Universidad de Alcalá
Escuela Politécnica Superior

Grado en ingeniería de computadores

Trabajo Fin de Grado

Desarrollo de un driver para acelerómetro en placa Nexys 4ddr e integración en un prototipo de software de aplicación de un instrumento a bordo de un satélite.

Autor: Daniel Muñoz Toledano
Tutor y cotutor (en su caso): Óscar Rodríguez Polo

2022

Universidad de Alcalá

Escuela Politécnica Superior

Grado en ingeniería de computadores

Trabajo de fin de grado

Desarrollo de un driver para acelerómetro en placa Nexys 4ddr e integración en un prototipo de software de aplicación de un instrumento a bordo de un satélite.

Autor: Daniel Muñoz Toledano

Tutor/es: Óscar Rodríguez Polo

TRIBUNAL:

Presidente: Óscar García Población

Vocal 1º: Juan Ignacio García Tejedor

Vocal 2º: Óscar Rodríguez Polo

FECHA: 20-09-2022

ÍNDICE

Índice de figuras

1. Resumen

Este proyecto se centrará en utilizar la FPGA Artix-7 de la placa Nexys 4 DDR, para diseñar, implementar y validar un *driver* que use el acelerómetro ADXL362 incorporado en la placa, además diseñar modos de funcionamiento del mismo que expongan las diferentes capacidades del sensor. Esto se ha llevado a cabo mediante la utilización del procesador sintetizable LEON3 y otros núcleos ip proporcionados por la biblioteca GRLIB, además de herramientas como Vivado o GRMON.

Palabras clave: FPGA, LEON3,GRLIB, Nexys 4 DDR, GRMON

2-Abstract

This project will consist in the use of the FPGA Artix-7 integrated on the Nexys 4DDR to design, implement and test a driver that uses the ADXL362 accelerometer. In addition, several operating modes will be implemented in order to showcase the sensor's capabilities, using the LEON3 synthesizable processor, other ip cores provided by Gaisler's GRLIB and tools like Vivado or GRMON in the process.

KeyWords: FPGA, LEON3,GRLIB, Nexys 4 DDR, GRMON

3-Introducción

3.1. Planteamiento del trabajo

Las *FPGAs* son circuitos integrados diseñados para ser reprogramados a voluntad después del proceso de manufacturación para ajustarse a una función específica. Este concepto fué ideado por Ross Freeman, quien, apoyándose en la ley de Moore, asumió que el coste de los transistores disminuiría, provocando que las *FPGAs* se convertirían en una alternativa viable a los chips *ASIC*. Abanderando esta idea, en 1984 Ross Freeman co-fundó la compañía Xilinx.
Las *FPGAs* de alto rendimiento, como la que dispone la placa Nexys 4ddr, la Artix-7, permiten sintetizar diseños complejos mediante biblioteca *VHDL*, como el procesador *LEON3*. Junto con el uso del acelerómetro ADLX362, el objetivo de este trabajo será diseñar, implementar y validar un driver para el mismo, que, respecto a diferentes modos de funcionamiento, nos permita medir las aceleraciones tanto estáticas como dinámicas en sus 3 ejes.

3.2. Objetivos a conseguir

Para llevar a cabo este planteamiento deberemos analizar las capacidades de placa Nexys A7, sintetizar el procesador LEON3 y ejecutar la aplicación en C que desarrollemos, para ello identificamos los siguientes objetivos:

- Análisis del estado de la tecnología de la placa.
- Análisis del funcionamiento del procesador sintetizable LEON3.
- Implementación de un driver que nos permita explotar las capacidades de un sensor .
- Testear el desempeño del driver en la placa.
- Documentar todo el proceso.

Para ello se debe hacer uso de las siguientes documentaciones:

- El manual de referencia de la placa Nexys 4 DDR.
- Biblioteca GRLIB, para informarse del LEON3 y de los diferentes IP cores necesarios.
- El manual de usuario de GRMON, para interactuar con LEON3.
- El datasheet del acelerómetro ADXL362.

4. Estado del arte

4.1. Placa Nexys 4 DDR

La Nexys 4 DDR es una placa para el desarrollo de circuitos digitales basada en la FPGA de alta capacidad Artix-7 comercializada por Xilinx.

Teniendo en cuenta todos su *hardware* como puertos, leds, interruptores, display de segmentos, etc… La placa puede llevar a cabo diferentes diseños, desde simples circuitos digitales a *System-on-Chip* (SOC) basados en procesadores de propósito general como lo es LEON3, el procesador sintetizable en VHDL que vamos a usar. La placa también cuenta con una gran variedad de periféricos, de los cuales vamos a centrarnos en utilizar su acelerómetro, el ADXL362.

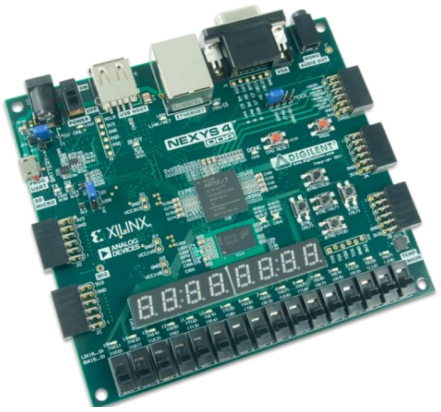


Figura 1: Placa Nexys 4 DDR

La Artix-7 está optimizada para lógica de alto rendimiento, ofreciendo más capacidad, rendimiento, y recursos que sus iteraciones pasadas, estas son algunas de sus características:

- 15850 capas lógicas, cada una con 4 LUTs de 6 inputs y 8 flip-flops
- Velocidades de reloj internas excediendo los 450MHz
- 4860 Kbits de block RAM rápida
- convertidor analógico-digital XADC incorporado

Figura 2: FPGA Artix-7

4.2. Biblioteca GRLIB

La biblioteca GRLIB de Gobham Gaisler AB consta de un set de *ip cores* diseñados para desarrollo de *System-on-Chip* (SOC), organizados en base a biblioteca VHDL, donde cada producto tiene asignado un nombre de biblioteca único.

Es con esta biblioteca a través de la cual vamos a tener acceso, mediante la licencia GPL, a los archivos fuentes del procesador sintetizable en VHDL, LEON3, y a un conjunto de controladores de diferentes periféricos de entrada/salida. La biblioteca está distribuida para sintetizarse en una amplia gama de FPGAs del mercado, estando entre ellas, la Artix-7 integrada en placa que vamos a usar, la Nexys 4 DDR.

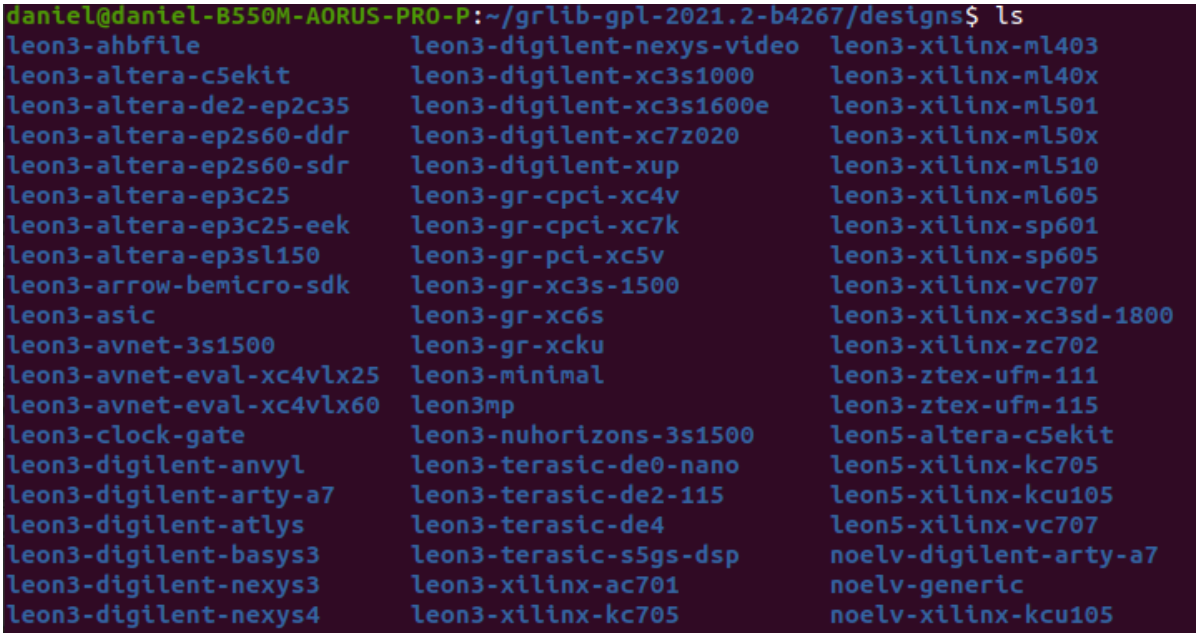


Figura 3: Diseños proporcionados por la biblioteca GRLIB

La ruta dentro de la biblioteca grlib donde están guardados los archivos del diseño del LEON3 para nuestra placa es la siguiente: *grlib/designs/leon3-digilent-nexys4ddr*. Si leemos el archivo *README.txt* se nos informa que el diseño del LEON3 soporta síntesis con Xilinix Vivado, y que ha sido testeado con Vivado 2017.3.

También, entre otra información, se nos indican una lista de los comandos más esenciales para manejar este diseño con Vivado, entre estos se encuentran los siguientes:

▪Para sintetizar el diseño y generar el *bitfile* usaremos el comando *make vivado*

▪Para programar la FPGA usaremos: *make vivado-prog-fpga*

4.2.1. Procesador sintetizable LEON3

El LEON3 es un procesador de 32 bits con arquitectura SPARC V8 sintetizable mediante a partir de su distribución en código VHDL. Este procesador es altamente configurable y especialmente adecuado para sistemas en chip. El código fuente completo de todo el SoC, incluido el LEON3, está disponible bajo la licencia GNU GPL, y, aunque no tenga todas las capacidades de las que disponen sus versiones de pago, permite un uso gratuito para investigación y educación. LEON3 también está disponible bajo una licencia comercial de bajo costo, lo que permite su uso en cualquier aplicación comercial a una fracción del costo de los núcleos ip comparables.

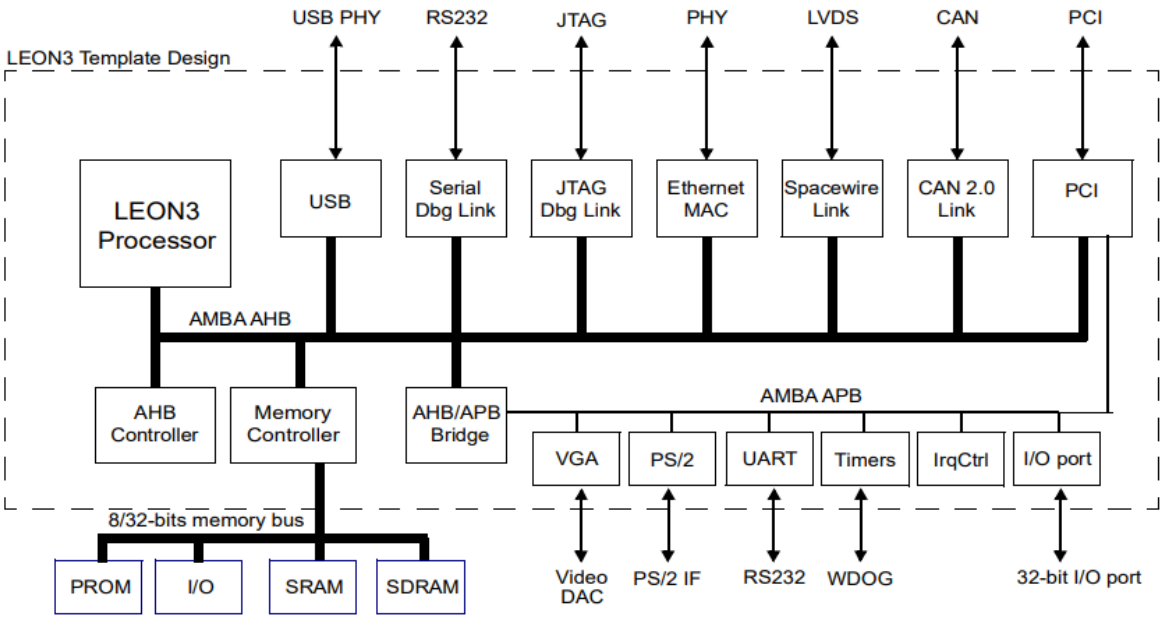


Figura 4: Diseño del procesador LEON3

4.3. Xilinx Vivado

Como ya hemos visto en el punto anterior, si queremos asegurarnos de que los archivos fuentes del diseño del LEON3 para la placa Nexys 4 DDR funcionan correctamente, deberemos utilizar la versión de Vivado 2017.3.

Para ello nos dirigiremos a la página web de Xilinx, y buscaremos la página de *Downloads*, que nos mostrará las últimas versiones disponibles de Vivado, si queremos la versión 2017.3 debemos ir al Vivado Archive, que es una página que contiene versiones antiguas de Vivado. Una vez ahí descargamos la versión "Vivado HLx 2017.3: WebPACK and Editions - Linux Self Extracting Web Installer (BIN - 100.61 MB)".

Tendremos que crear una cuenta de AMD para descargar el programa, ya que esta empresa adquirió Xilinx el 14 de febrero de este año 2022.

Una vez registrados nos descargará un .bin , con el cual instalaremos Vivado.

El proceso de instalación es sencillo, primero nos identificamos con nuestro usuario ID (correo electrónico de la cuenta AMD Xilinx) y su contraseña. Aceptamos términos y condiciones.

Seleccionamos la opción de instalar Vivado HL WebPACK.

Ahora debemos asegurarnos de que en Devices, 7 series, está seleccionada la casilla de Artix-7, la familia de FPGAs a la que pertenece la placa Nexys 4 DDR.

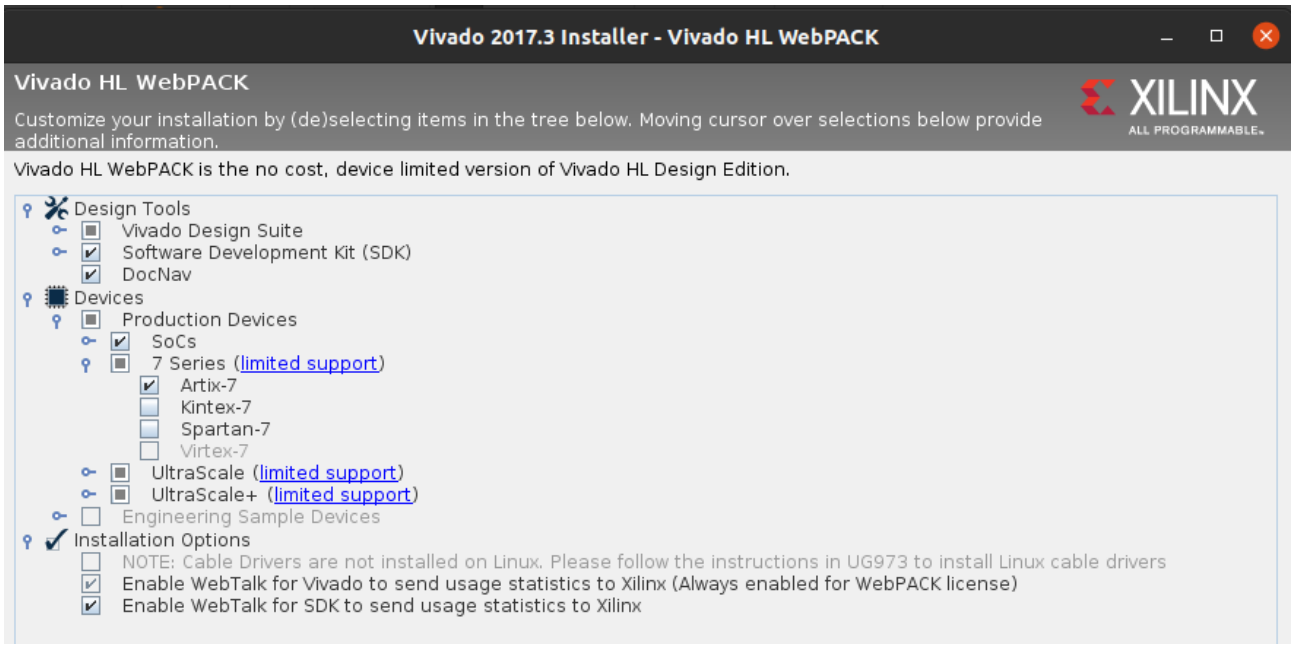


Figura 5: Intalación de vivado para Artix-7

Después seleccionamos directorio de instalación e instalamos. Únicamente faltaría instalar los controladores USB, cosa que haremos con el comando *./install_digilent.sh* en la ruta de la instalación de vivado, *data/xicom/cable_drivers/lin64/install_script/install_drivers*.

4.4. BCC (LEON Bare-C Cross Compilation System)

Una vez que tengamos el SoC basado en el procesador LEON3 sintetizado en nuestra placa, debemos usar el compilador BCC (*LEON Bare-C Cross-compiler system*) para poder compilar nuestros programas en C y que estos sean aptos para la arquitectura del LEON3.

BCC es un compilador cruzado diseñado para trabajar con procesadores Leon 2, Leon 3 y Leon 4. El compilador está basado en GNU y en la biblioteca newlib para C. El compilador cruzado nos permite desarrollar aplicaciones en C, o C++, usando un *host* basado en un PC de la arquitectura 0x86, y compilar, generando un ejecutable que pueda ser utilizado en una arquitectura SPARC, como la que tiene el LEON3.

4.5. GRMON

GRMON es un monitor para la depuración utilizado para el desarrollo de aplicaciones para sistemas LEON, de arquitectura SPARC, o de sistemas con NOEL-V, con arquitectura RISC-V. Este monitor permite la depuración de aplicaciones sobre diseños *System-On-Chip* basados en LEON3 y en el resto de núcleos ip de la biblioteca GRLIB IP.

El uso de esta herramienta nos va a permitir tener accesos de escritura y lectura a todos los registros y memoria del sistema, y cargar y ejecutar aplicaciones basadas en LEON/NOEL a través de interfaces de depuración como JTAG, o otros de propósito general como USB y UART.

Para poder usar GRMON deberemos descargarnos la versión académica de GRMON 3.2 en la pestaña de descargas de Gaisler. Después extraemos el contenido en el directorio deseado.

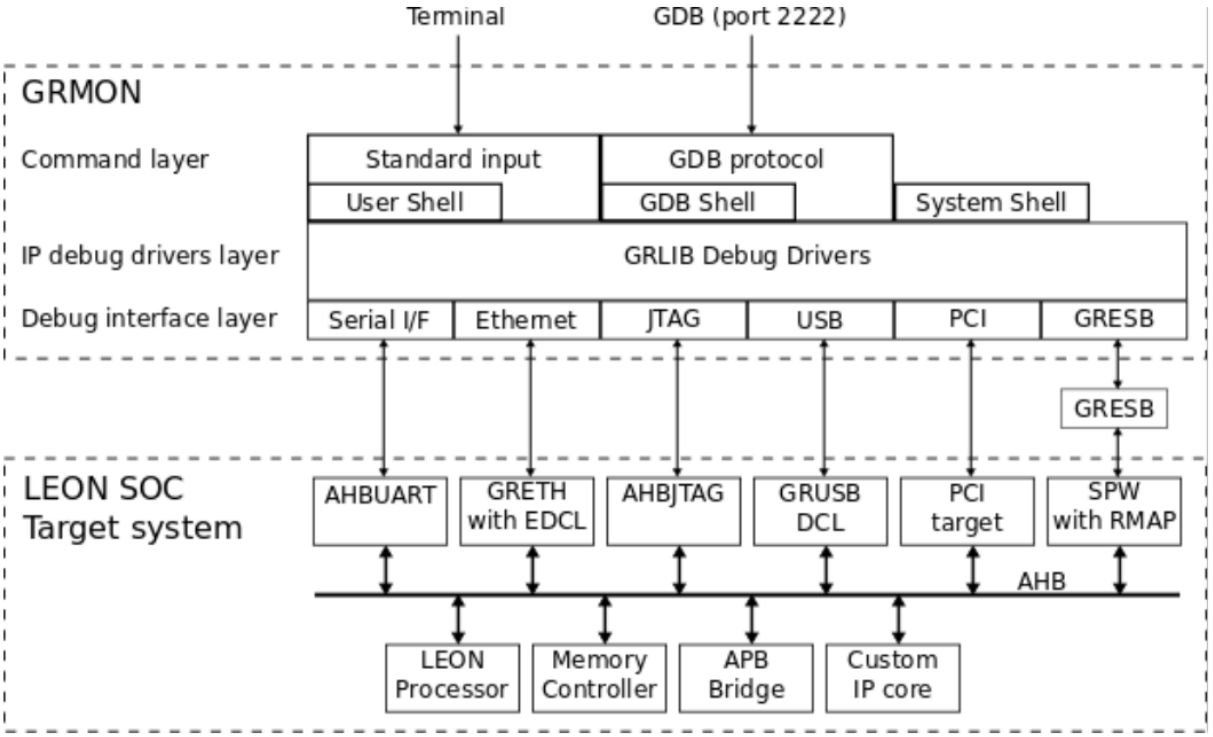


Figura 6: Relación entre GRMON y LEON3

5. Prueba piloto

Teniendo en cuenta la información, documentación y herramientas de los pasos anteriores, a continuación vamos a realizar un ejemplo de cómo se desarrollaría el proceso de hacer una aplicación en C y ejecutarla en el procesador LEON3 sintetizado en la placa.

5.1 Volcado del bitfile en la placa

En primer lugar, debemos sintetizar el SoC basado en LEON3 en la FPGA de la placa. Para hacer esto, debemos generar un bitfile a partir de las bibliotecas VHDL de SoCs basados en LEON3 proporcionadas por GRLIB, con el comando *make vivado*. La salida de este comando es muy larga, el final es este:

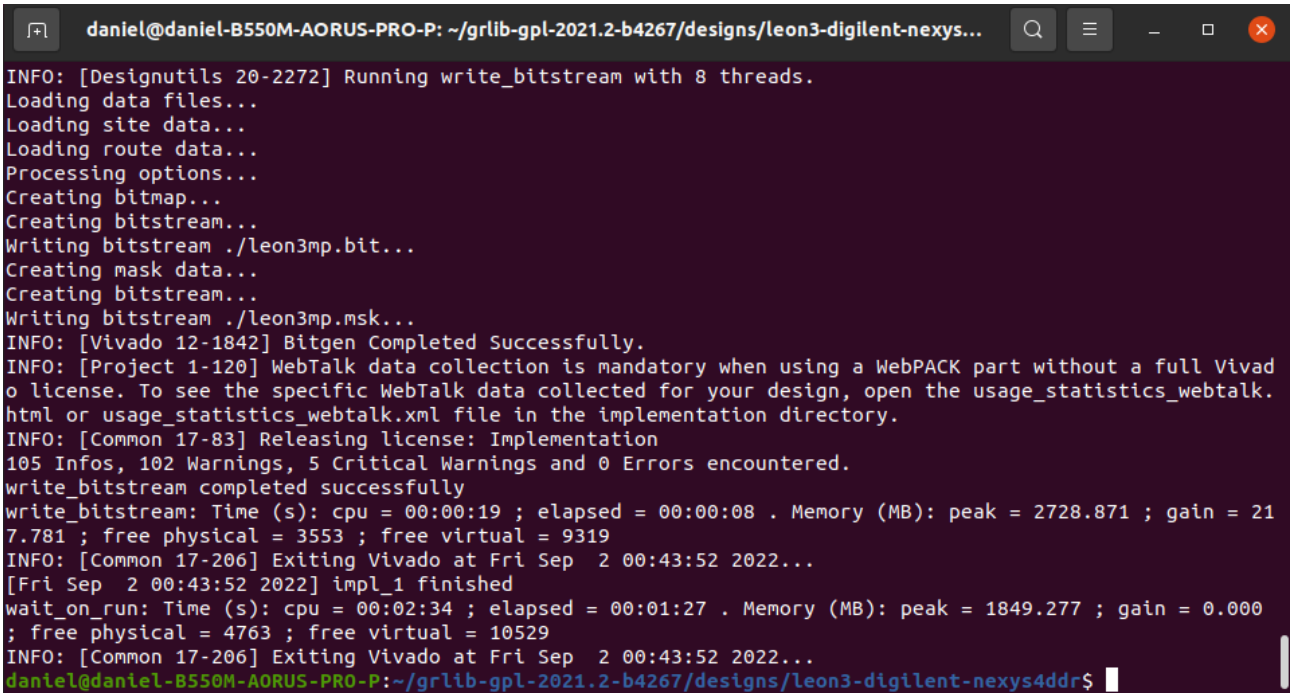


Figura 7: Salida del comando make vivado

Después procederemos a volcar este bitfile del SoC basado en LEON3 a la placa, que en nuestro caso estará conectada por USB al PC que hace de *host* usando para ello el comando *make vivado-prog-fpga*. Antes de hacer esto deberemos haber instalado los *cable drivers* para poder conectarnos con la placa por usb.

5.2 Desarrollo de una aplicación en C para el SoC basado en LEON3

En este caso, vamos a usar un ejemplo del clásico algoritmo de ordenamiento “burbuja”. Lo nombraremos *bubble.c*.

```
#include <stdio.h>

void bubble(int array[], int size) {
	// loop para acceder a cada elemento del array
	for (int step = 0; step < size - 1; ++step) {
		// loop para comparar los elementos del array
		for (int i = 0; i < size - step - 1; ++i) {
			// comparamos dos elementos adyacentes
			// > o < define si la ordenacion es ascendente o descendente
			if (array[i] > array[i + 1]) {
				// Si no están en orden cambiamos posición

				int aux = array[i];
				array[i] = array[i + 1];
				array[i + 1] = aux;
			}
		}
	}
}

void print_Array(int array[], int size) {
	for (int i = 0; i < size; ++i) {
		printf("%d ", array[i]);
	}
	printf("\n");
}
int main() {
	int array_data[] = {-2, 45, 0, 11, -9, 71, 3,99, -22, 255, 32,-69};


	int size = sizeof(array_data) / sizeof(array_data[0]);
	printf("Array proporcionado :\n");
	print_Array(array_data, size);

	bubble(array_data, size);
	printf("Array ordenado de menor a mayor:\n");
	print_Array(array_data, size);
}
```

5.3 Compilando el código con BCC

Para poder compilar *bubble.c* , necesitamos introducir el siguiente comando en la terminal: *export PATH=$PATH:/opt/bcc/bcc-2.2.0-gcc/bin*
Esto nos permitirá tener acceso al directorio donde ha sido instalado el BCC. Para compilar el archivo, en nuestro caso, se ha creado un archivo *Makefile* que nos permitirá compilar *bubbles.c* cómodamente con el comando *make all*, y eliminar el archivo compilado resultante con *make clean*.

```
CC = sparc-gaisler-elf-gcc

all:
$(CC) bubble.c -o bubble
clean:
rm -f *.o bubble
```

Algo importante a destacar de los comandos usados para compilar el código en c, es que la licencia que disponemos del procesador LEON3 no incluye una *Floating Point Unit*, es decir, no podemos disponer de un hardware que haga operaciones aritméticas con floats. Para poder hacer estas operaciones tendremos que añadir el parámetro *-msoft-float* al comando de compilación de BCC, esto nos proporcionará las bibliotecas software necesarias para solventar este problema.

5.4 Utilización de GRMON

Para utilizar el GRMON debemos localizarnos en el directorio donde hayamos instalado el GRMON. En mi caso es la siguiente:
/home/daniel/Desktop/grmon-eval-3.2.17/linux/bin64.

Después tecleamos *./grmon -u -digilent*. No sin antes conectar la placa vía USB y esperar a que el led *DONE* se encienda. Si al lanzar GRMON nos aparece el error *libdabs.so: cannot open shared object file: No such file or directory* procederemos con la instalación del paquete *digilent.adept.runtime_2.19.2-amd64.deb.*

Una vez dentro, el comando *info sys*, nos proporcionaŕa información de los IP cores de los dispositivos conectados al bus AHB. Algunos de la información más importante es:

```
cpu0      Cobham Gaisler  LEON3 SPARC V8 Processor
          AHB Master 0
```

Vemos que la FPGA tiene el *ip core* de LEON3

```
spi0      Cobham Gaisler  SPI Controller
          APB: 80000600 - 80000700
          IRQ: 5
          FIFO depth: 4, 1 slave select signals
          Maximum word length: 32 bits
          Controller index for use in GRMON: 0
```

El *ip core spi controller* nos permitirá comunicarnos con los periféricos

```
apbmst0   Cobham Gaisler  AHB/APB Bridge
          AHB: 80000000 - 80100000
```

Ip core AHB/APB Bridge que permite conectar estos dos buses

```
ddr2spa0  Cobham Gaisler  Single-port DDR2 controller
          AHB: 40000000 - 48000000
          AHB: fff00100 - fff00200
          16-bit DDR2 : 1 * 128 MB @ 0x40000000, 8 internal banks
          140 MHz, col 10, ref 7.8 us, trfc 135 ns
```

Ip core controlador de la memoria DDR2

Figura 9: salida del comando *info sys*, *ip cores* importantes

Para poder cargar y ejecutar un programa sobre el SoC de la Nexys4 DDR, primero deberemos introducir el comando *dd2delay scan*, el cual realiza una rutina de calibración de la memoria DDR. Una vez acabado este proceso, debemos salir y volver a entrar de GRMON para que el proceso surta efecto. Tras completar los pasos anteriores, podemos usar el comando *load* con la ruta del archivo compilado del programa de prueba *bubble.c* para cargar el programa en la memoria de la placa.

Si la carga ha tenido éxito, al usar el comando *run*, el programa debería ejecutarse correctamente:

```
grmon3> load /home/daniel/SPI-Nexys-Demo/bubble
          40000000 .text            60.8kB /  60.8kB    [===============>] 100%
          4000F350 .rodata           1.7kB /   1.7kB    [===============>] 100%
          4000FA30 .data             1.5kB /   1.5kB    [===============>] 100%
  Total size: 64.06kB (638.44kbit/s)
  Entry point 0x40000000
  Image /home/daniel/SPI-Nexys-Demo/bubble loaded

grmon3> run
Array proporcionado :
-2 45 0 11 -9 71 3 99 -22 255 32 -69
Array ordenado de menor a mayor:
-69 -22 -9 -2 0 3 11 32 45 71 99 255
  Program exited normally

grmon3>
```

Figura 10: carga y ejecución del algoritmo de la burbuja en GRMON

Vemos que el algoritmo funciona correctamente, el *array* se ordena de menor a mayor.

6. Acelerómetro ADLX362

Después de haber explicado todos los elementos básicos para el funcionamiento de cualquier aplicación en C, a continuación, vamos a proceder a explorar el componente clave de este trabajo de fin de grado, este es, es el sensor ADLX362.

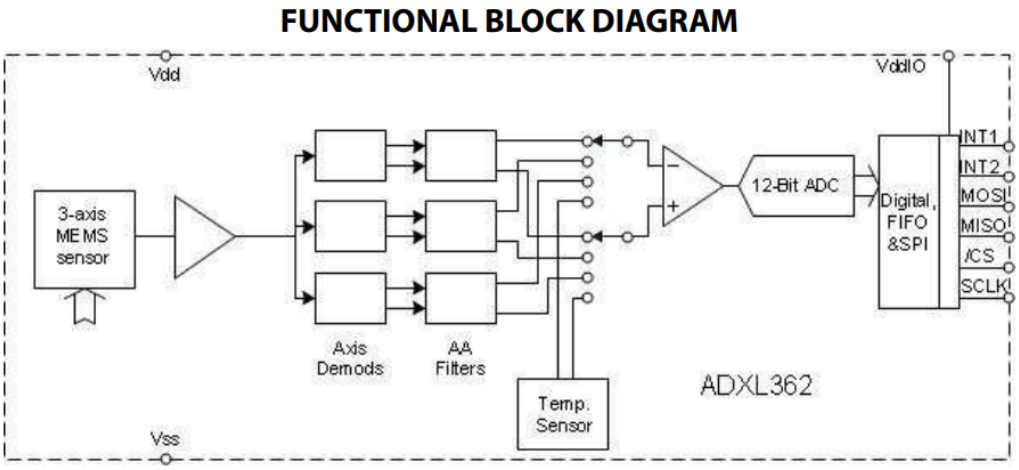


Figura 11: Diagrama del ADXL362

El sensor ADLX362 es un acelerómetro *MEMS* de 3 ejes, distribuido por la empresa Analog Devices, y el cual está integrado en la placa Nexys 4 DDR. El ADXL362 es un sensor de bajo consumo, que usa menos de 2μA a 100 Hz de *output data rate* y 270 nA cuando se detecta movimiento en el modo de activación por movimiento.

La tecnología *MEMS* (*microelectromechanical systems*) o las siglas en español SMEM, sistemas micro-electromecánicos, permite la creación de dispositivos electromecánicos miniaturizados. Su construcción está basada en las herramientas, técnicas e infraestructuras desarrolladas para la creación de circuitos integrados en silicio.
Estas estructuras micromecánicas se crean grabando patrones definidos en un sustrato de silicio para formar componentes como sensores o actuadores mecánicos que pueden moverse incluso fracciones de una micra. El tamaño de estos dispositivos suele rondar entre los 20 micrómetros y un milímetro.

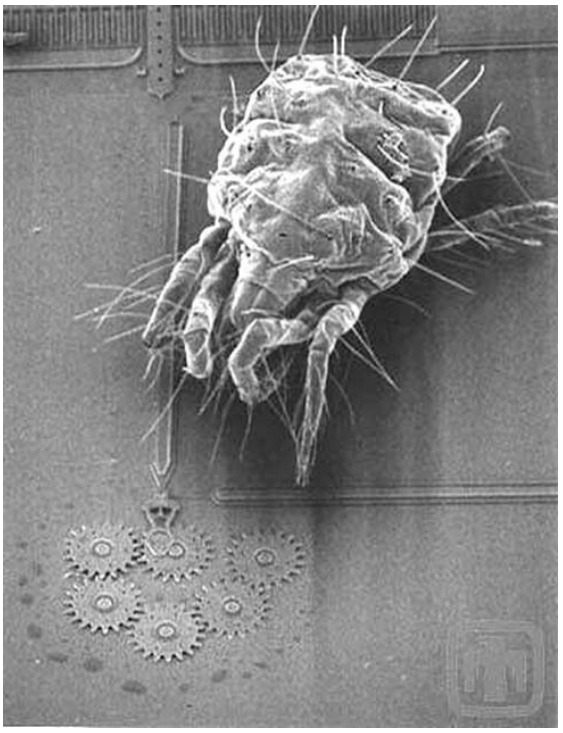


Figura 12: Ejemplo de estructuras MEMS comparadas con un ácaro

El elemento central de un acelerómetro SMEM es una pieza móvil que cambia la distancia

entre dos piezas fijas. La pieza móvil, de una masa conocida, está anclada por muelles, esto permite que la pieza central se mueva, respondiendo en un grado u otro a las diferentes aceleraciones a las que se vea sometida. Este movimiento cambia la distancia entre las placas fijas, variando la capacitancia entre placa móvil y fijas en el proceso, de esta manera es posible determinar la fuerza que está siendo aplicada a la pieza móvil.

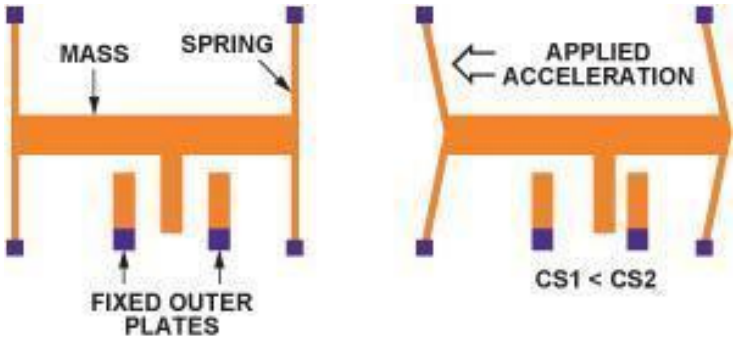


Figura 13: Representación del funcionamiento de un acelerómetro MEMS

Esto permite que el sensor pueda distribuirse en configuraciones de hasta 3 mm × 3.25 mm × 1.06 mm.
El ADLX362 siempre mide los valores de la aceleración de sus tres ejes con una resolución de salida de 12 bits, aunque, para la mayoría de las aplicaciones, es suficiente usar un formato de 8 bits que éste ofrece para tener una resolución aceptable , y así hacer transferencias de un solo byte, siendo más eficaces y eficientes.

Los rangos de mediciones de aceleraciones en los ejes del acelerómetro varía entre ±2g, ±4g, y ±8g (siendo 1g 9,8m/s^2), y contando con una resolución de 1mg por LSB en el rango de ±2 g.

La conexión entre la FPGA de la placa (Artix 7) y el sensor ADXL362 se puede ver en la figura. El ADXL362 actúa como dispositivo esclavo usando el protocolo de comunicación SPI.

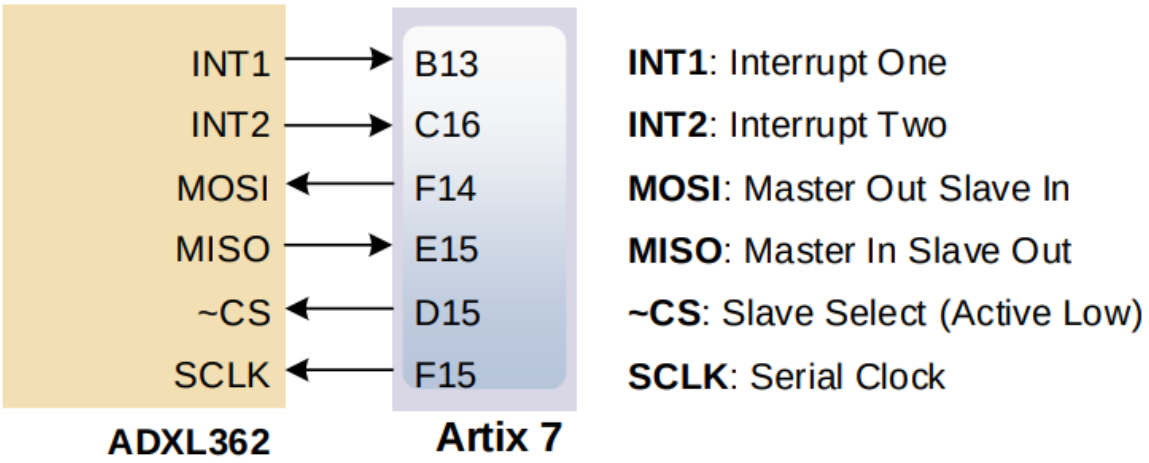


Figura 14: Diagrama de la comunicación entre la FPGA y el sensor

7. Diseño del controlador

7.1 Introducción y planteamiento

El objetivo es desarrollar un *driver* en C, que pueda hacer uso de las distintas capacidades que proporciona el ADLX362. Para demostrar el funcionamiento del mismo, se ha hecho uso del protocolo SPI, el cual hace posible la configuración del dispositivo, mediante la escritura en sus registros de control del acelerómetro y la lectura de los resultados, leyendo los registros de datos del mismo.

Esta capacidad de configurar el ADLX362 nos permite utilizar diferentes modos de funcionamiento que siguen unas características específicas dependiendo de cuál configuremos. En este trabajo se han incluido un total de cuatro modos de funcionamiento: 2 únicos y 2 modos parametrizables o *custom*.

7.2. Protocolo SPI

Como hemos dicho antes, usaremos el protocolo SPI para que la FPGA Artix-7 se comunique con el ADLX362, usando el *ip core* del *SPI Controller* de la biblioteca GRLIB.

7.2.1. Teoría

El *ip core* del *spi controller* está basado en los buses *on-chip* AMBA AHB y APB, que son la interfaz de interconexión on-chip que usa el LEON3 para la comunicación y la gestión de bloques funcionales en diseños SOC.

AMBA o *Advanced Microcontroller Bus Architecture* es un estándar de comunicación *on-chip* que fué diseñado para simplificar el diseño de microcontroladores empotrados, actualmente, su uso se ha extendido a más aplicaciones, siendo usado en diseños para ASICs y SOCs.

Esta arquitectura proporciona diversos beneficios, uno de los más importantes es que las especificaciones de AMBA proporcionan la interfaz necesaria para la reutilización de *ip cores*, de esta manera, hay una gran cantidad de diseños SOC y *ip core*s que lo utilizan. Esto se traduce en una aceleración en el diseño de aplicaciones y su consiguiente optimización en costes. También significa que los diseños que usen AMBA tendrán una mayor compatibilidad entre componentes *ip* y una mayor flexibilidad entre los mismos.

En este trabajo vamos a tratar dos de los cuatro buses que define AMBA: El bus AHB o *Advanced High-Performance Bus* y el bus APB o *Advanced Peripheral bus*

Las señales AHB y APB están agrupadas según funcionalidad en archivos VHDL, declaradas en la biblioteca GRLIB VHDL, más concretamente en el directorio *lib/grlib/amba*. Todos los cores de GRLIB comparten las mismas estructuras de datos para declarar las interfaces AMBA, y pueden por ello, pueden ser fácilmente conectadas unas con otras, tal y como hemos explicado antes.

En nuestro caso, al utilizar el *SPI Controller*, usaremos el AMBA APB, que es un bus de un solo *master* para interconectar unidades de baja complejidad y que usan una baja velocidad de datos.

El bus APB se comunica con el bus AHB a través de un esclavo que actúa como puente entre los dos buses, de esta manera podemos unir varios buses APB a un bus AHB. Esta funcionalidad es la que proporciona el *ip core AHB/APB Bridge*, *ip core* que utilizaremos para unir los buses AHB, APB y SPI.

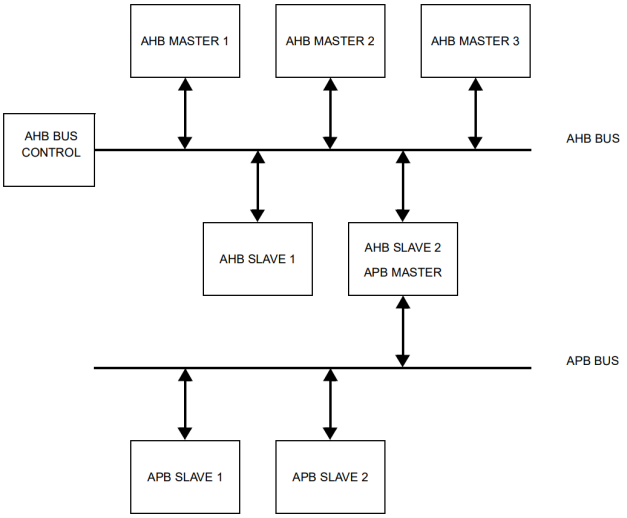


Figura 15: Diagrama de la relación entre los buses AHB y APB

El *ip core SPI Controller* permite comunicar el bus AMBA APB y el bus SPI, siendo este último un bus síncrono y *full-duplex*. La transmisión empieza cuando un dispositivo maestro selecciona un esclavo por medio de la señal SLVSEL o *Slave Select*. Los datos son transmitidos desde el maestro, por la señal *MOSI o Master-Output-Slave-Input* y desde la señal MISO o *Master-Input-Slave-Output.*

El ip *core* puede ser configurado para funcionar como master o como slave gracias a los registros mapeados en el espacio de memoria del APB. Los parámetros del SPI son altamente configurables mediante los registros, tamaño de palabra, ordenación de bit, *clock gap insertion*, transferencias automáticas periódicas…

Todos los modos de SPI están soportados, además de un modo de 3 líneas donde el *spi controller* usa una única línea de datos bidireccional.

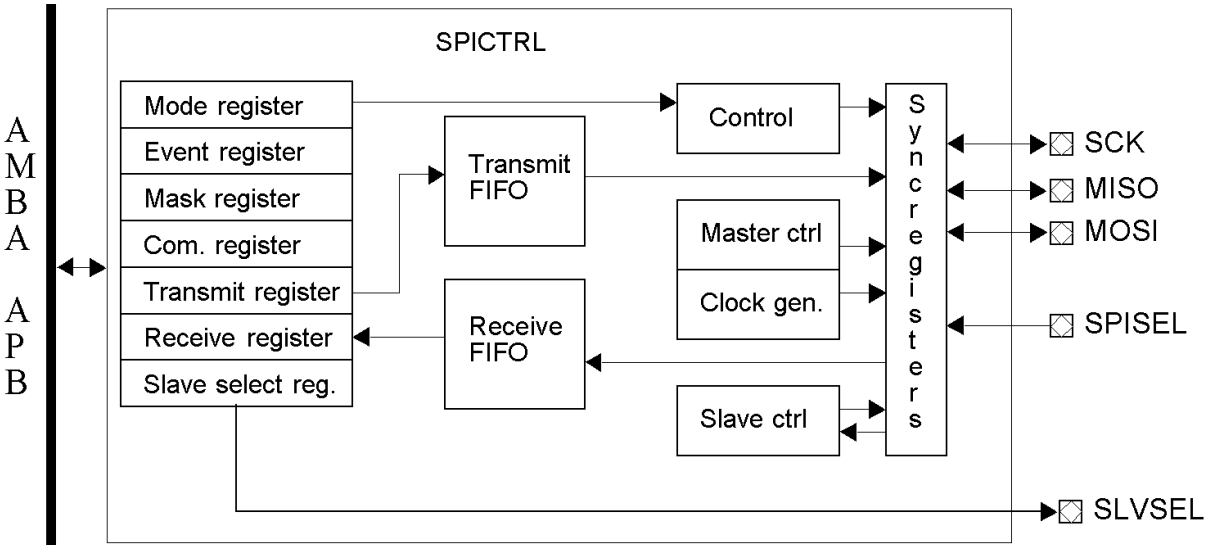


Figura 16: Diagrama del *ip core SPI Controller*

Los registros que vamos a emplear para la configuración del *SPI Controller* son los siguientes:

APB address offset	**Register**
0x00	Capability register 0
0x04	Capability register 1
0x08-0x1C	Reserved
0x20	Mode register
0x24	Event register
0x28	Mask register
0x2C	Command register
0x30	Transmit register
0x34	Receive register
0x38	Slave Select register (optional)

Figura 17: Registros usados por el *spi controller*

7.2.2. Código

Para la implementación de este proyecto, se han creado una serie de archivos, entre los cuales se encuentra el archivo ***leon3_types.h*** define los tipos de datos que vamos a usar, en el archivo ***nexys_spi.h***, definimos y asociamos las diferentes posiciones de memoria con los registros del sensor, para tener así un mejor y más fácil acceso a estos. El archivo *makefile* a usar será el siguiente:

```
CC = sparc-gaisler-elf-gcc
all:
        rm -f *.o spi_nexys_demo
        $(CC) -msoft-float main.c nexys_spi.c -o spi_nexys_demo
```

En el archivo ***nexys_spi.c*** definiremos las funciones necesarias para configurar el *spi controller*. A continuación, procederemos a explicar las funciones que hacen eso posible:

```
#include <stdio.h>
#include "nexys_spi.h"
struct SPI_regs * pADXL_SPI_REGS= (struct SPI_regs *) 0x80000600;
```

En primer lugar, haremos que el struct *SPI_regs* apunte a la posición de memoria 0x80000600 y denominamos como *pADXL_SPI_REGS*, para tener un acceso conveniente y adecuado a dichas posiciones de memoria.

A continuación describiremos la función *config_spi_adxl()*, que, como su nombre indica, configura el *spi controller* para poder utilizarlo.

```
void config_spi_adxl(void){
pADXL_SPI_REGS->Mode = 0; //Reset
pADXL_SPI_REGS->Mode &=~(1<<30); //LOOP mode disabled
pADXL_SPI_REGS->Mode |=(1<<25); //Master enabled
pADXL_SPI_REGS->Mode |=(1<<26); //REV enabled: MSB transmitted first, see TX/RX regs
pADXL_SPI_REGS->Mode |=(4<<16); //PM:4 70Mhz/20 => 3,5 Mhz (SCK)
pADXL_SPI_REGS->Mode |=(1<<2); //Ignore SPISEL
pADXL_SPI_REGS->Mode &=~(0xF<<20); //LEN = 0 (WLEN = 32 bits)
```

```
pADXL_SPI_REGS→Mode |=(1<<24); //Core enabled
printREGS();}
```

Para configurar el *spi controller* procederemos a usar *bitwise operations* fijándonos en el mapa del registro *mode* del spi controller:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
AMEN	LOOP	CPOL	CPHA	DIV16	REV	MS	EN	LEN				PM			
0	0	0	0	0	0	0	0	0				0			
rw*	rw	rw	rw	rw	rw	rw	rw	rw				rw			

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TWEN	ASEL	FACT	OD	CG					ASELDEL		TAC	TTO	IGSEL	CITE	R
0	0	0	0	0					0		0	0	0	*	0
rw*	rw*	rw	rw*	rw					rw*		rw	rw	rw	rw	r

Figura 18: Registro *Mode* del *spi controller*

Las operaciones necesarias son las siguientes:

▪Hacemos que todo el registro *mode* sea 0, haciendo un *reset*.

▪AND para poner el bit LOOP a 0, desactivando el modo *loopback*

▪OR para poner el bit MS a 1, significando que el core actuará como *master*, y no como *slave*.

▪OR para poner el bit REV a 1, para que sea el MSB el que sea transmitido primero.

▪OR para configurar los bits PM, *Prescale Modulus*, este valor es usado, en modo *master*, para dividir el clock del sistema, generando la señal de reloj, SCLK del *SPI Controller*.

▪OR para poner a 1 el bit IGSEL, haciendo que el core ignore el valor del *input SPI-SEL*.

▪AND para poner los bit LEN a 0, tomando el mayor valor de palabra que indica el *Capability Register* 0, que es 32 bits.

▪OR para poner a 1 el bit EN, haciendo que el core esté en modo *enabled.*

La función *printREGS()* saca por pantalla los registros del *spi controller*.

```
void printREGS(){

printf("---------------------------------\n" );
printf("Capability0: %08X\n", pADXL_SPI_REGS→Capability0 );
printf("Capability1: %08X\n", pADXL_SPI_REGS→Capability1 );
printf("Mode: %08X\n", pADXL_SPI_REGS→Mode );
printf("Event: %08X\n", pADXL_SPI_REGS→Event );
printf("Mask: %08X\n", pADXL_SPI_REGS→Mask );
printf("Slave Select: %08X\n", pADXL_SPI_REGS→SlaveSelect );
printf("---------------------------------\n" );}
```

Ahora procederemos a explicar la función *read_nexys_adxl()*. Esta función nos permitirá leer posiciones de memoria del ADXL362.

```
uint8_t read_nexys_adxl(uint8_t reg_id)
{
uint32_t write_timeout=0, word;
pADXL_SPI_REGS→Event |= (0x3<<11); //Clean flags UN and OV
while(((pADXL_SPI_REGS→Event & (1<<8))==0) && (write_timeout < 0xAAAAA))
{
write_timeout++; //Wait until NF=1
}
if(write_timeout < 0xAAAAA)
{
pADXL_SPI_REGS→SlaveSelect &= ~(1); // Slave select 1 goes down
// TX request: <CMD> : <REG_ID> : <0> : <0>
// RX data: <REG Value>:<REG+1 Value>
word=0;
word |= ADXL_READ << 24; // CMD
word |= reg_id << 16; // REG_ID

pADXL_SPI_REGS→Transmit=word; //Transmit word

while(pADXL_SPI_REGS→Event & (1<<31))
; // Wait while transfer in progress

pADXL_SPI_REGS→SlaveSelect |= (1); // Slave select 1 goes up

// Read received data
if (pADXL_SPI_REGS→Event & (1<<9))
{
// RX register 16 LSB bits contains REG value and REG+1 value
return (pADXL_SPI_REGS→Receive >> 8);
}
else
{
printf("Standing by no data at the moment...\n");
}
}
else{printf("Read reg NF Timeout...\n");}
return 0xFF;}
```

Antes de empezar a analizar esta función , deberemos tener el cuenta mapa del registro *Event*:

31	30 16	15	14	13	12	11	10	9	8	7 0
TIP	R	AT	LT	R	OV	UN	MME	NE	NF	R
0	0	0	0	0	0	0	0	0	0	0
r	r	r	wc	r	wc	wc	wc	r	r	r

Figura 19: Registro *Event* del *spi controller*

En cuanto al código, en primer lugar, limpiaremos la flag UN o *Underflow* y OV o *overflow* que únicamente debe estar a 1 cuando el *SPI Controller* actúa como esclavo (que no es nuestro caso).

En el primer *while*, incrementaremos el timeout en 1 siempre y cuando el bit *Not Full* (NF) esté a 0 y que el *write_timeout* sea menor que 0xAAAAA. Cuando el bit NF no esté en 0, procederemos a continuar con el resto de la función.

Para poder configurar la transferencia, antes debemos establecer el registro *Slave select* a 0, ya que, según la documentación del *spi controller*, las transferencias empiezan cuando un maestro elige un esclavo a través de la señal SLVSEL o *Slave Select*, poniendo la señal a cero dejamos de seleccionar el sensor ADLX362 como esclavo, dándonos la posibilidad de modificar los valores del *spi controller* antes de iniciar otra transmisión. Una vez hecho esto, haremos que el registro *Transmit*, tenga un valor específico, este valor debe estar formado por varios componentes:

- El código de la instrucción que queramos hacer, en este caso, queremos leer un registro, usaremos la instrucción de código 0x0B, READD. El *spi controller* es controlado desde el bus SPI mandando las siguientes instrucciones:

Instruction	**Description**	**Instruction code**	**Additional bytes**
RDSR	Read status/control register	0x05	Core responds with register value
WRSR	Write status/control register	0x01	New register value
READ	AHB read access	0x03	Four address bytes, after which core responds with data.
READD	AHB read access with dummy byte	0x0B	Four address bytes and one dummy byte, after which core responds with data
WRITE	AHB write access	0x02	Four address bytes followed by data to be written

Figura 20: Instrucciones del spi controller

▪La posición de memoria del registro que queramos leer, esta es la posición compuesta por 4 bytes que tiene en el bus AMBA.

▪Añadimos el dummy byte.

Esto provocará que el *spi controller* lleve a cabo una lectura de la posición del bus AMBA que le hemos especificado, en este caso, esa posición de memoria será un registro del ADXL362.

Estaremos en un *while* mientras que la transferencia se esté llevando a cabo, es decir, mientras que el bit TIP del registro *event* sea 1. (ver la figura 19)

Cuando el bit TIP del registro *Event* sea 0, es decir, cuando haya acabado la trasmisión, el registro SlaveSelect volverá a ser 1**,** indicando que el ADLX362 es seleccionado como esclavo, y que la transmisión debe comenzar.

Una vez hecho esto, procederemos a leer los datos recibidos si el bit NE o *not Empty* del registro *Event* no está a 0. Esto devolverá los 16 LSB de la dirección de memoria que hemos pedido.

En el caso de la función *write_nexys_adxl()*, como su nombre indica, posibilita que podamos escribir datos en una posición de memoria del bus AMBA que le especifiquemos. La idea es la misma que la función *read_nexys_adxl*, solo que, en vez de transmitir la instrucción 0x0b, la instrucción READD, transmitimos la señal 0x0A, la instrucción WRITE, por lo que, esta vez, no esperamos respuesta.

```
void write_nexys_adxl(uint8_t reg_id, uint8_t value)
{
uint32_t write_timeout=0, word;
uint8_t scratch;

pADXL_SPI_REGS→Event |= (0x3<<11); //Clean flags UN and OV
while(((pADXL_SPI_REGS→Event & (1<<8))==0) && (write_timeout < 0xAAAAA))
{
write_timeout++; //Wait until NF=1
}
if(write_timeout < 0xAAAAA){
pADXL_SPI_REGS→SlaveSelect &= ~(1); // Slave select 1 goes down
// TX request: <CMD> : <REG_ID> : <0> : <0>

word=0;
word |= ADXL_WRITE << 24; // CMD
word |= reg_id << 16; // REG_ID
word |= value << 8; // Value
pADXL_SPI_REGS→Transmit=word; //Transmit word

while(pADXL_SPI_REGS→Event & (1<<31))
; // Wait while transfer in progress
pADXL_SPI_REGS→SlaveSelect |= (1); // Slave select 1 goes up

// Clean received buffer
while (pADXL_SPI_REGS→Event & (1<<9)){
scratch = pADXL_SPI_REGS→Receive;}
}else{printf("Write reg NF Timeout...\n");}
return;}
```

7.3 Free Fall Mode

7.3.1 Teoría

Un uso común de los acelerómetros es la capacidad de detección de caída libre, por lo que vamos a implementar esta función en el driver para el ADLX362.

En primer lugar, hay que recordar que, si un objeto está en caída libre (físicamente hablando), el total de las fuerzas en él es 0, por lo que un acelerómetro en caída libre, mide una aceleración de 0g en todos sus ejes. Teniendo esto en mente, podemos usar las capacidades del ADXL362 para detectar cuando el acelerómetro se encuentra en esta situación, o en situaciones cercanas a ella.
El ADLX362 posee la capacidad de detectar tanto actividad, como inactividad. Estos estados pueden ser utilizados como *triggers* para manejar el modo de funcionamiento del acelerómetro, servir interrupciones al procesador *host*, o incluso funcionar de manera autónoma sin necesidad del mismo. Esta detección o no de movimiento, se verá reflejada en los distintos registros de control del acelerómetro.

El acelerómetro detecta actividad cuando la aceleración que mide en sus ejes se mantiene por encima de un umbral, el *activity threshold,* durante una cantidad de tiempo especificada, que llamaremos *activity timer*.

La inactividad es detectada cuando la aceleración medida se mantiene por debajo de un umbral, el *inactivity threshold*, durante una cantidad de tiempo especificada, que llamaremos *inactivity timer*.

Es importante destacar que el acelerómetro dispone de dos configuraciones, dentro de la detección de actividad u inactividad, para detectar el movimiento: estas son la configuración absoluta, y la configuración referenciada.

En el modo free fall usaremos la detección de movimiento absoluta, este tipo de detección de movimiento es usada para utilizar umbrales de aceleración y tiempos de actividad e inactividad especificados por el usuario para determinar si hay o no movimiento.

7.3.2 Código

Una vez explicados los conceptos básicos necesarios para entender este modo de funcionamiento, además de la funciones *write_nexys_adxl()*, procedemos a implementar el modo *free fall*. El código del modo es el siguiente:

```
void config_adxl_mode_free_fall(void)
{
// Configures the accelerometer to ±8 g range, 100 Hz
write_nexys_adxl( ADXL_FILTER_CTL, 0x83 );

// Enables absolute inactivity detection
write_nexys_adxl( ADXL_ACT_INACT_CTL, 0x04 );

// Free fall threshold to 600mg
write_nexys_adxl( ADXL_THRESH_INACTL, 0x96 );

// Free fall time to 30 ms
write_nexys_adxl( ADXL_TIME_INACTL, 0x03 );

//Map an interruption (optional)
write_nexys_adxl( ADXL_INTMAP2, 0x20);

//Set POWER CONTROL register
write_nexys_adxl( ADXL_POWER_CTL, 0x02 );
}
```

A continuación, pasaremos a explicar el funcionamiento y el objetivo de este código.
En primer lugar, tenemos esta sentencia:

```
// Configures the accelerometer to ±8 g range, 100 Hz
write_nexys_adxl( ADXL_FILTER_CTL, 0x83 );
```

La función *write_nexys_adxl(),* como explicamos en el apartado anterior, escribe un valor a un registro, todo ello pasado por parámetro. En este caso, el registro es el *ADXL_FILTER_CTL*, de dirección 0x2C, o lo que es lo mismo el *FILTER CONTROL REGISTER*.

FILTER CONTROL REGISTER

Address: 0x2C, Reset: 0x13, Name: FILTER_CTL

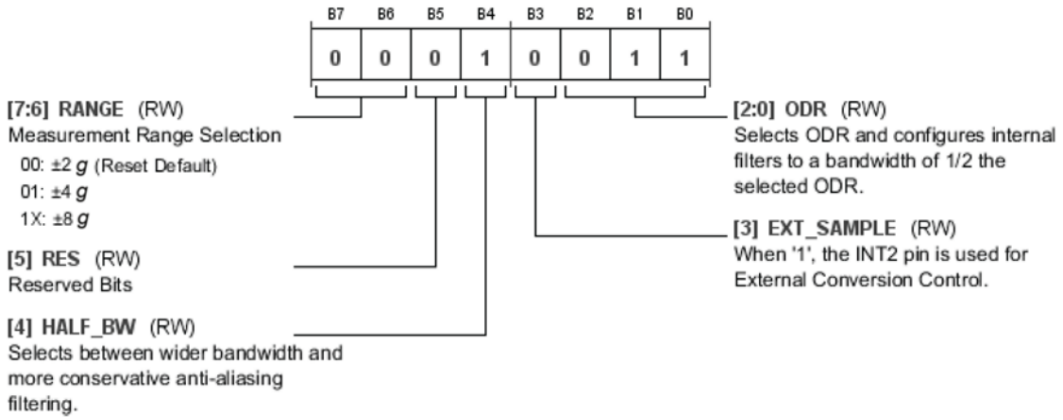


Figura 21: Registro *FILTER CONTROL* del sensor

Escribimos el valor 0x83 en este registro. Esto significa que el acelerómetro adoptará un rango g de operación de +-8g y que el *Output Data Rate* o ODR de 100Hz, según la descripción de los bits del registro . También establecemos, con el *HLF_BW* bit, que el ancho de banda de los filtros sea ½ del ODR.

Los posibles anchos de banda de los filtros que el bit *HLF_BW* permite configurar son ½ o ¼ del ODR seleccionado para asegurar que el teorema de Nyquist se cumple y que no se produce aliasing con las muestras tomadas.

En el modo free fall usaremos la detección de movimiento absoluta de inactividad, para ello, necesitaremos configurar el registro *ACT_INACT_CTL*:

```
// Enables absolute inactivity detection
write_nexys_adxl( ADXL_ACT_INACT_CTL, 0x04 );
```

Esta configuración se hará siguiendo el siguiente mapa del registro:

ACTIVITY/INACTIVITY CONTROL REGISTER

Address: 0x27, Reset: 0x00, Name: ACT_INACT_CTL

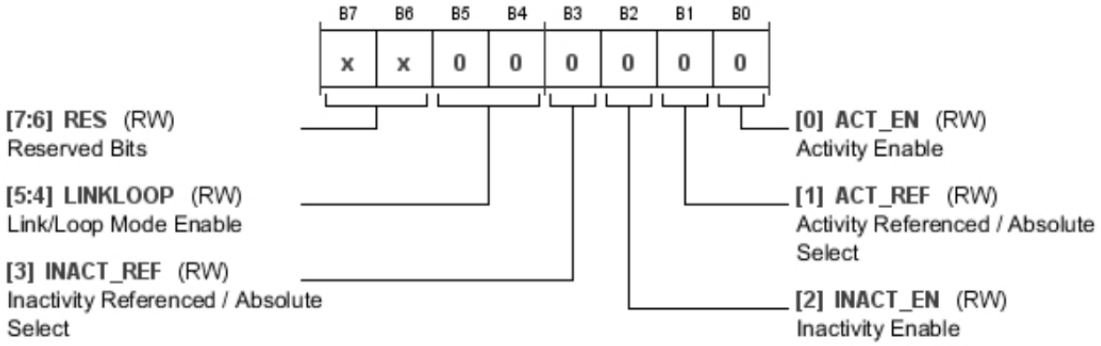


Figura 21: Registro *ACTIVITY/INACTIVITY CONTROL* del sensor

Escribiendo el valor 0x04, establecemos, en primer lugar, que el acelerómetro funcione en el modo por defecto, modo necesario para esta aplicación. Además, los valores del bit *INACT_EN* y el bit *INACT_REF*, hacen que se active la funcionalidad de detección inactividad y que la misma se realice con la configuración absoluta, respectivamente. No configuraremos la detección de actividad puesto que no la necesitamos.

A continuación tenemos la siguiente sentencia:

```
// Free fall threshold to 600mg
write_nexys_adxl( ADXL_THRESH_INACTL, 0x8D );
```

Al igual que la sentencia anterior, hacemos una operación de escritura al registro *ADXL_THRESH_INACTL*, de dirección 0x23, o lo que es lo mismo, el registro que, junto al *ADXL_THRESH_INACTH*, de dirección 0x24, componen los *INACTIVITY THRESHOLD REGISTERS*. En estos registros es donde se especifica el umbral de inactividad, de manera que, si el acelerómetro midiera valores de aceleración menores a este umbral durante el suficiente tiempo, un evento de inactividad sería detectado.

En este caso, disponer de una configuración óptima para ilustrar ese modo de funcionamiento, elegiremos un umbral de inactividad 600mg.

Para establecer este umbral de inactividad de 600mg, hay que determinar qué valor hay que escribir en los registros *INACTIVITY THRESHOLD* para que, efectivamente, el umbral sea de 600mg, la fórmula que hay que utilizar es la siguiente:

$$THRESH\_INACT = Threshold\ Value\ [g] \times Scale\ Factor\ [\text{LSB per } g]$$

Figura 22: Cálculo del valor del registro *THRESH_INACT*

Donde el *Threshold Value* [g] es valor en g que queremos usar para el umbral, en nuestro caso, queremos que el umbral sea de 600mg, es decir, de 0.6g, y el *Scale Factor*[LSB per g] es el factor de escalado, que depende de en qué rango de g's esté el acelerómetro configurado para funcionar, según la tabla 1. En nuestro caso, el rango de g's es de +-8g, así que el valor de *Scale Factor*[LSB per g] es de 235. Si hacemos esta multiplicación, el resultado es 141, que en hexadecimal es 0x8D.

Parameter	Test Conditions/Comments	Min	Typ	Max	Unit
Scale Factor at X_{OUT}, Y_{OUT}, Z_{OUT}	2 *g* range		1000		LSB/*g*
	4 *g* range		500		LSB/*g*
	8 *g* range		235		LSB/*g*

Figura 23: *Scale factor* por rango g.

De momento, hemos configurado tanto el rango de g's en los que el acelerómetro hará las mediciones, el ODR a la que hará las mediciones, hemos activado la detección absoluta de inactividad, y hemos establecido el umbral de inactividad a 600mg.

Con la siguiente sentencia, estableceremos el tiempo mínimo que tienen que estar registrándose medidas de aceleración menores al umbral de inactividad para que la inactividad se registre como tal. Se propone un valor de 30ms para ilustrar de manera óptima el funcionamiento del modo.

```
// Free fall time to 30 ms
write_nexys_adxl( ADXL_TIME_INACTL, 0x03 );
```

El valor que debemos dar a este registro para establecer el tiempo de inactividad en segundos es determinado por la siguiente fórmula:

$$TIME\_INACT = Time\ [\text{sec}] \times Data\ Rate\ [\text{Hz}]$$

Figura 24: Cálculo del valor del registro *TIME_INACT*

Donde *Time*[sec] es el valor en segundos que queremos que dure este tiempo de inactividad, en nuestro caso 30ms son 0.03s, y *Data Rate* [Hz] se refiere al ODR que hemos seleccionado en la primera sentencia, en nuestro caso, 100Hz. Esto hace que el valor que haya que escribir en el registro sea 0x03.

Solo faltaría mapear la interrupción de inactividad al pin INT1 o INT2 si queremos y nos es útil generar una, en nuestro caso, no es relevante para mostrar el funcionamiento del modo.

```
//Map an interruption (optional)
write_nexys_adxl( ADXL_INTMAP2, 0x20);
```

Por último, ya habiendo completado la configuración del Modo *Free Fall*, solo quedaría configurar el registro *POWER* .

```
//Set POWER CONTROL register
write_nexys_adxl( ADXL_POWER_CTL, 0x02 );
```

Para entender este registro utilizaremos el siguiente mapa:

POWER CONTROL REGISTER

Address: 0x2D, Reset: 0x00, Name: POWER_CTL

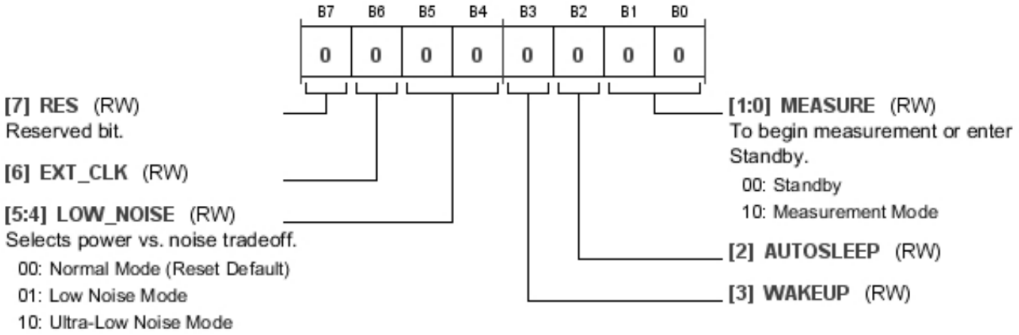


Figura 25: Registro *POWER CONTROL* del sensor

Escribiendo el valor 0x02 en este registro hace que el ADLX362 se ponga en Modo de medida, es aquí cuando podremos empezar a leer las mediciones de aceleración de los registros donde se almacenan. Dejaremos los demaś bits por defecto. Cabe destacar que, como se ilustra en el mapa, en este registro se puede configurar la relación entre consumo y ruido, los modos con menos ruido también son los que consumen más.

7.3.3. Resultados

Según la configuración que hemos definido, el ADLX362 debería de detectar una situación de caída libre en situaciones cercanas a esta. Para demostrar este funcionamiento, detectaremos esta condición cuando nos encontramos con una situación cercana a medir aceleración 0 en todos los ejes, cuando la medición de los ejes no sobrepase un umbral máximo de 600mg durante 30ms, y que cuando esto ocurra, el bit de inactividad se active en el registro de estado del acelerómetro. Una vez que compilamos el programa con BCC, lo carguemos con GRMON y lo iniciemos, la interfaz para seleccionar los distintos modos de funcionamiento es la siguiente:

```
grmon3> run
---------------------------------
Capability0:  01010486
Capability1:  00000000
Mode:         07040004
Event:        00000100
Mask:         00000000
Slave Select: 00000001
---------------------------------
Device ID:      AD
MEMS Device ID: 1D
Status:         C0
Enter desired mode (1-freefall  2-autonomous motion switch  3-free fall custom mode 4-motion switch custom mode)
```

Figura 26: GUI de la aplicación

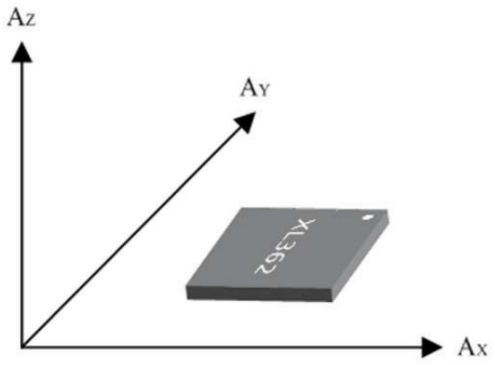
Al seleccionar el modo caída libre, tenemos la siguiente interfaz para entender las lecturas y el estado del acelerómetro

```
X_DATA: 80 Y_DATA: 80 Z_DATA: 80 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:44 | PWR CTRL:02
```

Figura 27 : Muestra 1 del modo *free fall*

X_DATA, *Y_DATA* y *Z_DATA* nos proporcionan los valores de aceleración que están siendo registrados. Los ejes se corresponden con la figura 28.

Figura 28 : Diagrama de ejes del sensor



A continuación se nos muestra el registro *status*, vamos a usar este registro para demostrar el funcionamiento de este modo. Además, se nos muestran tres registros más, que únicamente se muestran para visualizar más fácilmente la configuración del sensor. Esta información se irá repitiendo cada vez que el acelerómetro tome una medida nueva según el ODR estipulado.

Para demostrar el funcionamiento de este modo se ha seleccionado un umbral de 600mg, que dista de los 0g que se medirían si el sensor estuviera realmente en caída libre.

Gracias a esto, podemos simular una condición relativamente cercana a una caída libre simplemente dejando caer la nexys 4ddr con suavidad acompañada de nuestras manos, en vez de dejar que "caiga" verdaderamente, para evitar posibles daños al equipamiento.

Si el sensor detecta una condición de inactividad, es decir, menos de 600mg medidos en todos los ejes durante al menos 30ms, el registro *status* debería de registrarlo, activando el bit de inactividad, y esto es lo que efectivamente ocurre.

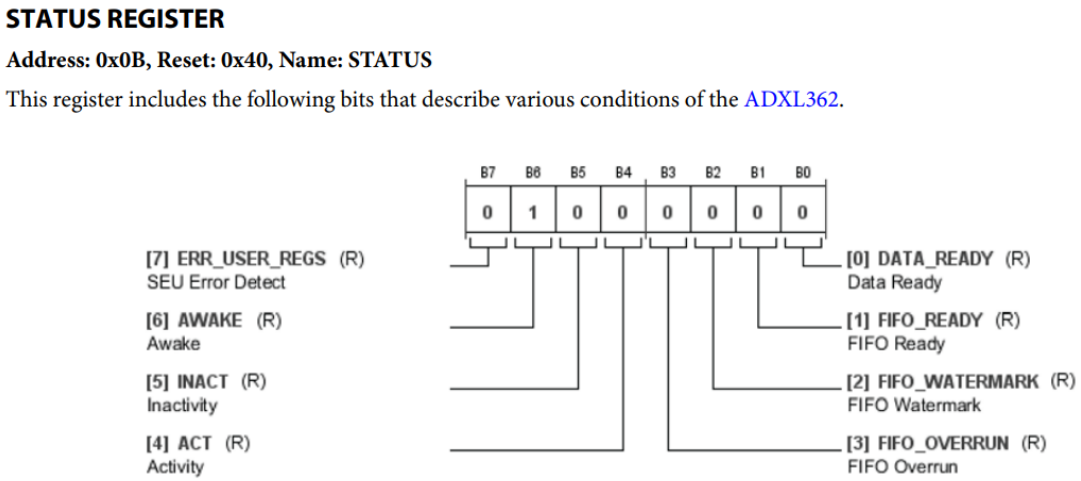


Figura 28: Regitro *STATUS* del sensor.

Teniendo el mapa del registro en cuenta, observamos el siguiente comportamiento:

```
X_DATA: 00 Y_DATA: 01 Z_DATA: F3 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
```

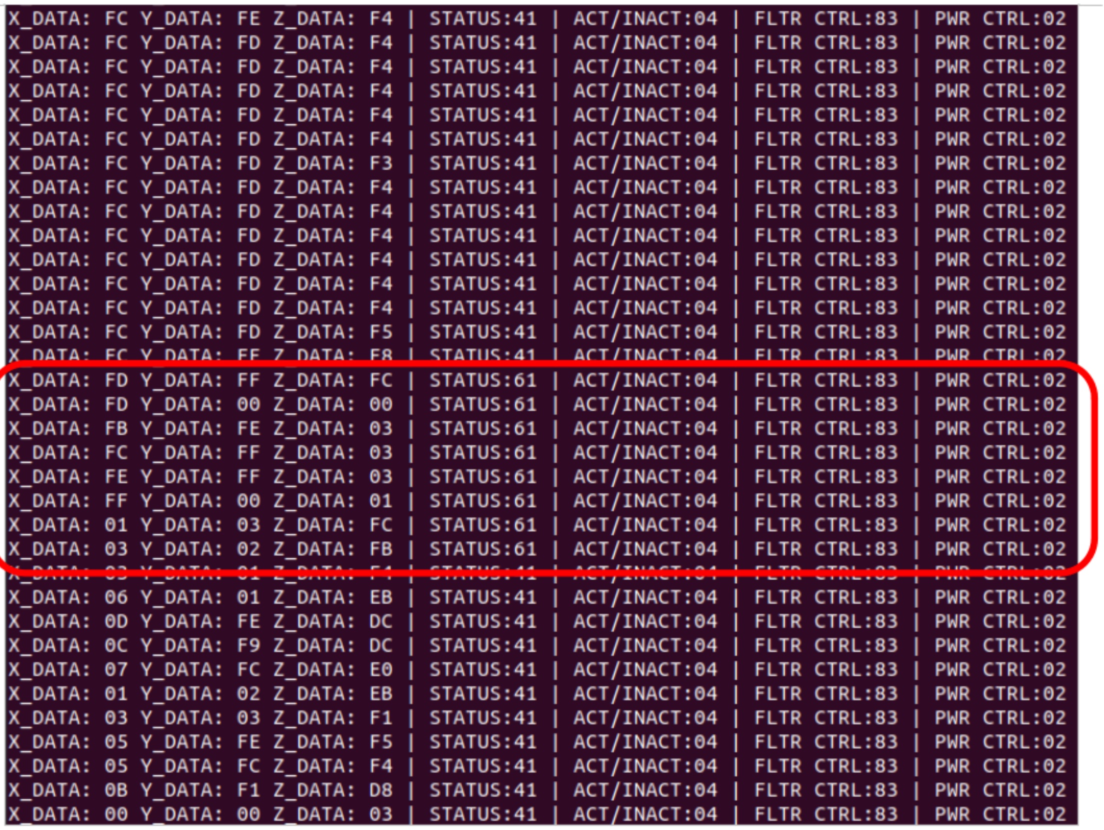
Figura 29: Muestra 2 del modo *free fall*

Cuando el acelerómetro está en reposo, los valores de *X_DATA* y *Y_DATA* son cercanos a 0 (cercanos entendiendo valores desde 0x00 a 0x01 si inclinamos hacia un lado y desde 0x00 a 0xFF si inclinamos hacia en contrario), mientras que el valor de *Z_DATA* siempre se mantiene en un valor constante. Esto es debido a que el acelerómetro está registrando la aceleración de la gravedad, si giramos la placa, esta aceleración de la gravedad se va distribuyendo entre los distintos ejes.

El registro *status*, según la figura del mapa del registro y el valor que tiene, indica dos cosas, que el sensor tiene datos nuevos que comunicar(el bit 0 vale 1), y que este se encuentra en el

estado *Awake*(el bit 6 está a 1), indicando que el sensor está activo. Este bit permanecerá en estado activo siempre, ya que el *Autosleep* está deshabilitado en este modo.

Si llevamos a cabo la prueba de “dejar caer” la placa con nuestras manos, podremos ver lo siguiente:



```
X_DATA: FC Y_DATA: FE Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F3 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FD Z_DATA: F5 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FE Z_DATA: F8 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FD Y_DATA: FF Z_DATA: FC | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FD Y_DATA: 00 Z_DATA: 00 | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FB Y_DATA: FE Z_DATA: 03 | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FC Y_DATA: FF Z_DATA: 03 | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FE Y_DATA: FF Z_DATA: 03 | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: FF Y_DATA: 00 Z_DATA: 01 | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 01 Y_DATA: 03 Z_DATA: FC | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 03 Y_DATA: 02 Z_DATA: FB | STATUS:61 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 03 Y_DATA: 01 Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 06 Y_DATA: 01 Z_DATA: EB | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 0D Y_DATA: FE Z_DATA: DC | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 0C Y_DATA: F9 Z_DATA: DC | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 07 Y_DATA: FC Z_DATA: E0 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 01 Y_DATA: 02 Z_DATA: EB | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 03 Y_DATA: 03 Z_DATA: F1 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 05 Y_DATA: FE Z_DATA: F5 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 05 Y_DATA: FC Z_DATA: F4 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 0B Y_DATA: F1 Z_DATA: D8 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
X_DATA: 00 Y_DATA: 00 Z_DATA: 03 | STATUS:41 | ACT/INACT:04 | FLTR CTRL:83 | PWR CTRL:02
```

Figura 31: Muestra 3 del modo *free fall*, condición de caída detectada

El sensor detecta valores normales y constantes hasta que lo dejamos “caer”, este movimiento causa dos cosas:

- La primera es que vemos que la lectura del registro *Z_DATA*, que en esta caso mide la aceleración de la gravedad cae hasta valores cercanos a cero durante un periodo breve de tiempo resaltado por el rectángulo rojo, este es el periodo de tiempo en el que la placa ha estado “cayendo”.

- La segunda, que también tiene que ver con la primera, es que el registro status cambia de valor, esto es debido a que el bit 4 del registro status, que indica que se ha detectado una condición de inactividad, ha pasado de 0 a 1, significando que el modo detecta la caída correctamente, hasta que este deja de caer.

7.4 Free Fall Mode Custom

Se ha diseñado un modo configurable a partir del modo *Free Fall*, que puede ser configurado por el usuario vía *inputs* en consola. Los parámetros que pueden ser configurados son el rango g, el ODR, el umbral de inactividad y el timer de inactividad. Los modos *custom* consisten en estructuras *switch-case* para seleccionar los parámetros que serán escritos en los distintos registros del ADXL362.

Lo único que cabe destacar de estos modos es que en el caso de seleccionar los valores de los umbrales de actividad/inactividad, o el tiempo de actividad/inactividad, hay que tener en cuenta parámetros como el rango g o el ODR elegidos para calcular qué valor hay que escribir en los registros. Un ejemplo de esto es el proceso para calcular el umbral de inactividad:

```
   printf("->Enter ANY desired Free fall threshold in mg (1g=1000mg).
100 mg , 300mg or 600mg are recommended\n");
       scanf("%d", &ffthresh);  // THRESH_INACT [g] = THRESH_INACT
[codes]/Sensitivity [codes per g]
       printf("%dmg Selected\n",ffthresh);
       double inact_thresh = 0.001*ffthresh;
       int code;
       int scale_factor;
       int low_masked;
       int high_masked;
       int high_masked_shifted;

       switch(f_fall_1.Parameter1){//g
           case 0x80://8g
               scale_factor=235;
               break;
```

```
            case 0x40://4g
                scale_factor=500;
                break;

            case 0x00://2g
                scale_factor=1000;
                break;

        }

        code=inact_thresh*scale_factor;
        low_masked = code & 0xff;
        f_fall_1.Parameter3=low_masked;
        high_masked = code & 0xf00;
        high_masked_shifted= high_masked>>8;
        f_fall_1.Parameter3_H=high_masked_shifted;
```

Como hemos explicado antes, el valor del registro debe ser el umbral en g deseado multiplicado por el *scale-factor* según el rango g elegido, considerando que si este valor es mayor a 0xFF, hay que separar el valor en dos registros, el *low* y el *high*. El código completo de los modos *custom* están disponibles en github.

Una vez seleccionados los parámetros, el modo se configura pasando los parámetros como struct:

```
void config_adxl_mode_free_fall_custom(struct
config_adxl_mode_free_fall_struct f_fall_1)
{
   // Free fall threshold to x mg
   write_nexys_adxl( ADXL_THRESH_INACTL, f_fall_1.Parameter3 );
   write_nexys_adxl( ADXL_THRESH_INACTH, f_fall_1.Parameter3_H );
   // Free fall time to x ms
   write_nexys_adxl( ADXL_TIME_INACTL, f_fall_1.Parameter4 );
   // Enables absolute inactivity detection
   write_nexys_adxl( ADXL_ACT_INACT_CTL, 0x04 );
   // Configures the accelerometer to ±x g range and ODR
   int var;
   var=(f_fall_1.Parameter1)|(f_fall_1.Parameter2);
   write_nexys_adxl( ADXL_FILTER_CTL, var );

   // Start mesurement
   write_nexys_adxl( ADXL_POWER_CTL, 0x02 );}
```

7.5 Autonomous Motion Switch Mode

7.5.1 Teoría

Las características del ADXL362 se prestan para configurar un modo en el que, una vez configurado el sensor, este funcione de manera independiente al procesador. Esto tiene numerosas ventajas, una de ellas es la mejora del consumo, ya que, si el propio sensor es el que controla en qué estado debe estar, (activo o inactivo), este puede regular el consumo dependiendo de las lecturas que tome. En esencia, el sensor podrá pasar de un estado activo o inactivo dependiendo de las lecturas que tome, ahorrando energía en el proceso.

Lo que queremos realizar es, como el nombre del modo indica, un modo en el que el sensor, mientras no detecte movimiento, esté en un estado "dormido", y que se "despierte" cuando detecte movimiento, este comportamiento puede ser utilizado en numerosas aplicaciones, como apagar cierta instrumentación mientras no se detecte movimiento, y encenderla únicamente cuando sí lo haya, optimizando el consumo de un sistema en gran medida.

Como ya hemos visto, el ADLX362 posee la capacidad de detectar tanto actividad, como inactividad. Esto se verá reflejado en los distintos registros de control del acelerómetro.

En el modo *autonomous motion switch* usaremos la detección de movimiento referenciada, que, al igual que el modo de detección de movimiento absoluto, utilizado en el modo *free fall*, consiste en establecer umbrales tanto de actividad como de inactividad, y un tiempo para determinar si hay movimiento o no.

Sin embargo, el modo referenciado, tanto para si es usado para la detección de inactividad o actividad, es especialmente útil para ignorar los efectos de aceleraciones estáticas sobre las lecturas de aceleración, como por ejemplo, el de la gravedad, y de esta manera, lograr que esta aceleración estática no interfiera en los umbrales que hemos establecido.

Una prueba de la utilidad de este comportamiento es la siguiente: con detección de movimiento absoluta, si el umbral de inactividad es menor que 1g, pongamos 500mg,

el sensor nunca detectaría un estado de inactividad, ya que en el eje z está midiendo permanentemente una aceleración de 1g, que es la fuerza de la gravedad. El modo de detección de movimiento referenciado soluciona este problema, descartando esta aceleración estática de la gravedad.

$$ABS(Acceleration - Reference) < Threshold$$

Figura 32: Configuración de detección de movimiento referenciada

El umbral de tiempo de inactividad puede ser configurado para variar entre los 2.5ms a los 90 minutos de inactividad.
Esto significa que el sensor podría esperar hasta 90 minutos para detectar un estado de inactividad, siempre y cuando las mediciones tomadas en esos 90 minutos hayan sido siempre menor al umbral de inactividad. Este rango de posibles valores nos da la posibilidad de hacer que el sensor quede "dormido" tras periodos muy pequeños de tiempo, favoreciendo así el consumo de energía sin sacrificar funcionalidad.

Pero para entender esto, debemos hablar de el cómo y el porqué es importante relacionar los estados de actividad y inactividad. Los estados de actividad e inactividad pueden ser usados para cambiar el funcionamiento del sensor.

En este modo utilizaremos el *loop mode*, esta configuración permite que el sensor no necesite a el procesador para su funcionamiento, de esta manera simplificamos la implementación del diseño y mejoramos el consumo. El funcionamiento del mismo se produce según la figura 30:

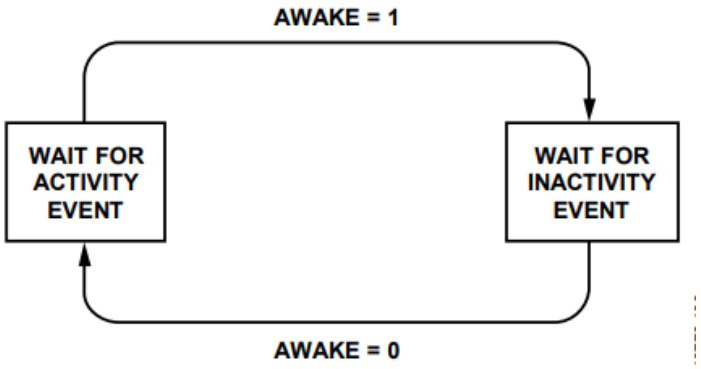


Figura 33: Diagrama del funcionamiento del modo *Loop*

Además, con objeto de optimizar el consumo al máximo, utilizaremos la funcionalidad de *Autosleep* la cual consiste en que, cuando un estado de inactividad ha sido detectado, el sensor entrará de manera automática a un estado llamado *Wake-up mode*.

Este modo es utilizado para la detección o no de movimiento a consumos extremadamente bajos, y es especialmente útil para implementar un interruptor activado por movimiento, ya que, permite que mientras no se detecte movimiento, el resto del sistema pueda ser "apagado", hasta que se detecte movimiento, cuando saldrá de este *Wake-up mode.*

Este modo, como ya hemos comentado, permite optimizar el consumo del sensor, al hacer únicamente 6 mediciones al segundo para determinar si hay o no hay movimiento. En caso de haberlo, el sensor podría lanzar una interrupción o despertar circuitería externa.

7.5.2 Código

Usaremos la misma función *write_nexys_adxl()*, ya explicada en el modo *free fall*, para escribir valores en posiciones de memoria específicas donde se encuentran registros de interés para configurar el funcionamiento del ADXL362.

```
void config_adxl_mode_autonomous_switch(void)
{
//set Activity Threshold
write_nexys_adxl( ADXL_THRESH_ACTL, 0xFA );
write_nexys_adxl( ADXL_THRESH_ACTH, 0x00 );

//set Inactivity Threshold
write_nexys_adxl( ADXL_THRESH_INACTL, 0x96 );
write_nexys_adxl( ADXL_THRESH_INACTH, 0x00 );

//Set Inactivity Time Threshold
write_nexys_adxl( ADXL_TIME_INACTL, 0x1E );

//Set ACTIVITY/INACTIVITY control register
write_nexys_adxl( ADXL_ACT_INACT_CTL, 0x3F);

//Map an interruption (optional)
write_nexys_adxl( ADXL_INTMAP2, 0x40);

//Set POWER CONTROL register
write_nexys_adxl( ADXL_POWER_CTL, 0x0E);
}
```

Escribiremos el valor 0xFA al registro *ADXL_THRESH_INACTL* y 0 al *ADXL_THRESH_INACTH*, que establecerá el umbral de actividad en 250mg.

Escribiremos el valor 0x96 al registro *ADXL_THRESH_INACTL* y 0 al *ADXL_THRESH_INACTH*, que establecerá el umbral de inactividad en 150mg.

Escribiremos 0x1E en el registro *ADXL_TIME_INACTL* para establecer el timer para la detección de inactividad en 5 segundos.

Escribiremos 0x3F en el registro *ADXL_ACT_INACT_CTL* para configurar el registro según el siguiente mapa:

ACTIVITY/INACTIVITY CONTROL REGISTER

Address: 0x27, Reset: 0x00, Name: ACT_INACT_CTL

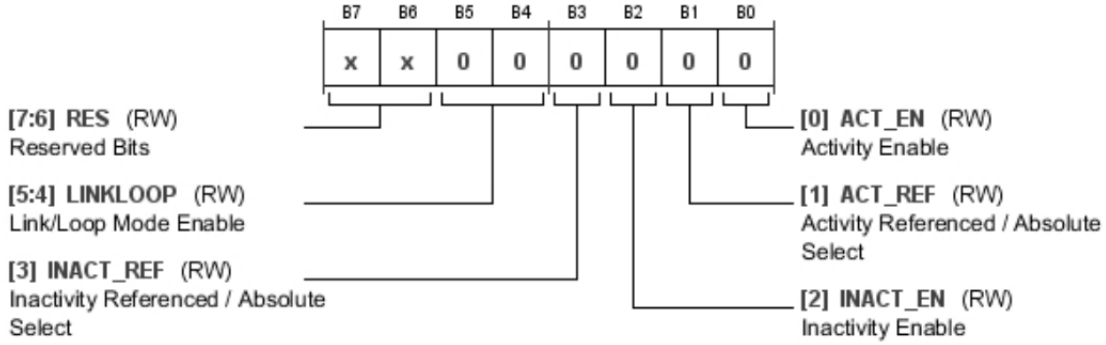


Figura 34: Registro *ACTIVITY/INACTIVITY CONTROL* del sensor

Con el valor 0x3F, activamos la detección tanto de inactividad como de actividad, las dos en configuración de detección referenciada. Además establecemos que el modo de funcionamiento sea el modo *Loop* cuyas propiedades hemos definido con anterioridad.

Podemos configurar que tanto el pin INT1 o INT2 tengan mapeadas interrupciones dependiendo de el estado en el que se encuentra el sensor, en este modo, configuraremos que se lance una interrupción siempre que el sensor tenga el bit *AWAKE* del registro de status a 1, indicando que se ha detectado movimiento.

Escribimos 0x0E en el registro ADXL_POWER_CTL para empezar a medir, además de activar la función *Autosleep.*

7.5.3 Resultados

En primer lugar, cuando seleccionamos el modo en cuestión, apreciamos lo siguiente:

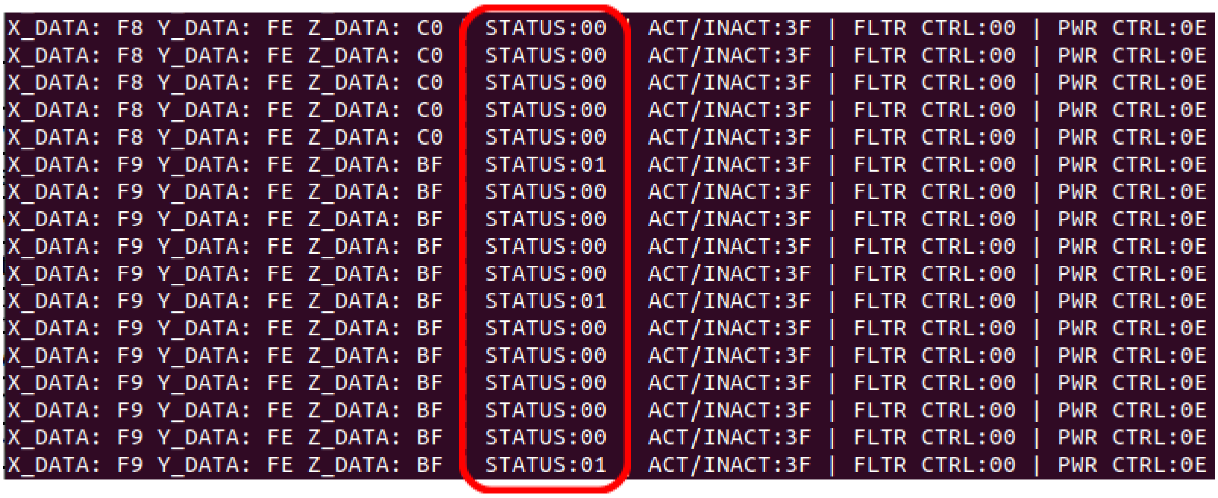


Figura 35: Muestra 1 del modo autonomous motion switch

El registro status varía entre valores 00 y 01 cuando la placa está en reposo, este comportamiento se debe a el bit 0 del registro de estado, el bit *DATA_Ready*, que indica que una nueva muestra está disponible para ser leída, y, cuando se hace la lectura, este bit se vuelve a poner a 0.

También es importante destacar que, el registro vale 00 o 01 ya que el bit *AWAKE* del registro status no se encuentra activo, es decir, la placa reconoce que está estacionaria, en un estado de inactividad.

Ahora nos disponemos a probar la detección de actividad y de inactividad. Según la configuración elegida y explicada en el punto anterior, el sensor debería registrar un estado de actividad cuando las muestras que mide rebasan el umbral de actividad seleccionado, en este caso, de 250mg. Para probar esto, podemos agitar la placa.

Si hacemos esto, observaremos el siguiente comportamiento:

```
X_DATA: F8 Y_DATA: FD Z_DATA: BE | STATUS:00 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: F8 Y_DATA: FD Z_DATA: BE | STATUS:00 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: F8 Y_DATA: FD Z_DATA: BE | STATUS:00 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: F8 Y_DATA: FD Z_DATA: BE | STATUS:00 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: F8 Y_DATA: FD Z_DATA: BE | STATUS:00 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: FC Y_DATA: E1 Z_DATA: B3 | STATUS:41 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: FC Y_DATA: E1 Z_DATA: B3 | STATUS:40 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: FC Y_DATA: E1 Z_DATA: B3 | STATUS:40 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: FC Y_DATA: E1 Z_DATA: B3 | STATUS:40 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: FC Y_DATA: E1 Z_DATA: B3 | STATUS:40 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: F8 Y_DATA: F7 Z_DATA: CB | STATUS:41 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
X_DATA: F8 Y_DATA: F7 Z_DATA: CB | STATUS:40 | ACT/INACT:3F | FLTR CTRL:00 | PWR CTRL:0E
```

Figura 36: Muestra 2 del modo *autonomous motion switch*, agitando la placa

El movimiento empieza cuando el valor del registro *status* cambia de 00/01 a 40/41.
Cuando esto ocurre, podemos hacer las siguientes observaciones:

- La primera es que el bit *DATA_READY* sigue funcionando como lo hacía en la figura anterior, cambiando cíclicamente entre 0 y 1.

- La segunda y más obvia, es que el valor del registro status pasa de valer 00 o 01 a valer 40 o 41. Esto es debido a que el bit *AWAKE* del registro de estado se ha activado. Esto significa que el sensor ha detectado el movimiento (*Activity event*) , y, según el funcionamiento del *loop mode*, ha activado el bit *AWAKE*.

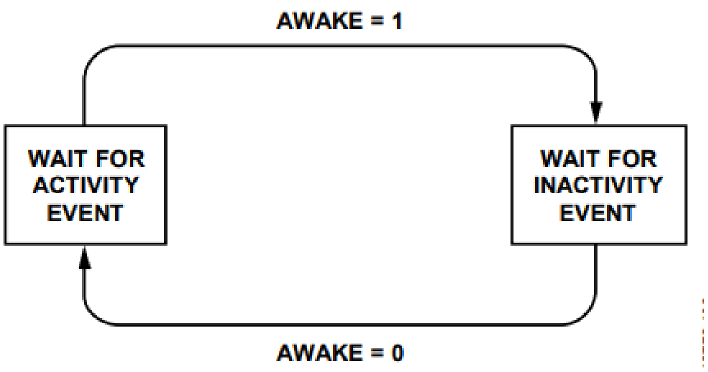


Figura 37: Diagrama del funcionamiento del modo *Loop*

Una vez hemos dejado de mover la placa, observaremos el siguiente comportamiento: El bit *AWAKE* se mantendrá activo un total de 5 segundos, pasado este tiempo, volverá a desactivarse, y el registro estatus volverá a valer 00 o 01.

Este tiempo está marcado por el umbral de inactividad y por el timer de inactividad. Según lo configurado, la inactividad será detectada siempre y cuando las muestras tomadas a lo largo del tiempo especificado por el timer de inactividad sean menores que el umbral de inactividad, en este caso, medidas menores a 150mg durante 5 segundos. Estos resultados indican que el modo funciona como debería hacerlo.

7.6 Autonomous Motion Switch Mode Custom

Se ha diseñado un modo configurable a partir del modo *Autonomous Motion Switch* que puede ser configurado por el usuario vía inputs en consola. Los parámetros que pueden ser configurados son el rango g, el ODR, el umbral de inactividad, el timer de inactividad y el umbral de actividad. El funcionamiento es el mismo que el *Free Fall custom mode*. El código completo de los modos *custom* están disponibles en github.

8. Conclusiones

8.1 Conocimientos adquiridos y desarrollo completado

Durante este trabajo se ha tenido que familiarizar con la placa Nexys 4 DDR y su FPGA, la Artix-7.

El estudio de la biblioteca GRLIB y sus numerosos ip cores ha sido vital para este trabajo, reconociendo el potencial de reutilizar los mismos.

El uso de XIlinx Vivado para crear el bitfile del LEON3 y volcarlo en la placa.

LEON3, de arquitectura SPARC 8V, ha sido el procesador que ha sido utilizado para ejecutar la aplicación programada en C.

Mediante el compilador cruzado BCC, la aplicación desarrollada ha sido compilada para poder ser ejecutada en procesadores de arquitectura SPARC.

Usando el debugger GRMON, se ha cargado el programa compilado en la placa y se ha ejecutado, permitiéndonos ver el funcionamiento de este.

El estudio del datasheet del sensor ADXL362 ha permitido comprender las distintas capacidades del mismo.

Todos estos conocimientos han permitido el desarrollo, la implementación y la validación de un driver que usa el sensor ADXL362 para medir aceleraciones tanto estáticas como dinámicas haciendo uso de diferentes modos de funcionamiento implementados.

8.2. Futuras líneas de trabajo

Una vez entendido el funcionamiento de tanto el sensor ADLX362 como de todos los componentes necesarios para ello, el próximo paso sería implementar este driver en un una aplicación de uso más avanzada que hiciera uso del mismo, como introducir circuitería adicional cuyo funcionamiento fuera activarse según si el modo *Autonomous Motion Switch* detectara movimiento o no.

9. Bibliografía

[1] Nexys 4 DDR Reference Manual - digilent.com

[2] GRLIB IP Core User's Manual - gaisler.com

[3]GRLIB IP Library User's Manual - gaisler.com

[4] ADXL362 Datasheet – analog.com

[5] Xilinx Vivado Archive - xilinx.com

[6] LEON Bare-C Cross Compilation System (BCC) - gaisler.com

[7] GRMON3 User's Manual - gaisler.com

[8]Milica Mitic and Mile Stojcev, "An Overview of On-Chip Buses", Faculty of Electronic Engineering, University of Niš. Serbia,2006.

[9] An introduction to AMBA AXI - developer.arm.com

[10]Bogdan Gabriel Voicila,"Cifrado de memoria en un procesador Leon-3", Grado en Ingeniería de Computadores, Universidad Complutense de Madrid.

[11]Micro electromechanical systems (MEMS) MESA - sandia.gov

[12] Código usado para este proyecto – github.com

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR