dato2send[1]=((Angle[1]+90)*255/180);

    if(dato2send[0]>255)
        dato2send[0]=255;
    else if(dato2send[0]<0)
        dato2send[0]=0;

    if(dato2send[1]>255)
        dato2send[1]=255;
}

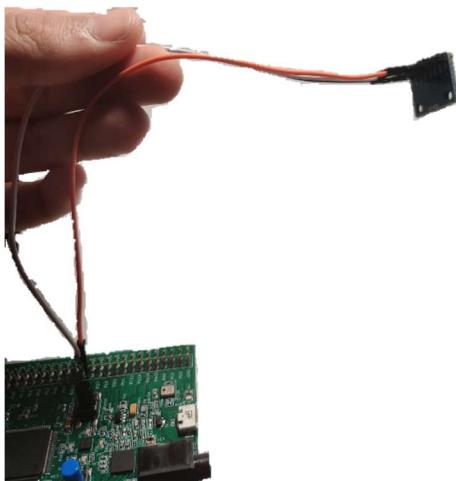
```

Al final de la función se transforma el valor obtenido en un entero de 8 bits para que pueda ser pasado a la FPGA mediante I2C.

4.3.4 Prueba y demostración de uso.

Probaremos para varias posiciones del sensor como varía la medida del giro. Tomando como cero el ángulo del sensor cuando apunta completamente hacia arriba.

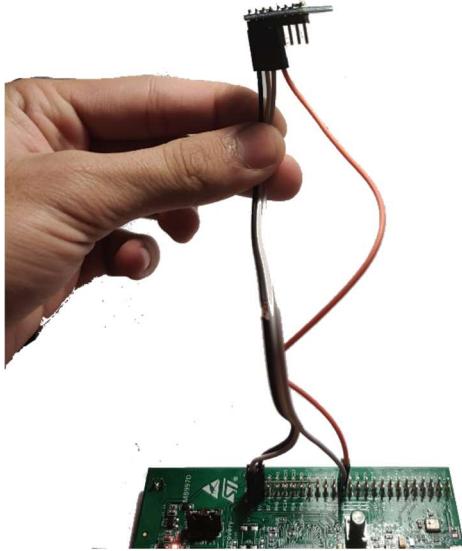
Si giramos el sensor para la derecha:



Expression	Type	Value
↳ Gx	float	-1.78625953
↳ Gy	float	5.46564865
↳ Gz	float	-2.19847322
↳ Gyro_X_RAW	int16_t	-163
↳ Gyro_Y_RAW	int16_t	908
↳ Gyro_Z_RAW	int16_t	-223
↳ Ax	float	-0.809326172
↳ Ay	float	0.0788574219
↳ Az	float	-0.165283203
▼ Acc	float [2]	[2]
↳ Acc[0]	float	-3.53493547
↳ Acc[1]	float	79.2173767
▼ Angle	float [2]	[2]
↳ Angle[0]	float	-5.73665524
↳ Angle[1]	float	80.0627289
▼ dato2send	uint8_t [2]	[2]
↳ dato2send[0]	uint8_t	119 'w'
↳ dato2send[1]	uint8_t	240 'd'

Las variables Angle[0] y Angle[1] se corresponden con el giro del sensor, puede verse que girando el sensor hacia la derecha producimos que el ángulo Y se ponga hasta casi 80 y se puede traducir a su correspondiente valor en un entero de 8 bits.

Apuntando el sensor hacia arriba obtenemos:



Expression	Type	Value
↳ Gx	float	-1.57251906
↳ Gy	float	4.68702269
↳ Gz	float	0.343511462
↳ Gyro_X_RAW	int16_t	-111
↳ Gyro_Y_RAW	int16_t	625
↳ Gyro_Z_RAW	int16_t	45
↳ Ax	float	0.171875
↳ Ay	float	-0.109863281
↳ Az	float	0.957275391
↳ Acc	float [2]	[2]
↳ Acc[0]	float	8.23485756
↳ Acc[1]	float	-10.3246222
↳ Angle	float [2]	[2]
↳ Angle[0]	float	7.26684189
↳ Angle[1]	float	-6.98278999
↳ dato2send	uint8_t [2]	[2]
↳ dato2send[0]	uint8_t	137 '211'
↳ dato2send[1]	uint8_t	117 'u'

Apuntando hacia arriba como habíamos aproximado antes, el valor se acerca mucho a 0 en cada uno de sus ángulos, no es un valor nulo puesto que la colocación no es perfecta. Además, se puede observar que los datos a enviar están más o menos en la mitad de 255 que era lo esperable. Si inclinamos el sensor para el lado que nos queda:

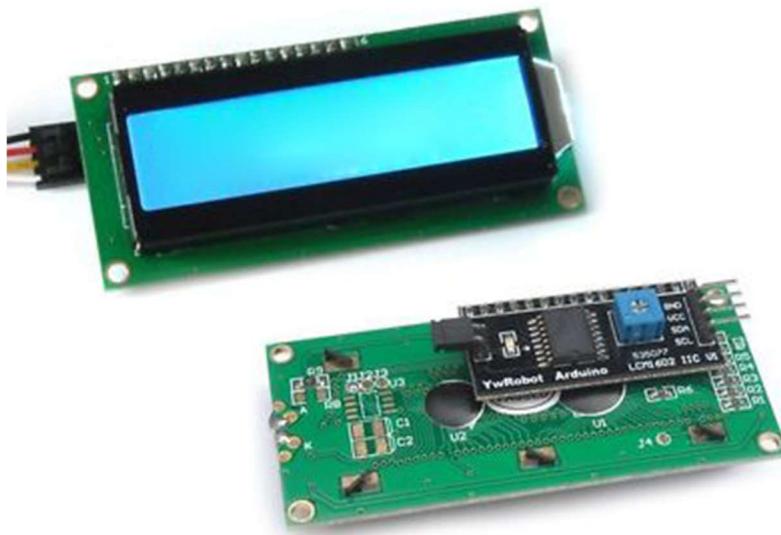


Expression	Type	Value
↳ Gx	float	3.07633591
↳ Gy	float	1.54961836
↳ Gz	float	0.114503816
↳ Gyro_X_RAW	int16_t	-317
↳ Gyro_Y_RAW	int16_t	789
↳ Gyro_Z_RAW	int16_t	20
↳ Ax	float	1.09814453
↳ Ay	float	0.0688476562
↳ Az	float	-0.189208984
↳ Acc	float [2]	[2]
↳ Acc[0]	float	-2.69008207
↳ Acc[1]	float	-79.8929443
↳ Angle	float [2]	[2]
↳ Angle[0]	float	-3.36325145
↳ Angle[1]	float	-75.8046188
↳ dato2send	uint8_t [2]	[2]
↳ dato2send[0]	uint8_t	122 'z'
↳ dato2send[1]	uint8_t	20 '\024'

Obtenemos un -80 en el eje de la Y como era esperable. Los valores del byte estarán cercanos a 0.

5. Pantalla LCD 16x2 con bus I2C conectado.

Como display para el usuario, se decidió implementar una pantalla LCD que mostrase información útil de la que devuelva la FPGA al maestro STM. La pantalla en cuestión consiste en dos líneas de 16 caracteres de largo. Como esta pantalla tiene 16 pines de conexión, la opción más viable es conectar un bus de I2C que maneja todos estos pines y la cual se conecta al microprocesador con tan solo cuatro pines: las señales SDA, SCL, una fuente de 5V y tierra.



5.1 Librerías y configuración

Para simplificar nuestro trabajo, se tomó para el manejo de la pantalla LCD junto con el bus I2C una librería externa: “i2c-lcd.h”, pensada para implementarse junto con las librerías HAL de la STM32CubeIDE.

Para que la librería funcione, se ha de editar dentro del archivo de código fuente .c la definición de la dirección del esclavo, a la que tenga nuestro dispositivo en nuestro caso; y también cambiar el “handler” del puerto I2C al que vamos a usar en nuestro proyecto.

Hay que tener en cuenta que el “handler” hay que editarlo en las tres declaraciones que tiene en todo el archivo.

```

2  /** Put this in the src folder */
3
4  #include "i2c-lcd.h"
5
6  extern I2C_HandleTypeDef hi2c3; // change your handler here accordingly
7
8  #define SLAVE_ADDRESS_LCD 0x4E // change this according to ur setup
9
10 void lcd_send_cmd (char cmd)
11 {
12     char data_u, data_l;
13     uint8_t data_t[4];
14     data_u = (cmd&0xf0);
15     data_l = ((cmd<<4)&0xf0);
16     data_t[0] = data_u|0x0C; //en=1, rs=0
17     data_t[1] = data_u|0x08; //en=0, rs=0
18     data_t[2] = data_l|0x0C; //en=1, rs=0
19     data_t[3] = data_l|0x08; //en=0, rs=0
20     HAL_I2C_Master_Transmit (&hi2c3, SLAVE_ADDRESS_LCD,(uint8_t *) data_t, 4, 100);
21 }
22
23
24 void lcd_send_data (char data)
25 {
26     char data_u, data_l;
27     uint8_t data_t[4];
28     data_u = (data&0xf0);
29     data_l = ((data<<4)&0xf0);
30     data_t[0] = data_u|0x0D; //en=1, rs=0
31     data_t[1] = data_u|0x09; //en=0, rs=0
32     data_t[2] = data_l|0x0D; //en=1, rs=0
33     data_t[3] = data_l|0x09; //en=0, rs=0
34     HAL_I2C_Master_Transmit (&hi2c3, SLAVE_ADDRESS_LCD,(uint8_t *) data_t, 4, 100);
35 }
```

Una vez tenemos la librería a punto, la pantalla se inicializará con la función **lcd_init()** dentro del código ‘main’.

5.2 Funciones y uso

Esta librería nos ofrece un conjunto de funciones muy fáciles de utilizar para manejar el display. Las que usaremos serán las siguientes:

- **void lcd_clear():** borra el texto en pantalla, tanto de la primera como de la segunda línea.
- **void lcd_put_cur(int linea, int columna):** coloca el cursor en una posición específica del display, desde el cual empezará a escribir la cadena de caracteres que le enviemos en la próxima función. En nuestra pantalla sólo hay dos opciones para elegir la línea: 0 para la primera línea y 1 para la segunda. La columna será la posición dentro de la línea, si queremos aprovechar los 16 caracteres que tiene cada línea lo normal será elegir la columna 0, que es el primer carácter de la línea en cuestión. Si esta función no se declara antes de enviar el texto, por defecto el cursor estará en la posición (0,0).
- **void lcd_send_string(char* texto):** una vez hemos colocado el cursor donde nos interesa, con esta función enviaremos una cadena de caracteres que será la que veamos en pantalla. Esta función llama a su vez a otra función, **void lcd_send_data(char dato)**, la cual gestiona cada carácter y lo envía por I2C con la función HAL.

Estas son todas las funciones para utilizar que nos interesan para nuestro proyecto. Con lcd_clear() limpiamos la pantalla, con lcd_put_cur(0,0) señalamos que queremos escribir en la primera línea de texto, con lcd_put_cur(1,0) escribiremos en la segunda línea de texto, y con lcd_send_string(LineaLCD) imprimiremos en pantalla el texto que haya en la cadena de caracteres LineaLCD.

La librería está implementada para enviar los datos por I2C por polling, por lo que para usar el LCD hemos creado una función que se llama desde el bucle while(1), y que cada 100 milisegundos actualiza los valores de sus cadenas de caracteres y las envía por la función lcd_send_string.

El objetivo es que la pantalla muestre información útil sobre la posición del servo, para ello se recoge del buffer Rx la información del ángulo del servo desde la FPGA. Esta información, declarada como número entero de 8 bits, se traduce con una simple fórmula matemática a su valor en grados, y tras ello se transfiere a la variable Linea2LCD gracias a la función “sprintf”. La primera línea de texto tendrá impreso el texto de la cadena de caracteres Linea1LCD, que será “Servo” FPGA: permanentemente; mientras que la segunda línea imprime los grados del servo, que cambiarán constantemente.

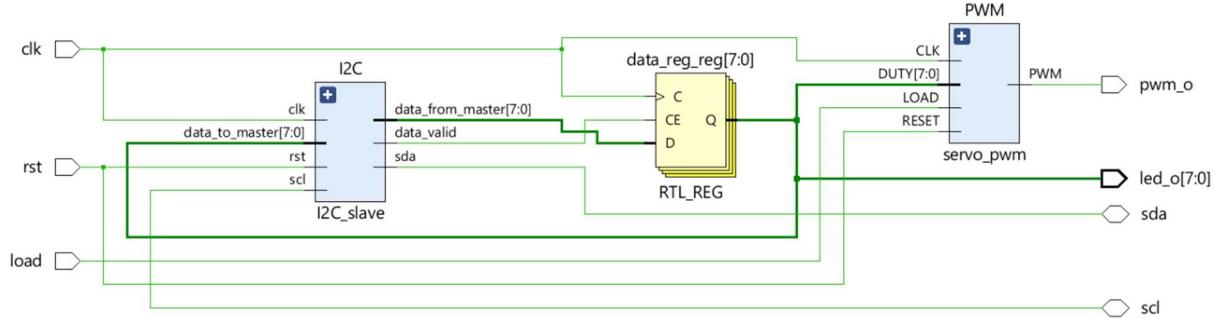
La pantalla mantiene cada carácter que le envíemos permanentemente en pantalla. Esto es un problema para nuestra segunda línea de texto, la cual puede pasar de tener 1 carácter (e.x. 4 grados) a tener, por ejemplo, 3 caracteres (e.x. -45 grados), y si posteriormente vuelve a haber un carácter en pantalla los 2 caracteres restantes permanecen en pantalla, ensuciando el resultado.

Para solucionar esto hay dos opciones, que le envíemos otro carácter que ocupe la misma posición o limpiemos la pantalla con lcd_clear. Al estar refrescando la información en pantalla constantemente cada 100ms, limpiar la pantalla con lcd_clear trae el problema de que el texto parpadee y se vea con menor intensidad. Nuestra decisión final ha sido añadir espacios al final de la cadena de Linea2LCD para que limpie el texto desactualizado.

6. Prueba final y conexiones.

6.1 Esquema de conexionado.

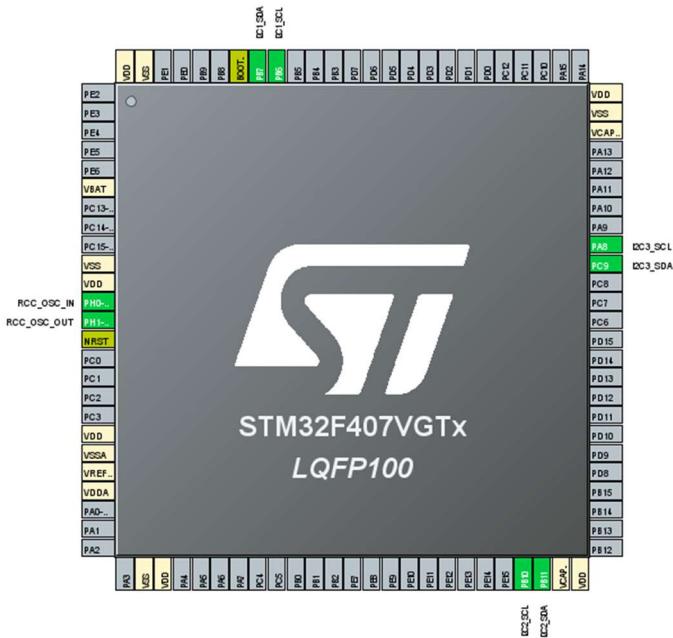
El esquemático del conexionado RTL en VIVADO queda como sigue, tras conectar los módulos en el fichero TOP.



Donde se envía de vuelta al master, el mismo dato que recibe. Dicho dato de vuelta, será el que se muestre en el display LCD transducido de digital a grados.

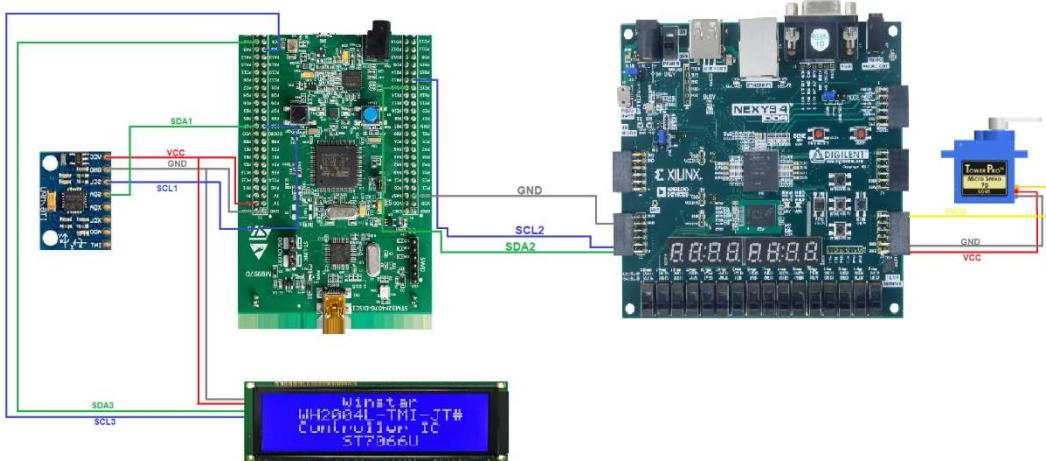
Por otro lado:

Tras configurar la herramienta CubeMx del STMCubeIDE con cada uno de los elementos explicados, quedará de la siguiente forma:



Se utilizarán los 3 puertos I2C que tiene la placa para establecer la conexión con los distintos periféricos, es verdad que una de las ventajas de la comunicación I2C es precisamente el poder conectar múltiples dispositivos con diferentes direcciones de esclavo a los mismos cables SCL y SDA. Sin embargo, se tuvieron problemas al implementar esto en la práctica, debido que, al conectar un dispositivo a la misma línea, por ejemplo, la FPGA, no se podía leer de la MPU, o se introducía demasiado ruido que hacía inviable el interpretar la información. Es por eso por lo que se determinó en pros de la fiabilidad, separar cada periférico en un puerto I2C diferente.

El esquema de conexión de todos los elementos del sistema quedaría de la siguiente forma:

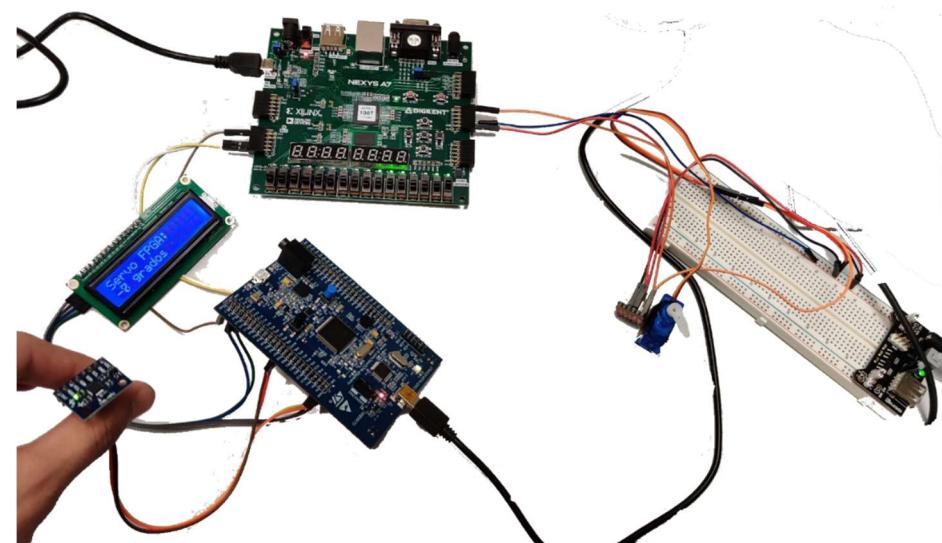


6.2 Prueba final.

Finalmente, se consiguió acudir al laboratorio y probar todas las características juntas funcionando a la vez, surgieron varios problemas, pero pudieron ser arreglados al momento.

Cuestiones que se tuvieron que arreglar en el laboratorio:

- I2C por interrupciones: Aunque se trató desde un primer momento introducir la comunicación con la pantalla mediante interrupciones y un temporizador para liberar la carga de trabajo del procesador principal, a la hora de depurar surgieron problemas en los que, sin llegar a averiguar la causa, el programa entraba en un bucle infinito si se llamaba a la función **HAL_GetTick()**. Por lo que se decidió dejar la comunicación con la placa por “polling”. Somos conscientes de que no es el método de comunicación óptimo, pero fue la única forma de implementar la funcionalidad del programa de forma estable.
- Lectura de la MPU: el código de los grados se encontraba desarrollado de forma que pudiera integrarse el valor de la velocidad angular para sacar la inclinación. Estas frecuencias de muestreo se mantenían estáticas a 10 ms sin nada conectado, pero al comunicar la placa y la pantalla, hubo que ajustar el tiempo ya que cada muestra paso de tomarse cada 30 ms, lo que deriva en una respuesta más escalonada en el movimiento del servo, pero poco apreciable por los valores que se manejan.



7. Conclusiones del proyecto.

Nuestro objetivo principal era usar una comunicación serie entre las placas STM y FPGA para realizar una tarea conjunta, y estamos satisfechos con el resultado obtenido.

Pese a las complicaciones que nos han ido surgiendo, se han conseguido conectar por I2C hasta 3 esclavos a una placa STM maestra. Mucho del tiempo del desarrollo de este trabajo se invirtió, principalmente, en la comunicación serie entre las placas. Durante este tiempo se intentó tanto realizar la comunicación por el protocolo SPI como por el I2C, sin éxito en un primer intento. Conseguimos solucionarlos, y gracias a ello hemos adquirido experiencia y hemos aprendido a investigar soluciones enfocadas a resultados.

Gracias a la teoría, se han podido contrastar los fundamentos de los protocolos de comunicación serie, así como cada uno de los protocolos que se describen en esta memoria. Además de esto, información de fuentes externas (libros o páginas web) nos han permitido introducirnos a estas tecnologías de una manera más específica.

Aunque en una primera aproximación se trató de implementar un movimiento 2D del eje X e Y mediante dos servos, debido a la complejidad y problemas que fueron surgiendo se decidió simplificar el diseño implementando únicamente uno de los servos.

Asimismo, se evaluó la posibilidad de introducir un KeyPad 4x4 con la funcionalidad de seleccionar manualmente el ángulo de giro del sensor, modificando el comportamiento entre 2 modos de funcionamiento: una opción sería el control del servo por los datos del MPU de forma automática, y en la otra opción el usuario seleccionaría con los números del KeyPad el valor del ángulo al mover el motor. Lamentablemente, no se pudo añadir adecuadamente esta funcionalidad ya que incrementaba drásticamente el tiempo de lectura de las medidas de la MPU al no poder realizarse mediante interrupciones, sino evaluando cada columna de manera cíclica por si se pulsaba algún botón.

En definitiva, el trabajo ha ido transformándose conforme avanzábamos en él, pero la idea inicial ha podido salir adelante y, como grupo, estamos plenamente satisfechos del resultado.

8. Bibliografía.

Manuales de referencia:

- [Nexys 4 DDR](#)
- [Basys 3](#)

Comunicación I2C:

- [Implementing I2C Slave on an FPGA/CPLD](#)
- [Conectar Arduino con una FPGA por I2C](#)
- [FPGA-I2C-Slave. Versión del modulo I2C finalmente empleado para el trabajo](#)
- [FPGA-I2C-Minion. Versión más reciente del modulo I2C.](#)

Comunicación SPI:

- [Modos de Funcionamiento](#)
- [Descripción del protocolo:VHDL](#)
- [Configuración Half_Duplex y Full_Duplex](#)

Control PWM del servomotor:

- [Datasheet](#)
- [RC servo controller using PWM from an FPGA PIN](#)

MPU 6050:

- [Controllerstech: How to interface MPU6050 \(GY-521\) with STM32](#)
- [TFG: Diseño de un Vehículo Aéreo no Tripulado multipropósito.](#)
- [MPU-6000 and MPU-6050 product specification.](#)
- [MPU-6000 and MPU-6050: Register Map and Descriptions](#)