Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica
Intensificación en Robótica y Automatización

Emulación Hardware de un robot implementado en una FPGA bajo la filosofía de Rapid Control Prototyping (RCP)

Autor: Luis Romero Ben
Tutor: Eduardo Galván Díez

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica
Intensificación en Robótica y Automatización

Emulación Hardware de un robot implementado en una FPGA bajo la filosofía de Rapid Control Prototyping (RCP)

Autor:

Luis Romero Ben

Tutor:

Eduardo Galván Díez

Catedrático

Dep. de Ingeniería Electrónica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2016

Resumen

Este proyecto trata numerosos aspectos relacionados con los conocimientos adquiridos en el grado, entre otros la electrónica analógica y digital, la simulación mediante el entorno de programación visual MATLAB® Simulink o la programación hardware de FPGA usando el lenguaje VHDL.

Además, se han utilizado recursos completamente nuevos en cuanto a contacto previo, tales como la creación de mundos virtuales empleando el lenguaje de programación VRML, el empleo del hardware y software de ROBOTIS® (controladora, servos, cableado), etc.

Todo esto en conjunto ha permitido simular los tres servos de un robot manipulador virtual, para comprobar y verificar el correcto funcionamiento de la controladora OpenCM9.04 de ROBOTIS.

Abstract

This project deals with several fields of engineering which students work on at my degree, as analogue and digital electronics, systems simulation employing graphical programming environments like MATLAB® Simulink or FPGA hardware programming using VHDL.

Moreover, I have used completely new resources regarding previous contact, such as developing virtual worlds by VRML programming language or using ROBOTIS® hardware and software (controllers, servos, wiring), etc.

All these aspects together have allow me to simulate three servos belonging to a virtual manipulator robot, in order to check the proper functioning of the ROBOTIS® OpenCM9.04 controller.

Índice

ÍNDICE DE FIGURAS

1 MEMORIA DESCRIPTIVA

No reneguemos del pasado.

- Leonardo Da Vinci -

Para comprender el desarrollo del proyecto, es decir, el motivo que subyace a la organización de trabajo seguida, debemos, en primer lugar, entender las razones que me han llevado a elegirlo entre otras opciones, ya que éstas son básicas para entender los objetivos que se persiguen.

Por otro lado, será necesario conocer los antecedentes de la idea del proyecto, destacando trabajos previos en el mismo tema y herramientas actuales que implementan las funcionalidades que se buscan.

Por último, habrá que exponer de forma clara y concisa los objetivos del proyecto, el alcance que se persigue y los detalles que se esperan para el resultado final.

1.1. Introducción

Durante el período de selección de Trabajo Fin de Grado, entre las ofertas expuestas, fui dando prioridad a aquellos que cumplieran una serie de requisitos:

- En primer lugar, el proyecto debía permitirme aplicar el mayor número de aspectos de la ingeniería electrónica y robótica posibles. No buscaba un trabajo fácil y rápido, sino aquél que resumiera aquellos conocimientos que he ido adquiriendo durante el grado y me permitiera explotarlos lo máximo posible (dentro del tiempo disponible).
- Por otra parte, tampoco quería ocuparme exclusivamente de recursos en los que ya hubiera trabajado. Aquellos proyectos que me ofrecieran nuevas posibilidades de aprendizaje también aumentaban en preferencia.
- Por último, buscaba que el proyecto tuviera un interés real, es decir, que su ampliación permitiera elaborar una herramienta efectiva y útil para el campo de la ingeniería que cubriese.

Así pues, finalmente opté por hacer una solicitud para realizar este proyecto, el cual reúne y cumple todos los requerimientos que buscaba.

Una vez comencé a trabajar, fui consciente de la gran oportunidad que el trabajo me brindaba con respecto al tercer requisito antes expuesto. Por ello, me puse dos objetivos principales, muy relacionados pero independientes a su manera:

- Primero, tomar un controlador comercial (en mi caso de la marca ROBOTIS®), y elaborar un

entorno que permitiera simular los servos de la misma casa electrónica con el objeto de chequear que el controlador funcione de forma correcta, antes de probarlo en un robot real. Así, podrían realizarse simulaciones fuera de línea que dieran lugar a grandes ventajas para fabricantes y consumidores.

- En segundo lugar, realizar una guía del proceso a seguir para crear un simulador propio para la controladora que necesite el usuario, con el objetivo de que sea capaz de repetir los pasos sin los múltiples problemas que surgen al hacerlo por primera vez.

Con respecto al primer objetivo cabe mencionar que se corresponde con uno de los puntos clave del proyecto.

Es cierto que las grandes empresas del ámbito de la robótica disponen en su mayoría de sus propios softwares de simulación, que dan a sus clientes con una licencia para que puedan trabajar con los robots de la compañía fuera de línea, y comprobar el correcto funcionamiento de sus controladoras, programa, etc.

No obstante, para pequeñas empresas, investigadores e incluso estudiantes, esto no es fácilmente realizable: el diseño e implementación completos de un software de simulación robótica no es asequible para estos grupos.

Por ello mi trabajo puede ser beneficioso para proporcionar las líneas a seguir con objeto de llevar a cabo simulaciones que justifiquen la correcta actividad del controlador diseñado.

Por otro lado, estas simulaciones pueden ser muy útiles para reducir el denominado ***time to market***: de tener una herramienta especialmente orientada a simular el funcionamiento que tendrá un dispositivo, podrían realizarse pruebas exhaustivas mucho más aclaratorias que aquellas que pudieran intentar llevarse a cabo sin ella, ahorrando tiempo de producción.

Por todo esto, pienso que este trabajo ha constituido un reto interesante e instructivo (tanto en conocimientos de ingeniería como de esfuerzo en general), y espero que pueda ser continuado para, finalmente, dar lugar a una herramienta útil para el desarrollo del mundo de la robótica.

1.2. Antecedentes

Como hemos comentado en la introducción, algunas de las grandes compañías en el campo de la robótica industrial tienen sus propios simuladores para probar sus robots de forma virtual, tanto para emplearlos como herramienta en sus propias fábricas, como para facilitárselo a sus clientes con el objetivo de que ellos también puedan disfrutar de las ventajas de una simulación previa al uso real del controlador o el robot.

En mi caso, tuve contacto con uno de estos softwares de simulación durante el cuatrimestre anterior. En una de las asignaturas empleamos el programa **RobotStudio**, propiedad de una de las empresas líderes en automatización e ingeniería eléctrica, **ABB®** (2016).

La misma empresa, en su página web, presenta el software diciendo que es "como tener un robot de verdad en el ordenador". Esta afirmación se parece bastante al objetivo final de este proyecto, si hablamos de una forma ambiciosa.

Siguiendo con la presentación de **RobotStudio**, comentan que la programación fuera de línea es "la mejor manera de maximizar la rentabilidad de inversión de los sistemas de robots". De nuevo se repiten las ideas que hemos comentado en la introducción: vemos como en el ámbito de la robótica se considera la posibilidad de tener un simulador virtual de nuestro robot como una gran ventaja.

En cuanto a mi experiencia personal con el software, debo mencionar que, a pesar de ciertas incompatibilidades y errores, funcionaba perfectamente e implementaba todas las funciones deseables para que podamos emplear el simulador con total confianza de su similitud con el robot real.

Aunque de forma más modesta, el objetivo de este proyecto va un poco más allá de lo que significa un entorno de simulación fuera de línea, ya que permite probar en tiempo real no solamente nuevo software desarrollado por el usuario, sino que también permite probar nuevos diseños hardware,

controladoras, sistemas de comunicaciones, servos, sensores y otras piezas de un robot sin necesidad de hacerlo sobre el sistema real. El alcance del proyecto no consiste sólo en una emulación software del robot, sino que abarca también la emulación hardware, usando los interfaces de comunicación y potencia que usaría el robot en la realidad. Este nuevo concepto de simulación se denomina Hardware In the Loop (HIL).

Este sería el objetivo a largo alcance del proyecto, dar lugar a una herramienta de estas características. No obstante, como todo proyecto tiene su principio, mi trabajo será el inicio de ese esfuerzo, que de ser continuado, podrá dar lugar a una aplicación útil de simulación de robots.

Cabe mencionar otras compañías importantes dentro del campo de la robótica que disponen de software de simulación propio: **KUKA®**, con su programa **KUKA.Sim**; o **FANUC®**, con el software de la familia de productos **ROBOGUIDE**, destacando el **ROBOGUIDE-HandlingPROn**.

A su vez, existen simuladores robóticos de empresas menos conocidas, o incluso de compañías dedicadas por completo a la simulación virtual. En este grupo destaca **V-REP** (Virtual Robot Experimetation Platform), perteneciente a la empresa **Coppelia Robotics**. De todos los softwares que he investigado para informarme sobre los antecedentes en el campo de la simulación robótica, este es uno de los que mejores me han parecido, principalmente por su versatilidad (los programas para sus controladores pueden escribirse en C/C++, Python, Java, Matlab u Octave; cada objeto del diseño puede controlarse de forma individual a través de *scripts* internos, nodos de ROS, soluciones *custom*, etc.) (Coppelia Robotics, s.f.).

Por último, entre los simuladores virtuales de libre distribución (**V-REP** sólo es gratuito para estudiantes) cabe destacar **BrazoRobot**, que permite simular en 3D un brazo robótico.

Sin embargo, si atendemos a todos los ejemplos de simuladores expuestos, ninguno permite una característica básica de este proyecto: la comunicación en tiempo real con un controlador externo. En los sistemas anteriormente citados, el hecho de simular también el controlador facilita ciertamente dicha comunicación, al no tener en cuenta los problemas que surgen comparando una implementación virtual con una física. Además, esto no permitiría depurar los problemas que pueden surgir en el futuro en un nuevo diseño de la propia controladora, o del sistema de comunicaciones u otros elementos hardware que se quieran probar en este sistema HIL.

En resumen, estos softwares podrán servirnos se inspiración en cuanto a la dificultad y al trabajo que les subyace, y a la gran cantidad de funcionalidades que implementan, llegando a ser casi tan "realistas" como el sistema físico.

1.3. Objetivos y alcance

Por último, como paso previo a comenzar a analizar el contenido del proyecto, debemos marcar los objetivos que se busca cumplir, de una forma más precisa y extensa. Como dijimos en la introducción, tenemos dos objetivos principales:

- En primer lugar, se plantea diseñar un simulador de un robot virtual que se comunique con una microcontroladora externa, para chequear el correcto funcionamiento de la misma. Para ello, debemos definir una serie de detalles:

    - **El robot**: tendrá 3 grados de libertad (GDL), con tres articulaciones rotativas independientes. Será diseñado empleando el lenguaje de programación de entornos virtuales 3D, VRML. Esto se hará desde MATLAB®, por lo que la simulación del mismo se llevará a cabo mediante el entorno de programación visual Simulink®.
    - **La FPGA**: se empleará la placa ZedBoard, debido a sus buenas características en relación a MATLAB®, ya que la empresa distribuidora la anuncia como una de las FPGA más adecuadas para trabajar en conjunto con ese software (ZedBoard(a), s.f). Esta se empleará para simular, en primer lugar, los nexos de comunicación y decodificación de datos enviados por la controladora a MATLAB®. Posteriormente, en caso de disponer de

tiempo y recursos, se aumentará el grado de complejidad de la simulación de cada servo del robot virtual, y del robot en su conjunto.

- **La microcontroladora**: como ya se ha especificado, se usará el modelo OpenCM9.04 de la empresa ROBOTIS®, debido a su disponibilidad. Se diseñarán programas para probar las funcionalidades más importantes de la microcontroladora. Una vez estas hayan sido conseguidas, se intentará aumentar la complejidad hasta donde sea posible. Esta se alimentará con una fuente de alimentación externa, empleando para ello entre 8 y 10V, dependiendo de lo que requiera la tarjeta.
- **La comunicación**: ya que la microcontroladora emplea el protocolo Dynamixel, se implementarán mediante la FPGA disponible los programas necesarios para simular dicho protocolo y extraer la información de los mensajes de la OpenCM9.04.
- **El software**: para la simulación del robot virtual y el control de los procesos que ocurran en la FPGA se empleará System Generator for DSP™ para MATLAB R2014b, aprovechando las características que ofrece de trabajo con las herramientas de Xilinx®. En cuanto a éstas, se usará principalmente ISE Design Suite 14.7, con el objetivo de diseñar programas que aprovechen la parte de lógica programable de la ZedBoard. Para programar la microcontroladora se usará la IDLE de ROBOTIS®, suministrada por la propia compañía.

- Como segundo objetivo principal, se desea crear un proyecto que sirva de guía a aquellas personas que, o bien quieran continuar y ampliar este proyecto, o que deseen iniciarse en alguna de las herramientas que se emplearán en este proyecto. Así pues, para cumplir con este objetivo deberé:

  - Relatar el proceso de diseño, implementación y comprobación de resultados de todos los pasos implicados en la creación de mi simulador virtual.
  - Explicar los fallos que me vayan surgiendo: tanto la causa que lo provoca como las posibles soluciones que pueden llevarse a cabo para arreglar el problema.
  - No escatimar detalle en las explicaciones y profundizar en los aspectos que puedan implicar una pérdida de tiempo del lector en caso de no haberle quedado suficientemente claros.
  - Analizar las ventajas y desventajas de las implementaciones que vaya realizando, con tal de que el lector tenga sopese las razones que pueden llevarle a usar una herramienta u otra.
  - En caso de no poder completar alguna tarea, intentar proponer e implementar varias soluciones, con el objetivo de mostrar las distintas posibilidades para elegir durante la implementación.

Además de estos dos objetivos, tal y como dije en la introducción, podemos extraer la meta a largo plazo del proyecto, que es la creación de un simulador de características similares a los antes expuestos como ejemplos, pero con la particularidad de estar orientado a la prueba de controladores externos.

Las razones que avalan esta motivación son:

- Reducir el ***time to market*** (este es el tiempo que transcurre desde que un producto es concebido hasta que sale finalmente al mercado para el público) de los productos relacionados con el ámbito de la robótica siguiendo la filosofía de *rapid prototyping systems*, de forma que tanto grandes como pequeñas empresas pudieran beneficiarse de este método para abaratar costes y tiempo en el diseño, producción y testeo de sus dispositivos.
- Beneficiar además a los consumidores particulares, brindándoles la posibilidad de probar sus propios controladores sin la necesidad de pagar una cuantía extra a alguna compañía para obtener un simulador. Además, la guía les permitiría implementar un simulador adaptado a sus necesidades, ahorrando tiempo en los testeos de sus controladores.

2 MEMORIA JUSTIFICATIVA

Nuestra recompensa se encuentra en el esfuerzo y no en el resultado. Un esfuerzo total es una victoria completa.

- Mahatma Gandhi -

Durante la realización de este proyecto entré en contacto con multitud de recursos, dispositivos y lenguajes de programación, por lo que antes de empezar a relatar y analizar mi labor en cada uno de los distintos campos que componen el trabajo, haré una breve introducción de cada uno de ellos, explayándome más en los que tengan un empleo menos común, con objeto de familiarizar al lector con los aspectos tratados y facilitar la comprensión de los mismos.

Tras esto, procederé a explicar el recorrido del proyecto: programas realizados, trabajo y aprendizaje sobre los distintos dispositivos empleados y sus características, pruebas acometidas y resultados de las mismas, etc. Recomiendo encarecidamente **leer este capítulo completo**, ya que los problemas y soluciones se irán comentando en **orden cronológico**.

2.1. Descripción de los elementos constituyentes del proyecto

2.1.1. ZedBoard

La ZedBoard es el núcleo del proyecto, alrededor del cual cobran sentido los demás elementos del mismo, y que actúa como nexo entre el dispositivo que deseamos testear (la controladora), y el mundo virtual que hace las veces de banco de pruebas.

La Zedboard es una placa de evaluación y desarrollo basada en Xilinx® XC7Z020-1CLG484C Zynq™ - 7000 AP SoC (System on Chip), y como el resto de placas de esta familia, una de sus características más atractivas es que integra la facilidad de programación software de los procesadores ARM® (en el caso concreto de la Zedboard disponemos de un sistema de procesado Corex-A9) con la facilidad de programación

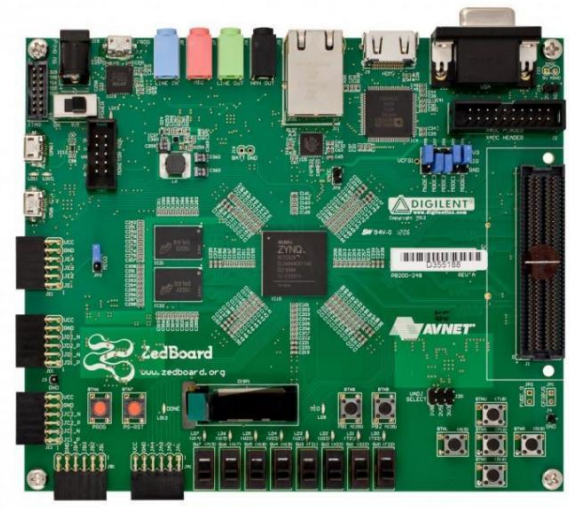


Figura 1 (fuente: ZedBoard (c), s.f.)

hardware de una FPGA (disfrutando de 85000 celdas Series-7 PL (Programmable Logic)), por lo que puede ser empleada ampliamente en numerosas y diferentes aplicaciones. Ambas partes, PS (processing system) y PL (programable logic) se comunican mediante una interfaz de alto ancho de banda basada en la especificación ARM AMBA® AXI (ZedBoard(b), 2014).

La extensibilidad de la placa la hace ideal para el prototipado rápido y el desarrollo de Pruebas de Concepto (PoC[1]).

Otro de los aspectos que la hacen destacar es su conexión con MATLAB®. Avnet Electronics, empresa diseñadora de la Zedboard, colabora con MathWorks para ofrecer un amplio abanico de herramientas optimizadas para la implementación y simulación de algoritmos usando un diseño basado en modelo, reduciendo tiempos y errores comunes en el desarrollo de código VHDL a mano (ZedBoard, s.f.).

No obstante, volveremos sobre el punto de la relación ZedBoard – MATLAB® más adelante, ya que la realización de este proyecto me ha permitido apreciar mejor cuán cerca de la realidad se encuentran las afirmaciones anteriormente aseveradas.

2.1.2. OpenCM9.04 (ROBOTIS®)

Constituye el dispositivo cuyo funcionamiento deseamos testear, para así comprobar de forma fehaciente que su diseño (en caso de que el controlador lo hayamos fabricado nosotros) es completamente operativo y cumple nuestras especificaciones y expectativas.

El OpenCM9.04 es una tarjeta microcontroladora basada en un procesador ARM Cortex-M3 de 32 bits. Este microcontrolador ofrece multitud de recursos a los usuarios: 26 pines de GPIO para usarlos como entradas o salidas digitales (además, algunos pines implementan otras funcionalidades como PWM, SPI o USART), 4 bloques de 5 pines a los que conectar numerosos tipos de sensores, 1 bloque de 4 pines usados para comunicación con otros periféricos… (ROBOTIS(a), s.f).

Figura 2 (fuente: ROBOTIS(a), s.f.)

Otro punto a favor de esta tarjeta microcontroladora es que tanto sus esquemáticos como librerías son *open-source*, lo que facilita el trabajo de investigación sobre la misma, al no tener límites en la comprensión de su funcionamiento impuestos por la compañía.

No obstante, el aspecto o recurso más interesante de esta tarjeta es que dispone de dos Dynamixel TTL Bus, empleados para la comunicación con los servomotores de la misma empresa, ROBOTIS®. Estos pines serán los que explotaremos con el objetivo de establecer una comunicación entre el

COMENTARIOS

[1]Prueba de concepto: implementación, a menudo resumida o incompleta, de un método o idea, realizada con el propósito de verificar que el concepto o teoría en cuestión es susceptible de ser explotada de forma útil (Wikipedia, 2014).

microcontrolador y el robot virtual que crearemos.

Tal y como indica su nombre, este sistema emplea el protocolo Dynamixel, propio de ROBOTIS®. Convendría explicar sus principales características, ya que será un punto clave en nuestro trabajo.

2.1.2.1. Dynamixel

Dynamixel implementa una Comunicación Serie Asíncrona de 8 bits, 1 bit de Stop y sin paridad, en forma de *Half Duplex UART*. Se usa este método (en el que transmisión y lectura no pueden ser ejecutadas simultáneamente por el mismo elemento) ya que Dynamixel está preparado para que un solo controlador pueda trabajar con varios servos, implementando una **Daisy Chain**, de forma que el microcontrolador se conecta a un servo, éste al siguiente, y así. Por ello, mientras uno de los elementos de la cadena transmite datos, el resto deben encontrarse en modo de lectura.

Otro punto importante que implica el empleo de este tipo de comunicación, y que Dynamixel implementa, es que cada servo conectado a la cadena debe disponer de una ID única, para que cuando el controlador envíe una serie de datos, aquél servo que los reciba compruebe la ID de destino, y de no coincidir con la suya, reenvíe los datos al siguiente en la cadena. En caso de que dos servos tuvieran la misma ID, la red acabaría teniendo problemas (ROBOTIS(b), s.f.).

En cuanto a cómo se envían los datos, el microcontrolador y los servos se intercambian información en forma de Paquetes. Existen dos tipos de Paquetes: de Instrucciones y de Estado (ROBOTIS(c), 2006).

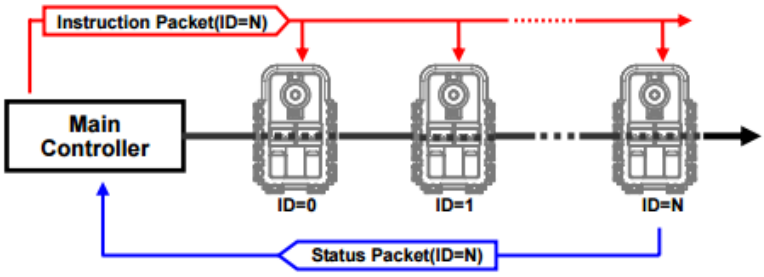


Figura 3 (fuente: ROBOTIS(c), 2006)

- Paquetes de Instrucciones: son aquellos que el microcontrolador envía al servo que corresponda. Su estructura es como sigue:

0xFF	0xFF	ID	LENGTH	INSTRUCTION	PARAMETER 1	…	PARAMETER N	CHECKSUM

  - Primero, dos bytes 0xFF indican el comienzo del Paquete.
  - Justo después, se envía la ID del servomotor destinatario de la información, comprendida entre 0 y 253 (el 254 se usa para *broadcast*).
  - Tras esto, se manda la longitud del Paquete, la cual se calcula como "Número de parámetros + 2"
  - La instrucción a realizar constituye el siguiente Paquete. Hay 7 en total, destacando algunas como PING, READ_DATA, WRITE_DATA, etc. Cada una tiene un byte identificativo propio.
  - Los parámetros son los datos extras que necesita conocer el servo para una determinada instrucción.
  - Por último llega la suma de comprobación, que sigue la siguiente fórmula:

$$CheckSum = \sim (ID + Length + Instruction + Parameter\ 1 + \cdots + Parameter\ N)$$

Siendo ~ el operador lógico NOT.

- Paquetes de Estado: son aquellos que el servo envía al microcontrolador como respuesta. Está constituido por:

0xFF	0xFF	ID	LENGTH	ERROR	PARAMETER 1	…	PARAMETER N	CHECKSUM

  - Al igual que en los Paquetes de Instrucciones, dos bytes 0xFF indican el comienzo del Paquete.
  - Después se manda la ID del servomotor emisor de este Paquete.
  - La longitud se calcula igual que en el caso del tipo de Paquete anterior y se envía a continuación.
  - Posteriormente se manda un byte que indica si algún error ha ocurrido durante la operación de Dynamixel, existiendo 7 errores disponibles más un estado de "no error".
  - Los parámetros se usan para dar información adicional en caso necesario.
  - Y por último, de nuevo, llega el byte de la suma de comprobación, que se calcula de la misma forma que en el caso anterior, excepto que se cambia el byte de Instrucción por el de Error.

En cuanto a cómo gestiona el servo las instrucciones que le llegan y los estados que le son pedidos, cabe destacar que internamente éste dispone de una memoria, dividida en dos secciones: una parte es memoria RAM, cuyos valores se ponen a los de fábrica al ser apagado y encendido el servo, y una EEPROM, que los mantiene al ser no-volátil. Así, cada entrada de la tabla representa un elemento de información sobre el estado y operación del servomotor (modelo AX-12 en este caso): ID, posición actual, velocidad actual, posición objetivo, velocidad objetivo (estos últimos 4 ocupan 2 bytes cada uno), e incluso otros aspectos tales como el *baud rate* y la temperatura. El servomotor es operado escribiendo valores en el lugar adecuado de la memoria, también denominada **tabla de control**, y el estado se comprueba leyendo dichos valores.

2.1.3. VRML

El lenguaje VRML (Lenguaje de Modelización de Realidad Virtual) es un formato específico de archivos empleado para describir objetos y entornos interactivos en 3D. Nos permite representar objetos y escenas que a su vez podremos enlazar con otros mundos virtuales, o con otras herramientas presentes en nuestro sistema (Gámez, I., 2001).

Esto último es muy importante, ya que implica que muchos recursos software emplean VRML como lenguaje de representación 3D, entre ellos aquél que nos interesa: MATLAB®. Desde Simulink® podemos acceder a ciertas herramientas que nos facilitan el diseño en este lenguaje, como programas de diseño visual del tipo de V-Realm Builder, gracias a la librería de **Simulink 3D Animation**.

Estas facilidades se deben principalmente a que VRML es un lenguaje abierto y no requiere de ningún software propietario para poder generar los mundos virtuales, lo que no ocurre con otros programas que realizan modelado 3D.

Figura 4 (fuente: LaWebdelProgramador, s.f.)

En cuanto a los campos de aplicación, la realidad virtual está disfrutando un auge muy destacado en la industria, y es ahí donde este proyecto encuentra su hueco y aparece su utilidad. También destaca en otros sectores como arquitectura, medicina, juegos…

Por todas estas características hemos elegido VRML como el lenguaje a emplear para implementar el mundo 3D donde estará nuestro robot virtual.

2.1.4. Herramientas software

Para acometer este proyecto fue necesario el empleo de ciertos recursos informáticos con el objetivo de crear los programas precisos para cada tarea. Entre ellos destacan:

- MATLAB®: éste ya ha sido nombrado en el documento, ya que es parte esencial del proyecto. En su mayor parte se ha dado uso a su entorno de programación visual Simulink, sobre todo en la primera parte del trabajo. La versión empleada ha sido la R2014b, aunque recomiendo emplear al menos la versión R2015a, por razones que expondré más adelante.

- Xilinx® ISE Design Suite 14.7: empleado para el diseño hardware en VHDL de aquellos programas que estaban destinados a ser ejecutados en la ZedBoard. Además del ISE WebPack descargado, se empleó el System Generator for DSP™, que permite trabajar en MATLAB® con ciertos recursos de Xilinx, controlando los diseños implementados en nuestra FPGA mediante Simulink.

- ROBOTIS® IDLE: para poder cargar los programas creados para el control de los servos dentro de la tarjeta microcontroladora, ROBOTIS® nos facilita un entorno de programación propio pero basado en el de la plataforma Arduino.

- CadSoft EAGLE: este software permite diseñar diagramas y PCBs. En nuestro caso, se empleó con este fin debido a que era necesaria una placa de adaptación entre la OpenCM9.04 y la Zedboard, al ser los pines del Dynamixel TTL Bus de la microcontroladora bastante específicos. El diseño y funcionalidades de la misma se explicarán más adelante.

- Vivado® Design Suite 2015.2: este programa ha sido empleado para la función de Hardware In-The-Loop del HDL Workflow Advisor de Simulink, útil para crear proyectos en VHDL e implementar el .bit correspondiente en la FPGA directamente desde Simulink.

- PicoScope 6: este software permite el manejo desde el PC de los osciloscopios de la marca PicoScope. En mi caso, necesite emplearlo para realizar ciertas pruebas y comprobaciones de funcionamiento, que se verán más adelante. Utilicé el modelo 4824 de Pico Technology.

Figura 5

2.2. Recorrido de trabajo

2.2.1. Primeros pasos

En un primer momento, me dediqué a recopilar información acerca de los distintos aspectos del proyecto antes nombrados: la ZedBoard, la controladora de ROBOTIS, el software a utilizar y el VRML, que era totalmente nuevo para mí.

Sobre todo me interesé por recursos webs que hicieran las veces de tutoriales introductorios a estos,

para empezar aprendiendo los aspectos básicos de cada elemento constituyente del trabajo.

Así, comencé por un *webinar* denominado ***Run a Simulink Modelo on Zynq***, disponible en la web de MathWorks (Levine, N., 2-4, s.f.).

Este tutorial abarca desde los requerimientos necesarios para la tarea en cuestión hasta el control y funcionamiento de un ejemplo dado por MathWorks (*hdlcoder_blinking*).

En cursos anteriores ya había trabajado con herramientas de Xilinx, por lo que ya conocía las limitaciones que estas sufrían al correr en el SO Windows 8. Al tener este sistema operativo instalado, decidí probar el ejemplo, ya que anteriormente había encontrado formas de solucionar algunos de estos problemas.

No obstante, al intentar implementarlo, el programa fallaba y no podía seguir el último paso del tutorial: construir el modelo en Simulink® en modo externo, para ejecutar el programa en la ZedBoard y poder comunicarla con el archivo .slx.

Por esta razón, el primer paso de mi trabajo consistió en instalar Windows 7 en mi ordenador, ya que las herramientas de Xilinx® están optimizadas para el mismo. Para ello, sin entrar en detalles, creé una partición del disco duro e instalé el SO en ella, estando éste disponible de forma gratuita para los estudiantes de la Universidad de Sevilla. La parte más ardua de este proceso fue poner el sistema a punto, ya que mi equipo se lanzó al mercado con Windows 8, y los drivers para Windows 7, aunque están disponibles, no siempre funcionan correctamente al ser instalados. Esto era ciertamente problemático debido a la necesidad de conectar la ZedBoard y la OpenCM9.04, por lo que los drivers del puerto serie eran necesarios, pero no los encontraba por ninguna parte. No obstante, buscando información acerca de la ZedBoard, me fije que usaba el Cypress CY7C64225 USB-to-UART (ZedBoard(b), 2014), de forma que lo descargué e instalé, y conseguí conectarme con la FPGA. El caso de la microcontroladora fue más sencillo ya que el paquete que descargué de ROBOTIS® con la IDLE proveía los drivers del mismo.

Así pues, una vez instalados MATLAB® R2014b y Xilinx® ISE Design Suite 14.7, procedí a reintentar el ejemplo. De éste (Levine, N., 2-4, s.f.) extraje una serie de pasos necesarios para comenzar a trabajar con ZedBoard (también aplicable a otras tarjetas de la familia de Xilinx®):

- En primer lugar, no debe abrirse MATLAB® directamente, ya que trabajar así puede producir fallos durante el proceso de compilación del código a VHDL, y posteriormente para producir el proyecto cuyo .bit introduciremos en la ZedBoard. Lo óptimo es configurar el System Generator for DSP™, que Xilinx® nos facilita en su ISE WebPack, para que abra MATLAB®, ya que así no tendremos problemas de este tipo (al menos por esta razón).
- En cuanto a la ZedBoard, para introducir programas en ella debemos tener en cuenta la posición de los jumpers, que debe ser una determinada. Es importante comprobar esto para no perjudicar la placa.
- En MATLAB®, debemos instalar una serie de *Hardware Support Package*s, que nos facilitarán las herramientas necesarias para trabajar con Xilinx® desde MATLAB®. En concreto, debemos instalar los paquetes de Xilinx® Zynq – 7000, que es la familia de nuestra FPGA. Para esto nos pedirá una tarjeta SD en la que escribirá un firmware para el soporte de Linux que debemos introducir en la ZedBoard. Lo que hará será actualizar el firmware ya existente de la tarjeta que trae la placa. Un posible problema es que esta operación se aborte por un fallo con el comando **format**, explicando la *textbox* que éste "no se reconoce como un archivo interno o externo, programa o archivo por lotes ejecutable". Esto se debe a un problema con las variables de entorno, más concretamente con la variable **PATH**, que debe tener el siguiente valor: **%SystemRoot%\system32;%SystemRoot%;%SystemRoot%\System32\Wbem;**. El valor de otras variables puede consultarse en la bibliografía (Carrasco, J., 2012).
- Una vez configurados los paquetes de herramientas, debemos configurar el *path* de MATLAB para emplear ISE Design Suite 14.7 como herramienta predeterminada para la compilación y

síntesis en VHDL, mediante el comando ***hdlsetuptoolpath***: hdlsetuptoolpath('ToolName','Xilinx ISE','ToolPath','C:\Xilinx\14.7\ISE_DS\ISE\bin\nt64\ise.exe');

- También debemos configurar la conexión con el hardware de Zynq mediante los comandos ***z = zynq***, para crear un objeto de servicio de Linux que permitirá conectar MATLAB® con el SO de Linux implementado en la SD acoplada a la ZedBoard; y ***z = z .setupZynqHardware( )***, para configurar y establecer finalmente la conexión por puerto serie (aquí vemos la importancia que tenía configurar correctamente los drivers de puerto serie).
- En caso de que MATLAB® haya sido instalado recientemente y no se haya asociado un compilador de C con éste, debemos emplear el comando ***mex –setup*** para hacerlo.

Aquí quiero hacer un breve inciso[1]: aunque en el tutorial se dé por hecho que este paso no debe dar problemas, en el caso de haber instalado el SO desde cero es posible que Windows no traiga Microsoft SDK instalado, que es el programa que queremos establecer como compilador predeterminado en MATLAB®. En ese caso, hay que descargarlo (recomiendo la versión 7.1) e instalarlo, y puede ser que esto también nos de problemas. El principal motivo (fallo que me surgió a mí) es el hecho de instalar Microsoft SDK 7.1 después de haber instalado Microsoft Visual Studio 2010 SP1, ya que si se hace en ese orden, la instalación de SDK elimina los principales compiladores del sistema (MATLAB Answers™, 2013). Hay dos soluciones factibles:

- Instalar el parche que Microsoft lanzó para arreglar el problema.
- Eliminar los dos *2010 Redistribuible Packages* (x86 y x64) de Microsoft Visual Studio C++ 2010 SP1, y tras esto instalar SDK 7.1 y por último reinstalar los paquetes.

Si esto no funciona, es posible que el conflicto se esté dando entre SDK y .NET Framework 4.5. La solución en este caso es también desinstalar este último, instalar el SDK y reinstalar el primer programa.

- Por último, debemos configurar la herramienta MakeFile, de forma que los *build processes* funcionen correctamente. Para ello debemos ejecutar el comando ***xmakefilesetup***, lo que abrirá una pestaña de configuración de usuario de XMakeFile (observable en la Figura 6).

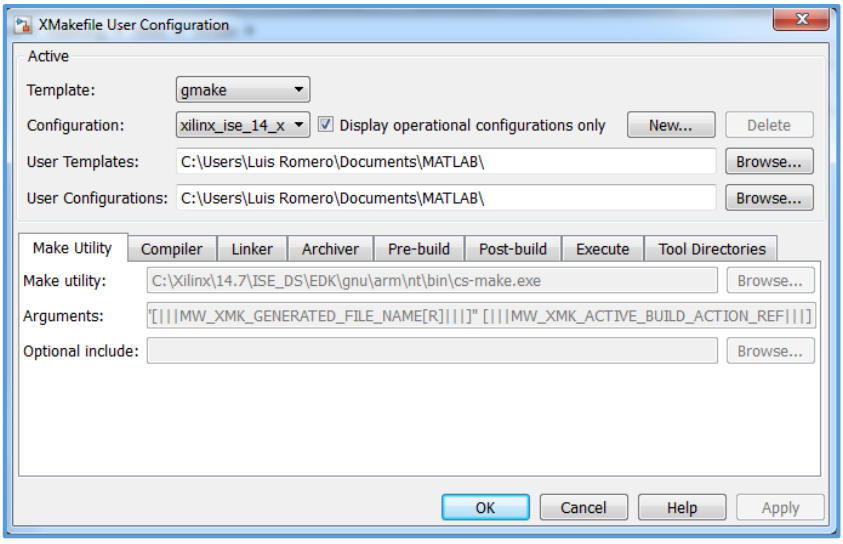


Figura 6

COMENTARIOS

[1]Quiero destacar que durante el desarrollo de la memoria iré incluyendo explicaciones como estas, ya que la primera causa de retraso en la evolución de trabajos como éste en los que se emplean multitud de herramientas nuevas son los fallos de instalación, configuración, etc…, que no siempre tienen una solución fácilmente implementable o localizable en la web. Así, con esto, pretendo reducir el tiempo perdido por quiénes empleen este trabajo como punto de partida para sus propios proyectos.

Para que ésta funcione para las herramientas de Xilinx®, en primer lugar deseleccionamos la opción *Display operational configurations only*, y buscamos entre las configuraciones disponibles. Si encontramos *xilinx_ise_14_x* la seleccionamos y en el campo *Make Utility* deberíamos ver algo como: C:\Xilinx\14.7\ISE_DS\EDK\gnu\arm\nt64\bin\cs-make.exe. En caso de no existir esta opción, debemos crear una nueva configuración y llamarla *xilinx_ise_14_x*. El *path* antes expuesto deberemos introducirlo nosotros.

Una vez seguidos todos estos pasos, ya estamos en disposición de correr nuestro programa dentro de la ZedBoard, trabajando desde MATLAB®.

Ya que estamos recorriendo el proceso en orden cronológico, pasemos a la siguiente sección trabajada tras este hito.

2.2.2. Diseño de PCB en Eagle

Debido a que los pines que emplean tanto la controladora OpenCM9.04 como los servos de Dynamixel son un tanto especiales, y a la necesidad de incluir ciertas funcionalidades de cara a protección o a una futura continuación del trabajo, decidí diseñar una PCB de adaptación entre la microcontroladora y los GPIO de la ZedBoard.

Así pues, el primer paso sería investigar sobre los pines de Dynamixel y los puertos de la FPGA, con el fin de diseñar la placa correctamente:

- 3 Dynamixel TTL Bus: antes que nada, debemos conocer la naturaleza y características de cada pin (ROBOTIS(a), s.f.).

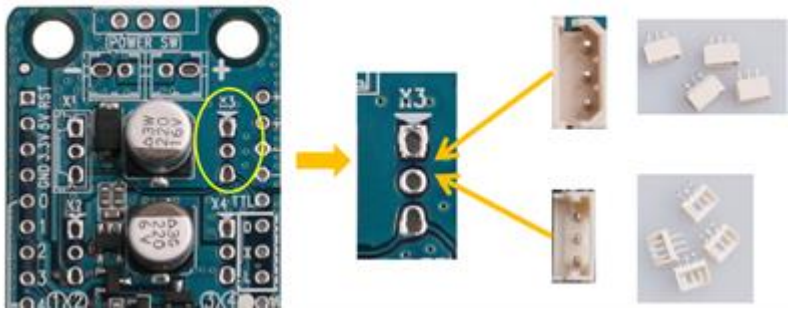


Figura 7 (fuente: ROBOTIS(a), s.f.).

Empezando por arriba, los pines se disponen en GND, VCC y DATA.
VCC será el voltaje con el que alimentemos la OpenCM9.04. Es importante tener en cuenta que los Dynamixel TTL Buses no pueden usarse si la microcontroladora es alimentada exclusivamente mediante el cable USB empleado para subir los programas a ésta. Por ello, en caso de usar servos reales, debemos alimentar la OpenCM9.04 de forma independiente al USB, y con un voltaje y amperaje suficientes para los servos que conectemos (el valor puede oscilar entre los 9 V y 11 V: menos puede no ser suficiente, y más puede ser perjudicial para el sistema. Además, se debe tener en cuenta que la controladora admite un máximo de 16 V). En la Figura 8 observamos las tres formas de alimentar la OpenCM9.04. En mi caso, he optado por alimentarla por la entrada de las baterías (1) con una fuente de tensión.

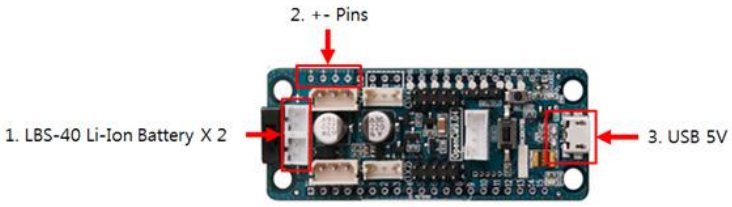


Figura 8 (fuente: ROBOTIS(a), s.f.).

DATA será el canal empleado para transmitir los bits de información al servo. Se puede apreciar como la salida de este pin a nivel alto es 5 V (TTL). Este canal constituirá el bus bidireccional de la UART.

- ZedBoard GPIOs: nuestra FPGA dispone de cinco Digilent Pmod™, que constituyen los únicos GPIO de la placa. De estos cinco, sólo cuatro corresponden a la parte de *Programmable Logic* de la ZedBoard, que es la que nos interesa para trabajar directamente con los puertos desde el software de Xilinx. Además, de estos cuatro restantes nos será más interesante trabajar con el JA1 y el JB1, ya que el JC1 y el JD1 trabajan de forma dual y tienen sus I/O conectadas de forma diferencial. Así pues, también disponen de una salida de 3.3 V y GND. Investigando la documentación de la placa, observamos que los Pmod™ están conectados al *bank* 13, que funciona a 3.3 V (ZedBoard(b), 2014).

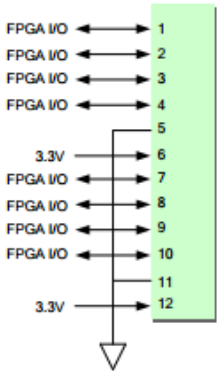


Figura 9 (fuente: ZedBoard(b), 2014)

Tras estos dos análisis apreciamos que surge un inconveniente a la hora de conectar ambos dispositivos: funcionan a voltajes diferentes. Por ello, una de las primeras cosas a tener en cuenta a la hora de diseñar la placa de adaptación sería que debemos convertir el voltaje de los 5 V que llegan de la controladora a los 3.3 V que deben entrar a la FPGA.

Sin embargo, en vez de incluir esta fase de conversión dentro de la placa, decidí adquirir un convertidor lógico de 5 a 3.3 V, ya que está especialmente diseñado para el propósito que perseguimos.

En mi caso, adquirí en concreto el modelo *Sparkfun Logic Level Converter* que vemos en la Figura 10.

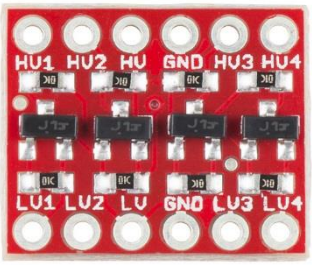


Figura 10 (fuente: TechMake Electronics, s.f.)

Para funcionar, éste sólo necesita ser conectado a 5 V en el pin de HV y a la tierra del dispositivo que trabaja a este voltaje en su GND respectivo, y a 3.3 V en LV e igual con la tierra pero para este dispositivo. Así, HV1/LV1 y HV4/LV4 servirían para pasar de 3.3 V a 5 V, y los restantes (que son los que nos interesan), para la operación inversa (Front&Back, 2014).

Así pues, tras encontrar esta solución, sigamos con el diseño de la PCB. Para que la explicación sea más dinámica, iré haciendo referencia a la Figura 11 (presente en la página siguiente) que representa el diseño final de la PCB, de forma que se entenderán mejor los pasos seguidos hasta llegar a ese punto.

- Los 3 pines marcados llamados **JP1** se corresponden al conector del Dynamixel TTL Bus. Empezando por arriba, el primer pin es GND. Aunque en el resto de elementos del diseño también hay varias conexiones a tierra, estas se generan al emplear la función *Rastnest* del software de EAGLE, empleado para reducir la cantidad de cobre que los líquidos empleados en la creación de la PCB física tienen que eliminar. Al emplear esta herramienta, todos los pines conectados a GND se unen al pin que estamos tratando.
  El segundo pin es VCC. Ya que la alimentación del servo real debe estar entre 9 V y 11 V, tomamos el valor ideal que el fabricante recomienda: 9.6 V (ZedBoard(b), 2014). Así pues, en la placa debemos adaptar esta tensión a 3.3V. No obstante, como después alimentaremos la controladora con un voltaje menor, el diseño de este aspecto es orientativo: se persigue obtener un voltaje inferior a 3,41 V, el máximo que soportan los Pmod™ de la ZedBoard. Para ello, se ha diseñado un divisor resistivo con una resistencia de 3k3 Ω y otra de 6k8 Ω (en la Figura 11 se aprecia un valor de 6k3 Ω, pero los valores serigrafiados en la imagen son orientativos). De esta forma el voltaje se reduce hasta el límite necesario. Además, se observa como se ha incluido un diodo zener a modo de garantía, para preservar los 3,3 V que constituyen su tensión de ruptura.

El tercer pin es DATA. Por éste se envían los bits que conforman los bytes de los paquetes de comunicación de Dynamixel, y tal y como comenté anteriormente, este protocolo se implementa mediante una UART, de forma que el canal se encuentra a nivel alto mientras nadie escribe en él.

Aunque puede implementarse desde el software de Xilinx® mediante una resistencia de pull-up "virtual", he optado por incluir una resistencia de pull-up física en la placa de adaptación, conectada por un lado a alimentación TTL (5V) y por otro al canal de datos[1].

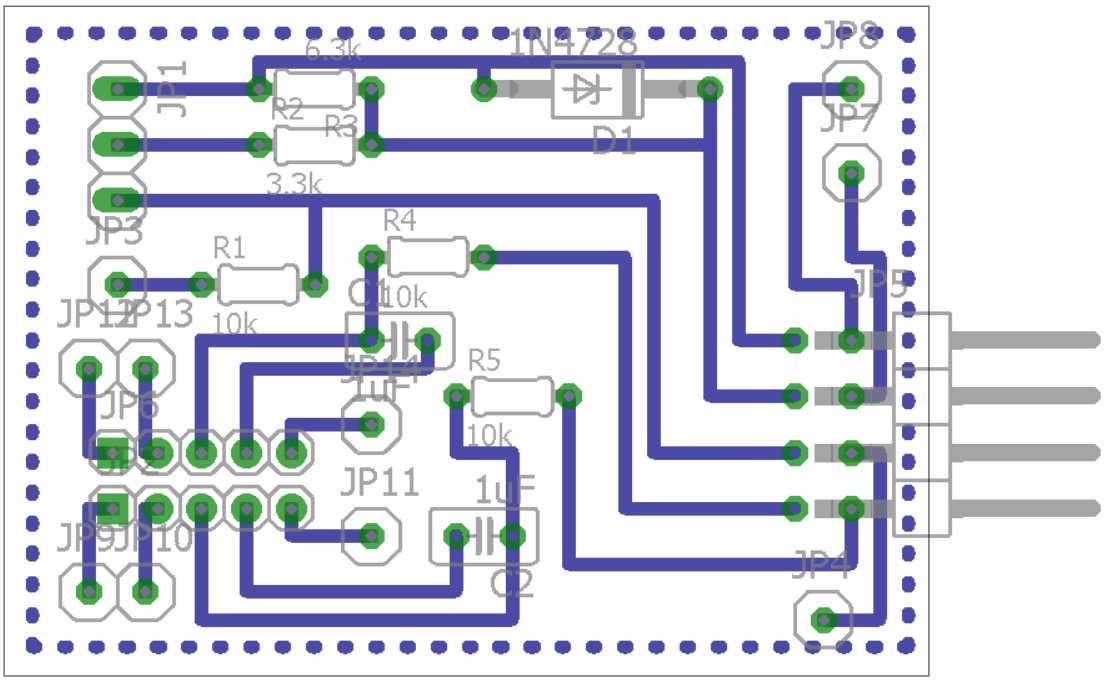


Figura 11

Esta es la parte básica de la PCB, ya que nos permite la comunicación entre la OpenCM9.04 y la ZedBoard en cuanto a envío de paquetes se refiere (para ver el esquemático, ir al **Anexo III**).

- Como parte de ampliación al proyecto se incluyó otra funcionalidad en la placa, a pesar de que ésta pudiera quedar inexplotada por falta de tiempo. Se trataba de la inclusión de sensores virtuales en nuestro mundo 3D, que leyeran, por ejemplo, la distancia del brazo robótico a un obstáculo y la enviasen a la controladora, de forma que ésta pudiera tomar decisiones en función del conocimiento que adquiere de esta medida, y por ende, del entorno que rodea al robot en el mundo virtual.
  Sin entrar en detalles sobre cómo se implementarían estos sensores en MATLAB® y/o VHDL en la ZedBoard (ya que de esto hablaremos más adelante), se incluyó la posibilidad de emplear dos sensores. En el *pinblock* acodado de la derecha de la Figura 11, que se conectaría a la ZedBoard, tenemos dos pines que constituyen salidas de la misma, y que transmitirían la información del estado del robot en el mundo virtual. La función de las resistencias y condensadores (agrupados en parejas como puede apreciarse para dar lugar a un

Figura 12 (modificado a partir de ROBOTIS(d), 2013).

COMENTARIOS

[1]El motivo para incluir esta resistencia conectada a 5V, a pesar de haber sido explicado, quedará aclarado por completo posteriormente, en el apartado 2.2.6.2.1.5 donde se explica el desarrollo de la UART para implementarla en la FPGA.

filtro RC en configuración paso bajo) será explicada nuevo más adelante, cuando se analice el código VHDL implementado para dar lugar a la salida de los sensores en la FPGA. Es más interesante centrarnos en el bloque de 2x5 pines al que se conectan. Éste sigue la estructura del **5-pin Port** (Figura 12) incluido en la controladora OpenCM9.04, y que a su vez es el mismo del que disponen los dispositivos 5-pin de ROBOTIS, los cuales son en su mayoría sensores. Así pues, si analizamos la Figura 11 en conjunción con la Figura 12, se aprecia cómo se introduce el valor de tensión proporcional al valor del sensor en el ADC, y GND se conecta a la tierra global (antes explicada). El resto de pines se dejan libres, por si se necesita usarlos en aplicaciones futuras.

- Por último, mencionar que se dejan tres pines, **JP4**, **JP7** y **JP8**, para propósito general, con posibilidad de conectarlos a los puertos de la ZedBoard en el caso de que se necesite.

Así pues, tras el diseño que hemos analizado anteriormente, el siguiente paso era evidente: trasladar el esquemático en EAGLE a la PCB física.

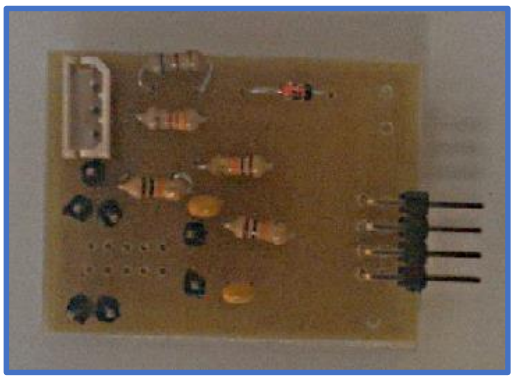


Figura 13

No explicaré aquí el proceso empleado para crearla ya que es bien conocido y está ampliamente documentado, aunque si destacaré la importancia de la experiencia de aprender este procedimiento, contando con un arma más en mi arsenal como ingeniero para resolver problemas.

Finalmente, destacar el resultado final, que puede apreciarse en la Figura 13. Puede observarse que los pines de propósito general y el *pinblock* de 2x5 para los sensores no han sido soldados. Esto se debe a que posiblemente no me daría tiempo a implementar su funcionalidad, y de no ser así, lo soldaría en un futuro.

2.2.3. Diseño de pruebas (I)

Una parte importante del proceso seguido en este trabajo ha consistido en diseñar distintos bancos de pruebas para analizar y comprender el funcionamiento concreto y real de Dynamixel, independientemente de los datos ofrecidos por el fabricante de la controladora en fase de pruebas, en este caso, la OpenCM9.04 de ROBOTIS®. Por ello, he dividido en varias secciones este arduo trabajo.

En esta primera parte el objetivo era transmitir los paquetes enviados por la controladora a MATLAB® para observar la exactitud de la descripción del protocolo, y aclarar los conceptos necesarios para desarrollar la UART propiamente dicha e implementarla en VHDL.

Por ello, y basándome en los primeros aspectos de la relación MATLAB® - Xilinx® que expuse anteriormente, comencé el diseño de un archivo Simulink® que posteriormente convertiría en un proyecto de ISE, para así controlar y observar lo que ocurre en la ZedBoard en estas simulaciones.

2.2.3.1. Xilinx Blockset

Para comprender el proceso de diseño que MATLAB® y Xilinx® ponen a nuestra disposición mediante System Generator for DSP™, es indispensable conocer las herramientas que éste y los *Hardware Support Packages* de Xilinx® nos ofrecen.

La más importante es el **Xilinx Blockset**, que junto con el **Xilinx Reference Blockset**, conforman lo que se conocen como **System Generator Blocksets**.

Xilinx Blockset (Xilinx, 2009) es una familia de librerías disponibles para Simulink®, que contiene bloques tanto de bajo nivel, proporcionando acceso a hardware específico de algunos dispositivos, como de alto nivel, implementando funciones de comunicación, control lógico, conversión entre tipos de datos, procesamiento digital de señales, matemáticas, memoria, etc.

Además, incorpora bloques de herramientas adicionales, que permiten entre otras cosas generar código VHDL, co-simulación VHDL o la estimación de recursos a emplear.

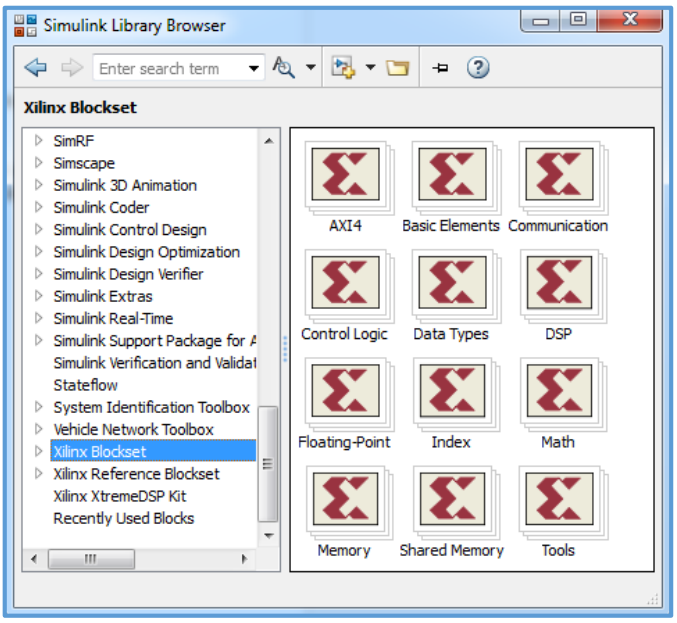


Figura 14

Estos bloques fueron básicos en el diseño del primer banco de pruebas que estamos analizando, y que podemos observar en el **Anexo II: Simulink**. Para entender el diseño, primero hablaré de los bloques que lo conforman, para posteriormente analizar el funcionamiento del conjunto.

- *Black Box*: constituye el núcleo del programa de Simulink®. Este bloque permite incorporar una "caja negra" en VHDL al diseño realizado con System Generator for DSP™, de forma que es posible crear un proyecto VHDL empleando ISE Design Suite e incluirlo en nuestro archivo de Simulink®. Cuando la *black box* es convertida a VHDL, su entidad es automáticamente "cableada" a los demás bloques del diseño, de forma que esta se agrega y conjunta correctamente con estos.
  Tal y como podemos apreciar, ésta es una herramienta muy potente para aquellos que dominan la programación VHDL pero son novatos en el manejo del System Generator, ya que pueden desenvolverse con las herramientas de Xilinx® que controlan; pero también para programadores que busquen funcionalidades que el **Xilinx Blockset** no pueda ofrecer, ya que el número de bloques de ésta es irremediablemente limitado y no puede abarcar todas las funciones posibles.
  Tras investigar sobre el empleo del bloque *Black Box*, y emplearlo en mis diseños, he reunido cierta información importante sobre éste que debe tenerse muy en cuenta a la hora de incluir dicho bloque en nuestros programas (System Generator Help, s.f.):

  - Si nuestro diseño en VHDL incluye algún puerto bidireccional, este no será mostrado como puerto en el System Generator. No obstante, esto no es un problema, ya que el puerto será incluido en el proyecto VHDL una vez realicemos el *nestlisting*.

- En el caso de *Black Box* para VHDL (también está disponible para Verilog), sólo se admiten puertos del tipo **STD_LOGIC** o **STD_LOGIC_VECTOR** (en este último caso, los bits deben estar ordenados desde el MSB al LSB). Es importante tenerlo en cuenta para que no surjan problemas durante la creación del proyecto.
- En cuanto a las fuentes de reloj y los *enables* de éstas, estos son considerados como puertos especiales y deben cumplir algunas reglas: tienen que ser de tipo **STD_LOGIC**, el nombre del *enable* debe ser el mismo que tiene el reloj, cambiando *clk* por *ce* (clock enable).

Se requiere tener estos requisitos en cuenta a la hora de diseñar el proyecto VHDL a introducir en la caja negra, ya que asegurando su cumplimiento podemos ahorrarnos fallos impensados que ralentizan el normal desarrollo de un proyecto.
Otro aspecto básico que conviene tener en mente, más importante si cabe que los anteriores, es que se debe suministrar cierta información sobre el código VHDL a introducir en la *Black Box* para poder incluirla en System Generator. Esto se hace mediante una MATLAB® *function*, que describe la interfaz y la implementación de la caja negra. Más concretamente, esta *M-function* realiza tareas como especificar el nombre de la entidad del *top-level* que introducimos en *Black Box*, seleccionar el lenguaje, describir puertos (sus tipos y tasas de datos), otros archivos necesarios para el proyecto, etc.

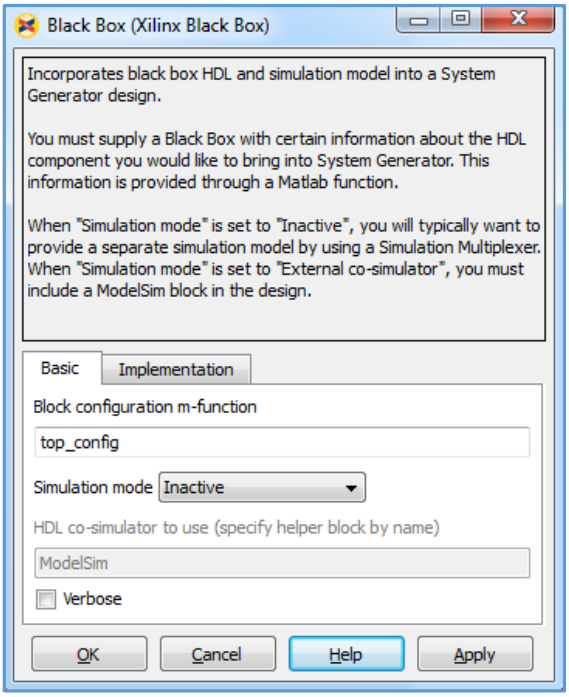


Figura 15

De los problemas que pueden surgir, uno de los más comunes es que el diseño VHDL a acoplar a nuestro Simulink® tenga lo que se conoce como *combinational path* (un ejemplo de esto es una conexión directa entre una entrada y una salida). En ese caso, es imprescindible indicarlo en la *M-function* mediante: `this_block.tagAsCombinational;`
Esta línea siempre aparece en el archivo (la función nos la crea el asistente de la *Black Box*, pero puede y **debe** modificarse para satisfacer las necesidades de nuestro diseño), pero se insta a quién emplee algún *combinational path* en su diseño a que no comente la línea. Además, en caso de tener el proyecto fuente archivos adicionales debe indicarse en este .m.
El último aspecto a tener en cuenta del bloque *Black Box* es la posibilidad de emplear una co-simulación[1] HDL entre Simulink® y un segundo simulador HDL. Para este último elemento encontramos dos opciones: emplear el simulador de la herramienta de Xilinx® asociada a nuestro MATLAB® (ISE Simulator para ISE Design Suite y Vivado Simulator para Vivado Design Suite), o un co-simulador externo conocido como ModelSim. Podemos apreciar como en la Figura 15 la opción marcada es *Inactive*. En caso de hacer esto, Simulink® hará cargo de la simulación al completo. Volveremos sobre el software de ModelSim cuando analicemos su bloque de Simulink, que podemos observar en el archivo en análisis.

COMENTARIOS

[1]La co-simulación de System Generator se encarga de iniciar el simulador HDL, generar VHDL adicional si es necesario, compilar el HDL existente, configurar los eventos de la simulación, manejar el intercambio de datos entre Simulink® y el simulador HDL, etc…

- *System Generator Token*: sirve como un "panel de control" para chequear el sistema y los parámetros de simulación. Este bloque es de uso obligatorio en caso de usar cualquier otro perteneciente al **Xilinx Blockset**. Éste permite indicar cómo queremos que se lleven a cabo la generación de código y la simulación mediante la selección de una serie de parámetros, entre los cuáles sobresalen los siguientes (deben establecerse según nuestras necesidades) (System Generator Help, s.f.):

Figura 16

  - *Compilation*: permite seleccionar el tipo de compilación que el *code generator* de HDL llevará a cabo. Las más importantes son *HDL Netlist Compilation*, que es el compilador por defecto, *Bitstream Compilation*, que permite compilar el diseño en un archivo de configuración *bitstream*, que podremos implementar sin problemas en nuestra FPGA; y *Hardware Co-simulation Compilation*, para compilar el diseño con una tarjeta concreta como objetivo, para realizar una co-simulación mediante Simulink® y otro simulador como explicamos antes. Así pues, este punto debe especificarse con cuidado dependiendo de cómo queramos trabajar con la placa de Xilinx®.
  - *Part*: especifica la placa a emplear. Para el caso de la ZedBoard, no se especifica con este nombre, sino que se la llama ZC702, ya que la ZedBoard pertenece a esta familia.
  - *Hardware Description Language*: puede elegirse VHDL o Verilog.
  - *Synthesis Tool*: permite elegir entre varias opciones, aunque yo recomiendo **Xilinx XST**.

  No obstante, no sólo importa la pestaña de compilación, sino que también debemos comprobar las características especificadas en la pestaña de temporización y ajustarlas:

  - *FPGA Clock Period*: se especifica el periodo del reloj de nuestra placa. En el caso de la ZedBoard, éste es de 10 ns (para la parte de PL).
  - *Clock Pin Location*: debemos introducir el pin de la placa asociado al reloj. Para la ZedBoard, este es el 'Y9'.
  - *Simulink System Period*: define el periodo de del sistema en Simulink®, en segundos. Este nos es aconsejado por el propio diseño al compilarlo, ya que cada bloque del **Xilinx Blockset** que introduzcamos tiene sus propias tasas de datos (que por otra parte pueden modificarse). Éste valor suele ser la menor de estas tasas, pero no tiene porque. Es cuestión del diseñador decidir el valor que más se adapte a lo que busca.

  Así pues, se aprecia la importancia de la correcta configuración de este bloque.

- ModelSim: tal y como comentamos anteriormente, en caso de que queramos emplear ModelSim como co-simulador junto con Simulink, es necesario incluir este bloque en nuestro diseño. No es necesario configurar nada en él, pero sí que debemos instalar el software de ModelSim[1].

Figura 17

  Como ventajas de éste podemos destacar que su bloque de Simulink® se encarga de crear el HDL necesario para que el contenido de la *Black Box* pueda simularse en ModelSim, abre directamente una sesión en este programa al empezar la simulación, comunica Simulink® con ModelSim y reporta posibles errores al compilar la caja negra. Todo esto, unido a la extensión de uso que tiene el software, hace de ModelSim una herramienta recomendable para la co-simulación.
  No obstante, tras probar el software como parte de mi trabajo de investigación, debo decir que si el programador es nuevo en el empleo de ModelSim, aun a pesar de tener experiencia con

COMENTARIOS

[1]Puede parecer obvio, pero es cierto que en la mayor parte de la documentación de ayuda de MATLAB® sobre las librerías de Simulink® para Xilinx, no se comenta nada acerca de que ModelSim sea externo a éste y, por tanto, deba instalarse algún software extra.

lenguajes HDL, no es recomendable, ya que implica un cierto tiempo de adaptación al software, y esto no es un problema si se simula en modo "inactivo" o incluso con el simulador de la herramienta Xilinx® a usar.

- *Gateway In* / *Gateway Out*: ambos bloques son necesarios para la correcta comunicación entre Simulink® y el programa ejecutándose en nuestra FPGA. Puede decirse que son el límite entre la parte del diseño implementada en MATLAB y la que se implementa en la ZedBoard, y se emplean principalmente para convertir los datos de unos tipos a otros (System Generator Help, s.f.):

    - *Gateway In*: se coloca tras los puertos de entrada al subsistema que implementaremos en la ZedBoard. Los datos en formato de tipo Simulink® (entero, doble, punto fijo) pueden ser convertidos a booleanos, punto fijo o punto flotante. En el caso de punto fijo, se nos permite elegir el tipo de aritmética usada (con o sin singo), el número de bits de precisión, la cuantización y la acción tras desborde. Así podremos introducir los datos a nuestro diseño de la forma más conveniente. El bloque también permite seleccionar el periodo de muestreo. Conviene ajustarlo teniendo en cuenta el periodo o latencia de otros bloques del diseño.
    - *Gateway Out*: se coloca antes de los puertos de salida del subsistema que implementaremos en la FPGA. También nos permite seleccionar entre varias opciones de funcionamiento, aunque recomiendo dejarlos por defecto, o propagar el tipo de salida.

Los bloques que siguen también forman parte de este banco de pruebas, por lo que procederé a explicarlos, pero será muy breve, ya que a pesar de ser bloques muy comunes y útiles, no son obligatorios para cualquier diseño (*Black Box* y ModelSim tampoco lo son, pero la primera es cuasi-necesaria en proyectos de cierta complejidad, y el segundo bloque es importante para entender al 100% las posibilidades de un bloque *Black Box*):

- *Parallel to Serial* / *Serial to Parallel*: estos bloques tienen una función que su nombre bien indica: convertir conjuntos de bits que le llegan en paralelo a una única salida serie, y viceversa, respectivamente (System Generator Help, s.f.).

    - *Parallel to Serial*: permite elegir cuál será el orden de la salida (empezando por el MSB o por el LSB), la precisión de bits de salida, etc. Es importante tener en cuenta la latencia, que define el número de periodos de reloj que la salida es retrasada. Debe ser analizado porque un periodo de reloj en Simulink® puede coincidir con varios periodos de reloj en la FPGA.
    - *Serial to Parallel*: tiene las mismas opciones de ajuste que el bloque anterior. Además permite elegir el ancho de bits que saldrán en paralelo del bloque.

    Comentar que en mi caso, empleo esto bloques para convertir los bytes que quiera transmitir a bits en serie (*Parallel to Serial)* y para agrupar los bits que reciba de la controladora en bytes (*Serial to Parallel*).

- *Convert*: es el bloque de Xilinx® que permite convertir de unos tipos de datos a otros, pero a diferencia que las *Gateway*, que convertían de tipo Simulink® a tipo Xilinx®, ésta conversión es entre tipos de datos de Xilinx®.
  A pesar de que esta funcionalidad es ciertamente útil, en muchos diseños se emplean con otro objetivo: introducir un cierto retraso en alguna señal, mediante la latencia que se le dé a este bloque. Por ello no es raro ver que el tipo de entrada y salida de este bloque sea el mismo.

Así pues, aunque en las explicaciones de alguno de los bloques he introducido su función en mi diseño, analicemos su funcionamiento como conjunto:

- Dentro del bloque *Black Box* hay un programa en VHDL cuya funcionalidad es bien sencilla: coger los datos que recibe por el puerto bidireccional DATA, de tipo **STD_LOGIC**, es decir, de

un bit, y sacarlos como salida por RXD. Además, los bytes enviados desde Simulink® se introducen por el puerto TXD con el objetivo de enviarlos a la controladora.

- El *System Generator Token* ha sido configurado para crear un bitstream, y la placa ZedBoard, obviamente. Los valores de temporización son los anteriormente comentados.
- Los bytes que entran por el puerto TXD del subsistema llegan a la *Gateway In*, convirtiéndose a datos de tipo Xilinx. Se hacen pasar por un *Converter* para obtener un retraso de un periodo de reloj de Simulink® en la salida de estos datos (esto se hizo por problemas que Simulink® indicaba en la temporización), y se pasan al *Parallel to Serial* para que vaya mandando bits al puerto TXD de *Black Box*. Así es como funcionaría el envío.
- Los bits que salen del puerto RXD de *Black Box* llegan al *Serial to Parallel*, agrupándose en bytes. Estos se convierten a datos de Simulink® con *Gateway Out*. Así sería la recepción.
- En cuanto al resto de entradas al subsistema, DutyCycle y DIR_PORT, tienen la siguiente funcionalidad:

    - DutyCycle: tiene que ver con cómo se simulan los sensores de los que hablamos anteriormente, que medían el estado del robot en el mundo virtual. Podemos ver cómo esta simulación se implementará creando un PWM en la FPGA con salida al exterior, y esta entrada a la FPGA le dará el valor de *dutycycle* al PWM.
    - DIR_PORT: se emplea para seleccionar la dirección del puerto bidireccional DATA. Esto se verá de forma más extensa posteriormente, cuando se expliquen los programas en VHDL.

- En cuanto al resto de salidas del subsistema, GND_conected y VDD_conected, servirían para detectar si la controladora estaba conectada y funcionando o no.

Cabe destacar que para esta primera prueba se pretendía únicamente llevar a cabo la recepción de datos, tal y como dijimos anteriormente, ya que se implementó para conocer la naturaleza del protocolo de forma precisa. Además, en el programa VHDL implementado en la *Black Box* se dejaron sin salida los puertos de GND y VDD, que se emplearían en el futuro de ser exitosa la prueba.

2.2.3.2. Subsistema final

La capa superior del archivo Simulink® sería tal y como podemos ver en la figura siguiente:

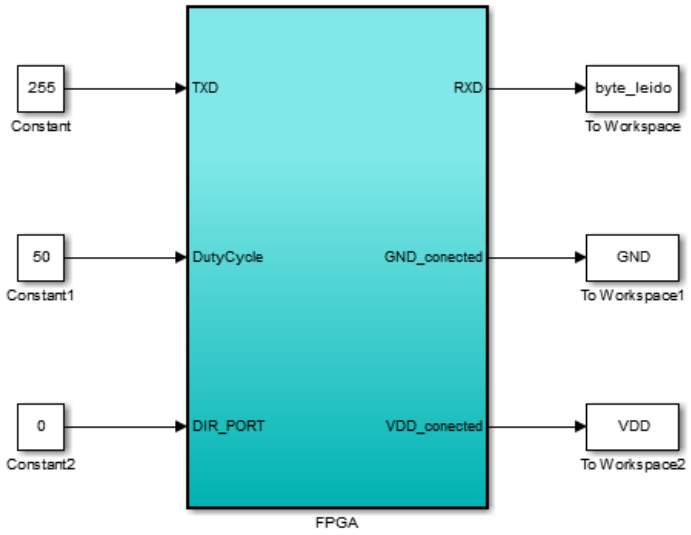


Figura 18

Es importante destacar que el subsistema debe ser configurado para tratarlo como una "unidad atómica". Esto se observa por el borde negro que tiene el subsistema. Para configurarlo, debemos entrar en sus propiedades de bloque y marcar la casilla correspondiente.

Esto sirve para que Simulink® trate al subsistema como a una unidad cuando esté determinando el orden de ejecución de los métodos de simulación de los bloques interiores al subsistema. Para entenderlo algo mejor, un ejemplo de esto sería que, para calcular la salida del subsistema, Simulink® invocaría los métodos de salida de todos los bloques del subsistema antes de invocar el método de salida de otros bloques de la misma capa que dicho subsistema. Esto es necesario para el correcto funcionamiento de la herramienta de la que hablaremos a continuación (MATLAB® Help, s.f.).

Cabe mencionar que todo lo que se encuentra englobado por el subsistema será compilado y convertido a VHDL, e implementado en la FPGA, ejecutándose allí, mientras que los bloques que empleamos fuera de este subsistema serán simulados en Simulink®.

2.2.4. Implementación de pruebas (I): HDL Workflow Advisor

En el tutorial que se comentó al principio del trabajo, una vez se tenía el diseño listo, se pasaba a implementar dicho diseño en la FPGA (Levine, N., 4-4, s.f.). Para ello, se contaba con una herramienta ciertamente útil, que posiblemente merecería una sección propia, ya que para aquellos que comenzamos a trabajar con System Generator for DSP™, proporciona una gran ayuda para, paso a paso, convertir el subsistema que hemos diseñado en Simulink® en un proyecto en VHDL implementable en FPGA, que podremos controlar y verificar desde el propio Simulink®. No obstante, dividiremos la explicación precisa de la herramienta en partes, al igual que hicimos con el diseño de las pruebas, ya que en este punto del trabajo sólo pensaba seguir los pasos del tutorial y crear un IP Core de nuestro subsistema. El resto de funcionalidades se nombrarán, pero sólo se explicarán en posteriores secciones aquéllas útiles para proyectos como éste.

Así pues, para comenzar el proceso de implementación del diseño, lanzamos el **HDL Workflow Advisor** (click derecho en el subsistema / HDL Code / HDL Workflow Advisor). Tras una serie de ajustes de inicialización de MATLAB®, nos aparecerá una ventana como la siguiente:

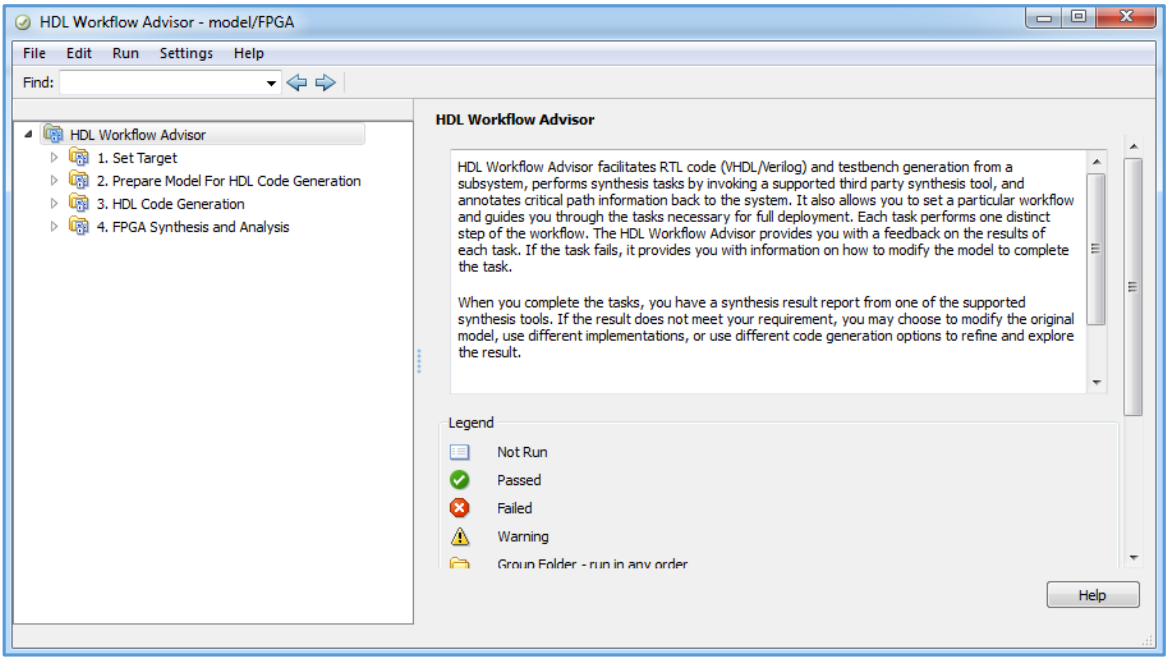


Figura 19

Esta es la pestaña básica de **HDL Workflow Advisor**. Brevemente, ésta es una herramienta que da soporte para cubrir las distintas etapas del proceso de diseño para FPGA mediante una serie de tareas. Algunas de éstas comprenden validación o comprobación del modelo, y otras se encargan de emplear el *HDL code generator* de MATLAB® u otras herramientas software de terceros (Xilinx® normalmente). La potencia de **HDL Workflow Advisor**, además de en lo dicho anteriormente, se aprecia cada vez que se ejecuta una tarea. La herramienta nos informa acerca del resultado de dicha tarea, y además, en caso de que haya fallado (y el error esté contemplado entre aquellos que la herramienta reconoce), nos indica como modificar nuestro modelo y sus parámetros para solucionar el problema y completar la tarea.

Volviendo a la Figura 19, vemos que existen cuatro carpetas en el nivel más alto de **HDL Workflow Advisor**. Cada una de ellas contiene una serie de tareas relacionadas. No obstante, es importante saber que las tareas y carpetas pueden cambiar dependiendo de la funcionalidad que queramos darle a la FPGA y al programa que implementemos en ella. En este apartado analizaremos las carpetas y tareas a cubrir para generar un IP Core, explicando para que sirven y el resultado que dan. Finalmente, llegaremos a conclusiones acerca de este uso de **HDL Workflow Advisor**.

Así pues, comencemos describiendo las carpetas y sus tareas internas (MATLAB® Help, s.f.):

- ***Set Target***: las tareas de esta categoría permiten, entre otras cosas, seleccionar la placa a emplear, la funcionalidad a implementar, o mapear las I/O de la placa a los puertos de tu modelo. Dispone de dos tareas internas:

    - *Set Target Device and Synthesis Tool*: nos permite ajustar los siguientes parámetros:

        - *Target Workflow*: se nos permite elegir entre los distintos *workflow* que **HDL Workflow Advisor** proporciona, es decir, las distintas formas de implementar y controlar el programa ejecutándose en la FPGA. Se nos dan varias opciones, pero como dijimos antes, para esta prueba escogimos **IP Core Generation**.
        - *Target Platform*: debemos elegir la placa que usaremos. En nuestro caso, ZedBoard. Esto hará que los campos de *Family, Device, Package* y *Speed* queden seleccionados para los valores propios de ZedBoard.
        - *Synthesis Tool*: como en los primeros pasos configuramos el *path* para Xilinx® ISE Design Suite 14.7, la herramienta que nos aparecerá es Xilinx® ISE. Podemos agregar Vivado como opción ajustando el *path* a esta herramienta.
        - *Project Folder*: especificamos el nombre de la carpeta donde guardaremos los resultados de compilación. Recomiendo elegir un *path* que no tenga espacios en blanco, por ejemplo, que no sea una carpeta que cuelga del Escritorio, ya que esto dará error.

    - *Set Target Inteface*: en esta parte es donde seleccionamos la conexión entre puertos, GPIO y buses de la FPGA con los puertos de nuestro subsistema. Para ello disponemos una tabla que, tras haber ejecutado a tarea anterior, contendrá los nombres de los puertos de nuestro subsistema. Para cada uno de ellos, distinguiendo entre entradas y salidas, se nos darán una serie de opciones, que serán específicas para la placa que seleccionáramos anteriormente, tales como GPIO, buses AXI-Lite, AXI o LEDs (solo salidas). En nuestro caso nos interesarían los puertos Pmod™ de la ZedBoard y el bus AXI-Lite para la comunicación MATLAB® - FPGA. Los pines de los puertos Pmod™, que debemos introducir aquí, pueden encontrarse en la ***ZedBoard User's Guide*** (ZedBoard(b), 2014). En la Figura 20 podemos apreciar el esquema de implementación seguido en el caso de IP Core, empleando el puerto AXI4 para comunicaciones.
      De esta tarea también debemos mencionar la posibilidad de cambiar el modo de sincronización entre la FPGA y el procesador de nuestro PC. Entre las opciones tenemos *Free running*, que no sincroniza automáticamente ambos elementos; o *Coprocessing*, existiendo dos modos para este último: que la FPGA sea más rápida que el procesador y

lo espere, o al revés. Estos dos últimos métodos son muy interesantes y potencialmente útiles, pero en mi caso sufrí de ciertos problemas usándolos, así que opté por *Free running*.

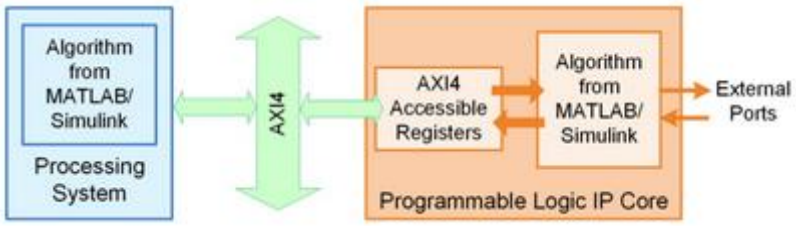


Figura 20 (fuente: MATLAB® Help, s.f.)

- ***Prepare Model for HDL Code Generation***: sus tareas internas comprueban la compatibilidad del modelo con la generación de código HDL. Así pues, en caso de haber bloques, ajustes, parámetros o condiciones indeseables para una correcta compilación, la herramienta nos informa y, casi siempre, nos indica cómo arreglarlo. Tiene cuatro tareas internas:

    - *Check Global Settings*: aquí se comprueban parámetros o ajustes generales que puedan ser un impedimento para la correcta generación del código. Ésta tarea suele ser útil ya que nos indica los parámetros potencialmente problemáticos, dándonos directamente la opción de modificarlos todos sin tener que salir de la herramienta. Esto es raro, ya que en el caso de otros errores debemos corregirnos manualmente.
    - *Check Algebraic Loops*: el HDL Coder™ de MATLAB® no puede ejecutarse correctamente si el modelo en cuestión tiene "bucles algebraicos". Estos ocurren cuando existe un bucle en alguna señal de un bloque de forma que su salida depende de su entrada, pero también de la propia salida. En la Figura 21 podemos ver un ejemplo de un *algebraic loop*. Esta tarea del **HDL Workflow Advisor** localiza estos bucles y avisa de que su existencia impide la generación de código, por lo que deben encontrarse y solucionarse manualmente.

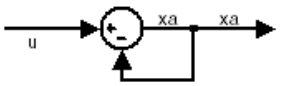


Figura 21 (fuente: MathWorks Documentation, s.f.)

    - *Check Block Compatibility*: revisa la compatibilidad de los bloques presentes en tu modelo con HDL Coder™, ya que de no ser compatibles, salta el aviso de error.
    - *Check Sample Times*: comprueba la compatibilidad de la generación HDL con el tiempo de muestro escogido, el *solver* seleccionado, el modo de ejecución de tareas… De nuevo, chequea estos aspectos e informa de errores, si los hay.

- ***HDL Code Generation***: en esta carpeta encontramos dos tareas internas, teniendo una de ellas dos subtareas:

    - *Set Code Generation options*: las subtareas de esta tarea permiten ajustar y validar los parámetros de generación de código HDL, y de *testbenches*, en caso necesario:

        - *Set Basic options:* se ajustan parámetros que afectan a la codificación HDL en general, seleccionando el lenguaje (VHDL o Verilog) y que informes se desean al finalizar la generación de código HDL.
        - *Set Advanced options*: permite ajustar parámetros que afectan específicamente a características concretas de la generación HDL, tales como fuente de reloj o reset, y extensiones de archivos generados.

    - *Generate RTL Code and IP Core*: aquí se ajustan los últimos parámetros del IP Core a generar, destacando su nombre, versión y carpeta de destino Además, en caso de estar usando una *Black Box*, en el campo *Additional source files* deberemos añadir los nombres

de aquellos archivos del proyecto ejecutado por el bloque.

- ***Embedded System Integration***: las tareas de esta categoría integran el IP Core generado con el procesador embebido. Se aprecian cuatro tareas internas:

    - *Create Project*: permite crear un proyecto en la herramienta de Xilinx® escogida.
    - *Generate Software Interface Model*: esto nos crea una interfaz sobre el modelo que estamos compilando en HDL, para poder controlar desde Simulink® como se ejecuta el programa dentro de la FPGA.
    - *Build FPGA Bitstream*: genera el bitstream del proyecto que hemos creado. Esto se hace llamando a las herramientas de Xilinx[1] que hayamos incluido en MATLAB®, de forma externa a este.
    - *Program Target Device*: esta tarea se corresponde con el último paso del flujo de trabajo. Así pues, una vez se ejecuta dicha tarea, se programa la FPGA con el bitstream antes creado.

Una vez vistas las distintas etapas que requiere el **HDL Workflow Advisor** para pasar del modelo en Simulink® a implementar el programa en la placa, explicaré mi toma de contacto con ella y sacaré algunas conclusiones.

En el momento de ejecutar la primera tarea, *Set Target Device and Synthesis Tool*, apareció el siguiente error:

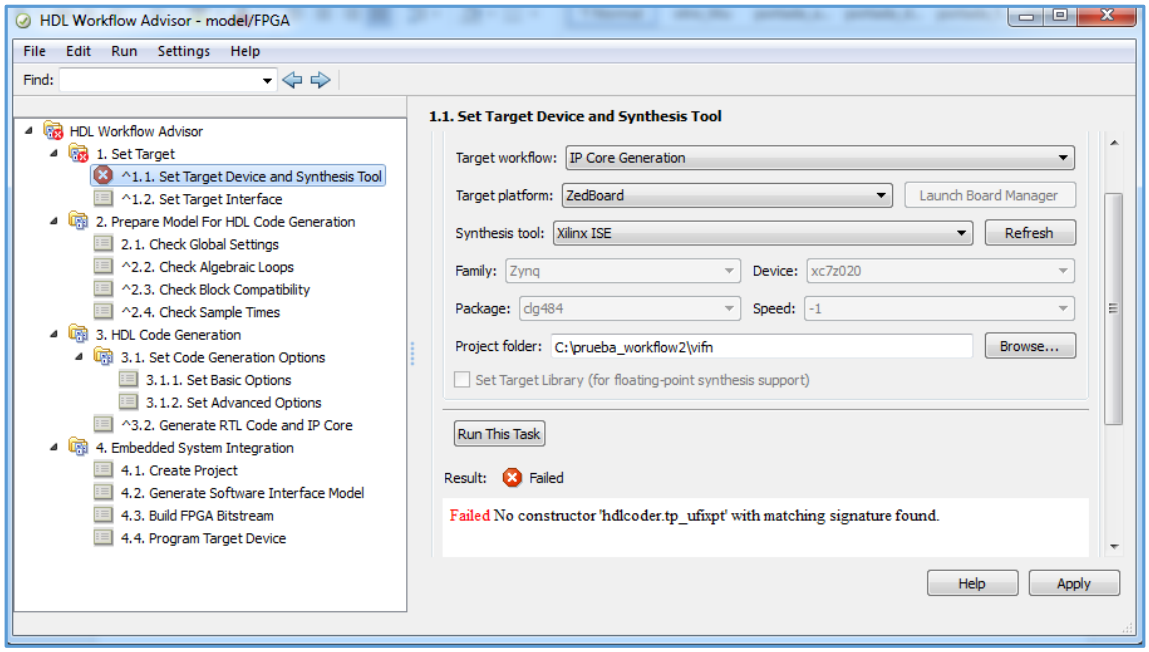


Figura 22

No lo reconocía como nada que hubiese visto mientras investigaba sobre **HDL Workflow Advisor**, por lo que decidí buscar en la Web, pero no había una sola entrada útil sobre 'hdlcoder.tp_ufixpt'.

COMENTARIOS

[1]Aunque cuando le damos a *Run this task* el tick de tarea finalizada se pone verde, dándonos a entender que podemos pasar a la siguiente, esto no es así. El tick sólo indica que la compilación ha comenzado sin problemas. Se abrirá un *shell* externo donde se ejecutarán en segundo plano las herramientas de Xilinx. En esta ventana es donde se nos indicará el éxito o fracaso en la compilación del proyecto.

Es más, cuando cerré la ventana del **HDL Workflow Advisor**, y probé a realizar cambios en el modelo para ver si podía arreglarlo, surgió un nuevo inconveniente: al intentar volver a abrir la herramienta, surgía de nuevo el error, pero sin permitirme acceder a ella:

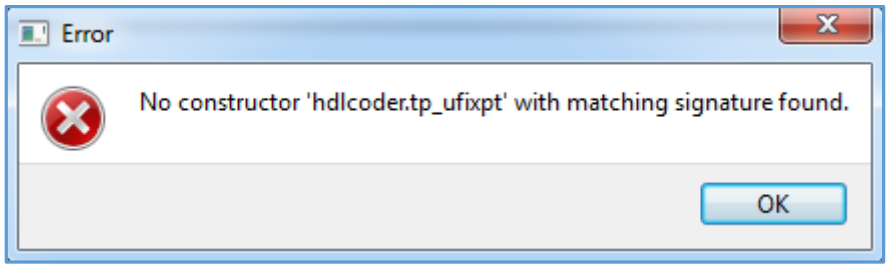


Figura 23

Esto fue un obstáculo que me retuvo bastante tiempo y entorpeció el desarrollo de mi trabajo. Por ello, creo importante comentar la solución, para que así aquellos a quienes les pase sepan cómo arreglarlo de forma rápida y sencilla.

Tras investigar la información de MATLAB® (en especial sobre las herramientas de Xilinx®), encontré dos comandos que podrían solucionar el problema (MathWorks Documentation, s.f.):

- **rehash toolboxcache**: este comando permite refrescar la caché del *path* del sistema para archivos y funciones.
- **hdlrestoreparams('prueba_xilinxblockset');** : con este comando propio del HDL Coder podemos devolver a su valor inicial los parámetros de bloques y herramientas HDL.

Empleando ambos comandos en conjunto, eliminamos el problema de acceso a **HDL Workflow Advisor**, ya que con uno "reiniciamos" la caché y con el otro "reiniciamos" la herramienta y sus parámetros. Con esto, podremos volver a acceder a la ventana inicial de **HDL Workflow Advisor**.

Finalmente, tras bastante tiempo de búsqueda y pruebas infructuosas, encontré la razón por la que fallaba la primera tarea del **HDL Workflow Advisor**. Indagando en la documentación de MATLAB® sobre cómo manejar estas herramientas de Xilinx®, encontré esto:

Requirements and Limitations for IP Core Generation

To generate a custom IP core:

- The DUT must be an atomic system.
- There cannot be both an AXI4 interface and AXI4-Lite interface in the same IP core.
- The DUT cannot contain Xilinx System Generator blocks or Altera DSP Builder Advanced blocks.
- If your target language is VHDL, and your synthesis tool is Xilinx ISE or Altera Quartus II, the DUT cannot contain a model reference.

Figura 24 (modificado a partir de MATLAB® Help, s.f.)

Es decir, **no pueden usarse bloques de las librerías de Simulink® facilitadas por Xilinx® para diseñar un sistema que queramos implementar como IP Core en una placa de la propia Xilinx®.**

No es 100% seguro que el fallo se deba exactamente a este motivo, pero dos cosas son seguras:

1) No podía usar el **Xilinx Blockset** para diseñar un IP Core.
2) Cuando probé a cambiar el modo de implementación, el fallo desapareció.

Esto supuso un gran problema para los esquemas que tenía pensados sobre cómo y en qué dirección dirigir el proyecto. Tras este descubrimiento, debía buscar un nuevo enfoque de cómo implementar los programas necesarios en la ZedBoard, para poder cumplir mi objetivo.

Puede parecer que todo lo que había hecho hasta este momento se volvía inútil, pero nada más lejos de la realidad. Todo lo que he aprendido sobre el manejo de herramientas de Xilinx® desde MATLAB® empleando System Generator for DSP™ es ciertamente útil para desarrollar proyectos de otras formas, y sobre todo, para no volver a cometer estos errores en el futuro.

Tras estas conclusiones, sigamos describiendo el proceso seguido durante el desarrollo del proyecto.

2.2.5. VRML: creación desde cero del robot virtual

Tal y como comenté al comienzo del trabajo, varios de los aspectos tratados en este proyecto eran nuevos para mí, y uno de estos, con los que menos contacto previo tuve, es el lenguaje VRML.

El objetivo de emplear VRML en este proyecto consistía en diseñar e implementar un "mundo virtual" simple, donde existiera un robot. Éste nos serviría como testeo de los resultados: las posiciones angulares deseadas para cada servo, enviados por la controladora OpenCM9.04, llegarían a cada una de las articulaciones del robot, de forma que se representasen los desplazamientos sufridos por el robot real y pudiéramos probar la controladora fuera de línea.

El código diseñado para este robot virtual se encuentra en el **Anexo I: Código**¸ y el banco de pruebas para chequear el correcto funcionamiento del mismo se encuentra en el **Anexo II: Simulink.**

Comencemos explicando de forma resumida las bases del VRML y de la forma de programar en este lenguaje. Posteriormente, entraremos en detalle en el código. Por último, veremos ciertos detalles sobre el archivo Simulink® para las pruebas, y los resultados de simulación.

2.2.5.1. Bases del lenguaje VRML

Virtual Reality Modeling Language es un lenguaje de programación basado en **nodos**. Cuando deseamos crear una figura en el espacio 3D, lo que hacemos es definir una serie de **nodos**. Cada uno de ellos tiene una serie de campos que indican la forma, apariencia, material…, colgando unos de otros. Así se entiende el VRML como un lenguaje bastante estructurado, aunque también difícil de manejar al principio (Gámez, I., 2001).

De acuerdo con este autor, los **nodo**s más importantes para poder entender un programa VRML son:

- *Shape*: este **nodo** permite la definición de la forma de un objeto. Dentro de éste se distinguen dos campos:

    - *geometry*: nos indica cuáles son las características de representación de un objeto en un espacio tridimensional. Las formas básicas son la esfera, el cilindro, el cubo y el cono.
    - *appearance*: define los cambios reflejados respecto a ciertas propiedades físicas, como la luz, de un objeto. Dentro de éste podremos encontrar varias características, pero destacan *material* (el material del que está construido el objeto) y *texture* (características de rugosidad y pliegues principalmente).

- *Transform*: este **nodo**, que abarca al anteriormente expuesto, permite además definir las características de posición y orientación del objeto en el espacio 3D. Además permite realizar traslaciones y rotaciones con respecto al eje del mundo o a los de otro objeto, siendo esto muy útil para construir estructuras móviles.

Así pues, modificando estos **nodos** y parámetros conseguimos definir un objeto para que se ajuste a nuestras necesidades. Creando varios objetos, y relacionando sus características entre sí, se pueden diseñar objetos aún más sofisticados y complejos (además de emplear herramientas que el propio lenguaje nos ofrece), por lo que las posibilidades del VRML son infinitas y sus límites se definen en función de la imaginación del programador.

Aunque no los he explicado en esta sección, existen otros tipos de **nodos** para definir numeroso parámetros, relacionados con el entorno, el punto de vista o la iluminación de la escena.

Estos los veremos con más detalle a continuación, poniendo como ejemplo de su uso aquellos que aparecerán en el código diseñado por mí.

2.2.5.2. Diseño en VRML: robot virtual

En esta sección analizaremos el código implementado para definir el robot en el mundo 3D. Recomiendo seguir el análisis observando el código para una mejor comprensión del mismo.

La primera línea del código es obligatoria para cualquier fichero que contenga una escena en VRML. Ésta indica al visualizador que se trata de un fichero VRML, de la versión 2.0 y que usa codificación *utf8*.

Tras esta se aprecian unos comentarios que informan del método de creación del fichero. En este caso se creó empleando V-Realm Builder, proporcionado por MATLAB®, y que permite crear programas VRML de forma visual, facilitando la tarea a quienes no tienen experiencia con el lenguaje. Mencionar que aunque el fichero se creó así, las partes más complicadas del programa tuvieron que hacerse de forma manual, editando el código desde MATLAB®[1].

A continuación, apreciamos un *SpotLight*. Éste nos permite crear un punto o foco de luz que ilumine la escena. No es imprescindible usarlo, pero en diseños más complejos puede resultar muy útil.

Entramos en el núcleo del programa: la definición del robot. Así pues, podemos apreciar sus distintas partes:

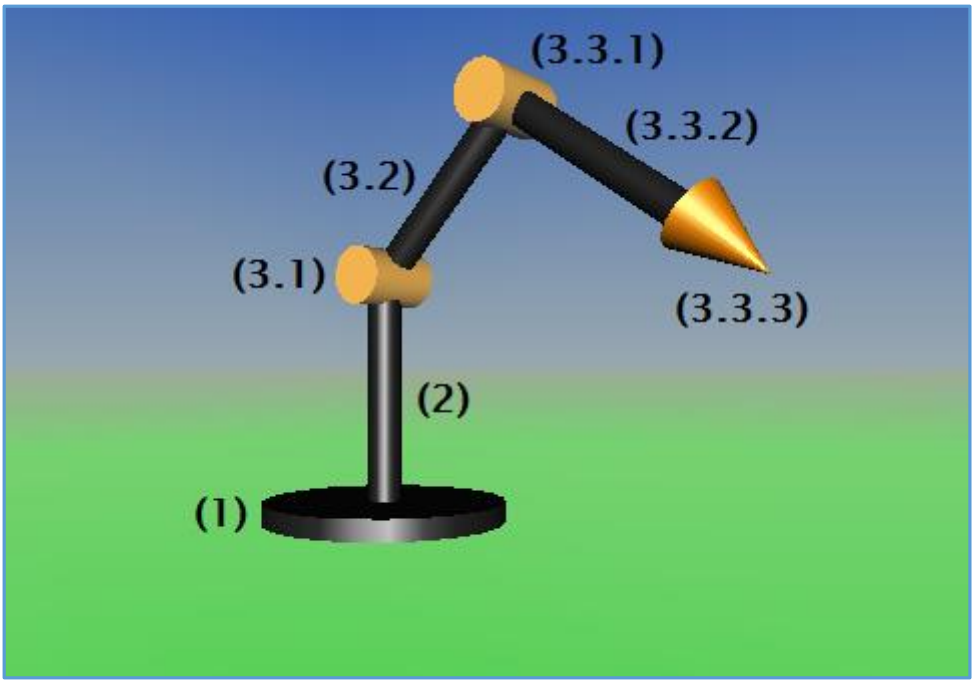


Figura 25

COMENTARIOS

[1]Si se quiere abrir el fichero en formato texto para editarlo desde MATLAB®, debe usarse el comando **edit** **nombre_del_archivo.wrl**. En caso de desear abrir el visualizador *3D World Editor* de MATLAB®, debe emplearse **vredit** **nombre_del_archivo.wrl**. Si quiere emplearse V-Realm Builder, recomiendo emplear primero el comando **vrinstall**, para instalar el software. Tras esto, ir a la pestaña *Home*, a la sección *Environment* y seleccionar en *Preferences -> Simulink 3D Animation*, V-Realm Builder como editor por defecto. Posteriormente añadir en Simulink® un bloque **VR Sink**, que nos permitirá añadir nuestro *source file* y editarlo empleando V-Realm Builder.

(1) **Base**: es el soporte del robot manipulador. Además se corresponde con la primera articulación del mismo, que es rotativa (en realidad al decir que gira la base, hacemos que gire el robot completo). De esta tan sólo hemos definido una pequeña traslación, para dejar el centro de su cara inferior en el punto (0,0,0), y los matices de color del material. Aunque es obvio, para crearla se ha empleado el módulo *Cylinder*, con una altura de 0,25 m y un radio de 1,5 m.

(2) **Barra 1**: esta es la primera barra del conjunto, tal y como su propio nombre indica. Aparte de la rotación habilitada por la articulación de la base (que afecta a todo el conjunto, y se produce en el mismo eje de esta barra), ésta no sufre ningún movimiento más. Su función principal es estructural y de soporte. De nuevo este objeto se ha trasladado, con tal de que la cara inferior de este *Cylinder* estuviera a la misma altura que la cara superior de la base. También se repite el color, y sus dimensiones son 2.5 m de altura y 0.2 de radio (estos valores, al igual que los demás de otros objetos, no se han asignado con tal de dar veracidad del diseño, sino para que el robot 3D quedara proporcionado).

(3) **Parte 1**: con tal de que el giro o traslación de un elemento afectara a los que le siguen en el robot, se fueron creando distintas partes como esta, que engloban a las barras y articulaciones que le siguen, y que deben moverse según el giro de una articulación. Así pues, dentro de **Parte 1** podemos distinguir entre:

   (3.1) **Joint 1**: se trata de la primera articulación (sin contar la incluida en la base). Tal y como se aprecia en la imagen, ésta es rotativa, y cuando se produce un giro de la misma, las barras y articulaciones que están detrás deben moverse en consonancia. Así pues, el giro de la **Parte 1** se produce en el eje de rotación de este *Cylinder*. La articulación está desplazada para ajustarse al final de la **Barra 1**, se encuentra rotada un cierto offest, ya que los objetos *Cylinder* son verticales por defecto, y nos interesaba que fuera horizontal; y el color del material ha cambiado a un tono anaranjado. Las dimensiones de la misma son 1 m de alto y 0.35 m de radio.

   (3.2) **Barra 2**: esta barra sigue a la **Joint 1**, y por tanto se movería en función de ésta. La barra está trasladada y rotada con el objetivo de proporcionarle una posición acorde a su función y que tenga un sentido en el diseño, estando unida su parte inferior a un lateral de la **Joint 1**. El color vuelve a ser negro, y sus dimensiones son las mismas que las del resto de barras.

   (3.3) **Parte 2**: al igual que pasaba con la **Parte 1**, observando la Figura 25 se aprecia cómo nos interesa que los elementos que siguen a la siguiente articulación se muevan cuando ésta gire. Así pues, están todos englobados en la **Parte 2**:

      (3.3.1) **Joint 2**: esta última articulación le permite hacer un giro más al robot, teniendo así 3 GDL. Al igual que muchos otros elementos, está trasladada y rotada para unirla al final de la **Barra 2** y al principio de la **Barra 3**. Los colores y dimensiones se corresponden de forma exacta con los de la **Joint 1**.

      (3.3.2) **Barra 3**: une la última articulación con el elemento final. Sus características geométricas y de apariencia son iguales a las de otras barras.

      (3.3.3) **Elemento_final**: éste se corresponde con la herramienta que el robot tiene en su extremo final, y que le permite operar. En este caso ha sido representada por un objeto de tipo *Cone*, con radio 0.4 m y altura 1 m; dando la imagen de ser una herramienta punzante (podría ser por ejemplo un robot manipulador para soldadura por arco, tan extendidos en el ámbito industrial). El elemento final ha sido trasladado, rotado e incluso escalado para ajustarlo a las necesidades del diseño. Sus colores son parecidos a los de las articulaciones, aunque los parámetros difieren en cierta medida.

Casi al acabar de describir estos elementos en el código, podemos ver cómo se define el centro y eje de rotación para cada una de las articulaciones, y por tanto, de las distintas partes. Esto se corresponde con la parte más difícil conceptualmente del código y del diseño, no por su dificultad intrínseca, sino por la poca información que encontré en los distintos recursos empleados. La solución la hallé

adaptando y reciclando varios conceptos que estos recursos me ofrecían, pero que no indicaban el método para implementarlo.

Por último, para finalizar el archivo encontramos un módulo *Background*, que permite dar un fondo a la escena para hacerla algo más realista, y un *Viewpoint*. Este último es más importante, ya que cada *Viewpoint* del que dispongamos añade una forma distinta de observar el entorno 3D.

2.2.5.3. Pruebas del diseño: Simulink®

Así pues, una vez analizado el programa que describe el entorno virtual, veamos cómo implementarlo en Simulink® para simularlo.

Para ello se ha creado un banco de pruebas que puede observarse en el **Anexo II: Simulink**.

Al igual que con el anterior banco de pruebas, explicaré los bloques empleados ya que pueden no ser tan conocidos y conviene tener clara su función dentro del programa (MATLAB® Help, s.f.):

- ***VR Sink***: este bloque de la librería **Simulink 3D Animation** permite escribir valores en los puertos del mismo, que se corresponden con características del entorno 3D asociado al bloque. La primera vez que introduzcamos el bloque en el fichero .xls y pinchemos sobre él, emergerá una ventana con una serie de campos importantes, destacando:

    - *Source file*: aquí debemos introducir el nombre del archivo de VRML que define el entorno virtual que deseamos unir a este bloque. En caso de no tener archivo, puede crearse uno mediante la opción *New*.
    - *Sample Time*: es el tiempo de muestreo de la imagen en simulación. Se recomienda adaptar este valor para que la experiencia visual sea la apropiada.
    - *VRML Tree*: nos muestra cómo se estructura el archivo de VRML y el entorno 3D. Los parámetros de los distintos objetos que se pueden modificar desde Simulink® a lo largo simulación son aquellos marcados con una caja. Si queremos que el bloque tenga un puerto de entrada para ese parámetro, debemos pulsar sobre la caja. Así es como podemos interactuar con el entorno 3D, y en nuestro caso, sobre las articulaciones del robot. Modificando el valor de la rotación de cada articulación, podemos recrear el movimiento del robot manipulador.

- ***VR Signal Expander***: permite "expandir" un vector de entrada en una señal para un campo VRML. Permite definir cuál es el ancho de la salida, es decir, del campo VRML, y en qué posición o posiciones guardar el valor del vector de entrada. Es importante conocer este bloque, ya que nos permitirá convertir señales de tipo Simulink® a valores perfectamente adaptados para su empleo en VRML.

Así pues, el banco de pruebas consiste en emplear tres bloques de rampa, con el objetivo de que el ángulo que representa cada uno de ellos varíe de forma lineal con el tiempo, y no de forma instantánea; para dar valor a los campos VRML de cada una de las tres rotaciones del robot (la del objeto **Robot** que engloba todo el conjunto, la del objeto **Parte 1** y la de **Parte 2**, que fueron explicadas con anterioridad).

COMENTARIOS

---

[1]Una vez estén ajustados todos los parámetros de la ventana y le demos a OK, al volver a pinchar en ella nos aparecerá un visualizador de la escena 3D. Si queremos volver a la ventana de opciones, debemos ir a la pestaña *Simulation* del visualizador, y clickar en *Block parameters…*

Veamos pues los resultados de la simulación:

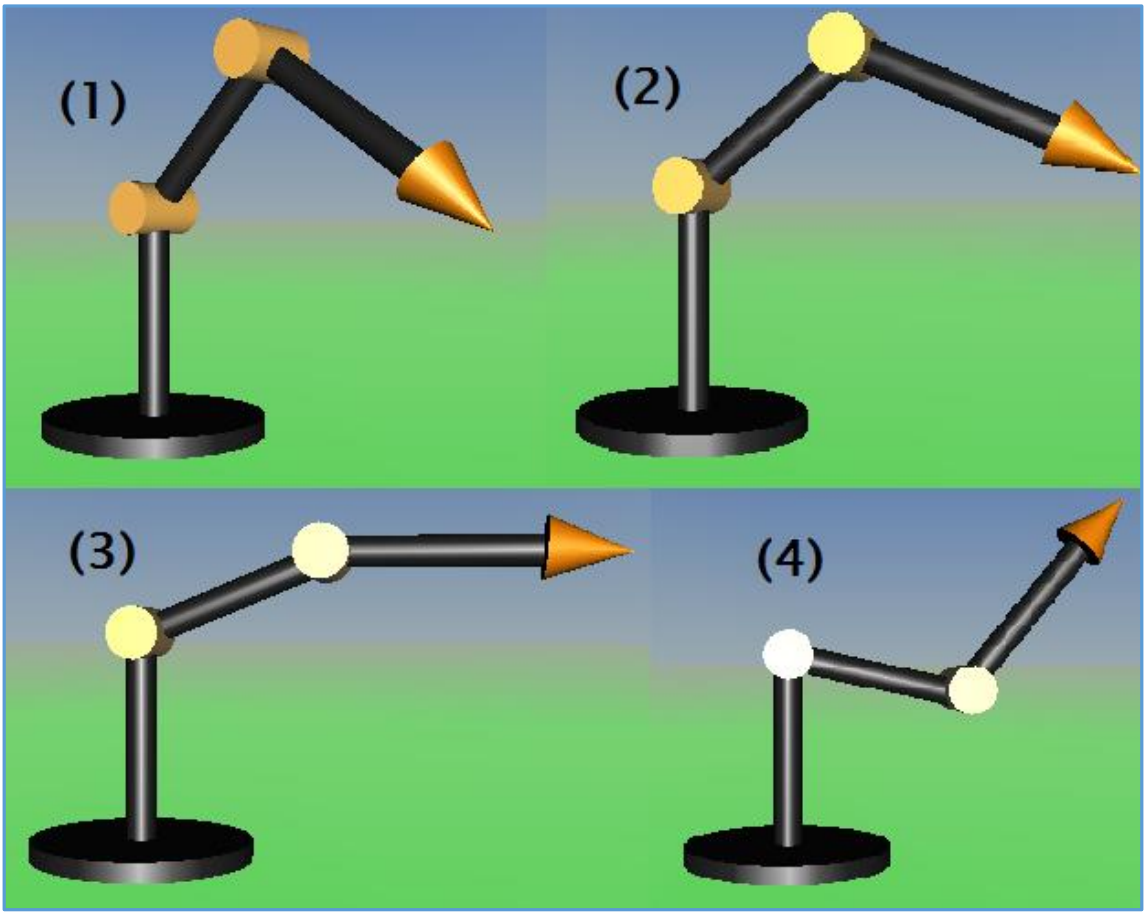


Figura 26

Como podemos apreciar, el robot virtual se mueve de forma correcta a partir de los ángulos que recibe del exterior. Por tanto, el entorno 3D donde visualizar las acciones de nuestro robot, y más concretamente, de nuestros servos, está listo.

Una vez terminado este banco de pruebas 3D en VRML, me propuse buscar soluciones para el problema surgido por las librerías de **Xilinx Blockset**.

Así pues, tras un cierto tiempo que empleé en investigar distintas posibilidades, llegué a dos soluciones plausibles, que expondré en la siguiente sección.

2.2.6. Diseño de pruebas (II)

Con el objetivo de poder finalizar la simulación y comunicación de la controladora con el robot virtual, desarrollé dos posibles soluciones totalmente opuestas:

- Desarrollar el mismo programa empleado en la sección de pruebas anterior, pero en vez de implementarlo como IP Core, realizar un FPGA-In-The-Loop.
- Romper con lo que había estado trabajando hasta el momento en el proyecto, es decir, no emplear System Generator for DSP™ (MATLAB® y las herramientas de Xilinx® de forma conjunta).

Veamos las ideas en más profundidad, y el alcance que tuvieron.

2.2.6.1. Solución integrada: FPGA-In-The-Loop

Esta fue la primera idea que intenté poner en práctica, ya que los diseños para implementar el IP Core ya estaban creados, y me servirían para realizar FIL (FPGA-In-The-Loop) con la ZedBoard.

No obstante, cuando fui a ponerla en práctica, me surgió el primer problema.

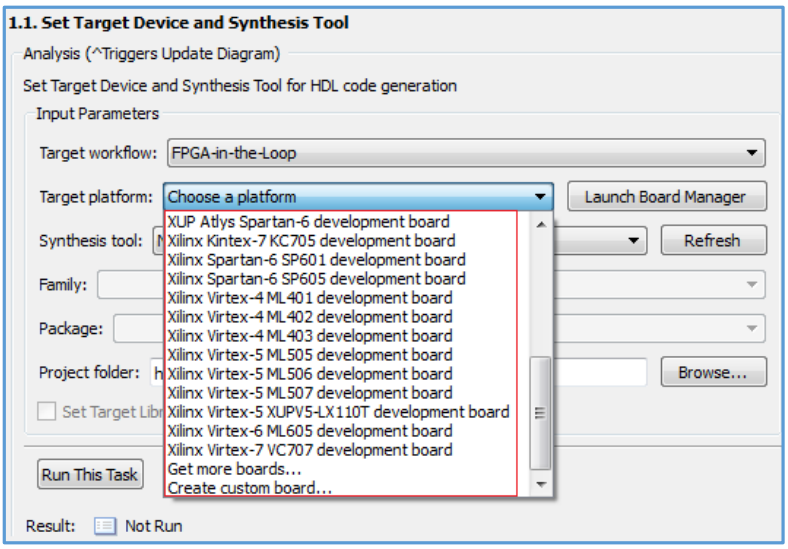


Figura 27

En la herramienta **HDL Workflow Advisor**, al elegir la placa en la que quería implementar mi FIL, resultó que ZedBoard no estaba entre las opciones disponibles. Es más, ninguna tarjeta de la familia Zynq lo estaba. Así pues, tras buscar información acerca de esto, descubrí lo siguiente:

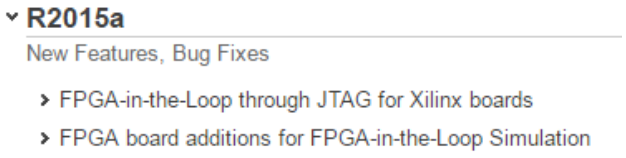


Figura 28 (fuente: MathWorks Documentation, s.f.)

La Figura 28 nos muestra un recorte obtenido de la web sobre *Release Notes* para el **HDL Verifier Support Package for Xilinx Boards** que The MathWorks, Inc. nos ofrece. Brevemente, el *support package* de HDL Verifier es el que debemos instalar para poder realizar FIL con nuestra FPGA desde MATLAB®, y las *Release Notes* nos informan de que características se incluyen en cada versión del software. Así pues, vemos que en la versión R2015a se incluyeron nuevas FPGAs para FIL, que no estaban en R2014b: entre ellas, por supuesto, se encuentra la ZedBoard (MathWorks Documentation, s.f.).

Así pues, el siguiente paso de mi proyecto volvía a ser la instalación del software necesario. En este caso, dos programas: MATLAB® R2015a, para poder elegir la ZedBoard para FIL, y Vivado Design Suite 2015.2. Éste último no se instaló principalmente por esto, sino porque más adelante se intentarían explotar sus cualidades para el empleo de la parte PL y PS de la placa de Zynq. No obstante, decidí emplearlo con MATLAB® R2015s, ya que son programas compatibles.

Una vez todo estuvo instalado, procedí a realizar los pasos explicados en los primeros apartados de este proyecto, con tal de configurar MATLAB® para emplear Vivado como herramienta HDL predeterminada. El proceso fue satisfactorio y no tuve problemas como la primera vez. Esto me fue útil para comprobar la potencia de una guía que explique los fallos que pueden surgir.

Cuando inicié **HDL Workflow Advisor** y revisé las tarjetas de Xilinx® que podían emplearse para FIL, la ZedBoard estaba entre ellas. Así pues, por esa parte todo había ido según lo esperado.

No obstante, surgieron complicaciones durante el seguimiento de los pasos de la herramienta, y aunque le dediqué un cierto tiempo, no obtuve resultados satisfactorios. Por ello, decidí abandonar la idea de emplear System Generator, ya que tenía otras soluciones posibles que veremos a continuación.

2.2.6.2. Solución no integrada: MATLAB® y Xilinx® por separado

Así pues, tal y como dije anteriormente, en vez de usar el System Generator para controlar los procesos internos ejecutándose en la ZedBoard desde el ordenador, emplearía MATLAB® y los programas de Xilinx® de forma no integrada: MATLAB® se encargaría de ejecutar el modelo 3D, y la ZedBoard haría de puente de comunicaciones entre la OpenCM9.04 y MATLAB®. No obstante, para que esto fuera posible, necesitaba un nexo entre la FPGA y el ordenador.

Para este diseño se me ocurrieron varias alternativas, que fui probando hasta solventar el problema.

Así pues, veamos cada una de ellas, con sus ventajas e inconvenientes. El código de estas soluciones se encuentra en el **Anexo I: Código**.

2.2.6.2.1. Alternativa I : Half Duplex UART (primera version)

Aunque las pruebas del IP Core no pudieron hacerse debido a los problemas de librerías de Simulink®, el modelo interno a la *Black Box* incluida en el archivo .slx era bastante simple: un puerto bidireccional creado en VHDL más la generación del PWM de salida para los sensores.

Para estas nuevas pruebas decidí diseñar una UART, ya que tarde o temprano me haría falta para poder implementar el protocolo de Dynamixel de forma correcta. Este diseño lo uní junto con el puerto bidireccional para dar lugar a un diseño más sofisticado y complejo, que a partir de los bits de entrada enviados por el OpenCM9.04 me diera como resultado los bytes (o *words* de otro número de bits) más relevantes del paquete, y que realmente necesitara para discernir a qué posición angular debe moverse el servo que corresponda. En el **Anexo I** este diseño puede encontrarse bajo el nombre de **uart**.

El PWM de salida que emula a los sensores no fue incluido en este diseño ni en los siguientes, ya que requería una gran cantidad de pines Pmod™ para darle el dutycycle de forma externa al programa, y estos estaban ya ocupados. No obstante, lo explicaré dentro de esta sección, y mostraré pruebas de su correcto funcionamiento.

Consiguientemente, analicemos el código VHDL, diseñado con el software ISE Design Suite 14.7, con el objeto de entender cómo se ha intentado resolver el problema de las comunicaciones entre OpenCM9.04 y la ZedBoard. Tras conocer sus distintas partes, veremos el desarrollo de este diseño y si surgieron problemas al implementarlo.

2.2.6.2.1.1. Bases del protocolo a tener en cuenta

El primer paso para implementar el protocolo de Dynamixel era tener claros sus aspectos básicos. Como se trata de una UART, debía conocer varias características de la misma:

- Velocidad en baudios: la velocidad de la comunicación entre servomotores y microcontroladora puede modificarse con una orden, pero a mí me interesaba el valor por defecto, que es de 1 Mbps (ROBOTIS(c), 2006).
- Nº de bits: la información nos la proporcionó un recurso web propio de ROBOTIS®, *Overview of Communication* (ROBOTIS(b), s.f.). La UART transmite 8 bits por palabra.
- Start/Stop bit: de nuevo usamos el mismo recurso (ROBOTIS(b), s.f.) para saber que Dynamixel emplea un bit de stop, aunque no se aclara si el bit de stop es a nivel alto o bajo (posteriormente veremos que esto no es así, pero es mejor explicarlo más adelante). Además, tampoco de especifica que haya bit de start, pero se presupone ya que es necesario.
- Paridad: este último parámetro lo obtenemos del mismo recurso (ROBOTIS(b), s.f.), y resulta que Dynamixel no utiliza paridad.

Tras esto, podemos comenzar a diseñar la UART en función de sus características específicas.

2.2.6.2.1.2. *Sampling*

La ZedBoard dispone de dos fuentes de reloj básicas, estando una destinada a la parte de *Programmable Logic* y la otra a la *Processing System*. Nos interesa la primera, ya que será el reloj que emplearemos en nuestro diseño, y que servirá para sincronizar los procesos hardware. El reloj empleado en la PL tiene una frecuencia de 100 MHz (ZedBoard(b), 2014).

No obstante, ya hemos dicho que nuestro protocolo emplea una tasa de bits de 1 Mbps.

Esto quiere decir que, si por cada tick de reloj de la ZedBoard, leyéramos el bit que hay en el canal y lo guardásemos, acabaríamos almacenando 100 veces cada bit. Se puede apreciar que esto no es interesante.

Sin embargo, aprovechando esa diferencia en frecuencia diseñamos una especie de divisor. Si hiciéramos una sóla lectura por bit, es decir, si diseñáramos este divisor para dar una salida a una frecuencia de 1 MHz, podríamos cometer errores de sincronización y perder información, o captar bits de forma errónea.

Su implementación es ampliamente conocida, por lo que seré muy breve explicándola: se basa en crear un contador. Cada vez que ocurre un tick del reloj del sistema, aumentamos la cuenta en una unidad; mientras tanto la salida del divisor permanece a nivel bajo. Así, al llegar al número de ticks correspondiente a la división en frecuencia que desearíamos realizar, cambiamos la salida a nivel alto durante un ciclo de reloj, y devolvemos la cuenta a cero, comenzando el ciclo.

Así, por cada bit tendremos diez ticks del divisor. Esto nos servirá para leer sólo durante los ticks centrales del bit, de forma que se reduzcan notablemente los errores por desincronización del sistema

Tras su implementación, se diseñó un *test bench* en VHDL para comprobar su correcto funcionamiento. Podemos ver el resultado en la siguiente imagen:

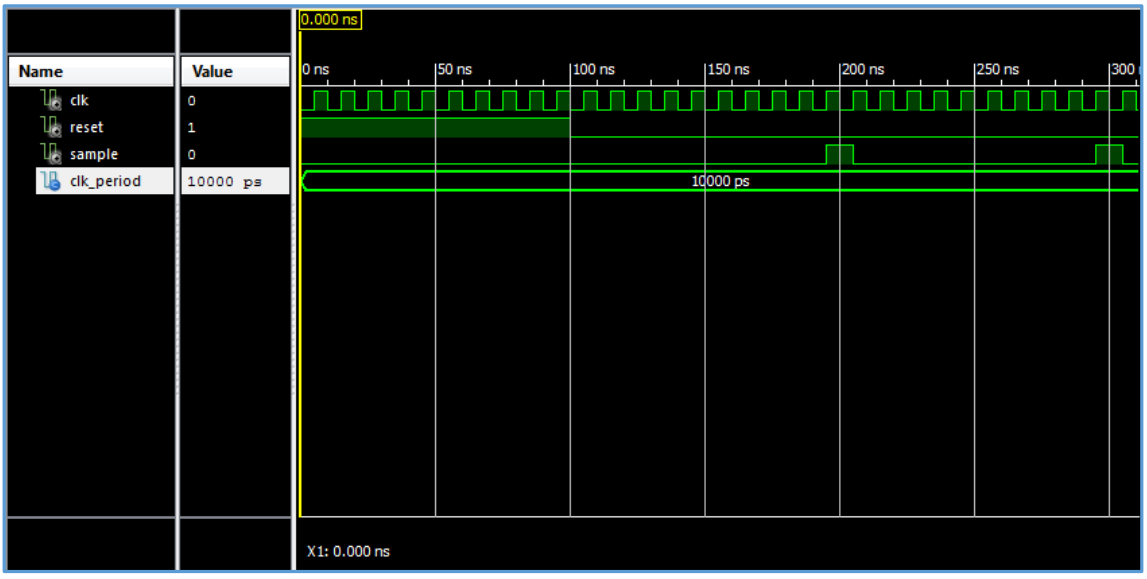


Figura 29

Se aprecia que, cuando el reset del sistema se encuentra a nivel bajo, se produce un tick en la salida simple con una frecuencia de 10 MHz (observando su primera subida, a 190 ns, y la segunda a 290 ns, se aprecia que han transcurrido 100 ns, es decir, una frecuencia de 10 MHz).

2.2.6.2.1.3. Módulo de lectura y creación de bytes

Para el diseño de este módulo me he basado en la implementación propuesta por **Eric Bainville** en su página web (Bainville, E., 2013), ya que me pareció bastante compacto y fácil de entender al leerlo.

Aunque tuve que corregir ciertos errores de funcionamiento, el recurso fue ciertamente útil para aprender nuevos conceptos de programación VHDL.

La funcionalidad que este módulo implementa es la captura de los bits que conforman un byte del paquete, para agruparlos formando dicho byte, que será la salida del mismo. Así, reunimos los bits individuales que recibimos del canal y conseguimos entender su significado como conjunto.

Así pues, el módulo recibirá como entrada dichos bits individuales (además de la señal de reloj del sistema, el reset y la señal de *sampling*), y como salida emitirá los bytes más un bit individual que servirá de *enable*: cuando este bit sea un 1 significará que tenemos un nuevo byte, de forma que aseguramos que no se pase el mismo byte más de una vez al módulo siguiente.

Un concepto interesante, que como antes comenté hace el código más compacto, es el uso de *types* y *records* de VHDL. El empleo de *type* nos permite crear un nuevo tipo de variable o señal, cuyos estados definimos previamente. Esto es ciertamente útil para diseñar una máquina de estados cuyos estados (valga la redundancia) estén relacionados con lo que representan, en vez de ser números o valores que pueden resultar confusos. Además, usando *records* podemos copiar la funcionalidad de las estructuras de C: crear variables dentro de las cuales tendremos distintos campos que nos permitirán describir el estado actual del sistema. En esta idea se cimienta la base del código del módulo de lectura, y por ello debemos entender que significa cada campo del *record* definido en el programa:

- fsm_state: es una señal de tipo fsm_state_t, el cual ha sido previamente definido como un nuevo *type*. Sus posibles estados son *idle* y *active* (después explicaremos su significado en la fsm).
- counter: es señal empleada para contar el número de veces que la entrada de *sampling* se pone a nivel alto.
- bits: en esta señal iremos "almacenando" los bits que lleguen hasta formar un byte. Se asociará a la salida del módulo.
- nbits: funciona como un contador de los bits del byte actual que han llegado, de forma que podamos tener constancia de cuántos más deben llegar.
- enable: nos servirá para indicar que el nuevo byte está listo. También se asociará a la salida del módulo.

Se definen dos señales de estado de lectura, de forma que una se corresponda al estado actual, y otra al que le seguirá.

Para seguir describiendo el módulo, cabe destacar que éste está dividido en tres procesos:

- Actualización del estado: en cualquier diseño VHDL que necesite un cierto nivel de sincronía de sus elementos, es una buena práctica separar los procesos secuenciales de los combinacionales. Por ello, el primer proceso diseñado es el de actualización del estado: cuando se detecta un flanco de subida del reloj del sistema, el estado que anteriormente era el próximo estado pasa a ser el estado actual (en este proceso suele incluirse una condición que asegure que cuando llegué un reset, el módulo vuelva a su estado inicial).

- Lectura y almacenamiento de bits: es en este proceso donde encontramos la máquina de estados del sistema. Ésta contempla dos opciones, tal y como se comentó anteriormente:

  - *idle*: se corresponde con el estado en el que no estamos recibiendo bits, sino que el canal se encuentra a nivel alto porque nadie está escribiendo en él. Así pues, en este estado sólo debemos realizar dos tareas: asegurarnos de que todos los campos de la señal de próximo estado de lectura estén a su valor inicial, y esperar la llegada del bit de start, de forma que cuando detectemos que la entrada de bits se pone a nivel bajo, pasamos al estado siguiente.
  - *active*: si estamos en este estado significa que están llegando bits de un byte del paquete. Así pues, el proceso que se sigue en este estado se basa en capas. En la primera, se discierne si nos llega una señal de *sampling*, de forma que si no es así mantenemos el estado, pero de llegar comprobamos la siguiente capa. En ésta se chequea el valor del campo *counter* de la señal de estado de lectura (ya dijimos que mide el número de señales de *sampling* que nos han llegado

desde que emplezamos el bit), especificándose tres opciones importantes.
Cuando el valor sea 5 significará que estamos en medio del bit, por lo que debemos leer el valor del mismo e introducirlo en el campo *bits*. Además, aumentaremos el contador de bits *nbits* en una unidad. No obstante, esto no ocurrirá cuando dicho contador de bits valga 9[1]. En ese caso, activaremos la señal *enable* para indicar el fin del byte, y volveremos al estado *idle* a esperar la llegada de un start bit. Lo que si hacemos en cualquier caso es aumentar *counter* en uno, para contar el *sample* que ha llegado.
Cuando el valor de *counter* sea 9, significará que hemos llegado al final de un bit, por lo que deberemos volver a poner este contador a 0.
Cuando *counter* valga cualquier otra cosa, tan sólo tendremos que aumentarlo cuando llegue el *sampling*.

Como vemos, este proceso concentra la parte más importante del programa.

- Salida de bytes: este último proceso se encarga de extraer la información sobre el campo *enable* de la señal de estado de lectura, y sobre el campo *bits* de la misma, de forma que saca los bytes del paquete e indica mediante el *enable* que éste ya puede leerse.

Una vez diseñado, se creó un *test bench* y se probó su funcionamiento. Este es el resultado pasándole los siguientes bits en este orden: 0 (bit start), 0,1,0,1,0,0,1,1,1 (bit stop)[2]:

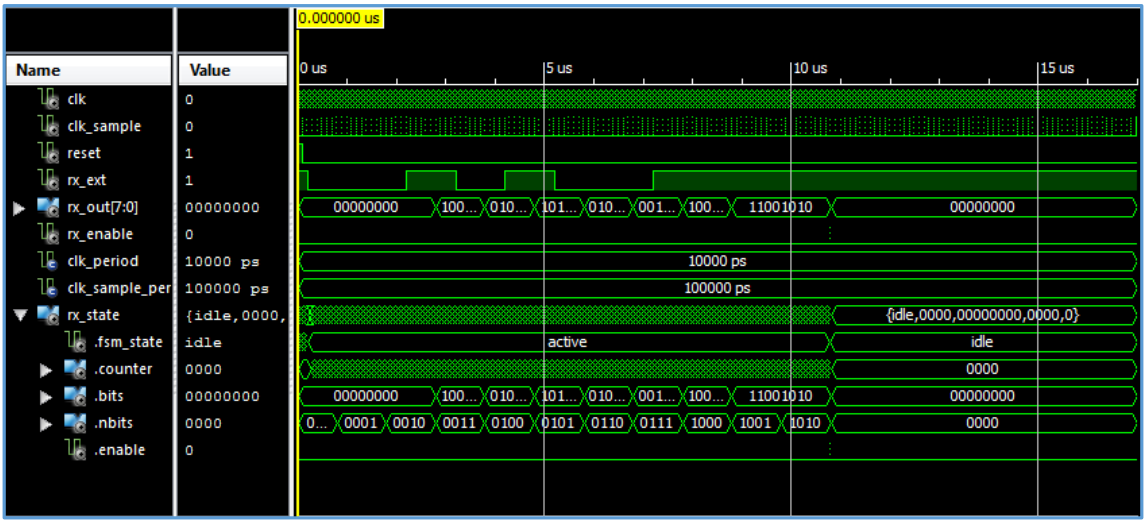


Figura 30

Puede apreciarse que el resultado es correcto[3], obteniendo "11001010" cuando **rx_enable** es '1'.

COMENTARIOS

---

[1]Si miramos el código en el **Anexo I**, observamos otro caso extra, que es cuando *nbits* vale 10. Esto se debe a lo que comenté sobre el stop bit en la sección sobre las bases del protocolo. Según la web de ROBOTIS®, Dynamixel emplea un stop bit. No obstante, esto es falso, ya que emplea dos. En secciones posteriores comentaré cómo lo descubrí, ya que fue parte importante del desarrollo.

Así pues, arriba comentaré el código tal y como lo diseñe en un principio: para un solo stop bit, con objeto de seguir la cronología del proyecto de forma correcta. Comentar pues en este inciso que, en la segunda versión del código, contemplando el segundo stop bit, cuando *nbits* es 9 no hacemos nada, y esperamos al siguiente bit de stop, el décimo, para poner *enable* a 1 y pasar al estado *idle*.

[2]Este *test bench* también funciona para la segunda versión con dos bits de stop, ya que se mantiene el nivel alto en la entrada, y se toma el siguiente bit como un 1.

[3]Este comportamiento es el que obtenemos con el programa para 2 bits de stop del **Anexo I**, pero sirve para ejemplificar el resultado del programa de 1 stop bit ya que es muy similar.

2.2.6.2.1.4. Selección de bytes de información real

La función de este bloque es extraer, de entre todos los bytes que conforman un paquete de Dynamixel, aquellos que contienen información útil para el servo: la ID de aquél a quién va dirigido y los dos bytes de parámetros que contienen la posición de destino.

Así pues, por ahora se ha contemplado la opción de pasarle posiciones nuevas al robot virtual. No obstante, no sería difícil incluir más funcionalidades, tales como indicarle una nueva velocidad o una nueva ID. Para ello habría que incluir el byte del primer parámetro como información útil, ya que éste indica la dirección de la tabla de control del servomotor a la que queremos acceder.

Este módulo recibe como entrada (además de la fuente de reloj y el reset) el *enable* y los bytes provenientes del anterior bloque, y como salida extrae los bytes de información junto a un *enable* propio que indica que ese byte es importante. Además, saca una señal que indica si ha habido error.

Ya que aquí también emplearemos una máquina de estados, emplearemos la función *type* para crear una variable cuyos estados posibles serán: FF_1, FF_2, ID, LONG, INSTR, PARAM, CHECKS y ERR (se explicarán a continuación).

Para diseñar este bloque se ha seguido la misma estructura de tres procesos que en el caso anterior, siendo estos:

- Actualización del estado: al igual que en bloque de lectura de bits, en este caso también separamos los procesos secuenciales de los combinacionales. Para ello, en este proceso nos encargamos de fijar el comportamiento al llegar el reset, reiniciar el sistema, y al llegar un flanco de subida de reloj, actualizar el estado, el contador para el número de parámetros y el *checksum*.
- Reconocimiento y selección de bytes: este proceso se corresponde con la máquina de estados propiamente dicha. No obstante, algo que se hace en cualquier caso es introducir el byte entrante en una señal intermedia que se pasará a la salida. Esto quiere decir que siempre sacaremos el byte que nos llegue, por lo que haberlo seleccionado o no dependerá del *enable*.
  Así pues, entremos a ver los estados posibles:

  - FF_1: se corresponde con la espera al primer byte de comienzo de paquete 0xFF. Así pues, tomamos este estado como el inicial, poniendo a 0 las variables de *checksum*, contadores, **byte_t** (es el *enable* de que tenemos un byte de información real) y error. Estaremos en este estado mientras el *enable* proveniente del bloque anterior no nos indique que un nuevo byte está listo. En ese caso, comprobaremos si se trata de un 0xFF, distinguiéndose dos posibilidades: si lo es, pasamos al estado FF_2; si no lo es, vamos al estado ERR. Esta estructura de doble comprobación se repetirá en el resto de estados.
  - FF_2: en este estado esperamos la llegada del segundo 0xFF de inicio de paquete, comprobándolo cuando el *enable* de aviso de nuevo byte se pone a nivel alto. Las señales que inicializamos a 0 en el estado anterior se mantienen sin alteración.
  - ID: este es el primer estado con información útil, y por ello su trato es algo especial (de éste obtendremos la ID del servo destinatario del paquete). Así pues, el trato de algunas señales es distinto: mantenemos siempre a 0 las señales de error y el contador de parámetros, y las señales **byte_t**, estado y *checksum* dependerán del *enable* de entrada, de forma que cuando éste pase a nivel alto indicaremos que el byte tiene información real poniendo un 1 en **byte_t**, introduciremos el valor del byte en *checksum* y pasaremos al estado LONG.
  - LONG: de nuevo un estado de espera intermedio. Cuando llega el *enable* de nuevo byte, pasamos al siguiente estado, INSTR, y añadimos el byte de la longitud del paquete a la variable de *checksum*. Este estado contendría información muy útil en caso de habilitar más instrucciones de la microcontroladora, ya que algunas requieren el paso de un número de parámetros mayor. Sin embargo, ya que estamos considerando el caso de que la OpenCM9.04 sólo enviará posiciones de destino para el robot virtual, la longitud del

paquete será siempre 5 (3 parámetros + 2)[1]. Por ello, no almacenamos su valor para contar cuántos parámetros deberemos esperar, sino que usamos un contador.

- INSTR: se corresponde del estado de espera a la llegada del byte que nos indica la instrucción. Tal como dijimos antes, esta instrucción será escritura, es decir, la número 3 de Dynamixel. Por tanto, reiteramos el proceso típico de espera: cuando llegué el *enable* de nuevo byte, comprobamos si dicho byte es un 3. De serlo, añadimos el valor a *checksum* y pasamos al estado PARAM. Si no lo es, iremos al estado ERR.
- PARAM: este es el estado más complejo de todos, y el último que contiene información útil. Ya que estamos trabajando sobre la instrucción de escritura sobre la posición de destino, sabemos que recibiremos tres parámetros, de forma que diseñamos una nueva máquina de estados dentro de este estado, que dependerá del valor de un contador. Así, una vez llegué el *enable* de nuevo byte, podremos estar en estos tres estados:

    - "00": en caso de que el contador valga 0, estaremos esperando la llegada del primer parámetro. Este siempre se corresponde con la dirección de la tabla de control del servo a la que queremos acceder. Para el caso de la posición de destino, esta dirección es la 30 (comienza en la 30, pero ocupa dos bytes de espacio, 30 y 31), por lo que el byte en llegar será 0x1E. Así pues, lo que haremos en este estado será comprobar el valor del byte, y de ser el correcto, aumentaremos el contador en una unidad, sin olvidarnos de añadir el valor a *checksum*.
    - "01": si el contador vale 1, significará que el siguiente byte en llegar será la parte menos significativa de la palabra de 16 bits que conforma la posición de destino. En este caso simplemente aumentaremos el contador, añadiremos el valor a *checksum* e indicaremos que es un byte de información importante, dando a la señal **byte_t** un valor de 2.
    - "10": el valor de 2 del contador se corresponde con el byte que almacena la parte más significativa de la posición de 16 bits. En este caso, repetimos la misma operación que en el caso anterior, pero en vez de poner un 2 en **byte_t**, le damos un valor de 3, y además el siguiente estado será CHECKS.

- CHECKS: se corresponde con el estado de chequeo de la suma de comprobación. Básicamente, si el byte entrante es igual a los 8 bits menos significativos de la señal *checksum* negados, el paquete habrá llegado correctamente. Así, de cumplirse esto, pasamos de nuevo al estado FF_1 para esperar un nuevo paquete. Si no se cumple, vamos al estado ERR.
- ERR: es el estado de error, al que llegamos desde los distintos puntos de comprobación si no se cumple la condición esperada. Nuestra única tarea aquí es poner la señalización de error a nivel alto, y volver al estado FF_1.

- Proceso de salida: de nuevo, tenemos un proceso dedicado únicamente a asociar las señales internas del bloque que almacenan la información que nos interesa con las salidas de dicho bloque. Así, sacamos el byte que nos ha llegado, el *enable* que nos indica si el byte es de información útil o no, y que tipo de información tiene, y la señal de error.

Una vez diseñado el bloque, se creó un *test bench* para él, y se realizaron pruebas para comprobar su funcionamiento, que podemos ver en la Figura 31.

COMENTARIOS

[1]Si no se recuerda porque se calcula así la longitud, acudir a la página 3, a la explicación del Paquete de Instrucciones.

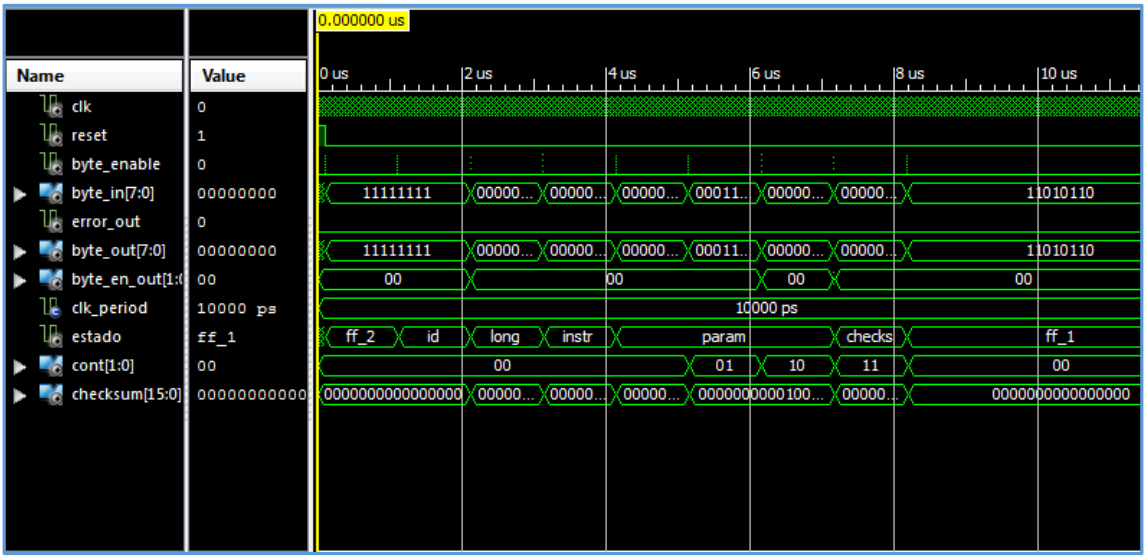


Figura 31

Se aprecia como:

- La variable de estado abarca todos los posibles y en el orden correcto.
- La variable que hace las veces de *enable* de salida, para indicar que hay un byte útil, cambia de valor cuando llega uno de estos bytes (ID, PARAM ("01") y PARAM ("10")).
- El *checksum* va variando su valor cada vez que pasa un nuevo byte por la fsm. Sabemos que la secuencia ha sido correcta, ya que del estado CHECKS se pasa a FF_1 y no a ERR.

Así pues, vemos que el programa funciona de forma correcta.

2.2.6.2.1.5. Bloque de capa superior

Finalmente, llegamos al último bloque de este programa. Se corresponde con la capa de más alto nivel, que llama a los bloques antes explicados para usar sus prestaciones.

Como entradas, este bloque *top* recibe la fuente de reloj, el reset, la señal DATA (en realidad es un puerto bidireccional, por donde nos llegarán los bits de los paquetes enviados por la microcontroladora) y la señal DIR_PORT (es la que nos indica si el puerto DATA será de entrada o de salida).

Como salidas dispone de dos señales: RXD, que es una palabra de 10 bits donde almacenamos la posición de destino o la ID, en base binaria, y RXD_t, que nos servirá para discernir cuál de estas dos es la palabra que nos está llegando en cada momento.

Tanto las entradas como las salidas pueden se encuentran en el archivo de extensión .ucf, **pinout.ucf**, empleado en los proyectos de VHDL para indicar a qué puerto físico de la FPGA se conecta cada puerto virtual del programa.

En este bloque *top* podemos encontrar cinco elementos importantes:

- Instancias de componentes: tal y como dijimos antes, este bloque aglutina a los anteriores, empleándolos como si fueran cajas negras. Por ello, debemos definir los bloques anteriores como componentes en primer lugar. Una vez hecho esto, declaramos una instancia del bloque, asignando las entradas y salidas de éste a entradas, salidas o señales internas del bloque de capa superior, según corresponda.
- Control del reset: para el reseteo de la ZedBoard, hemos habilitado el uso de uno de los múltiples botones que la placa nos ofrece (BTNU), de forma que si éste no es pulsado, la señal de reset se mantiene a nivel bajo, pero mientras se mantenga apretado, el sistema entrará y se mantendrá en un estado de inicialización, a la espera de que se suelte dicho botón.

- Puerto bidireccional: mediante una sola línea, indicamos que DATA será una salida (ponemos que siempre esté a nivel alto, ya que no lo usaremos por ahora) si DIR_PORT está a nivel bajo, y que será una entrada en alta impedancia si está a nivel alto.

  En la Figura 32 podemos ver el esquema que ilustra el circuito interno que sirve de nexo entre el servo y la controladora. Así pues, este mismo circuito también lo encontramos dentro del servo, de forma que, como ya hemos dicho antes, se implemente una comunicación por *half dúplex UART*.

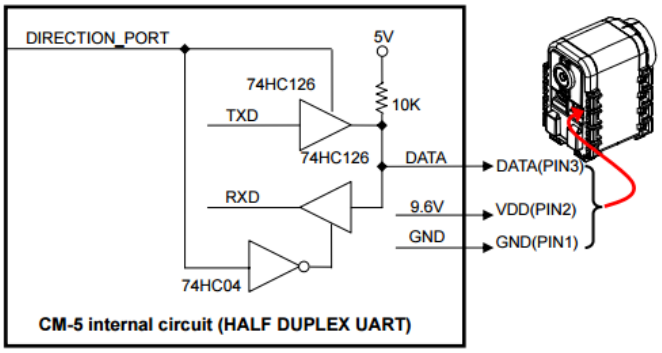


Figura 32 (fuente: ZedBoard(b), 2014)

  En la sección **2.2.2** sobre el diseño de la PCB de adaptación se dijo que se incluía en ella una resistencia conectada a un pin, que servirían para emular una resistencia de pull-up (de hecho, la que se aprecia en la Figura 32). A pesar de su inclusión en el diseño, finalmente decidí no emplearla, y crear una entrada a nivel alto (alta impedancia) en VHDL tal y como he explicado, ya que entendí que sería más fácil y preciso que hacerlo directamente en el hardware. No obstante, este pin queda disponible para diseños que lo requieran.
- Actualización de estado: brevemente, mencionar que este bloque también tiene parte secuencial, que se implementa mediante este proceso.
- Proceso de salida: por último, para tratar las salidas del bloque de la máquina de estados de obtención de información, empleamos este proceso. Su estructura se asemeja a una fsm, pero en vez de emplear el formato switch-case, se utilizan if-elsif-else. Así pues, se comprueban las siguientes condiciones (con el consiguiente orden de importancia):

  - En primer lugar, se chequea si **RXD_type** (que almacena la información sobre qué tipo de byte ha salido de la fsm de obtención de bytes útiles) vale 1. Si se cumple, significará que el byte obtenido es la ID del servo de destino. Por ello, asociamos la salida RXD al byte que contiene esta ID (más dos ceros delante para llegar a los diez bits de RXD), RXD_t se une a RXD_type, para indicar que ese byte que enviamos fuera de la FPGA es una ID, y damos valor a tres variables más (sin contar con un contador que explicaremos más adelante, y que ponemos a 0):

    - RXD_keep: nos permite almacenar el valor del byte en cuestión, de forma que podremos mantenerlo en el tiempo (empleando biestables) para usarlo más delante de ser necesario. En el caso de la ID, le damos el valor del byte que la representa.
    - RXD_t_keep: tiene la misma función que la anterior, pero para los dos bits que indican el tipo de byte. En el caso de la ID, almacena el valor de RXD_type.
    - RXD_ten: de nuevo la misma función que RXD_keep, pero esta vez en formato de diez bits. Para la ID, no se le da valor (se pone a 0), ya que no es necesario.

  - Si **RXD_type** no vale 1, se comprueba si vale 2. Si esto es así, el byte obtenido se corresponderá con los 8 bits menos significativos de la posición de destino. Por ello, ahora no queremos sacar el byte como salida, sino almacenarlo para después formar la palabra de diez bits que indique dicha posición. Así pues, en RXD ponemos todos los bits a 1, indicando que no es una palabra relevante, guardamos el tipo de byte en RXD_t, usamos las variables con **keep** para almacenar byte y tipo de byte y volvemos a reiniciar el contador.
  - Si **RXD_type** no vale ni 1 ni 2, se chequea si su valor es 3. De serlo, significará que el byte actual se corresponde con los 8 bits más significativos de la posición (aunque solo nos importan los dos menos significativos de este byte). Por ello, pasaremos a la salida RXD (de diez bits) el byte anterior, guardado en RXD_keep, en las posiciones menos significativas, y

esos dos bits que nos importan del byte actual, en las dos posiciones más significativas. De esa forma, habremos creado la palabra de 10 bits que indica la posición de destino. Además de eso, guardamos en RXD_t y RXD_t_keep el tipo de byte, mantenemos el estado de RXD_keep, ya que lo necesitamos como hemos visto, y en RXD_ten almacenamos lo mismo que sale por RXD.

- En caso de que **RXD_type** no sea 1,2 o 3, habrá que comprobar el contador que pusimos a cero cada vez que se cumplían los puntos anteriores. Si este contador es menor de 3, mantenemos el estado según nos interese. Básicamente, esto nos permite alargar tres ciclos de reloj más las salidas RXD y RXD_t, para que su lectura sea más fácil.
- En caso de que el contador sea mayor o igual a 3, y **RXD_t** no sea 1, 2 o 3, pasaremos como salida RXD todo unos, para indicar que no es un dato de posición o ID válido.

Así pues, con estos cinco elementos conseguimos completar el diseño de nuestra UART con selección de información útil. Veamos una prueba de simulación del proyecto completo:

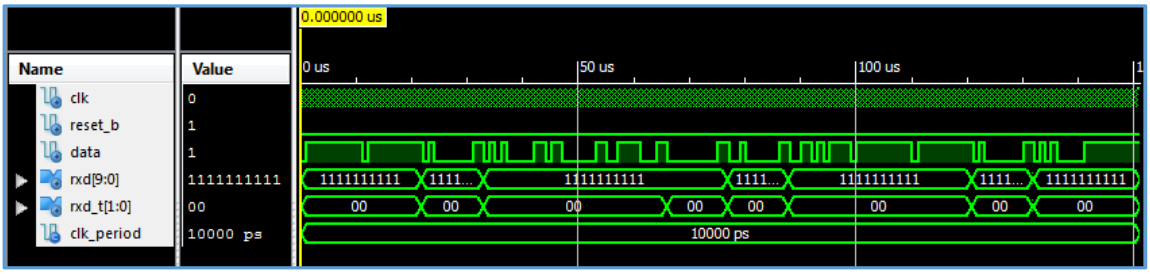


Figura 33(a)

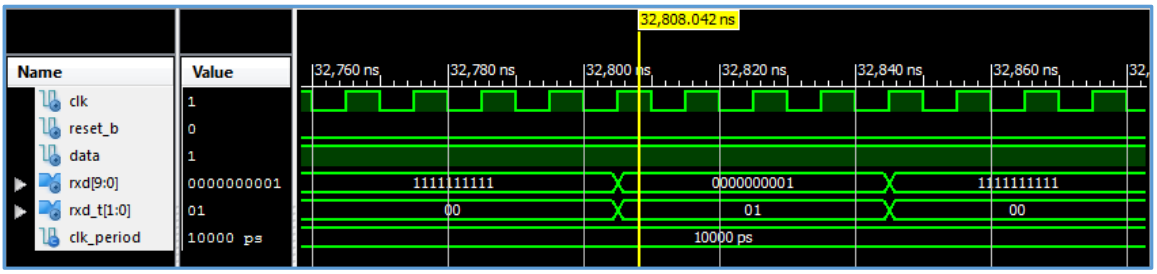


Figura 33(b)

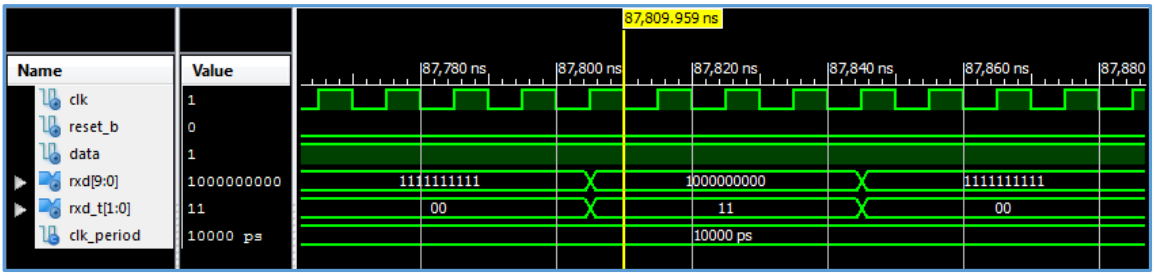


Figura 33(c)

En la Figura 33(a) se aprecia como RXD en su mayor parte del tiempo tiene como salida todo unos, indicando que en ese momento no hay información válida.

No obstante, si nos fijamos en las Figura 33(b) y Figura 33(c), vemos como cuando se obtienen datos importantes (33(b) – ID / 33(c) – posición), RXD cambia mostrando el valor del dato, y RXD_t contiene el tipo de dato.

Así pues, se comprueba el correcto funcionamiento del proyecto VHDL.

2.2.6.2.1.6. PWM

Tal y como se dijo anteriormente, la salida PWM no fue incluida en el diseño, debido a que finalmente no se emplearía para simular los sensores. No obstante, ya que fue incluida la posibilidad de usarla en la placa de adaptación hardware y el diseño estaba listo y funcional en software, considero oportuno mencionar cómo se realizó esta función de salida PWM, para futuras ampliaciones del proyecto. El diseño de este PWM se basa en el de **Carlos Ramos**, subido a su blog (Ramos, C., 2012).

La creación de esta salida PWM se lleva a cabo en el proyecto **inout**[1], disponible en el **Anexo I**.

Básicamente, el diseño consta de dos procesos bien diferenciados:

- **clk_500hz:** este proceso se encarga de generar un flanco de subida de una señal interna (**clk2**) en función de la fuente de reloj del sistema. Es decir, disponemos de un contador que crece por cada tick de reloj, de forma que cuando el contador llegue al valor que nos interese (según la frecuencia de PWM que deseemos) pondremos la señal interna a nivel alto y reiniciaremos el contador. En este caso, ya que estaba buscando una señal de PWM de salida de 500 Hz, el contador llegará, desde 0 a 1999[2], de forma que cada 2000 ciclos de reloj, tendremos un tick de la señal interna.
- **pwm_signal:** este proceso es básicamente un contador que llega hasta 100 y que aumenta la cuenta cada vez que la señal interna **clk2** tenga un flanco de subida. Con esta cuenta conseguimos reducir la frecuencia de 50 kHz a los deseados 500 Hz.

Puede parecer que es poco eficiente dividir el proceso en dos partes, ya que se podría haber contado en el primer proceso 100 veces más, obteniendo el mismo resultado. No obstante, esta división es interesante para obtener la salida PWM, ya que ésta se pone a nivel alto mientras el contador del segundo proceso sea menor o igual que el dutycycle, y a nivel bajo mientras sea mayor. De este modo podemos pasar el dutycycle en su forma de porcentaje, lo que es más cómodo.

Veamos pues los resultados de simulación:

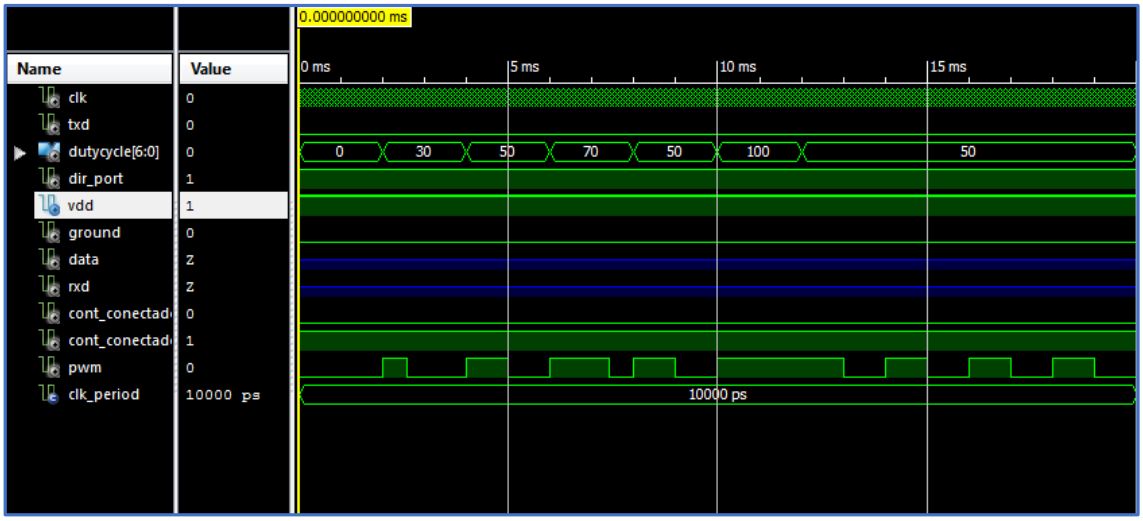


Figura 34

COMENTARIOS

[1]Este proyecto es el que se diseñó para ser introducido en el bloque *Black Box* de System Generator. Consta de un puerto bidireccional para la entrada y salida directa de los datos de Dynamixel, y de la salida PWM.

[2]Nuestro reloj interno es de 100 MHZ, es decir, su período es de 10 ns. Si multiplicamos ese tiempo por 2000 obtenemos 20000 ns, o lo que es lo mismo, 0.00002 s, que se corresponden con una frecuencia de 50 kHz. A continuación veremos cómo llegamos de ahí a los 500 Hz.

Se aprecia como la salida PWM tiene el deseado periodo de 2 ms (500 Hz), y la correspondencia entre la entrada de dutycycle y el dutycycle real de la señal final. Así pues, el módulo de creación del PWM sería operativo.

2.2.6.2.1.7. Resultado de las pruebas

Una vez terminado el inciso sobre el PWM, centrémonos de nuevo en el programa de la UART. Para probar el buen funcionamiento de mi código, necesitaba un nexo entre la FPGA y MATLAB®. Entre las varias opciones que surgieron, decidí usar un Arduino, ya que me permitía conectar directamente las salidas Pmod™ de la ZedBoard a las entradas digitales de Arduino. Además, la facilidad de manejo de la plataforma la hacían perfecta para probar el código con facilidad.

No obstante, las pruebas no fueron como esperaba, ya que los resultados no tuvieron nada que ver con los de simulación: los datos obtenidos no concordaban con ninguna palabra de 10 bits que pudiera esperar.

Tras esto estuve razonando porque no había funcionado el código, y pensé en varias posibilidades:

- La velocidad "lenta" de toma de datos de Arduino: en mi caso uso Arduino MEGA 2560, y este tiene un procesador de 16 MHz, mientras que la FPGA tiene un reloj para la parte de PL de 100 MHz, tal y como hemos comentado anteriormente.
- Los tiempos de comprobación de bits y bytes dentro del programa. Puede que fueran demasiado justos, y por pequeñas imperfecciones de sincronización se produjeran fallos.
- Algún problema con los datos enviados por la OpenCM9.04.

Decidí comprobar esto último en primer lugar, ya que de ser cierto podría tener un mayor problema.

2.2.6.2.2. Pruebas intermedias: empleo del osciloscopio

Así pues decidí probar mi proyecto **inout**, para ver si transmitiendo los datos de la OpenCM9.04 a la FPGA y sacando estos intactos por el puerto Pmod™ de la ZedBoard, podía leerlos con el osciloscopio.

Tal y como comenté durante la explicación de los elementos básicos del proyecto, para esto se empleó el software PicoScope 6 junto con el modelo 4824 de Pico Technology. Este osciloscopio se conecta al PC mediante USB, y permite monitorizar la entrada donde se conecte desde la pantalla. Es una herramienta ciertamente útil, y como se verá, solucionó muchos de mis problemas.

Una vez implementado el programa VHDL en la ZedBoard, y conectada la microcontroladora[1] a ésta, el resultado de la toma de datos fue el siguiente:

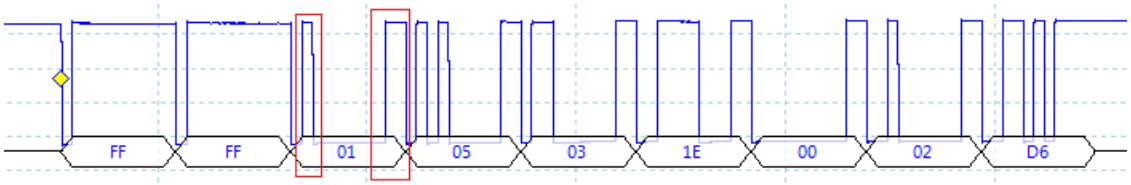


Figura 35

COMENTARIOS

[1]El programa introducido en la OpenCM9.04 fue el ejemplo básico *Basic* de Dynamixel, pero modificado para que enviara como posición objetivo 512 (0x200 en HEX), es decir, 180º. Posteriormente se comentará este código, ya que tiene algunos requerimientos. Podrá encontrarse en el **Anexo I: Código**.

Como vemos en la Figura 35, el protocolo se cumple perfectamente. Primero llegan dos 0xFF, posteriormente la ID del servo de destino, 1; luego la longitud del paquete, 5; la instrucción, 3; los parámetros: 0x1E para el comienzo de dirección en la tabla de control, 0 como primer byte de la posición y 2 como segundo (para formar 0x200); y por último el *checksum*, 0xD6.

Un detalle que no pasé por alto fue que, tal y como puede verse comparando el contenido de los dos rectángulos rojos de la Figura 35, el protocolo empleaba **dos** bits de stop, en lugar de uno como se indica en la web (ROBOTIS(b), s.f.). En la explicación del proyecto de la UART ya se hizo mención a esto, pero no se dijo cómo se descubrió el error, por lo que me parecía importante mencionarlo.

Además, aunque no se aprecie en la imagen, comprobando la escala de tiempos, cada bit ocupaba 1 µs, tal y como pensaba.

Toda esta información me permitió crear un nuevo proyecto VHDL, que explotaría las características del conocimiento del protocolo para conseguir un funcionamiento correcto, reduciendo la complejidad del código. Por ello, dejé apartado el proyecto de la UART (propiamente dicha), y comencé un nuevo diseño para proseguir con las pruebas.

2.2.6.2.3. Alternativa II : Diseño VHDL específico para Dynamixel

Una vez había confirmado que los paquetes de Dynamixel eran enviados correctamente y tal y como esperaba, decidí diseñar un programa que, en cierta forma, aunara los distintos bloques que conformaban la UART anterior, pero en uno solo por simplicidad, y cuyo objetivo específico fuera comprobar, con la ayuda del osciloscopio, si era capaz de detectar la llegada de los distintos bytes de un paquete concreto. En este caso, el paquete fue el mismo de la Figura 35.

Así pues, como es entendible, el programa sufrió múltiples modificaciones desde su comienzo hasta su puesta a punto final, ya que a cada paso se realizaban comprobaciones con PicoScope, con tal de asegurar las bases de su funcionamiento. Por ello, en esta sección lo que explicaremos es el diseño final, aunque hagamos cierta mención a ideas clave que dieron forma a esta solución.

El código de este proyecto puede encontrarse en el **Anexo I**, con el nombre de **hduart**.

En primer lugar comentemos las entradas y salidas del bloque que conforma en programa: como entradas tenemos la fuente de reloj y el reset (cobrará especial importancia en este diseño), la entrada de bits de Dynamixel, **DATA** (realmente este puerto es bidireccional como anteriormente, pero ya que sólo lo explotaremos como entrada lo adjunto a esta sección); y la señal de selección de funcionamiento del puerto **DATA**, **DIR_PORT** (se deja siempre a nivel alto para garantizar el comportamiento de entrada de **DATA**). Como salidas podemos distinguir entre **rxd_out**, que nos permite extraer del programa los bytes de información junto con un número de 4 bits que hace las veces de indicador del tipo de byte (por tanto, **rxd_out** tiene 12 bits); y **salida**, que no tiene utilidad para el funcionamiento final del programa, pero se empleó para comprobaciones de osciloscopio. Estas entradas y salidas están en el archivo de extensión .ucf llamado **ucefe.ucf**.

Una vez se conocen los puertos, entremos a comentar los distintos componentes y procesos que conforman el bloque VHDL:

- Reset: este reset está basado directamente en aquél explicado para la UART anterior, por lo que no entraré en su diseño. Destacar de nuevo que tendrá un papel importante, pero la razón la veremos más adelante.
- Actualización del estado: de nuevo, al igual que en todos los demás bloques VHDL que hemos visto, separamos la parte secuencial del programa de la parte combinacional. Así pues, este proceso se encarga de, en primer lugar, reiniciar los valores de todas las señales necesarias en caso de pulsar el reset; y en segundo lugar, de actualizar dichas señales a su próximo valor al detectarse un flanco de subida de reloj. Estas señales las iremos presentando conforme sea necesario.
- Máquina de estados: la función de este proceso es conformar los bytes del paquete de Dynamixel a partir de los bits que nos van llegando por el puerto **DATA** (que almacenamos en la señal interna **data_in**). Para ello, se ha diseñado una máquina de estados, que depende de una señal de tipo

estado_t, tipo creado con la función ***type***, y que puede tener los estados siguientes:

- IDLE: se corresponde con el estado de reposo, en el que esperamos recibiendo bits a nivel alto (ya que el canal es de alta impedancia), hasta que se detecta un flanco de bajada en la señal **data_in** antes mencionada. En este primer estado es importante presentar las variables a las que daremos valor en el proceso (en el código aparecerán con el sufijo **_next**, ya que en los bloques combinacionales se da valor al estado próximo):

  - **enable**: en los bits que tienen verdadera información, es decir, aquellos que no son *start bit* o *stop bit*, empleamos esta señal para que a la hora de recoger el bit, tan sólo podamos hacerlo una vez, asegurando así el buen funcionamiento de este bloque. De no usar este *enable*, como veremos después, meteríamos más de una vez el bit dentro de la señal de salida de bytes.
  - **cont**: empleamos este contador para, valga la redundancia, contar hasta 99, de forma que estemos en cada estado durante 100 ticks de reloj, lo que se traduce en 1 µs por bit, que es el tiempo que ocupa cada bit de Dynamixel. Así aseguramos de nuevo el buen funcionamiento del proceso.
  - **e2**: este segundo *enable* no será empleado en este proceso, sino en el siguiente, para asegurar que la comprobación o actuación sobre los bytes se realiza sólo cuando estos están completos.
  - **rxd**: en esta señal de 8 bits es donde vamos guardando los bits individuales de Dynamixel hasta completar el byte.
  - **estado**: esta es la señal de tipo *estado_t* que nos permite saber y comprobar en qué estado nos encontramos.

  Así pues, mientras no detectemos el flanco de bajada en **data_in**, nos dedicamos a mantener los valores de estas señales a sus valores de inicio, es decir, a cero, y a mantener el estado actual. Cuando llega dicho flanco, pasamos al siguiente estado.
- BIT_START: es el estado en el que esperamos que pase el bit de start que acaba de comenzar tras el flanco de bajada detectado anteriormente. Lo único que haremos será contar hasta 99 para pasar al siguiente estado.
- BIT1/BIT2/BIT3/BIT4/BIT5/BIT6/BIT7/BIT8: estos son los estados en los que debemos almacenar un bit de información. Ya que la siguiente estructura se repite en todos ellos, se comentará una única vez. Su funcionamiento básico se extrae del estado anterior: contar hasta 99 para pasar al siguiente estado, pasando de uno a otro hasta llegar al primer bit de stop. No obstante, la diferencia radica en que, en estos 8 estados que conformarán el byte, mientras el contador esté entre 45 y 55, y **enable** sea 0, guardaremos el bit en la señal **rxd**, introduciéndolo por la izquierda. Ya que dentro de esta posibilidad ponemos **enable** a nivel alto, se hace clara su función: tal y como dijimos antes, garantizamos gracias a **enable** que sólo introducimos una vez el bit en **rxd**. Por ello también es importante, cuando el contador alcance el valor de 99, poner **enable** a 0 y pasar al siguiente estado.
- BIT_STOP1/BIT_STOP2: estos dos estados representan el paso de los bits de stop. Al igual que con el bit de start, lo que hacemos es contar hasta 99 para pasar al siguiente estado. Sin embargo, en el BIT_STOP1 hay una diferencia: mientras estemos en este estado, **e2** estará a nivel alto. Tal y como dijimos antes, esto permitirá que el proceso siguiente "sepa" que el byte está listo. Lo colocamos en ese estado porque sabemos que el byte se habrá formado.

➢ Proceso de salida: al llegar a este proceso es fácil darse cuenta que este bloque sigue la misma estructura de programación de tres bloques principales: actualización de estado, fsm y proceso de salida. En este último, sin embargo, no damos la salida directamente, sino que es un paso previo a eso, tal y como veremos a continuación. Al igual que con la máquina de estados, es conveniente introducir las señales con las que trataremos, para mejor comprensión del código:

- **rxd**: es la señal que contiene los bits cada byte de los paquetes de Dynamixel, y que es cargada de valores en el proceso antes explicado.
- **e2**: de nuevo una señal proveniente del proceso anterior. En este caso, es el *enable* que nos indica la disponibilidad de un byte completo.
- **cont2**: este contador hace las veces de variable de estado. Ya que el proceso es una estructura if-elsif-else y cada caso se corresponde con un valor del contador, es como si éste nos indicara en qué estado nos encontramos y que byte ha llegado, y de hecho así es, tal y como veremos más adelante. No se ha usado una estructura switch-case como en el proceso anterior, ya que el hecho de "encontrarnos en un estado u otro" depende de más variables, como son **rxd** y **e2**.
- **cont3**: se corresponde con el contador que implementa el paso de "un estado a otro". En la explicación posterior veremos que implica esta afirmación.
- **salida**: esta no es una señal interna, sino que se corresponde con una salida física del sistema. Nos indicará que se ha completado un byte, tal y como hace **e2**, pero nos permitirá medirlo desde el exterior.
- **write_en**: esta señal hace las veces de *enable* para la salida final de la información. Posteriormente veremos cómo se lleva a cabo esta tarea.
- **checksum**: en esta señal iremos almacenando el valor de los bytes necesarios para realizar al final la suma de comprobación, y verificar que el paquete no tenga fallos.

Así pues, una vez sabemos para que se emplean estas señales, entremos a ver el funcionamiento del proceso:

- La primera comprobación se realiza con el objetivo de captar el primer byte del paquete, 0xFF. Así pues, si **rxd** contiene este valor, **cont2** vale 0 y **e2** está a nivel alto, significará que el primer byte en llegar es correcto. Entonces, contaremos hasta 98 (así dejamos una mínima holgura por sí hay algún error de sincronización con los 99 contados en el proceso anterior) y entonces aumentaremos **cont2** en una unidad. Así, el próximo ciclo de reloj no cumpliremos las condiciones (**cont2** no será 0) y deberemos esperar al siguiente byte para cumplir la siguiente condición. Este comportamiento de **cont2** se repetirá en el resto de comprobaciones. Cabe mencionar que **salida** se pone a nivel bajo ya que, como veremos después, se deja ese canal a nivel alto por defecto. Esto se hará en todos los demás estados.
- En caso de no cumplir la condición anterior, comprobamos si **rxd** es 0xFF, si **cont2** contiene un 1 y si **e2** está a nivel alto. De cumplirse todo esto, sabremos que ha llegado el segundo 0xFF del paquete. Así pues, repetiremos el comportamiento anterior, contando hasta 98 para aumentar **cont2**.
- Si las condiciones anteriores no se cumplen, se comprueba si **cont2** vale 2 y **e2** está a nivel alto. En ese caso, estaremos en la condición de llegada del byte indicador de la ID, y es por esto que no comprobamos el valor de **rxd**: no sabemos de antemano cuánto valdrá, ya que esto es lo que deseamos averiguar. Por ello, este caso es algo distinto a los anteriores: aunque se sigue contando hasta 98 para pasar aumentar el valor del contador **cont2**, en caso de que **cont3** sea menor que uno llevaremos a cabo dos acciones (que sólo queremos que se ejecuten una vez), poner **write_en** a nivel alto, para indicar que es un byte de información útil, y añadir el byte a la señal **checksum**. Así, sólo pasamos una vez el byte de la ID y no lo repetimos en la *checksum*. Aparte de esto, el comportamiento es igual al de las condiciones previas.
- En caso de que no se cumpla nada de lo anterior se comprueba si **rxd** vale 5, si **cont2** contiene un 3 y si **e2** es 1. Si se cumple, habrá llegado el byte de la longitud del paquete. El comportamiento en este caso será idéntico al caso anterior, excepto porque cuando **cont3** sea menor que 1 tan sólo añadiremos el byte a **checksum**, y no podremos **write_en** a 1, al no ser un byte de información útil. Este comportamiento se repetirá para los casos de llegada del byte de instrucción (**rxd** = 3 / **cont2** = 4 / **e2** = 1); y del primer parámetro, indicador de la dirección de comienzo de la tabla de control del servo (**rxd** = 0x1E / **cont2** = 5 / **e2** = 1).
- Si ninguno de los casos anteriores se da, tendremos que comprobar que haya llegado el primer byte de información real de posición, que nos indica los ocho valores de bits menos

significativos de esta posición (**cont2** = 6 / **e2** = 1). De nuevo, como en el caso de la ID, no comprobamos el valor de **rxd**, ya que lo que debemos hacer es ver cuánto vale, no comprobarlo sabiéndolo de antemano. El comportamiento también será idéntico: contar con **cont3** hasta 98 para aumentar **cont2**, pero cuando **cont3** sea menor de 1, emitiremos un pulso de **write_en** y añadiremos el byte a **checksum**. Esto se hará igual para el caso de llegada del byte que contiene los ocho bits más significativos de la posición, pero comprobando si **cont2** es 7 y si **e2** es 1.

- En caso de que nada de lo anterior se haya cumplido, chequearemos si **cont2** contiene un 8, si **e2** está a nivel alto y haremos la comprobación del *checksum*, para garantizar la corrección de los bytes llegados anteriormente. Para ello, comprobamos si **rxd**, es decir, el último byte que nos ha llegado de ese paquete de Dynamixel, es igual a los 8 bits menos significativos de nuestra señal interna **checksum** (que hemos ido llenando en las comprobaciones con los bytes anteriores) pero negada, es decir, con sus bits invertidos. Si todo esto se cumple, contaremos hasta 98 para pasar al estado de espera del primer 0xFF del siguiente paquete. Aquí es donde resalta la importancia del reset en nuestro diseño. En caso de que la suma de comprobación sea errónea, el programa se bloquea, es decir, no hay forma de volver al estado de espera del primer 0xFF. Esto se ha hecho para evitar fallos en el movimiento del robot virtual. Así pues, pulsando el reset reiniciamos el sistema y los bytes vuelven a ser procesados.
- Si no se ha cumplido ninguna de las condiciones anteriores, seguramente significará que **e2** está a nivel bajo. En este caso, mantendremos el valor de las señales, excepto por **salida**, que estará a nivel alto tal y como comentamos anteriormente.

- FIFO: en una primera instancia, cuando estaba probando la primera versión de mi diseño, me encontré con un problema: la placa Arduino no era capaz de leer los bytes que le enviaba con suficiente precisión de forma automática, pero sí que le estaban llegando, ya que cuando especificaba que encontrara un byte en concreto era capaz de hacerlo. Así pues, entendí que esto se debía a la velocidad de envío de los bytes por parte de la FPGA, o más bien, al tiempo que se mantenía ese byte en el canal.
  Por ello, tras probar varias opciones que se me ocurrieron para alargar este periodo de exposición de los bytes, hallé la solución ideal: una FIFO. Con ella podría introducir los bytes de información útil en el momento que quisiera, y sacarlos como salida también cuando yo quisiera, durando en el canal el tiempo que fuera necesario.
  Así pues, empleé una herramienta de Xilinx® ISE Design Suite, llamada **IP (CORE Generator & Architecture Wizard)**, que permite añadir a nuestro proyecto IP COREs diseñados por Xilinx, y que implementan distintas funciones. Entre ellos, disponemos de un *wizard* para generar FIFOs, por lo que decidí emplearlo. En él se nos ofrecen varias opciones de implementación, pero en este caso he elegido que funcione como registro de desplazamiento, ya que encaja perfectamente con la funcionalidad que busco. Además, es importante ver el tamaño de la FIFO: la profundidad de memoria es menos importante, ya que se irá vaciando progresivamente, pero por si acaso le he dado un tamaño de 1024 posiciones de memoria. En cuanto al ancho de cada posición de memoria, he ajustado la FIFO para que éste sea de 12 bits, de los cuáles los 8 menos significativos se corresponderán con el byte de información útil **rxd**, y los 4 más significativos con la variable **cont2**, que nos permitirá identificar en Arduino que tipo de byte estamos recibiendo.
  Una vez conocemos cómo se organiza la salida de datos mediante la FIFO, se hace más clara la utilidad de la señal **write_enable** que antes comentamos: ésta se pone a nivel alto una única vez por byte de información útil, por lo que hará las veces de *enable* de escritura de la FIFO. Para entender cómo se ejecuta la lectura, debemos entrar a hablar del siguiente proceso.
- Generación del *enable* de salida: con tal de alargar el tiempo que se encuentra cada byte en el canal, he diseñado un proceso bastante simple, que básicamente cuenta hasta cierto número, y al llegar a esa cifra, da un pulso a nivel alto en el *enable* de lectura de la FIFO, mediante la señal **read_en**. En mi caso fui probando varias cifras de forma heurística, hasta que ajustando el contador para llegar a 128000 conseguí un comportamiento correcto y constante de la salida de bytes.

- Entrada de datos: los paquetes de Dynamixel nos llegan, al igual que en la UART, por el puerto bidireccional DATA, dejando el canal en alta impedancia.

Así pues, ya comprendemos cómo funciona este programa, y en cierta forma, esbozos del proceso seguido para desarrollarlo.

2.2.7. Implementación de pruebas (II)

Tal y como dije en el punto anterior, el diseño de este programa fue paulatino, y las pruebas se fueron llevando a cabo paso a paso. Por ello, veamos un resumen de la evolución que éstas siguieron (aunque no estarán reflejados todos los pasos seguidos, sí que veremos los más relevantes):

2.2.7.1. 1ª prueba : comprobación de lectura de bytes

Así pues, la primera versión del diseño no estaba destinada a extraer la información de los bytes de Dynamixel hacia el exterior para su procesado. El objetivo en un primer momento fue, tal como avancé anteriormente, chequear que la máquina de estados y el proceso de salida funcionaban correctamente por medio de la señal **salida** (descrita en el anterior punto). Brevemente, recordar que ésta estaba por defecto a nivel alto, y que cuando se cumplían las condiciones de salida, se ponía a nivel bajo. Para esta prueba se cargó en la OpenCM9.04 el programa Basic ya explicado enviando la posición de destino de 180º (0x200). Por ello todos los bytes del paquete eran conocidos, e incluso en comprobaciones como la ID o los bytes de posición se exigía que los bytes entrantes fueran los necesarios para ese paquete. Así chequearíamos de forma fiable que nuestro programa es capaz de distinguir los bytes, y reconocer que llegan correctamente.

Se empleó el osciloscopio con el software de PicoScope para dilucidar la corrección de este comportamiento, obteniéndose el siguiente resultado:

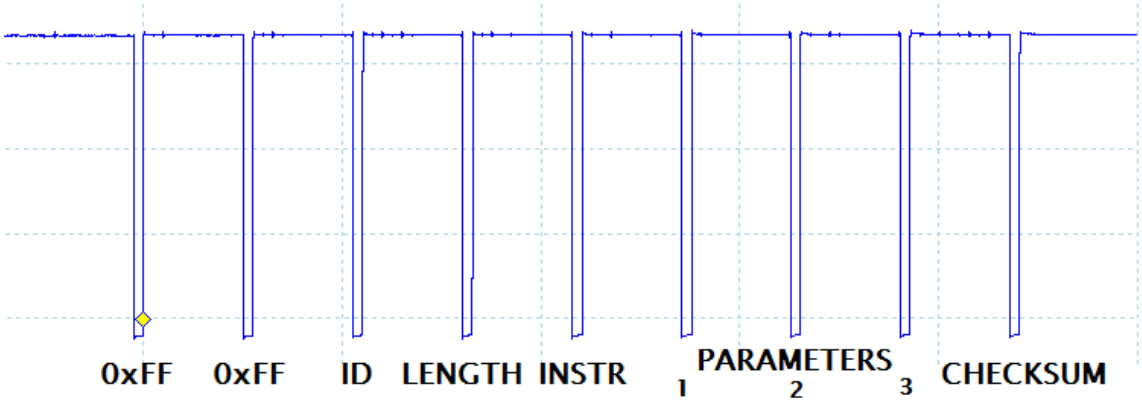


Figura 36

Como podemos apreciar, se producen nueve pulsos a nivel bajo, correspondiéndose uno a uno con los bytes del paquete de Dynamixel enviado. De esta forma se demuestra el éxito de esta primera prueba.

2.2.7.2. 2ª prueba : recepción y decodificación de bits en Arduino

Esta nueva prueba, como nos indica el título, constaba de dos partes:

- En primer lugar, conseguir capturar los bits enviados desde la ZedBoard de forma correcta, para poder trabajar con ellos.
- Segundo, procesar dichos bits de forma que consigamos decodificar la posición de destino y la ID del servo al que va dirigida, para enviárselo a MATLAB®.

Para entender cómo se resolvieron ambas cuestiones, es recomendable revisar el código implementado en Arduino[1]. Éste podemos encontrarlo en el **Anexo I**.

En primer lugar podemos observar una serie de definiciones de variables globales, entre las que destacan aquellas usadas para referirnos a los pines digitales de Arduino de una forma más entendible (desde **RXD_0** hasta **cont_3**), la constante **val**, que almacena el valor 9999 (luego veremos con qué objetivo), las variables donde almacenamos el valor del bit que nos llega por cada entrada digital (desde **RXD0** hasta **cont3**) o las variables **pos** e **id**, donde acaba almacenándose el valor que enviaremos por puerto serie. El resto sirven en su mayor parte para procesos y cuentas internas.

En cuanto a la configuración, basta con iniciar una comunicación por puerto serie, que usaremos para comunicarnos con MATLAB®. Ésta la he configurado para 250000 baudios, el máximo que ofrece Arduino, con el fin de evitar los máximos retrasos posibles por el envío de datos.

Así pues, lo que hacemos en la función *loop* del programa es ejecutar otra función llamada **Byte2** de forma continua, y que implementa el siguiente comportamiento:

- En primer lugar, realizamos una lectura del valor de los pines de entrada digital, con tal de capturar los bits que representan el byte de información útil (**RXD0** – **RXD7**) y aquellos que contienen la información sobre el tipo de byte que ha llegado (**cont0** – **cont3**).
- Una vez tenemos estos bits realizamos una decodificación de binario a decimal, obteniendo la posición o ID y el contador que nos indica cuál de los dos es.
- A continuación, dependiendo del valor de contador que hayamos recibido, guardaremos el byte en la variable **id** de corresponderse con la ID del servo de destino (**cont** = 2), en la variable **byte_0** de ser el primer byte de la palabra que conforma la posición (**cont** = 6) o en la variable **byte_1** de corresponderse con el segundo byte que conforma la palabra de posición (**cont** = 7). Además, para garantizar que sólo entremos una vez en cada caso por conjunto ID-posición, implementamos la filosofía de *enables* del último bloque VHDL explicado: comenzamos con cada *enable* a ceroy, cuando entramos en una condición, ponemos el *enable* correspondiente a uno, de forma que la condición de que éste se encuentre a cero ya no se cumplirá, entrando una únca vez.
- Una vez los tres *enables* se encuentren a uno comenzamos el procesamiento. Primero, revisamos que la ID que ha llegado no sea la misma que recibimos la última vez (esto se hace por cómo está implementada la recepción en MATLAB®, por lo que luego veremos el porqué de esta comprobación). Si esto se cumple, calculamos en primer lugar la ID de salida. Para esto podemos ver que sumamos la constante **val** con la ID recibida la última vez, y almacenada en **pre_id**. La razón de sumar **val**, de valor 9999, es facilitar la diferenciación de las ID y las posiciones en MATLAB®. Podríamos decir que es una codificación para discernir entre ambas. Además, la razón de emplear la ID anterior puede parecer ilógica, pero está tomada en función de la implementación seguida en el archivo **Basic** implementado en la OpenCM9.04. En este programa se emplea una mínima espera entre cada envío de una posición para un servo, ya que de no realizar dicha espera se producen grandes fallos en la recepción de datos en Arduino. No obstante, ese

COMENTARIOS

[1]En este punto me gustaría ampliar las razones de uso de Arduino. Ésta no fue la primera opción: en un principio, se exploró la posibilidad de comunicar directamente la ZedBoard con el ordenador, barajándose dos opciones: el USB OTG o la UART. Con el primero surgió un problema sin solución, y es que en la propia documentación de ZedBoard se dice que una vez ésta había sido diseñada, se encontró un fallo de temporización en la interfaz que controla el USB, por lo que no se recomienda para nuevos diseños con Xilinx Zynq. Así pues, se trabajó en la segunda opción, y se consiguieron algunos progresos, pero el resultado final no llegó y la comunicación entre placa y ordenador mediante la UART del *processing system* tuvo que ser desechada. Finalmente, como alternativa, se empleó Arduino para intentar solucionar el problema. Sin embargo, considero interesante y posible la comunicación empleando la UART, e insto a quien continúe este proyecto a trabajar en este interesante aspecto.

lapso de tiempo introduce un cierto retardo, que se acrecienta con el retraso que induce la lectura de bits en los puertos de entrada de Arduino. Esto en conjunto produce que, al empezar a leer los bits de contador, se produzca un cambio en el envío, y al leer el byte, se esté recibiendo realmente el byte siguiente. En resumen, se produce una descoordinación entre IDs y posiciones. No obstante, su arreglo es muy sencillo: tanto para calcular la ID a enviar como para calcular la posición se emplean dos elementos de la anterior iteración, **pre_id**, que ya hemos explicado, y **pre_byte_0**. Con este arreglo, los datos enviados por puerto serie coinciden exactamente con los que cabría esperar. Así pues, la posición a enviar se calcula sumando el **byte_1** multiplicado por 256 (es el byte más significativo de la palabra) más **pre_byte_0**. Por último, volvemos a iniciar el valor de los *enables* a 0, para empezar la siguiente recepción.

Así pues, una vez explicado el código, se entiende mejor como recibimos los bits y decodificamos la información. Por último, cabría mencionar un detalle sobre esta recepción de bits: cuanto más aumentábamos el tiempo de permanencia de la información en el canal (que vimos que se hacía con proceso en VHDL), mejor y con menos errores llegaban los bits. Digo esto porque el ajuste del valor del contador de espera fue un proceso paulatino.

Finalmente, tras todo este proceso de diseño del nexo entre la ZedBoard y MATLAB®, el resultado enviado por puerto serie fue el siguiente:

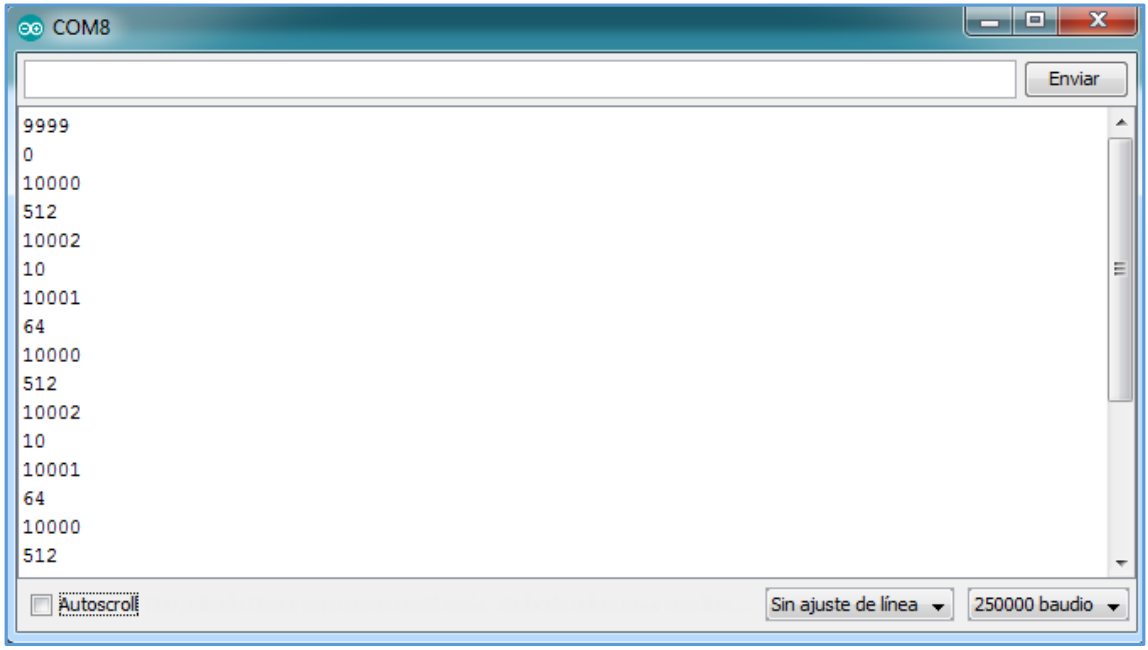


Figura 37

Lo que observamos en la Figura 37 es el terminal de puerto serie facilitado por la IDLE de Arduino. Se enviaron desde la microcontroladora OpenCM9.04 tres[1] paquetes distintos, uno para cada servo (Servo 1 – Posición 180º (512) / Servo 2 – Posición 22.5º (64) / Servo 3 – Posición 3.5º (10)), que podemos apreciar en la imagen. La llegada del primer doble cero (9999+0 (ID) y 0 (posición)) se produce por defecto por cómo está implementado el programa, pero no afecta al

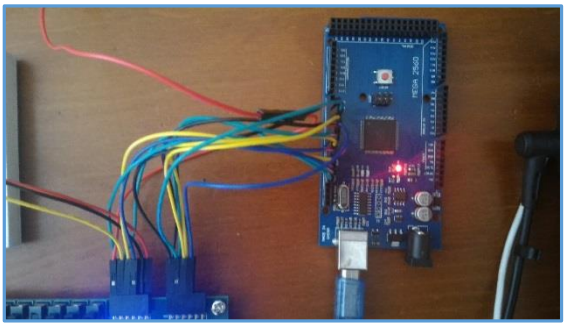


Figura 38

COMENTARIOS

[1]La prueba con tres paquetes siendo cada uno para un servo distinto fue la última implementada. No obstante, y como es obvio, previamente se realizaron pruebas enviando un solo paquete para un solo servo, y desde el éxito de esa prueba se fue evolucionando hacia este resultado final.

desarrollo del proceso, ya que es filtrado en MATLAB®.

En la Figura 38 podemos observar el conexionado entre la ZedBoard y Arduino.

2.2.7.3. 3ª prueba : procesado de información en MATLAB®

Una vez completada la prueba anterior, ya habíamos conseguido comunicar la microcontroladora OpenCM9.04 con la ZedBoard, y ésta con Arduino, de forma satisfactoria. Así pues, quedaba el último paso: comunicar Arduino con MATLAB®. Para ello se emplearon funciones de trabajo con puerto serie, de forma que en vez de recibir los datos enviados en el terminal de la IDLE de Arduino, el destinatario fuera el propio MATLAB®.

Para esto se diseñó un programa en Simulink® que nos permitiera pasar de los datos "crudos" sin procesar a las posiciones de cada servo, y pasar estas a nuestro robot virtual en VRML. Este programa llamado **prueba_simulink.slx** puede verse en el **Anexo II: Simulink**, junto al código de todas las funciones que usa, que estará en el **Anexo I: Código**.

Al igual que con las demás pruebas implementadas y ya explicadas, el proceso de creación de este módulo fue paulatino, pero en esta sección explicaremos el resultado final. Así pues, procedamos a ver las distintas partes del archivo Simulink®, describiendo sus funciones en orden de aparición.

En primer lugar nos centraremos en una función llamada **com_matlab_arduino**, implementada en el bloque *Interpreted MATLAB Function* que se aprecia en el *top level*.

Esta función tiene una única entrada, el tiempo de simulación, y cinco salidas: **pos1** (es el valor de posición para el servo con ID 1), **pos2** (igual para el servo con ID 2), **pos3** (igual para el servo con ID 3), **stop** (cuando tiene un valor distinto de 0 se para la simulación) y **datos** (permite sacar los datos "crudos" recibidos por puerto serie). Tras esta introducción, entremos en la funcionalidad implementada:

- En primer lugar se comprueba si el tiempo de ejecución es menor de $1e^{-8}$ s. De esta forma, las acciones que llevemos a cabo dentro de la condición sólo se ejecutarán una vez: cuando el tiempo de simulación sea cero. Así pues, cuando esto se cumple, realizamos tareas de iniciación del sistema: revisamos si el puerto serie (en mi caso el COM8) está abierto, y de estarlo lo cerramos, creamos una variable de puerto serie, que debe ser de tipo **persistent**. Este tipo de variable mantiene su valor entre iteraciones, lo que será necesario para no perder la variable de puerto serie. En el programa usaremos una variable más de este tipo: **id**. Ésta será donde guardemos el valor de la ID que leamos por puerto serie (a continuación veremos porqué debe ser persistente). Además configuramos el puerto para una velocidad de 250000 baudios, como en Arduino; y abrimos el puerto.
- A partir de aquí empezamos con las acciones que se ejecutan normalmente en el programa. La primera, obviamente, es leer un dato del puerto serie. Este se lo pasamos también a la salida **datos**, como comentamos previamente.
- A continuación encontramos dos estructuras condicionales, cada una con una función determinada:

    - En la primera comprobamos si el dato recibido menos 9999 es mayor que cero. Así pues, esta es la etapa de decodificación de la información. En caso de que se cumpla, sabremos que se trata de una ID, ya que en Arduino le sumamos 9999 a su valor, por lo que guardaremos el valor recibido (menos 9999) en la variable **persistent id**. A su vez ponemos el valor de las posiciones a 50000, que será la cifra que usaremos para representar valor nulo y sin información. En caso de que no se cumpla la condición sabremos que el dato recibido es una posición, así que la almacenamos en la variable **pos_pre**.
    - En la segunda estructura condicional tenemos una especie de máquina de estados, dependiente del valor de **id**. Aquí es donde vemos la importancia de que **id** sea persistente: cuando nos llega una ID, la almacenamos en esa variable y **pos_pre** valdrá 50000. De esa forma, al entrar en esta estructura condicional, sea cual sea la ID, la variable de salida de

posición de los tres servos valdrá 50000, indicando que no tenemos aún la posición para esa ID. En la siguiente iteración, tal y como hemos planteado este código y el de Arduino, nos llegará la posición para la ID anterior, guardándola en **pos_pre**. Como la ID se ha mantenido almacenada en **id**, cuando entramos a la máquina de estados, asignamos el valor de posición al servo correspondiente a esa ID, y a los otros dos les damos el valor de defecto de 50000.

Así, con estas dos estructuras condicionales conseguimos extraer la posición de cada servo por la salida correcta.

- Por último, en caso de que el tiempo de simulación llegué a cierto número (en mi caso he puesto 5 segundos, pero esto es orientativo), mandaremos un 1 por **stop** para parar la simulación, y cerraremos todos los puertos abiertos.

Con esta explicación se aprecia mejor cómo pasamos de los datos "crudos" a dar a cada servo su valor de posición deseada. Como ejemplo para afianzar aún más la comprensión sobre este bloque, recomiendo ver la Figura 39 (a), situada al final de la descripción de los programas de MATLAB®. Tras esto, sigamos desglosando el archivo **prueba_simulink.slx**.

Fijándonos de nuevo en la estructura del diseño Simulink® vemos como a continuación de la función ya comentada, tras las tres salidas de posición, tenemos tres subsistemas. Estos son equivalentes, es decir, implementan las mismas funciones (aunque son distintos ficheros para evitar problemas de MATLAB®) pero cada uno para uno de los servos. Por ello, entraremos a comentar uno sólo, en concreto, el subsistema para el servo de ID 1.

En un primer vistazo, observamos dos bloques de ganancia. Estos implementan la conversión de posiciones en formato Dynamixel, que abarcan de 0 a 1023 posiciones, a radianes, que es el formato de trabajo angular para VRML. También vemos dos bloques *To Workspace*, uno previo a las ganancias y otro posterior. El previo es el que transmite los datos de posición deseada (en formato Dynamixel), que antes comenté que tenía un ejemplo en la Figura 39 (a). El bloque posterior guarda los valores de posición en radianes, y su ejemplo podremos apreciarlo en la Figura 39 (b).

Tras esto observamos dos bloques *Interpreted MATLAB Function*, teniendo cada uno una función específica, pero a su vez complementaria a la alterna:

- El primer bloque, llamado *calc_dif_ang1* (el número depende del subsistema) contiene la función **posicion** (aunque es algo confuso, el nombre de este archivo no lleva número. Para el caso del segundo subsistema sería **posicion1** y para el tercero **posicion2**. Igual pasará con la función siguiente). Esta función tiene dos entradas, los valores de posición destino del servo en radianes y el tiempo de ejecución; y una única salida, **pos_out**, que almacena cuánto debemos girar con respecto a la posición actual (de reposo, es decir, el giro total que tendrá que afrontar el servo). Veamos pues la funcionalidad que implementa este módulo:

    - En primer lugar observamos que se resta π a la posición deseada. Por ahora haremos cómo si esta línea no estuviera, ya que se puede entender la funcionalidad del bloque de igual manera, y es preferible explicar el sentido de esta sustracción más adelante.
    - Así pues, obviando la resta de π, lo primero que haríamos sería la inicialización. De ser el tiempo menor de $1e^{-8}$ s, damos a la variable de **persistent next_pos** el valor de 0. En ella será donde almacenaremos la diferencia de posición entre actual y deseada del servo.
    - Una vez inicializado el sistema, comprobamos si el valor de posición entrante es menor de 3π. Esto se hace para eliminar los valores de 50000 (ese es su valor en formato Dynamixel, en radianes será menor pero seguirá siendo mayor de 3π), de forma que sólo nos quedemos con los valores de posición correctos. Una vez filtrados estos valores erróneos comprobamos que la posición a la que se nos pide ir sea diferente de la actual, y en ese caso almacenamos en la variable de salida **pos_out** la diferencia entre posiciones (actual y destino). En caso de que no se cumplan las condiciones, se indica que no haya cambio de posición con un 50000.

- El segundo bloque, denominado *slew_rate1* contiene la función **slew_rate**. Esta función permitirá que el cambio de posición no sea instantáneo, sino que se irá produciendo de forma paulatina. La

función tiene de nuevo dos entradas, siendo éstas el tiempo de simulación y la diferencia de ángulo a girar dada por la otra función, y una salida, **out_in**, que irá dando valores intermedios entre la posición de origen y la posición de destino, de forma que las articulaciones del robot virtual giren de forma progresiva. Su funcionalidad es la siguiente:

- En un primer momento inicializamos las variables de tipo **persistent** del bloque: **cont**, que os permitirá contar hasta el número de puntos intermedios que deseemos; **top**, que almacena el nuevo giro que nos ha llegado desde la función anterior; **div**, que guarda el valor de la división entre el nuevo giro y el número de puntos; y **current**, que va almacenando la cuenta de la posición que debemos darle al servo.
- Así pues, si nos llega un nuevo giro que sea distinto de 50000 (no hay cambios), guardamos en **top** este giro, calculamos cuándo debemos girar cada iteración según los puntos intermedios elegidos y reiniciamos **cont** y **current**.
- Una vez hecho esto, y mientras el giro siga siendo el mismo, iremos contando con **cont** de forma que hasta que lleguemos al número de puntos más uno vamos sumando una división más a la posición actual **current**. Cuando el contador sobrepase el umbral, mantenemos dicha posición, la cual en todos los casos es almacenada en la variable de salida **out_in**.

Así pues, entendiendo estas dos funciones, comprendemos de forma completa y precisa cómo llegamos de los datos "crudos" a las posiciones del robot virtual. Los bloques de **Simulink 3D Animation** no serán explicados, ya que se hizo en la sección **2.2.5.3**.

2.2.7.4. Resultados finales

Tras acabar este proceso de decodificación y procesado de información en MATLAB®, era el momento de realizar las pruebas finales para acabar de crear el simulador robótico para la microcontroladora OpenCM9.04 de ROBOTIS.

En primer lugar se hizo una prueba con el programa **Basic** (puede verse en el **Anexo I**, como los demás que presentaremos). Éste envía las siguientes posiciones: Servo 1 – Posición 180º (512) / Servo 2 – Posición 22.5º (64) / Servo 3 – Posición 3.5º (10).

Veamos antes que nada los datos que se obtuvieron en los distintos *To Workspace* del archivo Simulink® (para el servo de ID 1):

Como puede apreciarse, son los resultados esperados:

- La variable **pos1** almacena la posición deseada para este servo, junto con los valores 50000, indicadores de que llegó una ID o una posición para otro servo.

- La variable **DATA1** guarda los mismos valores que la anterior, pero en vez de hacerlo en formato Dynamixel lo hace en radianes. Aquí ya vemos como 512 implica un giro de π radianes.

- Por último, la variable **giro1** nos da el

pos1 =	DATA1 =	giro1 =
50000	306.7962	0
50000	306.7962	0
50000	306.7962	0
512	3.1416	0.0314
50000	306.7962	0.0628
50000	306.7962	0.0942
50000	306.7962	0.1257
50000	306.7962	0.1571
50000	306.7962	0.1885
512	3.1416	0.2199
50000	306.7962	0.2513
50000	306.7962	0.2827
50000	306.7962	⋮
512	3.1416	2.8274
50000	306.7962	2.8588
50000	306.7962	2.8903
50000	306.7962	2.9217
50000	306.7962	2.9531
50000	306.7962	2.9845
512	3.1416	3.0159
50000	306.7962	3.0473
		3.0788
		3.1102
		3.1416
		3.1416
		3.1416
(a)	(b)	(c)

Figura 39

valor de posición que el servo va tomando en cada iteración. Se aprecia cómo pasa de 0 a π radianes, obteniendo el resultado deseado.

Por último, veamos algunos *frames* del movimiento del robot virtual para este caso y uno alternativo (**Basic2**):

- Servo 1 – Posición 180º (512) / Servo 2 – Posición 22.5º (64) / Servo 3 – Posición 3.5º (10):

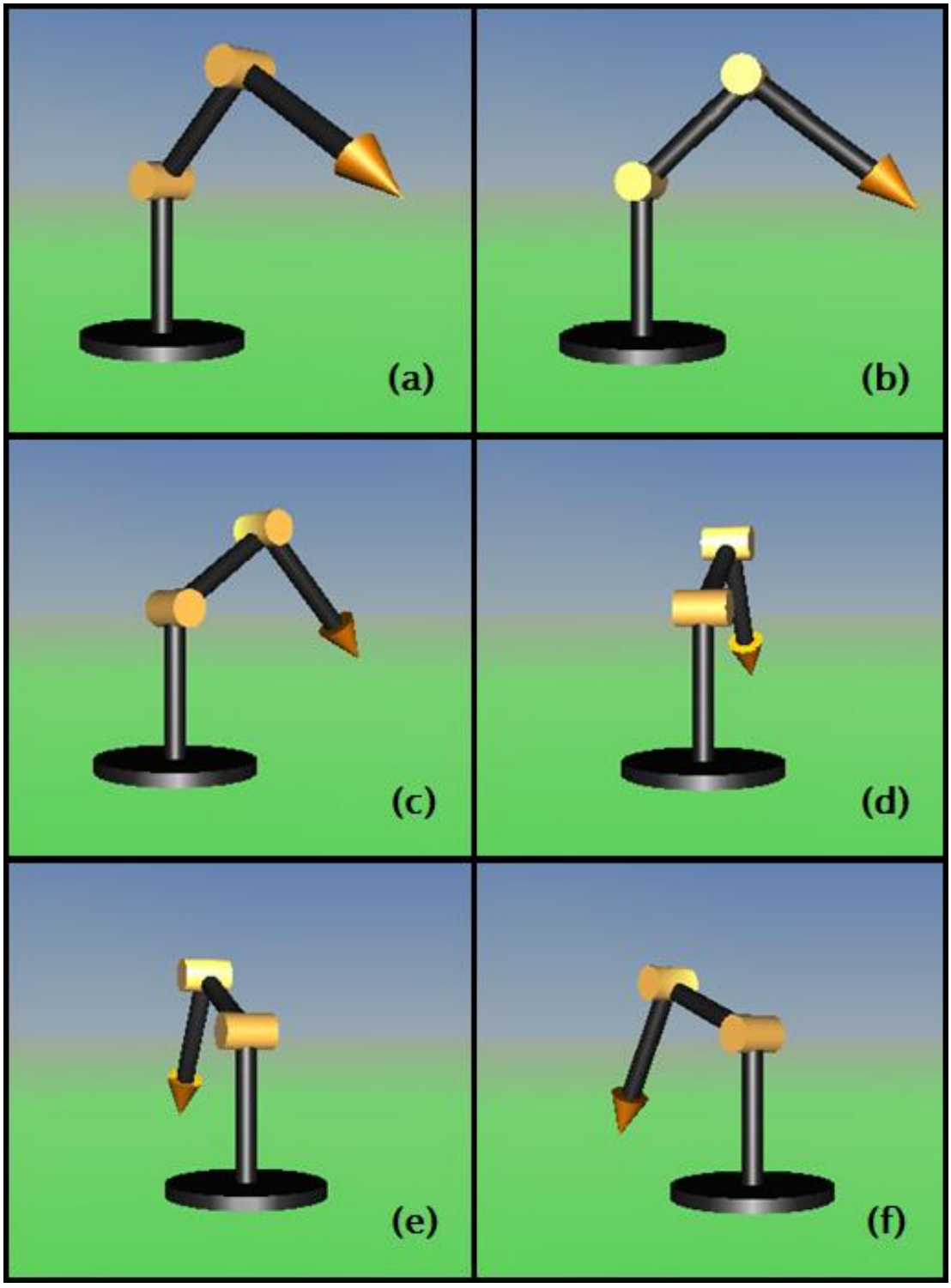


Figura 40

- Servo 1 – Posición 281.25º (800) / Servo 2 – Posición 35.16º (100) / Servo 3 – Posición 17.58º (50):

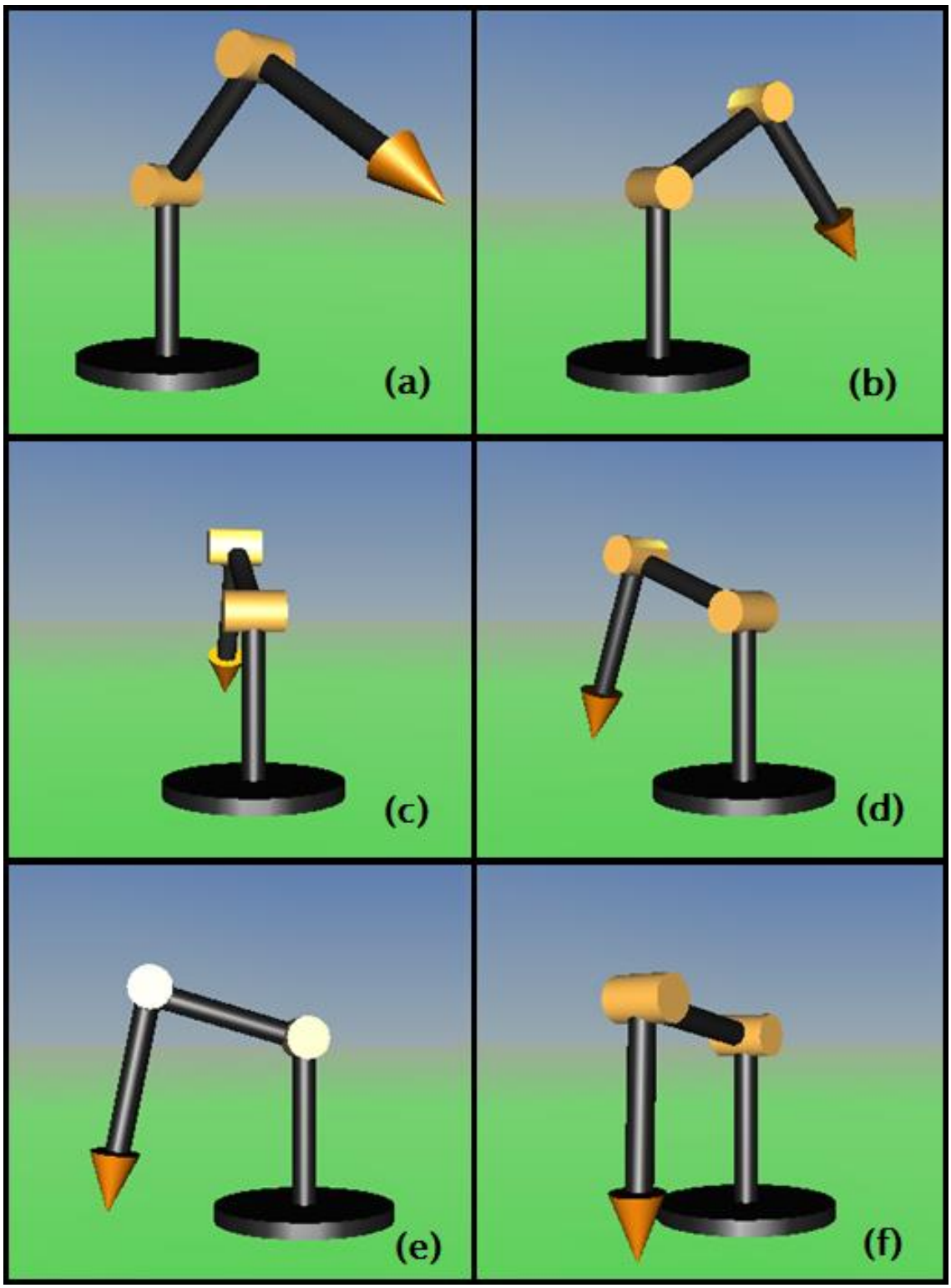


Figura 41

Antes de finalizar, comentar una mejora realizada a posteriori de las pruebas arriba expuestas. Si recordamos la explicación de la primera función existente dentro del subsistema en el archivo **prueba_simulink.slx**, dijimos que en ese momento obviaríamos una línea de código.

En ella, la posición objetivo era restada por π, y el resultado de esa sustracción era la posición que llegaba realmente a la función.

El objetivo de esta modificación es cambiar la posición de reposo actual, de forma que cuando antes esa posición era 0, ahora será π. Esto se llevó a cabo porque me di cuenta de un detalle en el diseño VRML: las posiciones que envío desde la microcontroladora abarcan un rango de 0 a 360º, pero no grados negativos. Con esto quiero decir que, en las articulaciones 2 y 3, para llegar a ciertas posiciones con la configuración inicial tendría que "atravesar" al propio robot, lo que es físicamente imposible (aunque el visualizador 3D lo permita).

Por ello introduje ese offset de π radianes, de forma que puedo abarcar un espacio de trabajo mucho mayor. Veamos el resultado tras esta modificación del siguiente ejemplo (tengamos en cuenta que ahora 512 se corresponde con la posición de un giro de 0º):

- Servo 1 – Posición -90º (256) / Servo 2 – Posición 39.72º (625) / Servo 3 – Posición -60.47º (340):

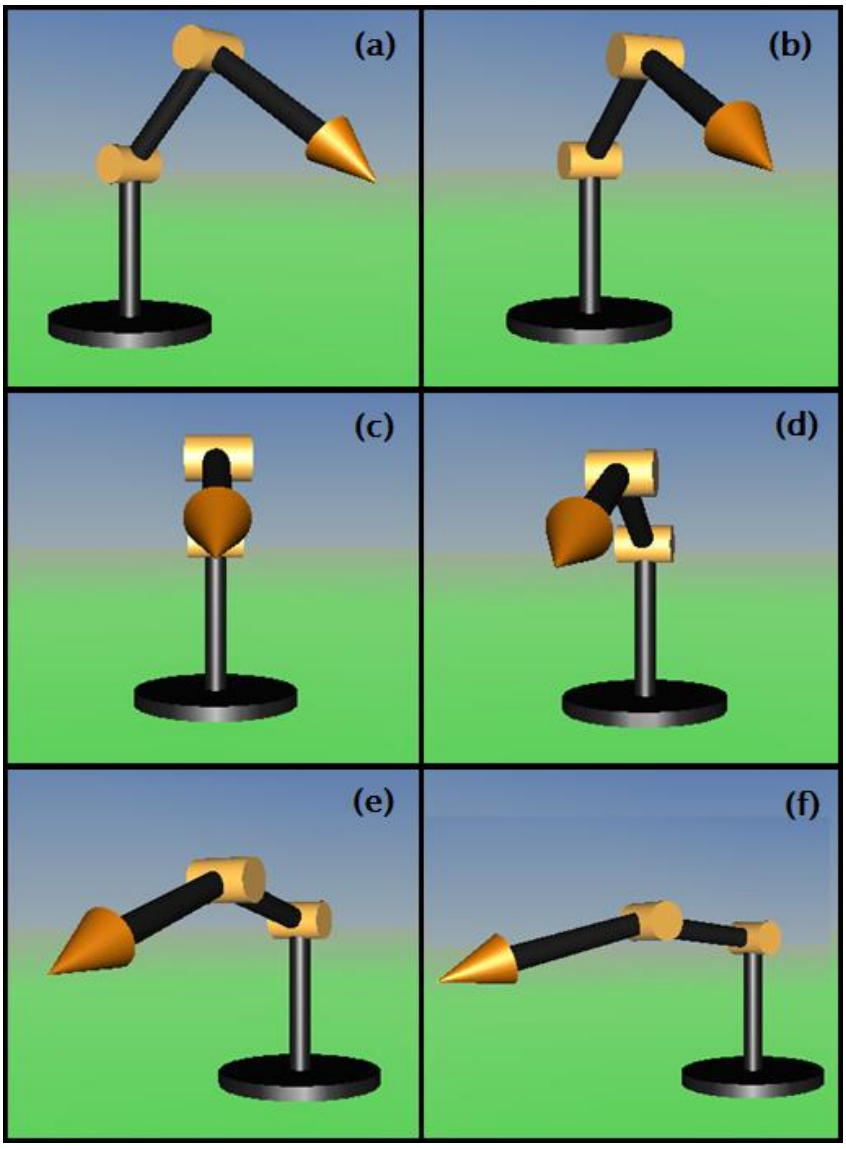


Figura 42

2.3. Conclusiones

Una vez finalizada la descripción sobre el desarrollo del proyecto, es el momento de analizar su progresión, alcance, sus puntos flojos y/o ampliables, sus partes más sólidas e importantes, su utilidad final respecto a los objetivos marcados, etc. Así pues, para concluir, examinemos cada uno de los puntos expuestos en este proyecto.

2.3.1. Primeros pasos

Esta sección, que hace las veces de punto de partida del trabajo, tiene una utilidad y una potencia que pueden pasar desapercibidas a priori, debido a su condición de punto introductorio.

El hecho de comenzar a trabajar con un elemento nuevo, tal y como era System Generator for DSP™ (aunque MATLAB® y Xilinx® por separado fueran conocidos para mí), siempre retrasa en cierto grado el progreso de trabajo, ya que durante la fase de inicialización y toma de contacto con este componente del trabajo se cometen múltiples errores.

Tal y como se aprecia analizando esta sección, ha sido orientada a servir de guía para iniciarse en el empleo de las herramientas de Xilinx® desde MATLAB®. Se explica el proceso a seguir para, empezando de cero, establecer los parámetros necesarios para comenzar a trabajar. Paralelamente a esto, se explican errores comunes cometidos por la gran mayoría de iniciados en estas herramientas (incluido yo), tanto el motivo de su aparición como su solución.

Y es justo ahí donde reside la utilidad de la sección, así como el trabajo que la subyace: aunque ésta está basada en un tutorial (Levine, N., s.f.), dicho recurso web no atiende a posibles fallos, sino que avanza explicando los pasos a seguir, presuponiendo que ninguno de ellos dará problemas. Mientras, la primera sección de mi proyecto toma estos pasos presentados en el *webinar* y los completa, añadiendo estos posibles errores y su solución, con el objetivo primordial de ayudar a quien se esté enfrentando por primera vez a System Generator for DSP™ a avanzar a través de estos pasos de inicialización, sin perder el tiempo necesario para descubrir el porqué de cada fallo y como arreglarlo.

En cuanto a las conclusiones técnicas que pueden extraerse, la principal es que recomiendo fervientemente emplear directamente MATLAB® R2015a, o incluso la última versión disponible, por las razones que hemos visto durante el proyecto y en las que ahondaremos más adelante.

Como herramienta de Xilinx® asociada a MATLAB® es cierto que yo empleé ISE Design Suite principalmente. No obstante, trabajando en este proyecto he descubierto la potencia de Vivado Design Suite, ya que éste permite no sólo trabajar con la parte de PL de la FPGA, sino con el conjunto PS-PL, incluso desde System Generator for DSP™ (de nuevo, entraremos más en el tema en secciones posteriores).

2.3.2. Diseño de PCB en EAGLE

De nuevo un elemento del proyecto ciertamente "nuevo", y es que trabajé con EAGLE y PCBs en 2º para una asignatura, pero fue algo excesivamente ligero y sin profundidad. Por ello, tuve que recuperar ciertos tutoriales del software para comenzar a trabajar con él.

Ésta no era una parte trivial, en el sentido de que no podía permitirme estar fabricando PCBs por errores míos: debía hacer sólo una y debía ser perfecta para mi diseño. Por esta razón dediqué bastante tiempo a escudriñar los posibles fallos en el esquemático de la placa de adaptación, hasta que consideré que estaba listo para crearla físicamente.

Esto fue llevado a cabo con la inestimable ayuda de **Agustín Díaz Cárdenas**, encargado del taller del Laboratorio de Electrónica de la escuela. Fue él quien me enseñó cómo fabricar la PCB, y me prestó todo el material necesario, desde soldadores hasta componentes para la placa.

Es cierto que la creación de una placa PCB no me permitió extraer demasiadas conclusiones, aunque

sí querría destacar la versatilidad y amplia gama de soluciones que ofrece poder diseñar y crear tu propia placa, ajustada a las necesidades de tu proyecto. Por ello recomiendo tomar un cierto contacto con software de edición de PCB, sea cuál sea, ya que se convertirá en una potente herramienta para cualquier ingeniero.

En cuanto al conversor lógico, también tratado en esta sección, fue la solución más apropiada para un problema grave, ya que no podía comunicar dos dispositivos trabajando a distintos niveles de tensión de forma segura, a menos que convirtiera uno de ellos al nivel del otro. Se exploraron otras posibilidades como un divisor de tensión simple por ejemplo, pero ninguna cumplía con los requerimientos mínimos de seguridad en la conversión.

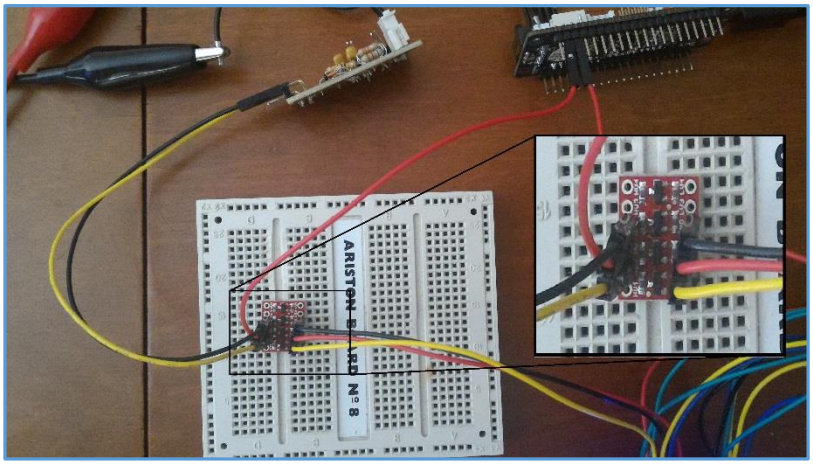


Figura 43

Sin embargo, aunque fue una solución, también dio lugar a un problema a la hora de ampliar el proyecto: el protocolo Dynamixel contempla el uso de un canal bidireccional por donde tanto microcontroladora como servo/s envían y reciben información. No obstante, el conversor lógico empleado no es bidireccional en un canal, sino que tiene un canal para cada dirección. Esto imposibilita enviar los Paquetes de Datos desde MATLAB® hacia la OpenCM9.04, ya que necesitaría otro canal. Para solucionarlo podrían barajarse varias opciones como:

- Probar si el segundo 3 Pin TTL de la microcontroladora, libre en nuestro caso, puede implementarse para recibir paquetes, y así implementar una comunicación de doble canal.
- Buscar una nueva forma de convertir los 5V de la OpenCM9.04 a los 3.3V de ZedBoard, y viceversa, produciendo una comunicación monocanal.

Por último cabría comentar la inclusión de sensores simulados para aumentar la complejidad del proyecto. Como ya indicamos durante la memoria, la placa de adaptación está preparada (a falta de soldar los pines) para recibir la señal de PWM de la ZedBoard (de dutycycle proporcional a la distancia del robot a cierto objeto de referencia), y pasarla por un filtro RC para obtener una tensión analógica proporcional a esta distancia, que sería leída por la microcontroladora a través de sus 5 pines empleados en medición sensorial.

Dejo estas ideas propuestas para una ampliación del proyecto, ya que por falta de tiempo no han podido ser implementadas.

2.3.3. Diseño de pruebas (I)

En esta sección volvemos a encontrar una parte de guía para comenzar a trabajar con System Generator for DSP™, una vez ya hemos acometido los primeros pasos de iniciación y configuración de las herramientas software.

En ella no tratamos con errores y soluciones, ya que esto tendrá su lugar en la siguiente sección. El fuerte de este punto es la descripción de los elementos más importantes de cualquier diseño Simulink® empleando los bloques facilitados por Xilinx® mediante su **Xilinx Blockset**. Tal y como apreciamos al leer esta sección, se entra a analizar cada uno de estos bloques, comentando sus características, funcionalidad, campos a rellenar (lo cual es importante, ya que no siempre sabemos si debemos dejar un campo por defecto o no), requerimientos y especificaciones…; de forma que conseguimos una

mayor comprensión de la programación con bloques de Xilinx®-Simulink®.

No obstante, en esta sección solo se refleja cierta parte del trabajo realizado en este campo: la investigación sobre System Generator for DSP™ y sus posibilidades empleando **Xilinx Blockset** fue exhaustiva con tal de comprender cómo debía implementar mi diseño, ya que es una herramienta con demasiados detalles que requieren atención como para dejar cabos sueltos en nuestro trabajo con ella.

Tras mi contacto con ella he concluido que, de saber emplearla, conociendo sus límites y posibilidades, se convierte en una herramienta potentísima para producir diseños que aúnen y necesiten de la capacidad de procesado paralelo de una FPGA con la facilidad de manejo y potencia de un software como MATLAB®. Considero ciertamente recomendable el estudio de esta herramienta para proyectos que permitan el tiempo suficiente para comprenderla y emplearla con seguridad.

Así mismo, creo que hay realmente poca información sobre cómo utilizar o iniciarse en System Generator. Sí que se encuentran documentos descriptivos sobre sus funciones, pero no hay tantos tutoriales de uso que sean realmente útiles para aprender a trabajar con **Xilinx Blockset** de forma paulatina, lo que sería muy deseable.

2.3.4. Implementación de pruebas (I)

Este punto del trabajo podríamos dividirlo en dos partes, que a mi parecer están bastante diferenciadas.

Por un lado, se hace una descripción exhaustiva de toda la información recopilada sobre la herramienta **HDL Workflow Advisor**. Como ya comentamos, ésta se hace casi indispensable para aquellos que hayan comenzado recientemente a trabajar con System Generator for DSP™, ya que permite implementar nuestro diseño paso a paso, pasando del archivo Simulink® a un proyecto HDL que podremos introducir directamente en nuestra FPGA.

Considero que es muy importante conocer todos los detalles de una herramienta antes de comenzar a usarla, ya que así puedes reducir en gran forma los fallos de común ocurrencia; y con esa idea incluí este tutorial descriptivo de **HDL Workflow Advisor**: la información que recopilé y analicé para entender cómo debía emplear la herramienta permitirá a quiénes consulten esta memoria tener una introducción a sus características y funcionalidades.

Por otro lado, se hace hincapié en el punto de inflexión que marcó el desarrollo del trabajo, tal y como fue el error descrito en las páginas de esta sección. El hecho de encontrar este error me hizo sacar varias conclusiones:

- Primero, la dificultad de aunar toda la información sobre la herramienta. Esta búsqueda se hizo principalmente en la documentación del propio MATLAB®, ya que es donde más información del tema existe. No obstante, estos datos no están reunidos bajo un mismo directorio, sino que la herramienta de ayuda con la documentación está preparada para la búsqueda de información específica. Esto hace que se escapen detalles que pueden resultar cruciales, como fue el hecho de no poder emplear los bloques de **Xilinx Blockset** para implementar un IP Core.
- Además, fue aquí donde comprendí realmente la importancia de ayudarse de información externa para iniciarse en un nuevo software. Empleé muchísimo tiempo en descubrir cuál era el motivo de que mi diseño no fuera implementable, lo que supuso un retraso demasiado grande para el proyecto.
- Por último, entendí de primera mano que la potencia que ofrece System Generator al aunar MATLAB® con las herramientas HDL de Xilinx® se pierde y torna en una desventaja en caso de no ser usuario habitual de la herramienta, y conocer sus requerimientos, limitaciones y posibilidades

Querría comentar para acabar que una conclusión extra que obtuve trabajando en esto fue que MATLAB® y Xilinx® no han perfeccionado aún la unión de sus herramientas, ya que si esto fuera así, en mi opinión, falta información acerca del empleo de las mismas. Por ello, reitero mi conclusión y consejo más importante: darles uso sólo en caso de saber manejarlas, pero no entrar a emplearlas si no se tienen el tiempo necesario para comprenderlas o la necesidad real de utilizarlas.

2.3.5. VRML: creación desde cero del robot virtual

Aunque no había tenido ningún contacto previo con el lenguaje VRML, resultó ser ciertamente fácil de entender para quienes nos iniciamos en él, al menos en sus bases.

Con esto quiero decir que, tras trabajar con el lenguaje, aprecié como existe una gran cantidad de información acerca del mismo: su uso, particularidades, funcionalidades, etc. Comencé a estudiarlo a través de un libro, que me enseñó lo básico que necesitaba conocer acerca de cómo programar en VRML, o más bien, que significaba cada nodo del mismo. Para realmente comenzar a diseñar me basé en tutoriales de la web, que comentaban ejemplos básicos, pero que me presentaron herramientas como V-Realm Builder, muy útiles para inexpertos en la materia.

Así, uniendo ambas fuentes de información, desarrollé mi diseño del robot virtual en sus formas más básicas. No obstante, me resultó mucho más difícil encontrar información sobre otros aspectos que me hacían falta para completarlo, siendo el principal el giro de elementos del diseño con respecto a otros ejes que no fueran los principales. Esta funcionalidad la implementé deduciéndola a través de la información básica del protocolo, pero no encontré en ningún lado como hacerlo.

En resumen, trabajando con VRML concluí que es ciertamente un lenguaje útil y fácilmente entendible, y que seguramente esté extendido en la industria y otros campos gracias a ser un lenguaje libre; pero también entendí que aún falta para que se extienda de forma completa a otros ámbitos como el educativo.

2.3.6. Diseño de pruebas (II)

A partir de esta sección se redujo el contenido didáctico y de guía de la memoria y del trabajo, y se incidió más en los diseños realizados para alcanzar el objetivo primario: simular el robot virtual.

Tal y como se comenta en la sección se encontraron dos caminos para intentar resolver el problema surgido dos puntos antes:

- La primera, que consistía en seguir empleando System Generator for DSP™ para implementar FPGA In-The-Loop, en vez de un IP Core, no funcionó finalmente. Esto en gran parte se debió a incompatibilidades que me surgieron con las licencias de Xilinx® en la herramienta de Vivado, que no permitían a MATLAB® llamar a estas herramientas en segundo plano de forma efectiva. A pesar de este problema que no me permitió ahondar en el uso de esta funcionalidad, entendí que la implementación de FIL empleando System Generator es una herramienta muy útil a la hora de probar los diseños y trabajar simultáneamente en la FPGA y MATLAB®. Por ello, recomendaría a quienes tuvieran que afrontar un proyecto como este el empleo de estas herramientas, de nuevo, en caso de tener tiempo para estudiarlas.
- La segunda, que rompía con el uso de System Generator y optaba por emplear sus herramientas por separado, resultó ser exitosa. No obstante, tal y como vimos en esta sección, se crearon dos diseños, de los cuáles se acabó empleando el último.

Una vez finalizado el proyecto, con otra perspectiva, acabé llegando a la conclusión de que hubiera sido mejor haber empezado a implementarlo al revés: comenzar creando el diseño con las herramientas por separado, con las que sí tengo experiencia de manejo, y al haber tenido éxito en esta implementación, haber comenzado a trasladarla a System Generator, e incluso haberle añadido mejoras.

No obstante, también entendí que de esa forma no habría llegado al nivel de comprensión de System Generator, y sus opciones, posibilidades y características, que he adquirido de esta forma y que quedan reflejadas en esta memoria.

Por tanto, no considero que haya sido un error haber invertido tiempo en estudiar esta herramienta, ya que ayudará a todo aquél que se enfrente de primeras con System Generator for DSP™.

2.3.7. Implementación de pruebas (II)

Una vez realizadas las pruebas y comprobado el correcto funcionamiento del simulador fui plenamente consciente de las posibilidades que ofrece crear un simulador de robot virtual *custom* para tu microcontroladora. El hecho de poder probar las distintas funciones de ésta fuera de línea permite asegurar su comportamiento para, una vez todo sea funcional, emplear nuestra microcontroladora en el sistema real que estábamos simulando virtualmente.

Cierto es que en mi caso la función implementada ha sido el envío de posiciones de destino para cada uno de los servos. No obstante, en el caso de ROBOTIS, existen muchas otras posibilidades tales como envío de nuevas ID, velocidades objetivo o la petición de información del servo.

Así pues, la implementación exitosa de estas pruebas me ha enseñado el camino para, en caso de ampliar el proyecto, implementar todas las demás funciones importantes que la microcontroladora ofrece y que necesitaríamos probar.

2.3.8. Resumen de conclusiones finales

Así pues, una vez finalizado el proyecto, quiero comentar las conclusiones positivas y negativas generales que puedo extraer de mi trabajo:

2.3.8.1. Aspectos a mejorar

Viendo el proyecto desde la posición de haberlo terminado puedo decir que es ciertamente ampliable. Por ejemplo, podrían implementarse:

- Una mejor caracterización de la dinámica de los servos.
- La cinemática y dinámica del robot en su conjunto, para respetar restricciones físicas.
- Nuevas funcionalidades de la microcontroladora y envío de datos a ésta (antes comentado).
- Una mejor solución para la comunicación[1] [2].

Todas estas ideas, algunas de las cuales ya han sido tratadas en el proyecto, y se ha explicado cómo deberían llevarse a cabo, se dejan como **propuestas de continuación**.

En mi opinión, este es un proyecto que no tiene límite definido, y que podría ser profundizado por otros alumnos para mejorar sus prestaciones, y tal y como dije en la introducción, acabar creando una herramienta útil para el ámbito de la robótica industrial.

COMENTARIOS

---

[1]Tal y como comenté anteriormente, en vez de emplear Arduino como nexo entre ZedBoard y ordenador (lo que introducía ciertos retrasos que explicamos en la sección correspondiente), se intentó previamente emplear la UART de la que dispone la FPGA para realizar el envío de información por puerto serie. Se estudió como hacerlo, pero no dio tiempo a implementarlo. Por si alguien desea ampliar el proyecto por aquí, recomiendo el empleo de PlanAhead y SDK en conjunto con EDK (todos herramientas de Xilinx®, ya que así podremos acceder a las funcionalidades de la parte de PS de la ZedBoard (donde se encuentra la UART), y podremos hacerla accesible desde el puerto EMIO de la misma (el cual también es accesible desde la parte de PL, que es lo que buscamos). Otra opción para conseguir esto es emplear Vivado, aunque en las pruebas que hice me dio algunos problemas, por lo que recomendaría la opción anterior.

[2]Como posible mejora recomiendo unificar los programas VHDL de la UART (que está sin utilizar actualmente) y de decodificación de información (el empleado ahora mismo) para aprovechar las mejores características de cada uno: la modularidad del programa de la UART en conjunción con los detalles de implementación añadidos al segundo programa.

2.3.8.2. Puntos fuertes

No obstante, a pesar de tener posibilidades de ampliación, considero el proyecto ciertamente completo en función de los siguientes puntos:

- Este trabajo sirve de inicio para lo dicho anteriormente, ampliaciones posteriores, pero por eso es precisamente importante: sin una base sólida sobre la que cimentarse, estas "mejoras" no serían posibles, y considero que el proyecto cumple de sobra con este cometido.
- También puede resultar ciertamente útil como guía para iniciarse en algunas de las herramientas empleadas en este trabajo. No todas cuentan con mucha información en la web, y algunas de las que sí, la tienen muy dispersa. En esta memoria queda constancia del trabajo con dichas herramientas, y como se ha reiterado varias veces, se dan consejos de uso y soluciones a errores.
- Se ha demostrado que la comunicación entre OpenCM9.04 / ZedBoard /MATLAB® es posible, y al igual que se ha conseguido con estos elementos, también podrían usarse los pasos explicados en este documento para chequear el funcionamiento de otros dispositivos.

Así, con estas reflexiones, doy por concluido el Trabajo Fin de Grado.

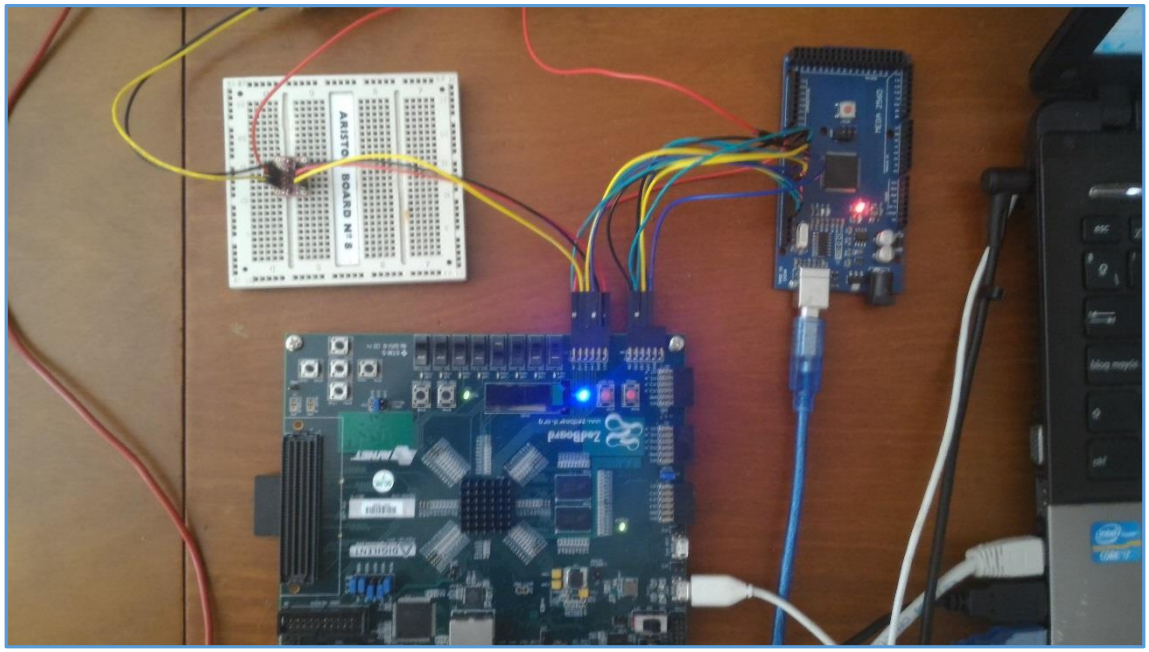


Figura 44

BIBLIOGRAFÍA

- **Libros**

- GÁMEZ, Ilde; MARTÍN, Martín; VRML: curso de iniciación: aprenda a crear mundos virtuales en 3D para Internet; Barcelona: Inforbook's, 2001; 415 p.; 978-8495318701.

- **Herramientas de ayuda de software**

- System Generator Help (archivo online), s.f., file:///C:/Xilinx/14.7/ISE_DS/ISE/sysgen/help_ids/html_help/wwhelp/wwhimpl/js/html/wwhelp.htm#href=Xilinx_Blockset.11.012.html
- MATLAB® Help, s.f., (recurso offline).

- **Recursos web**

- ABB, RobotStudio, 2016, http://new.abb.com/products/robotics/es/robotstudio
- BAINVILLE, Eric [Página Web]; FPGA Simple UART, 2013, http://www.bealto.com/fpga-uart_io.html
- CARRASCO, Javier [Página Web]; CMD no me reconoce los comandos en Windows 7, 16 de mayo de 2012, http://www.javiercarrasco.es/2012/05/16/cmd-no-me-reconoce-los-comandos-en-windows-7/
- Coppelia Robotics, s.f., http://www.coppeliarobotics.com/
- Front&Back [Blog] - CALDERÓN, José Vicente; ¿Qué son los niveles lógicos? Conectando Arduino y Raspberry, 19 de mayo de 2014, http://www.frontandback.org/laboratory/como_conectar_arduino_raspberry_pi
- La Web del Programador, Curso de VRML 2.0, s.f., http://www.lawebdelprogramador.com/cursos/vrml/vrml_1.php
- LEVINE, Noam; Run a Simulink Model on Zynq: ZedBoard Set Up (2 of 4) [Video], s.f., http://es.mathworks.com/videos/run-a-simulink-model-on-zynq-zedboard-set-up-2-of-4-89509.html?requestedDomain=www.mathworks.com
- LEVINE, Noam; Run a Simulink Model on Zynq: ZedBoard Set Up (4 of 4) [Video], s.f., http://es.mathworks.com/videos/run-a-simulink-model-on-zynq-code-generation-and-deployment-4-of-4-89511.html
- MathWorks Documentation: algebraic loops, s.f., http://es.mathworks.com/help/simulink/ug/algebraic-loops.html
- MathWorks Documentation: hdlrestoreparams, s.f., http://es.mathworks.com/help/hdlcoder/ref/hdlrestoreparams.html
- MathWorks Documentation: HDL Verifier Support Package for Xilinx FPGA Boards Release Notes, s.f., http://es.mathworks.com/help/supportpkg/xilinxfpgaboards/release-notes.html
- MathWorks Documentation: rehash, s.f., http://es.mathworks.com/help/matlab/ref/rehash.html
- MATLAB Answers™; MathWorks Support Team (12 de Agosto de 2013 – 30 de julio de 2015), *How do I install Microsoft Windows SDK 7.1?*, https://es.mathworks.com/matlabcentral/answers/101105-how-do-i-install-microsoft-windows-sdk-7-1

- RAMOS, C. [Blog]; Señal de control para motor de DC mediante PWM y VHDL, 27 de julio de 2012, http://www.estadofinito.com/motor-dc-pwm-vhdl/
- ROBOTIS(a), OpenCM9.04 ROBOTIS e-Manual, versión 1.21.00, s.f., http://support.robotis.com/en/product/auxdevice/controller/opencm9.04.htm
- ROBOTIS(b), Overview of Communication ROBOTIS e-Manual, versión 1.27.00, s.f., http://support.robotis.com/en/product/dynamixel/dxl_communication.htm
- ROBOTIS(c), Dynamixel AX-12 User's Manual, 14 de junio de 2006, http://www.generationrobots.com/media/Dynamixel-AX-12-user-manual.pdf
- ROBOTIS(d), Bottom View, OpenCM9.04, 9 de diciembre de 2013, http://support.robotis.com/en/baggage_files/opencm/opencm9.04__rev_1.0(131009)-bottom.pdf
- TechMake Electronics, s.f., http://www.techmake.com/00155.html
- Wikipedia, Prueba de concepto, 26 de febrero de 2014, https://es.wikipedia.org/wiki/Prueba_de_concepto
- Xilinx, System Generator for DSP™ User's Guide, versión 11.4, 2 de diciembre de 2009, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sysgen_user.pdf
- ZedBoard(a), MathWorks Design Package, s.f., http://zedboard.org/product/mathworks-design-package
- ZedBoard(b), Hardware User's Guide, version 2.2, 27 de enero de 2014, http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf
- ZedBoard(c), ZedBoard Product, s.f., http://zedboard.org/product/zedboard

ANEXO I: CÓDIGO

Durante toda la memoria se han ido comentando programas, códigos, etc…; de softwares distintos, implementando diferentes funciones para llegar a nuestro objetivo. Así pues, en este Anexo podemos encontrar el código de esos programas, de forma que la comprensión de las explicaciones sea completa.

I) Código para la microcontroladora OpenCM9.04

- Basic.ino

```
void setup() {
    // Initialize the dynamixel bus:
    Dxl.begin(1);
}

void loop() {
    delay(500);
    Dxl.writeWord(1, 30, 512);
    delay(500);
    Dxl.writeWord(2, 30, 64);
    delay(500);
    Dxl.writeWord(3, 30, 10);
}
```

- Basic2.ino

```
void setup() {
    // Initialize the dynamixel bus:
    Dxl.begin(1);
}

void loop() {
    delay(50);
    Dxl.writeWord(1, 30, 800);
    delay(50);
    Dxl.writeWord(2, 30, 100);
    delay(50);
    Dxl.writeWord(3, 30, 50);
}
```

- **Basic_180.ino**

```
void setup() {
    // Initialize the dynamixel bus:
    Dxl.begin(1);
}

void loop() {
    delay(50);
    Dxl.writeWord(1, 30, 256);
    delay(50);
    Dxl.writeWord(2, 30, 625);
    delay(50);
    Dxl.writeWord(3, 30, 340);
}
```

II) Código de Arduino

- **prueba_arduino.ino**

```
int RXD_0=2;
int RXD_1=3;
int RXD_2=4;
int RXD_3=5;
int RXD_4=6;
int RXD_5=7;
int RXD_6=8;
int RXD_7=9;
int cont_0=10;
int cont_1=11;
int cont_2=12;
int cont_3=13;
int pre_cont,pre_2_cont;
int val=9999;
int val2=6666;
int byte_out,pre_byte_out,pre_2_byte_out,cont0,cont1,cont2,cont3,cont;
int RXD0,RXD1,RXD2,RXD3,RXD4,RXD5,RXD6,RXD7,RXDenable,cont1bit,cont4bit;
int id,pos,byte_0,byte_1,en1,en2,en3,pre_byte_0,pre_id,id_sal,pre_byte_1;

void setup() {
  Serial.begin(250000);
}

void loop() {
  Byte2();
}

void Byte2(){

  cont0 = digitalRead(cont_0);
  cont1 = digitalRead(cont_1);
```

```
  cont2 = digitalRead(cont_2);
  cont3 = digitalRead(cont_3);
  RXD0 = digitalRead(RXD_0);
  RXD1 = digitalRead(RXD_1);
  RXD2 = digitalRead(RXD_2);
  RXD3 = digitalRead(RXD_3);
  RXD4 = digitalRead(RXD_4);
  RXD5 = digitalRead(RXD_5);
  RXD6 = digitalRead(RXD_6);
  RXD7 = digitalRead(RXD_7);

  cont = cont0+(2*cont1)+(4*cont2)+(8*cont3);
  byte_out=(128*RXD7)+(64*RXD6)+(32*RXD5)+(16*RXD4)+(8*RXD3)+(4*RXD2)+(2*RXD1)+RXD0;

 ////////////////////////////////
  if(cont==2 && en1 == 0){

    id = byte_out;
    en1=1;
  }
 if(cont==6 && en2 == 0){
    byte_0 = byte_out;
    en2=1;
  }
  if(cont==7 && en3 == 0){
    byte_1 = byte_out;
    en3=1;
  }
  if((en1 == 1) && (en2==1) && (en3==1)) {
    if(id != pre_id){
    id_sal = pre_id + val;
    pre_id = id;
    Serial.println(id_sal);
    delay(10);

    pos = 256*byte_1+pre_byte_0;

    pre_byte_0 = byte_0;

    Serial.println(pos);
    delay(500);
    }

    en1=0;
    en2=0;
    en3=0;

  }
}
```

III) Código VHDL

- **Proyecto: inout**

- **top.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity top is
    Port (  clk : in STD_LOGIC;
            TXD : in  STD_LOGIC;
            RXD : out  STD_LOGIC;
            dutycycle : in STD_LOGIC_VECTOR(6 downto 0);
            cont_conectado_g : out STD_LOGIC;
            cont_conectado_v : out STD_LOGIC;
            pwm : out STD_LOGIC;
            DIR_PORT : in  STD_LOGIC;
            DATA : inout  STD_LOGIC;
            VDD : in  STD_LOGIC;
            Ground : in  STD_LOGIC);
end top;

architecture Behavioral of top is
signal cont : std_logic_vector(10 downto 0) := "00000000000";
signal clk2 : std_logic;
signal cnt : std_logic_vector(6 downto 0) := "0000000";
begin

cont_conectado_g <= Ground;
cont_conectado_v <= VDD;

DATA <= TXD when DIR_PORT='0' else 'Z';
RXD <= DATA;

clk_500hz : process(clk,cont)
begin
if(rising_edge(clk)) then
   if(cont = "11111001111") then
      clk2 <= '1';
      cont <= "00000000000";
   else
      cont <= cont + "001";
      clk2 <= '0';
   end if;
end if;
end process;

pwm_signal : process(clk2)
begin
if(rising_edge(clk2)) then
   if(cnt = 99) then
```

```
      cnt <= (others => '0');
   else
      cnt <= cnt + 1;
   end if;
end if;
end process;
pwm <= '1' when (cnt < dutycycle) else '0';
end Behavioral;
```

- **prueba.vhd**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

ENTITY prueba IS
END prueba;

ARCHITECTURE behavior OF prueba IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT top
    PORT(
         clk : IN  std_logic;
         TXD : IN  std_logic;
         RXD : OUT  std_logic;
         dutycycle : IN  std_logic_vector(6 downto 0);
         cont_conectado_g : OUT  std_logic;
         cont_conectado_v : OUT  std_logic;
         pwm : OUT  std_logic;
         DIR_PORT : IN  std_logic;
         DATA : INOUT  std_logic;
         VDD : IN  std_logic;
         Ground : IN  std_logic
        );
    END COMPONENT;

    --Inputs
    signal clk : std_logic := '0';
    signal TXD : std_logic := '0';
    signal dutycycle : std_logic_vector(6 downto 0) := "0000000";
    signal DIR_PORT : std_logic := '1';
    signal VDD : std_logic := '1';
    signal Ground : std_logic := '0';

    --BiDirs
    signal DATA : std_logic;

    --Outputs
    signal RXD : std_logic;
    signal cont_conectado_g : std_logic;
    signal cont_conectado_v : std_logic;
    signal pwm : std_logic;
```

```
   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: top PORT MAP (
          clk => clk,
          TXD => TXD,
          RXD => RXD,
          dutycycle => dutycycle,
          cont_conectado_g => cont_conectado_g,
          cont_conectado_v => cont_conectado_v,
          pwm => pwm,
          DIR_PORT => DIR_PORT,
          DATA => DATA,
          VDD => VDD,
          Ground => Ground
        );

   -- Clock process definitions
   clk_process :process
   begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 2000000 ns;
      dutycycle <= "0011110";
      wait for clk_period*200000;
      dutycycle <= "0110010";
      wait for clk_period*200000;
      dutycycle <= "1000110";
      wait for clk_period*200000;
      dutycycle <= "0110010";
      wait for clk_period*200000;
      dutycycle <= "1100100";
      wait for clk_period*200000;
      dutycycle <= "0110010";
      -- insert stimulus here

      wait;
   end process;

END;
```

- **conexionado.ucf**

```
NET "Ground" LOC = "AA9" | IOSTANDARD = "LVCMOS33";
NET "VDD" LOC = "Y10" | IOSTANDARD = "LVCMOS33";
NET "DATA" LOC = "AA11" | IOSTANDARD = "LVCMOS33";
NET "pwm" LOC = "Y11" | IOSTANDARD = "LVCMOS33";
NET "clk" LOC = "Y9" | IOSTANDARD = "LVCMOS33";
NET "dutycycle(6)" IOSTANDARD = "LVCMOS33";
NET "dutycycle(5)" IOSTANDARD = "LVCMOS33";
NET "dutycycle(4)" IOSTANDARD = "LVCMOS33";
NET "dutycycle(3)" IOSTANDARD = "LVCMOS33";
NET "dutycycle(2)" IOSTANDARD = "LVCMOS33";
NET "dutycycle(1)" IOSTANDARD = "LVCMOS33";
NET "dutycycle(0)" IOSTANDARD = "LVCMOS33";
NET "cont_conectado_g" IOSTANDARD = "LVCMOS33";
NET "cont_conectado_v" IOSTANDARD = "LVCMOS33";
NET "DIR_PORT" LOC = "AB11" | IOSTANDARD = "LVCMOS33";
#NET "TXD" IOSTANDARD = "LVCMOS33";
NET "RXD" LOC = "W12" | IOSTANDARD = "LVCMOS33";
```

- **Proyecto: uart**

- **divisor.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity divisor is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           sample : out  STD_LOGIC);
end divisor;

architecture Behavioral of divisor is
constant cont : integer := 4;
constant divisor : integer := 10;
signal sample_cont : std_logic_vector(cont-1 downto 0);
begin

   sample_process: process(clk,reset) is
   begin

      if reset = '1' then
         sample_cont <= (others => '0');
         sample <= '0';
      elsif rising_edge(clk) then
         if sample_cont = divisor-1 then
            sample <= '1';
            sample_cont <= (others => '0');
         else
```

```
            sample <= '0';
            sample_cont <= sample_cont+1;
         end if;
      end if;

   end process;

end Behavioral;
```

- **divisor_pr.vhd**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

ENTITY divisor_pr IS
END divisor_pr;

ARCHITECTURE behavior OF divisor_pr IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT divisor
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         sample : OUT  std_logic
        );
    END COMPONENT;

   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '1';

   --Outputs
   signal sample : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: divisor PORT MAP (
          clk => clk,
          reset => reset,
          sample => sample
        );

   -- Clock process definitions
   clk_process :process
   begin
      clk <= '0';
      wait for clk_period/2;
```

```
      clk <= '1';
      wait for clk_period/2;
   end process;

   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 r
      wait for 100 ns;
      reset <= '0';
      wait for clk_period*10;

      -- insert stimulus here

      wait;
   end process;

END;
```

- **rx_module.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity rx_module is
    Port ( clk : in  STD_LOGIC;
           clk_sample : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           rx_ext : in  STD_LOGIC;
           rx_out : out  STD_LOGIC_VECTOR (7 downto 0);
           rx_enable : out  STD_LOGIC);
end rx_module;

architecture Behavioral of rx_module is

type fsm_state_t is (idle,active);
type rx_state_t is
record
   fsm_state: fsm_state_t;
   counter: std_logic_vector (3 downto 0);
   bits: std_logic_vector (7 downto 0);
   nbits: std_logic_vector (3 downto 0);
   enable: std_logic;
end record;

signal rx_state,rx_state_next: rx_state_t;

begin

   clk_upd_process: process(clk,reset) is
   begin
      if reset = '1' then
```

```
      rx_state.fsm_state <= idle;
      rx_state.counter <= (others=> '0');
      rx_state.bits <= (others=> '0');
      rx_state.nbits <= (others=> '0');
      rx_state.enable <= '0';
   elsif rising_edge(clk) then
      rx_state <= rx_state_next;
   end if;
end process;

rx_process: process(rx_state,clk_sample,rx_ext) is
begin
   case rx_state.fsm_state is

   when idle =>
      rx_state_next.counter <= (others=> '0');
      rx_state_next.bits <= (others=> '0');
      rx_state_next.nbits <= (others=> '0');
      rx_state_next.enable <= '0';
      if rx_ext = '0' then
         rx_state_next.fsm_state <= active;
      else
         rx_state_next.fsm_state <= idle;
      end if;

   when active =>
      if clk_sample = '1' then
         if rx_state.counter = 5 then
            if rx_state.nbits = 9 then
               rx_state_next.fsm_state <= active;
               rx_state_next.enable <= '0';
               rx_state_next.nbits <= rx_state.nbits+1;
               rx_state_next.bits <= rx_state.bits;
            elsif rx_state.nbits = 10 then
               rx_state_next.fsm_state <= idle;
               rx_state_next.enable <= rx_ext;
               rx_state_next.nbits <= rx_state.nbits;
               rx_state_next.bits <= rx_state.bits;
            else
               rx_state_next.bits <= rx_ext & rx_state.bits(7 downto 1);
               rx_state_next.nbits <= rx_state.nbits+1;
               rx_state_next.fsm_state <= active;
               rx_state_next.enable <= '0';
            end if;
            rx_state_next.counter <= rx_state.counter+1;
         elsif rx_state.counter = 9 then
            rx_state_next.counter <= (others => '0');
            rx_state_next.fsm_state <= active;
            rx_state_next.enable <= '0';
            rx_state_next.nbits <= rx_state.nbits;
            rx_state_next.bits <= rx_state.bits;
         else
            rx_state_next.counter <= rx_state.counter+1;
            rx_state_next.fsm_state <= active;
            rx_state_next.enable <= '0';
            rx_state_next.nbits <= rx_state.nbits;
```

```
                rx_state_next.bits <= rx_state.bits;
            else
                rx_state_next.counter <= rx_state.counter+1;
                rx_state_next.fsm_state <= active;
                rx_state_next.enable <= '0';
                rx_state_next.nbits <= rx_state.nbits;
                rx_state_next.bits <= rx_state.bits;
            end if;
         else
            rx_state_next <= rx_state;
         end if;
      when others =>
         rx_state_next <= rx_state;
      end case;
   end process;

   rx_output: process(rx_state) is
   begin
      rx_enable <= rx_state.enable;
      rx_out <= rx_state.bits;
   end process;

end Behavioral;
```

- **rx_module_pr.vhd**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY rx_module_pr IS
END rx_module_pr;

ARCHITECTURE behavior OF rx_module_pr IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT rx_module
    PORT(
         clk : IN  std_logic;
         clk_sample : IN  std_logic;
         reset : IN  std_logic;
         rx_ext : IN  std_logic;
         rx_out : OUT  std_logic_vector(7 downto 0);
         rx_enable : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal clk_sample : std_logic := '0';
   signal reset : std_logic := '1';
   signal rx_ext : std_logic := '1';
```

```
   --Outputs
   signal rx_out : std_logic_vector(7 downto 0);
   signal rx_enable : std_logic;

   -- Clock period definitions
   constant clk_period : time := 10 ns;
   constant clk_sample_period : time := 100 ns;

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: rx_module PORT MAP (
          clk => clk,
          clk_sample => clk_sample,
          reset => reset,
          rx_ext => rx_ext,
          rx_out => rx_out,
          rx_enable => rx_enable
        );

   -- Clock process definitions
   clk_process :process
   begin
      clk <= '0';
      wait for clk_period/2;
      clk <= '1';
      wait for clk_period/2;
   end process;
   clk_sample_process :process
   begin
      clk_sample <= '0';
      wait for 9*clk_sample_period/10;
      clk_sample <= '1';
      wait for clk_sample_period/10;
   end process;


   -- Stimulus process
   stim_proc: process
   begin
      -- hold reset state for 100 ns.
      wait for 100 ns;
      reset <= '0';
      wait for 100 ns;

      -- insert stimulus here
      rx_ext <= '0';
      wait for 1 us;
      rx_ext <= '0';
      wait for 1 us;
      rx_ext <= '1';
      wait for 1 us;
      rx_ext <= '0';
      wait for 1 us;
      rx_ext <= '1';
      wait for 1 us;
```

```
      rx_ext <= '0';
      wait for 1 us;
      rx_ext <= '0';
      wait for 1 us;
      rx_ext <= '1';
      wait for 1 us;
      rx_ext <= '1';
      wait for 1 us;
      rx_ext <= '1';
      wait for 1 us;
      rx_ext <= '1';
      wait;
   end process;

END;
```

- **fsm_byte.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity fsm_byte is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           error_out : out STD_LOGIC;
           byte_enable : in  STD_LOGIC;
           byte_in : in  STD_LOGIC_VECTOR (7 downto 0);
           byte_out : out  STD_LOGIC_VECTOR (7 downto 0);
           byte_en_out : out STD_LOGIC_VECTOR (1 downto 0));
end fsm_byte;

architecture Behavioral of fsm_byte is
type estado_t is (FF_1,FF_2,ID,LONG,INSTR,PARAM,CHECKS,ERR);
signal estado,estado_next: estado_t;
signal checksum,checksum_next: std_logic_vector (15 downto 0);
signal byte_t,cont,cont_next: std_logic_vector (1 downto 0);
signal byte_s: std_logic_vector (7 downto 0);
signal longitud: std_logic_vector (2 downto 0);
signal error: std_logic;
begin

   clk_process: process(clk,reset) is
   begin
      if reset = '1' then
         estado <= FF_1;
         cont <= "00";
         checksum <= (others => '0');
      elsif rising_edge(clk) then
         estado <= estado_next;
         cont <= cont_next;
         checksum <= checksum_next;
      end if;
   end process;
```

```
fsm_process: process(byte_enable,byte_in,estado,checksum,cont,error)
begin
   byte_s <= byte_in;
   case estado is
      when FF_1 =>
   --    estado_next <= estado;
         checksum_next <= (others => '0');
         cont_next <= "00";
         byte_t <= "00";
         error <= '0';
         if byte_enable = '1' then
            if byte_in = "11111111" then
               estado_next <= FF_2;
            else
               estado_next <= ERR;
            end if;
         else
            estado_next <= FF_1;
         end if;

      when FF_2 =>
         error <= '0';
         byte_t <= "00";
         estado_next <= estado;
         checksum_next <= checksum;
         cont_next <= cont;
         if byte_enable = '1' then
            if byte_in = "11111111" then
               estado_next <= ID;
            else
               estado_next <= ERR;
            end if;
         else
            estado_next <= FF_2;
         end if;

      when ID =>
         error <= '0';
         estado_next <= estado;
         cont_next <= cont;
         if byte_enable = '1' then
            byte_t <= "01";
            estado_next <= LONG;
            checksum_next <= checksum + ("00000000" & byte_in);
         else
            byte_t <= "00";
            estado_next <= ID;
            checksum_next <= checksum;
         end if;

      when LONG =>
         error <= '0';
         byte_t <= "00";
         estado_next <= estado;
         cont_next <= cont;
         if byte enable = '1' then
```

```
            --longitud <= byte_in(2 downto 0) - "010";
            estado_next <= INSTR;
            checksum_next <= checksum + ("00000000" & byte_in);
         else
            --longitud <= (others => '0');
            estado_next <= LONG;
            checksum_next <= checksum;
         end if;

      when INSTR =>
         error <= '0';
         byte_t <= "00";
         estado_next <= estado;
         cont_next <= cont;
         if byte_enable = '1' then
            if byte_in = x"03" then
               estado_next <= PARAM;
               checksum_next <= checksum + ("00000000" & byte_in);
            else
               estado_next <= ERR;
               checksum_next <= (others => '0');
            end if;
         else
            estado_next <= INSTR;
            checksum_next <= checksum;
         end if;

      when PARAM =>
         error <= '0';
         estado_next <= estado;
         if byte_enable = '1' then
            case cont is
               when "00" =>
                  if byte_in = x"1E" then
                     estado_next <= PARAM;
                     cont_next <= cont + 1;
                     checksum_next <= checksum + ("00000000" & byte_in);
                  else
                     estado_next <= ERR;
                     cont_next <= cont;
                     checksum_next <= (others => '0');
                  end if;
                  byte_t <= "00";
               when "01" =>
                  cont_next <= cont+1;
                  byte_t <= "10";
                  estado_next <= PARAM;
                  checksum_next <= checksum + ("00000000" & byte_in);

               when "10" =>
                  cont_next <= cont+1;
                  byte_t <= "11";
                  estado_next <= CHECKS;
                  checksum_next <= checksum + ("00000000" & byte_in);
               when others =>
                  estado next <= estado;
```

```
                    cont_next <= cont;
                    byte_t <= "00";
                    estado_next <= PARAM;
                    checksum_next <= checksum;
                end case;
            else
                estado_next <= PARAM;
                cont_next <= cont;
                byte_t <= "00";
                checksum_next <= checksum;
            end if;

        when CHECKS =>
            error <= '0';
            byte_t <= "00";
            estado_next <= estado;
            cont_next <= cont;
            checksum_next <= checksum;
            if byte_enable = '1' then
                if byte_in = NOT checksum(7 downto 0) then
                    estado_next <= FF_1;
                else
                    estado_next <= ERR;
                end if;
            else
                estado_next <= CHECKS;
            end if;
        when ERR =>
            byte_t <= "00";
            cont_next <= cont;
            estado_next <= FF_1;
            error <= '1';
            checksum_next <= (others => '0');
        when others =>
            estado_next <= estado;
            cont_next <= cont;
            checksum_next <= checksum;
            byte_t <= "00";
    end case;
end process;

byte_output_process: process(byte_t,byte_s,error)
begin
    byte_out <= byte_s;
    byte_en_out <= byte_t;
    error_out <= error;
end process;

end Behavioral;
```

- **fsm_byte_pt.vhd**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

ENTITY fsm_byte_pt IS
END fsm_byte_pt;

ARCHITECTURE behavior OF fsm_byte_pt IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT fsm_byte
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         error_out : OUT  std_logic;
         byte_enable : IN  std_logic;
         byte_in : IN  std_logic_vector(7 downto 0);
         byte_out : OUT  std_logic_vector(7 downto 0);
         byte_en_out : OUT  std_logic_vector(1 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '1';
   signal byte_enable : std_logic := '0';
   signal byte_in : std_logic_vector(7 downto 0) := (others => '0');

   --Outputs
   signal error_out : std_logic;
   signal byte_out : std_logic_vector(7 downto 0);
   signal byte_en_out : std_logic_vector(1 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: fsm_byte PORT MAP (
          clk => clk,
          reset => reset,
          error_out => error_out,
          byte_enable => byte_enable,
          byte_in => byte_in,
          byte_out => byte_out,
          byte_en_out => byte_en_out
        );
```

```
-- Clock process definitions
clk_process :process
begin
   clk <= '0';
   wait for clk_period/2;
   clk <= '1';
   wait for clk_period/2;
end process;


-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 100 ns.
   wait for 100 ns;
   reset <= '0';
   byte_in <= "11111111";
   byte_enable <= '1';
   wait for clk_period;
   byte_in <= "11111111";
   byte_enable <= '0';
   wait for (clk_period*100+clk_period/2);
   byte_in <= "11111111";
   byte_enable <= '1';
   wait for clk_period;
   byte_in <= "11111111";
   byte_enable <= '0';
   wait for 1000 ns;
   byte_in <= "00000001";
   byte_enable <= '1';
   wait for clk_period;
   byte_in <= "00000001";
   byte_enable <= '0';
   wait for 1000 ns;
   byte_in <= "00000101";
   byte_enable <= '1';
   wait for clk_period;
   byte_in <= "00000101";
   byte_enable <= '0';
   wait for 1000 ns;
   byte_in <= "00000011";
   byte_enable <= '1';
   wait for clk_period;
   byte_in <= "00000011";
   byte_enable <= '0';
   wait for 1000 ns;
   byte_in <= x"1E";
   byte_enable <= '1';
   wait for clk_period;
   byte_in <= x"1E";
   byte_enable <= '0';
   wait for 1000 ns;
   byte_in <= "00000000";
   byte_enable <= '1';
   wait for clk_period;
   byte_in <= "00000000";
```

```
      byte_enable <= '0';
      wait for 1000 ns;
      byte_in <= "00000010";
      byte_enable <= '1';
      wait for clk_period;
      byte_in <= "00000010";
      byte_enable <= '0';
      wait for 1000 ns;
      byte_in <= x"D6";
      byte_enable <= '1';
      wait for clk_period;
      byte_in <= x"D6";
      byte_enable <= '0';
      -- insert stimulus here

      wait;
   end process;

END;
```

- **top.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity top is
    Port ( clk : in  STD_LOGIC;
           reset_b : in  STD_LOGIC;
           DATA : in  STD_LOGIC;
           --DATA : inout  STD_LOGIC;
           --DIR_PORT : in STD_LOGIC;
           RXD : out  STD_LOGIC_VECTOR (9 downto 0);
           RXD_t : out STD_LOGIC_VECTOR (1 downto 0));
           --TXD : in  STD_LOGIC;
           --VDD : in  STD_LOGIC;
           --Ground : in  STD_LOGIC);
end top;

architecture Behavioral of top is
component divisor is
PORT(
      clk : in  STD_LOGIC;
      reset : in  STD_LOGIC;
      sample : out  STD_LOGIC);
end component;

component rx_module is
PORT (
```

```
      clk : in  STD_LOGIC;
      clk_sample : in  STD_LOGIC;
      reset : in  STD_LOGIC;
      rx_ext : in  STD_LOGIC;
      rx_out : out  STD_LOGIC_VECTOR (7 downto 0);
      rx_enable : out  STD_LOGIC);
end component;

component fsm_byte is
PORT (
      clk : in  STD_LOGIC;
      reset : in  STD_LOGIC;
      error_out : out STD_LOGIC;
      byte_enable : in  STD_LOGIC;
      byte_in : in  STD_LOGIC_VECTOR (7 downto 0);
      byte_out : out  STD_LOGIC_VECTOR (7 downto 0);
      byte_en_out : out STD_LOGIC_VECTOR (1 downto 0));
end component;

signal clk_sample,RXD_e,reset,error : std_logic;
signal RXD_s,RXD_out,RXD_out_ant : std_logic_vector (7 downto 0);
signal RXD_keep,RXD_keep_next : std_logic_vector (7 downto 0);
signal RXD_ten,RXD_ten_next : std_logic_vector (9 downto 0);
signal RXD_type,contador,contador_next : std_logic_vector (1 downto 0);
signal RXD_t_keep,RXD_t_keep_next : std_logic_vector (1 downto 0);
begin

   --DATA <= '1' when DIR_PORT='0' else 'Z';
   divisor_inst: divisor
   PORT MAP(
   clk=>clk,
   reset=>reset,
   sample=>clk_sample
   );

   rx_inst: rx_module
   PORT MAP(
   clk=>clk,
   clk_sample=>clk_sample,
   reset=>reset,
   rx_ext=>DATA,
   rx_out=>RXD_s,
   rx_enable=>RXD_e
   );
   clk_process: process(clk,contador_next,reset) is
   begin
      if reset = '1' then
         contador <= "11";
         RXD_keep <= (others => '1');
         RXD_t_keep <= (others => '1');
         RXD_ten <= (others => '1');
      elsif rising_edge(clk) then
         contador <= contador_next;
         RXD_keep <= RXD_keep_next;
         RXD_t_keep <= RXD_t_keep_next;
```

```
      RXD_ten <= RXD_ten_next;
   end if;
end process;

fsm_byte_inst: fsm_byte
PORT MAP(
clk=>clk,
reset=>reset,
error_out=>error,
byte_enable=>RXD_e,
byte_in=>RXD_s,
byte_out=>RXD_out,
byte_en_out=>RXD_type
);

output: process(contador,RXD_type,RXD_out,RXD_ten) is
begin
   if RXD_type = "01" then
      RXD <= "00" & RXD_out;
      RXD_t <= RXD_type;
      RXD_keep_next <= RXD_out;
      RXD_t_keep_next <= RXD_type;
      RXD_ten_next <= (others => '0');
      contador_next <= "00";
   elsif RXD_type = "10" then
      RXD <= (others => '1');
      RXD_t <= RXD_type;
      RXD_keep_next <= RXD_out;
      RXD_t_keep_next <= RXD_type;
      RXD_ten_next <= RXD_ten;
      contador_next <= "00";
   elsif RXD_type = "11" then
      RXD <= RXD_out(1 downto 0) & RXD_keep;
      RXD_t <= RXD_type;
      RXD_keep_next <= RXD_keep;
      RXD_t_keep_next <= RXD_type;
      RXD_ten_next <= RXD_out(1 downto 0) & RXD_keep;
      contador_next <= "00";
   elsif contador < 3 then
      if RXD_t_keep = "11" then
         RXD <= RXD_ten;
      elsif RXD_t_keep = "10" then
         RXD <= (others => '1');
      else
         RXD <= "00" & RXD_keep;
      end if;
      RXD_t <= RXD_t_keep;
      contador_next <= contador + 1;
      RXD_keep_next <= RXD_keep;
      RXD_t_keep_next <= RXD_t_keep;
      RXD_ten_next <= RXD_ten;
   else
      RXD <= (others => '1');
      RXD_t <= RXD_type;
      contador_next <= contador;
      RXD_keep_next <= RXD_keep;
```

```
            RXD_t_keep_next <= RXD_t_keep;
            RXD_ten_next <= RXD_ten;
         end if;
      end process;

      reset_control: process (reset_b) is
      begin
         if reset_b = '1' then
            reset <= '1';
         else
            reset <= '0';
         end if;
      end process;

end Behavioral;
```

- **top_pr.vhd**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY top_pr IS
END top_pr;

ARCHITECTURE behavior OF top_pr IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT top
    PORT(
         clk : IN  std_logic;
         reset_b : IN  std_logic;
         DATA : IN  std_logic;
         RXD : OUT  std_logic_vector(9 downto 0);
         RXD_t : OUT  std_logic_vector(1 downto 0)
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal reset_b : std_logic := '1';
   signal DATA : std_logic := '1';

   --Outputs
   signal RXD : std_logic_vector(9 downto 0);
   signal RXD_t : std_logic_vector(1 downto 0);

   -- Clock period definitions
   constant clk_period : time := 10 ns;

BEGIN

   -- Instantiate the Unit Under Test (UUT)
   uut: top PORT MAP (
```

```
        clk => clk,
        reset_b => reset_b,
        DATA => DATA,
        RXD => RXD,
        RXD_t => RXD_t
      );

-- Clock process definitions
clk_process :process
begin
   clk <= '0';
   wait for clk_period/2;
   clk <= '1';
   wait for clk_period/2;
end process;


-- Stimulus process
stim_proc: process
begin
   -- hold reset state for 100 ns.
   wait for 100 ns;
   reset_b <= '0';
   wait for 100 ns;

   -- insert stimulus here
   DATA <= '0';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';

   wait for 1 us;
   DATA <= '0';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
   wait for 1 us;
   DATA <= '1';
```

```
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
```

```
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
```

```
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '0';
wait for 1 us;
```

```
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';
wait for 1 us;
DATA <= '1';

wait for 1 us;
DATA <= '0';
```

```
        wait for 1 us;
        DATA <= '1';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '1';
        wait for 1 us;
        DATA <= '1';

        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '1';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '1';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '0';
        wait for 1 us;
        DATA <= '1';
        wait for 1 us;
        DATA <= '1';


        wait;

    end process;

END;
```

- **pinout.ucf**

```
NET clk LOC = "Y9" | IOSTANDARD=LVCMOS33;
#NET Ground LOC = "AA9" | IOSTANDARD=LVCMOS33; #JA4
#NET VDD LOC = "Y10" | IOSTANDARD=LVCMOS33; #JA3
NET DATA LOC = "AA11" | IOSTANDARD=LVCMOS33; #| PULLUP; #JA2
NET DIR_PORT LOC = "Y11" | IOSTANDARD=LVCMOS33; #JA1
NET RXD_t(0) LOC = "W12" | IOSTANDARD=LVCMOS33; #JB1
NET RXD(1) LOC = "W11" | IOSTANDARD=LVCMOS33; #JB2
NET RXD(2) LOC = "V10" | IOSTANDARD=LVCMOS33; #JB3
NET RXD(3) LOC = "W8" | IOSTANDARD=LVCMOS33; #JB4
NET RXD(4) LOC = "V12" | IOSTANDARD=LVCMOS33; #JB7
NET RXD(5) LOC = "W10" | IOSTANDARD=LVCMOS33; #JB8
NET RXD(6) LOC = "V9" | IOSTANDARD=LVCMOS33; #JB9
NET RXD(7) LOC = "V8" | IOSTANDARD=LVCMOS33; #JB10
NET RXD(8) LOC = "AB11" | IOSTANDARD=LVCMOS33; #JA7
NET RXD(9) LOC = "AB10" | IOSTANDARD=LVCMOS33; #JA8
NET RXD(0) LOC = "AB9" | IOSTANDARD=LVCMOS33; #JA9
NET RXD_t(1) LOC = "AA8" | IOSTANDARD=LVCMOS33; #JA10
NET reset_b LOC = "T18" | IOSTANDARD=LVCMOS33; #BTNU
```

- **Proyecto: hduart**

- **top_layer.vhd**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;

entity top_layer is
 Port (    clk : in  STD_LOGIC;
           reset_b : in  STD_LOGIC;
           DATA : inout  STD_LOGIC;
           DIR_PORT : in STD_LOGIC;
           salida : out STD_LOGIC;
           rxd_out : out STD_LOGIC_VECTOR (11 downto 0));
end top_layer;

architecture Behavioral of top_layer is
type estado_t is (IDLE,BIT_START,BIT1,BIT2,BIT3,BIT4,BIT5,BIT6,BIT7,BIT8,BIT_STOP1,BIT_STOP2);
```

* Se empequeñece línea por su gran tamaño.

```
signal estado,estado_next: estado_t;
signal reset,data_in,enable,enable_next : std_logic;
signal write_en,write_en_next,read_en,read_en_next : std_logic;
signal e2,e2_next,salida_2_int,salida_2_int_next : std_logic;
signal full2,empty2,full,empty : std_logic;
signal rxd,rxd_next : std_logic_vector (7 downto 0);
signal cont4,cont4_next : std_logic_vector (16 downto 0);
signal checksum,checksum_next : std_logic_vector (15 downto 0);
signal cont,cont_next : std_logic_vector (6 downto 0);
```

```
signal rxd_int,rxd_int_next : std_logic_vector (11 downto 0);
signal cont3,cont3_next : std_logic_vector (6 downto 0);
signal cont2,cont2_next : std_logic_vector (3 downto 0);
signal enable_cont,enable_cont_next : std_logic_vector (3 downto 0);

component p
   PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(11 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(11 DOWNTO 0);
    full : OUT STD_LOGIC;
    empty : OUT STD_LOGIC
   );
end component;

begin

rxd_int <= cont2 & rxd;

fifo_rxd : p
   PORT MAP(
   clk=>clk,
   rst=>reset,
   din=>rxd_int,
   wr_en=>write_en,
   rd_en=>read_en,
   dout=>rxd_out,
   full=>full,
   empty=>empty
   );

contador_ret: process(cont4) is
begin
   if cont4 < 128000 then
      cont4_next <= cont4 + 1;
      read_en_next <= '0';
   else
      cont4_next <= (others => '0');
      read_en_next <= '1';
   end if;
end process;

DATA <= '1' when DIR_PORT='0' else 'Z';
data_in <= DATA;

reset_control: process (reset_b) is
   begin
      if reset_b = '1' then
         reset <= '1';
      else
         reset <= '0';
      end if;
   end process;
```

```
clk_process: process(clk,reset) is
   begin
      if reset = '1' then
         estado <= IDLE;
         enable <= '0';
         cont <= (others => '0');
         cont2 <= (others => '0');
         e2 <= '0';
         rxd <= (others => '0');
         cont3 <= (others => '0');
         cont4 <= (others => '0');
         read_en <= '0';
         checksum <= (others => '0');
         write_en <= '0';

      elsif rising_edge(clk) then
         estado <= estado_next;
         enable <= enable_next;
         rxd <= rxd_next;
         cont <= cont_next;
         cont2 <= cont2_next;
         e2 <= e2_next;
         cont3 <= cont3_next;
         cont4 <= cont4_next;
         read_en <= read_en_next;
         checksum <= checksum_next;
         write_en <= write_en_next;
      end if;
   end process;

fsm: process(estado,cont,rxd,data_in,enable) is
begin
   case estado is
      when IDLE =>
         enable_next <= '0';
         cont_next <= (others => '0');
         e2_next <= '0';
         rxd_next <= (others => '0');
         if falling_edge(data_in) then
            estado_next <= BIT_START;
         else
            estado_next <= IDLE;
         end if;

      when BIT_START =>
         rxd_next <= rxd;
         enable_next <= enable;
         if cont = 99 then
            estado_next <= BIT1;
            cont_next <= (others => '0');
            e2_next <= '0';
         else
            estado_next <= BIT_START;
            cont_next <= cont + 1;
            e2_next <= '0';
```

```
    end if;

when BIT1 =>
    if cont = 99 then
        estado_next <= BIT2;
        rxd_next <= rxd;
        enable_next <= '0';
        cont_next <= (others => '0');
        e2_next <= '0';
    elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
        estado_next <= BIT1;
        enable_next <= '1';
        rxd_next <= data_in & rxd(7 downto 1);
        cont_next <= cont + 1;
        e2_next <= '0';
    else
        estado_next <= BIT1;
        rxd_next <= rxd;
        enable_next <= enable;
        cont_next <= cont + 1;
        e2_next <= '0';
    end if;

when BIT2 =>
    if cont = 99 then
        estado_next <= BIT3;
        rxd_next <= rxd;
        enable_next <= '0';
        cont_next <= (others => '0');
        e2_next <= '0';
    elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
        estado_next <= BIT2;
        enable_next <= '1';
        rxd_next <= data_in & rxd(7 downto 1);
        cont_next <= cont + 1;
        e2_next <= '0';
    else
        estado_next <= BIT2;
        rxd_next <= rxd;
        enable_next <= enable;
        cont_next <= cont + 1;
        e2_next <= '0';
    end if;

when BIT3 =>
    if cont = 99 then
        estado_next <= BIT4;
        rxd_next <= rxd;
        enable_next <= '0';
        cont_next <= (others => '0');
        e2_next <= '0';
    elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
        estado_next <= BIT3;
        enable_next <= '1';
        rxd_next <= data_in & rxd(7 downto 1);
        cont_next <= cont + 1;
```

```
            e2_next <= '0';
        else
            estado_next <= BIT3;
            rxd_next <= rxd;
            enable_next <= enable;
            cont_next <= cont + 1;
            e2_next <= '0';
        end if;

    when BIT4 =>
        if cont = 99 then
            estado_next <= BIT5;
            rxd_next <= rxd;
            enable_next <= '0';
            cont_next <= (others => '0');
            e2_next <= '0';
        elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
            estado_next <= BIT4;
            enable_next <= '1';
            rxd_next <= data_in & rxd(7 downto 1);
            cont_next <= cont + 1;
            e2_next <= '0';
        else
            estado_next <= BIT4;
            rxd_next <= rxd;
            enable_next <= enable;
            cont_next <= cont + 1;
            e2_next <= '0';

        end if;

    when BIT5 =>
        if cont = 99 then
            estado_next <= BIT6;
            rxd_next <= rxd;
            enable_next <= '0';
            cont_next <= (others => '0');
            e2_next <= '0';
        elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
            estado_next <= BIT5;
            enable_next <= '1';
            rxd_next <= data_in & rxd(7 downto 1);
            cont_next <= cont + 1;
            e2_next <= '0';
        else
            estado_next <= BIT5;
            rxd_next <= rxd;
            enable_next <= enable;
            cont_next <= cont + 1;
            e2_next <= '0';
        end if;

    when BIT6 =>
        if cont = 99 then
            estado_next <= BIT7;
            rxd_next <= rxd;
            enable_next <= '0';
```

```
            cont_next <= (others => '0');
            e2_next <= '0';
        elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
            estado_next <= BIT6;
            enable_next <= '1';
            rxd_next <= data_in & rxd(7 downto 1);
            cont_next <= cont + 1;
            e2_next <= '0';
        else
            estado_next <= BIT6;
            rxd_next <= rxd;
            enable_next <= enable;
            cont_next <= cont + 1;
            e2_next <= '0';
        end if;

        when BIT7 =>
        if cont = 99 then
            estado_next <= BIT8;
            rxd_next <= rxd;
            enable_next <= '0';
            cont_next <= (others => '0');
            e2_next <= '0';
        elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
            estado_next <= BIT7;
            enable_next <= '1';
            rxd_next <= data_in & rxd(7 downto 1);
            cont_next <= cont + 1;
            e2_next <= '0';
        else
            estado_next <= BIT7;
            rxd_next <= rxd;
            enable_next <= enable;
            cont_next <= cont + 1;
            e2_next <= '0';
        end if;

    when BIT8 =>
        if cont = 99 then
            estado_next <= BIT_STOP1;
            rxd_next <= rxd;
            enable_next <= '0';
            cont_next <= (others => '0');
            e2_next <= '0';
        elsif (cont > 45) AND (cont < 55) AND (enable = '0') then
            estado_next <= BIT8;
            enable_next <= '1';
            rxd_next <= data_in & rxd(7 downto 1);
            cont_next <= cont + 1;
            e2_next <= '0';
        else
            estado_next <= BIT8;
            rxd_next <= rxd;
            enable_next <= enable;
            cont_next <= cont + 1;
            e2_next <= '0';
```

```
            end if;

         when BIT_STOP1 =>
            if cont = 99 then
               estado_next <= BIT_STOP2;
               rxd_next <= rxd;
               cont_next <= (others => '0');
               enable_next <= '0';
               e2_next <= '0';
            else
               estado_next <= BIT_STOP1;
               rxd_next <= rxd;
               cont_next <= cont + 1;
               enable_next <= enable;
               e2_next <= '1';
            end if;

         when BIT_STOP2 =>
            if cont = 99 then
               estado_next <= IDLE;
               rxd_next <= rxd;
               cont_next <= (others => '0');
               enable_next <= '0';
               e2_next <= '0';
            else
               estado_next <= BIT_STOP2;
               rxd_next <= rxd;
               cont_next <= cont + 1;

               enable_next <= enable;
               e2_next <= '0';
            end if;

      end case;
   end process;

   output_process: process(rxd,cont2,cont3,e2,checksum) is
   begin
      if (rxd = "11111111") AND (cont2 = 0) AND e2 = '1' then
         if cont3 < 98 then
            cont2_next <= "0000";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '0';
            checksum_next <= (others => '0');
         else
            cont2_next <= "0001";
            salida <= '0';
            cont3_next <= (others => '0');
            write_en_next <= '0';
            checksum_next <= (others => '0');
         end if;
      elsif (rxd = "11111111") AND (cont2 = 1) AND e2 = '1' then
         if cont3 < 98 then
            cont2_next <= "0001";
            salida <= '0';
            cont3 next <= cont3 + 1;
```

```
      write_en_next <= '0';
      checksum_next <= checksum;
   else
      cont2_next <= "0010";
      salida <= '0';
      cont3_next <= (others => '0');
      write_en_next <= '0';
      checksum_next <= checksum;
   end if;
elsif (cont2 = 2) AND e2 = '1' then
   if cont3 < 98 then
      if cont3 < 1 then
         cont2_next <= "0010";
         salida <= '0';
         cont3_next <= cont3 + 1;
         write_en_next <= '1';
         checksum_next <= checksum + ("00000000" & rxd);
      else
         cont2_next <= "0010";
         salida <= '0';
         cont3_next <= cont3 + 1;
         write_en_next <= '0';
         checksum_next <= checksum;
      end if;
   else
      cont2_next <= "0011";
      salida <= '0';
      cont3_next <= (others => '0');
      write_en_next <= '0';
      checksum_next <= checksum;
   end if;
elsif (rxd = "00000101") AND (cont2 = 3) AND e2 = '1' then
   if cont3 < 98 then
      if cont3 < 1 then
         cont2_next <= "0011";
         salida <= '0';
         cont3_next <= cont3 + 1;
         write_en_next <= '0';
         checksum_next <= checksum + ("00000000" & rxd);
      else
         cont2_next <= "0011";
         salida <= '0';
         cont3_next <= cont3 + 1;
         write_en_next <= '0';
         checksum_next <= checksum;
      end if;
   else
      cont2_next <= "0100";
      salida <= '0';
      cont3_next <= (others => '0');
      write_en_next <= '0';
      checksum_next <= checksum;
   end if;
elsif (rxd = "00000011") AND (cont2 = 4) AND e2 = '1' then
   if cont3 < 98 then
      if cont3 < 1 then
```

```
            cont2_next <= "0100";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '0';
            checksum_next <= checksum + ("00000000" & rxd);
        else
            cont2_next <= "0100";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '0';
            checksum_next <= checksum;
        end if;
    else
        cont2_next <= "0101";
        salida <= '0';
        cont3_next <= (others => '0');
        write_en_next <= '0';
        checksum_next <= checksum;
    end if;
elsif (rxd = x"1E") AND (cont2 = 5) AND e2 = '1' then
    if cont3 < 98 then
        if cont3 < 1 then
            cont2_next <= "0101";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '0';
            checksum_next <= checksum + ("00000000" & rxd);
        else
            cont2_next <= "0101";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '0';
            checksum_next <= checksum;
        end if;
    else
        cont2_next <= "0110";
        salida <= '0';
        cont3_next <= (others => '0');
        write_en_next <= '0';
        checksum_next <= checksum;
    end if;
elsif (cont2 = 6) AND e2 = '1' then
    if cont3 < 98 then
        if cont3 < 1 then
            cont2_next <= "0110";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '1';
            checksum_next <= checksum + ("00000000" & rxd);
        else
            cont2_next <= "0110";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '0';
            checksum_next <= checksum;
        end if;
```

```
      else
         cont2_next <= "0111";
         salida <= '0';
         cont3_next <= (others => '0');
         write_en_next <= '0';
         checksum_next <= checksum;
      end if;
   elsif (cont2 = 7) AND e2 = '1' then
      if cont3 < 98 then
         if cont3 < 1 then
            cont2_next <= "0111";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '1';
            checksum_next <= checksum + ("00000000" & rxd);
         else
            cont2_next <= "0111";
            salida <= '0';
            cont3_next <= cont3 + 1;
            write_en_next <= '0';
            checksum_next <= checksum;
         end if;
      else
         cont2_next <= "1000";
         salida <= '0';
         cont3_next <= (others => '0');
         write_en_next <= '0';
         checksum_next <= checksum;

      end if;
   elsif (rxd = NOT checksum(7 downto 0)) AND (cont2 = 8) AND e2 = '1' then
      if cont3 < 98 then
         cont2_next <= "1000";
         salida <= '0';
         cont3_next <= cont3 + 1;
         write_en_next <= '0';
         checksum_next <= checksum;
      else
         cont2_next <= "0000";
         salida <= '0';
         cont3_next <= (others => '0');
         write_en_next <= '0';
         checksum_next <= checksum;
      end if;
   else
      cont2_next <= cont2;
      salida <= '1';
      cont3_next <= cont3;
      write_en_next <= '0';
      checksum_next <= checksum;
   end if;
end process;

end Behavioral;
```

- **p.xco** : es la FIFO instanciada. El código puede obtenerse desde el programa ISE Design Suite. No se incluye por no se código diseñado por mí y porque no es necesario para comprender el funcionamiento del proyecto.

- **ucefe.ucf**

```
NET clk LOC = "Y9" | IOSTANDARD=LVCMOS33;
NET DATA LOC = "AA11" | IOSTANDARD=LVCMOS33; #| PULLUP; #JA2
NET DIR_PORT LOC = "Y11" | IOSTANDARD=LVCMOS33; #JA1
NET reset_b LOC = "T18" | IOSTANDARD=LVCMOS33; #BTNU
NET salida LOC = "AA8" | IOSTANDARD=LVCMOS33; #JA10
##########################################################
NET rxd_out(8) LOC = "Y10" | IOSTANDARD=LVCMOS33; #JA3
NET rxd_out(9) LOC = "AA9" | IOSTANDARD=LVCMOS33; #JA4
NET rxd_out(10) LOC = "AB11" | IOSTANDARD=LVCMOS33; #JA7
NET rxd_out(11) LOC = "AB10" | IOSTANDARD=LVCMOS33; #JA8
##########################################################
NET rxd_out(0) LOC = "W12" | IOSTANDARD=LVCMOS33; #JB1
NET rxd_out(1) LOC = "W11" | IOSTANDARD=LVCMOS33; #JB2
NET rxd_out(2) LOC = "V10" | IOSTANDARD=LVCMOS33; #JB3
NET rxd_out(3) LOC = "W8" | IOSTANDARD=LVCMOS33; #JB4
NET rxd_out(4) LOC = "V12" | IOSTANDARD=LVCMOS33; #JB7
NET rxd_out(5) LOC = "W10" | IOSTANDARD=LVCMOS33; #JB8
NET rxd_out(6) LOC = "V9" | IOSTANDARD=LVCMOS33; #JB9
NET rxd_out(7) LOC = "V8" | IOSTANDARD=LVCMOS33; #JB10
##########################################################
```

IV) Código de MATLAB®

- **Programas de las pruebas realizadas para lectura desde Arduino**

- **com_matlab_arduino.m**

```
function dato_recibido = com_matlab_arduino(t)

persistent puerto_serial id

if t<1e-8
    delete(instrfind({'Port'},{'COM8'}));
    puerto_serial=serial('COM8');
    set(puerto_serial,'Baudrate',250000);
    id = 0;

    fopen(puerto_serial); %Abro el puerto
end

DATA = fscanf(puerto_serial,'%d');
datos = DATA;
if (DATA-9999) > 0
   id = DATA-9999;
   pos1 = 50000;
   pos2 = 50000;
   pos3 = 50000;
   pos_pre = 50000;
else
   pos_pre = DATA;
end

if id == 1
    pos1 = pos_pre;
    pos2 = 50000;
    pos3 = 50000;
else if id == 2
        pos2 = pos_pre;
        pos1 = 50000;
        pos3 = 50000;
    else if id == 3
            pos3 = pos_pre;
            pos2 = 50000;
            pos1 = 50000;
        else
           pos2 = 50000;
           pos1 = 50000;
           pos3 = 50000;
        end
```

```
    end
end

if t > 5
   stop = 1;
   delete(instrfindall);
else
    stop = 0;
end


dato_recibido = [pos1 pos2 pos3 stop datos];
```

- **posicion.m**

```
function out = posicion(in)

persistent next_pos

pos_in = in(1);
t = in(2);

pos = pos_in - pi;

if t<1e-8
   next_pos = 0;
end

if pos < 3*pi
    if next_pos ~= pos
        next_pos = pos;%-next_pos;
        pos_out = next_pos;
    else
        pos_out = 50000;
    end
else
    pos_out = 50000;
end

out = pos_out;

end
```

- **posicion1.m**

```
function out = posicion1(in)

persistent next_pos1

pos_in = in(1);
t = in(2);

pos = pos_in - pi;

if t<1e-8
   next_pos1 = 0;
end

if pos < 3*pi
    if next_pos1 ~= pos
        next_pos1 = pos;%-next_pos1;
        pos_out = next_pos1;
    else
        pos_out = 50000;
    end
else
    pos_out = 50000;
end

out = pos_out;

end
```

- **posicion2.m**

```
function out = posicion2(in)

persistent next_pos2

pos_in = in(1);
t = in(2);

pos = pos_in - pi;

if t<1e-8
   next_pos2 = 0;
end

if pos < 3*pi
    if next_pos2 ~= pos
```

```
        next_pos2 = pos;%-next_pos2;
        pos_out = next_pos2;
    else
        pos_out = 50000;
    end
else
    pos_out = 50000;
end

out = pos_out;

end
```

- **slew_rate.m**

```
function out = slew_rate(in)

persistent top cont div current

new_giro = in(1);
t = in(2);

nptos = 100;

if t<1e-8
   cont = 1;
   top = 0;
   div = 0;
   current = 0;
end

if new_giro ~= 50000
    top = new_giro;
    div = top/nptos;
    cont = 1;
    current = 0;
end

if cont < nptos + 1
    current = current + div;
    out_in = current;
    cont = cont + 1;
else
    out_in = current;
end
out = out_in;
end
```

- **slew_rate1.m**

```
function out = slew_rate1(in)

persistent top1 cont1 div1 current1

new_giro = in(1);
t = in(2);

nptos = 100;

if t<1e-8
   cont1 = 1;
   top1 = 0;
   div1 = 0;
   current1 = 0;
end

if new_giro ~= 50000
    top1 = new_giro;
    div1 = top1/nptos;
    cont1 = 1;
    current1 = 0;
end

if cont1 < nptos + 1
    current1 = current1 + div1;
    out_in = current1;
    cont1 = cont1 + 1;
else
    out_in = current1;
end
out = out_in;
end
```

- **slew_rate2.m**

```
function out = slew_rate2(in)

persistent top2 cont2 div2 current2

new_giro = in(1);
t = in(2);

nptos = 100;

if t<1e-8
```

```
    cont2 = 1;
    top2 = 0;
    div2 = 0;
    current2 = 0;
end

if new_giro ~= 50000
     top2 = new_giro;
     div2 = top2/nptos;
     cont2 = 1;
     current2 = 0;
end

if cont2 < nptos + 1
     current2 = current2 + div2;
     out_in = current2;
     cont2 = cont2 + 1;
else
     out_in = current2;
end

out = out_in;
end
```

V) Código VRML

- **Archivo que contiene al robot virtual**

- **prueba_robot.wrl**

```
#VRML V2.0 utf8

#Created with V-Realm Builder v2.0
#Integrated Data Systems Inc.
#www.ids-net.com


SpotLight {
    location    0 5 0
}

DEF Robot Transform {
    children [
        DEF Base Transform {
            translation 0 0.125 0
            children Shape {
                appearance  Appearance {
                    material    Material {
                        ambientIntensity    0.2
                        diffuseColor    0.14 0.14 0.14
                        emissiveColor   0.02 0.02 0.02
                        specularColor   0.71 0.71 0.71
                    }

                }

                geometry    Cylinder {
                    height  0.25
                    radius  1.5
                }
            }
        },
        DEF Barra1 Transform {
            translation 0 1.5 0
            center  0.1 0.1 0.1
            children Shape {
                appearance  Appearance {
                    material    Material {
                        ambientIntensity    0.2
                        diffuseColor    0.14 0.14 0.14
                        emissiveColor   0.02 0.02 0.02
```

```
                specularColor   0.71 0.71 0.71
            }

        }

        geometry    Cylinder {
            height  2.5
            radius  0.2
        }

    }
},

DEF Parte1 Transform {
    children [
        DEF Joint1 Transform {
            translation 0 3 0
            rotation    0.707107 0 0.707107  1.5708
            children Shape {
                appearance  Appearance {
                    material    Material {
                        ambientIntensity    0.2
                        diffuseColor    1 0.685305 0.183673
                        emissiveColor   0.17 0.17 0.17
                        specularColor   0.65 0.65 0.65
                    }

                }

                geometry    Cylinder {
                    height  1
                    radius  0.35
                }

            }
        },
        DEF Barra2 Transform {
            translation 0.8 4.1 0.8
            rotation    -0.707107 0 0.707107  2.35619
            children Shape {
                appearance  Appearance {
                    material    Material {
                        ambientIntensity    0.2
                        diffuseColor    0.14 0.14 0.14
                        emissiveColor   0.02 0.02 0.02
                        specularColor   0.71 0.71 0.71
                    }
```

***Quito sangría para que quepa el código dentro de los límites.**

```
            }

            geometry    Cylinder {
                height  2.5
                radius  0.2
            }

        }
    },
    DEF Parte2 Transform {
        children [
            DEF Joint2 Transform {
                translation 1.29238 4.89522 1.27728
                rotation    0.707107 -2.98023e-008 0.707107  1.5708
                center  -0.0216104 5.16197e-006 -0.0504873
                children Shape {
                    appearance  Appearance {
                        material    Material {
                            ambientIntensity    0.2
                            diffuseColor    1 0.685305 0.183673
                            emissiveColor   0.17 0.17 0.17
                            specularColor   0.65 0.65 0.65
                        }
                    }
                    geometry    Cylinder {
                        height  1
                        radius  0.35
                    }
                }
            },
            DEF Barra2 Transform {
                translation 0.8 4.1 0.8
                rotation    -0.707107 0 0.707107  2.35619
                children Shape {
                    appearance  Appearance {
                        material    Material {
                            ambientIntensity    0.2
                            diffuseColor    0.14 0.14 0.14
                            emissiveColor   0.02 0.02 0.02
                            specularColor   0.71 0.71 0.71
                        }

                    }

                    geometry    Cylinder {
                        height  2.5
                        radius  0.2
```

```
        }

    }
},
DEF Parte2 Transform {
    children [
        DEF Joint2 Transform {
            translation 1.29238 4.89522 1.27728
            rotation    0.707107 -2.98023e-008 0.707107  1.5708
            center  -0.0216104 5.16197e-006 -0.0504873
            children Shape {
                appearance  Appearance {
                    material    Material {
                        ambientIntensity    0.2
                        diffuseColor    1 0.685305 0.183673
                        emissiveColor   0.17 0.17 0.17
                        specularColor   0.65 0.65 0.65
                    }
                }
                geometry    Cylinder {
                    height  1
                    radius  0.35
                }
            }
        },
        DEF Barra3 Transform {
            translation 2.1 4.1 2.1
            rotation  0.945148 0.316344 -0.0813703  2.42791
            scale   1 1 0.999999
            center  -0.0217838 -0.0212669 0
            children Shape {
                appearance  Appearance {
                    material    Material {
                        ambientIntensity    0.2
                        diffuseColor    0.14 0.14 0.14
                        emissiveColor   0.02 0.02 0.02
                        specularColor   0.71 0.71 0.71
                    }
                }
                geometry    Cylinder {
                    height  2.5
                    radius  0.2

                }
            }
        },
```

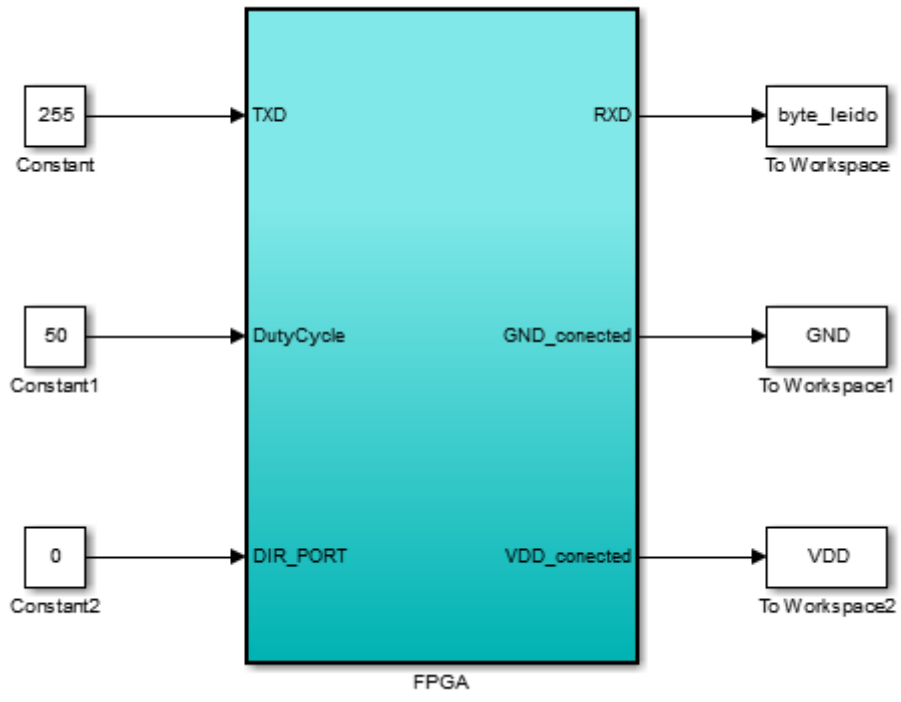
```
                    DEF Elemento_final Transform {
                        translation 3.1229 3.0437 3.1395
                        rotation    0.7554 0.0186 -0.655  2.25375
                        scale   0.999997 0.999997 0.999997
                        children Shape {
                            appearance  Appearance {
                                material    Material {
                                    ambientIntensity    0.2
                                    diffuseColor 1 0.550849 0.0373489
                                    specularColor   0.75 0.75 0.75
                                }

                            }

                            geometry    Cone {
                                bottomRadius    0.4
                                height  1
                            }

                        }
                    }
                ]
                center 1.29238 4.89522 1.27728
                rotation 1 0 -1 -1}
        ]
        center 0 3 0
        rotation 1 0 -1 1}
]
rotation 0 1 0 0}

Background {
    groundAngle [ 0.9, 1.5, 1.57 ]
    groundColor [ 0 0.8 0,
        0.174249 0.82 0.187362,
        0.467223 0.82 0.445801,
        0.621997 0.67 0.600279 ]
    skyAngle    [ 0.1, 1.2, 1.57 ]
    skyColor    [ 0.76238 0.8 0.1427,
        0.277798 0.219779 0.7,
        0.222549 0.390234 0.7,
        0.60094 0.662637 0.69 ]
}
Viewpoint {
    fieldOfView 0.69
    position    0 2.8538 0
    description "My_View"
}
```

ANEXO II: SIMULINK

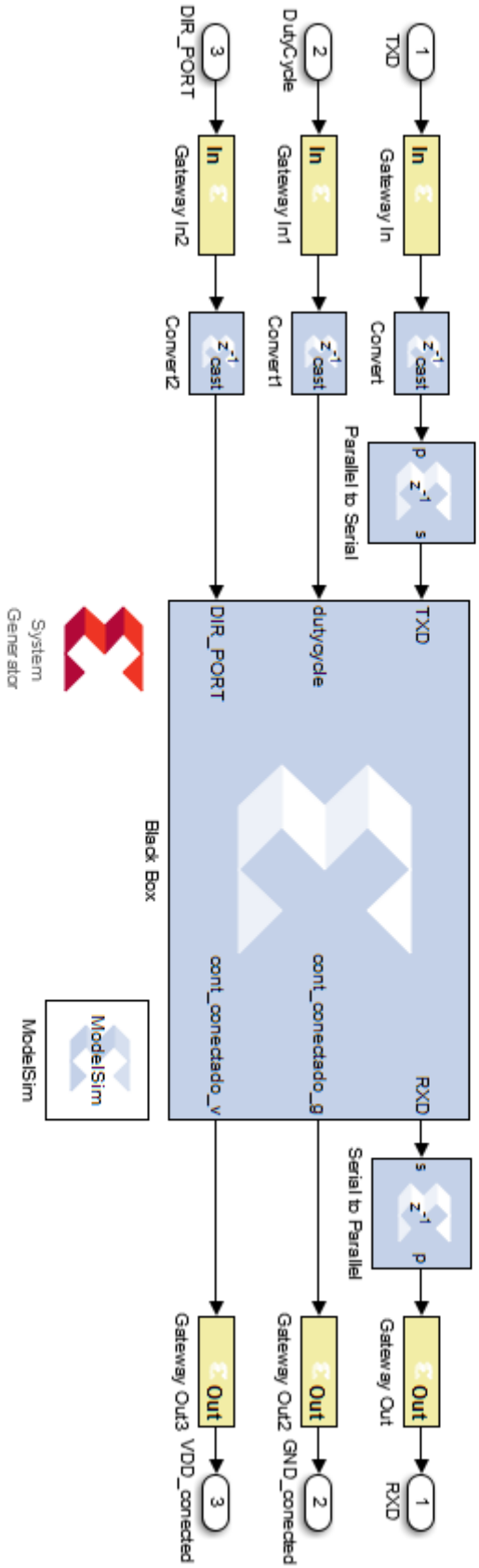
En este Anexo se incluyen imágenes de los distintos archivos de Simulink® empleados y diseñados durante la realización del proyecto. Se mostrarán tanto las capas superiores del archivo como los posibles subsistemas que contenga.

A) Diseño empleando el Xilinx Blockset

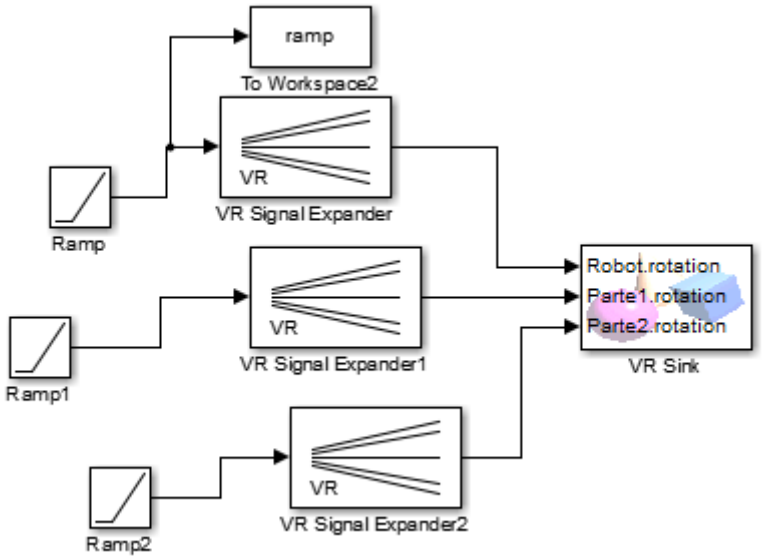
- Capa superior



- **Subsistema “FPGA”**

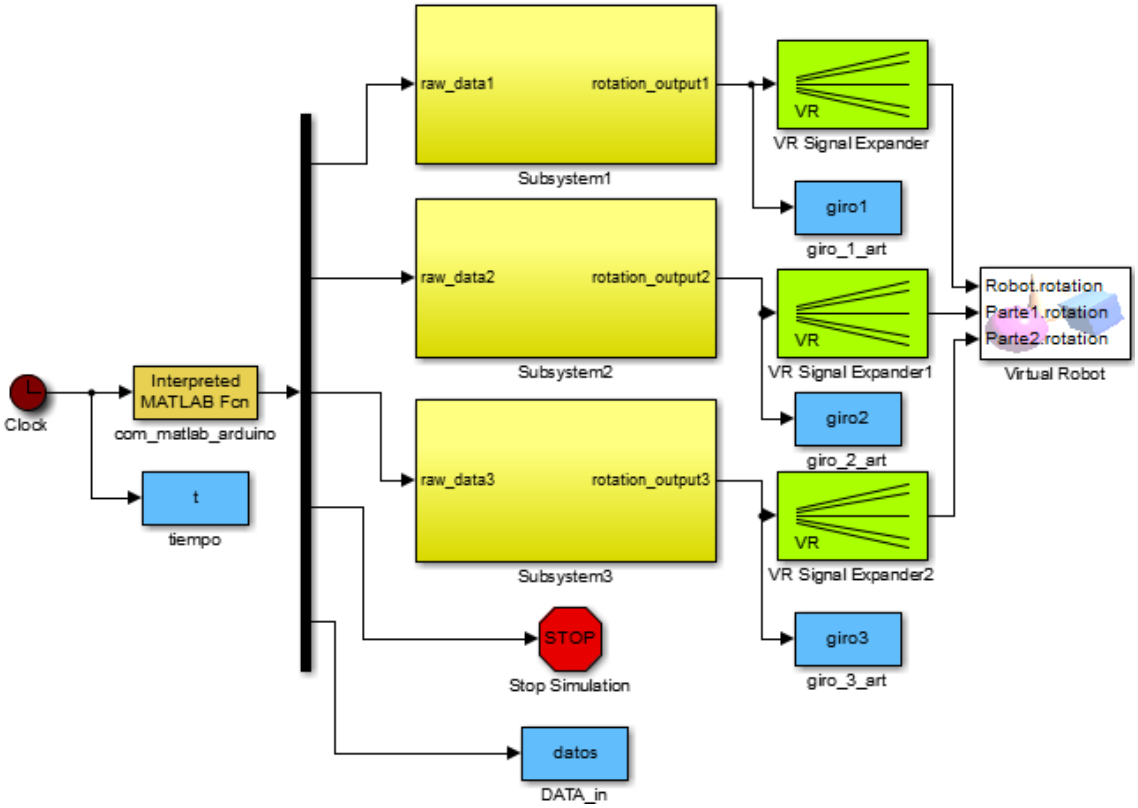


B) Banco de pruebas del robot virtual

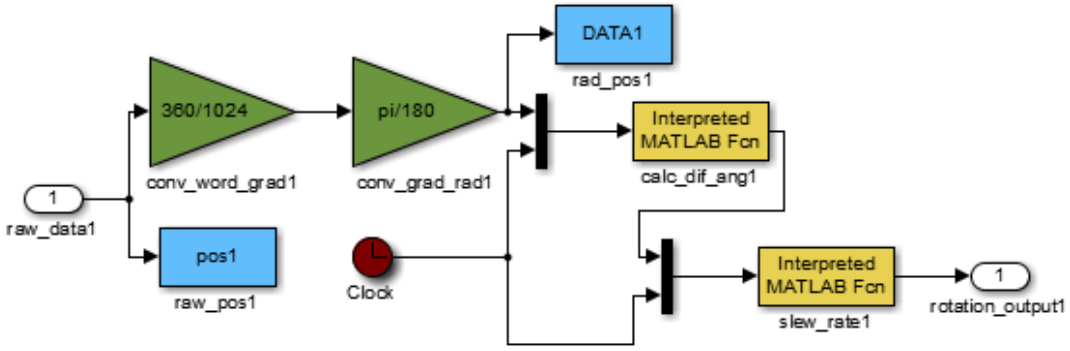


C) Diseño para pruebas de comunicación empleando Arduino

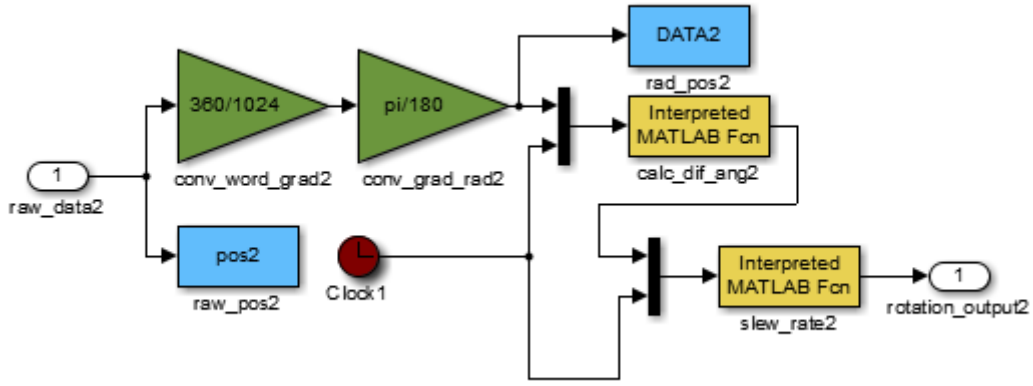
- **Capa superior**



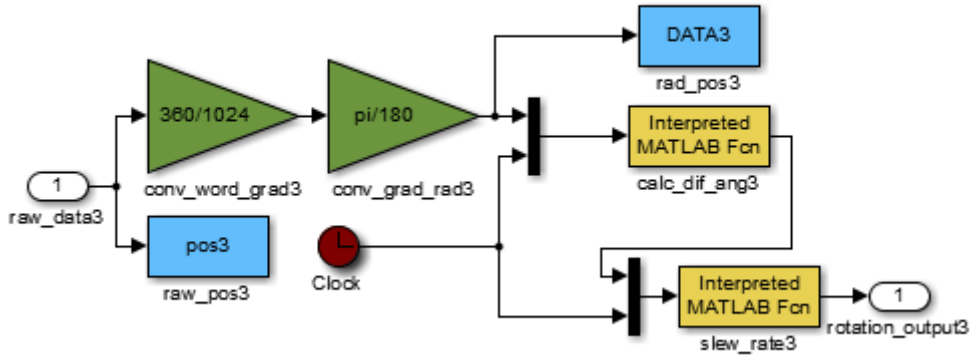
- **Subsystem1**



- **Subsystem2**



- **Subsystem3**

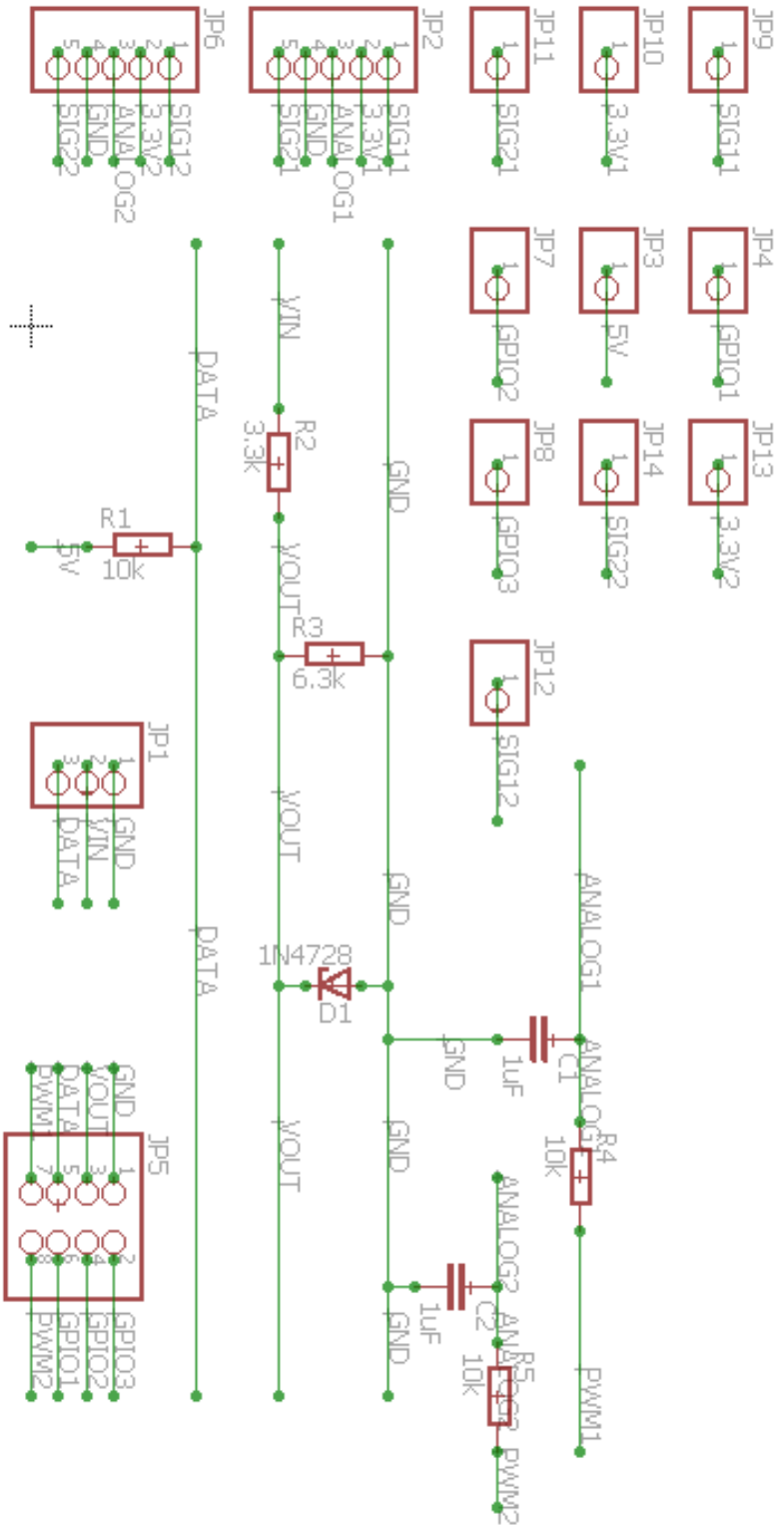


ANEXO III: PCB

Este breve Anexo contiene los esquemáticos e imágenes de la PCB diseñada y fabricada por mí. Algunas de éstas se encuentran también en la memoria, pero he creído conveniente dedicarle un Anexo propio a este recurso.

Las imágenes se muestran a partir de la siguiente página con tal de poder mostrarlas con mayor resolución.

- **Esquemático de la placa**



- **Diseño en formato .brd (EAGLE)**