Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de Sistema de Monitorización de Errores para Ensayos de Vibración

Autor: Ramón José Soldado Arroyo

Tutor: Fernando Muñoz Chavero
Álvaro Ricca Soaje

Departamento de Ingenería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Desarrollo de Sistema de Monitorización de Errores para Ensayos de Vibración

Autor:
Ramón José Soldado Arroyo

Tutor:
Fernando Muñoz Chavero – Profesor titular
Álvaro Ricca Soaje – ALTER Technology

Departamento de Ingenería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Proyecto Fin de Carrera: Desarrollo de Sistema de Monitorización de Errores para Ensayos de Vibración

Autor: Ramón José Soldado Arroyo

Tutor: Fernando Muñoz Chavero
Álvaro Ricca Soaje

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El Secretario del Tribunal

A mi familia y amigos

Agradecimientos

Con este trabajo termina una etapa que comenzó hace ya casi cuatro años, dura, pero a la vez gratificante, en la que todas las experiencias han hecho que me forme como la persona que soy hoy y en la que me he encontrado con gente maravillosa.

Este trabajo está dedicado a toda mi familia y amigos que me han apoyado y animado en los momentos más difíciles, ya que sin su ayuda todo se habría hecho más cuesta arriba.

También está dedicado a todos los buenos profesores que me han enseñado y transmitido su pasión por lo que enseñan, algo que ha hecho que cada vez esté más contento de haber elegido este camino.

Por último, este trabajo no sería posible sin la gente de ALTER, que ha sido muy amable en todo momento, dedicandome un tiempo muy valioso, muestra de ello es este documento.

Ramón José Soldado Arroyo

Sevilla, 2018

Resumen

Los componentes electrónicos destinados para aplicaciones delicadas como la aeroespacial precisan una fiabilidad excepcional en condiciones muy adversas. Para verificar que estos componentes son aptos, deben realizarse una serie de ensayos de diversos tipos.

Este trabajo describe el desarrollo completo de un sistema de monitorización de errores en termistores durante un ensayo de vibración. Este sistema abarca desde el soporte físico del termistor que es ensayado hasta un software en el que el usuario observa el tiempo real los errores que se estan cometiendo en el transcurso del ensayo. Todo esto, en armonía con las diferentes normativas y especificaciones de las mayores agencias espaciales a nivel mundial.

Por tanto, el fin último de este sistema es el de automatizar y paralelizar la realización de medidas durante dicho ensayo, de forma que se ahorre tiempo y dinero al aplicarse en ensayos reales.

Abstract

Electronic components used in delicate applications like aerospace industry require a exceptional reliability in very adverse conditions. In order to verify that those components are valid, certain tests of different matters must be done.

This project describes the full development of a error monitoring system in thermistors during a vibration test. This system encompasses from the thermistor physical support which is tested to the software where the user observe in real time the errors happening in the course of the test. All of this, according with all different regulations and specifications from the most important spacial agencies worldwide.

Therefore, the last purpose of this system is to automate and parallelize the realization of measures during the test so that can save time and money when applied in real tests.

Índice

Índice de Tablas

Índice de Figuras

1 INTRODUCCIÓN

En ciertas industrias como la aeroespacial, la eléctronica es un elemento clave que debe presentar unas condiciones de fiabilidad excepcionales, debido a la delicadeza de dichas aplicaciones. Por esto, existe un mercado de componentes eléctronicos de alta fiabilidad, que precisa de un riguroso control de calidad y se ajusta a una serie de normas y recomendaciones de diversas entidades como la NASA o la ESA.

ALTER Technology es una empresa que se dedica a realizar los pertinentes ensayos a los componentes eléctronicos dedicados a aplicaciones que requieran una alta fiabilidad, como la ya citada industria aeroespacial. En el marco de una cátedra conjunta entre ALTER y la Universidad de Sevilla, se han estudiado estos ensayos y observado la rigurosidad y precauciones necesarias para llevarlos a cabo. Esto ha concluido en el presente trabajo: el desarollo de un sistema de monitorización eléctrica para componentes durante el transcurso de un ensayo de vibración.

Existe un amplio rango de dispositivos cuyas aplicaciones hacen imprescindible que deban funcionar correctamente en entornos con vibraciones muy fuertes y de grandes frecuencias, por lo que es esencial verificar que en estas condiciones el funcionamiento no se ve alterado ni se producen deterioros físicos.

En particular, el sistema desarrollado se centra en la monitorización de termistores, componentes necesarios en gran cantidad de sistemas de medida, por lo que juegan un papel fundamental en los procesos de sensado más críticos.

Como punto de partida, el sistema parte de un ensayo previo realizado a termistores P1K0.0805.4P.B.S de *Innovative Sensor Technology* [1], por lo que el sistema se diseñará en base a las carasterísticas de este dispositivo.

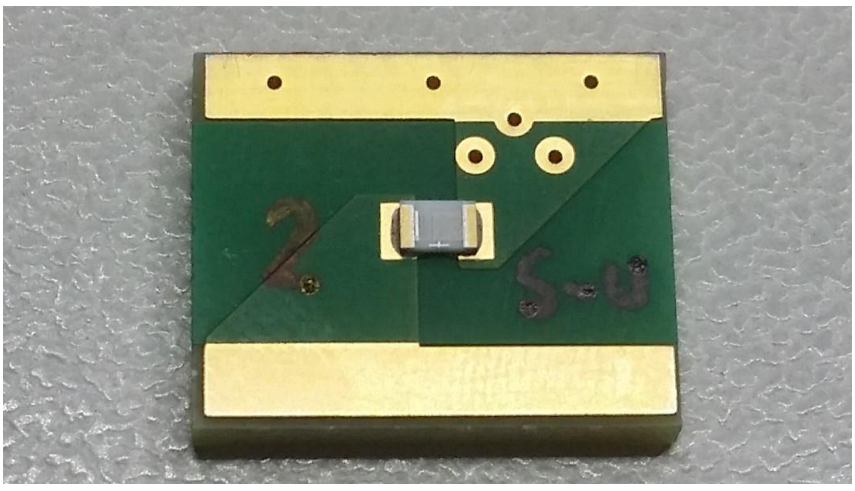


Figura 1-1. Imagen del termistor P1K0.0805.4P.B.S

1.1. Objetivos y alcance

El objetivo principal del proyecto es el de desarrollar un sistema que monitorice eléctricamente el estado de una serie de termistores a los que se le realiza un ensayo de vibración. Este sistema busca automatizar la detección de errores en la medida de lo posible, así como aumentar el número de termistores que se monitorizan a la vez, haciendo posible que en un solo ensayo se prueben varios. Como es lógico, este sistema de monitorización debe ser lo menos intrusivo posible, para no perturbar el desarrollo del ensayo, ya que la monitorización eléctrica solo es una de varias pruebas que se realizan al componente durante dicho ensayo.

Es esencial aplicar las pertinentes normativas de las agencias espaciales que fijan el nivel de fiabilidad necesario para los componentes en determinados entornos.

Este trabajo pretende abarcar el desarrollo completo del sistema, desde el mismo soporte físico sobre el que se ensaya el componente, hasta una interfaz en la que el operador del ensayo pueda consultar con facilidad los resultados. Como parte del desarrollo, el sistema va a ser diseñado, implementado y verificado.

El objetivo es que el sistema sea capaz de monitorizar hasta diez termistores que se ensayan conjuntamente y detectando tanto errores de circuito abierto como de cortocircuito en el termistor.

1.2. Estructura del sistema

El sistema a diseñar sigue una estructura definida de la siguiente manera:

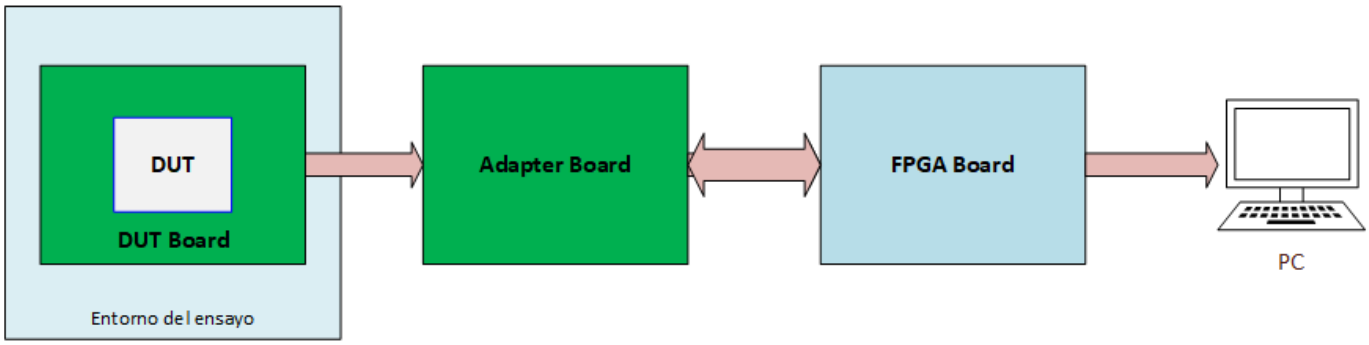


Figura 1-2. Diagrama general de sistema

- **Placa del DUT *(DUT Board):*** Se trata de un PCB diseñado desde cero que cumpla la función de soportar físicamente al DUT para el ensayo, así como de proporcionarle conectividad. Por lo tanto, este PCB estará sometiendose al mismo ensayo, por lo que deberá de tener el mínimo de elementos posibles, para evitar generar errores que no sean producidos por el DUT.
- **Placa de adaptación *(Adapter Board):*** También diseñado desde cero, este PCB tiene una doble función: la de proporcionar la alimentación y estímulos eléctricos necesarios para el DUT durante el ensayo y la de medir las respuestas de este y comprobar si se producen errores, estando fuera del entorno del ensayo para no verse comprometido.
- **Placa de FPGA *(FPGA Board):*** En este caso, se utiliza una placa de FPGA comercial que se encargará de procesar los datos recibidos de la placa de adaptación, debido a su gran eficiencia procesando datos en paralelo.
- **Terminal PC*:*** Como parte final del sistema, se coloca un PC con un software diseñado a medida para comunicarse con la placa de FPGA y obtener la información del ensayo tomada durante este durante el desarrollo del ensayo. El software incluye una GUI que facilita la visualización de resultados y avisa sobre la detección de errores.

1.3. Fases de diseño

Una vez planteado el modelo de sistema que se va a realizar, se plantea la división del trabajo en diversas fases diferenciadas, con el fin de realizar una mejor organización. Estas fases son las siguientes:

1. **Estudio de la metodología de ensayo, normativa y equipos:** Es necesario conocer en profundidad la naturaleza del ensayo sobre los que se va a trabajar, así como de los errores que deben ser detectados, que vienen descritos en la normativa correspondiente. Es necesarios llevar a cabo esta fase rigurosamente para poder realizar las siguientes con criterio suficiente.

2. **Diseño funcional del sistema:** Se realiza un planteamiento general de los requisitos y funcionalidades del sistema y se aborda que soluciones pueden utilizarse para cumplirlas. Esta tarea concluye con el sistema planteado a un nivel considerable de detalle, conociendo qué debe hacer, de qué forma se va a resolver la funcionalidad y cómo se va a organizar esta solución.

3. **Diseño técnico del sistema:** El planteamiento anterior se aborda realizando el paso de bloques funcionales ideales e ideas a componentes y subsistemas reales existentes, verificando mediante simulaciones y pruebas que se cumplen los requisitos establecidos anteriormente.

4. **Implementación/Fabricación del sistema:** Una vez el sistema está completamente diseñado y simulado, se procede a la fabricación o implementación de este, dependiendo de cual sea la naturaleza del sistema diseñado.

5. **Verificación del sistema:** El sistema fabricado completo debe verificar el correcto funcionamiento que se planteó en el diseño funcional. Para ello, se llevan a cabo diversas pruebas que comprueben que el comportamiento es el esperado.

1.4. Estructura de la memoria

Con el objetivo de exponer el trabajo realizado de la manera más eficiente posible y respetando el flujo de trabajo expuesto, se va a organizar la presente memoria de la siguiente forma:

1. Introducción
2. Descripción del ensayo de vibración
3. Diseño de PCB
4. Diseño hardware de FPGA
5. Diseño software de PC
6. Pruebas
7. Conclusiones y Desarrollos futuros

Con esta estructura se pretende desarrollar las fases de diseño ya expuestas sobre los diferentes componentes del sistema, que como se ha visto son de diferente naturaleza, por lo que se precisarán herramientas y técnicas diferentes.

2 DESCRIPCIÓN DEL ENSAYO DE VIBRACIÓN

Antes de poder abordar el diseño del sistema, necesitamos conocer con todo detalle el ensayo que se realiza durante la monitorización eléctrica. Es imprescindible abordar la normativa y estándares de diferentes agencias espaciales y cuerpos militares que se aplican en este tipo de ensayos.

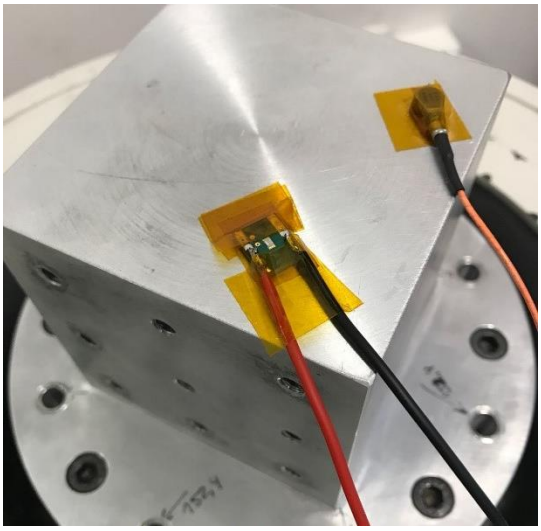


Figura 2-1. Setup de un ensayo de vibración

La normativa y especificaciones relativa tanto a ensayos de vibración de alta frecuencia (*Vibration, High Frequency)* como de termistores fijan una serie de condiciones y requisitos que se deben cumplir.

Sobre las condiciones de vibración [2] a las que se debe someter a los termistores obtenemos una serie de parámetros a aplicar. Pese a que no influyen de ninguna manera en las carasterísticas del sistema de monitorización, es conveniente conocer la magnitud y duración del ensayo.

Tabla 2-1. Parámetros de vibración para el ensayo en termistores

Frecuencia	**Amplitud**	**Tolerancia**
10 – 80.54 Hz	0.06 pulgadas (pico a pico)	±10%
80.54 – 2000 Hz	20 G* (pico simple)	±10%
Duración	**Ciclos**	**Ejes**
20 minutos / ciclo	12 ciclos / eje	3 ejes perpendiculares

[∗] $G = 0{,}0512 * f^2 * DA$, siendo f la frencuencia en hercios y DA la amplitud doble en pulgadas

En lo relativo a la medida eléctrica, se especifica que cada termistor debe de ser monitorizado para determinar una discontinuidad eléctrica por un tiempo de al menos 0,1 ms. Este es el objetivo principal del sistema, detectar errores de estas carasterísticas [3].

Relativo al montaje y adecuamiento del dispositivo, se especifica que el DUT debe ir montado en una PCB diseñada de forma que no cause ningún fallo que pueda hacer detectar al sistema un error ajeno al ensayo o el termistor. Es posible ubicar más de un DUT en la misma PCB, siempre que el área libre de esta sea de al menos cuatro veces el área ocupada por los dispositivos [3].

Para soldar los DUT al PCB se debe usar una resina epoxi tipo 2 aprobada para actividades de calificación. Debe ser curada al menos a 150 ºC durante 30 minutos [3].

Algunos procedimientos necesarios para los ensayos de vibración como las medidas de *zero power* o las inspecciones visuales [3] [4] son ajenos a la monitorización eléctrica y por tanto al desarollo de la funcionalidad del sistema, por lo que no se van a describir.

3 DISEÑO DE PCB

Siguiendo el flujo de diseño previsto y teniendo en cuenta los requisitos y carasterísticas necesarias obtenidas a partir de los estándares y de las propias condiciones de diseño impuestas al sistema, se comienza el diseño de las PCB que abarcará los dos primeros elementos del sistema: la placa DUT y la de adaptación.

3.1. Diseño funcional

La carga funcional de esta parte recae casi enteramente sobre la placa de adaptación, ya que como la placa del DUT va a estar expuesta al entorno de vibración, su única función será albergar el DUT y proporcionar conectividad eléctrica.

En una primera aproximación, los termistores son tratados como resistencias simples de valor fijo, suponiendo que la única variación de su valor de resistencia pueda ser causada por la acción de la vibración, por lo que no tendremos en cuenta la variación causada por la temperatura.

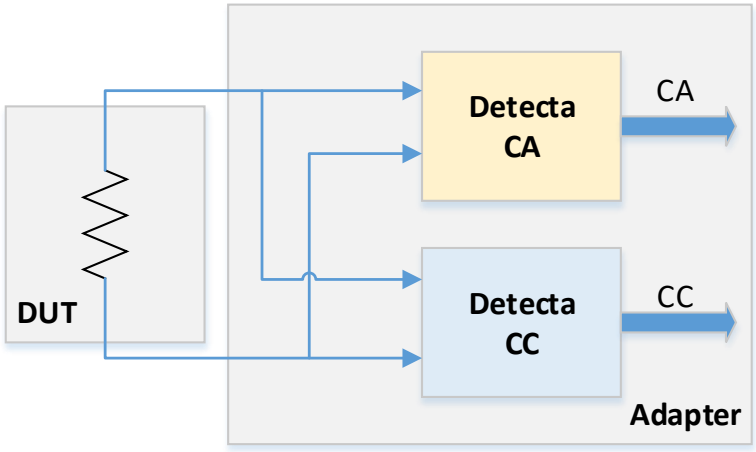


Figura 3-1. Diagrama funcional de ambas PCB

Principalmente, la función de la placa de adaptación será la de detectar cuando el termistor produce un circuito abierto o un circuito cerrado. Es decir, cuando el termistor pasa a tener resistencia infinita o nula, respectivamente. Cuando esto ocurra, se generará una señal discreta de un solo bit, que indique a la FPGA que está ocurriendo el error pertinente, por lo que aunque la entrada sea analógica, la salida será digital.

La forma de conocer si se producen estos errores consiste en observar las variaciones de corriente sobre el DUT al ser alimentado con una tensión constante. Esto se realiza con un circuito cuya función sea medir la corriente, y cuya salida sea una tensión. En el posterior circuito comparador se realizan comparaciones de esta tensión para que cuando alcancen cierto valor umbral, se envíe la señal correspondiente al error.

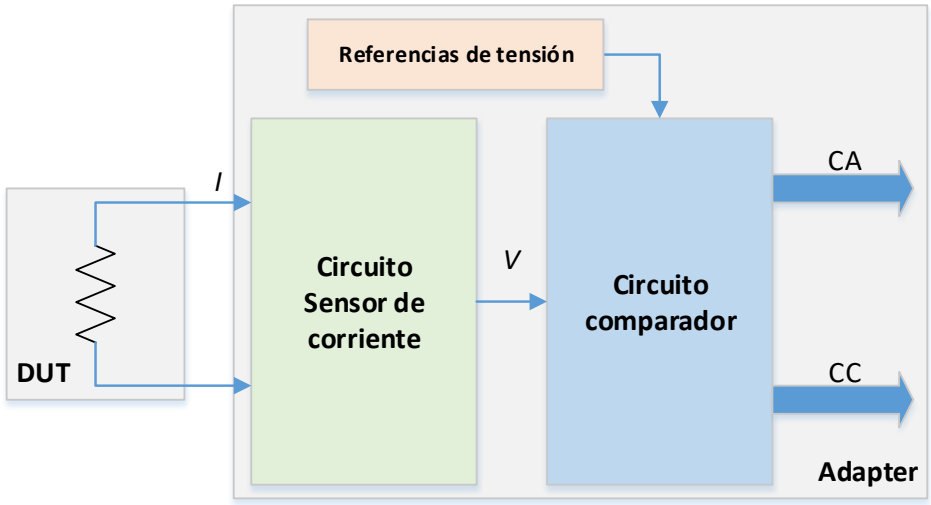


Figura 3-2. Diagrama funcional avanzado de ambas PCB

Una vez se especifican los diferentes subcircuitos necesarios y sus funcionalidades, se puede empezar a realizar el diseño de estos a un nivel más técnico, empleando componentes reales y aplicando parámetros.

3.2. Diseño esquemático del circuito principal

Se va a realizar un diseño esquemático del circuito que realiza la funcionalidad definida anteriormente. Elementos secundarios como fuentes de alimentación, conectores o capacidades de desacoplo no serán incluidos todavía.

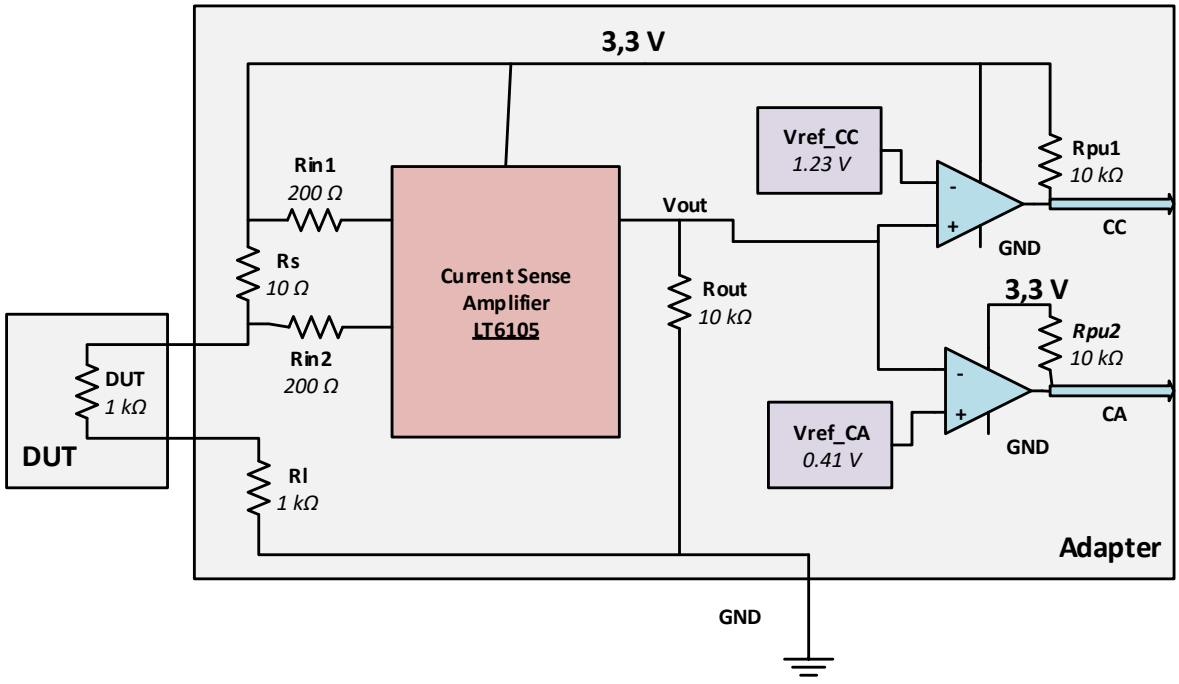


Figura 3-3. Esquemático del circuito principal

Para el circuito sensor de corriente se ha elegido una configuración en la que mediante un sensor de corriente (Current Sense Amplifier del modelo **LT6105)** se mide y amplifica la corriente que pasa por una resistencia de shunt (**R_S)**, que es la misma corriente que pasa por el DUT, ya que están en serie.

Existe una resistencia de carga **(R_L)** que sirve para limitar la corriente a un valor aceptable, ya que Rs es muy pequeña y en caso de cortocircuito en el DUT se dispararía el consumo de corriente.

El integrado **LT6105** permite ajustar la ganancia mediante las resistencias **R_{in}** y **R_{out}** , por lo que ajustando los valores de las diferentes resistencias, se puede controlar el rango de valores que habrá a la salida del subcircuito sensor de corriente, para facilitar el diseño del siguiente subcircuito.

$$I_{nom} = \frac{V_{DD}}{(R_S + DUT + R_L)} = \frac{3{,}3\ V}{(10\ \Omega + 1\ K\Omega + 1\ K\Omega)} = 1{,}6418\ mA \quad (3\text{-}1)$$

$$G = \frac{R_{out}}{R_{in}} = \frac{10\ K\Omega}{200\ \Omega} = 50 \quad (3\text{-}2) \qquad V_{out} = I_{nom} * R_S * G = 0.82\ V \quad (3\text{-}3)$$

Estos parámetros seleccionados garantizan una tensión de salida en un rango cómodo para trabajar, que no llega a superar en ningún momento la tensión de alimentación y con valores de resistencias y ganancias dentro de lo recomendado por el fabricante del integrado LT6105.

$$Si\ ocurre\ CC: \quad DUT = 0\ \Omega \quad I_{CC} = 3{,}267\ mA \quad V_{out}^{CC} = 1{,}633\ V \quad (3\text{-}4)$$

$$Si\ ocurre\ CA: \quad DUT = \infty\ \Omega \quad I_{CA} = 0\ mA \quad V_{out}^{CA} = 0\ V \quad (3\text{-}5)$$

Se comprueba que cuando el valor de la resistencia del DUT llega a valores extremos, se producen tensiones que también están dentro de un rango aceptable.

Respecto al subcircuito que se denomina circuito comparador, se utilizan dos comparadores que comparan la salida V_{out} con ciertos valores de tensión que vienen fijados como referencias.

El criterio para fijar estos valores viene dado por un umbral elegido en base a un criterio propio, ya que la norma no especifica concretamente. Este umbral se ha fijado en ± 50% de la intensidad nominal, que se ha considerado variación suficiente para poder ser considerado fuente de discontinuidad eléctrica.

$$Umbral\ de\ CC: \quad I_{UCC} = 2{,}463\ mA \quad V_{out}^{UCC} = 1{,}23\ V \quad (3\text{-}6)$$

$$Umbral\ de\ CA: \quad I_{UCA} = 0{,}82\ mA \quad V_{out}^{UCA} = 0{,}41\ V \quad (3\text{-}7)$$

Estos comparadores se han elegido con salida en colector abierto, de modo que la parte analógica quede completamente separada de la digital, por lo que es necesario colocar resistencias de pull-up a la salida del comparador.

3.2.1 Simulación del circuito principal

Antes de seguir con el diseño de los diferentes elementos que faltan en el circuito, se realiza una simulación con la finalidad de comprobar si el comportamiento teórico se cumple. Para ello se va a utilizar la herramienta **LTSpice.**

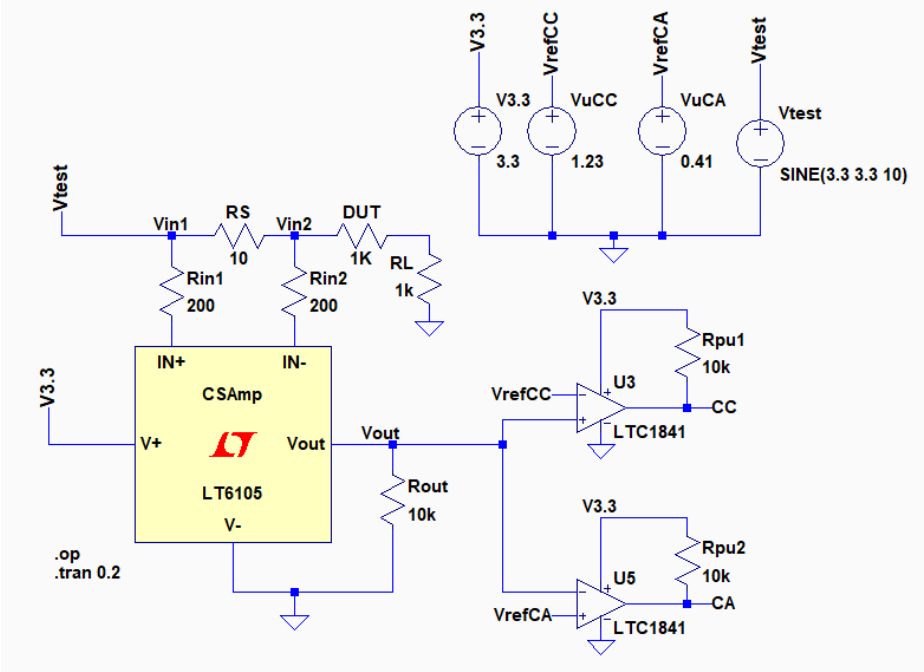


Figura 3-4. Esquemático del circuito principal en LTSpice

Para comprobar si el funcionamiento de los comparadores es tal y como esperamos ante variaciones del valor de resistencia del DUT, se coloca una fuente de tensión senoidal como alimentación a las resistencias. Este comportamiento no es exactamente el que ocurre en un escenario real, pero el efecto que tiene sobre la resistencia R_s sí es el mismo, por lo que se considera válido para realizar esta primera simulación.

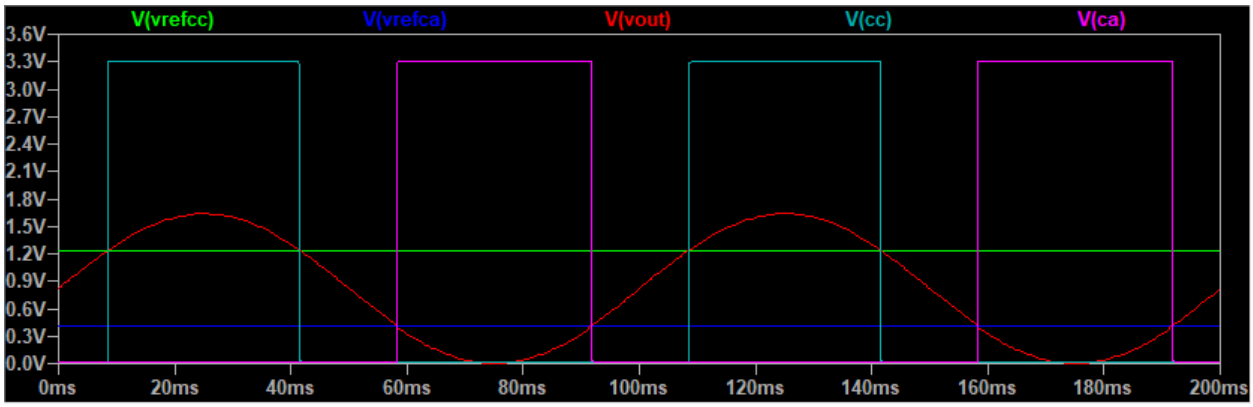


Figura 3-5. Resultados de la simulación en LTSpice

En los resultados de la simulación se observa como se consigue que las salidas CC y CA que van a la FPGA se comporten de forma binaria, solo activandose al sobrepasar el umbral fijado anteriormente. Por lo tanto podemos concluir que los parámetros elegidos en este circuito son correctos.

3.3. Diseño esquemático del circuito final

Una vez simulado el circuito que va a detectar los errores en cada DUT, se plantea el circuito completo teniendo en cuenta que se necesita capacidad para ensayar hasta 10 DUT a la vez. Además, es necesario añadir los elementos de alimentación, así como conectores y algunos detalles restantes. Para realizar el esquemático definitivo se utiliza el programa **Altium Designer**, que es la herramienta que utilizaremos a lo largo del resto del diseño.

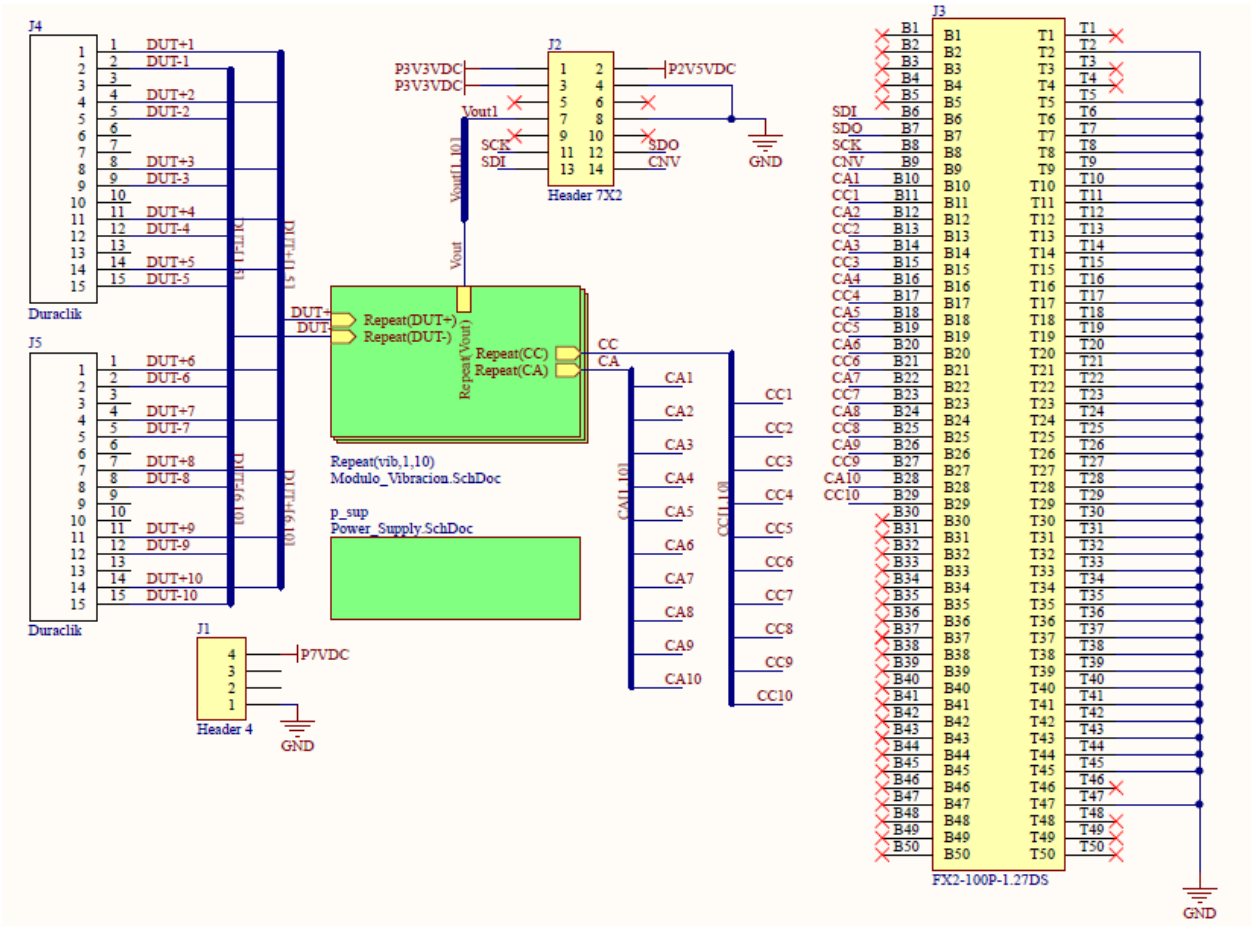


Figura 3-6. Esquemático completo modularizado en Altium

El circuito toma una envergadura suficiente, por lo que se modulariza para facilitar el trabajo. En este caso tenemos dos módulos diferentes llamados **Power_Supply** y **Modulo_Vibracion.** Además de los módulos, existen conectores para las diferentes entradas y salidas de los módulos.

Una caraterística no mencionada anteriormente es la interconexión y alimentación de un ADC de manera complementaria, que controlado por la FPGA proporcionará el valor digital de la tensión a la salida del LT6105 en un DUT determinado. Esta funcionalidad extra permite conocer la variación de la resistencia de un DUT durante el ensayo.

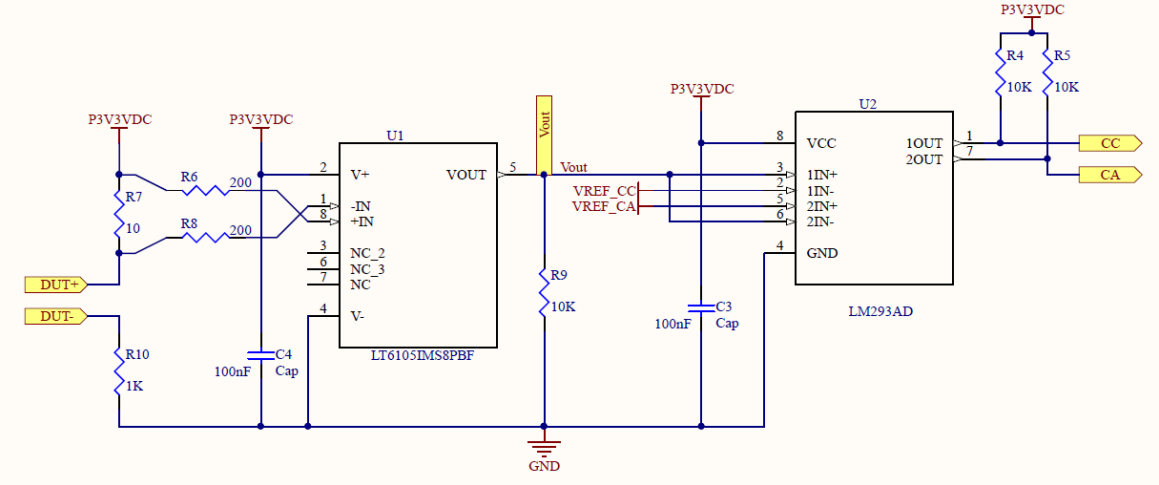


Figura 3-7. Esquemático de Modulo_Vibracion

El módulo llamado **Modulo_Vibracion** es esencialmente el circuito principal ya descrito. Este módulo se encuentra replicado diez veces en el esquemático principal mediante la función *Repeat* que nos proporciona el programa. Como diferencia del anterior esquemático, se ha elegido el integrado LM293AD que contiene dos comparadores y se han añadido unas capacidades de desacoplo de 100 nF en las alimentaciones de ambos integrados.

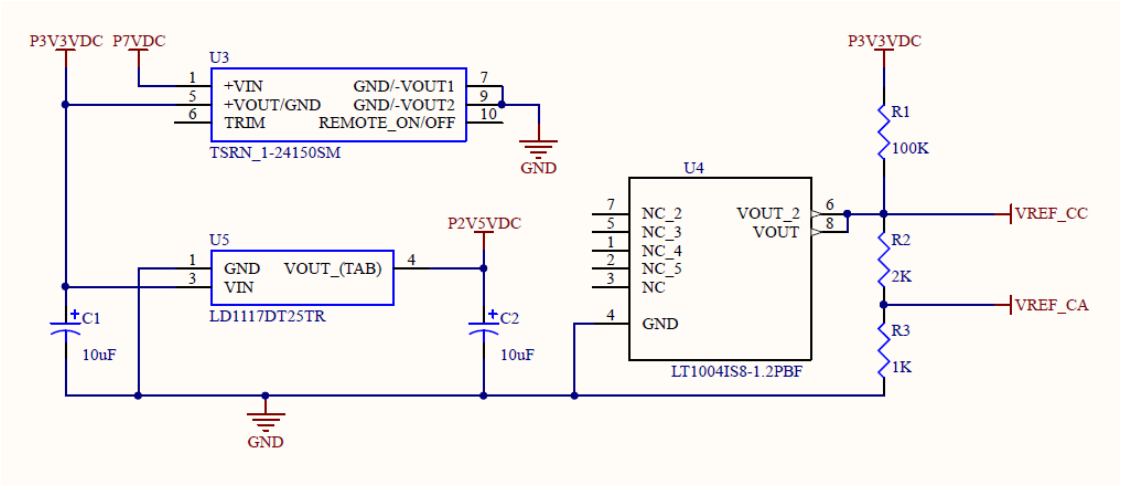


Figura 3-8. Esquemático del módulo de alimentación

El módulo **Power_Supply** se encarga de proporcionar las alimentaciones necesarias para los diferentes integrados y elementos del resto del circuito. En general, todo el circuito está alimentado a 3,3 V, por lo que se utiliza un regulador de tensión conmutado TRACOPOWER para abastecer a todo el circuito a partir de una alimentación externa que aunque puede ser variable, se fija a 7 V.

Además, el ADC precisa una alimentación de 2,5 V**,** por lo que se coloca un regulador LDO que se alimenta de la salida del anterior regulador. El fabricante recomienda la colocación de dos condensadores de 10 µF a las entradas y salidas de tensión.

Por ultimo, se coloca una referencia de tensión de 1,253 V y una precisión de 0,3 % para crear los valores de voltaje umbrales de los comparadores. Esta referencia actúa como un diodo zéner. Usamos su valor de salida para la referencia umbral de cortocircuito y para la de circuito abierto realizamos un divisor resistivo con resistencias de alta precision, con 0,1 % de tolerancia.

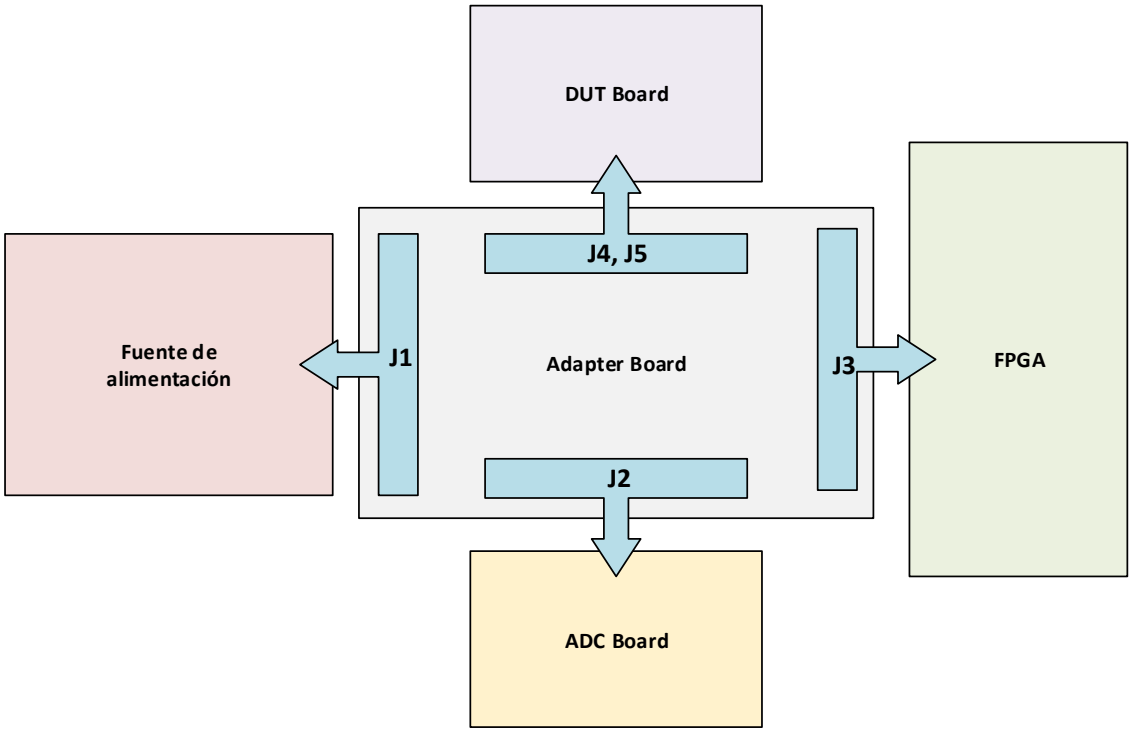


Figura 3-9. Diagrama de conexiones de la placa de adaptación

Respecto a los diferentes conectores presentes en el esquemático general:

- La **alimentación** del circuito, prevista a 7 V, se realiza mediante un conector sencillo de cuatro pines, de los que solo se utilizarán dos: el postivo y la tierra.
- Para la conexión con la **placa de los DUT** se han elegido unos conectores especiales de la serie *Molex Duraclik* especialmente diseñados para aguantar entornos de vibración, ya que de sebe evitar que el conector cause los errores que se pretenden medir. Estos conectores tienen quince contactos, por lo que será necesario colocar dos de ellos para poder llevar a cabo las veinte conexiones existentes.
- La conexión con la **placa del ADC** se realiza mediante un conector estándar de catorce contactos que sirve para enviar las alimentaciones necesarias, las señales a digitalizar y las señales digitales con las que se comunica con la FPGA. Para adecuar todas en el mismo conector se han sobredimensionado los contactos, permitiendo así dejar contactos libres e impeder cualquier tipo de interferencia posible entre diferentes tipos de señales.
- La conexión con la **FPGA** se realiza mediante un conector de *Hirose FX2* de 100 pines, ya que es el que se encuentra en la placa de FPGA que vamos a utilizar, la *Spartan 3E Starter Board* [5]. Hacia la FPGA van las diferentes señales de salida de error de los DUT, que son veinte en total. Además, se incluyen las señales de control, entrada y salida del ADC, para permitir su utilización.

3.3.1 Worst Case Analysis

Antes de dar por finalizado el diseño, se va a realizar una última comprobación mediante un análisis de peor caso posible *(Worst Case Analysis)*. El objetivo es comprobar si el funcionamiento es correcto en el peor caso posible, teniendo en cuenta las tolerancias de los componentes involucrados en la realización de la funcionalidad.

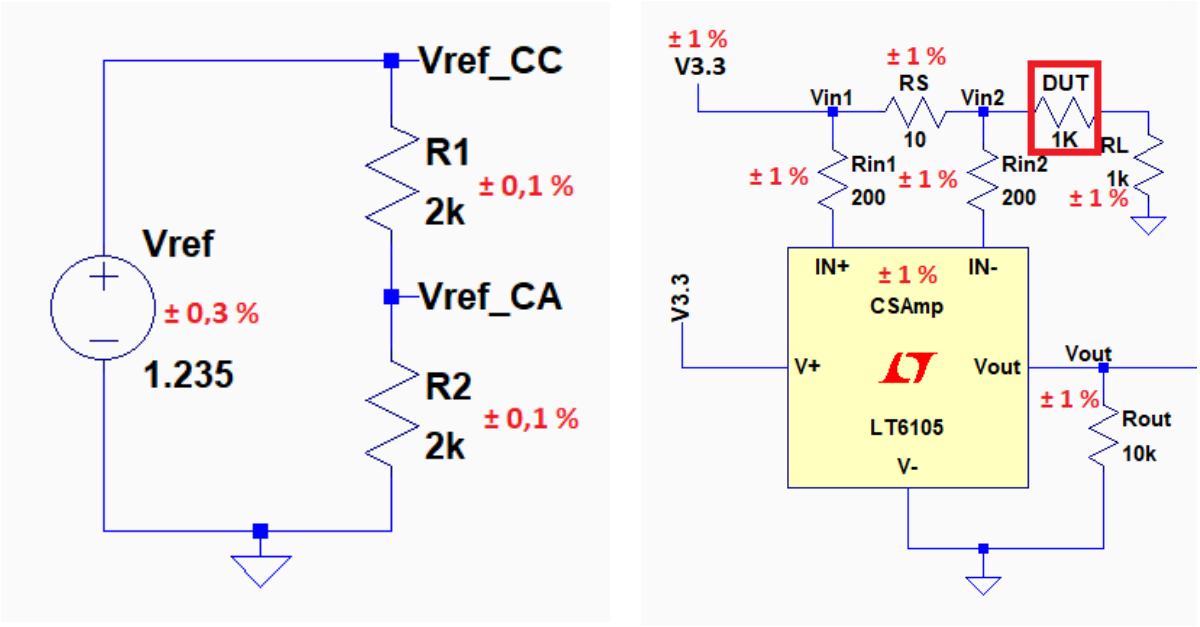


Figura 3-10. Esquemáticos con las tolerancias de error indicadas

Consultando las hojas de datos del fabricante de cada componente obtenemos las diferentes tolerancias de los dispositivos, que son todas del 1% a excepción de los componentes de precisión utilizados para generar ambas tensiones de referencias.

Tabla 3-1. Intervalos de valores posibles para los parámetros del circuito

V_{REF}^{CC}	V_{REF}^{CA}	G
$[\,1{,}231\,;\,1{,}239]\,V$	$[\,0{,}410\,;\,0{,}414]\,V$	$[\,48{,}52\,;\,51{,}52]$

Parámetro	**Normal**	**Cortocircuito**	**Circuito Abierto**
V_{RS}	$[16\,;\,16{,}83]\,mV$	$[\,31{,}72\,;\,33{,}66]\,mV$	$[0\,;\,0]\,mV$
V_{OUT}	$[0{,}776\,;\,0{,}867]\,V$	$[1{,}539\,;\,1{,}734]\,V$	$[0\,;\,0]\,V$

Se realizan los cálculos correspondientes para verificar que los valores extremos de las referencias nunca llegan a entrar en conflicto con los valores teóricos extremos referentes a los tres estados posibles del DUT. Además, se comprueba que hay cierto margen para que el posible ruido no cree falsos errores.

3.4. Diseño PCB

Una vez diseñados los esquemáticos y comprobada la funcionalidad del circuito, se pasa de un diseño esquemático a uno PCB. Usaremos la herramienta **Altium Designer**, partiendo del esquemático completo creado en esta misma plataforma.

Para pasar a PCB es necesario obtener las huellas de los diferentes componentes, las cuales pueden ser obtenidas de librerias de fabricantes, del propio programa o de sitios externos como *SnapEDA*. En el caso de algunos componentes más raros como es el caso del conector *Hirose FX2* es necesario crear la huella desde cero, siguiendo los planos proporcionados por el fabricante.

Es muy importante comprobar antes de comenzar que todas las huellas se corresponden a los componentes elegidos, verificando con los planos ofrecidos por el fabricante. Una vez que se ha comprobado que todas las huellas se adaptan a los componentes utilizados, puede comenzar el rutado de ambas placas.

Se va a realizar un rutado a dos caras, intentando realizar todo el rutado posible sobre la superior, donde van a ir colocados los componentes, y reservando la inferior para los usos en los que no fuese posible usar la capa superior.

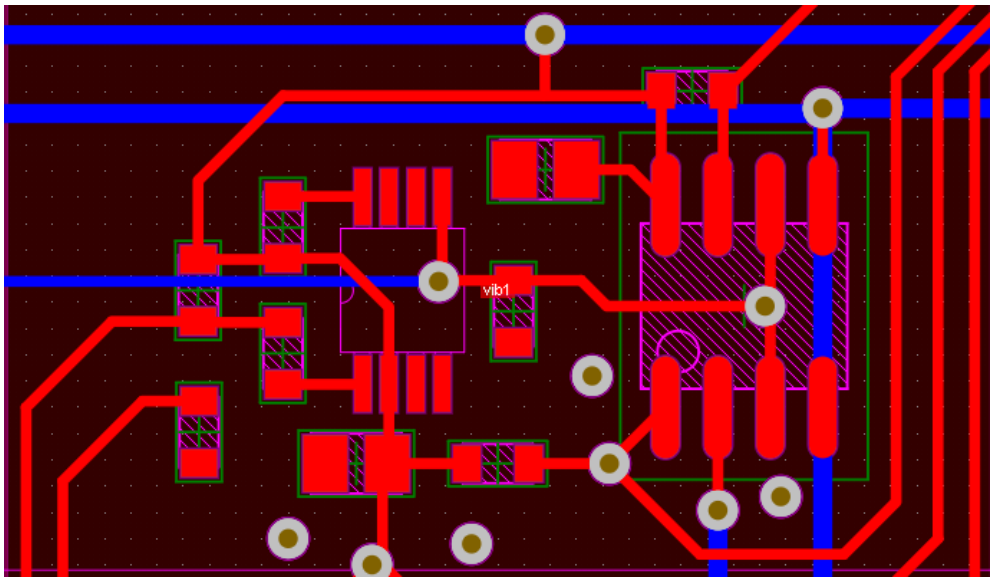


Figura 3-11. Rutado de Modulo_Vibracion

Aprovechando el diseño modular utlizado, el programa crea automáticamente unas áreas (*rooms*) que abarcan solamente los componentes de un módulo determinado, por lo que de nuevo el diseño se realiza primero a nivel de módulos para terminar finalmente interconectandolos entre sí.

Cada **Modulo_Vibración** contiene pocos componentes, que inicialmente deben ubicarse de forma que el rutado sea lo más simple posible, teniendo en cuenta la necesidad de dejar espacio para las conexiones con los conectores, que se ubicarán en los extremos de la placa.

Una vez ubicados los componentes, comienza el rutado, que se ha realizado siguiendo algunas pautas generales en toda la placa:

- Ancho de pista predeterminado de 0,3 mm. Variará en función de la necesidad.
- Giros en pistas de 45º, buscando que haya cierta separación entre giros próximos para disminuir el efecto sobre la señal.
- Uso de vías para cambiar de plano y para hacer conexiones en estrella incluso en el mismo plano.
- Colocación de los condensadores de desacoplo lo más próximos posible al pad de alimentación del integrado que les corresponda.

En un primer rutado no se realiza ninguna conexión a tierra, ya que al final del diseño se rellenará el espacio sin ocupar por pistas de la placa con un plano a tierra al que se conectarán los terminales que lo necesiten.

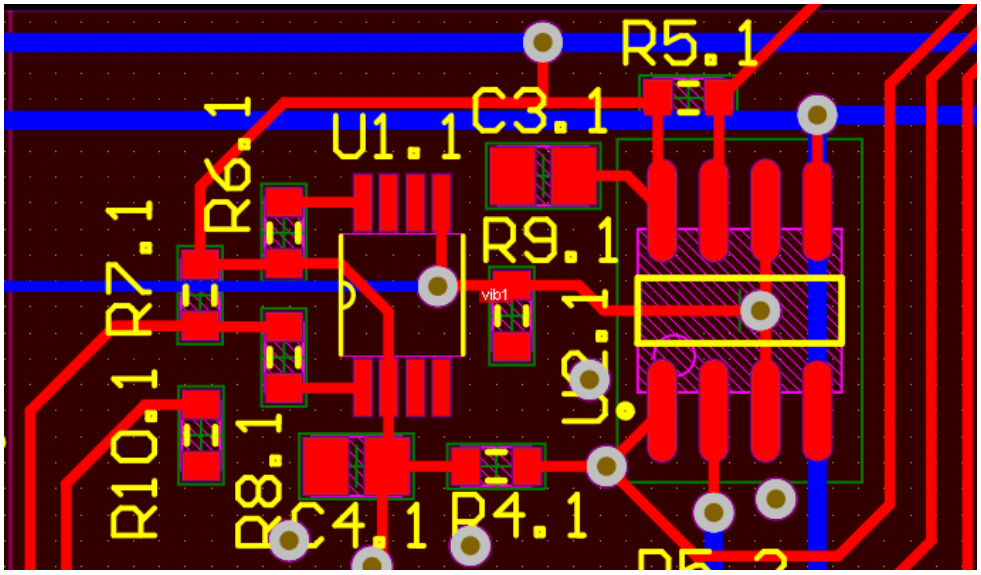


Figura 3-12. Rutado de Módulo_Vibracion con serigrafía incluido

Una vez se termina el rutado del módulo, activamos la capa de serigrafía (*Top Overlay*) en el editor para colocar los nombres de los componentes en un lugar visible, evitando pads y vías en la medida de lo posible. Es algo importante de cara a la fabricación del dispositivo, ya que nos permitirá localizar y distinguir las huellas de los diferentes componentes una vez fabricado.

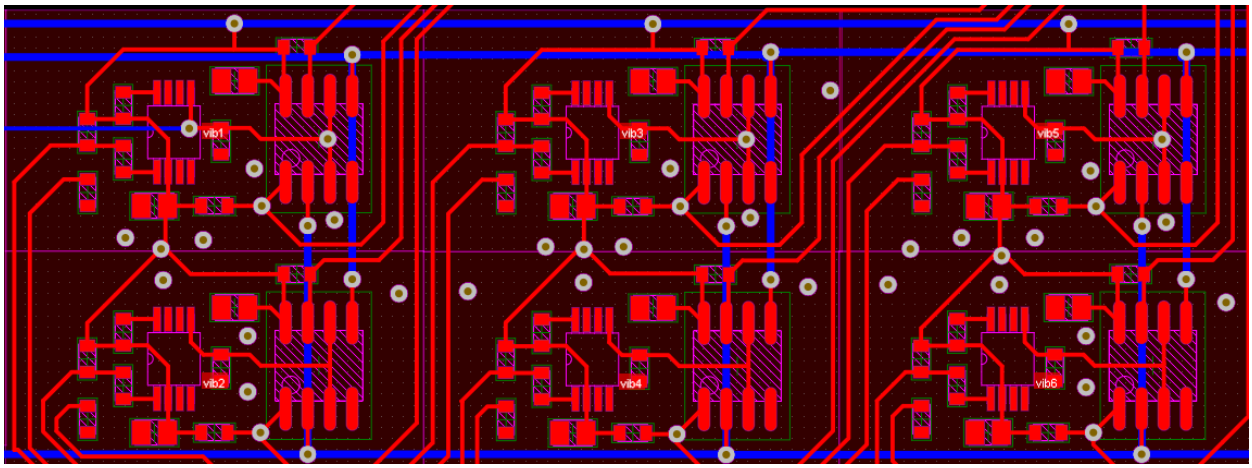


Figura 3-13. Rutado de varios módulos interconectados entre sí

Utilizando las herramientas de módulos que nos ofrece el editor, podemos copiar el rutado de una room en el resto de sus repeticiones. Esto simplifica la tarea, aunque es necesaria la revisión cuidadosa de cada uno de los módulos para verificar que todo se ha conectado como debía.

Se realizan rutados entre módulos para elementos comunes como alimentaciones o referencias de tensión y se preveen espacios suficientes para poder realizar las conexiones con los conectores. Este proceso es complejo debido al gran número de señales y precisa una buena organización previa y posiblemente varias iteraciones para encontrar un diseño cómodo y funcional.

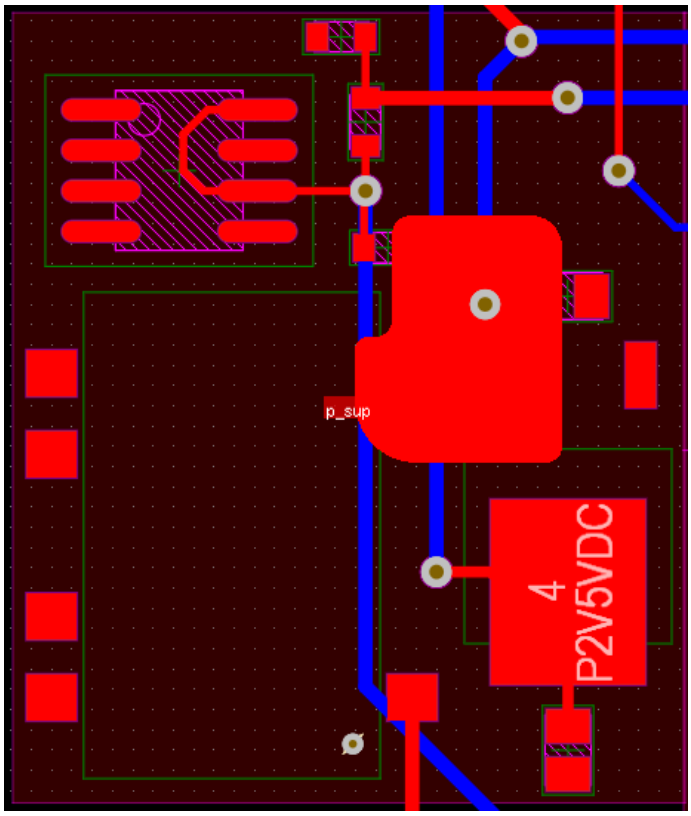


Figura 3-14. Rutado del módulo de alimentación

El **módulo de alimentación** es relativamente sencillo ya que solo cuenta con unas cuantas salidas, por lo que se puede hacer buscando una configuración que respete la disposición de las alimentaciones de los módulos de vibración sin mucha dificultad.

En este caso se ha colocado una pestaña en la salida de la tensión principal de alimentación a 3,3 V, ya que como por ahí pasará casi toda la corriente consumida, se busca reducir la resistencia lo máximo posible.

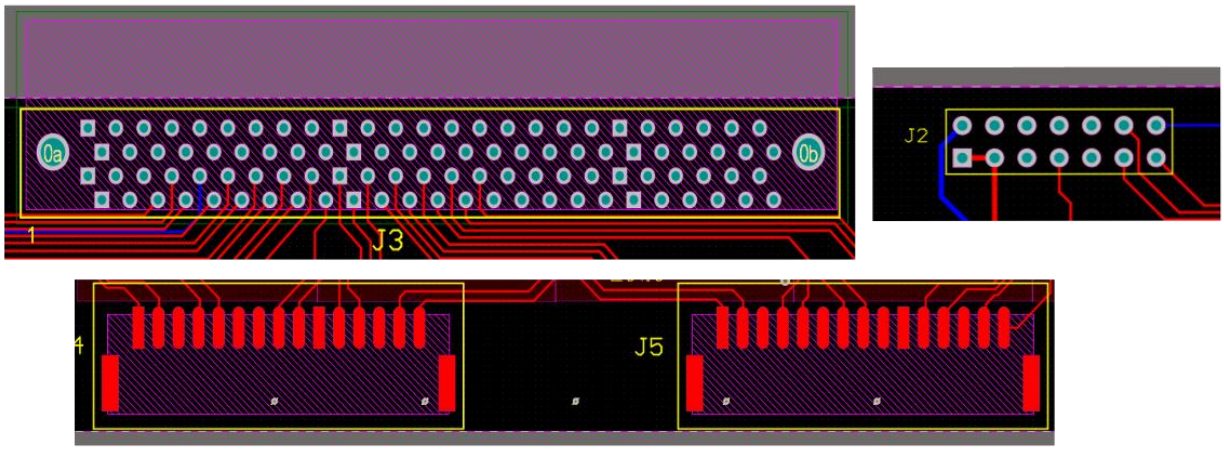


Figura 3-15. Rutado y colocación de conectores

Por último, los conectores deben ir colocados en una zona cómoda para el rutado de las señales, teniendo en cuenta otros factores como el tamaño de la placa y los espacios entre sí.

El conector más grande, que va hacia la FPGA, se ha colocado en la parte superior de la placa. Su huella ha tenido que ser diseñada desde cero siguiendo los planos del fabricante. Es imprescindible conocer la configuración del conector en la FPGA, para rutar las señales a los pines que correspondan con entradas/salidas de propósito general [5].

Igualmente, es necesario saber la configuración de las otras dos placas con las que se realizará la conexión para que los pines sean los que correspondan.

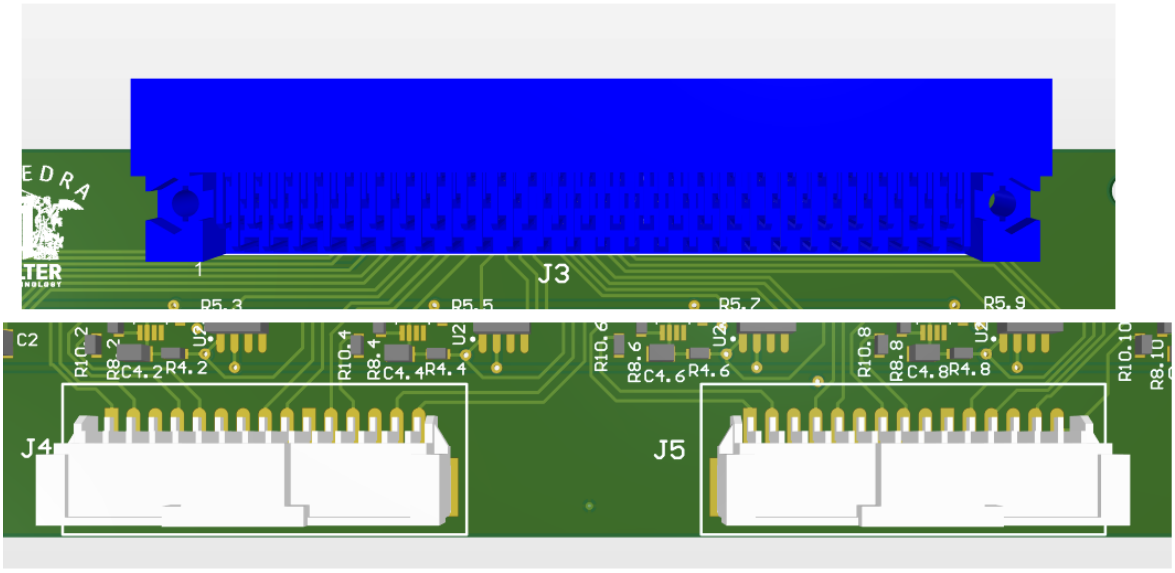
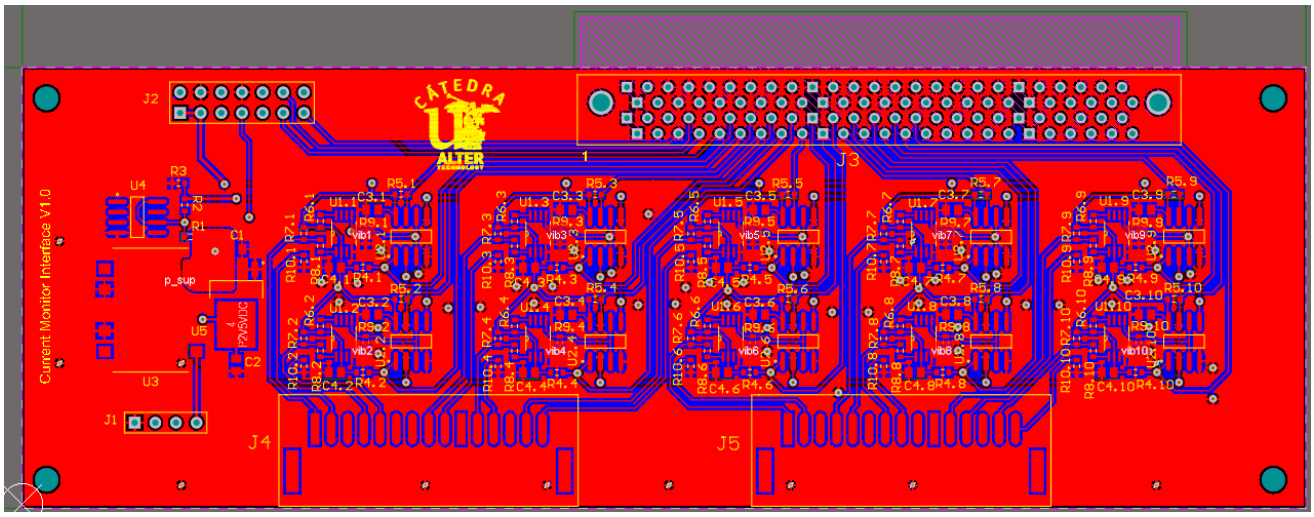


Figura 3-16. Modelo 3D de los conectores ubicados

El editor nos permite añadir los modelos 3D que ofrecen los fabricantes, lo que sumado a la huella que ya teníamos nos permite comprobar que el conector está ubicado correctamente y medir la distancia al borde necesaria. En este caso observamos que el conector del DUT está completamente dentro de la placa, pero el de la FPGA sale, lo cual es necesario para que encaje la otra parte del conector.



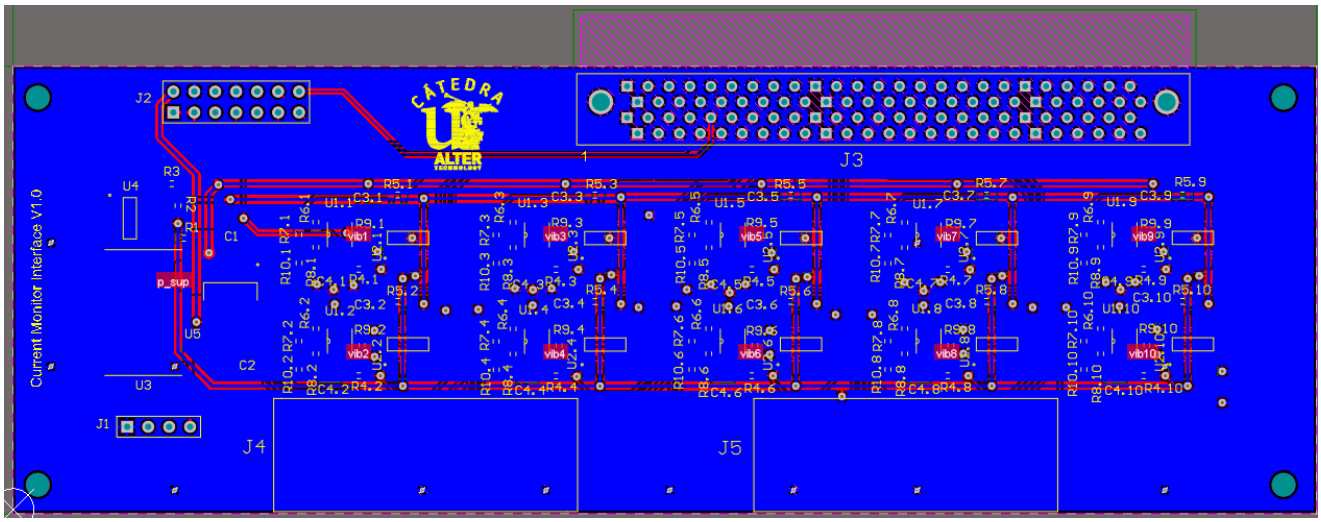


Figura 3-17. Resultado final del rutado del PCB de adaptación en ambas capas

Para finalizar el rutado, se terminan las conexiones entre todas las pistas y pads necesarios. Una vez realizado, se localizan las pistas que mayor carga de corriente vayan a tener, especialmente las alimentaciones y se ensanchan hasta 0,5 mm de grosor. También se realiza esto con las pistas de las referencias de tensión, para evitar que las diferencias de longitudes hasta los diferentes comparadores produzcan una variación en la tensión recibida.

Una vez todo está rutado y revisado, se colocan los planos de tierra en ambas capas. Ya que como por la tierra pasa de vuelta toda la corriente del circuito, es necesario que su resistencia sea la menor posible. El editor posee una herramienta que generará el plano y lo conectará a los pads de tierra que anteriormente quedaron sin conectar.

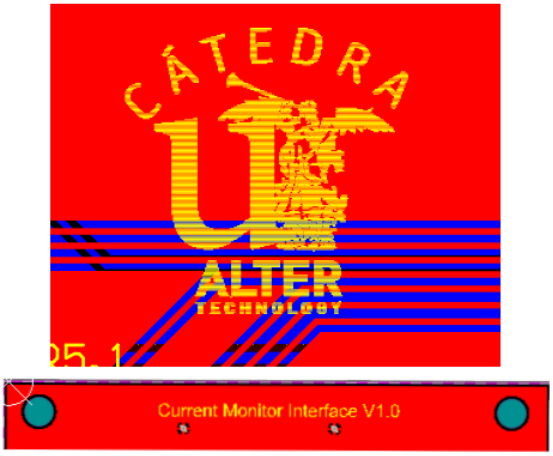


Figura 3-18. Detalles estéticos finales en la placa de adaptación

Por último, se añaden detalles estéticos de serigrafía como el logo de la cátedra en la que se enmarca el trabajo o el nombre para la placa. También se añaden agujeros en las esquinas para colocar soportes.

Por otro lado, la **PCB de los DUT** se realiza de manera similar, pero simplificando mucho el proceso ya que este circuito solo cuenta con los DUT. Es necesario recordar que la normativa requería que el espacio libre del PCB fuera al menos cuatro veces mayor al ocupado por los DUT.

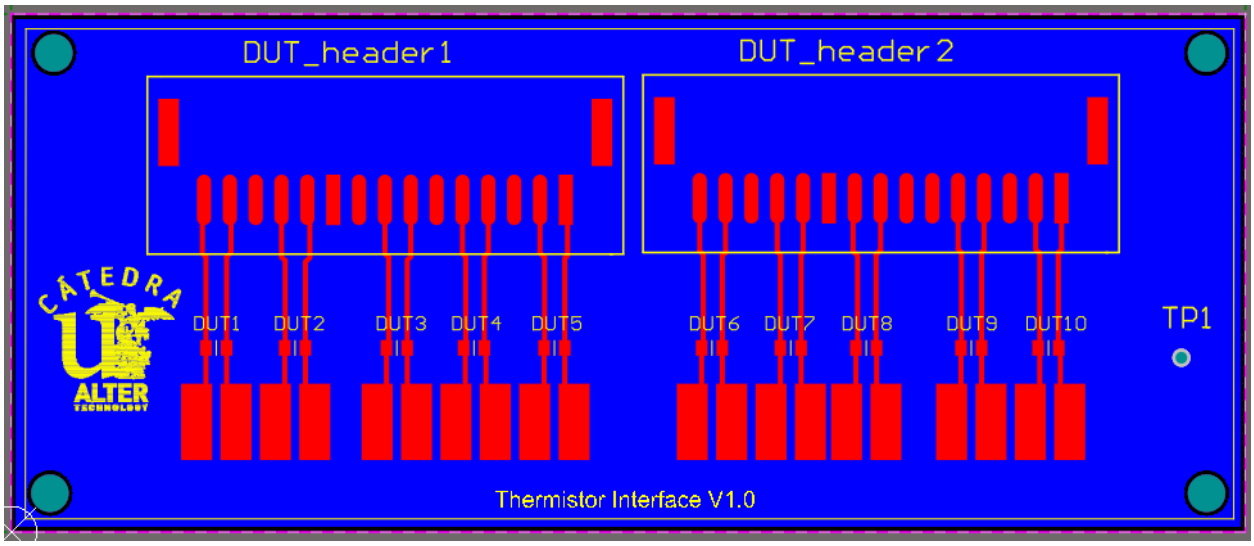


Figura 3-19. Resultado final del rutado del PCB de DUT

En este caso, se prevee la posibilidad de colocar unos conectores *Molex Duraclik* de la misma manera que en la de adaptación, ya que están protegidos ante entornos de vibración, pero debido a que los ensayos de vibración suelen tener un carácter destructivo, usualmente se sueldan los cables de conexión directamente a pads situados en la misma placa.

Se observa como se ha dejado espacio para ambas cosas, unos pads que permitan la conexión directa de los cables y la huella de los conectores. Esto hace que el requisito de espacio libre se cumpla sin problema.

3.4.1 Design Rule Check

Para considerar que ambos diseños están listos para fabricar, debemos comprobar que se cumplen las reglas de diseño. Estas reglas marcan las limitaciones de fabricación e incumplirlas puede llevar a resultados indeseados.

Cabe destacar que estas reglas son modificables, pero deben de modificarse sin interferir en las limitaciones del fabricante, con mucho cuidado y conocimiento.

Altium Designer

Design Rule Verification Report

Date:	20/06/2018		
Time:	12:57:34	**Warnings:**	0
Elapsed Time:	00:00:01	**Rule Violations:**	0
Filename:	C:\Users\Ramón Soldado\Dropbox\ALTER\Altium\V1_vibracion\Adapter_Board\PCBvib.PcbDoc		

Figura 3-20. Resultado del Design Rule Check en placa de adaptación

Una vez obtenemos un informe del DRC que indica que se cumplen todas las reglas de diseño, se puede considerar que las placas están terminadas y generar los archivos de salida para su fabricación.

3.5. Fabricación y montaje

El último paso una vez finalizadas todas las fases de diseño consiste en llevar a cabo la fabricación y montaje de ambas placas. En este caso es necesario realizar pedidos a tiendas especializadas de impresión PCB, componentes, etc.

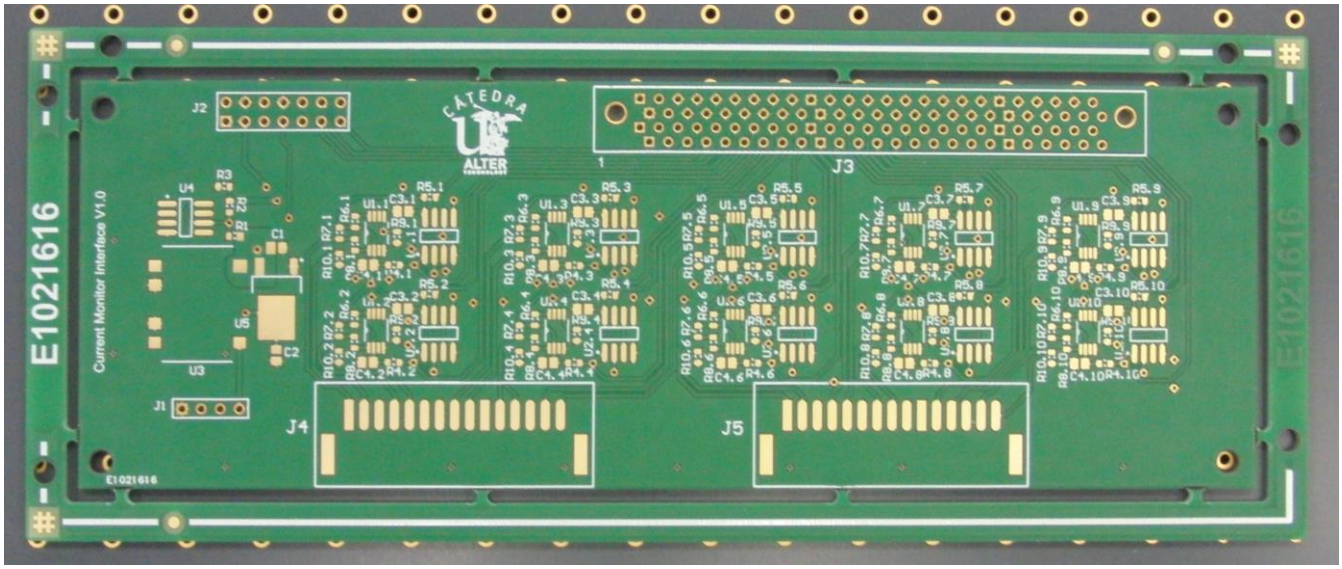


Figura 3-21. PCB de adaptación fabricada

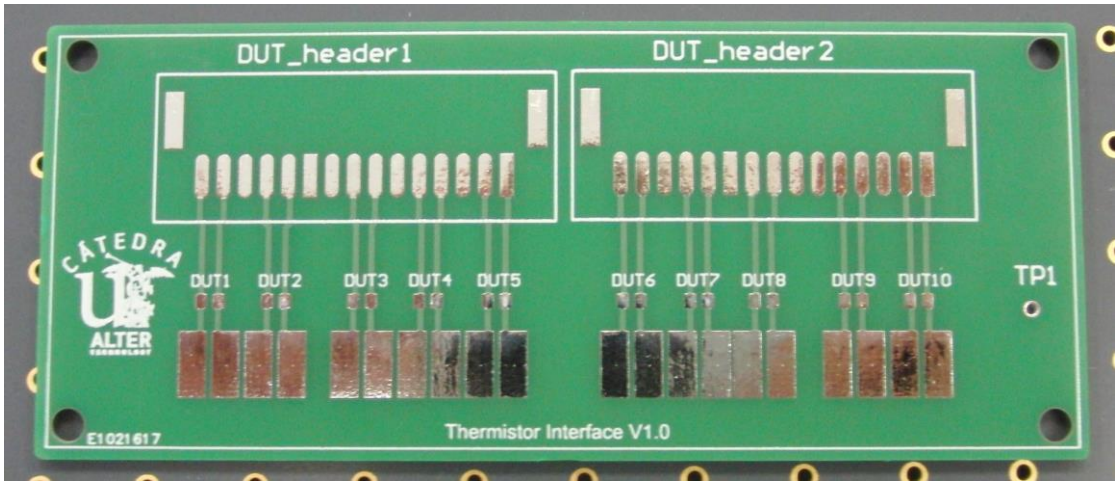


Figura 3-22. PCB de DUT fabricada

Una vez realizados y recibidos los correspondientes pedidos, debe de realizarse el montaje de los componentes en las PCB. Se va a describir el proceso realizado para el montaje de la placa de adaptación.

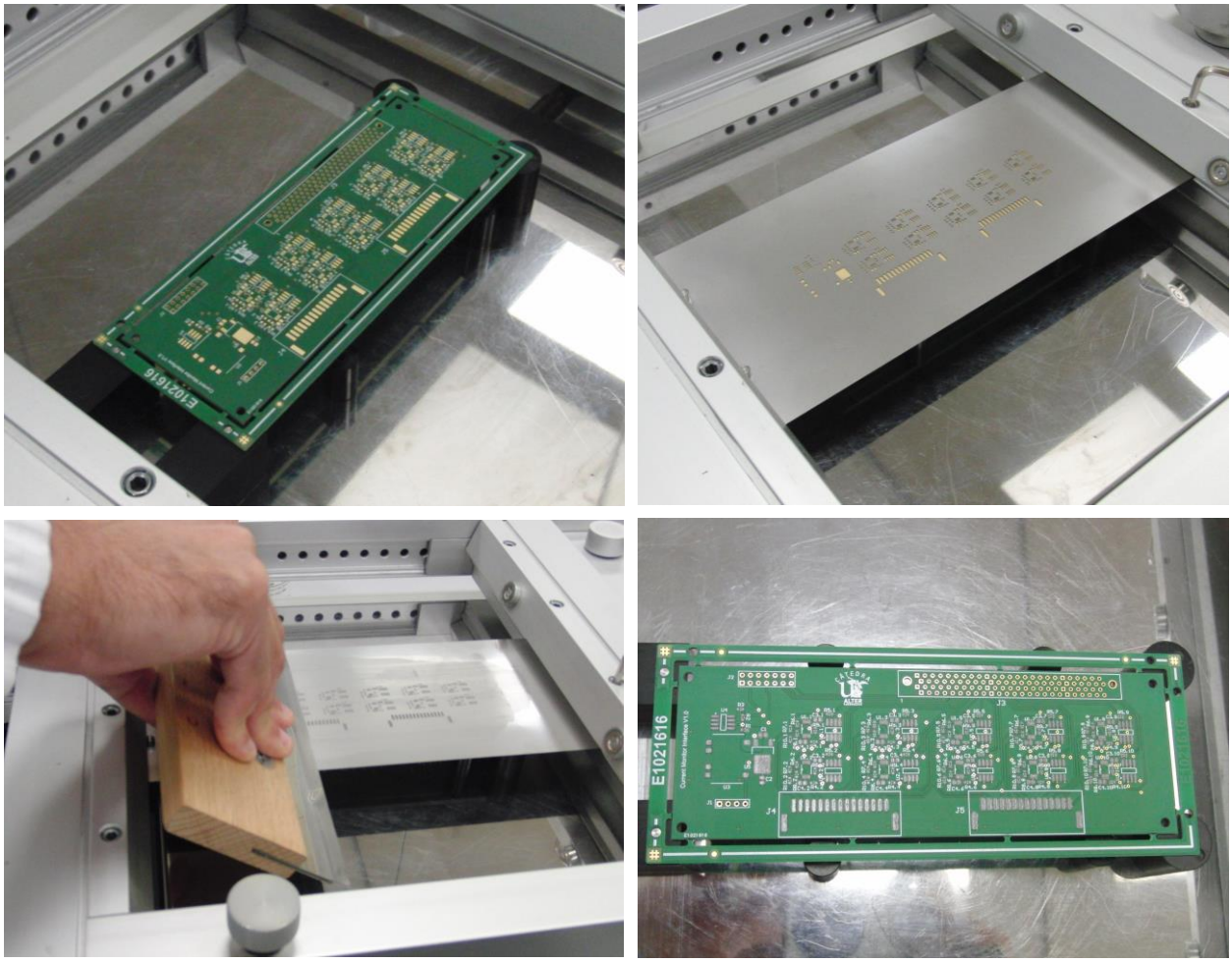


Figura 3-23. Proceso de colocación de la pasta de soldadura

El primer paso que se realiza es la colocación de la pasta de soldadura. Se coloca la PCB en un soporte especial para esta tarea, colocando encima de ella el *stencil* recibido en el mismo pedido, haciendo que coincida perfectamente con los pads donde va a colocarse la pasta.

Una vez bien fijado y colocado correctamente se coloca un poco de pasta y se extiende por el *stencil* con una especie de espátula con el ancho necesario para que ocupe toda la placa. Deben darse varias pasadas para asegurar que todos los pads tienen su capa de pasta.

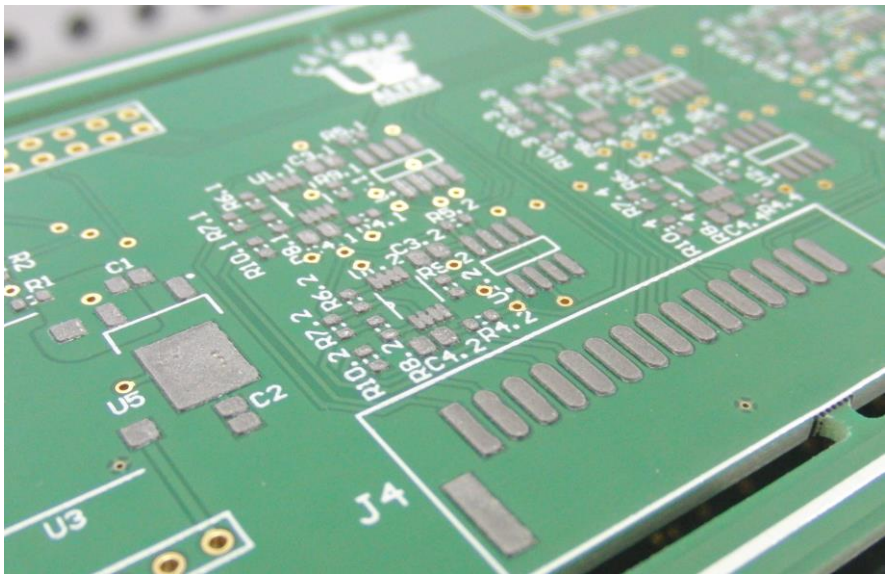


Figura 3-24. Detalle de la pasta de soldadura colocada en los pads

Se saca la placa del soporte, se limpian los posibles restos de pasta en las diferentes herramientas utilizadas y ya se puede comenzar la colocación de componentes.

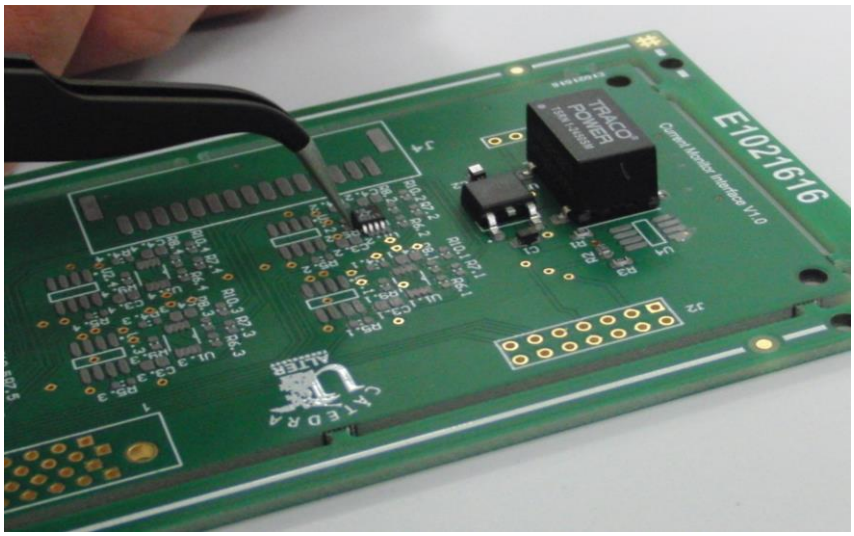


Figura 3-25. Colocación de los componentes en sus respectivas huellas

La colocación de los componentes SMD precisa de una atención y cuidado especial, ya que los componentes son generalmente muy pequeños y pueden confundirse elementos muy similares como resistencias de diferentes valores. La capa de serigrafía antes diseñada sirve de guía para la colocación de componentes y de indicación de los pines de ciertos encapsulados.

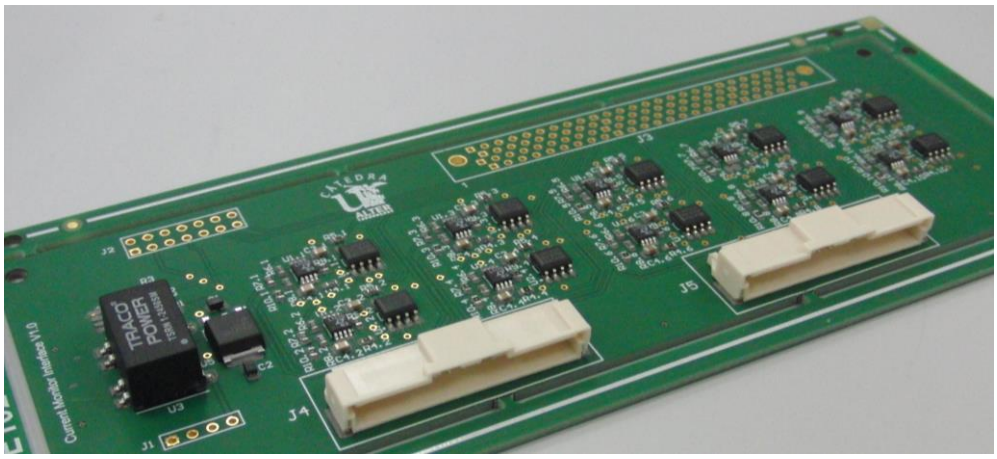


Figura 3-26. PCB con todos los componentes SMD colocados

Una vez colocados todos los componentes, se revisa que no falte nada y todos los componentes esten en su sitio, ajustando los que puedan estar algo desplazados de su lugar.

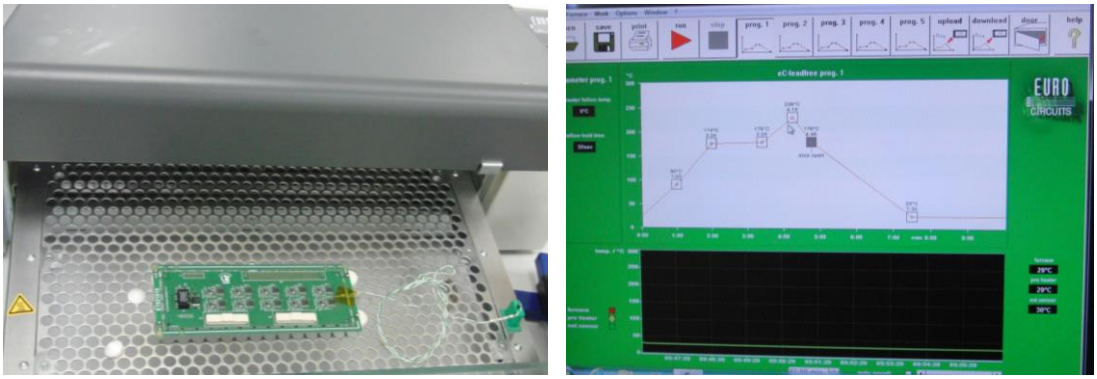


Figura 3-27. Colocación y programación del horno

Una vez finalizada la colocación de los componentes SMD se aplica un programa de calentamiento en un horno específico para fundir la pasta y que los componentes queden totalmente soldados.

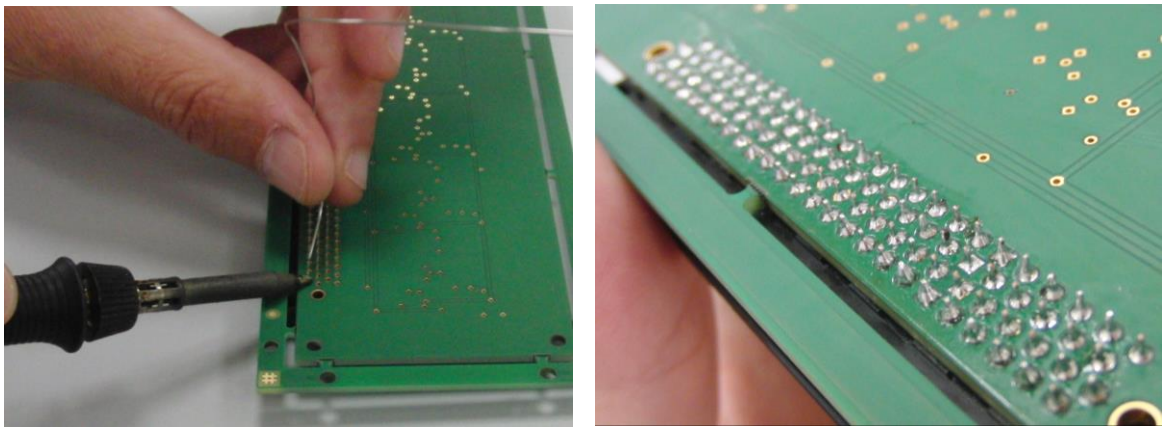


Figura 3-28. Soldadura de componentes threshold

Una vez la placa ha salido del horno con los componentes SMD soldados, es necesario llevar a cabo la soldadura de los compentes *threshold* o de orificio pasante. En este caso se trata de los conectores y la soldadura se hace por la capa contraria a la que se colocan los componentes SMD.
Para finalizar, se realizan limpiezas de imperfecciones en la soldadura para evitar cortocircuitos y se verifica que todos los pads esten conectados correctamente.

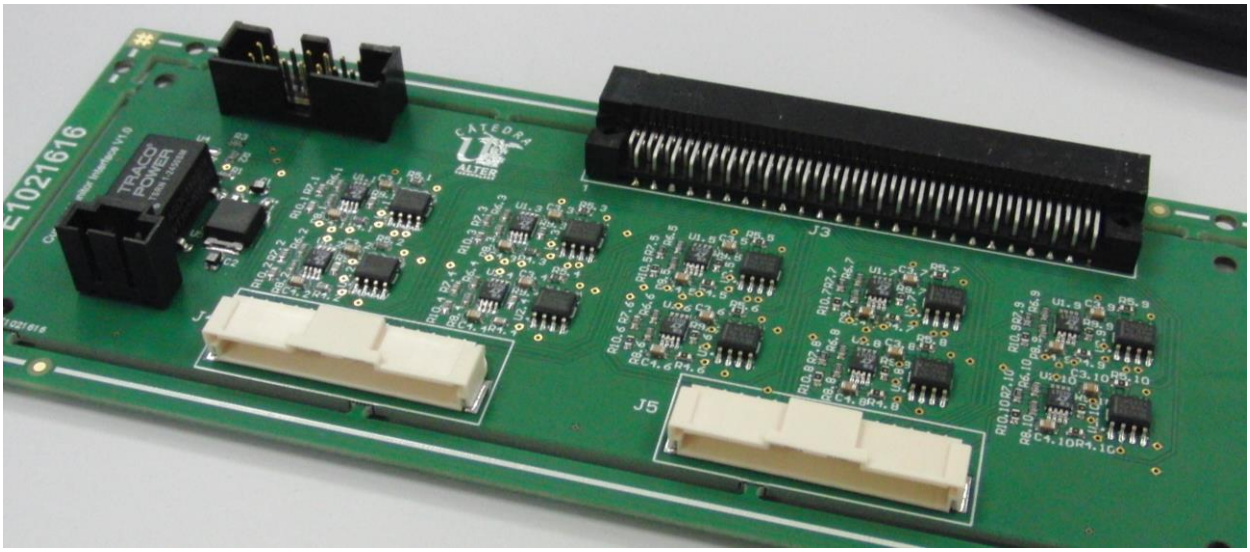


Figura 3-29. Resultado final de la PCB de adaptación montada

3.6. Verificación

Para verificar que tanto el diseño como la fabricación ha sido realizada correctamente, se realizan medidas sobre la placa alimentada, esperando obtener los valores obtenidos en las simulaciones y cálculos teóricos.

Es necesario alimentar la placa con 7 V tal y como se prevee en el diseño, para que se generen las alimentaciones necesarias para el circuito. Con un multímetro se verifica que las diferentes tensiones de alimentación y de referencia se generan correctamente.

Para verificar el correcto funcionamiento de los módulos que detectan los errores, se han colocado unos pines entre las salidas del DUT de dos de los circuitos. De esta forma, es posible comprobar si la salida del comparador refleja la detección de un circuito abierto, un cortocircuito o incluso funcionamiento normal, usando una placa de pruebas y diversos elementos como resistencias o jumpers.

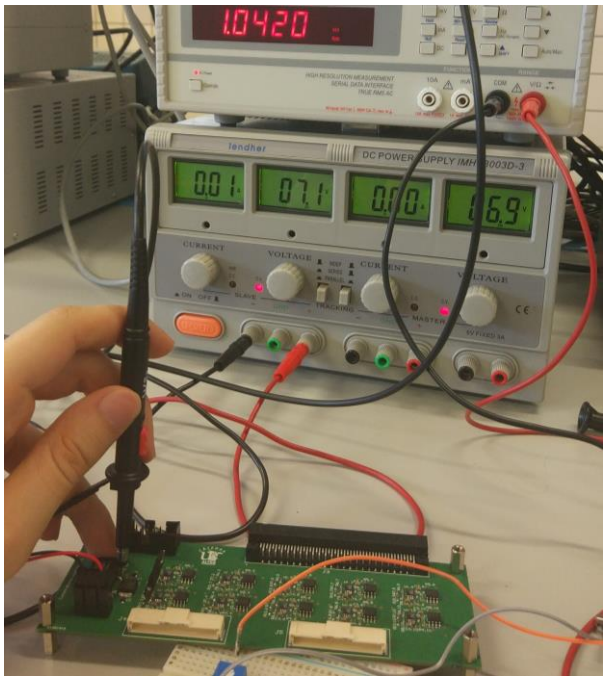


Figura 3-30. Montaje de verificación de PCB

4 DISEÑO HARDWARE DE FPGA

La tercera parte del sistema está formado por una placa de FPGA comercial, en este caso se ha elegido la *Spartan 3E Starter Board*. Este modelo incluye muchos conectores, una cantidad suficiente de pines de entrada/salida y varios periféricos que podrían ser de utilidad. [5]

A diferencia del diseño de PCB, la FPGA proporciona mayores herramientas de simulación, testeo y una importante posibilidad de corregir los errores cometidos sin tener que repetir la fabricación.

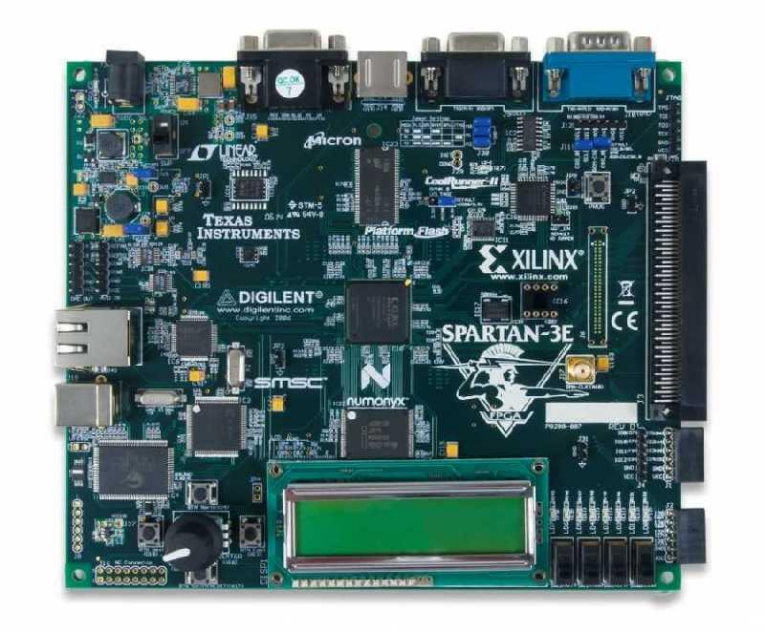


Figura 4-1. Imagen de la Spartan 3E Starter Board

El flujo de diseño es parecido al realizado anteriormente, ya que primero se busca encontrar una estructura que fije las funciones a realizar y luego se explicará el diseño realizado en código VHDL que implementa esas funcionalidades dentro de la FPGA.

4.1. Diseño funcional

Debido a la mayor flexibilidad y potencia de procesamiento, la FPGA se encarga de adaptarse a las señales que le lleguen de la placa de adaptación, procesarlas y conseguir implementar la funcionalidad necesaria. En este caso hay que tener en cuenta que se trabaja con valores digitales, por lo que la funcionalidad se debe abarcar consecuentemente a esto.

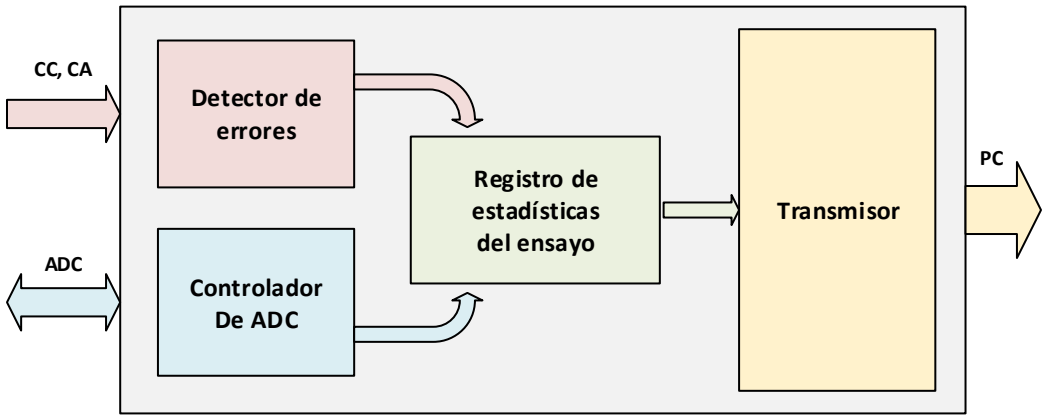


Figura 4-2. Diagrama funcional de la FPGA

Las señales recibidas de la placa de adaptación son señales de detección de errores y de control del ADC, por tanto, las principales funciones de la FPGA serán tratar estas señales. Una vez obtenidos los errores y los datos del ADC, se almacenan en unos registros internos de la FPGA para ese ensayo. Esta información se deberá enviar periódicamente al PC que es el siguiente y ultimo elemento del sistema, por lo que hará falta algún tipo de transmisor.

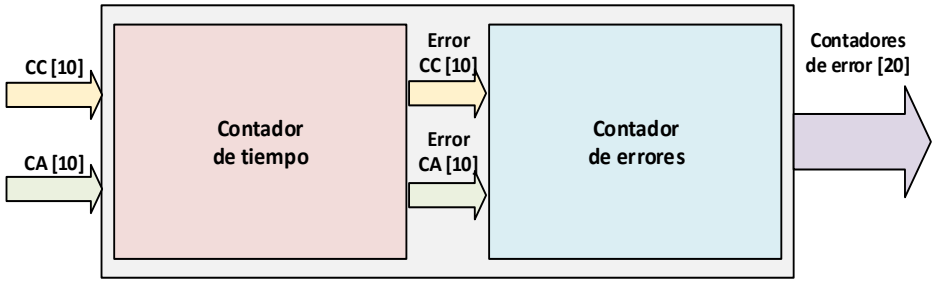


Figura 4-3. Diagrama funcional del detector de errores

Como la placa de adaptación ya realiza la detección de cortocircuitos y circuitos abiertos, la única condición necesaria para que se considere error es, según los estándares, que supere los 100 µs.

Para ello, se colocan contadores de tiempo que cuando reciban una señal de la placa de adaptación por más de este tiempo, generen una señal que indica que se ha detectado un error. A continuación, unos contadores de errores se incrementarán cada vez que los contadores de tiempo generen dicha señal.

La información de los contadores de errores debe ser accesible por el resto de bloques si fuese necesario, ya que es la información que se envía al PC.

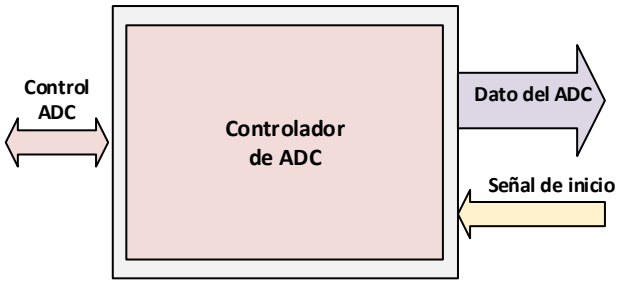


Figura 4-4. Diagrama funcional del controlador del ADC

El ADC que transmite información sobre uno de los DUT necesita comunicarse mediante un protocolo definido en su hoja de carasterísticas. El controlador deberá implementar este protocolo asegurando que se cumplen los tiempos establecidos y se recoge la información que devuelve. Las conversiones del ADC comenzarán cuando se le indique externamente al controlador mediante una señal de inicio, ya que no es necesario estar constantemente solicitando conversiones al ADC.

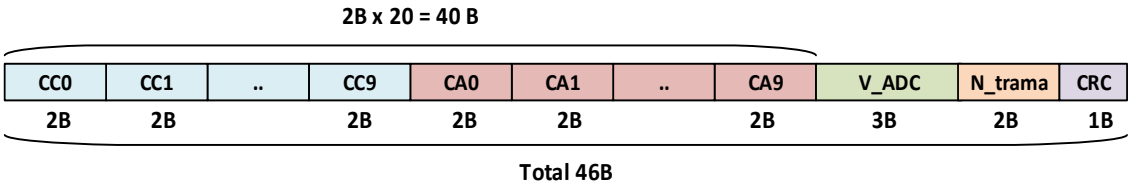


Figura 4-5. Diagrama de trama usada para enviar información al PC

Para poder definir el resto de los bloques de la FPGA es necesario fijar cual es la información que se va a enviar. Los valores almacenados en la FPGA son enviados durante el transcurso del ensayo al PC, esto significa que una trama con los datos necesarios es enviada periódicamente.

Los datos que contiene la trama son:

- Contadores de errores (**CC** y **CA),** que con un tamaño cada uno de 2 bytes permiten contar hasta 65536 errores ocurridos de un mismo tipo para un mismo dispositivo.
- Valor del ADC (**V_ADC)**, que contiene un numero de 18 bits (resolución del ADC) con el valor del voltaje de salida del DUT 0 en el instante que se forma la trama. Para que el número de bytes sea redondo, estos 18 bits se rellenan con seis ceros como bits más significativos, de forma que no modifiquen el valor.
- Contador de número de tramas enviadas (**N_trama**) de 2 bytes, que cuenta las tramas que se han enviado desde que comienza el ensayo, con hasta 65536 valores.
- Código de redundancia cíclica (**CRC**) de un byte, generado con los bits de la misma trama. Este valor sirve para que el receptor verifique con cierta confianza que la trama se ha recibido correctamente.

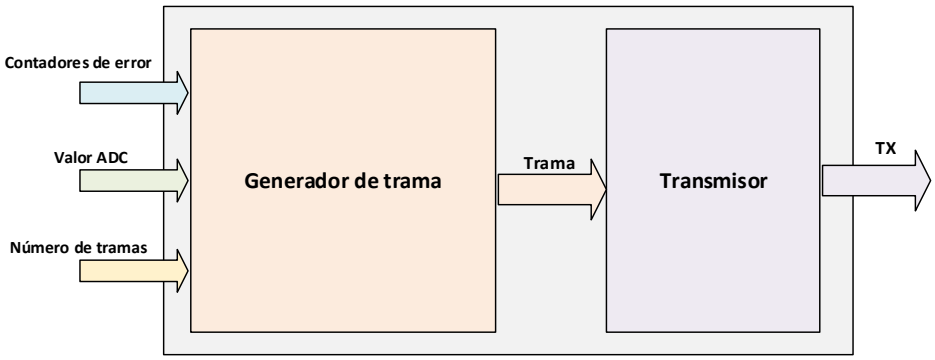


Figura 4-6. Diagrama funcional del generador de trama y transmisor

El bloque **generador de trama** se encarga de leer los datos que se van a enviar, generar el CRC y organizar todo en una trama que se le envía al transmisor. El **transmisor** se encarga de implementar un protocolo de comunicación suficiente y válido para poder realizar la transmisión de datos al siguiente elemento del sistema.

Ambos sistemas dependerán el uno del otro, puesto que hasta que la trama no esté lista el transmisor no puede comenzar a funcionar. El tiempo entre tramas debe de ser suficientemente grande para que de tiempo a realizar todas estas acciones y la transmisión se complete.

4.2. Diseño bloque a bloque

Debido aque estamos ante lógica programable, elementos funcionales descritos anteriormente son muy fáciles de generar e implementar en la FPGA, a un alto nivel. El programa completo se desarrolla en el entorno del fabricante *ISE 14.7*, que además ofrece herramientas de simulación, sintetización e implementación en la FPGA.

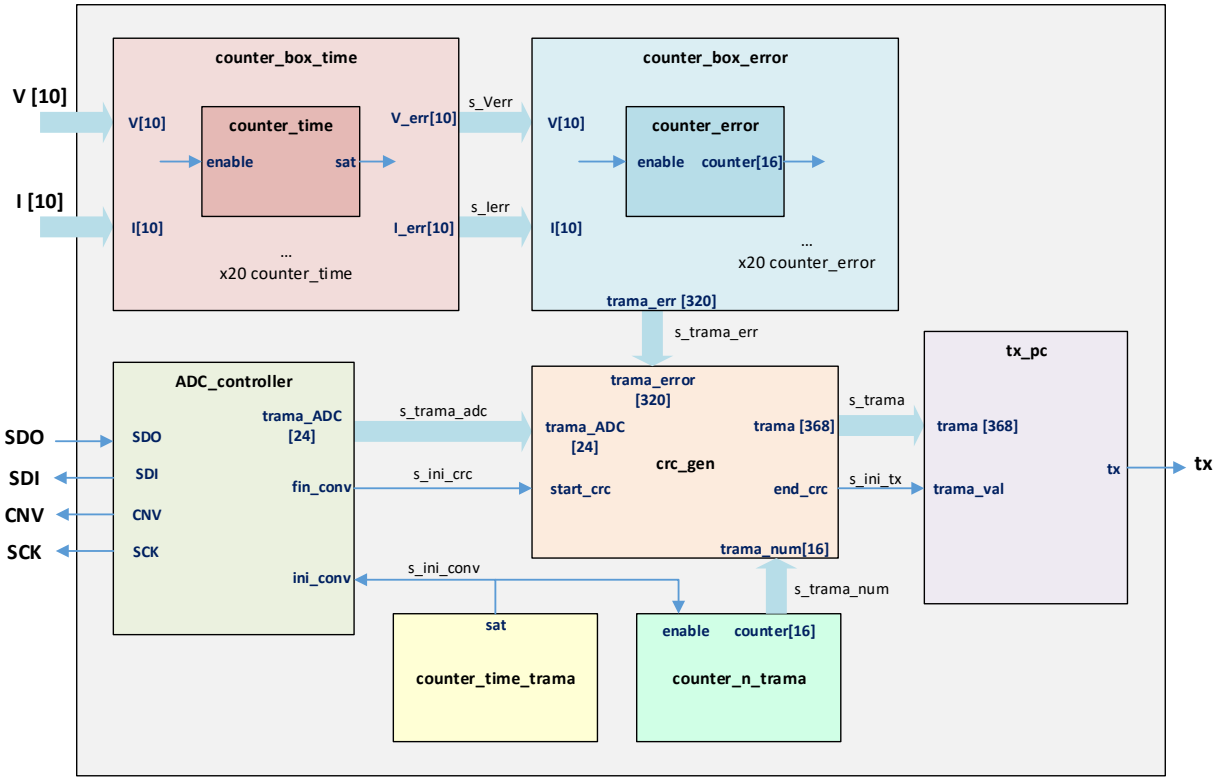


Figura 4-7. Diagrama de bloques completo del diseño FPGA

Además de todas las señales que se describen, existen en todos los bloques existen unas señales comunes **clk** y **reset** que funcionan como señal de reloj y como reset asíncrono respectivamente. Van a describirse uno por uno cada uno de los bloques creados, explicando su funcionamiento detalladamente.

4.2.1 counter_box_time

Este primer bloque consiste en un contenedor de contadores de tiempo. Debido a que se debe monitorizar veinte entradas de errores, se ubican veinte contadores de tiempo en este bloque.

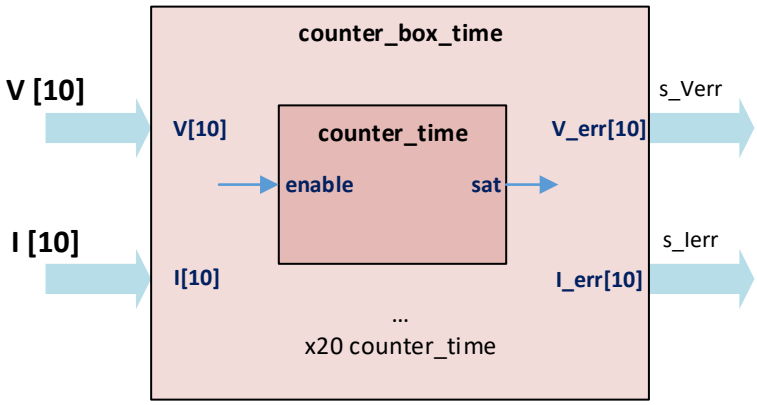


Figura 4-8. Diagrama de bloques de counter_box_time

En este caso se programa un solo contador (*counter_time)* y se instancia veinte veces en el contenedor (*counter_box_time)*. Los contadores se activan con las señales de error que provienen de la placa de adaptación. En este caso se realiza un cambio en la notación utilizada hasta ahora, ya que *V* es el vector de errores provocados por cortocircuito e *I* el propio para circuito abierto.

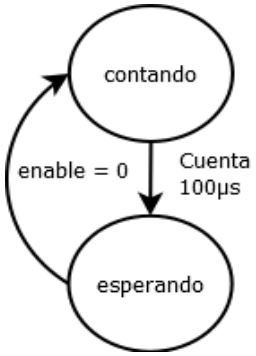


Figura 4-9. Diagrama de bolas de counter_time

El contador counter_time está formado por una simple máquina de estados. En el estado principal *contando* se mantiene tanto cuando esté esperando que la señal de entrada (*enable*) se active, como cuando está activa y empieza a contar hasta 100 µs, como indica la normativa del ensayo de vibración.

Una vez alcance este valor, se genera un pulso por la salida (*sat*) para indicar al siguiente bloque que se ha producido un error que hay que registrar. Además, se pasa al estado *esperando* donde se espera que la señal se desactive para volver a *contando*. De esta forma, no se cuentan múltiples errores cuando son más largos del tiempo mínimo de error.

La placa de FPGA tiene un reloj con frecuencia de 50 MHz que utilizaremos para nuestra señal de reloj. Esta frecuencia nos proporciona un periodo de ciclo de reloj de 20 ns, por lo que los contadores de tiempo de los diferentes bloques deberán contar teniendo en cuenta esto. En este caso, se necesita contar hasta 5000 para que equivalga a 100 µs.

4.2.2 counter_box_error

La estructura de este bloque es muy similar a la del anterior, un contenedor de veinte bloques, que en este caso cuentan eventos puntuales en lugar de tiempo. Estos eventos puntuales son los errores de más de 100 µs cometidos.

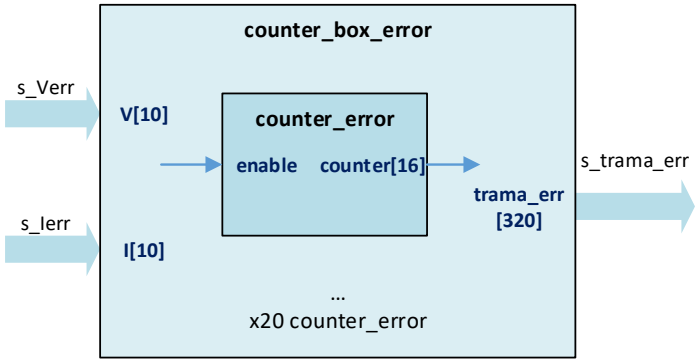


Figura 4-10. Diagrama de bloques de counter_box_error

De la misma forma que los anteriores, los contadores *counter_error* tienen una entrada *enable* que se asocia con una de las veinte señales de error que salen del bloque *counter_box_time,* que generan un pulso cada vez que se detecta un error de más del tiempo establecido. En este caso, la salida del bloque *counter_box_error* es una parte de la trama final de 320 bits, *trama_err.* Está formada por las salidas de los veinte contadores, que en este caso es el número de errores contados *counter* de 16 bits, por lo que se registran hasta 65536 errores de cada tipo para cada DUT.

Este contador no se resetea a menos que se fuerce un reset externo y en caso de llegar al número máximo del contador, se mantiene en este valor máximo.

4.2.3 counter_time_trama y counter_n_trama

Como el envío de tramas es periódico, precisa de un contador que cuente este periodo de tiempo y avise al resto de bloques de que deben actuar. Además, si se incluye información sobre el número de tramas, otro contador deberá contar cuantos de estos periodos han ocurrido. Estos dos bloques son necesarios para sincronizar el comportamiento de todos los bloques y siguen una estructura casi idéntica a la de los contadores vistos anteriormente.

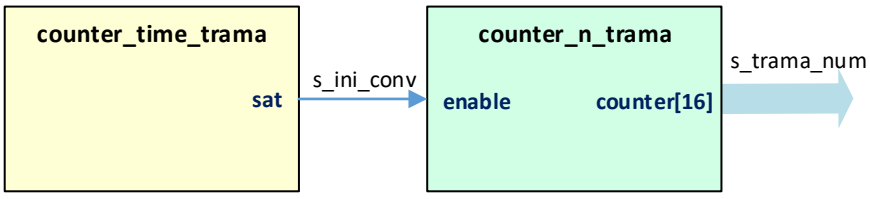


Figura 4-11. Diagrama de bloques de counter_time_trama y counter_n_trama

En este caso, el contador de tiempo *counter_time_trama* no tiene entrada que lo active, si no que cuenta constantemente. En previsión de que cada trama es de 46 bytes y se transmite a 9600 baudios, más el tiempo necesario para tomar el dato del ADC y generar el CRC, se considera que este periodo no debería ser menor de 0,1 segundo. Aun así, la funcionalidad no precisa del periodo más pequeño posible, por lo que se fija a **1 segundo** finalmente.

La salida de *counter_time_trama* indica el comienzo de una serie de procesos necesarios para el envío de trama y el primero de estos es el de obtener el dato del ADC. Además, el bloque *counter_n_trama* utiliza esta señal para contar, siendo su salida *counter* la cuenta de 16 bits del número de tramas ocurridas total.

4.2.4 ADC_controller

Este bloque se encarga de controlar el ADC *AD7982* ubicado en una placa externa, que tiene una resolución de 18 bits y cuenta con la posibilidad de realizar una comunicación SPI. La comunicación debe ser implementada tal y como indica la hoja de carasterísticas que proporciona el fabricante.

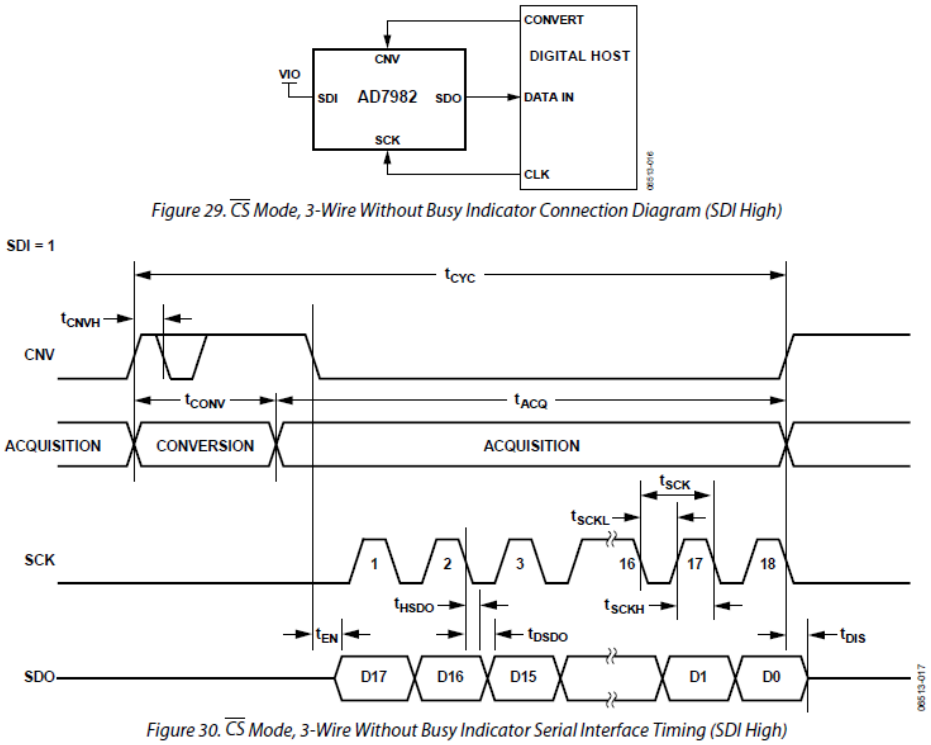


Figura 4-12. Interfaz de comunicación del AD7982 en modo 3-Wire

De entre todas las formas posibles de comunicación con el ADC, se elige la que el fabricante llama *CS Mode, 3-Wire without busy indicator*. Este modo es el recomendado para realizar comunicación con un solo dispositivo como es este caso, ya que es el caso más sencillo.

El proceso tiene dos fases diferenciadas: la conversión, donde el ADC es avisado de que se requiere un dato y lo obtiene y la adquisición donde se realiza la obtención del dato por parte de la FPGA en este caso.

Es importante respetar el timing establecido en la hoja de carasterísticas para asegurar que el ADC actúa tal y como se indica en la figura proporcionada.

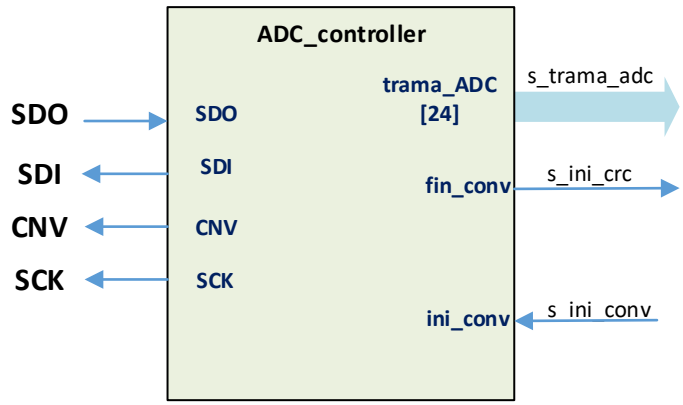


Figura 4-13. Diagrama de bloques de ADC_controller

Por lo tanto, el bloque ADC_controller debe tener las salidas y entradas *SDO, SDI, CNV y SCK* para poder realizar la comunicación. Como este proceso entra dentro de la cadena de procesos ya mencionada que se realiza cuando se cumple el periodo entre tramas marcado por counter_time_trama, necesita una señal *ini_conv* que marque el comienzo del proceso y una señal de salida *fin_conv* que indique al siguiente bloque que el dato del ADC contenido en *trama_ADC* ya está disponible y se puede continuar con el siguiente proceso.

Aunque el dato enviado por el ADC es de 18 bits, se devuelve una trama de 24 bits para que el número de bytes de la trama final sea redondo y facilite la transmisión. Los 6 bits que se añaden son ceros a la izquierda del valor recibido, por lo que no afecta al valor en ningún momento.

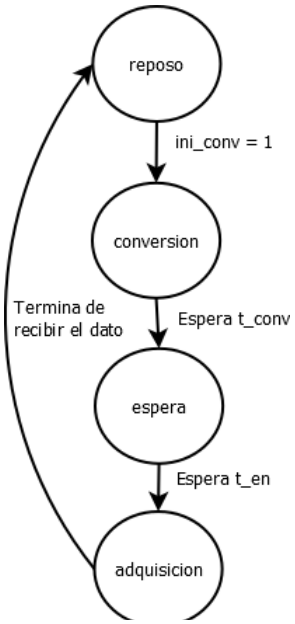


Figura 4-14. Diagrama de bolas de ADC_controller

Las señales de control *CNV* y *SCK* se mantienen a nivel bajo mientras no sean necesarias, pero la señal *SDI* se mantiene durante todo el tiempo a nivel alto, de forma que se configure la forma de comunicación 3-Wire.

Respecto al funcionamiento interno, el bloque está controlado por una máquina de estados cuyos estados son equivalentes a las diferentes fases por las que pasa el ADC.

Comienza en estado de *reposo* dondo unicamente se espera que la señal de inicio *ini_conv* se active para pasar al estado *conversión* donde se comienza el proceso. En *conversión*, se activa la señal de conversion *CNV* que supone una petición al ADC para obtener el dato. Este estado se mantiene durante 780 ns, algo más del tiempo máximo de conversion indicado por el fabricante, por dejar cierto margen de seguridad.

Una vez transcurrido el tiempo de conversión, pasamos al estado *espera*, bajamos CNV para indicar que queremos obtener el dato y nos mantenemos un ciclo de reloj (20 ns) en este estado para esperar el tiempo de enable indicado en la especificación temporal.

Finalmente llegamos al estado *adquisición* que se encarga de recibir el dato. Para ello, tal y como se especifica, se genera una señal de reloj en *SCK* de 18 ciclos de duración. El ADC especifica que los bits se envian por *SDO* en el flanco de bajada de la señal *SCK*, por lo que una vez bajamos la señal podremos leer el dato.

Se almacenan los datos en los bits menos significativos de *trama_ADC* dejando los más significativos para los seis ceros indicados anteriormente.

TIMING SPECIFICATIONS

VDD = 2.37 V to 2.63 V, VIO = 2.3 V to 5.5 V, T_A = −40°C to +85°C, unless otherwise noted.[1]

Table 4.

Parameter	Symbol	Min	Typ	Max	Unit
CONVERSION AND ACQUISTION TIMES					
Conversion Time: CNV Rising Edge to Data Available	t_{CONV}	500		710	ns
Acquisition Time	t_{ACQ}	290			ns
Time Between Conversions	t_{CYC}	1000			ns
CNV PULSE WIDTH ($\overline{CS}$ MODE)	t_{CNVH}	10			ns
SCK					
SCK Period ($\overline{CS}$ Mode)	t_{SCK}				
VIO Above 4.5 V		10.5			ns
VIO Above 3 V		12			ns
VIO Above 2.7 V		13			ns
VIO Above 2.3 V		15			ns
$\overline{CS}$ MODE					
CNV or SDI Low to SDO D17 MSB Valid	t_{EN}				
VIO Above 3 V				10	ns
VIO Above 2.3 V				15	ns

Figura 4-15. Tiempos recogidos en la hoja de carasterísticas del AD7982

Es importante que además de seguir el procedimiento, se cumplan rigurosamente los tiempos marcados por el ADC o no conseguiremos el funcionamiento esperado. En este caso, nos interesan principalmente tres valores:

- **t_{cnv}:** se trata del tiempo de conversión máximo. Debido al funcionamiento del modo 3-Wire debemos mantenerlo a nivel alto por más del tiempo máximo de conversión, para asegurar que se produzca. Este tiempo máximo se especifica en *710 ns* y en el bloque se ha implementado el contador para esperar *780 ns,* por lo que cumplirá correctamente.
- **t_{sck}:** especifica el periodo mínimo de los ciclos de reloj producidos en *SCK*. En nuestro caso, con una alimentación V_{IO} de 3,3 voltios, debería ser superior a *12 ns.* Nuestro ciclo de *SCK* se produce en dos ciclos de la FPGA, que equivale a *40 ns*, más que suficiente.
- **t_{en}:** es un tiempo que tarda el ADC desde que se baja la señal de conversión *CNV* hasta que está disponible para empezar la transmisión de datos. Se especifican *10 ns* y en nuestro caso esperamos un ciclo de reloj, es decir *20 ns*.

4.2.5 CRC_gen

Ya se han generado las tres tramas necesarias, pero antes de transmitirlas, es necesario crear un método para que el receptor verifique que se ha recibido la trama sin errores. En este caso hemos optado por implementar un CRC de *8-bit CCITT.*

Una vez el receptor reciba la trama, puede realizar el mismo procedimiento que este bloque y verificar si el CRC coincide, para aceptar o deshechar la trama. Al ser de solo 8 bits y la trama tan larga, no es fiable completamente, pero aun así es muy difícil que la trama se reciba erronea y el CRC coincida.

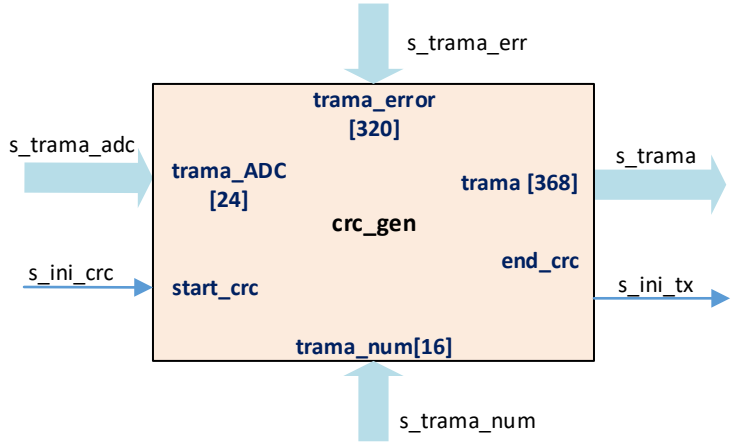


Figura 4-16. Diagrama de bloques de crc_gen

Al bloque le llegan todas las partes de trama vistas hasta ahora, y se le indica cuando comenzar a generar el CRC con la señal *start_crc*. Esta señal le llega una vez el dato del ADC este leido y actualizado en *trama_ADC,* ya que los otros trozos de trama son la salida de contadores y no necesitan actualización.

Una vez generada la trama con su CRC, se forma *trama* de *368 bits,* es decir, *46 bytes* y se activa la correspondiente señal para avisar al transmisor de que la trama está preparada y lista para enviarse.

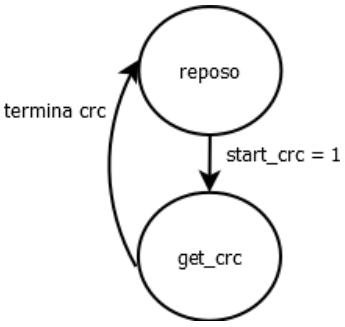


Figura 4-17. Diagrama de bolas de crc_gen

Este bloque se basa en una sencilla máquina de estados con un estado de *reposo* donde espera la señal *start_crc* para pasar a *get_crc* y comenzar el proceso.

En el estado *get_crc* se produce un bucle que mediante un índice recorre los 360 bytes de la trama, formada a partir de las señales de entrada. Durante cada iteración se calcula el CRC, realizando una serie de operaciones a nivel de bit.

Cuando se llega al último bit de la trama se añaden 7 ceros para terminar el desplazamiento de los bits de la trama. Una vez se termina, se obtiene un CRC de 8 bits que se incluye como último byte de la trama y se avisa al siguiente bloque activando la señal *end_crc.*

```
n_crc(0) <= trama_aux(index) xor crc(7);
n_crc(1) <= crc(0) xor crc(7);
n_crc(2) <= crc(1) xor crc(7);
n_crc(7 downto 3) <= crc(6 downto 2);
```

Figura 4-18. Código que implementa el CRC

Existen muchos tipos de CRC, según su polinomio generador. En este caso se ha usado un *CRC de 8 bits CCITT*, cuyo polinomio generador es x^8+x^2+x+1. Mediante un desarrollo matemático se llega a la implementación con puertas XOR y desplazamiento de bits.

4.2.6 TX_pc

Como último bloque tenemos el transmisor que se encarga de mandar la trama al PC mediante determinado protocolo de comunicaciones. En este caso, aprovechando que la *Spartan 3E Starter Board* posee puertos [5] y es sencillo de implementar, se va a realizar una comunicación serie asíncrona por *RS-232* [6].

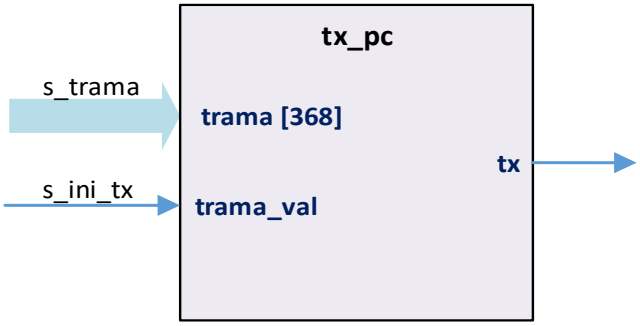


Figura 4-19. Diagrama de bloques de tx_pc

Las entradas del bloque son la *trama* de 368 bits y la señal *trama_val* que indica que puede empezar la transmisión de esta. A la salida solo tendremos una señal de bit, ya que solo se realiza la función de transmitir del protocolo porque no es necesario recibir ningún tipo de información.

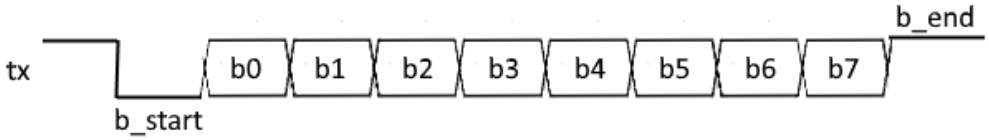


Figura 4-20. Estructura de la tranmisión RS-232

El protocolo implementado funciona usando una única señal *tx* y envía los datos de byte en byte. En reposo, la señal *tx* estará a nivel alto, y deberá bajar en el momento que comience. La velocidad de transmisión es 9600 Baudios, lo que significa que cada símbolo ocupara 1/9600 segundos activo.

Una vez lanzado el bit de comienzo que no contiene información nignuna, se envian los 8 bits que conforman el byte desde el menos significativo al más significativo. Al terminar, se debe enviar el bit de finalización, que consiste en dejar el mismo tiempo la señal a nivel alto antes de seguir transmitiendo.

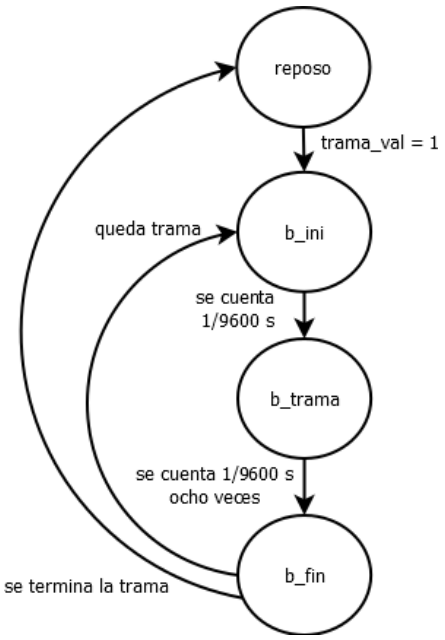


Figura 4-21. Diagrama de bolas de tx_pc

El protocolo se implementa mediante una máquina de estados que controla los bits que deben enviarse. Comienza en el estado *reposo* a la espera de que la señal de entrada *trama_val* se active indicando que puede comenzar la transmisión.

La máquina de estados cuenta con dos contadores: uno para contar el número de bytes enviados y otro que cuenta los bits enviados dentro de un mismo byte.

En el estado *b_ini* se realiza la transmisión del bit de inicio, que mantiene la salida *tx* a nivel bajo durante 1/9600 s, ya que como se indicó anteriormente, la velocidad de transmisión elegida es 9600 Baudios.

Una vez transcurrido ese tiempo se pasa al estado *b_trama* donde se comienza la transmisión de los bits de información. Usando ambos contadores (byte y bit) como índice, se accede al dato necesario de la *trama* de entrada y se coloca en la salida *tx*, manteniendolo durante el mismo tiempo.

Una vez el contador de bit cuenta 8 bits enviados, se resetea y añade una unidad al contador de byte, pasando al estado *b_fin*, donde se mantiene la salida a nivel alto, indicando que la transmisión de byte ha terminado. Esto es de utilidad en transmisiones asíncronas ya que como los relojes de transmisor y receptor acabarán provocando error, al volver a transmitir un bit de inicio con un flanco de bajada, volverán a sincronizarse. Por ese motivo se recomienda hacer transmisiones cortas, en este caso de byte.

Una vez terminado el tiempo del bit final, se comprueba si se han enviado los 46 bytes que conforman una trama, pasando a *reposo* si se han enviado y a *b_ini* para continuar la transmisión si no.

4.3. Simulación

El entorno de Xilinx cuenta con una herramienta de simulación para el código VHDL llamada *iSim*. Para realizar la simulación, se debe crear un fichero de pruebas llamado *TestBench* donde se programan estímulos sobre nuestro sistema para observar su funcionamiento.

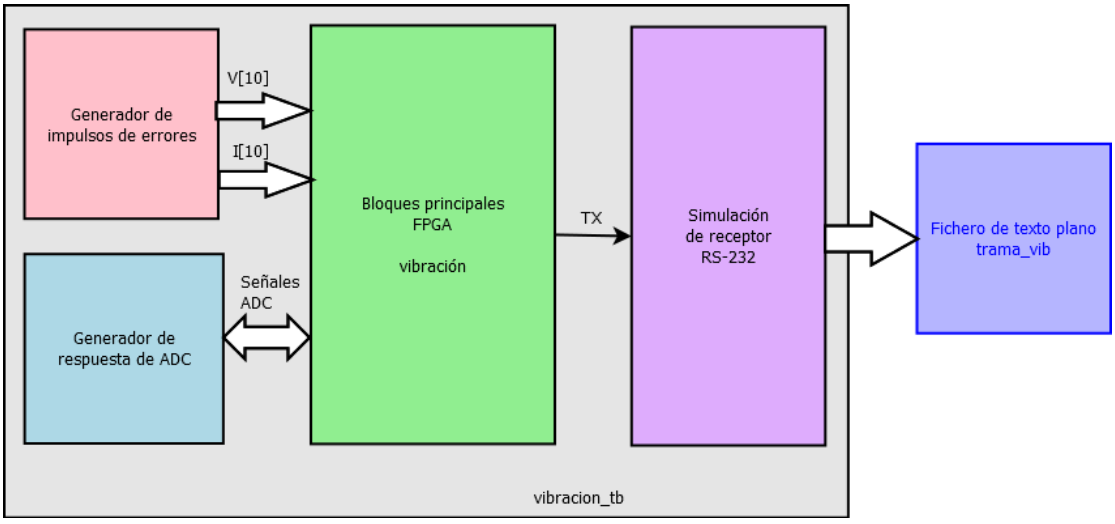


Figura 4-22. Diagrama de bloques del TestBench

Creado un fichero de simulación llamado *vibración_tb* que posee diferentes partes:

- **Estímulos de errores:** existe un proceso que genera diferentes estímulos a las entradas de los errores, por lo que se simulan errores en un DUT a lo largo del tiempo. Cada error equivale a una señal a nivel alto, que volverá a bajarse después de un tiempo, por lo que se usarán duraciones menores y mayores que 100 µs para verificar que solo cuenta como errores los que duran más.
- **Estímulos de ADC:** otro proceso de estímulos se encargará de simular el comportamiento del ADC conforme a lo establecido en su hoja de carasterísticas. Para cada solicitud de dato, el proceso proporcionará un valor para comprobar si el bloque controlador del ADC lo maneja correctamente.
- **Recepción de datos:** ya que como las tramas y señales son muy complejas para visualizarlas directamente en la interfaz del simulador, se crea un proceso que simula el funcionamiento de un receptor serie. Usando la librería *TXTIO* se genera un fichero de texto plano llamado *vib_trama* que almacena todos los bytes que se envian respetando los bits más significativos, como haría cualquier receptor. Este fichero nos permite ver de una forma más visual y directa si los datos que se tranmiten se forman de una manera correcta.

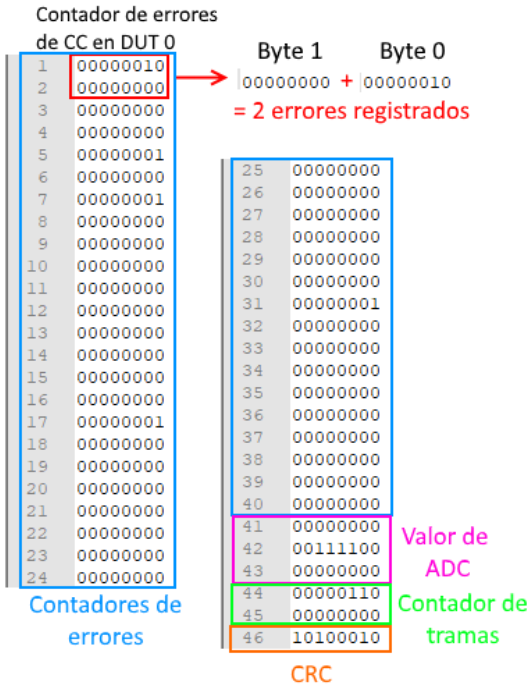


Figura 4-23. Fragmento de vib_trama con una trama simulada

Ya que los diferentes elementos dentro de la trama poseen en general más de un byte, observamos que el orden de estos está invertido, debido a la forma de recorrer la trama al transmitir. Esto es una característica que se debe tener en cuenta a la hora de diseñar el software de recepción.

4.4. Implementación

Una vez realizado todo el código y depurado, es necesario llevar a cabo la implementación en la FPGA. Para ello, es necesario generar el fichero implementable que se genera llevando a cabo varios procesos de sintetización, rutado y mapeo con herramientas proporcionadas por el propio entorno *ISE*.

vibracion Project Status (06/27/2018 - 09:06:37)			
Project File:	vibracion.xise	**Parser Errors:**	No Errors
Module Name:	vibracion	**Implementation State:**	Programming File Generated
Target Device:	xc3s500e-4fg320	**• Errors:**	No Errors
Product Version:	ISE 14.7	**• Warnings:**	No Warnings
Design Goal:	Balanced	**• Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	**• Timing Constraints:**	All Constraints Met
Environment:	System Settings	**• Final Timing Score:**	0 (Timing Report)

Device Utilization Summary [-]				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	735	9,312	7%	
Number of 4 input LUTs	1,587	9,312	17%	
Number of occupied Slices	1,189	4,656	25%	
Number of Slices containing only related logic	1,189	1,189	100%	
Number of Slices containing unrelated logic	0	1,189	0%	
Total Number of 4 input LUTs	2,195	9,312	23%	
Number used as logic	1,587			
Number used as a route-thru	608			
Number of bonded IOBs	27	232	11%	
Number of BUFGMUXs	1	24	4%	
Average Fanout of Non-Clock Nets	2.90			

Figura 4-24. Reporte del proyecto de vibración

Durante el proceso, se genera un reporte del proyecto en el que se indican los posibles errores, advertencias del sistema o la cantidad de recursos de la FPGA que se han utilizado para generar la funcionalidad programada.

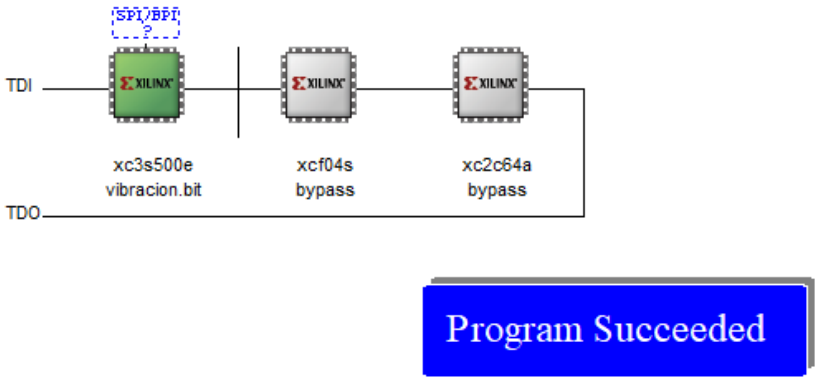


Figura 4-25. Captura de la implenetación completada

Una vez obtenido el fichero implementable, se utiliza una utilidad de Xilinx llamada *iMPACT* para detectar la placa FPGA conectada mediante un cable USB al PC y realizar la implementación del fichero.

5 DISEÑO SOFTWARE

Como último elemento del sistema tenemos un PC, para el cual se diseña un software específico que realice las funciones restantes del sistema. Se ha decidido realizar la programación de dicho software en Python 3, debido a la gran cantidad de módulos existentes y el conocimiento previo del lenguaje.

5.1. Diseño funcional

Este último elemento del sistema debe concluir las funcionalidades comenzadas en los elementos anteriores como la transmisión de datos y debe implementar otras como el visionado de contadores por pantalla para completar la función completa del sistema.

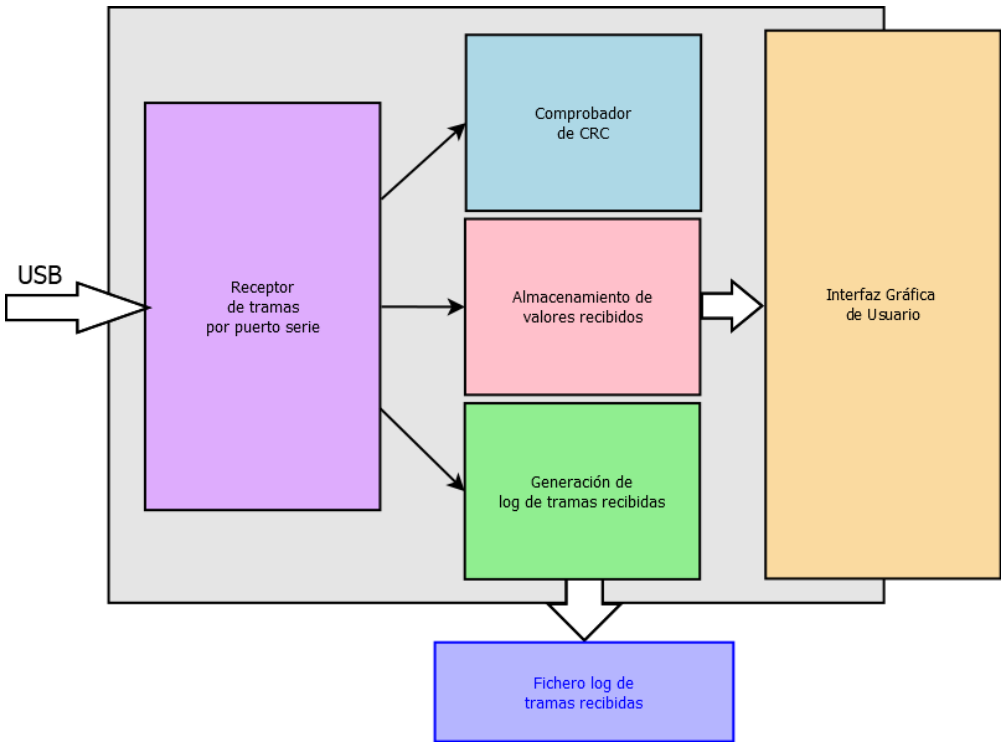


Figura 5-1. Diagrama funcional del software

Las funcionalidades que debe cumplir el software son:

- **Recibir del puerto serie:** debe recibir las tramas enviadas por la FPGA siguiendo protocolo RS-232 utilizado. En este caso, el puerto de entrada es USB.
- **Comprobar el CRC:** para saber si una trama se ha recibido correctamente, debe generarse el CRC igual que en la FPGA y compararse con el recibido en la trama.

- **Almacenar los valores recibidos:** todos los contadores y valores que contienen en la trama deben de almacenarse en variables accesibles por el programa para poder se utilizadas cómodamente.
- **Generar un log de las tramas recibidas:** conforme se reciben tramas deben almacenarse sus valores en un fichero externo de forma que los resultados del ensayo no se pierdan si se cierra el programa.
- **Mostrar los datos al usuario en pantalla:** para ello se debe crear una interfaz gráfica de usuario que muestra los datos almacenados que pueden ser de interés al usuario que realiza el ensayo.

Todos estos procesos deben de realizarse constantemente, ya que las tramas llegan periódicamente y se necesita que el usuario pueda ver el desarrollo del ensayo en vivo.

Todos los procesos y funcionalidades que se realicen en el programa deben estar gobernadas por la interacción del usuario con la interfaz gráfica, de forma que en la medida de lo posible, el usuario tenga el control de las estas acciones.

5.2. Programación

Se utiliza un entorno de desarrollo llamado *PyCharm* que permite un desarrollo más cómodo a la vez que implementa una consola para realizar pruebas de código.

Los diferentes procesos que realiza este software se vertebran en torno a la interfaz gráfica que proporciona las órdenes a través de la acción humana. Por esto, se va a describir la configuración de la FPGA en torno a la interfaz gráfica, explicando el funcionamiento y como realiza las funcionalidades.

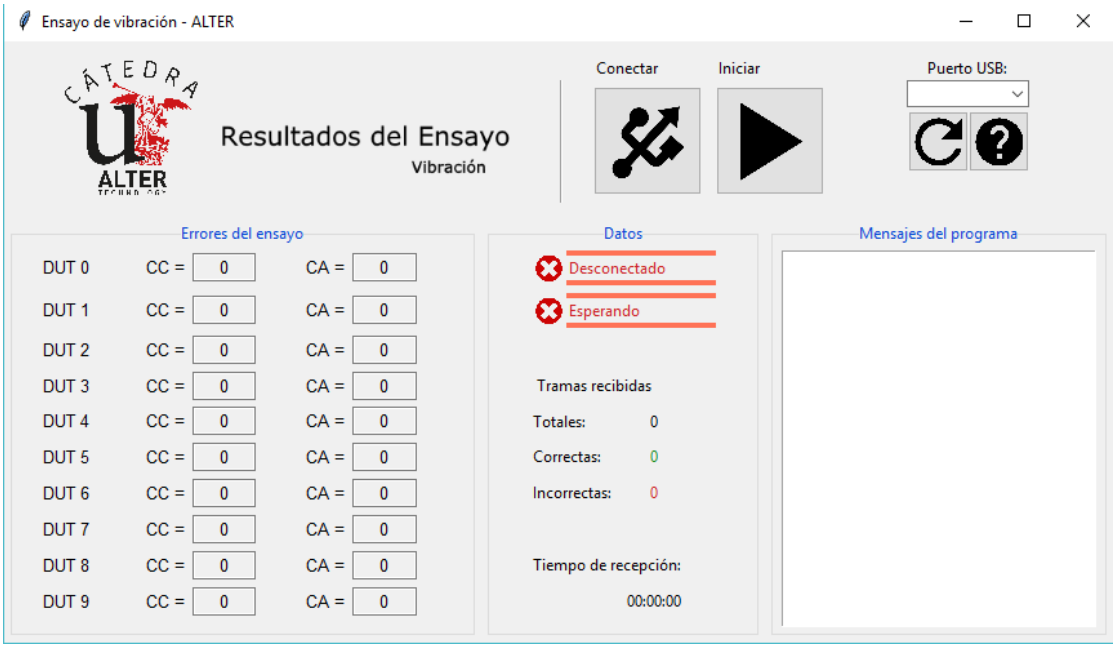


Figura 5-2. Interfaz gráfica del Software

La interfaz gráfica se ha realizado con un módulo de Python llamado *TKinter*. Este módulo ofrece diferentes elementos como contenedores, etiquetas, botones o pestañas desplegables para poder formar la interfaz a medida.

La distribución de los elementos en la interfaz gráfica se realiza en torno a cuatro áreas:

- **Barra superior:** contiene elementos estéticos como el logo y título del programa en su parte izquierda. En la parte derecha contiene los diferentes botones de control del programa: conexión, inicio, busqueda

de puertos y ayuda. Además, existe una barra desplegable donde aparecen los puertos detectados.

- **Contadores de errores:** el área principal de la interfaz contiene veinte cajas de texto que almacenan el valor de los contadores recibidos, organizados por número de DUT y por tipo de error (CC o CA).
- **Datos del ensayo:** esta área contiene unos indicadores de estado que muestran al usuario de una manera muy directa con texto y colores si el programa se encuentra conectado e iniciado. Además, muestra una estadística de las tramas recibidas desglosadas en erroneas y válidas. Por último, se incluye un contador que muestra el tiempo transcurrido desde que se comienza la recepción.
- **Mensajes del programa:** un cuadro de texto simula el comportamiento de una consola, mostrando mensajes importantes al usuario sobre el desarrollo de la ejecución del programa.

Con el fin de describir el funcionamiento interno del programa, se va a describir un ejemplo de uso en el que se describe el código que lleva a cabo las operaciones realizadas.

Figura 5-3. Detalle del botón de búsqueda de puertos y lista de puertos

La primera tarea que debe llevarse a cabo para que el programa pueda realizar su funcionamiento, es la de realizar una conexión con el puerto USB al que se conecta la FPGA. Para ello, en la barra superior existe una lista desplegable de los puertos disponibles detectados en el propio PC y un botón que la actualiza.

Se ha programado una función llamada *refresh_ports* que se encarga de actualizar la lista de puertos con conexión serie disponible. Mediante el módulo de comunicación *PySerial* se realiza un bucle que intenta una conexión con todos los posibles puertos posibles del PC, recogiendo en una lista unicamente los que han logrado completar la conexión.

Una vez se obtiene la lista de puertos disponibles, se actualiza la variable de la interfaz correspondiente a la lista de puertos USB. Esta función se ejecuta al comienzo del programa y se asocia al botón de búsqueda de puertos.

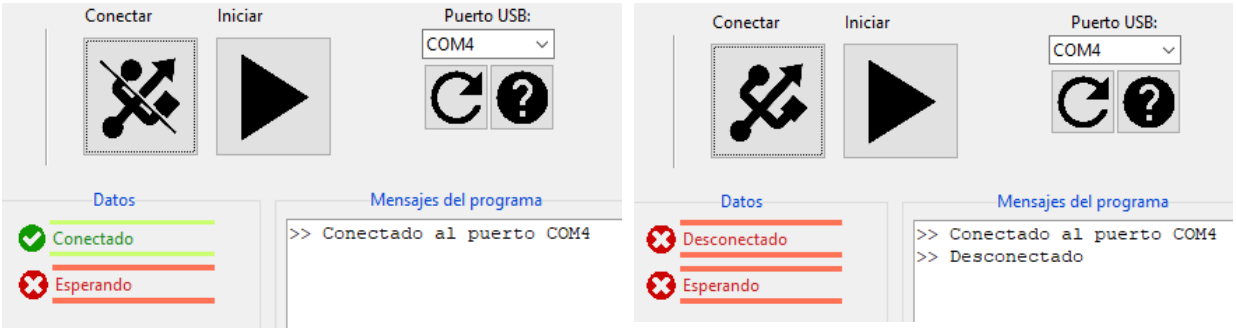


Figura 5-4. Detalle del estado de conexión y desconexión

Una vez seleccionado el puerto deseado de la lista desplegada, se pulsa el botón de conexión que ejecuta la función *pulsa_conexion* que está gobernada por una variable global de estado *f_con* que indica si el sistema está conectado o no. De esta forma, se aprovecha el mismo botón para realizar la conexión y desconexión, controlando dicha variable.

Cuando se ejecuta dicha función con la variable *f_con* indicando que no está conectado, se intenta iniciar una conexión con el puerto seleccionado en la lista desplegable de puertos. La conexión se realiza especificando que se va a realizar una velocidad de transmisión de 9600 Baudios, no hay bit de paridad y se ajusta un tiempo de timeout de 5 segundos.

Si la conexión es exitosa, se realizan un conjunto de modificaciones estéticas que indican al usuario que se encuentra conectado: indicador de estado que cambian de color y contenido, el botón de conexión cuyo icono se vuelve icono de desconexión y un mensaje por el panel de mensajes del programa.

En caso de de que se vuelva a pulsar el botón de conexión se ejecuta la misma función, pero en este caso la variable *f_con* estará indicando que está conectado, por lo que se realizará la desconexión del puerto y vuelta al estado original de los elementos visuales, así como un mensaje que indique la desconexión.

Figura 5-5. Errores previstos de conexión/desconexión

La función *pulsa_conexion* ha sido diseñada para detectar errores en la conexión, de forma que, si el puerto seleccionado no está disponible para la conexión, lo indicará mediante un mensaje y volverá al estado de desconectado. Además, si el usuario intenta desconectar el puerto mientras el programa está iniciado, el programa lo impedirá ya que causaría un mal funcionamiento dentro del resto de procesos. Estos dos casos van acompañados de los pertinentes mensajes por pantalla al usuario.

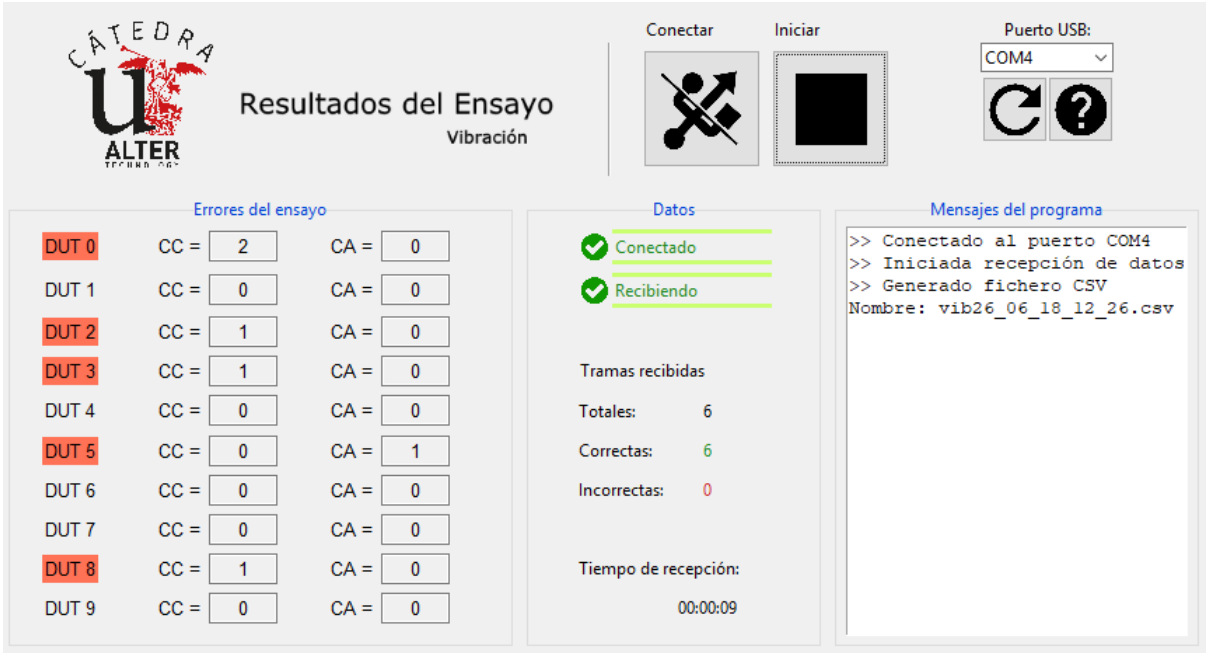


Figura 5-6. Interfaz durante el funcionamiento del programa principal

Para comenzar el funcionamiento del programa una vez tenemos la conexión con la FPGA realizada, se pulsa el botón de inicio que llama a la función *pulsa_inicio*. De una manera similar a la función de conexión, una variable *f_ini* se encarga de realizar el inicio o parada según su estado, cambiando los elementos estéticos pertinentes y mostrando mensajes por consola.

Al pulsar el botón para que inicie el programa se lanzan dos funciones en diferentes hilos, gracias al módulo de python *threading*. La primera de ellas es la función principal del programa *recibe* que coordina todas las funciones que debe realizar el programa. La segunda es la función *reloj* que controla el cronómetro presente en el área de datos del ensayo, que cuenta el tiempo transcurrido con el programa iniciado.

Es necesario usar hilos ya que la función principal *recibe* necesita estar recibiendo y tratando datos en todo momento, pero a la vez la interfaz debe actualizarse y ser capaz de realizar las funciones de control pertinentes a la misma vez, igual que el cronómetro.

Ambas funciones en hilos son bucles controlados por las variables *f_ini* y *f_tiempo* para que desde la interfaz se pueda parar estas funciones de ser necesario.

```
# Función principal de la funcionalidad
def recibe():
    global trama, n_tramas, n_tramas_bien, n_tramas_mal, men_prog
    while f_ini:
        trama = port_con.read(46) # Lee 46 Bytes del puerto serie
        if(len(trama) != 46):     # Detecta si se cumple el timeout de lectura
            men_prog.insert(INSERT, '>> Conexión perdida\n')
            pulsa_inicio()
            pulsa_conexion()
            break
        save_data()    # Almacena las variables recibidas
        if comp_crc(): # Verifica el CRC
            act_cont() # Actualiza los contadores
            gen_log()  # Actualiza el fichero de salida
        n_tramas = n_tramas_bien + n_tramas_mal
        act_GUI()      # Actualiza la interfaz gráfica
```

Figura 5-7. Función principal que realiza la funcionalidad recibe

La función *recibe* se encarga de realizar todas las funcionalidades previstas para este elemento del sistema ciclicamente mientras no se interrumpa externamente la ejecución mediante la pulsación del botón de parada o se produzca una desconexión.

Lo primero que se realiza es una lectura de 46 bytes por el puerto conectado, que equivale a una trama de las que envía la FPGA. La función de lectura se mantiene en espera hasta que lleguen los 46 bytes o se cumpla el tiempo de timeout. Como la FPGA está configurada para mandar tramas periódicas cada segundo, se considera que si se produce un timeout es que se ha perdido la conexión, por lo que se para la ejecución del programa, se desconecta el puerto y se manda un mensaje de aviso, para que el usuario revise las conexiones y vuelva a intentarlo.

En caso de recibir los 46 bytes se llama a la función *save_data* que se encarga de tratar los bytes recibidos, primero convirtiendolos en cadenas de caracteres de 8 bits para facilitar la posterior lectura de estos y luego convirtiendo estos en los valores numéricos que equivalen, segun sean enteros de contadores de errores, valor de ADC o número de trama.

Una vez los datos estan recibidos, se llama a la función *comp_crc* que vuelve a realizar el proceso de generación de CRC tal y como se realizó en la FPGA para comprobar si la trama ha llegado correctamente. En unos contadores internos se almacenan la cantidad de tramas que son correctas y las que no.

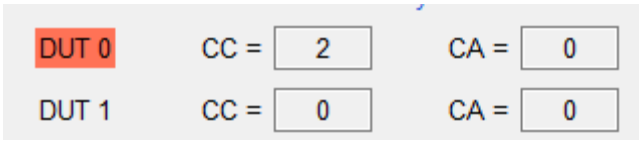


Figura 5-8. Detalle de los contadores de errores en la interfaz

En caso de que el CRC coincida y por tanto la trama haya sido validada, se llama a la función *act_cont* que usa las variables almacenadas anteriormente para actualizar los contadores de la interfaz gráfica. Además de actualizar el número de errores de determinado tipo para un DUT determinado, si se ha detectado algún error se añade un fondo rojo sobre la etiqueta del DUT en cuestión para indicar que ha fallado el ensayo.

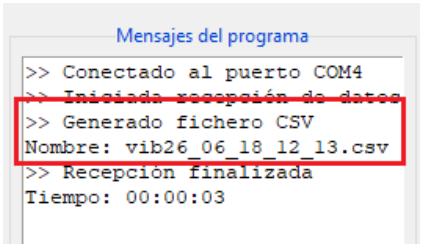


Figura 5-9. Detalle del mensaje de generación del fichero CSV

A continuación, se llama a la función *gen_log*, cuya función es generar un fichero de salida que almacene todas las tramas recibidas de forma que sea muy fácil obtener los datos y trabajar con ellos. Se ha implementado de forma que la generación sea automática al recibir una trama correcta y el formato de los datos sea CSV, debido a la gran compatibilidad con herramientas que permiten tratar los datos.

Para la generación del fichero se utiliza el módulo de python *CSV*, que facilita la escritura de los datos en dicho formato. Cuando el fichero es generado automáticamente, se indica al usuario mediante un mensaje de programa, indicando el nombre que contendrá la fecha y hora del momento en el que se genere.

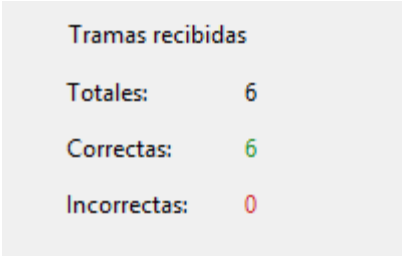


Figura 5-10. Detalle de la estadística de tramas recibidas en la interfaz

Se haya validado la trama o no, antes de leer por el puerto serie de nuevo, la función llama a la función *act_GUI* que actualiza los contadores de tramas de la interfaz, que muestran la cantidad de tramas recibidas y un desglose de las cuales han sido correctas y cuales incorrectas y por tanto deshechadas.

Al finalizar la ejecución del programa pulsando el botón de inicio de nuevo se muestra un mensaje indicando el tiempo transcurrido desde que se inició. Además, se vuelven a preveer utilizaciones prohibidas del programa como intentar iniciar el programa sin haber realizado conexion con un puerto, que se solucionan evitando que esto ocurra.

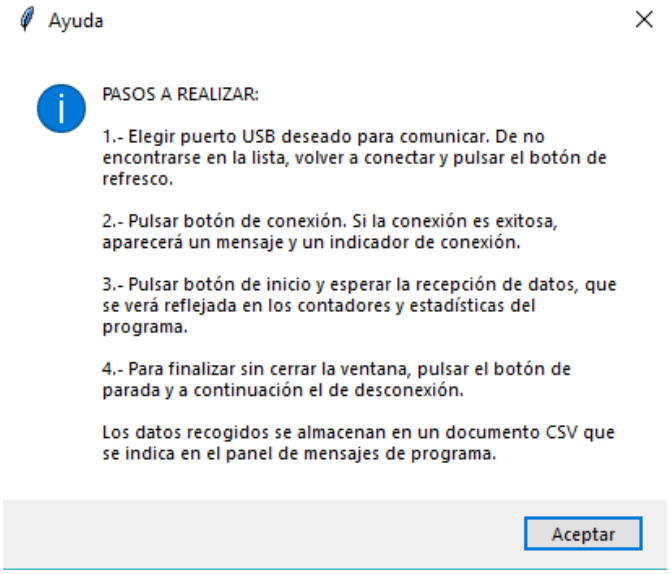


Figura 5-11. Mensaje de ayuda generado al pulsar el botón ayuda

Una última funcionalidad añadida al programa se encuentra al pulsar el botón de ayuda, que genera una ventana emergente que indica los pasos a realizar para ejecutar el programa correctamente.

5.3. Verificación

La verificación del software se realiza simulando la transmisión de tramas mediante un dispositivo externo que se comunique con el PC por puerto serie y envíe tramas tal y como lo haría la FPGA. Para realizar las pruebas y depuración de código se han utilizado los ficheros de salida de la simulación del código VHDL, que contienen las tramas simuladas con determinados estímulos fijados por el usuario, por lo tanto conocidos.

Figura 5-12. Microcontrolador Arduino UNO

Para poder realizar una transmisión cómoda y modificable de manera sencilla, se ha utilizado una placa de microcontrolador Arduino UNO, que es muy versatil y con la que se pueden realizar transmisiones serie de una manera muy sencilla.

De esta forma, se puede verificar en una primera aproximación que las diferentes funciones del software desarrollado funcionan correctamente.

```
/* Simulación de Software vibración */

// Datos que se envian
const byte data[] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
const int byte_trama = 46;
int imax = sizeof(data)/byte_trama;
int i;

void setup() {
  Serial.begin(9600);
  i = 0;
  delay(1000);

}

void loop() {
  if(i < imax){
    Serial.write(&data[byte_trama*i], byte_trama);
    delay(1000);
    i++;
  }
}
```

Figura 5-13. Script de Arduino que envía los datos de simulación

Es necesario adaptar los valores recibidos del fichero de simulación convirtiendolos en enteros para almacenarlos como variables en la memoria del Arduino. Estos datos van a ser enviados en tramas de 46 bytes seguidos, dejando un segundo entre envios de 46 bytes, tal y como realizaría la FPGA.

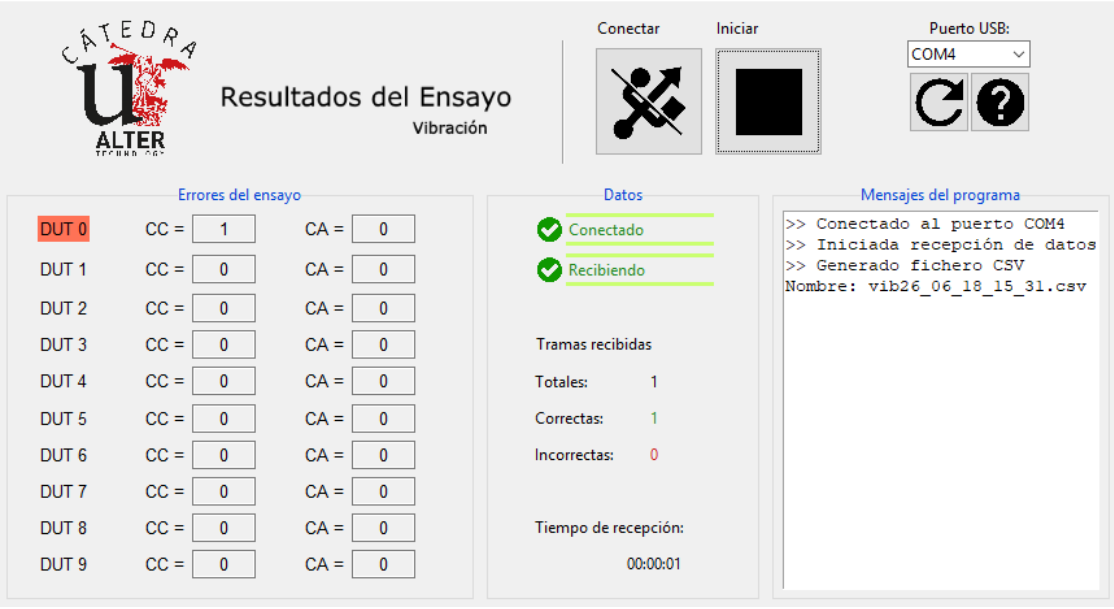


Figura 5-14. Captura del software mientras se reciben tramas

Una vez iniciado el software, realizamos los pasos descritos anteriormente para ubicar el puerto USB al que se ha conectado el microcontrolador, conectar e iniciar el programa principal.

Se observa como cada segundo se actualiza el valor de los contadores de errores, así como el de las estadísticas de tramas recibidas y se generan los correspondientes mensajes del programa.

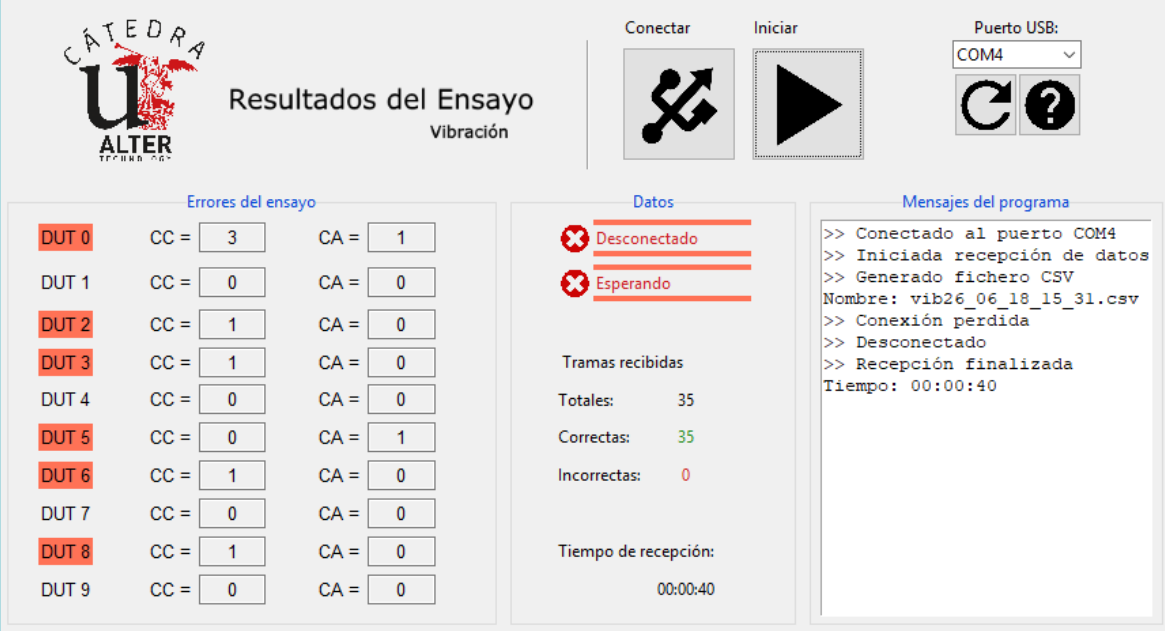


Figura 5-15. Capturas del software una vez se termina la transmisión

Una vez se reciben todas las tramas almacenadas en la memoria del Arduino, este queda en estado de reposo sin hacer nada, lo que el programa detectará al cabo de 5 segundos como una pérdida de conexión y finalizará tanto el programa como la conexión del puerto.

	A	B	C	D	E	F	G	H	I
1	Nº trama,CC0,CA0,CC1,CA1,CC2,CA2,CC3,CA3,CC4,CA4,CC5,CA5,CC6,CA6,CC7,CA7,CC8,CA8,CC9,CA9,Dato ADC								
2	1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15360								
3	2,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15360								
4	3,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15360								
5	4,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15360								
6	5,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,15360								
7	6,2,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
8	7,2,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
9	8,2,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
10	9,2,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
11	10,2,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
12	11,2,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
13	12,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
14	13,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
15	14,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
16	15,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
17	16,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
18	17,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
19	18,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
20	19,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
21	20,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
22	21,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								
23	22,3,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,1,0,0,0,15360								

Figura 5-16. Fichero CSV generado a la salida

En la carpeta *log* se genera el dichero CSV correspondiente, que, aunque por sí mismo no es muy útil, es un formato muy extendido del cual es muy fácil extraer la información y tratarla.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	Nº trama	CC0	CA0	CC1	CA1	CC2	CA2	CC3	CA3	CC4	CA4	CC5	CA5	CC6	CA6	CC7	CA7	CC8	CA8	CC9	CA9	Dato ADC
2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15360
3	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15360
4	3	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15360
5	4	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15360
6	5	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15360
7	6	2	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
8	7	2	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
9	8	2	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
10	9	2	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
11	10	2	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
12	11	2	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
13	12	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
14	13	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
15	14	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
16	15	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
17	16	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
18	17	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
19	18	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
20	19	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
21	20	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
22	21	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
23	22	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
24	23	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
25	24	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
26	25	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360
27	26	3	0	0	0	1	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	15360

Figura 5-17. Datos obtenidos a partir del fichero CSV en MS Excel

Con Microsoft Excel cargamos dicho fichero CSV para generar una tabla con los datos ordenados con la que se simplifican mucho las operaciones que quieran hacerse sobre los datos.

Una vez visto el correcto funcionamiento ante una recepción perfecta, se va a verificar el funcionamiento ante un escenario en el que la transmisión es problemática y se reciben tramas erroneas. Esto se realiza cambiando algunos valores de los almacenados en la memoria del Arduino y repitiendo la prueba.

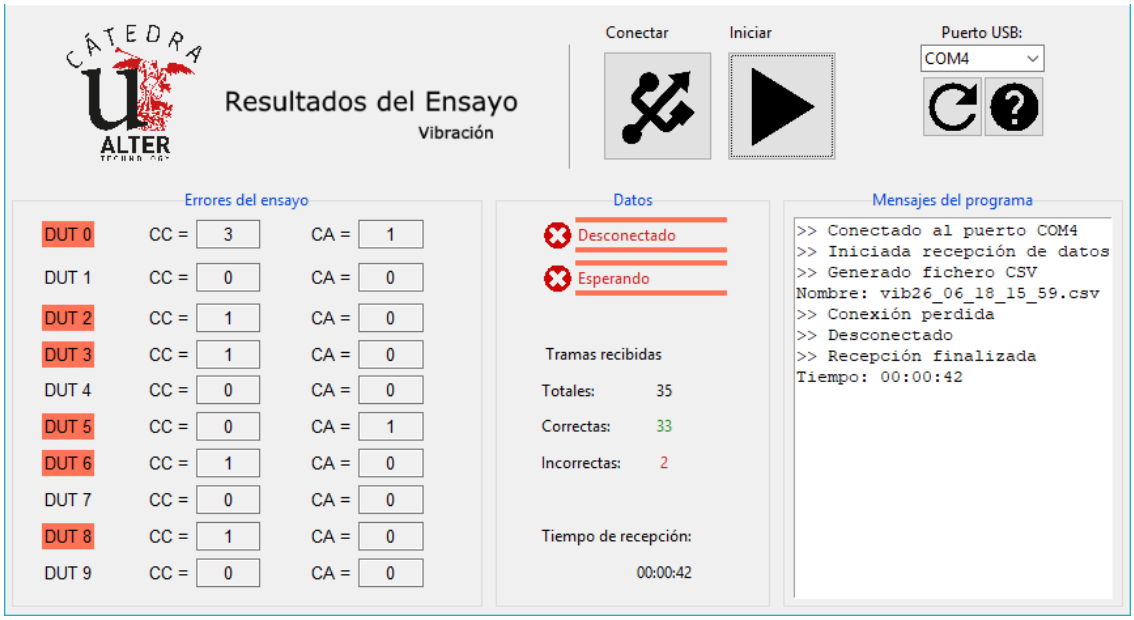


Figura 5-18. Captura del software en prueba con datos incorrectos

Se observa que aparecen dos tramas deshechadas por el software, lo que indica que la función de comprobación del CRC funciona correctamente. La funcionalidad se mantiene y los datos mostrados en pantalla son correctos incluso fallando la transmisión en un momento concreto.

6 VERIFICACIÓN DE LA COMUNICACIÓN

Pese a las pruebas realizadas con las simulaciones del código VHDL y del software mediante la placa Arduino, es necesario verificar que realmente los dos sistemas implementan la comunicación de forma fiable y eficaz, comportandose de la manera prevista.

La comunicación es uno de los elementos claves del sistema, ya que sin ella los datos de errores obtenidos nunca podrían ser interpretados y obtenidos. Para realizar la comunicación serie entre ambos elementos, se utiliza un cable adaptador de USB a conector serie RS-232.

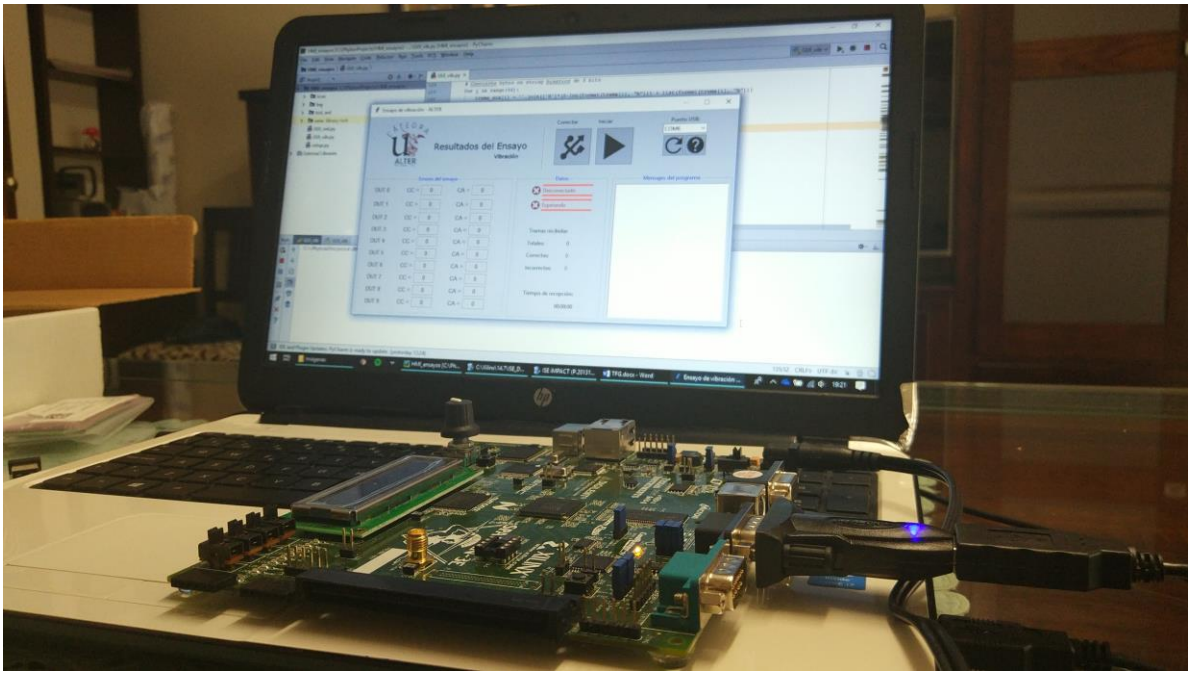


Figura 6-1. Montaje de la prueba de transmisión

Es necesario alimentar la placa de FPGA y verificar que el cable de transmisión se encuentra perfectamente ajustado. A su vez, debe iniciarse el software de la manera indicada anteriormente y realizar la conexión al puerto COM6 que utiliza el adaptador USB.

Una vez realizado el montaje, se resetea la FPGA para asegurar que los contadores empiezan a cero y se inicia la recepción en el software.

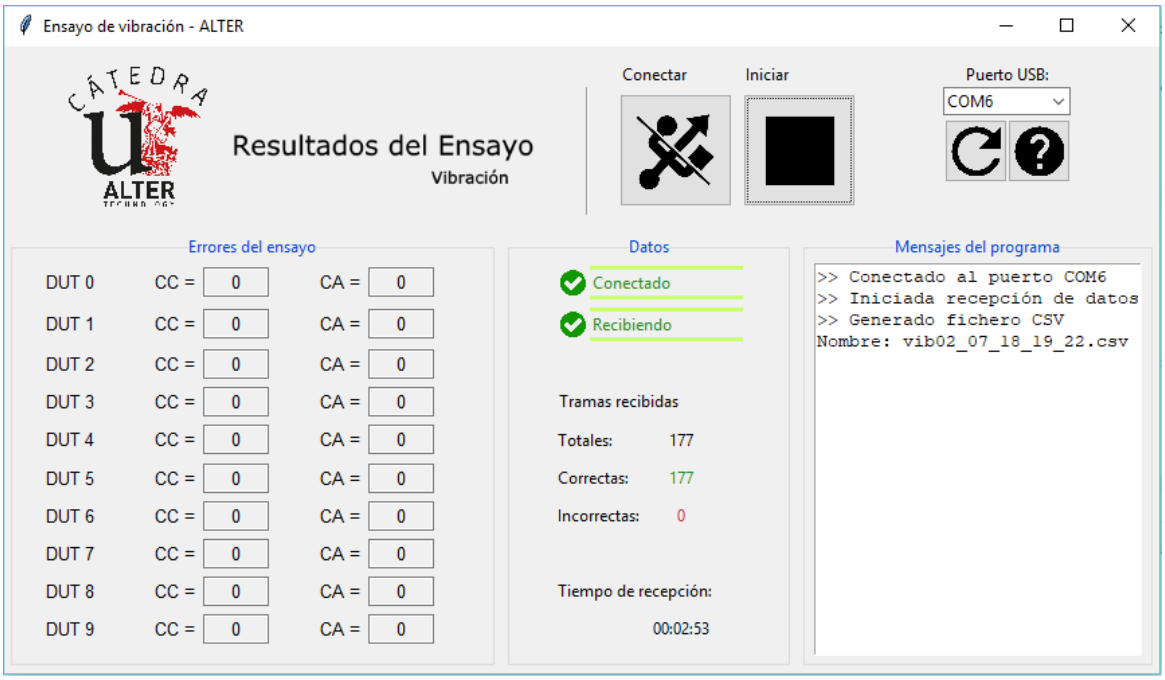


Figura 6-2. Captura del software durante la prueba de transmisión

Se empiezan a recibir datos procedentes de la FPGA. Las tramas son procesadas y se verifica el CRC de cada una, por lo que, aunque no se perciban cambios en los contadores (debido a la ausencia de estímulos), se confirma que la transmisión es exitosa observando las estadísticas ofrecidas de tramas recibidas.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	Nº trama	CC0	CA0	CC1	CA1	CC2	CA2	CC3	CA3	CC4	CA4	CC5	CA5	CC6	CA6	CC7	CA7	CC8	CA8	CC9	CA9	Dato ADC
2	14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	27	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	28	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	29	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figura 6-3. Tramas recibidas en la prueba de transmisión

Gracias al log generado es posible confirmar que, aunque las tramas posean contadores vacios, el valor del número de tramas se incrementa, por lo que el CRC será distinto en cada una.

De esta forma, se verifica que la comunicación diseñada funciona correctamente, cada parte realiza su función y ambas se entienden, independientemente del contenido de las tramas.

7 CONCLUSIONES Y DESARROLLOS FUTUROS

En este último capítulo se va a realizar una exposición de las conclusiones del presente trabajo. Además, se expondrán una serie de ideas y proyectos que se han valorado o planificado para la expansión del sistema desarrollado.

7.1. Conclusiones

Una vez concluido el trabajo, puede confirmarse que se ha alcanzado el objetivo de crear un sistema funcional y válido para montorizar errores eléctricos durante ensayos de vibración. Una de las claves ha sido que pese a que el proceso ha abarcado numerosas fases de muy distintos ambitos, se han ido completando y verificando una a una, haciendo más sencillo obtener el resultado final en el que confluyen todas estas fases.

El hecho de haber llegado hasta la fabricación del sistema ha hecho posible realizar una serie de pruebas reales que han servido para verificar que las funcionalidades planteadas a comienzo de cada parte del sistema han tomado forma correctamente.

Por ello, se concluye que el desarrollo del sistema ha sido un éxito, ya que se ha podido verificar que realiza todas las funcionalidades previstas, cumpliendo los requisitos tanto referentes a normativas y especificaciones como los propios de diseño.

7.2. Desarrollos futuros

El sistema actual es la primera versión de una serie de sistemas de monitorización de errores para ensayos de componentes de alta fiabilidad. Debido a la numerosa cantidad de ensayos realizados a numerosos tipos de dispositivos, se presentan una gran cantidad de posibilidades sobre las que realizar un sistema similar.

Debido a que precisa unas carasterísticas de monitorización eléctrica similares, es interesante plantear un sistema de monitorización de errores en ensayos de radiación. Los ensayos de radiación son una parte fundamental de los componentes de la industria aeroespacial, ya que es fuente de numerosos tipos de error en multitud de componentes.

Por esto, está previsto que una futura segunda versión del sistema se realice para la monitorización de errores en un ensayo de radiación para convertidores analógico-digital, componente clave en todo tipo de sistemas eléctronicos realizados actualmente.

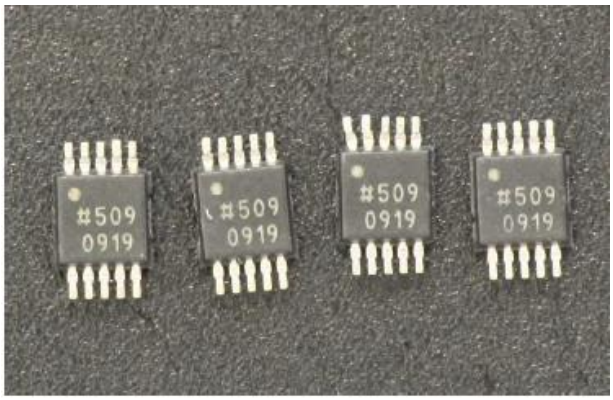


Figura 7-1. Dispositivos AD7982

Al igual que este caso, el sistema se plantea en torno a un ensayo ya realizado, en este caso para el ADC *AD7982* de *Analog Devices.* Se pretenden detectar y medir los errores de radiación conocidos como *Single Event Efects*, en concreto los conocidos como *Latch-up* y *Upset.*

En esta primera versión del sistema se han desarrollado elementos reaprovechables para esta u otras versiones futuras como la comunicación SPI entre la FPGA y el ADC o la comunicación serie entre FPGA y PC.

Manteniendo la esencia de la estructura del sistema, que abarca desde el mismo elemento ensayado hasta una interfaz gráfica en la que se reflejen los resultados, se pueden realizar cambios que mejoren la eficiencia o simplemente se adapten a los diferentes tipos de ensayo, cada uno con sus particularidades.

REFERENCIAS

[1] ALTER Technology, «Screening Plan - Based on Platinum Temperature Sensor P1K0.0805.4P.B.S from Innovative Sensor Technology,» 2017.

[2] United States Department of Defense, «Test Method Standard - Method 204: Vibration, High Frecuency,» 2015.

[3] United States Department of Defense, «Resistor, Chip, Thermal (Thermistor), General Specification for - MIL-PRF-32192,» 2012.

[4] NASA, «Thermistor (Thermally Sensitive Resistor), Chip, Negative Temperature Coefficient or Positive Teperature Coefficient - S-311-P-827,» 2009.

[5] Xilinx, «Spartan-3E Starter Kit Board User Guide,» 2006.

[6] Dallas Semiconductor, «Application Note 83 - Fundamentals of RS-232 Serial Communications,» 2001.

[7] «Wikipedia,» [En línea]. Available: https://en.wikipedia.org/wiki/Cyclic_redundancy_check.

[8] ESA, «Single Events Effects Test Method and Guidelines - 25100,» 1995.

GLOSARIO

NASA: *National Aeronautics and Space Administration* 1

ESA: *European Space Agency* 1

DUT: *Device Under Test* 2

PCB: *Printed Circuit Board* 2

FPGA: *Field-Programmable Gate Array* 2

GUI: *Graphic User Interface* 2

ANEXOS

Anexo A: Listado de componentes de la placa de adaptación

Tabla A-1. Listado de componentes de la placa de adaptación

Componente	Carasterísticas	Referencia de fabricante	#	Precio/ud
Regulador de conmutación	3,3 V, 1 A	TSRN 1-2433SM	1	12,14 €
Regulador de tensión LDO	2,5 V, 0,8 A	LM1117DT-2.5	1	1,27 €
Referencia de tensión	1,235 V, ± 0,3%	LT1004IS8-1.2	1	4,11 €
Amplificador de corriente	Ganancia ajustable	LT6105IMS8	10	2,11 €
Comparador	Colector abierto, dos canales	LM293AD	10	0,14 €
Condensador de tántalo	10 µF, 10 V, 0805	TPSP106M010	2	0,27 €
Condensador cerámico	100 nF, 16 V, 0805	C0805C104K4RAC	20	0,17 €
Resistencia de precision	1 kΩ, ± 0.1%	ERA3AEB102V	1	0,15 €
Resistencia de precision	2 kΩ, ± 0.1%	ERA3AEB202V	1	0,26 €
Resistencia de precision	100 kΩ, ± 0.1%	ERA3AEB104V	1	0,25 €
Resistencia	10 Ω, ±1%	CRG0603F10R	10	0,03 €
Resistencia	200 Ω, ±1%	ERJPA3F2000V	20	0,03 €
Resistencia	1 kΩ, ±1%	CRG0603F1K0	10	0,01 €
Resistencia	10 kΩ, ±1%	CR0603-FX-1002	30	0,03 €
Conector PCB macho	4 pines, 2,54 mm	70543-0038	1	0,83 €
Carcasa de conector	4 contactos, 2,54 mm	50-57-9404	1	0,26 €
Contacto terminal crimpado		16-02-0088	2	0,15 €
Conector PCB macho	Serie Duraclik, 15 pines, 2 mm	502352-1500	2	1,15 €
Carcasa de conector	Serie Duraclik, 15 pines, 2mm	502351-1500	2	0,2 €

Componente	Carasterísticas	Referencia de fabricante	#	Precio/ud
Contacto terminal crimpado	Serie Duraclik	560085-0101	20	0,04 €
Soporte especial conector	Serie Duraclik, 15 pines, 2 mm	505152-1500	2	0,10 €
Cable conexión	AWG 22, color blanco	3251 WH005	1	12,50 €
Conector PCB macho	Serie FX2, 100 pines, 1,27 mm	FX2-100P-1.27DS(71)	1	6,14 €
Conector IDC	Serie FX2, 100 pines, 1,27 mm	FX2BA-100SA-1.27R	2	8,67 €
Cable cinta plano	50 vias, 1,27 mm	289-9981	1	21,32 €
Conector PCB macho	14 pines, 2,54 mm	T821114A1S100CEU	1	0,36 €
Conector IDC	14 contactos, 2,54 mm	T812114A101CEU	1	0,48 €
Cable cinta plano	14 vias, 1,27 mm	289-9852	1	6,45 €

Anexo B: Código VHDL

- **vibracion.vhd**

```
-- Bloque principal que contiene el programa completo --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity vibracion is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           V : in  STD_LOGIC_VECTOR (9 downto 0);
           I : in  STD_LOGIC_VECTOR (9 downto 0);
           SDO : in STD_LOGIC;
           CNV : out STD_LOGIC;
           SDI : out STD_LOGIC;
           SCK : out STD_LOGIC;
           tx : out STD_LOGIC
              );
end vibracion;

architecture Behavioral of vibracion is

    COMPONENT counter_box_time
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        Vext : IN std_logic_vector(9 downto 0);
        Iext : IN std_logic_vector(9 downto 0);
        Verr : OUT std_logic_vector(9 downto 0);
        Ierr : OUT std_logic_vector(9 downto 0)
        );
    END COMPONENT;

    COMPONENT counter_box_error
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        Verr : IN std_logic_vector(9 downto 0);
        Ierr : IN std_logic_vector(9 downto 0);
        trama_err : OUT std_logic_vector(319 downto 0)
        );
    END COMPONENT;

    COMPONENT ADC_controller
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        SDO : IN std_logic;
        ini_conv : IN std_logic;
        CNV : OUT std_logic;
        SDI : OUT std_logic;
        SCK : OUT std_logic;
        fin_conv : OUT std_logic;
        trama_adc : OUT std_logic_vector(23 downto 0)
        );
    END COMPONENT;
```

```
    COMPONENT counter_time_trama
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        sat : OUT std_logic
        );
    END COMPONENT;

    COMPONENT counter_n_trama
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        enable : IN std_logic;
        counter : OUT std_logic_vector(15 downto 0)
        );
    END COMPONENT;

    COMPONENT crc_gen
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        start_crc : IN std_logic;
        trama_err : IN std_logic_vector(319 downto 0);
        trama_adc : IN std_logic_vector(23 downto 0);
        trama_num : IN std_logic_vector(15 downto 0);
        end_crc : OUT std_logic;
        trama : OUT std_logic_vector(367 downto 0)
        );
    END COMPONENT;

    COMPONENT tx_pc
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        trama : IN std_logic_vector(367 downto 0);
        trama_val : IN std_logic;
        tx : OUT std_logic
        );
    END COMPONENT;

    signal s_ini_conv, s_ini_crc, s_ini_tx : std_logic;
    signal s_Verr, s_Ierr: std_logic_vector(9 downto 0);
    signal s_trama_err : std_logic_vector(319 downto 0);
    signal s_trama_adc : std_logic_vector(23 downto 0);
    signal s_trama_num : std_logic_vector(15 downto 0);
    signal s_trama : std_logic_vector(367 downto 0);
begin

    B1: counter_box_time PORT MAP(
        clk => clk,
        reset => reset,
        Vext => V,
        Iext => I,
        Verr => s_Verr,
        Ierr => s_Ierr
    );

    B2: counter_box_error PORT MAP(
        clk => clk,
        reset => reset,
```

```
        Verr => s_Verr,
        Ierr => s_Ierr,
        trama_err => s_trama_err
    );

    B3: ADC_controller PORT MAP(
        clk => clk,
        reset => reset,
        SDO => SDO,
        CNV => CNV,
        SDI => SDI,
        SCK => SCK,
        ini_conv => s_ini_conv,
        fin_conv => s_ini_crc,
        trama_adc => s_trama_adc
    );

    B4: counter_time_trama PORT MAP(
        clk => clk,
        reset => reset,
        sat => s_ini_conv
    );

    B5: counter_n_trama PORT MAP(
        clk => clk,
        reset => reset,
        enable => s_ini_conv,
        counter => s_trama_num
    );

    B6: crc_gen PORT MAP(
        clk => clk,
        reset => reset,
        start_crc => s_ini_crc,
        trama_err => s_trama_err,
        trama_adc => s_trama_adc,
        trama_num => s_trama_num,
        end_crc => s_ini_tx,
        trama => s_trama
    );

    B7: tx_pc PORT MAP(
        clk => clk,
        reset => reset,
        trama => s_trama,
        trama_val => s_ini_tx,
        tx => tx
    );
end Behavioral;
```

- **counter_box_time.vhd**

```
-- Contiene los contadores de tiempo que detectan errores --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_box_time is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
```

```
           Vext : in  STD_LOGIC_VECTOR (9 downto 0);
           Iext : in  STD_LOGIC_VECTOR (9 downto 0);
           Verr : out  STD_LOGIC_VECTOR (9 downto 0);
           Ierr : out  STD_LOGIC_VECTOR (9 downto 0));
end counter_box_time;

architecture Behavioral of counter_box_time is

COMPONENT counter_time
    PORT(
        clk : IN std_logic;
        reset : IN std_logic;
        enable : IN std_logic;
        sat : OUT std_logic
        );
END COMPONENT;

begin

    VC0: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(0),
        sat => Verr(0)
    );

    VC1: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(1),
        sat => Verr(1)
    );

    VC2: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(2),
        sat => Verr(2)
    );

    VC3: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(3),
        sat => Verr(3)
    );

    VC4: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(4),
        sat => Verr(4)
    );

    VC5: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(5),
        sat => Verr(5)
    );
```

```
    VC6: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(6),
        sat => Verr(6)
    );

    VC7: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(7),
        sat => Verr(7)
    );

    VC8: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(8),
        sat => Verr(8)
    );

    VC9: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Vext(9),
        sat => Verr(9)
    );

    IC0: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(0),
        sat => Ierr(0)
    );

    IC1: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(1),
        sat => Ierr(1)
    );

    IC2: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(2),
        sat => Ierr(2)
    );

    IC3: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(3),
        sat => Ierr(3)
    );

    IC4: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(4),
        sat => Ierr(4)
    );
```

```
    IC5: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(5),
        sat => Ierr(5)
    );

    IC6: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(6),
        sat => Ierr(6)
    );

    IC7: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(7),
        sat => Ierr(7)
    );

    IC8: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(8),
        sat => Ierr(8)
    );

    IC9: counter_time PORT MAP(
        clk => clk,
        reset => reset,
        enable => Iext(9),
        sat => Ierr(9)
    );

end Behavioral;
```

- **counter_time.vhd**

```
-- Cuenta 100 us para detectar errores --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_time is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           enable : in  STD_LOGIC;
           sat : out  STD_LOGIC);
end counter_time;

architecture Behavioral of counter_time is
    type states is (contando, esperando);
    signal state, n_state : states;
    signal count, n_count : unsigned (12 downto 0);
begin

    sinc: process(clk, reset)
```

```
    begin
        if (reset = '1') then
            state <= contando;
            count <= (others => '0');
        elsif (rising_edge(clk)) then
            state <= n_state;
            count <= n_count;
        end if;
    end process;

    comb: process(state, count, enable)
    begin
        sat <= '0';
        n_state <= state;
        n_count <= count;
        case state is
            -- Estado natural, cuenta si se activa enable
            when contando =>
                if (enable = '1') then
                    -- 5000 * 20 ns = 100 us
                    if (count = "1001110000111") then
                        n_count <=(others => '0');
                        sat <= '1';
                        n_state <= esperando;
                    else
                        n_count <= count + 1;
                    end if;
                else
                    -- Si se desactiva enable, se resetea el contador
                    n_count <= (others => '0');
                end if;

            -- Espera que se desactive la entrada para volver a contar
            when others =>
                if (enable = '0') then
                    n_state <= contando;
                end if;
        end case;
    end process;

end Behavioral;
```

- **counter_box_error.vhd**

```
-- Contiene los contadores que cuentan los errores cometidos --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_box_error is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           Verr : in  STD_LOGIC_VECTOR (9 downto 0);
           Ierr : in  STD_LOGIC_VECTOR (9 downto 0);
              trama_err : out STD_LOGIC_VECTOR (319 downto 0)
             );
end counter_box_error;

architecture Behavioral of counter_box_error is
```

```
    COMPONENT counter_error
        PORT(
            clk : IN std_logic;
            reset : IN std_logic;
            enable : IN std_logic;
            counter : OUT std_logic_vector(15 downto 0)
            );
    END COMPONENT;

begin

    VC0: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(0),
        counter => trama_err(15 downto 0)
    );

    VC1: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(1),
        counter => trama_err(31 downto 16)
    );


        VC2: counter_error PORT MAP(

        clk => clk,
        reset => reset,
        enable => Verr(2),
        counter => trama_err(47 downto 32)
    );

    VC3: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(3),
        counter => trama_err(63 downto 48)
    );

    VC4: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(4),
        counter => trama_err(79 downto 64)
    );

    VC5: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(5),
        counter => trama_err(95 downto 80)
    );

    VC6: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(6),
        counter => trama_err(111 downto 96)
    );
```

```
    VC7: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(7),
        counter => trama_err(127 downto 112)
    );

    VC8: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(8),
        counter => trama_err(143 downto 128)
    );

    VC9: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Verr(9),
        counter => trama_err(159 downto 144)
    );

    IC0: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(0),
        counter => trama_err(175 downto 160)
    );

    IC1: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(1),
        counter => trama_err(191 downto 176)
    );

    IC2: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(2),
        counter => trama_err(207 downto 192)
    );

    IC3: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(3),
        counter => trama_err(223 downto 208)
    );

    IC4: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(4),
        counter => trama_err(239 downto 224)
    );

    IC5: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(5),
        counter => trama_err(255 downto 240)
    );
```

```
    IC6: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(6),
        counter => trama_err(271 downto 256)
    );

    IC7: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(7),
        counter => trama_err(287 downto 272)
    );

    IC8: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(8),
        counter => trama_err(303 downto 288)
    );

    IC9: counter_error PORT MAP(
        clk => clk,
        reset => reset,
        enable => Ierr(9),
        counter => trama_err(319 downto 304)
    );

end Behavioral;
```

- **counter_error.vhd**

```
-- Cuenta el número de errores cometidos --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_error is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           enable : in  STD_LOGIC;
           counter : out  STD_LOGIC_VECTOR (15 downto 0));
end counter_error;

architecture Behavioral of counter_error is
    signal count, n_count : unsigned (15 downto 0);
begin
    sinc: process(clk, reset)
    begin
        if (reset = '1') then
            count <= (others => '0');
        elsif (rising_edge(clk)) then
            count <= n_count;
        end if;
    end process;

    comb: process(count, enable)
    begin
        counter <= std_logic_vector(count);
```

```
        n_count <= count;
        if (enable = '1') then
            if (count = "111111111111111") then
                n_count <= count;
            else
                n_count <= count + 1;
            end if;
        end if;
    end process;
end Behavioral;
```

- **counter_time_trama.vhd**

```
-- Cuenta el tiempo entre tramas --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_time_trama is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           sat : out  STD_LOGIC);
end counter_time_trama;

architecture Behavioral of counter_time_trama is
    signal count, n_count : unsigned (25 downto 0);
begin

    sinc: process(clk, reset)
    begin
        if (reset = '1') then
            count <= (others => '0');
        elsif (rising_edge(clk)) then
            count <= n_count;
        end if;
    end process;

    comb: process(count)
    begin
        -- 50000000 * 20 ns = 1 s
        if (count = "10111110101111000001111111") then
            n_count <= (others => '0');
            sat <= '1';
        else
            n_count <= count + 1;
            sat <= '0';
        end if;
    end process;

end Behavioral;
```

- **counter_n_trama.vhd**

```
-- Cuenta el número de tramas enviadas --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity counter_n_trama is
```

```
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           enable : in  STD_LOGIC;
           counter : out  STD_LOGIC_VECTOR(15 downto 0));
end counter_n_trama;

architecture Behavioral of counter_n_trama is
    signal count, n_count : unsigned (15 downto 0);
begin

    sinc: process(clk, reset)
    begin
        if (reset = '1') then
            count <= (others => '0');
        elsif (rising_edge(clk)) then
            count <= n_count;
        end if;
    end process;

    comb: process(count, enable)
    begin
        counter <= std_logic_vector(count);
        n_count <= count;
        if (count = "1111111111111111") then
            n_count <= count;
        elsif (enable = '1') then
            n_count <= count + 1;
        end if;
    end process;

end Behavioral;
```

- **ADC_controller.vhd**

```
-- Realiza la comunicación SPI con el ADC --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ADC_controller is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           SDO : in  STD_LOGIC;
           CNV : out  STD_LOGIC;
           SDI : out  STD_LOGIC;
           SCK : out  STD_LOGIC;
           ini_conv : in STD_LOGIC;
           fin_conv : out STD_LOGIC;
           trama_adc : out STD_LOGIC_VECTOR(23 downto 0));
end ADC_controller;

architecture Behavioral of ADC_controller is
    type states is (reposo, conversion, espera, adquisicion);
    signal state, n_state : states;
    signal tcount, n_tcount : unsigned (5 downto 0);  -- Contador de tiempo
    signal index, n_index : integer range 0 to 23;    -- Contador de bits
    signal adc_data, n_adc_data : std_logic_vector (23 downto 0);
begin
```

```
    sinc: process(clk, reset)
    begin
        if (reset = '1') then
            state <= reposo;
            tcount <= (others => '0');
            index <= 17;
            adc_data <= (others => '0');
        elsif (rising_edge(clk)) then
            state <= n_state;
            tcount <= n_tcount;
            index <= n_index;
            adc_data <= n_adc_data;
        end if;
    end process;

    comb: process(state, tcount, index, adc_data, ini_conv, SDO)
    begin
        n_state <= state;
        n_tcount <= tcount;
        n_index <= index;
        n_adc_data <= adc_data;

        trama_adc <= adc_data;

        fin_conv <= '0';
        CNV <= '0';
        SDI <= '1';
        SCK <= '0';

        case state is
            -- Estado de reposo, espera aviso para actuar
            when reposo =>
                if (ini_conv = '1') then
                    n_state <= conversion;
                end if;

            -- Fase de conversion, se le avisa al ADC para que comience
            when conversion =>
                CNV <= '1';
                -- 39 * 20 ns = 780 ns
                if (tcount = "100111") then
                    n_tcount <= (others => '0');
                    n_state <= espera;
                else
                    n_tcount <= tcount + 1;
                end if;

            -- Espera el tiempo recomendado entre fases del ADC
            when espera =>
                n_state <= adquisicion;

            -- Fase de adquisición, se recoge el dato del ADC
            when others =>
                if (tcount = "000001" ) then -- Semiperiodo a nivel bajo
                    n_tcount <= (others => '0');
                    n_adc_data(index) <= SDO;
                    if (index = 0) then -- Termina la recepción
                        n_index <= 17;
                        n_state <= reposo;
                        fin_conv <= '1';
                    else
                        n_index <= index - 1;
```

```
                    end if;
                else -- Semiperiodo a nivel alto
                    SCK <= '1';
                    n_tcount <= tcount + 1;
                end if;

        end case;
    end process;
end Behavioral;
```

- **crc_gen.vhd**

```
-- Genera el CRC y forma la trama final --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity crc_gen is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           start_crc : in STD_LOGIC;
           trama_err : in  STD_LOGIC_VECTOR (319 downto 0);
           trama_adc : in  STD_LOGIC_VECTOR (23 downto 0);
           trama_num : in  STD_LOGIC_VECTOR (15 downto 0);
           end_crc : out STD_LOGIC;
           trama : out  STD_LOGIC_VECTOR (367 downto 0));
end crc_gen;

architecture Behavioral of crc_gen is
    type states is (reposo, get_crc);
    signal state, n_state : states;
    signal trama_aux : std_logic_vector (367 downto 0);
    signal index, n_index : integer range 0 to 367;
    signal crc, n_crc : std_logic_vector (7 downto 0);
begin

    sinc: process(clk, reset)
    begin
        if (reset = '1') then
            state <= reposo;
            index <= 0;
            crc <= (others => '0');
        elsif (rising_edge(clk)) then
            state <= n_state;
            index <= n_index;
            crc <= n_crc;
        end if;
    end process;

    comb: process(state, start_crc, index, crc, trama_aux, trama_err,
trama_adc, trama_num)
    begin
        n_state <= state;
        n_index <= index;
        n_crc <= crc;

        trama_aux(319 downto 0) <= trama_err;
        trama_aux(343 downto 320) <= trama_adc;
        trama_aux(359 downto 344) <= trama_num;
        trama_aux(367 downto 360) <= crc;
```

```
        trama <= trama_aux;
        end_crc <= '0';

        case state is
            -- Espera acción para poder iniciar el proceso
            when reposo =>
                if (start_crc = '1') then
                    n_crc <= (others => '0');
                    n_state <= get_crc;
                end if;

            -- Forma el CRC y avisa al transmisor
            when others =>
                if (index = 367) then
                    end_crc <= '1';
                    n_index <= 0;
                    n_state <= reposo;
                elsif (index > 359) then
                    n_crc(0) <= '0' xor crc(7);
                    n_crc(1) <= crc(0) xor crc(7);
                    n_crc(2) <= crc(1) xor crc(7);
                    n_crc(7 downto 3) <= crc(6 downto 2);
                    n_index <= index + 1;
                else
                    n_crc(0) <= trama_aux(index) xor crc(7);
                    n_crc(1) <= crc(0) xor crc(7);
                    n_crc(2) <= crc(1) xor crc(7);
                    n_crc(7 downto 3) <= crc(6 downto 2);
                    n_index <= index + 1;
                end if;

        end case;
    end process;
end Behavioral;
```

- **tx_pc.vhd**

```
-- Transmite la trama al PC por RS-232 --
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity tx_pc is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           trama : in  STD_LOGIC_VECTOR (367 downto 0);
           trama_val : in  STD_LOGIC;
           tx : out  STD_LOGIC);
end tx_pc;

architecture Behavioral of tx_pc is
    Type states is (reposo, b_ini, b_trama, b_fin);
    signal state, n_state : states;
    signal index_bit, n_index_bit : integer range 0 to 7;
    signal index_trama, n_index_trama : integer range 0 to 367;
    signal tcount, n_tcount : unsigned (12 downto 0);
begin
    sinc: process(clk, reset)
    begin
```

```
        if (reset = '1') then
            state <= reposo;
            index_bit <= 0;
            index_trama <= 0;
            tcount <= (others => '0');
        elsif (rising_edge(clk)) then
            state <= n_state;
            index_bit <= n_index_bit;
            index_trama <= n_index_trama;
            tcount <= n_tcount;
        end if;
    end process;

    comb: process(state, index_bit, index_trama, tcount, trama_val)
    begin
        n_state <= state;
        n_index_bit <= index_bit;
        n_index_trama <= index_trama;
        n_tcount <= tcount;
        tx <= '1';
        case state is
            -- Espera acción para comenzar transmisión
            when reposo =>
                if (trama_val = '1') then
                    n_state <= b_ini;
                end if;
            -- Envía el bit de inicio
            when b_ini =>
                tx <= '0';
                -- 5208 * 20 ns = 104,16 us (1/9600 Baud)
                if (tcount = "1010001010111") then
                    n_state <= b_trama;
                    n_tcount <= (others => '0');
                else
                    n_tcount <= tcount + 1;
                end if;

            -- Envía los bits de información
            when b_trama =>
                tx <= trama(index_trama);
                if (tcount = "1010001010111") then
                    n_tcount <= (others => '0');
                    if (index_bit = 7) then
                        n_state <= b_fin;
                        n_index_bit <= 0;
                        if (index_trama = 367) then
                            n_index_trama <= 0;
                        else
                            n_index_trama <= index_trama + 1;
                        end if;
                    else
                        n_index_bit <=  index_bit + 1;
                        n_index_trama <= index_trama + 1;
                    end if;
                else
                    n_tcount <= tcount + 1;
                end if;

            -- Envía el bit final
            when others =>
                tx <= '1';
```

```
                if (tcount = "1010001010111") then
                    n_tcount <= (others => '0');
                    if (index_trama = 0) then
                        n_state <= reposo;
                    else
                        n_state <= b_ini;
                    end if;
                else
                    n_tcount <= tcount + 1;
                end if;
        end case;
    end process;
end Behavioral;
```

- **vibracion_tb.vhd**

```
LIBRARY ieee;
LIBRARY STD;

USE STD.TEXTIO.ALL;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;

ENTITY vibracion_tb IS
END vibracion_tb;

ARCHITECTURE behavior OF vibracion_tb IS

    COMPONENT vibracion
    PORT(
         clk : IN  std_logic;
         reset : IN  std_logic;
         V : IN  std_logic_vector(9 downto 0);
         I : IN  std_logic_vector(9 downto 0);
         SDO : IN  std_logic;
         CNV : OUT  std_logic;
         SDI : OUT  std_logic;
         SCK : OUT  std_logic;
         tx : OUT  std_logic
        );
    END COMPONENT;


   --Inputs
   signal clk : std_logic := '0';
   signal reset : std_logic := '0';
   signal V : std_logic_vector(9 downto 0) := (others => '0');
   signal I : std_logic_vector(9 downto 0) := (others => '0');
   signal SDO : std_logic := '0';

    --Outputs
   signal CNV : std_logic;
   signal SDI : std_logic;
   signal SCK : std_logic;
   signal tx : std_logic;

    --Output file
    file trama: text open write_mode is "python/vib_trama";

BEGIN
```

```
uut: vibracion PORT MAP (
       clk => clk,
       reset => reset,
       V => V,
       I => I,
       SDO => SDO,
       CNV => CNV,
       SDI => SDI,
       SCK => SCK,
       tx => tx
     );

clk <= not clk after 10 ns;


-- Proceso de estímulos de errores
estimulos: process
begin
   reset <= '1';
     wait for 100 ns;
     reset <= '0';
     wait for 5 ms;
     V(0) <= '1';
     wait for 150 us;
     V(0) <= '0';
     wait for 150 ms;
     V(0) <= '1';
     wait for 300 us;
     V(0) <= '0';
   wait for 400 ms;
     V(3) <= '1';
     V(2) <= '1';
     I(5) <= '1';
     V(8) <= '1';
     wait for 4 ms;
     V(8) <= '0';
     V(2) <= '0';
     wait for 600 ms;
     V(0) <= '1';
     wait for 4 ms;
     V(0) <= '0';
     I(5) <= '0';
     wait for 1000 ms;
     V(3) <= '0';
     I(9) <= '1';
     wait for 50 us;
     I(9) <= '0';
     wait for 500 ms;
     V(6) <= '1';
     wait for 30 ms;
     I(0) <= '1';
     wait for 5 ms;
     V(6) <= '0';
     wait for 40 ms;
     I(0) <= '0';
     wait;
end process;

-- Proceso que simula el ADC
SPI_ADC: process
```

```
   begin
        for i in 17 downto 0 loop
            wait until rising_edge(SCK);
            case i is
                when 10 to 13 =>
                    SDO <= '1';
                when others =>
                    SDO <= '0';
            end case;
        end loop;
        wait for 40 ns;
        SDO <= '0';
    end process;



   -- Proceso que recibe bytes y genera fichero de salida
    get_data: process
        variable l: line;
        variable b: bit_vector (7 downto 0) := "00000000";
    begin
        wait until falling_edge(tx);
        wait until falling_edge(clk);
        get_byte: for i in 0 to 7 loop
            wait for 5208 * 20 ns; -- 1/9600
            b(i) := to_bit(tx);
        end loop get_byte;
        wait for 5208 * 20 ns; -- 1/9600
        write(l,b);
        writeline(trama,l);
    end process;
END;
```

Anexo C: Código Python

```
### Interfaz de vibracion ###

## Modulos importados
from tkinter import *
from tkinter import ttk, messagebox
import threading, sys, serial, csv, time

## Decalaración de variables
# Variables que almacenan datos recibidos
trama = [None] * 46
trama_aux = [''] * 46
n_err = [0]*20
val_adc = 0
crc_rx = ""
n_trama_rx = 0
port_con = None
# Estadísticas internas
n_tramas = 0
n_tramas_bien = 0
n_tramas_mal = 0
# Variables de control
f_con = 0
f_ini = 0
f_tiempo = 0
# Variables de los elementos de la GUI
CC = [None]*20
CA = [None]*20
n_tramas_lab = [None]
n_tramas_bien_lab = [None]
n_tramas_mal_lab = [None]
tiempo_lab = [None]
fram_DUT = [None]*10
logname = ""
mensaje_ayuda = "PASOS A REALIZAR:\n\n1.- Elegir puerto USB deseado para
comunicar. De no encontrarse en la lista, volver a conectar" \
                " y pulsar el botón de refresco.\n\n2.- Pulsar botón de
conexión. Si la conexión es exitosa, aparecerá un mensaje " \
                "y un indicador de conexión.\n\n3.- Pulsar botón de inicio y
esperar la recepción de datos, que se verá " \
                "reflejada en los contadores y estadísticas del
programa.\n\n4.- Para finalizar sin cerrar la ventana, pulsar el botón " \
                "de parada y a continuación el de desconexión.\n\nLos datos
recogidos se almacenan en un documento CSV que se indica " \
                "en el panel de mensajes de programa."
str_con_off = "Desconectado"
str_con_on = "Conectado"
str_ini_off = "Esperando"
str_ini_on = "Recibiendo"
col_texto_green = "forest green"
col_texto_red = "firebrick3"
col_fondo_green = "DarkOliveGreen1"
col_fondo_red = "coral1"

## Funciones de los botones principales
# Pulsar el botón de conexión
def pulsa_conexion():
    global f_con, f_ini, bot_con, fram_est_con, icon_est_con, lab_est_con,
est_con, port_list, port_con, men_prog
```

```
    f_con = not f_con
    if f_con: # Conecta
        try:
            port_con = serial.Serial(port_list.get(), timeout=5)
            men_prog.insert(INSERT, '>> Conectado al puerto ' +
port_list.get() + '\n')
            bot_con.configure(image=icon_decon)
            fram_est_con.configure(background=col_fondo_green)
            icon_est_con.configure(image=icon_led_verde)
            lab_est_con.config(foreground=col_texto_green)
            est_con.set(str_con_on)
        except (OSError, serial.SerialException):
            f_con = not f_con
            men_prog.insert(INSERT, '>> Error al conectar al puerto ' +
port_list.get() + '\n')
    else: # Desconecta
        if f_ini:
            men_prog.insert(INSERT, '>> Error al desconectar\n')
            f_con = not f_con
        else:
            port_con.close()
            men_prog.insert(INSERT, '>> Desconectado\n')
            bot_con.configure(image=icon_con)
            fram_est_con.configure(background=col_fondo_red)
            icon_est_con.configure(image=icon_led_rojo)
            lab_est_con.config(foreground=col_texto_red)
            est_con.set(str_con_off)

# Pulsar el botón de inicio
def pulsa_inicio():
    global f_ini, f_con, f_tiempo, bot_ini, fram_est_ini, icon_est_ini,
lab_est_con, est_ini, men_prog
    f_ini = not f_ini
    if f_ini: # Recibe
        if not f_con:
            men_prog.insert(INSERT, '>> Error al iniciar\n')
            f_ini = not f_ini
        else:
            t1 = threading.Thread(name="Recibe", target=recibe)
            t1.start()
            f_tiempo=1
            t2 = threading.Thread(name="Crono", target=reloj)
            t2.start()
            men_prog.insert(INSERT, '>> Iniciada recepción de datos\n')
            bot_ini.configure(image=icon_stop)
            fram_est_ini.configure(background=col_fondo_green)
            icon_est_ini.configure(image=icon_led_verde)
            lab_est_ini.config(foreground=col_texto_green)
            est_ini.set(str_ini_on)
    else: # Para
        f_tiempo=0
        bot_ini.configure(image=icon_ini)
        fram_est_ini.configure(background=col_fondo_red)
        icon_est_ini.configure(image=icon_led_rojo)
        lab_est_ini.config(foreground=col_texto_red)
        est_ini.set(str_ini_off)

## Funciones auxiliares
# Función principal de la funcionalidad
def recibe():
    global trama, n_tramas, n_tramas_bien, n_tramas_mal, men_prog
    while f_ini:
```

```
        trama = port_con.read(46) # Lee 46 Bytes del puerto serie
        if(len(trama) != 46):     # Detecta si se cumple el timeout
            men_prog.insert(INSERT, '>> Conexión perdida\n')
            pulsa_inicio()
            pulsa_conexion()
            break
        save_data()     # Almacena las variables recibidas
        if comp_crc(): # Verifica el CRC
            act_cont() # Actualiza los contadores
            gen_log()  # Actualiza el fichero de salida
        n_tramas = n_tramas_bien + n_tramas_mal
        act_GUI()       # Actualiza la interfaz gráfica

# Trata los bytes recibidos y almacena los valores en variables
def save_data():
    global trama, trama_aux, n_err, val_adc, n_trama_rx
    # Convierte bytes en string binarios de 8 bits
    for i in range(46):
        trama_aux[i] = ''.join(['0']*(8-len(format(trama[i], "b"))) +
list(format(trama[i], "b")))
    # Realiza la conversión a enteros y almacena en variables
    for i in range(20):
        n_err[i] = int(trama_aux[2*i+1]+trama_aux[2*i],2)
    val_adc = int(trama_aux[42]+trama_aux[41]+trama_aux[40],2)
    n_trama_rx = int(trama_aux[44]+trama_aux[43],2)

# Comprueba el CRC
def comp_crc():
    global trama_aux, n_tramas_bien, n_tramas_mal
    crc_rx = trama_aux[45]
    crc = [0]*8
    n_crc = [0]*8
    # Bucle que implementa el mismo procedimiento de CRC
    for i,byte in enumerate(trama_aux):
        for j,bit in enumerate(reversed(list(byte))):
            if (i*8 + j) == 367:
                break
            elif (i*8 + j) > 359:
                n_crc[0] = 0 ^ crc[7]
            else:
                n_crc[0] = int(bit, 2) ^ crc[7]
            n_crc[1] = crc[0] ^ crc[7]
            n_crc[2] = crc[1] ^ crc[7]
            n_crc[3] = crc[2]
            n_crc[4] = crc[3]
            n_crc[5] = crc[4]
            n_crc[6] = crc[5]
            n_crc[7] = crc[6]
            crc = list(n_crc)
    # Se invierte para coincidir y se hace string
    crc = reversed(crc)
    crc = ''.join(str(i) for i in crc)
    # Comparación con CRC recibido
    if crc == crc_rx:
        n_tramas_bien += 1
        return 1
    else:
        n_tramas_mal += 1
        return 0

# Actualiza valores de la interfaz
```

```
def act_GUI():
    global n_tramas_lab, n_tramas_bien_lab, n_tramas_mal_lab
    n_tramas_lab.set(str(n_tramas))
    n_tramas_bien_lab.set(str(n_tramas_bien))
    n_tramas_mal_lab.set(str(n_tramas_mal))

# Actualiza los contadores de la interfaz
def act_cont():
    global CC, CA
    for i in range(10):
        CC[i].set(str(n_err[i]))
        CA[i].set(str(n_err[i+10]))
        if n_err[i] or n_err[i+10]:
            fram_DUT[i].configure(background=col_fondo_red)

# Refresca la lista de puertos detectados
def refresh_ports():
    global port_list
    port_list_val = []
    # Obtiene lista de puertos posibles por plataforma
    if sys.platform.startswith('win'): # Si estamos en Windows
        ports = ['COM%s' % (i + 1) for i in range(256)]
    else:                              # Error si estamos en otra plataforma
        raise EnvironmentError('Unsupported platform')
    # Busca los puertos a los que es posible conectarse y los lista
    for port in ports:
        try:
            s = serial.Serial(port)
            s.close()                  # Si consigue conectarse, lo guarda
            port_list_val.append(port)
        except (OSError, serial.SerialException):
            pass                       # Si no puede conectarse, no hace nada
    # Actualiza la variable de la interfaz
    port_list.configure(values = port_list_val)
    if len(port_list_val) > 0:
        port_list.set(port_list_val[0])

# Genera el mensaje de ayuda
def show_help():
    messagebox.showinfo(title="Ayuda", message=mensaje_ayuda)

# Genera ficheros de salida necesarios
def gen_log():
    global n_err, val_adc, n_trama_rx, men_prog, logname
    if n_tramas_bien == 1:
        logname = "vib" + time.strftime('%d_%m_%y_%H_%M') + ".csv"
        men_prog.insert(INSERT, '>> Generado fichero CSV\nNombre: ' + logname
+ '\n')
        with open("log/" + logname, "w", newline='') as f:
            writer = csv.writer(f)
            writer.writerow(["N° trama", "CC0", "CA0", "CC1", "CA1", "CC2",
"CA2", "CC3", "CA3", "CC4", "CA4",
                             "CC5", "CA5", "CC6", "CA6", "CC7", "CA7", "CC8",
"CA8", "CC9", "CA9", "Dato ADC"])
    with open("log/" + logname, "a", newline='') as f:
        writer = csv.writer(f)
        writer.writerow([str(n_trama_rx), str(n_err[0]), str(n_err[10]),
str(n_err[1]), str(n_err[11]), str(n_err[2]),
                         str(n_err[12]), str(n_err[3]), str(n_err[13]),
str(n_err[4]), str(n_err[14]), str(n_err[5]),
                         str(n_err[15]), str(n_err[6]), str(n_err[16]),
str(n_err[7]), str(n_err[17]), str(n_err[8]),
```

```
                        str(n_err[18]), str(n_err[9]), str(n_err[19]),
str(val_adc)])


# Función que controla el tiempo transcurrido
def reloj():
    global tiempo_lab, men_prog
    seg = 0
    while f_tiempo:
        time.sleep(1)
        seg += 1
        s_h = int(seg/3600)
        s_m = int((seg%3600)/60)
        s_s = int((seg%3600)%60)
        s_tiempo = '%02d:%02d:%02d' % (s_h, s_m, s_s)
        tiempo_lab.set(s_tiempo)
    men_prog.insert(INSERT,">> Recepción finalizada\nTiempo: " + s_tiempo +
'\n')

## Código de la interfaz gráfica
root = Tk()
root.title("Ensayo de vibración - ALTER")

# Creación de variables de Tkinter
for i in range(10):
    CC[i] = StringVar()
    CA[i] = StringVar()
n_tramas_lab = StringVar()
n_tramas_bien_lab = StringVar()
n_tramas_mal_lab = StringVar()
tiempo_lab = StringVar()

# Actualización inicial de las variables de la interfaz
act_cont()
act_GUI()
tiempo_lab.set("00:00:00")

### 1.- Barra superior: Logos y botones de control
a1 = ttk.Frame(root, width=800, height=75).grid(rowspan=3, columnspan=25,
sticky='NW', padx=5, pady=5, ipadx=5, ipady=5)

# Logo y título
icon_catedra = PhotoImage(file="icon/catedra.png").subsample(15,15)
ttk.Label(a1, image=icon_catedra).grid(column=1, row=0, columnspan=6,
rowspan=3)
icon_titulo = PhotoImage(file="icon/tit-vib.png")
ttk.Label(a1, image=icon_titulo).grid(column=6, row=1, columnspan=10,
rowspan=2)

# Separadores
ttk.Separator(a1, orient=VERTICAL).grid(column=16, row=1, sticky=NS)

# Iconos
icon_con = PhotoImage(file="icon/conn.png").subsample(7,7)
icon_decon = PhotoImage(file="icon/deconn.png").subsample(7,7)
icon_ini = PhotoImage(file="icon/play.png").subsample(7,7)
icon_stop = PhotoImage(file="icon/stop.png").subsample(7,7)
icon_refresh = PhotoImage(file="icon/refresh.png").subsample(14,14)
icon_help = PhotoImage(file="icon/help.png").subsample(14,14)
icon_led_rojo = PhotoImage(file="icon/led_rojo.png").subsample(7,7)
icon_led_verde = PhotoImage(file="icon/led_verde.png").subsample(7,7)
```

```
# Botones de acción
ttk.Label(a1, text="Conectar").grid(row=0, column=17, columnspan=2,
sticky=SW)
ttk.Label(a1, text="Iniciar").grid(row=0, column=19, columnspan=2, sticky=SW)
# Boton de conexión
bot_con = ttk.Button(a1, image=icon_con, command=pulsa_conexion)
bot_con.grid(column=17, row=1, columnspan=2, sticky=W)
# Boton de inicio
bot_ini = ttk.Button(a1, image=icon_ini, command=pulsa_inicio)
bot_ini.grid(column=19, row=1, columnspan=2, sticky=W)
# Boton de refresco
bot_ref = ttk.Button(a1, image=icon_refresh, command=refresh_ports)
bot_ref.grid(column=22, row=1, sticky=E)
# Boton de ayuda
bot_help = ttk.Button(a1, image=icon_help, command=show_help)
bot_help.grid(column=23, row=1, sticky=W)

# Lista desplegable
ttk.Label(a1, text="Puerto USB:").grid(row=0, column=22, columnspan=2,
sticky=S)
port_list = ttk.Combobox(a1, state="readonly", width=12)
port_list.grid(row=1, column=22, columnspan=2, sticky=N)
refresh_ports()

### 2.- Contador de errores cometidos durante el ensayo
a2 = ttk.LabelFrame(root, text="Errores del ensayo", width=350, height=300,
labelanchor=N)\
     .grid(row=3, rowspan=17, columnspan=15, sticky=NSEW, padx=5, pady=5,
ipadx=5, ipady=10)

# Etiquetas de DUT
fram_DUT[0] = ttk.Label(a2, text="DUT 0", font=("TkDefaultFont", 10))
fram_DUT[0].grid(row=7, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=7, column=4,
columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=7, column=9,
columnspan=2, sticky=E)
fram_DUT[1] = ttk.Label(a2, text="DUT 1", font=("TkDefaultFont", 10))
fram_DUT[1].grid(row=8, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=8, column=4,
columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=8, column=9,
columnspan=2, sticky=E)
fram_DUT[2] = ttk.Label(a2, text="DUT 2", font=("TkDefaultFont", 10))
fram_DUT[2].grid(row=9, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=9, column=4,
columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=9, column=9,
columnspan=2, sticky=E)
fram_DUT[3] = ttk.Label(a2, text="DUT 3", font=("TkDefaultFont", 10))
fram_DUT[3].grid(row=10, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=10,
column=4, columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=10,
column=9, columnspan=2, sticky=E)
fram_DUT[4] = ttk.Label(a2, text="DUT 4", font=("TkDefaultFont", 10))
fram_DUT[4].grid(row=11, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=11,
column=4, columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=11,
column=9, columnspan=2, sticky=E)
```

```
fram_DUT[5] = ttk.Label(a2, text="DUT 5", font=("TkDefaultFont", 10))
fram_DUT[5].grid(row=12, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=12,
column=4, columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=12,
column=9, columnspan=2, sticky=E)
fram_DUT[6] = ttk.Label(a2, text="DUT 6", font=("TkDefaultFont", 10))
fram_DUT[6].grid(row=13, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=13,
column=4, columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=13,
column=9, columnspan=2, sticky=E)
fram_DUT[7] = ttk.Label(a2, text="DUT 7", font=("TkDefaultFont", 10))
fram_DUT[7].grid(row=14, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=14,
column=4, columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=14,
column=9, columnspan=2, sticky=E)
fram_DUT[8] = ttk.Label(a2, text="DUT 8", font=("TkDefaultFont", 10))
fram_DUT[8].grid(row=15, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=15,
column=4, columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=15,
column=9, columnspan=2, sticky=E)
fram_DUT[9] = ttk.Label(a2, text="DUT 9", font=("TkDefaultFont", 10))
fram_DUT[9].grid(row=16, column=1, columnspan=2)
ttk.Label(a2, text="CC = ", font=("TkDefaultFont", 10)).grid(row=16,
column=4, columnspan=2, sticky=E)
ttk.Label(a2, text="CA = ", font=("TkDefaultFont", 10)).grid(row=16,
column=9, columnspan=2, sticky=E)

# Valores de errores
ttk.Entry(a2, state="readonly", textvariable=CC[0], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=7, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[0], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=7, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[1], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=8, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[1], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=8, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[2], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=9, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[2], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=9, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[3], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=10, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[3], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=10, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[4], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=11, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[4], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
```

```
    .grid(row=11, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[5], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=12, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[5], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=12, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[6], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=13, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[6], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=13, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[7], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=14, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[7], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=14, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[8], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=15, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[8], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=15, column=11, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CC[9], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=16, column=6, columnspan=2, sticky=W)
ttk.Entry(a2, state="readonly", textvariable=CA[9], justify=CENTER,
width="10", font=("TkDefaultFont", 10))\
    .grid(row=16, column=11, columnspan=2, sticky=W)

## 3.- Cuadro de datos
a3 = ttk.LabelFrame(root, text="Datos", width=200, height=300,
labelanchor=N)\
     .grid(row=3, column=15 ,rowspan=17, columnspan=5, sticky=NSEW, padx=5,
pady=5, ipadx=5, ipady=10)

# Barras de estado
fram_est_con = Frame(a1, width=100, height=20, background=col_fondo_red)
fram_est_con.grid(column=16, row=7, columnspan=3)
fram_est_ini = Frame(a1, width=100, height=20, background=col_fondo_red)
fram_est_ini.grid(column=16, row=8, columnspan=3)

# Iconos de estado
icon_est_con = ttk.Label(fram_est_con, image=icon_led_rojo)
icon_est_con.grid(column=16, row=7, sticky=W)
icon_est_ini = ttk.Label(fram_est_ini, image=icon_led_rojo)
icon_est_ini.grid(column=16, row=8, sticky=W)

# Descriptores de estado
est_ini = StringVar()
est_con = StringVar()
est_ini.set(str_ini_off)
est_con.set(str_con_off)
lab_est_con = ttk.Label(fram_est_con, textvariable=est_con, width=25,
foreground=col_texto_red)
lab_est_con.grid(column=17, row=7, columnspan=2)
lab_est_ini = ttk.Label(fram_est_ini, textvariable=est_ini, width=25,
foreground=col_texto_red)
lab_est_ini.grid(column=17, row=8, columnspan=2)
```

```
# Estadisticas
ttk.Label(root,text="Tramas recibidas").grid(row=10, column=15, columnspan=3,
sticky=E)
ttk.Label(root,text="Totales:").grid(row=11, column=16, columnspan=2,
sticky=W)
ttk.Label(root, textvariable=n_tramas_lab).grid(row=11, column=17,
columnspan=2)
ttk.Label(root,text="Correctas:").grid(row=12, column=16, columnspan=2,
sticky=W)
ttk.Label(root, textvariable=n_tramas_bien_lab,
foreground=col_texto_green).grid(row=12, column=17, columnspan=2)
ttk.Label(root,text="Incorrectas:").grid(row=13, column=16, columnspan=2,
sticky=W)
ttk.Label(root, textvariable=n_tramas_mal_lab,
foreground=col_texto_red).grid(row=13, column=17, columnspan=2)
ttk.Label(root,text="Tiempo de recepción:").grid(row=15, column=16,
columnspan=3, sticky=W)
ttk.Label(root, textvariable=tiempo_lab).grid(row=16, column=17,
columnspan=2)


## 4.- Log de mensajes de ejecución
a4 = ttk.LabelFrame(root, text="Mensajes del programa", width=250,
height=300, labelanchor=N)\
     .grid(row=3, column=20 ,rowspan=17, columnspan=5, sticky='NSWE', padx=5,
pady=5, ipadx=5, ipady=10)

men_prog = Text(a4, width=30, height=18)
men_prog.grid(row=5, column=20, rowspan=13, columnspan=5, sticky='S')

act_GUI()
root.mainloop()
```