Universidad del País Vasco Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN

SISTEMAS ELECTRÓNICOS AVANZADOS

TRABAJO FIN DE MÁSTER

Entorno de test automatizado para la verificación de microcontroladores basado en el software de National Instruments

Estudiante *Barrutia, Sanz, Aitor*
Director/Directora *Etxebarria, Egizabal, Ainhoa*
Departamento *Tecnología Electrónica*
Curso académico *2021-2022*

Bilbao, 15, 6, 2022

Resumen

Verificar la funcionalidad de las tarjetas de desarrollo a utilizar en cualquier proyecto, es fundamental para asegurar que los errores generados durante el desarrollo se deben a la programación y no a la disfunción del hardware. Con el auge de nuevas tecnologías como la RISC-V, que además son de código abierto, la verificación se convierte en más importante dado que las opciones en el mercado son muy amplias y los estándares de verificación de hardware varían mucho entre fabricantes.

Se ha implementado un entorno de test utilizando LabVIEW y TestStand para verificar los periféricos de varias tarjetas de evaluación necesarios para realizar el control de velocidad de un motor síncrono de imanes permanentes (PMSM), utilizando la técnica FOC, de control de campo orientado. Para dicho algoritmo es necesario el uso de los GPIOs de la tarjeta, generación PWM, lectura ADC, decoder y comunicaciones SPI e I2C. Se ha implementado un protocolo de comunicación serial de tamaño variable específico para el testeo de dispositivos para comunicar el PC y el dispositivo a testear utilizando UART.

Las pruebas necesarias para realizar los tests han sido programadas en dos tarjetas de desarrollo diferentes, una con arquitectura RISC-V del fabricante GigaDevice y una Delfino de Texas Instruments.

Se ha probado el software LabVIEW NXG para la implementación de las pruebas, pero se han obtenido resultados muy negativos, dado los errores que tiene y que National Instruments no va a sacar más versiones.

Ambas tarjetas de desarrollo han superado el entorno de verificación exitosamente, concluyendo que son aptas para el control del motor. Además, se ha programado el control en la tarjeta que contiene la RISC-V, y se ha comprobado que puede controlar el motor.

Palabras clave: RISC-V, Delfino, Entorno de test, Verificación Hardware

Laburpena

Garapen txartelen funtzionalitatea berifikatzea ezinbestekoa da, proiektu batean gertatu daitezkeen akatsak programazioagatik sortuak direla eta hardwarraren disfunzioaren erru ez direla zihurtatzeko. RISC-Va bezalako kode irekiko teknologia berrien gorakadarekin gainera, berifikazioa are garrantzitsuagoa bihurtzen da, merkatuko aukerak asko zabaltzen direlako eta ekoizle desberdinek egindako probak oso desberdinak izan daitezkeelako.

LabVIEW eta TestStand softwareak erabiliz, test ingurune bat implementatu da iman permanenteko motor sinkrono baten kontrolerako beharrezkoak diren funtzionalitateak berifikatzeko, FOC teknika erabiliz. Kontrol horren algoritmoa implementazeko GPIOen erabilera, ADCarena, PWMarena, decoder batena, eta SPI eta I2C interfazeak beharrezkoak diren. Ingurunea exekutatu ahal izateko tamaina aldakorreko komunikazio protokolo bat garatu da, PCa eta testeatuko diren gailuak UART bidez komunikatzeko.

Testak exekutatzeko beharrezkoak diren probak inplementatzeko bi garapen txartel desberdin erabili dira, bat GigaDevice ekoizlearena eta RISC-V arkitekturaduna, bestea, aldiz, Texas Instruments-en Delfino bat.

Probak inplementatzeko orduan LabVIEW NXG softwarea probatu da, baina lortutako emaitzak ez dira onak izan, akats asko baititu eta gainera National Instruments-ek bertsio berri gehiago aterako ez baititu.

Erabilitako bi garapen txartelek arazo gabe gainditu dute berifikazioa. Gainera, RISC-V a duen txartelean motorraren kontrola implementatu da, eta motorra arazorik gabe kontrolatzen duela ikusi da.

Gako hitzak: RISC-V, Delfino, Test ingurunea, Hardware berifikazioa

Abstract

Functional verification of the development cards that are going to be used in a project, is fundamental to ensure that the errors that could occur during the development of the project are caused by the programming and not by a hardware malfunction. With the boom of new technologies as the RISC-V in the last years, that much are also open source, lots of manufacturers have been developing their own products based on that technology. Quality control may vary a lot between manufacturers, so hardware verification gains even more importance.

A test environment has been implemented using LabVIEW and TestStand for the verification of the interfaces needed for a control of a permanent magnet synchronous motor using the Field Oriented Control (FOC) technique. Those interfaces are the GPIOs of the card, the ADC, PWM generators, a decoder and SPI and I2C communications. A unit testing specific variable length communication protocol has been used for the communication between the PC and the device under test (DUT).

The tests needed to execute the environment have been implemented in two different development cards, of the them is a GigaDevice card that uses RISC-V architecture, and the other one is a Texas Instruments Delfino.

National Instruments new hardware LabVIEW NXG has been tested for developing the environment, but the results obtained have been unfavourable, given the amount of bugs that the software has and the fact that Nationaln Instruments is not going to release more versions of the software.

Both of the development cards have sucesfully surpassed the verification environment without any issues. In addittion, the FOC control has been implemented on the RISC-V, and it has been verifyed that the board can control the motor correctly.

Keywords: Risc-V, Delfino, Test environment, Hardware verification

Índice general

Índice de figuras

Índice de tablas

1 | Introducción y Objetivos

1.1 | Introducción

El uso de las pruebas automatizadas durante todo el ciclo de desarrollo de diferentes productos permite reducir el tiempo y coste del desarrollo, además de garantizar un mayor nivel de calidad. En el proceso industrial de desarrollo de un producto, el coste producido por un cambio aumenta exponencialmente a medida que avanzamos en el ciclo de desarrollo del producto. Es por eso que detectar posibles errores en la fase de diseño evita errores en la producción, lo que conlleva una disminución muy grande del coste de producción [1].

Para evitar esos errores se utilizan entornos de test automáticos, los cuales sirven para ejecutar una serie de pruebas clave que permitan verificar la funcionalidad del prototipo, o detectar errores a tiempo, en un modo automático [2].

Ikerlan posee una maqueta de control de motores síncronos de imanes permanentes (PMSM) llamada *back to back*. En dicha maqueta una Delfino controla la velocidad de uno de los motores utilizando control de campo orientado (FOC), y controla también un segundo motor para generar una carga al primero. Dado el tremendo auge que está teniendo RISC-V en los últimos años, y las buena previsión que tiene para los que van a venir, Ikerlan quiere integrarse más con dicha tecnología, empezando a hacer uso de ella en diferentes proyectos.

Uno de esos proyectos consiste en realizar el control de los motores de la maqueta *back to back*, sustituyendo la Delfino por una tarjeta de desarrollo basada en RISC-V. Para ello, una estudiante realizó una investigación de mercado, y encontró que al ser de uso libre, hay una gran cantidad de tarjetas en el mercado, las cuales a veces no poseen mucha documentación, o ofrecen pocas especificaciones de sus periféricos.

En este proyecto se va implementar un entorno de test automático para verificar la funcionalidad de dos microcontroladores para implementar dicho algoritmo de control. Para ello se ha implementado un protocolo de comunicación de tamaño variable y los tests necesarias utilizando LabVIEW, y después se han automatizado utilizando TestStand.

También se ha estudiado la viabilidad de utilizar el nuevo software de National Instruments, LabVIEW NXG para la implementación de estos tests.

A continuación se detallan más en profundidad los principales aspectos trabajados.

1.1.1 | RISC-V

El proceso de verificación es crítico cuando se trata de desarrollos que utilicen nuevas tecnologías, como es el caso de RISC-V. La arquitectura de conjunto de instrucciones RISC-V, además de ser de libre uso y código abierto, está diseñada para ser muy flexible, lo que la hace compatible con una gran variedad de dispositivos y por tanto es usada para muchos tipos de propósitos.

Al ser de uso libre, permite crear plataformas de hardware, modificar y mejorar el software, etc [3]. Por tanto, cualquier fabricante puede diseñar y poner en el mercado una tarjeta basada en RISC-V, sin seguir los procedimientos de calidad que ofrecen las empresas grandes que producen sus propios micros. Esto permite al comprador de dichas tarjetas utilizar la potencia de esos micros con un precio reducido, pero con la desventaja de que la calidad de fabricación puede ser inferior, y pueden tener errores.

Esto hace que el testeo automático de los periféricos y funcionalidades del micro se convierta en una herramienta imprescindible en el desarrollo del producto.

Aun así, lo mismo ocurre con tecnologías más maduras como las Delfino de Texas Instruments, dado que en el proceso de producción de cualquier microcontrolador es posible que ocurran errores, y es fundamental verificar las diferentes funcionalidades del hardware que se va a utilizar en la aplicación a desarrollar.

Existen diferentes softwares para la generación de test automatizado. Una opción muy usada y disponible comercialmente es TestStand, software de NI (National Instruments) capaz de generar secuencias de test a partir de diferentes lenguajes de programación como LabVIEW, Python o C++, por ejemplo. Aun así, el mayor rendimiento y compatibilidad se obtienen utilizando el software de la casa, ya sea LabVIEW o LabVIEW NXG (Next Generation).

1.1.2 | TestStand

Se ha visto conveniente la idea de realizar un entorno de test automatizado, con el cual una vez programada la tarjeta para realizar los tests necesarios, se puedan verificar el funcionamiento de los periféricos y los drivers de dicha tarjeta.

Este proyecto se ha centrado en verificar la funcionalidad de los periféricos necesarios para la implementación de la maqueta, pero con la idea de poder ser fácilmente ampliable a nuevos tests en el futuro. Para ello se ha utilizado TestStand, software de National Instruments el cual

permite generar tests en diferentes lenguajes de programación, como pueden ser LabVIEW, C, o Python por ejemplo y añadir diferentes estructuras de control como bucles *for*. Permite dotar a los tests con parámetros modificables antes de la ejecución, para realizar así una gran variedad de pruebas. Además permite ejecutar el entorno varias veces seguidas, guardando los resultados de cada dispositivo a testear, con un número de identificación.

De este modo, el entorno de TestStand permite reemplazar el dispositivo a testear y ejecutar las pruebas para un nuevo dispositivo. Esto es realmente útil en varios casos. Por una parte, como las tarjetas basadas en RISC-V tienen un precio reducido, es habitual en las empresas comprar más de una, por si alguna fallase. Este entorno nos permite que una vez que se haya cargado el código en todas, ejecutar las pruebas en todas de una manera automática y ordenada.

Por otra parte, como la tecnología RISC-V es de código abierto y uso libre, es probable que en un futuro cercano Ikerlan empiece a desarrollar y fabricar sus propias placas basadas en RISC-V, por lo que el entorno permite verificar que los periféricos funcionan correctamente, que no ha habido ningún error con los drivers y que no ha habido errores en el diseño Hardware de la tarjeta. Además, en este caso el número de placas fabricadas sería grande, para reducir costos, por lo que la opción de ejecutar el entorno en un modo secuencial para toda la producción es una ventaja muy grande, la cual reduciría enormemente el tiempo de verificación.

Se ha utilizado LabVIEW para programar las pruebas individuales que verificarán las interfaces y se han utilizado las estructuras de control de TestStand para ejecutar el mismo VI con diferentes parámetros de una forma automática.

1.1.3 | LabVIEW NXG

Estos últimos años LabVIEW ha estado desarrollando una nueva familia de su software, la versión NXG, con el objetivo de renovar su interfaz gráfica, y obtener así un flujo de trabajo más optimizado, capaz de ofrecer a gente sin grandes conocimientos en programación la capacidad de generar programas a través de su interfaz gráfica. La versión estándar de LabVIEW se creó con el mismo fin, pero dadas las pocas modificaciones que ha tenido durante sus 30 años de vida, el entorno ha quedado algo anticuado, dificultando así el comienzo de aprendizaje a personas sin un previo conocimiento de programación. LabVIEW NXG pretende hacer que la experiencia del usuario sea más agradable, facilitando la búsqueda de bloques y su configuración, brindando al usuario con tutoriales, cambiando el aspecto de las ventanas, etc.

TestStand permite ejecutar sus tests utilizando LabVIEW NXG, por lo que tras implementar el entorno, se migró el proyecto de LabVIEW a LabVIEW NXG, para realizar pruebas y comparar

el funcionamiento entre ambas versiones y las posibles ventajas que podría ofrecer LabVIEW NXG.

Aun así, este proceso ha sido tedioso, dado que la versión NXG no está bien optimizada y tiene muchos bugs, lo que hace que su utilización no sea nada positiva para el usuario. Además, NI ha anunciado que la versión 5.1 de LabVIEW NXG va a ser la última y que no van a sacar nuevas actualizaciones. En un anuncio escrito en el foro de NI [4], la empresa dice que aunque la recepción de NXG ha sido positiva, han recibido muchos mensajes en los que los usuarios mostraban sus preocupaciones por lo que supone migrar su trabajo a un entorno LabVIEW completamente nuevo.

Aunque se haya conseguido migrar el proyecto de LabVIEW a LabVIEW NXG, la ejecución del entorno ha sido muy negativa dada la gran cantidad de fallos que ocurren. Se analizarán más a fondo en una sección específica de LabVIEW NXG los problemas que han surgido.

1.1.4 | Protocolo de comunicación

Para el desarrollo de los tests, es necesario que el PC y la tarjeta de desarrollo puedan comunicarse. Esta comunicación permite que el PC envíe los datos necesarios para la realización de cada prueba a la tarjeta, y un comando identificador de la prueba a realizar. De este modo se consigue que el programa de la tarjeta se compile y cargue una sola vez antes del inicio de la prueba, y después el PC y la tarjeta puedan seguir el flujo de las pruebas comunicándose entre ellas.

La comunicación se va a ejecutar a través de UART y se va a implementar un protocolo específico diseñado para el testeo de dispositivos, de longitud de mensaje variable. Utilizar un protocolo de mensaje variable es muy útil en estos casos, dado que permite enviar mensajes cortos cuando la cantidad de datos necesarios para el tests son reducidos, y enviar una gran cantidad de datos, cuando la prueba así lo necesita. De esta manera se gana velocidad en los casos que la cantidad de datos en pequeña, sin perder la capacidad de realizar pruebas que requieren de muchos datos, ganando versatilidad.

1.2 | Objetivos

Los objetivos a desarrollar a lo largo del proyecto son los siguientes:

- Continuar con la implementación de un escenario automatizado de verificación basado en LabVIEW, para verificar la funcionalidad de diferentes interfaces de una tarjeta de

evaluación basada en RISC-V, para poder ser usado en el control de campo orientativo de motores síncronos de imanes permanentes.

- Implementar el entorno de test utilizando TestStand, para conseguir un entorno de verificación adaptable y fácil de ejecutar.
- Implementar un protocolo de comunicación UART de tamaño variable que se utilizará para comunicar por puerto serie el PC con la placa a testear.
- Programar el protocolo de comunicación y los tests individuales utilizando LabVIEW.
- Programar las interfaces y los test realizados en la RISC-V utilizando la herramienta IAR Embedded Workkbench for RISC-V.
- Implementar el escenario de test en una tarjeta de evaluación Delfino de la empresa Texas Instruments utilizando el software de desarrollo de TI (Code Composer Studio).

1.3 Beneficios

Desarrollar el entorno de test otorga los siguientes beneficios:

- Automatiza la verificación de los dispositivos hardware que se quieran utilizar en diferentes proyectos, permitiendo la verificación y la comparación de resultados entre diferentes unidades a testear, gracias a los recursos de TestStand y su generador de reportes.
- El entorno puede estar en constante desarrollo, añadiendo nuevas pruebas, para así ampliar de manera sencilla la versatilidad del entorno. Cuanto mayor es el número de pruebas implementadas, más útil es el entorno en comparación con el testeo no automatizado.
- Permite seleccionar rápidamente un grupo de pruebas a realizar y saltar las pruebas no necesarias, permitiendo así crear un entorno general, en el cual se pueden seleccionar fácilmente las pruebas necesarias para la verificación de diferentes funcionalidades dependiendo de los requisitos del proyecto.

2 | Diseño del entorno de prueba

El entorno de pruebas va a consistir en la verificación de las interfaces de una tarjeta de evaluación para el control de campo orientado de un motor PMSM. Los motores PMSM se utilizan generalmente junto con un driver, el cual alimenta las tres fases del motor. Además, el driver tiene unos amplificadores que convierten la corriente de las fases en tensión, para que así puedan ser leídas por la tarjeta. De tal modo, la tarjeta puede leer dicha tensión con sus ADCs, y convertirla a corriente de fase. Para controlar la corriente que se genera en cada fase, el driver recibe 3 señales PWM, una para cada fase, y genera corriente dependiendo del Duty Cycle de cada PWM.

El algoritmo de control de campo orientado, también llamado control vectorial, consiste en realizar las transformada de Clark y Park a las 3 fases del motor, para así cambiar de un eje estacionario, a un eje rotacional que rota síncrono al motor. Así, se transforman las corrientes de las 3 fases del motor a 2 corrientes estacionarias ortogonales. La primera es la corriente de par, la cual representa el par que generan las 3 fases en el motor. La segunda es la corriente de campo, la cual representa el flujo del estator del motor. El control de campo orientado consiste en controlar la corriente de flujo para que sea nula, haciendo el control más eficiente. Esto pasa porque el flujo es perpendicular al torque, por lo que no genera fuerza de rotación al motor. Por otro lado, se puede controlar la corriente de torque para controlar la fuerza que se ejerce sobre el motor. Normalmente se utiliza un controlador PI para cada una de las corrientes, y otro controlador externo para controlar la referencia de la corriente de torque. La referencia de la corriente de flujo es siempre 0.

Tras obtener las corrientes de flujo y torque, se aplican las transformadas inversas de Clark y Park, para transformar las 2 corrientes controladas a las 3 señales PWM que se envían al driver para alimentar cada una de las fases.

Además, la tarjeta debe configurar el driver antes de empezar a controlar el motor, cambiando parámetros esenciales como la ganancia de los amplificadores de corriente o el offset, el modo de funcionamiento del driver, o parámetros que controlan la alimentación del driver, como el trigger de undervoltage por ejemplo. Dicha comunicación se suele realizar por I2C o SPI dependiendo del driver, en el caso de la maqueta *back to back* se utiliza I2C.

Para habilitar el driver se utiliza una señal digital, y además el driver tiene una salida digital

para señalar que ha ocurrido un error. Si un error ocurre, se pueden leer los registros de error utilizando I2C.

Por tanto, se deben verificar el funcionamiento de las siguientes interfaces:

- Entradas y salidas de uso general para encender el driver y leer la señal de Error.
- Generación de PWM para el control de las fases del driver.
- Lectura ADC para leer las corrientes de fase.
- Lectura de Decoder para computar la posición y velocidad del motor.
- Comunicaciones I2C y SPI. Aunque el driver utilice I2C, hay muchos drivers que utilizan SPI, por lo que se van a implementar ambos test.

En un proyecto anterior [5] realizado por otro estudiante se habían implementado las pruebas de entradas y salidas digitales, PWM y ADC. En este proyecto se han implementado la lectura del encoder, tanto de posición como de velocidad, y las comunicaciones I2C y SPI, tanto la lectura como la escritura para ambos protocolos.

Se ejecutará el entorno completo para ambos DUTs, tanto la parte nueva como las pruebas anteriormente implementadas. Además, dado que la adaptador I2C/SPI utilizada tiene GPIOs de propósito general, se han modificado las pruebas posibles para ejecutarse en el adaptador.

2.1 Flujo de las pruebas

Todas las pruebas comienzan con el PC enviando un mensaje al dispositivo bajo testeo (DUT). Dicho mensaje contiene el comando del test y datos necesarios para la prueba, como se analizará posteriormente en el protocolo de comunicación.

Posteriormente, el DUT enviará un mensaje de confirmación al PC, para informarle que está listo para realizar la prueba. Este mensaje no contiene ningún dato, se utiliza para que el DUT pueda realizar las preparaciones necesarias para la prueba. En algunos tests este mensaje no es necesario por lo que no se envía.

Después se realiza la prueba, generando o adquiriendo los datos necesarios desde el PC.

Finalmente, el DUT envía un último mensaje al PC. Este mensaje contiene los resultados de la prueba, en caso de que la prueba así lo requiera.

El flujo de la prueba se puede observar en la Figura 2.1.

Figura 2.1: Flujo de las pruebas

2.2 | Descripción de las pruebas

A continuación se describen las nuevas pruebas implementadas.

2.2.1 | Lectura de posición de encoder

Esta prueba se realiza para verificar la capacidad del decoder para leer los pulsos generados por el encoder. Para tener total control de la cantidad de pulsos generados y así verificar la precisión de la interfaz de la tarjeta, se va a realizar una simulación del encoder.

La prueba va a consistir en una simulación de encoder a velocidad constante con duración determinada. La duración se enviará a través del protocolo UART para que la tarjeta pueda computar el final de la prueba.

Se realizarán más de una prueba a distintas velocidades, aumentando la velocidad en cada prueba, para verificar que el decoder es capaz de leer la señal de encoder simulada con frecuencia equivalente a la velocidad máxima del motor. En este caso el motor cuenta con un encoder de 1000 pulsos por vuelta, los cuales se leerán en codificación x4.

Es importante realizar las pruebas en todo el rango de funcionamiento del motor, y sobre todo en las frecuencias altas donde el decoder puede tener más problemas. La velocidad máxima del motor es de 6000 rpm, lo que equivale a una frecuencia de 100KHz en cada canal del encoder a velocidad máxima.

$$f = \frac{1000\ pulsos}{1\ revolucion} * \frac{6000\ revoluciones}{1\ minuto} * \frac{1\ minuto}{60\ segundos} = 100KHz$$

Además, para verificar que el decoder funciona en ambos sentidos de giro se realizarán pruebas en ambas direcciones.

Para que la prueba pueda ser utilizada en diferentes motores y así aumentar la funcionalidad del entorno de test, se ha diseñado el test para que el resultado de la posición sea absoluta, sin tener en cuenta el reset de las vueltas, es decir, el número total de pulsos recibidos en la prueba.

Se ha determinado que el resultado de la prueba será positivo si la lectura se realiza con un error inferior al 1 %.

2.2.2 | Lectura de velocidad de encoder

En esta prueba se va a verificar la capacidad de computar la velocidad del encoder. Para el cálculo de la velocidad se ha optado por medir la diferencia de pulsos entre dos periodos de muestreo de periodo constante, de este modo, la velocidad será la división entre los pulsos adquiridos y el periodo de muestreo. Se ha utilizado un periodo de muestreo de 1ms, razonable para el algoritmo de control utilizado.

Como en la prueba anterior, la velocidad se irá aumentando en cada prueba y se realizarán pruebas en ambas direcciones de giro.

En este caso no es necesario enviar la duración de la prueba, dado que el DUT dará la prueba por finalizada tras computar la velocidad en el segundo periodo de muestreo, como se observará más en profundidad posteriormente, en la sección de programación hardware.

Como en la prueba anterior, el test se dará por exitoso si el error es menor al 1 %. El error de cuantización para el periodo utilizado es de $\pm 250 pulsos/segundo$, lo que habrá que tener en cuenta a la hora de configurar los parámetros de la prueba.

2.2.3 | Lectura I2C y SPI

Se han definido como lectura I2C y SPI las pruebas en las que el DUT lee utilizando dichas interfaces, por tanto, en los tests de lectura el PC es el que envía los datos.

Para verificar que la DUT lee los datos correctamente, se va a seguir el siguiente procedimiento:

- El PC envía el mensaje de inicio de test.
- La DUT envía un mensaje de confirmación cuando esté preparado para la lectura.
- El PC escribe los datos de la prueba por I2C o SPI, dependiendo de la prueba.

- El DUT envía el mensaje de fin de prueba. Utiliza el mensaje para enviar los datos leídos.

De tal forma, el DUT lee los datos por la interfaz seleecionada, y los envía por UART, para que el PC compruebe que los datos enviados y recibidos sean iguales.

La longitud del mensaje escrito es fácilmente modificable como se explicará en la implementación en TestStand.

Para que la prueba sea exitosa todos los datos deben coincidir.

2.2.4 | Escritura I2C y SPI

En este caso, el DUT es el que escribe por I2C o SPI, y el PC es el que recibe el mensaje.

Para poder comprobar que el DUT escribe los datos correctamente es necesario conocer dichos datos. Para ello, se utiliza el mensaje de inicio de test, de la siguiente manera, como se puede observar en la Figura 2.2:

- El PC envía el mensaje de inicio de test, con los datos a utilizar en la prueba.
- La DUT envía un mensaje de confirmación cuando esté preparado para la escritura.
- El DUT escribe los datos de la prueba por I2C o SPI, dependiendo de la prueba, utilizando los datos que anteriormente le ha enviado el PC.
- El DUT envía el mensaje de fin de prueba. En este caso el mensaje es solo una confirmación de que la prueba ha finalizado, no se envía ningún dato.

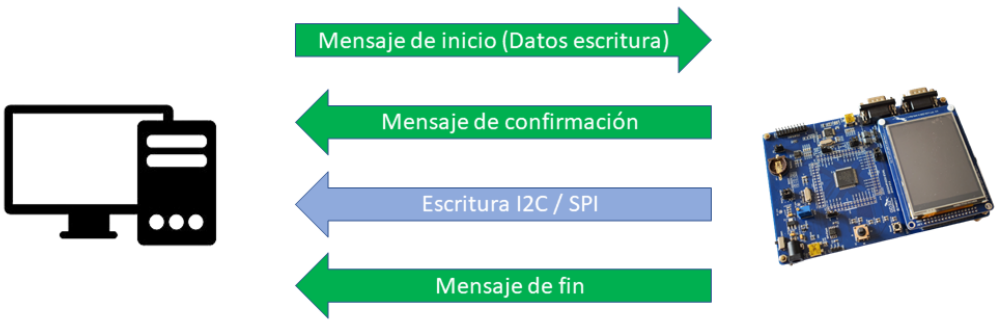

Figura 2.2: Flujo de las pruebas de escritura I2C y SPI

De este modo podemos los datos que el DUT debe escribir son conocidos de antemano, por lo que se pueden comparar al final de la prueba con los recibidos por el PC.

Al igual que en la prueba anterior, todos los datos deben coincidir para dar la prueba como exitosa.

2.2.5 | Escritura y lectura simultánea SPI

A diferencia del I2C, el SPI utiliza canales separados para la lectura y la escritura, por lo que se pueden realizar ambas simultáneamente. Se va a implementar una prueba para la verificación de dicha funcionalidad.

Como en la prueba anterior, los datos de escritura del DUT tiene que ser conocidos para después poder verificarlos.

El diseño de la prueba es el siguiente:

- El PC envía el mensaje de inicio de test, con los datos que el DUT escribirá.
- La DUT envía un mensaje de confirmación cuando esté preparado para la transmisión.
- El DUT escribe los datos de la prueba por SPI, utilizando los datos que anteriormente le ha enviado el PC, y el PC envía otros datos simultáneamente.
- El DUT envía el mensaje de fin de prueba, que contiene los datos leídos en la DUT.

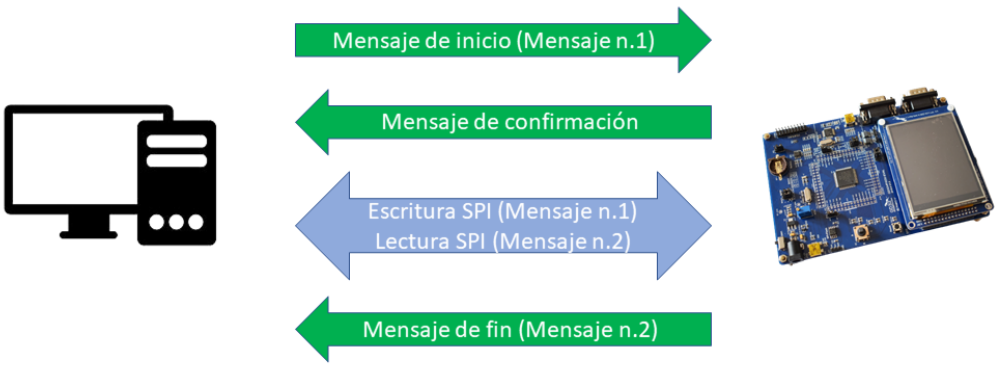

Figura 2.3: Flujo de las pruebas de escritura y lectura SPI

Por tanto, en este caso se realizan 2 comparaciones. Por un lado se comprueban los datos que el PC ha enviado por UART en el mensaje de inicio, con los que el DUT ha enviado por SPI. Por

otro lado, se comprueban los datos que el PC ha escrito por SPI con los que el PC ha recibido por UART en el mensaje de fin de prueba.

En la Figura 2.3 se puede observar el flujo de la prueba y la transmisión de ambos mensajes.

3 | Montaje Hardware

Para implementar el entorno de verificación se ha utilizado el siguiente Hardware:

- Un PC para desarrollar todo el software y ejecutar el entorno de verificación.
- Tarjetas de desarrollo que van a ser verificadas con el entorno. Se han utilizado 2 tarjetas diferentes, una basada en RISC-V y una Delfino.
- Plataforma de adquisición y generación cDAQ, con los módulos necesarios, para la generación de las señales necesarias para las pruebas, como la simulación del encoder.
- Adaptador USB-Serial para comunicar el PC con la tarjeta de desarrollo bajo test.
- Interfaz de comunicación I2C y SPI. La interfaz se comunica al PC por USB, para así poder comunicarse con las interfaces SPI e I2C de la DUT.

En la Figura 3.1 se puede observar el montaje completo. Aunque en la foto aparezca la placa basada en RISC-V, la DUT puede ser cualquier dispositivo a verificar. La Delfino también ocupa el mismo lugar.

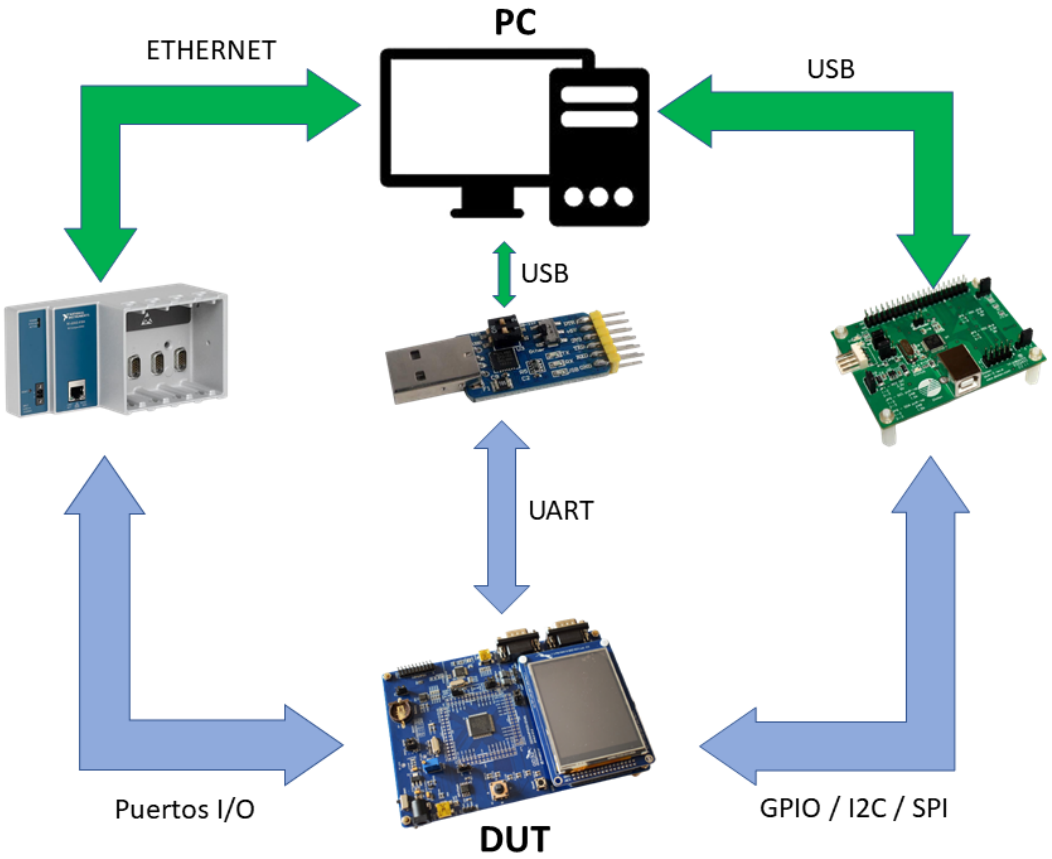

Figura 3.1: Montaje Hardware del entorno de verificación

3.1 Plataforma de adquisición de datos

Para adquirir y generar los datos necesarios para los tests se ha utilizado una CompactDAQ de National Instruments. La cDAQ utiliza módulos intercambiables, con los que podemos obtener y generar señales de diferentes tipos dependiendo de la aplicación objetivo. En este caso se ha utilizado una cDAQ-9184 [6] que consta de 4 ranuras depara módulos y una conexión ethernet para la comunicación e intercambio de datos con el PC. El chasis también cuenta con 4 contadores integrados de 32 bits, con los que se pueden adquirir y generar pulsos de tren. A continuación se describen los módulos utilizados en este trabajo.

Figura 3.2: Chasis cDAQ-9184

3.1.1 | Módulo NI 9401

El módulo NI 9401 [7] cuenta con 8 canales de entrada/salida digitales configurables. Pueden ser configurados todos como output, todos en input o 4 en input y 4 en output. Los canales se conectan a las entradas o salidas de los contadores internos de la cDAQ que se pueden utilizar tanto para adquisición como generación de PWM y trenes de pulsos. Como disponíamos de dos módulos de este tipo y ranuras de sobra se han utilizado ambos. Uno para la adquisición de la PWM generada en el DUT y el otro para la simulación del encoder.

Figura 3.3: Módulo NI 9401

3.1.2 | Módulo NI 9263

El módulo NI 9263 [8] cuenta con 4 salidas analógicas de actualización simultánea, con rápida velocidad de respuesta y 100 kS/s/canal. Cada canal puede generar voltajes de $\pm$ 10 V con una resolución de 16 bits. Se ha utilizado dicho módulo para testear el ADC del DUT, prueba que ya estaba programada.

Figura 3.4: Módulo NI 9263

3.2 Adaptador USB-Serial

Para la comunicación entre el PC y la tarjeta de desarrollo se utiliza la comunicación UART. La tarjeta de desarrollo cuenta con dos puertos serie RS-232, por los cuales podemos configurar la salida de la señal UART. En el caso del PC, dado que se está utilizando un portátil moderno, no cuenta con ningún puerto RS-232, por lo que se utilizará un puerto USB. Por tanto, se necesita un adaptador de USB a RS-232. Se ha utilizado el adaptador de la marca Dollatek que se observa en la Figura 3.5, el cual se conecta al USB del PC y se puede conectar a los pines del puerto RS-232 de la DUT. El adaptador utiliza el chip CP2102 de Silabs para convertir la señal USB a UART.

Para la conexión UART que estamos utilizando solo son necesarios 3 pines de los 9 del puerto RS-232; Gnd, Rx (pin de recepción) y Tx (pin de transmisión). Se han de conectar las tierras de ambos dispositivos, y los pines de datos de ambos dispositivos intercambiados, es decir, el pin Tx de la DUT con el Rx del PC y el pin RX de la DUT al Tx del PC.

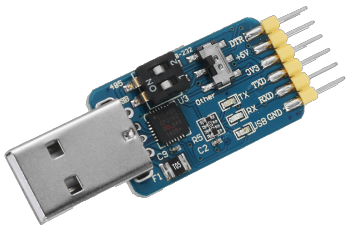

Figura 3.5: Adaptador USB-232

La conexión UART se ha configurado a un baudrate de 115200, 8 bits de datos, y sin bit de paridad.

3.3 Interfaz de comunicación SPI e I2C

Dado que la cDAQ no tiene módulos que ofrezcan comunicación SPI o I2C se ha utilizado el adaptador Diolan Dln-2 [9]. Este adaptador cuenta con 32 pines I/O de propósito general, interfaz USB-I2C e interfaz USB-SPI.

En el proyecto anterior se utilizaba la cDAQ para la generación de las entradas y salidas digitales de propósito general.En este caso, dado que el adaptador cuenta con pines de propósito general, se han utilizado dichos pines para las pruebas de entradas y salidas digitales. Se ha tomado esta decisión para comprobar que se pueden realizar las pruebas de la misma forma con el adaptador, el cual tiene un precio muy reducido en comparación con la cDAQ y sus módulos.

Solo se han modificado las pruebas de la entrada digital y la salida digital. La adquisición de PWM no se puede hacer con esta interfaz porque no cuenta con contadores, por lo que esa prueba se mantiene en la cDAQ como ya se ha mencionado en su apartado. Por otra parte, la interfaz puede generar PWM, pero no puede sincronizar diferentes PWMs, cosa que es necesaria para la generación de la simulación del encoder como se verá en la implementación de LabVIEW de la prueba. Es por eso que para las pruebas del encoder también se ha utilizado la cDAQ.

La interfaz USB-I2C soporta frecuencias de 1KHz hasta 1MHz. La interfaz USB-SPI soporta frecuencias de hasta 18MHz. Ambas de estas interfaces funcionan en modo maestro. El propio fabricante pone a disposición librerías para poder programar la interfaz tanto en C/C++ como en LabView.

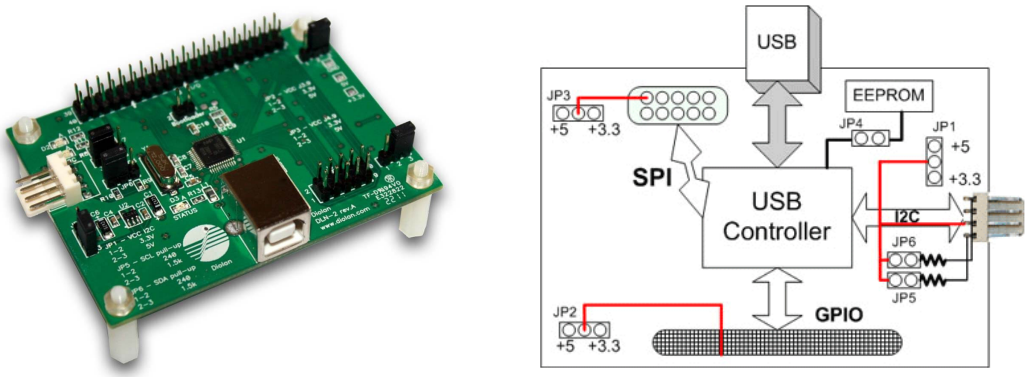

Figura 3.6: Módulo Diolan Dln-2

3.4 | Placa de evaluación GD32VF103V-EVAL basada en RISC-V

Uno de los dispositivos a testear ha sido una placa de evaluación GD32VF103V-EVAL de la marca GigaDevice [10]. Dicha placa cuenta con un microprocesador con arquitectura RISC-V y un gran número de pines conectados a diferentes periféricos integrados.Los periféricos utilizadas han sido los GPIO, el ADC, el PWM, el Decoder y las interfaces de comunicación I2C y SPI.

La placa de evaluación cuenta con un microprocesador GD32VF103VBT6 con una frecuencia de 108 MHZ y bajo consumo.

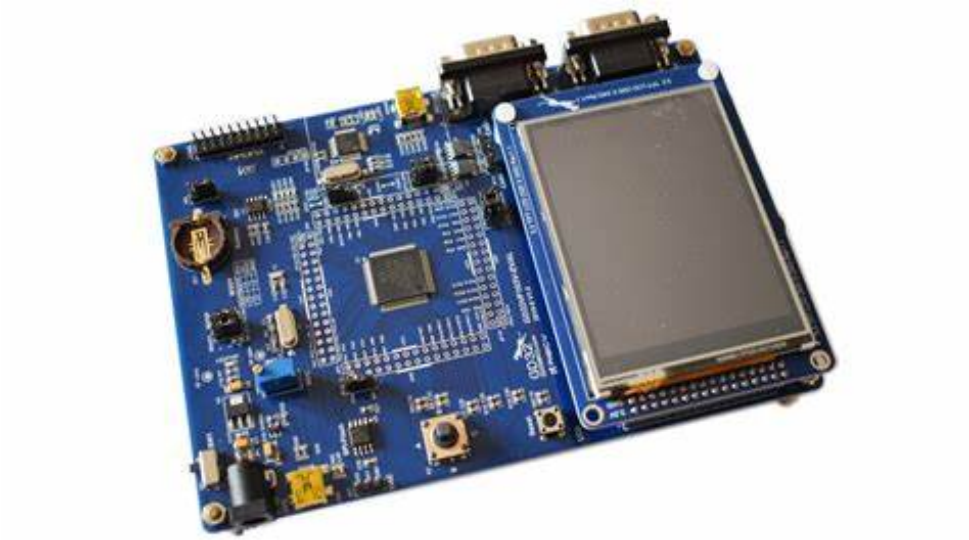

Figura 3.7: Placa GD32VF103V-EVAL

3.4.1 | Conexiones GD32VF103V-EVAL

Para la conexión de los periféricos de la placa GD32VF103V-EVAL con los dispositivos conectados al PC, se han utilizado cables dupont. En la Tabla 3.1 se pueden observar las conexiones entre la placa y los diferentes módulos instalados en la cDAQ. El módulo 1 se ha utilizado para generar la simulación del encoder. El módulo 2 para generar las señales analógicas que van al ADC de la placa. El módulo 4 para obtener las PWM generadas por la placa. En la Tabla 3.2 se pueden observar las conexiones entre la placa y los GPIO de la Diolan, que se han utilizado como salidas y entradas digitales. En la Tabla 3.3 se pueden observar las conexiones entre la placa y las interfaces de comunicación SPI e I2C de el adaptador Diolan Dln-2.

Módulo 1 - 9401(Digital Out)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
DIO6	23	PA0
DIO7	25	PA1
COM	13	GND
Módulo 2 - 9263(Analog In)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
AO0	0	PC1
COM	9	GND
Módulo 4 - 9401(Digital In)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
DIO4	20	PA11 (Timer0_CH3)
COM	13	GND

Tabla 3.1: Conexiones de la RISC-V y los módulos de la cDAQ

DIOLAN GPIO (In/Out)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
C0 (Out)	13 (GPIO 16)	PE0
C1 (Out)	14 (GPIO 17)	PE1
C2 (Out)	15 (GPIO 18)	PE2
C3 (Out)	16 (GPIO19)	PE3
B0 (In)	25 (GPIO 8)	PD0
B1 (In)	26 (GPIO 9)	PD1
B2 (In)	27 (GPIO 10)	PD2
B3 (In)	28 (GPIO 11)	PD3
GND	11	GND

Tabla 3.2: Conexiones de la RISC-V y los GPIO del adaptador Diolan

DIOLAN I2C		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
I2C_SDA	I2C Pin 1	PB7
I2C_SCL	I2C Pin 4	PB6
GND	I2C Pin 2	GND
DIOLAN SPI		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
SCK	SPI Pin 7	PA5
MISO	SPI Pin 3	PA6
MOSI	SPI Pin 1	PA7
NSS0	SPI Pin 5	PA4
GND	SPI Pin 11	GND

Tabla 3.3: Conexiones de la RISC-V y las interfaces SPI/I2C del adaptador Diolan

3.5 Delfino

Además de la RISC-V, también se ha usado el entorno de verificación para validar una tarjeta Delfino de Texas Instruments, para poder comparar las ventajas que nos ofrece cada uno a la hora de programar sus periféricos.

La Delfino utilizada en este proyecto ha sido la F28379D. La tarjeta de desarrollo, F28379D, la cual podemos ver en la Figura 3.8 cuenta con 2 CPUs TMS320C28x de 32 bits fabricadas por Texas Instruments. La frecuencia de las CPUs es de 200MHz y además cuenta con 2 CLAs (*Control Law Accelerators*) de también 200Mhz. Incluye una gran variedad de periféricos, entre ellos los siguientes:

- 169 GPIOs de propósito general.
- 2 módulos CAN.
- 3 módulos SPI de hasta 50MHz.
- 4 interfaces SCI/UART.
- 2 interfaces SPI.
- 4 ADCs de 16 bits
- 24 canales PWM
- 3 Decoders de cuadratura

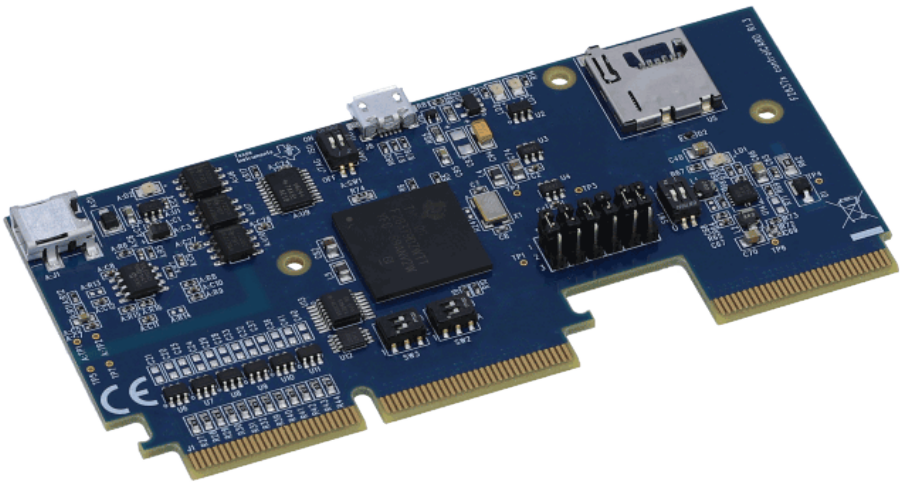

Figura 3.8: Tarjeta de desarrollo Delfino

Se ha utilizado la placa experimental, la cual viene acompañada con una base de 180 pines como la que se puede observar en la Figura 3.9, la cual permite acceder a todos sus periféricos de manera muy sencilla. Con ella podemos acceder a todos los periféricos de la tarjeta y también utilizarla como *breadboard* si fuese necesario, dado que junto a los pines de la tarjeta tenemos alimentación, tierra y una serie de pines hembra no conectados que se pueden utilizar para generar los circuitos de prueba necesarios.

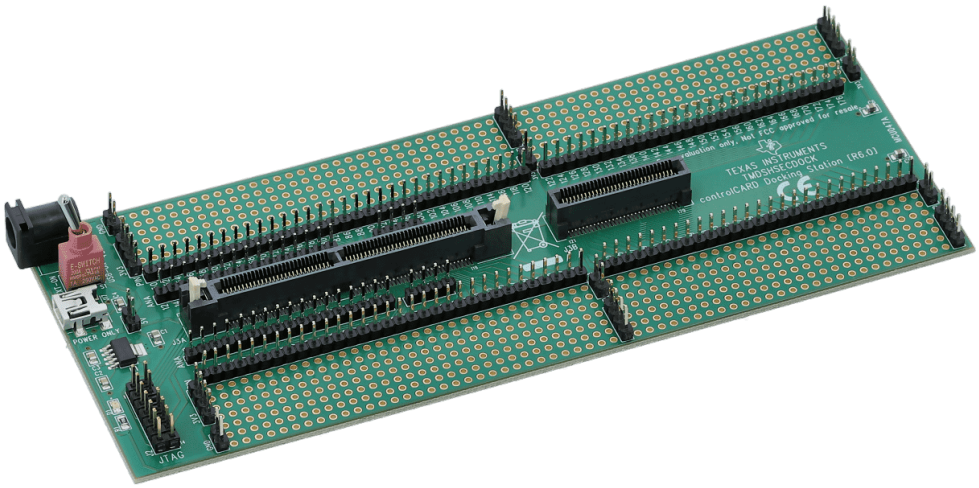

Figura 3.9: Dock de 180 pines de la Delfino

3.5.1 | Conexiones Delfino

Para la conexión de la Delfino y las diferentes interfaces y dispositivos, se han utilizado cables dupont utilizando los pines del dock. En la Tabla 3.4 se pueden observar las conexiones entre la Delfino y los diferentes módulos instalados en la cDAQ. En la Tabla 3.5 se pueden observar las conexiones entre la Delfino y los GPIO de la Diolan, que se han utilizado como salidas y entradas digitales. En la Tabla 3.6 se pueden observar las conexiones entre la Delfino y las interfaces de comunicación SPI e I2C de el adaptador Diolan Dln-2.

Módulo 1 - 9401(Digital Out)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
DIO6	23	68 (GPIO-20) QEP1A
DIO7	25	70 (GPIO-21) QEP1B
COM	13	GND
Módulo 2 - 9263(Analog In)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
AO0	0	9 (ADC-0)
COM	9	GND
Módulo 4 - 9401(Digital In)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
DIO4	20	49 (GPIO0-PWM1A)
COM	13	GND

Tabla 3.4: Conexiones de la Delfino y los módulos de la cDAQ

DIOLAN GPIO (In/Out)		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
C0 (Out)	13 (GPIO 16)	86 (GPIO-34)
C1 (Out)	14 (GPIO 17)	88 (GPIO-39)
C2 (Out)	15 (GPIO 18)	90 (GPIO-44)
C3 (Out)	16 (GPIO19)	92 (GPIO-45)
B0 (In)	25 (GPIO 8)	122 (GPIO-36)
B1 (In)	26 (GPIO 9)	124 (GPIO-38)
B2 (In)	27 (GPIO 10)	126 (GPIO-61)
B3 (In)	28 (GPIO 11)	128 (GPIO-63)
GND	11	GND

Tabla 3.5: Conexiones de la Delfino y los GPIO del adaptador Diolan

DIOLAN I2C		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
I2C_SDA	I2C Pin 1	85 (GPIO-32 I2CSDAA)
I2C_SCL	I2C Pin 4	87 (GPIO-33 I2CSCLA)
GND	I2C Pin 2	GND
DIOLAN SPI		
Canal del Módulo	Pin del Módulo	Pin de la tarjeta
SCK	SPI Pin 7	71 (GPIO-18 SPICLKA)
MISO	SPI Pin 3	69 (GPIO-17 SPISOMIA)
MOSI	SPI Pin 1	67 (GPIO-16 SPISIMOA)
NSS0	SPI Pin 5	73 (GPIO-19 SPISTEA)
GND	SPI Pin 11	GND

Tabla 3.6: Conexiones de la Delfino y las interfaces SPI/I2C del adaptador Diolan

4 | Protocolo de Comunicación

Para que las pruebas puedan ser ejecutadas correctamente es necesario un protocolo de comunicación entre el PC y el DUT. Este protocolo permite que ambos dispositivos se puedan comunicar, intercambiando información necesaria para la ejecución de las pruebas. Una de las grandes ventajas de esta comunicación es que el PC le puede enviar un identificador de la prueba a realizar a la DUT, en el mensaje que se envía al inicio de cada test. De este modo la DUT se puede programar con un gran número de pruebas diferentes, y que elija la ejecución de la prueba necesaria dependiendo de dicho identificador.

Esto hace que solo haya que programar 1 vez la DUT para el número total de las pruebas, agilizando mucho el proceso de verificación.

En el trabajo utilizado como punto de partida el protocolo de comunicación tenía un tamaño fijo de 8 Bytes, lo cual para las pruebas realizadas hasta el momento era suficiente, pero puede resultar muy escaso para otras pruebas. Por ejemplo, como se ha visto antes, las pruebas de escritura requieren que el PC envíe los datos del test a la DUT en el mensaje inicial. Con el protocolo anterior tan solo 4 de los 8 Bytes estaban reservados para los datos del test, por lo que en las pruebas de escritura y lectura I2C/SPI solo se podrían transmitir 4 Bytes.

Para solucionar este problema, se ha implementado un nuevo protocolo. Además, se ha decidido optar por un protocolo de tamaño variable porque aunque la programación del protocolo sea más compleja tiene grandes beneficios. Uno de los mayores beneficios es la mejora de la velocidad de las comunicaciones. Como la comunicación se realiza por un puerto serie, los datos se van mandando uno tras otro, por lo que el tiempo de transmisión es proporcional a la cantidad de Bytes. Por lo tanto, un protocolo de tamaño variable hace posible que el mensaje pueda portar muchos Bytes de datos si la aplicación así lo necesita, como en el caso de los comunicaciones SPI e I2C. Ocurre lo mismo si se quiere transferir texto a través del protocolo. Como el entorno de verificación tiene como objetivo poder ser fácilmente ampliable para futuras verificaciones, es posible que alguna aplicación requiera transmisión de texto, para verificar la escritura en una tarjeta SD por ejemplo.

Por el contrario, en los casos en los que el mensaje enviado no tiene muchos datos, como pasa en el caso de los mensajes de confirmación que no necesitan de ninguno, se envía un mensaje con el tamaño mínimo posible y así se gana mucha velocidad.

4.1 Diseño del protocolo

El protocolo de comunicación establece la estructura que deben seguir los mensajes. Cada uno de los mensajes está compuesto por 6 campos. Cada campo cumple con una función específica, para hacer que el tratamiento de los mensajes sea más sencillo.

Los campos *Interfaz, Comando* y *Datos* forman la Unidad de Datos. Esta Unidad de Datos es la que porta la información de la prueba a realizar, y es única para cada prueba. El campo de *Interfaz* y *Comando* siempre tienen 1 y 2 Bytes respectivamente. El tamaño variable se consigue gracias al campo de *Datos*, el cual puede tener desde 0 hasta 250 Bytes, de modo que el mensaje tenga siempre un máximo de 256 Bytes.

El resto de campos, *Cabecera, Tamaño* y *CRC* guardan información sobre el propio mensaje, independientemente del test a realizar.

En la Figura 4.1 se pueden observar los campos que forman el mensaje, los cuales se detallan más en profundidad a continuación.

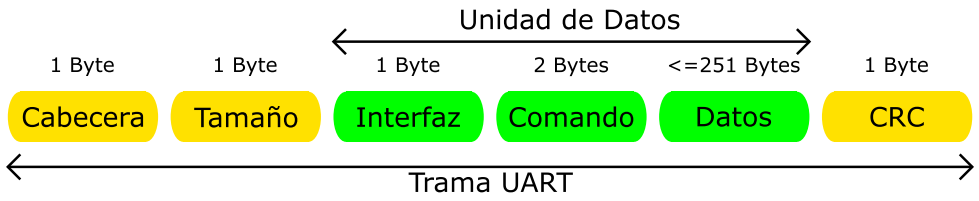

Figura 4.1: Esquema de la trama de datos

- **Cabecera:** La cabecera se utiliza para determinar la dirección del mensaje. El bit de menor valor de la cabecera debe tener el valor 1 si el mensaje se envía desde el PC al DUT y 0 si se envía desde el DUT al PC. Los demás bits de la cabecera quedan en desuso por el momento, pudiendo implementar funcionalidades extra en el futuro.

- **Tamaño:** El byte de tamaño indica el número de bytes restantes que tiene el mensaje, es decir, el tamaño a partir del campo *Interfaz*. Dado que los campos *Cabecera* y *Tamaño* tienen 1 Byte cada uno, el mensaje completo tendrá *Tamaño* + 2 Bytes, con un máximo de 256 Bytes.

- **Interfaz:** El Byte de interfaz se utiliza para seleccionar la interfaz de comunicación con la que se debe realizar el test. Se reserva para las interfaces de comunicación dado que las placas de desarrollo suelen incorporar varias. En caso de que la prueba no utilice ninguna interfaz de comunicación se seleccionará 0. Por ejemplo, tanto la interfaz SPI como la I2C realizan pruebas de escritura, por lo que el byte de interfaz se utilizaría para seleccionar la deseada.

De este modo ambas pruebas de escritura contarán con el mismo comando, pero con diferente interfaz.

- **Comando:** Para determinar el comando se utilizan dos Bytes, el primero de ellos siendo el LSB y el segundo el MSB. El comando indica al DUT el test a realizar. Cada uno de los tests tiene un comando reservado, y por norma general los mensajes de respuesta enviados por el DUT utilizan el mismo comando que el recibido, siempre que no se detecte un error, lo cual tiene un comando propio.

  Hay un comando reservado para la identificación de errores, 0x00FF. Si el DUT identifica algún error, colocará el comando de error en el mensaje de respuesta, haciendo saber al PC que ha ocurrido un error durante la recepción del mensaje.
  Dado que el PC es el que controla el flujo de la comunicación, el PC no envía comandos de error al DUT.

- **Datos:** El campo de datos se utiliza para mandar los diferentes datos o parámetros necesarios para el test, como se indica en cada uno de los tests [11].
  En el caso de la DUT, los datos enviados son generalmente resultados del test realizado.

  En caso de error, el campo de *Datos* se utiliza para transferir varios datos que ayudan a identificar el error y su causa. El primer dato siempre será el tipo de error, los cuales podemos observar en [12]. A continuación se envían diferentes datos dependiendo del tipo de error. Por ejemplo, si en una prueba con cantidad de datos fija como la lectura de posición de encoder se detecta que no se han recibido la cantidad de datos adecuados, se enviará la cantidad de datos esperada y la cantidad recibida.

- **CRC:** El último Byte de la trama es siempre un byte de verificación de redundancia cíclica (CRC). El CRC es un código de detección de errores. En nuestro caso, el CRC se calcula computando el XOR de los bytes de los campos *Tamaño, Interfaz, Comando* y *Datos*.

  De este modo, el receptor del mensaje puede recalcular el CRC realizando el XOR de los campos mencionados, y compararlo con el campo *CRC* del mensaje recibido. Si ambos valores no son idénticos significa que ha ocurrido un error en la transmisión en algunos de los Bytes del mensaje.

En la Tabla 4.1 se puede observar un resumen de los campos que forman el protocolo.

Protocolo comunicación serial PC - DUT			
Campo	Posición	Bytes	Detalles
Cabecera	0	1	PC ->DUT 0x00 DUT ->PC 0x01
Tamaño	1	1	Bytes restantes (Interfaz, comando, datos y CRC)
Interfaz	2	1	Utilizado para identificar la interfaz utilizada. 0x00 para tests sin interfaz.
Comando	3	2	Byte 1 LSB Byte 2 MSB
Datos	5	0-250	Desde 0 hasta 250 Bytes, dependiendo del comando
CRC	-1	1	XOR de todos los Bytes menos la Cabecera

Tabla 4.1: Protocolo PC-DUT serial

4.1.1 | Comandos

Los códigos de todos los comandos vienen definidos en la documentación del protocolo [13].

Además de para informar al micro sobre la prueba que se va a realizar, también se definen una serie de comandos genéricos:

- Test de conexión (0x00FD):
  Para iniciar la comunicación entre el PC y el DUT, el primer mensaje siempre debe ejecutar un test de conexión. El PC enviará un mensaje con el comando de test de conexión, y el DUT deberá responder con el mismo comando, verificando así que la comunicación UART funciona correctamente y se pueden realizar el resto de los tests.

- Fin de test (0x00FE):
  Tras realizar todos los tests, el PC enviará un mensaje de Fin de Tests, indicando así al DUT que los tests han finalizado.
  Por su parte, el DUT responderá con el mismo comando, indicando en un Byte de datos si ha habido errores en la ejecución de los Tests (!0) o se han realizado correctamente (0).

- Comando de error (0x00FF):
  Cuando el DUT encuentra un error en el mensaje, responde al PC colocando el comando de error (0x00FF) en el campo *Comando*. El error puede darse por los siguientes motivos:

  - Error de Cabecera: Cabecera diferente a 0.
    1 Byte -> Código de error (0).
    1 Byte -> Cabecera recibida.

- Error de Tamaño: Tamaño no adecuado para el comando recibido.
  1 Byte -> Código de error (1).
  1 Byte -> Tamaño esperado.
  1 Byte -> Tamaño recibido.
- Error de Interfaz: Interfaz no adecuada para el comando recibido.
  1 Byte -> Código de error (2).
  1 Byte -> Interfaz recibida.
- Error de Comando: EL comando del mensaje no esta programado en la DUT.
  1 Byte -> Código de error (3).
  1 Byte -> Comando recibido.
- Error de Datos. Los datos recibidos son incorrectos.
  1 Byte -> Código de error (4).
- Error de CRC: CRC recibido y calculado no son iguales.
  1 Byte -> Código de error (5).
  1 Byte -> CRC esperado.
  1 Byte -> CRC recibido.
- Timeout: Ha ocurrido un Timeout en el PC.
  1 Byte -> Código de error (6).

4.2 | Implementaciones del protocolo en LabVIEW

Dado que el PC es el que controla el flujo de ejecución del entorno de verificación en todo momento, va a ser LabVIEW el que genere los mensajes y verifique que la respuesta obtenida es correcta. Para ello, se van a implementar sub-VIs que ejecuten el envío y la recepción de los mensajes. Posteriormente, esos sub-VIs se utilizarán en los VIs que ejecutan las pruebas y forman el entorno de TestStand, para realizar las comunicaciones necesarias dentro de cada prueba.

Como se analizará en la sección de implementación del entorno en TestStand y se ha mencionado con anterioridad, TestStand puede manipular los parámetros de entrada con los cuales ejecuta los VIs, por lo que se pueden utilizar VIs con señales de entrada, y manipularlas dependiendo de la acción a realizar.

Con el protocolo ocurre lo mismo, se generan VIs generales para el envío y la recepción de datos, y después TestStand manipula las entradas de los VIs para cambiar el mensaje a enviar. De tal modo, cambiando el comando por ejemplo, se consiguen ejecutar las diferentes pruebas del entorno, utilizando el mismo VI de envío.

4.2.1 | Envío

Un VI de LabVIEW se divide en dos ventanas. Por un lado tenemos el panel frontal, el cual utiliza el operario. El panel frontal se compone principalmente de controladores e indicadores. Con ellos el operario puede modificar los valores de diferentes variables y ver como se modifican las salidas en los indicadores, ya sea de manera numérica o gráfica.

Por otro lado, se encuentra el diagrama de bloques. Este diagrama contiene la programación de la aplicación, utilizando los bloques y subVIs necesarios. En un caso de uso típico el desarrollador es quien programa el diagrama de bloques, y el operario final solamente utiliza el panel frontal.

Dado que TestStand llama externamente a los VIs, el panel frontal queda inutilizado, dado que TestStand accede al VI conectándose a las entradas y a las salidas del mismo.

Por tanto, para que el entorno final sea cómodo de utilizar, es importante conseguir que los datos estén bien organizados, para que después sean fácilmente modificables desde TestStand.

Para ello, se ha creado un cluster que contiene los datos de los mensajes, de esta manera el mensaje completo se agrupa en un parámetro. Dentro del cluster se utiliza un control para cada uno de los campos de la trama, utilizando un array de 2 Bytes para el campo *Comando* y uno de tamaño 250 para el campo de *Datos*. Cada uno de los datos consiste en un número entero de 8 Bits, dado que la comunicación UART está así configurada.

En el caso del array de datos, se le ha asignado el tamaño máximo posible, y posteriormente se seleccionan la cantidad necesaria dependiendo del valor del campo *Tamaño*.

Figura 4.2: Control personalizado del mensaje a enviar

En la Figura 4.2 se puede observar el cluster de control utilizado. Los campos *Cabecera* y *CRC* están ocultos, dado que no se pueden modificar por el usuario. La cabecera se genera

automáticamente, dado que solo depende de la dirección del mensaje, y el CRC se calcula automáticamente como se verá más adelante.

Para enviar los datos por UART, se han utilizado los bloques de la librería *Serial* de *Instrument I/O* de LabVIEW. Los bloques de dicha librería realizan el envío de datos a partir de un string, por lo que para poder enviar el mensaje se debe convertir el cluster de entrada a una cadena de texto.

En la Figura 4.3 se puede observar el proceso de envío, que consta de 3 partes. El primer subVI, *createMessage*, se encarga de generar la cabecera del mensaje, calcular el CRC y delimitar el array de datos. El segundo se encarga de convertir el mensaje a una cadena de texto. El tercero, *serialWrite*, se encarga de enviar los datos a la DUT a través de UART.

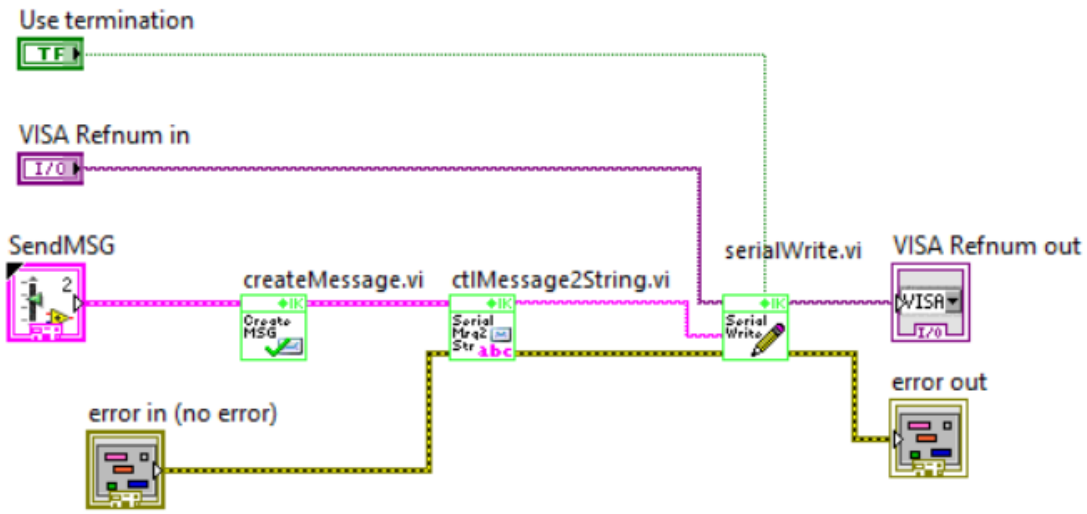

Figura 4.3: Proceso de envío de mensaje

4.2.2 | Recepción

El proceso de recepción es más complejo, dado que hay que realizar una serie de comprobaciones para verificar que la comunicación se está realizando correctamente. Por un lado, hay que implementar un timeout para asegurar que TestStand no se quede esperando a una respuesta indefinidamente si el micro no responde. Por otro lado, si la DUT responde con un mensaje de error, el programa detecta que se ha recibido el comando de error y genera un error para que TestStand lo guarde en el reporte.

Además, dado que el DUT siempre responde con el mismo comando que se le ha enviado y algunos comandos responden con mensajes de tamaño concreto, se puede comprobar que el mensaje recibido tiene el comando y el tamaño correcto.

Finalmente, se computa el CRC del mensaje recibido y se comprueba que el mensaje es correcto comparándolo con el campo *CRC*.

Para ello, se ha separado el proceso en dos partes, primero, se ejecuta la recepción del mensaje como tal, implementando el timeout, la cual se observa en la Figura 4.4 y verificando la integridad del mensaje.

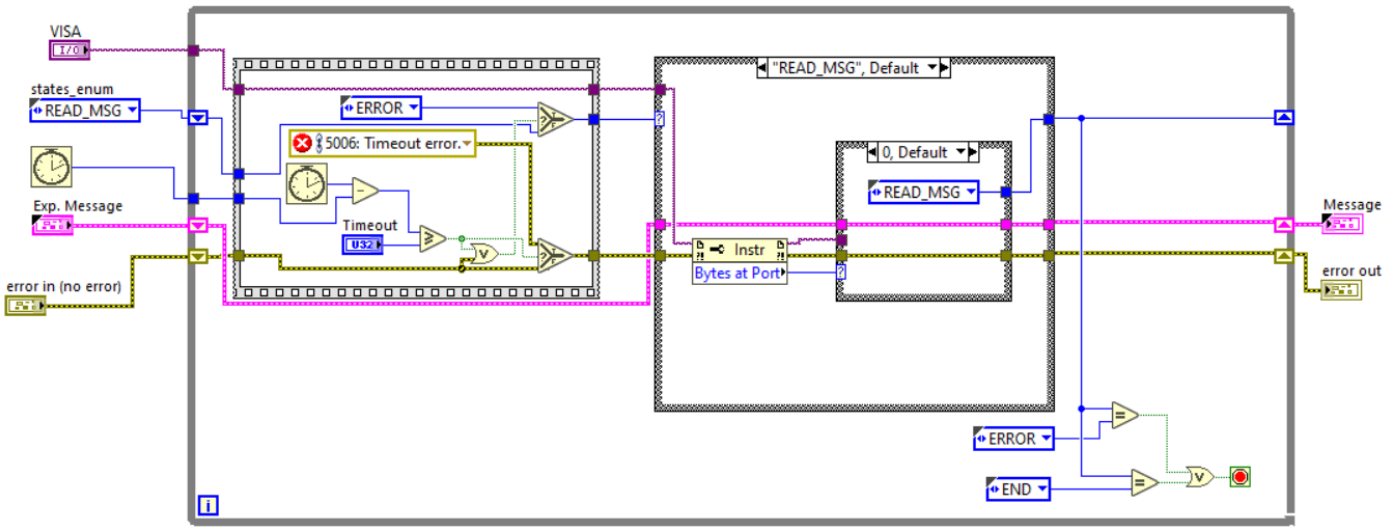

Figura 4.4: Proceso de recepción de mensaje

En esta figura se ejecuta un bucle que comprueba si hay datos en el puerto. Si no los hay, sigue en el mismo estado, siempre que no ocurra un timeout. Si ha ocurrido un timeout, se genera un error y se finaliza el programa.

Si por el contrario el puerto tiene datos disponibles, se ejecuta un subVI encargado de leer el mensaje y organizarlo en el cluster. Si no se detecta ningún error, se pasa a la comprobación

del mensaje. En el se comprueba que el mensaje recibido no contenga el comando error, lo que significaría que el micro ha detectado un error, y que el CRC es correcto.

En la Figura 4.5 se puede observar como se realiza el proceso. En la parte superior se selecciona el próximo estado, error o fin, dependiendo de si ha sucedido un error o no.

En el centro, si se ha obtenido el comando de error, se generan el error correspondiente, utilizando los datos recibidos por la DUT, de acuerdo a los códigos de error mencionados en el protocolo, en la sección 4.1.1. Este error incluye un breve mensaje informativo del motivo del error y datos adicionales, por ejemplo en el caso de un error de tamaño el mensaje indica el tamaño esperado para el mensaje y el tamaño obtenido.

En la parte de abajo, se comprueba que el CRC es correcto, y si no lo fuese se genera el error correspondiente. En este caso, también se incluye información sobre el error, y se indica el CRC del mensaje recibido (valor del campo *CRC*) y el calculado.

Todos estos errores se conectan a una salida del VI que ejecuta TestStand, para que así TestStand lo incluya en el reporte, guardando así los mensajes de error en los resultados de la prueba.

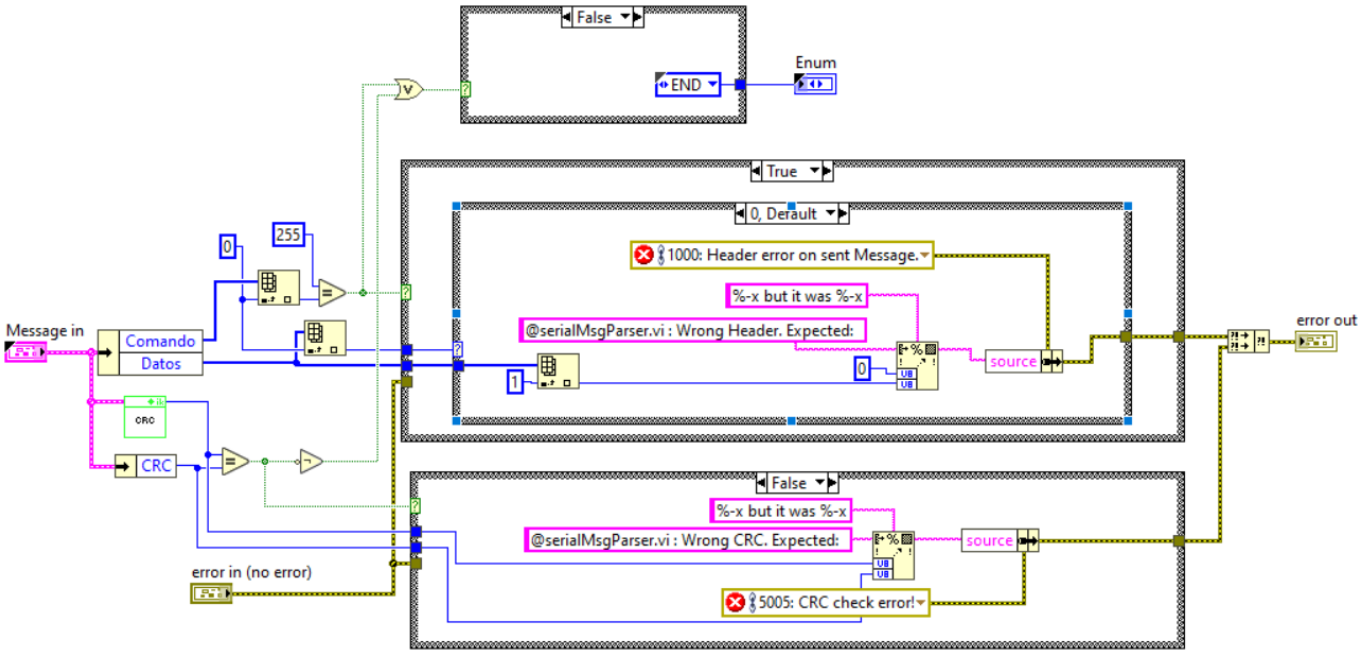

Figura 4.5: Comprobación de errores

Finalmente, se ejecuta la otra parte, la cual verifica que el mensaje obtenido es válido, de acuerdo a la prueba realizada, es decir, verifica la cabecera del mensaje, que el comando obtenido sea el mismo que el comando de la prueba, que el tamaño del mensaje sea correcto (en las pruebas con tamaño fijo) y que el campo de interfaz sea el mismo que en el mensaje enviado.

Estos errores también generan mensajes que el operador que esté utilizando TestStand puede leer para saber que ha ido mal en la prueba. Además, para ser identificados más fácilmente, se les ha dado también un código de error, el cual va a ser 5000 + su código de la sección 4.1.1. Por ejemplo, un error de tamaño encontrado en la DUT tiene código 1, por lo que un error del tamaño recibido en LabVIEW tendrá 5001.

Estos códigos simplemente se generan para dar información al operario de TestStand. Dado que el PC está realizando la verificación del micro, no tiene sentido enviar un mensaje al micro informándole de un error. El error se saca como una salida de LabVIEW y TestStand lo reporta.

4.3 Implementacion del protocolo en los micros

Del mismo modo que en el PC, también hay que implementar el protocolo en los dispositivos que se van a testear. Se ha realizado la misma separación que en LabVIEW respecto a la transmisión del mensaje y la gestión de los errores.

Por una parte se realiza la transmisión de los mensajes. Esto consiste en la recepción del mensaje, y el envío de la respuesta correspondiente. Además, junto con la recepción se realiza la verificación de la integridad del mensaje, es decir, la validación de la cabecera y del CRC. Como ocurría en LabVIEW, esto se realiza separado del resto de verificaciones porque son verificaciones puramente independientes del test a realizar, y sirven para verificar que no ha ocurrido ningún error de comunicación.

Por otra parte, se realiza la verificación de los datos necesarios para la prueba, es decir, que el tamaño sea correcto, la interfaz sea válida, los datos sean válidos, etc. Como estas verificaciones dependen de la prueba a realizar, se realizan dentro de cada prueba. Por tanto, cada vez que se quiera implementar una nueva prueba para el entorno de test, ha de programarse también la gestión del mensaje recibido, dependiendo de los parámetros necesarios para la prueba. Estas comprobaciones incluyen la interfaz y el tamaño de los datos y su contenido. Dado que el tamaño es variable, hay pruebas que no podrán comprobar que el tamaño de los datos es válido.

Si el entorno de verificación se expande mucho, puede ocurrir que se intente realizar a un dispositivo una prueba que no tiene programada. Aunque esto realmente es un error del operador y no debería ocurrir si la ejecución del entorno se ha configurado bien, se ha decidido implementar un comando específico para dicho error, el error de comando de la sección 4.1.1.

En el caso del envío, dado que el tamaño del mensaje lo estable la prueba realizada, se envían ese número de Bytes.

La recepción se realiza de manera similar a como se hace en LabVIEW, primero se reciben 2 Bytes. El segundo contiene el tamaño del mensaje restante, por lo que se lee su valor y después se reciben ese número de Bytes.

5 | Implementación del entorno de verificación

La implementación del entorno de verificación se puede separar en dos grandes grupos, la implementación del propio entorno en el PC, y la implementación de la ejecución de las pruebas en los micros.

En cuanto al PC, se ha utilizado LabVIEW para generar VIs que ejecutan las pruebas individuales, es decir, un VI para cada prueba. Después, se utiliza TestStand para organizar las pruebas en secciones y gestionar la ejecución de las pruebas con diferentes parámetros. Todo esto, una vez programado, se puede iniciar con un click, lo que ejecutará el entorno entero automáticamente, y ofrece la opción de hacerlo repetidamente para más dispositivos, como se verá en la implementación de TestStand, lo cual es imprescindible para la verificación de fabricaciones en masa de placas [14].

En cuanto a los micros, se utilizan los drivers proporcionados por los fabricantes para programar los periféricos de la tarjeta para realizar los tests del entorno. Se programa el micro para que ejecute la prueba especificada por el comando del mensaje recibido, y así conseguir tener que cargar una sola vez el programa en el micro para la ejecución del entorno completo.

5.1 | Diseño de los tests en LabVIEW

Como ya se ha comentado, cada test de TestStand va a constar de un VI de LabVIEW. Por tanto, las pruebas tienen que ejecutarse completas en un solo VI. Además, todas las pruebas deben empezar con el envío de un mensaje y terminar con la recepción de otro. Estos dos mensajes se colocarán como entrada y salida del VI respectivamente, para poder controlar los datos que se le envían al micro, y los datos que devuelve al final de la prueba.

Las pruebas que necesiten de parámetros extra, como la frecuencia del encoder, también se colocan como entradas del VI, para así poder modificarlas posteriormente en TestStand.

5.1.1 | Prueba de posición de Encoder

La prueba consiste en generar una señal simulada de encoder durante un tiempo determinado. La frecuencia de la simulación será fija durante la prueba, y la dirección también. Se ha decidido así para asegurar que el sistema funciona en tiempo real. Esto se consigue generando todas las señales con los timers internos de la cDAQ, sin modificación del PC mientras dure la prueba.

Un encoder de cuadratura lo componen dos señales (A y B) desfasadas ± 90 eléctricos, dependiendo de la dirección de giro. A velocidad de giro constante, esto significa dos señales cuadradas con la misma frecuencia, desfasadas por 1/4 de pulso. Para simular estas dos señales con el desfase correcto, se ha optado por generar una tercera señal, a doble de frecuencia. Esta señal va a ser interna, no se va a conectar a ningún puerto de la cDAQ, y se va a utilizar solo como referencia de las otras dos señales.

De tal manera, se inicia una de las señales con el flanco ascendente de la señal interna, y la otra con el flanco descendente, consiguiendo así el desfase de 90º eléctricos del encoder. Para realizar la simulación en el otro sentido, se invierten los flancos de inicialización. En la Figura 5.1 se puede ver un esquema de este método.

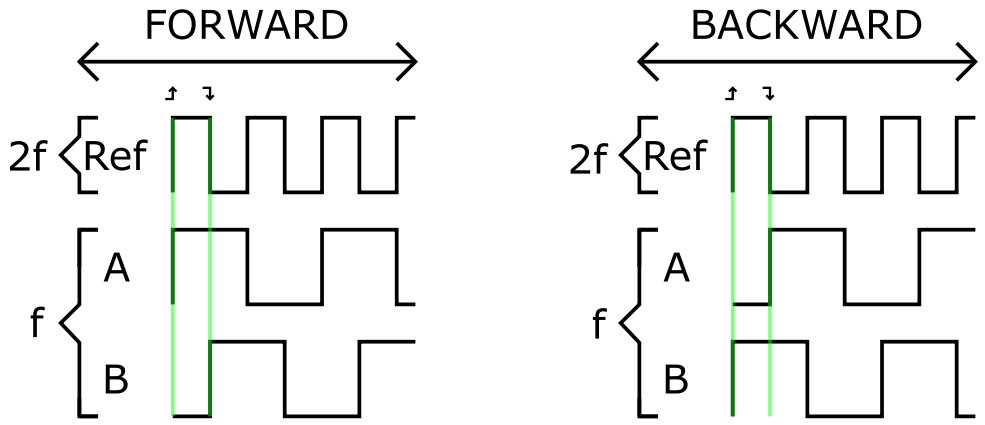

Figura 5.1: Simulación del Encoder

Como la señal interna solo se utiliza como referencia, basta con generar un único pulso. Para las señales A y B del encoder, se generan $T * f$ pulsos, siendo T el tiempo de la simulación en segundos y f la frecuencia de las señales.

En la Figura 5.2 se puede observar como se generan las señales. Se configuran los canales A y B, con Duty Cycle de 0,5 y la frecuencia deseada en la parte superior del VI.

El primer bloque configura la frecuencia y el Duty Cycle. Después el canal de salida. El bloque de reloj configura el número de muestras, y el siguiente el flanco de inicio, el cual utiliza como referencia la salida interna del contador 0. En el bloque central del VI se calcula el número de muestras, $T * f$, y se selecciona el flanco de subida o bajada dependiendo de la dirección.

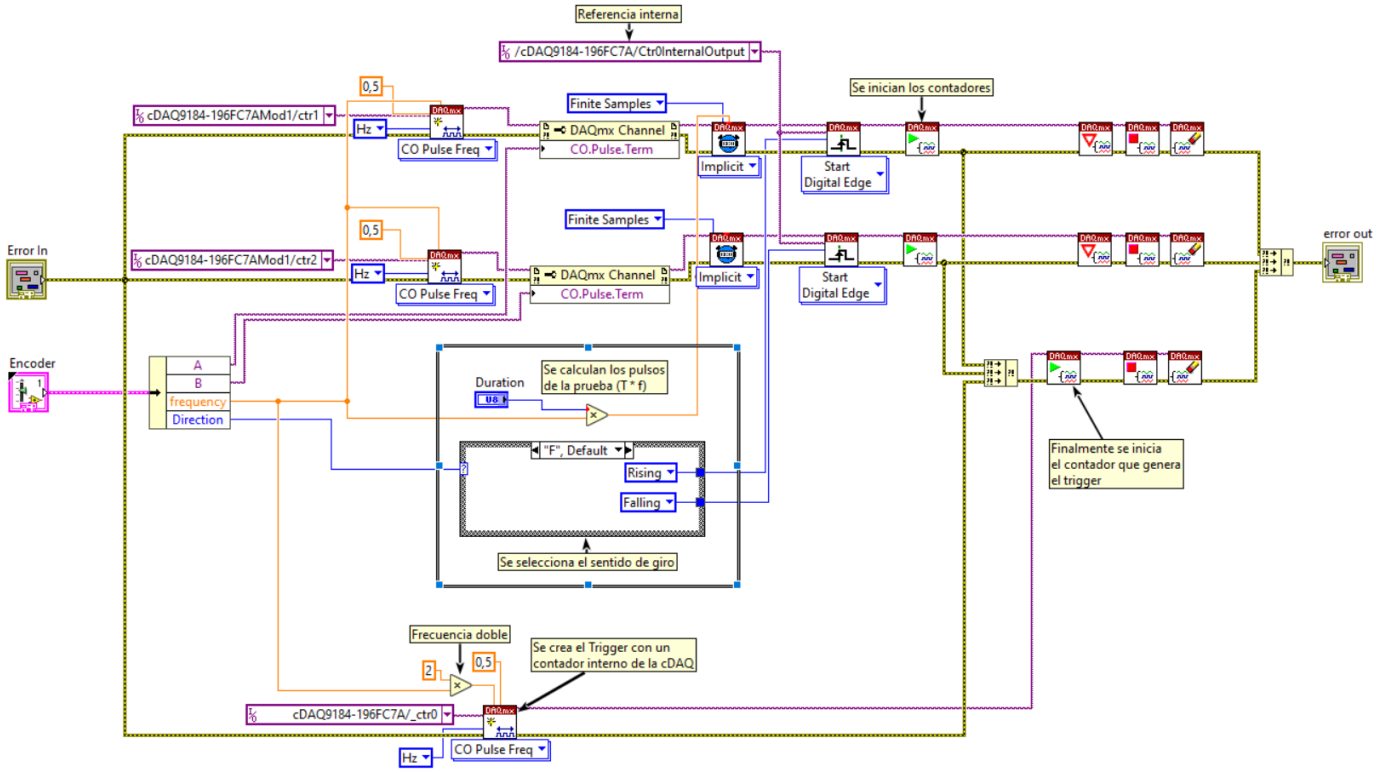

Figura 5.2: Generación de la señal simulada de encoder en LabVIEW

En la parte inferior se genera la salida interna, configurando el contador 0 al doble de frecuencia.

Primero se inician los dos contadores que simulan cada uno de los canales del encoder. Es importante hacer esto antes de iniciar el canal interno de referencia, para asegurar que los triggers utilizados sean dos flancos consecutivos.

Finalmente, se inicia el contador que genera el trigger.

Como se ha seleccionado clock implícito para los dos canales, generarán la simulación con el timer de la cDAQ, hasta generar los pulsos configurados. Mientras tanto, el PC se quedará a la espera de que la cDAQ le informe de que la simulación ha terminado.

La simulación del encoder forma el subVI *SimulateEncoderPosition* que después se utiliza dentro del VI de la prueba, el cual se puede observar en la Figura 5.3.

Este VI empieza ejecutando el subVI de comunicación *sendMessageAndParseAnswer*, enviando el mensaje de la prueba al micro, y comprobando su respuesta. Como se explicaba en el diseño de la prueba, en la sección 2.2.1, el DUT responde con un mensaje vacío, cuando esté listo para adquirir los datos. Por tanto, la comprobación del mensaje se hace con el cluster constante ExpectedANS-ACK, dado que el mensaje de confirmación siempre será constante: Sin datos, con el mismo comando que la prueba, e interfaz 0.

Una vez que el micro confirma que está listo para la lectura, se ejecuta la simulación del encoder.

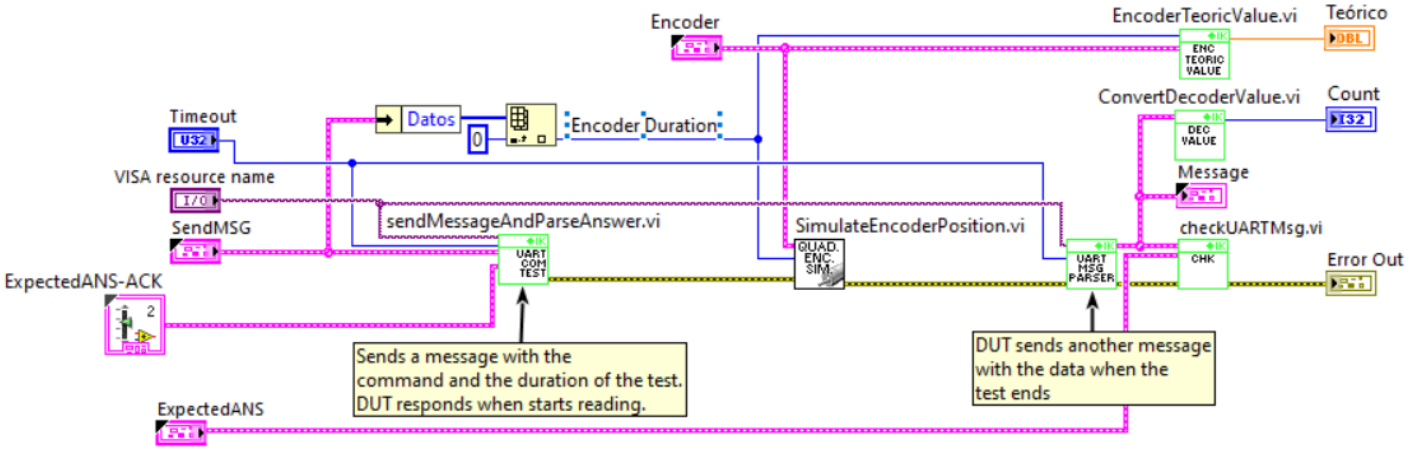

Figura 5.3: Prueba de posición de encoder de LabVIEW

Para ello se utiliza el subVI de la Figura 5.2 que se acaba de analizar.

Finalmente, se espera hasta recibir un último mensaje de la DUT. Este mensaje contiene los resultados de la prueba, es decir, la cantidad de pulsos que ha leído la DUT. Dicho valor se saca como salida, *Count*. Además, se calcula el valor real o teórico de pulsos generados, al que se le ha llamado *Teórico*. Este valor será $4 * T * f$, dado que el decoder se utiliza en modo cuadratura x4.

El VI saca ambos valores como salida y es TestStand el encargado de verificar si la prueba ha sido exitosa.

5.1.2 | Prueba de velocidad de Encoder

La programación de la prueba de velocidad ha sido algo diferente, ya que en este caso el ritmo de la prueba lo marcará el DUT, dado que para el cómputo de la velocidad se calcula la cantidad de pulsos en un periodo de muestreo, es decir, la diferencia de pulsos entre dos muestreos sucesivos. Por tanto, la prueba finaliza cuando el DUT envíe un mensaje con los resultados al PC.

Se podría haber utilizado el mismo procedimiento que en la prueba de posición, pero eso conllevaría que cada prueba duraría un mínimo de 1 segundo. De este modo la duración de la prueba se reduce a tan solo 2 periodos de muestreo, y no es necesario esperar a que acabe la simulación, lo que si se realizan muchas pruebas puede superar una gran cantidad de tiempo.

En este caso por tanto, no es necesario decirle a la cDAQ cuantas muestras generar, se ejecutará la simulación hasta que la prueba haya terminado y el PC le ordene detener.

Por otra parte, dado que la cantidad total de pulsos no es importante, solo la diferencia entre dos periodos, la simulación puede empezar antes de que el DUT empiece la prueba, por lo que no es necesario un mensaje de confirmación.

En la Figura 5.4 se puede observar el VI de la prueba.

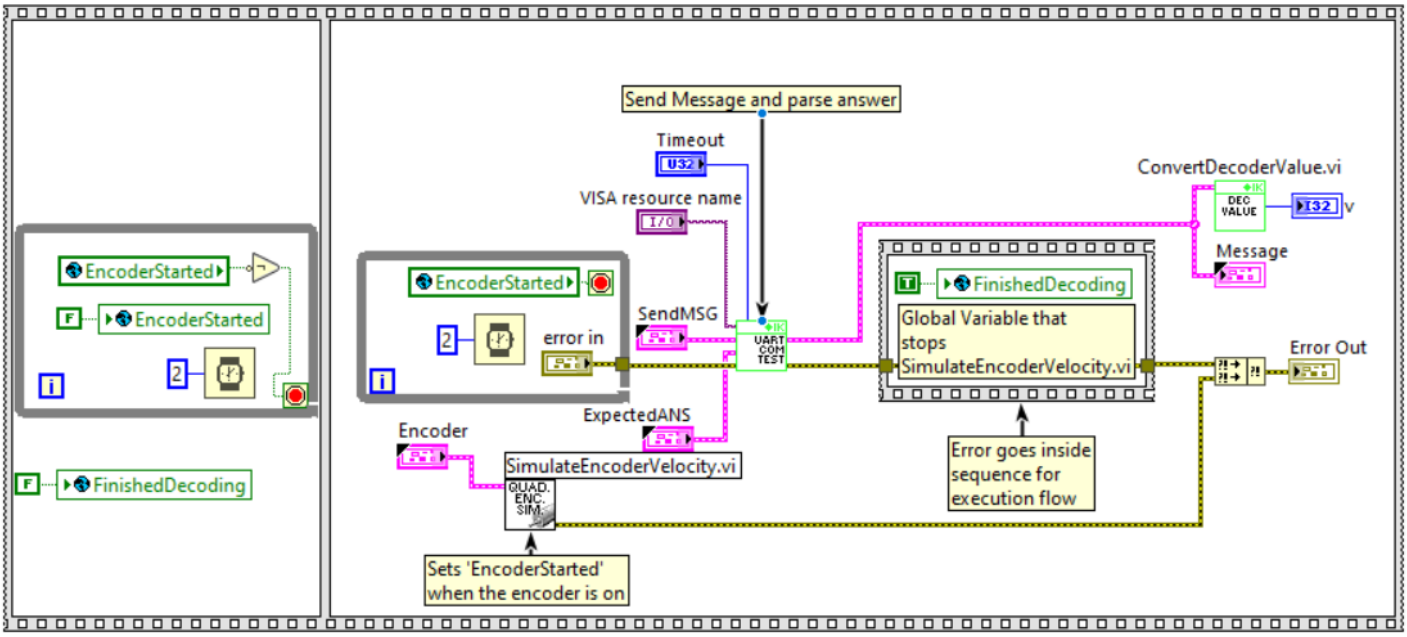

Figura 5.4: Prueba de velocidad de encoder de LabVIEW

El VI empieza restableciendo 2 variables globales. Estas variables se utilizan para controlar el flujo de las diferentes acciones que ocurren en paralelo.

Después, se ejecuta el subVI *SimulateEncoderVelocity*, este genera la señal simulada del encoder, de la misma forma que lo hacía la de la posición en la anterior prueba, pero se utiliza el método de muestras continuas para que genere señal hasta que se le ordene terminar. En paralelo, el bucle *while* se ejecuta, esperando hasta que se active la variable *EncoderStarted*.

Cuando la simulación haya comenzado, dicha variable se activará, lo que hará que el bucle finalice, y se envíe el mensaje de la prueba a la DUT. Mientras tanto, la simulación de encoder se ejecuta en paralelo hasta que la variable *FinishedDecoding* se active.

El subVI de comunicación se quedará a la espera de que el DUT responda. Cuando se obtenga la respuesta, se activara la variable global*FinishedDecoding* y el subVI *SimulateEncoderVelocity* detendrá la simulación y terminará.

Finalmente, se le envía el resultado obtenido *v* a TestStand como salida, para que el lo analice.

5.1.3 | Prueba de Lectura I2C y SPI

Las pruebas de lectura I2C y SPI se van a analizar juntas, dado que son muy similares de estructura, solo cambian los subVI de envío, los cuales se analizarán con más detalle para cada caso.

Como se comentaba en el diseño de las pruebas, la prueba de lectura I2C y SPI verifica la capacidad de leer de el DUT, por lo que es PC está realizando una escritura.

En la Figura 5.5 se puede observar la estructura general de estas pruebas. En este caso concreto, se trata de la prueba de I2C.

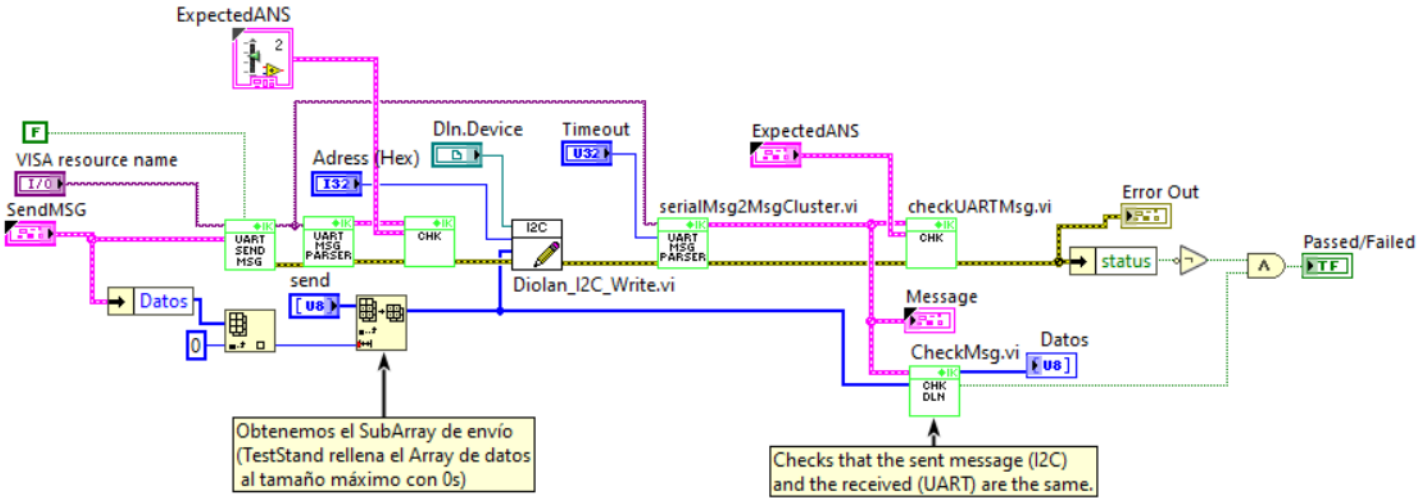

Figura 5.5: Prueba de Lectura I2C

Primero, se prepara el mensaje a enviar por I2C. Dicho mensaje se le da como entrada a TestStand. Para seleccionar la cantidad de mensajes a escribir, se utiliza el Byte nº0 del campo de Datos. Por tanto, para que no haya errores se recorta el array de entrada de TestStand al tamaño correcto.

En paralelo, se envía el primer mensaje al DUT para informarle de que se va a realizar la prueba de lectura SPI/I2C y se recibe un mensaje de confirmación cuando el DUT esta listo, como ocurría en la prueba de lectura de posición.

Después, se ejecuta el subVI que se encarga de realizar la escritura. Este subVI es diferente para cada interfaz y se han programado utilizando la librería que Diolan ofrece para su interfaz de comunicación Dln-2.

Tras completar la escritura, se vuelve a recibir otro mensaje por UART, el cual contiene los datos que el DUT ha leído durante la prueba. A diferencia de las pruebas de los encoders, esta vez el resultado de la prueba lo calcula LabVIEW. Para ello compara que los datos enviados y los recibidos coincidan en todos los Bytes. Después, envía el resultado a TestStand utilizando un booleano de salida, *Passed/Failed*.

Además, también se envían los valores escritos y los leídos a TestStand, para que se guarden en el reporte. Así, si la prueba fallase se podría ver que ha ocurrido en los datos. De tal forma es más sencillo encontrar el origen del error, dado que no es lo mismo fallar la prueba por no recibir ningún dato, o que algunos datos sean correctos y otros no.

5.1.3.1 Diferencias entre I2C y SPI en la lectura

La mayor diferencia entre el subVI de escritura I2C y el de SPI es la necesidad de dirección en I2C. El protocolo SPI funciona con una señal *ChipSelect* para la comunicación. Por tanto, en una red SPI con 5 dispositivos conectados, se requiere de 1 señal para cada uno de los dispositivos. En I2C, por el contrario, se utilizan direcciones. Todos los dispositivos de la red leen los mensajes, pero solo hacen caso a ellos si la dirección del mensaje es la suya.

Es por esto que la implementación es un poco diferente. En el caso del SPI, debemos conectar físicamente la señal de *ChipSelect*, y activarla cuando se realice la comunicación.

En el caso del I2C, el registro de dirección depende del dispositivo. Algunos dispositivos tienen dirección modificable, y otros lo tienen fijo. Por tanto, la dirección se ha implementado como una entrada de TestStand, para poder modificarla si el dispositivo no tiene dirección modificable. En este caso concreto, ambos DUTs ofrecen la posibilidad de modificar su dirección, por lo que se les ha configurado como 0x41.

5.1.4 | Prueba de Escritura I2C y SPI

Como en la anterior prueba, la escritura de ambas interfaces se va analiza conjuntanmente, dado que tienen la misma estructura.

Esta prueba verifica la capacidad de escribir por I2C/SPI del DUT, por lo que el PC en este caso recibe.

En la Figura 5.6 se puede observar el VI que implementa el test, en este caso concreto es el test de I2C.

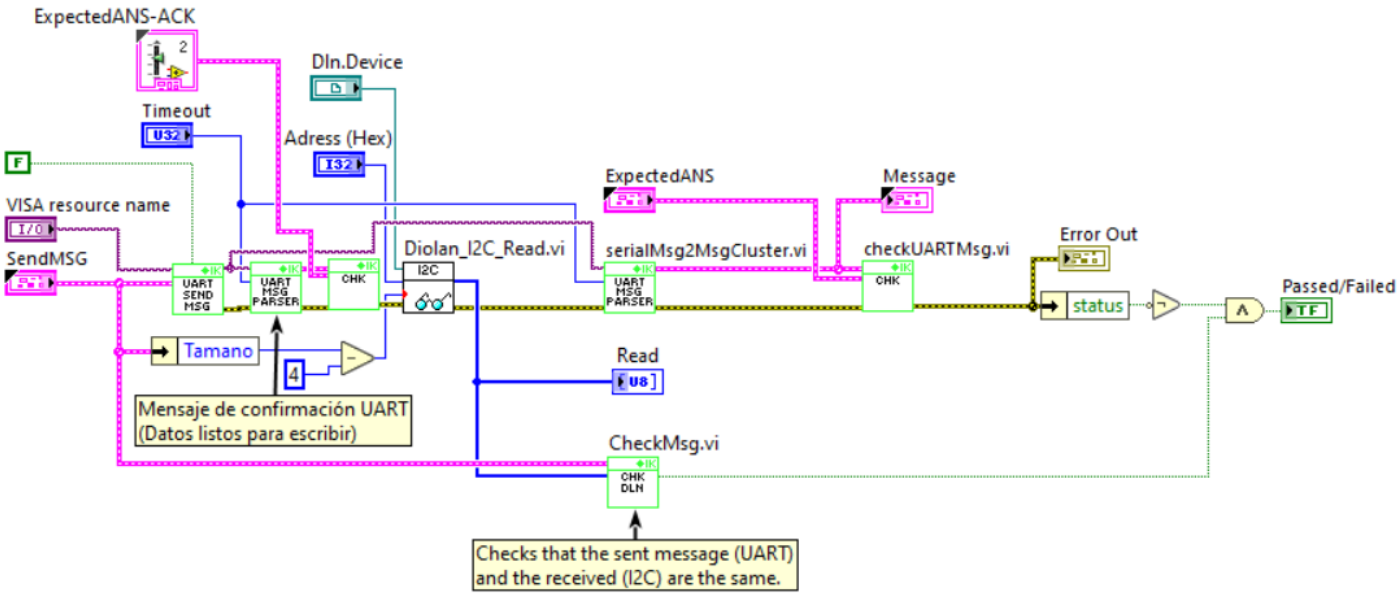

Figura 5.6: Prueba de Escritura I2C

Para que el DUT pueda escribir un mensaje conocido que el PC después pueda comprobar, se le envian dichos datos al principio de la prueba, utilizando la comunicación UART. Se van a introducir en el campo *Datos* del protocolo, y se puede calcular su tamaño a partir del campo *Tamaño*, calculando el tamaño del campo de *Datos*, el cual es siempre el valor de tamaño - 4.

De esta forma, dado que TestStand es quien configura el mensaje que se envía por UART, también configurará el mensaje a escribir por I2C/SPI, y deberá a su vez configurar el tamaño de forma adecuada.

Para dar comienzo a la prueba, el PC envía el mensaje con los datos al DUT, y después espera hasta recibir una respuesta. La respuesta es el mensaje de confirmación, que el DUT envía cuando está listo para escribir. Cuando recibe la confirmación, el PC ejecuta la lectura de n Bytes a través de la interfaz a probar, siendo n tamaño - 4.

Dado que en este caso se está haciendo una lectura, se implementa un timeout, para que el PC no se quede eternamente esperando si la comunicación no es correcta. El tiempo de timeout también se configura como entrada para poder ser modificado por TestStand.

En caso de que el PC consiga leer los datos de la prueba, se verifica que los datos leídos sean iguales a los datos que se han enviado anteriormente por UART, en el subVI *CheckMsg*. Como en la prueba anterior se saca el resultado del test, *passed/failed*, junto con ambos arrays de datos, el enviado por UART y el recibido por I2C/SPI.

5.1.5 | Prueba de Escritura y Lectura SPI

A diferencia de I2C, SPI utiliza líneas diferentes para el envío y la recepción de datos, por lo que es posible realizar la lectura y escritura simultáneamente.

Por tanto, esta prueba va a consistir en una combinación de las anteriores dos pruebas. Para facilitar la ejecución de la prueba, se ha seleccionado que ambos mensajes sean iguales. Aun así, después se comprobará que las líneas funcionan por separado y que el funcionamiento es correcto.

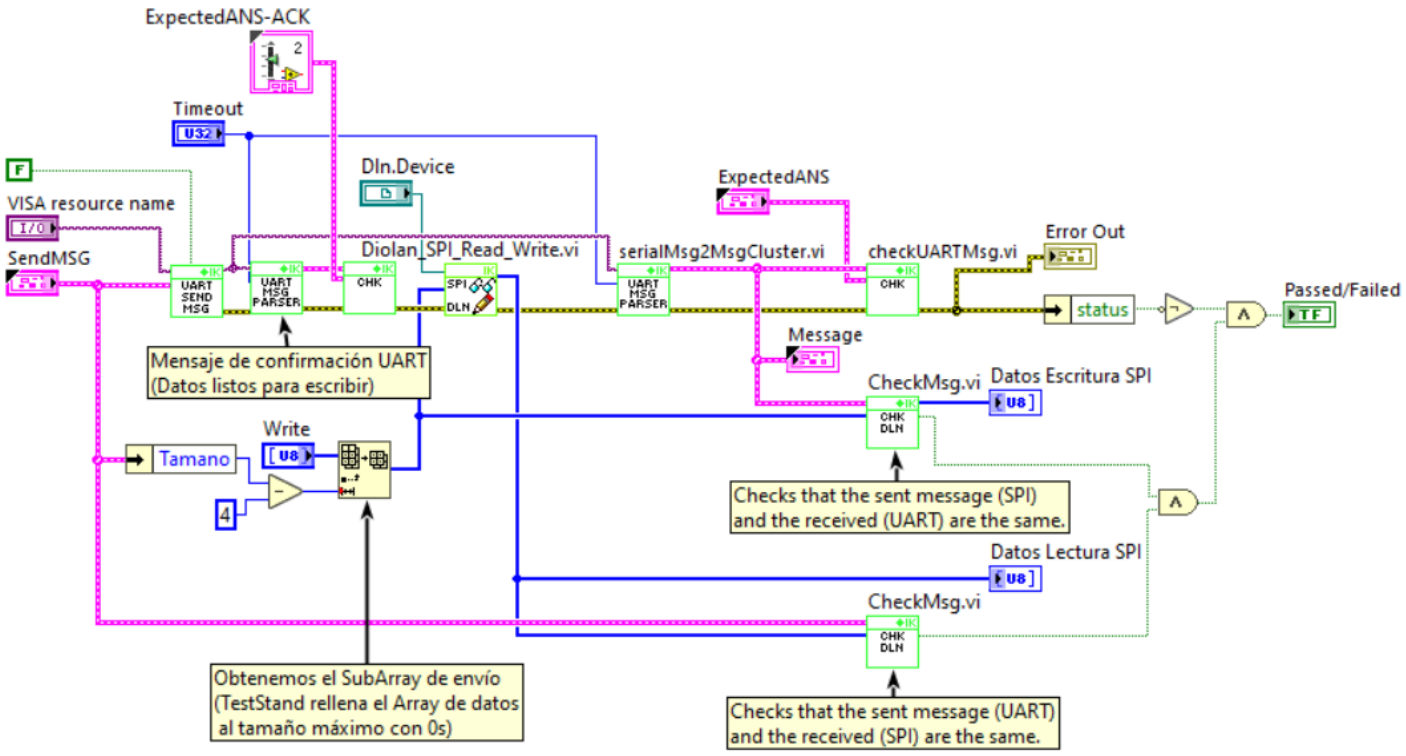

Figura 5.7: Prueba de Lectura y Escritura simultánea SPI

En la Figura 5.7 se puede observar el VI que ejecuta la prueba. La prueba comienza con el PC enviando a la DUT el mensaje que después tendra que escribir por SPI. Cuando la DUT responde con el mensaje de confirmación se ejecuta la lectura y escritura simultánea, subVI *Diolan_SPI_Read_Write*. Como se puede observar, este subVI toma como entrada el array de datos a enviar, de la misma forma que se hacía en la prueba de escritura, sacándolos del mensaje UART. Como salida tiene el array de datos leído por SPI. Ambos arrays deben tener la misma longitud, dado que la escritura y lectura simultánea lee un dato por cada vez que escribe.

Una vez ha terminado la transferencia de los datos, se pasa a verificarlos. En este caso, hay que realizar dos verificaciones.

Por un lado, se verifica que la lectura del DUT sea correcta, es decir, que los datos que el PC ha enviado por SPI y los que el DUT ha leído por SPI y devuelto por UART son iguales. Este bloque lo realiza el subVI *CheckMessage* superior.

Por otro lado, se verifica que la escritura del DUT sea correcta, es decir, que los datos que el PC envía por UART sean iguales a los leídos por SPI. Esto se realiza en el bloque inferior.

5.2 | Diseño de los tests en los DUTs

Para analizar el diseño de los tests en los micros, primero se va a analizar el diseño general para ambos micros, y después las diferencias que ha habido entre ambos.

En el caso de la Delfino, también ha habido que implementar las pruebas que la RISC-V tenía implementadas como base, es decir, las pruebas de lectura y escritura de GPIOs, la lectura del ADC y la generación de PWM.

5.2.1 | Test de Decoder de posición

Para los tests del decoder se han utilizado dos Timers. El primero de ellos es el Timer que se utiliza como Decoder. Para ello ambos micros tienen funciones especiales de configuración del Timer en sus drivers, para configurar que las señales A y B como entrada del Timer.

El Timer se configura en cuadratura, de manera que se actualice con ambos flancos de ambas señales.

El otro Timer se utiliza como periodo de muestreo. Por una parte se actualiza el valor de la posición en cada periodo de muestreo, y por otra parte sirve para medir el tiempo de ejecución de la prueba. Se ha configurado el timer para que el tiempo de muestreo sea de 1ms, el cual es adecuado para el control del motor que se quiere utilizar en la aplicación final, y además así el cómputo de los segundos es sencillo.

La prueba comienza cuando el comando obtenido es el correspondiente a esta prueba. Entonces, se lee del mensaje UART el tiempo de ejecución de la prueba. Se configuran ambos Timers y se inicializan, y se le suma un pequeño offset al tiempo de ejecución, para dar tiempo al PC a generar la señal.

Después, se envía un mensaje de confirmación al PC, el cual recibe LabVIEW y empieza a generar la simulación del encoder, como ya se ha observado en su diseño.

Mientras el tiempo de ejecución sea inferior al límite, el micro lee el valor del Decoder en cada

periodo de muestreo, y actualiza una variable que guarda la posición, utilizando la interrupción enerado por el Timer del periodo de muestreo.

Finalmente, cuando el tiempo supera el límite, se envía el mensaje final al PC, con el resultado de la prueba en el campo de datos.

En el caso de la RISC-V los Timers tienen 16 bits, por lo que el decoder sufre overflow entre periodos de muestreo a frecuencias altas. Para que esto no ocurra, se ha implementado una interrupción en la que se actualiza el valor de la variable cuando ocurra un overflow o underflow.

En el caso de la Delfino los Timers tienen 32 bits, por lo que no es necesaria la implementación de esta interrupción.

5.2.1.1 Interrupción de Overflow de Decoder en RISC-V

Para saber si la interrupción se ha generado por overflow o underflow, se utiliza el bit de dirección del Encoder de la manera que se puede observar en la Figura 5.8.

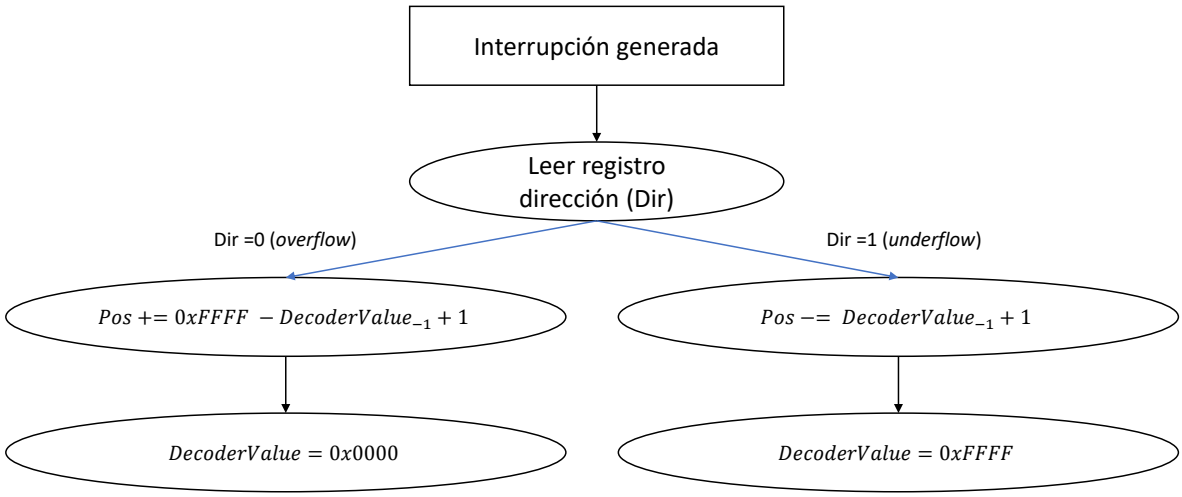

Figura 5.8: Funcionamiento de la interrupción del Decoder

Si tiene valor cero, la dirección es ascendente, lo que quiere decir que el contador está en valor 0xFFFF y ha entrado otro pulso positivo. Por tanto, los pulsos que se han sumado son la diferencia entre 0xFFFF y el valor anterior más uno, dado que ha entrado otro pulso tras 0xFFFF, lo que nos deja la siguiente ecuación.

$$Pos+ = 0xFFFF - DecoderValue_{-1} + 1$$

Dado que en el mismo periodo de muestreo puede ocurrir otra interrupción, es necesario reiniciar el valor de la variable DecoderValue a 0x0000. Si en cambio el bit tiene valor 1, la dirección es descendente, lo que indica que el contador tenía valor nulo, 0x0000, y ha detectado un pulso descendente en el encoder. En este caso, debemos restar a la posición el valor anterior del decoder +1, dado que el encoder ha decrementado hasta cero y ha entrado otro pulso adicional. En este caso, DecoderValue se reinicia a 0xFFFF, el valor máximo que puede obtener el decoder.

5.2.2 | Test de Decoder de velocidad

Para la implementación del Decoder de velocidad, se han utilizado también los 2 mismos Timers.

En este caso el desarrollo es un poco diferente.

Se empieza recibiendo el mensaje del test, el cual en este caso no tiene datos.

Después, se configuran ambos Timers.

Cuando pase un periodo de muestreo, se obtiene el primer valor de Decoder. Después se espera hasta el siguiente periodo de muestreo, y se vuelve aobtener otro valor del Decoder

Para calcular la velocidad del motor, se restan ambos valores y el resultado se divide entre el periodo de muestreo, 1ms, o lo que es igual, se multiplica por 1000.

Finalmente, el resultado de la prueba se envía al PC a través del mensaje UART.

Como en la anterior prueba, también es necesario implementar la interrupción en la RISC-V, para gestionar los overflows o underflows que ocurran.

5.2.3 | Test de Lectura I2C y SPI

Para los tests de lectura se ha utilizado un buffer de datos en ambos dispositivos. Se reserva la memoria para el buffer en la configuración inicial del micro, dándole el tamaño máximo posible, 250.

La prueba empieza cuando se recibe el mensaje del PC. Dado que el tamaño de la recepción puede variar, el mensaje de UART indica la cantidad de Bytes a leer por I2C/SPI. Se obtiene ese valor del primer Byte del campo *Datos* del protocolo.

Después, se configura la interfaz I2C/SPI para la lectura, configurando el buffer mencionado para la recepción y el tamaño de la recepción.

Cuando el micro esta preparado se envía un mensaje por UART al PC, el cual después envía los datos por I2C/SPI.

Finalmente, la DUT lee los datos del buffer que se ha configurado y los envía de vuelta por UART.

5.2.4 | Test de Escritura I2C y SPI

En este caso se sigue el procedimiento adverso. En este caso también se utiliza un buffer, el cual se inicializa en la configuración inicia del micro.

La prueba comienza cuando el micro recibe el mensaje UART, el cual en este caso tiene los datos de la escritura. El micro obtiene esos datos y los guarda en el buffer.

Después, se configura la interfaz I2C/SPI con el buffer de envío y el tamaño.

Cuando se ha configurado, se envía el mensaje de confirmación al PC.

Tras terminar la escritura, se envía un último mensaje de confirmación, para finaliza el test.

5.2.5 | Test de Lectura y Escritura simultánea SPI

En este caso, se juntan los dos anteriores tests. Se utilizan ambos buffers.

La prueba empieza con la recepción del mensaje UART. Este mensaje tiene los datos de escritura, los cuales también indican el tamaño de la prueba.

Esos datos se escriben en el buffer de escritura, y se vacía el buffer de lectura.

Se configura el SPI para realizar una lectura y escritura, utilizando ambos buffers. Cuando se termina la configuración, se envía el mensaje de confirmación al PC.

Cuando acaba la transmisión, los datos del buffer de lectura se colocan en el mensaje de UART, y se envía un último mensaje, con los datos de lectura.

5.3 | Diseño del entorno en TestStand

Tras haber implementado cada una de las pruebas en LabVIEW, se va a implementar el entorno automático utilizando TestStand. Este entorno permite realizar la verificación de las interfaces de los micros de una forma automática.

Además, como se verá en detalle más a continuación, dota al operador de muchas herramientas que hacen que la verificación además de automática sea útil en distintos escenarios, como por ejemplo la verificación en serie, la cual puede ser muy interesante en tiradas de hardware o cuando se han adquirido más de una unidad del mismo dispositivo.

Primero se va a analizar el diseño del entorno, y posteriormente se analizarán las diferentes herramientas de ejecución y de generación de reportes.

El entorno de verificación tiene la siguiente estructura, la cual podemos observar en la Figura 5.9:

- **Sección inicial, configuración:** En esta sección se configuran e inicializan la conexión del puerto serie y la interfaz Diolan. Primero se configuran los diferentes parámetros de la conexión serial, como el puerto del PC utilizado, el Baudrate y la paridad. Después se abre la conexión y se comprueba su correcto funcionamiento. Para esto el protocolo tiene un comando definido (0x00FD) en el que el DUT debe hacer hecho, confirmando así que la conexión se ha establecido correctamente. Finalmente se abre la conexión con el adaptador Diolan para terminar con la inicialización de la prueba.

- **Sección principal:** En esta sección se realizan las pruebas del test. Se genera una secuencia para cada prueba diferente. Cada sección puede constar de una o varias pruebas, teniendo todas las secciones mínimo una prueba con resultado. En el caso de las secciones con más de una prueba, la sección resultara satisfactoria si todas las pruebas individuales de la sección son satisfactorias.

  Los VIs que se ejecutan en cada sección deben de ser configurados. Se deben asignar valores a las entradas del VI y se debe especificar qué salida se utilizará para determinar si la prueba ha sido superada o no. Para ello, TestStand nos ofrece la posibilidad de utilizar variables locales o globales.

  Dentro de cada sección, además de ejecutarse los VIs que ejecutan las pruebas, también se realizan acciones adicionales, como la ejecución de bucles, como puede ser una estructura *for* o *while*, para la ejecución del mismo VI con diferentes valores, cambiando así el valor de diferentes variables locales. De esta manera se consigue testear la misma interfaz para diferentes puntos de operación en una misma sección. Esto se verá mejor detallado cuando se analicen las secciones.

Las variables globales se utilizan generalmente para los datos que se mantienen constantes durante diferentes secciones de la prueba. En el caso concreto de este escenario, se hace uso de ellas para pasar a los VIs las referencias de la interfaz del puerto serie y del adaptador Diolan.

- **Final de la prueba:** En esta sección se finaliza la prueba y se realiza la limpieza de las interfaces abiertas. Primero se ejecuta el comando de fin de test (0x00FE), el cual devuelve un dato booleano dependiendo de si la prueba se ha realizado correctamente o ha ocurrido algún error. En el caso del entorno de verificación no se va a hacer uso de ese dato dado que el propio entorno verifica si los tests se han realizado satisfactoriamente. Aun así, se ejecuta para cumplir con el protocolo establecido. Finalmente, se cierran las conexiones de las interfaces, primero la del Diolan y después la del puerto serie.

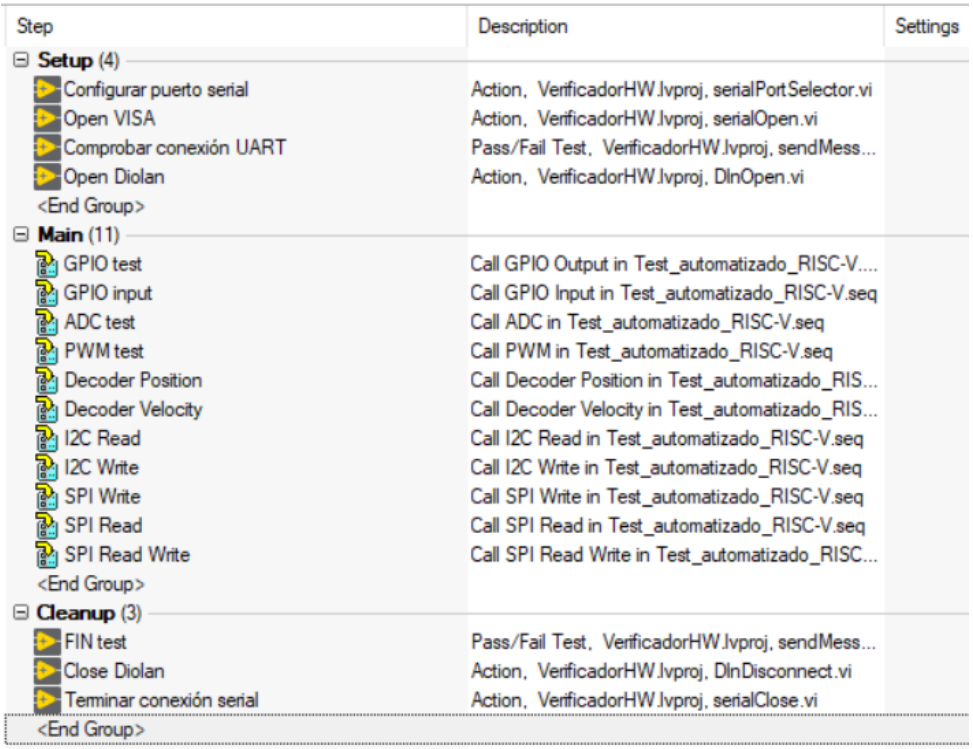

Figura 5.9: Secuencia principal del entorno de verificación

Se va a utilizar la primera sección, la de la posición del decoder, para explicar el funcionamiento de TestStand.

5.3.1 | Secuencia de prueba de Decoder de posición

La interfaz de usuario de TestStand se divide en 3 partes principales.

Por un lado, está la parte donde se definen los pases que se realizan en la sección. Por otro lado, hay otra ventana para crear y modificar variables. Finalmente, hay una ventana para configurar la ejecución de los VI y los bucles.

Para verificar el correcto funcionamiento del decoder de pocisión, se han realizado varias ejecuciones del VI, con diferentes parámetros. Se ha optado por realizar primero las pruebas en una dirección, incrementando la frecuencia, y después se realiza la misma operación en la otra dirección.

En la Figura 5.10 se puede observar la primera parte de TestStand, la estructura de los pasos de la sección. En este caso está formada por un bucle *for* externo, para la selección de ambas direcciones, y otro *for* interno para las diferentes frecuencias. Dentro de ambos bucles se encuentra el paso que llama al VI.

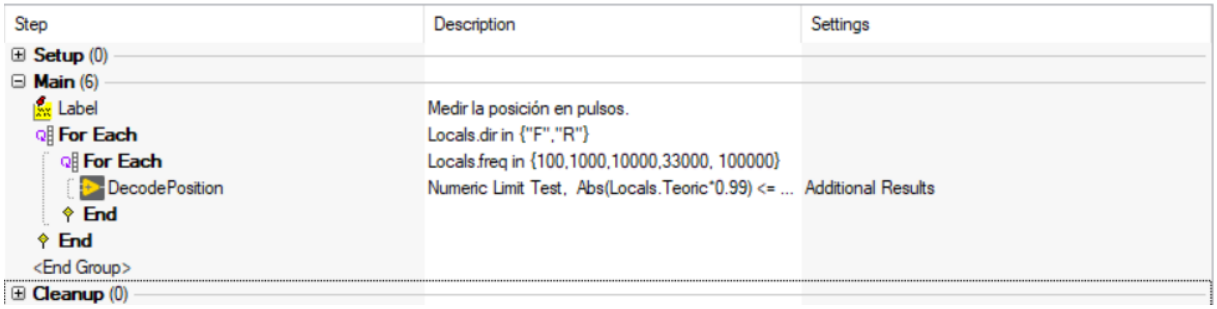

Figura 5.10: Sección de las pruebas de posición del Decoder

Se ha decidido empezar con frecuencias bajas e ir incrementando, dado que el decoder puede tener problemas para adquirir señales de alta frecuencia, y así se puede observar a partir de que frecuencias empieza a fallar, si así ocurriese.

En el caso concreto del motor que se quiere utilizar, cuenta con un encoder de 1000 pulsos por vuelta y una velocidad máxima de 6000rpm, por lo que como se ha mencionado en el diseño de la prueba en la sección 2.2.1, la frecuencia máxima del encoder será de 100KHz. La frecuencia mínima de la prueba ha sido de 100Hz, lo que equivale a 6rpm.

En la Figura 5.12 se puede ver la ventana de configuración de parámetros.

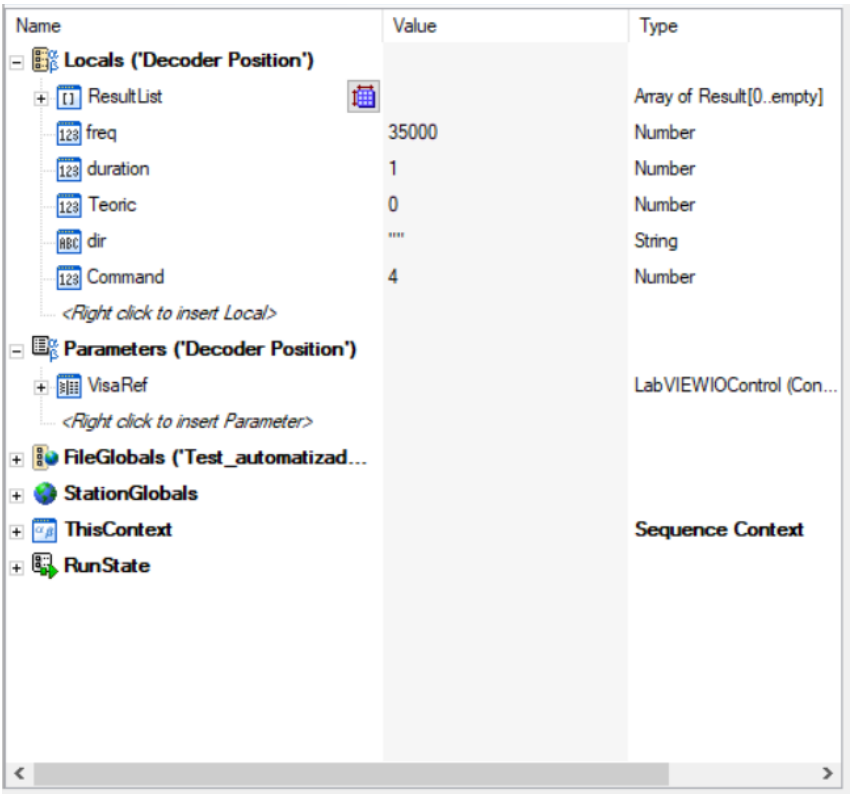

Figura 5.11: Ventana de configuración de parámetros de TestStand

Se puede observar que la ventana está dividida en varios grupos. Los que más se han utilizado son los dos primeros.

En primer lugar está el grupo de las variables locales. Estas variables son locales para la sección. Se pueden utilizar para asignarlas a entradas de los *step*, o para guardar las salidas. Esto permite hacer cosas realmente interesantes, por ejemplo, permite realizar una ejecución de un VI, guardar la salida en una variable local, y después ejecutar el procesamiento en otro lenguaje de programación, como un programa de Python, C o Matlab.

Después, se encuentran los parámetros globales. En esta prueba solo hace falta la referencia del puerto UART, el que se guarda en VisaRef, en las pruebas de I2C y SPI que utilizan el adaptador Dln-2 también se verá como se utiliza su referencia.

En cuanto a las variables locales, se puede observar cómo se utilizan los diferentes parámetros que se han explicado durante el diseño de las pruebas.

El primero, ResultList, es un parámetro que TestStand utiliza para guardar los diferentes resultados que se obtienen en la sección.

Los siguientes son los parámetros que se le pasan al VI, como se verá a continuación en la configuración del VI. En la Figura 5.10 se ve como los bucles *for* operan sobre dichas variables, cambiando su valor.

Finalmente, se va a analizar la ventana de configuración de VI.

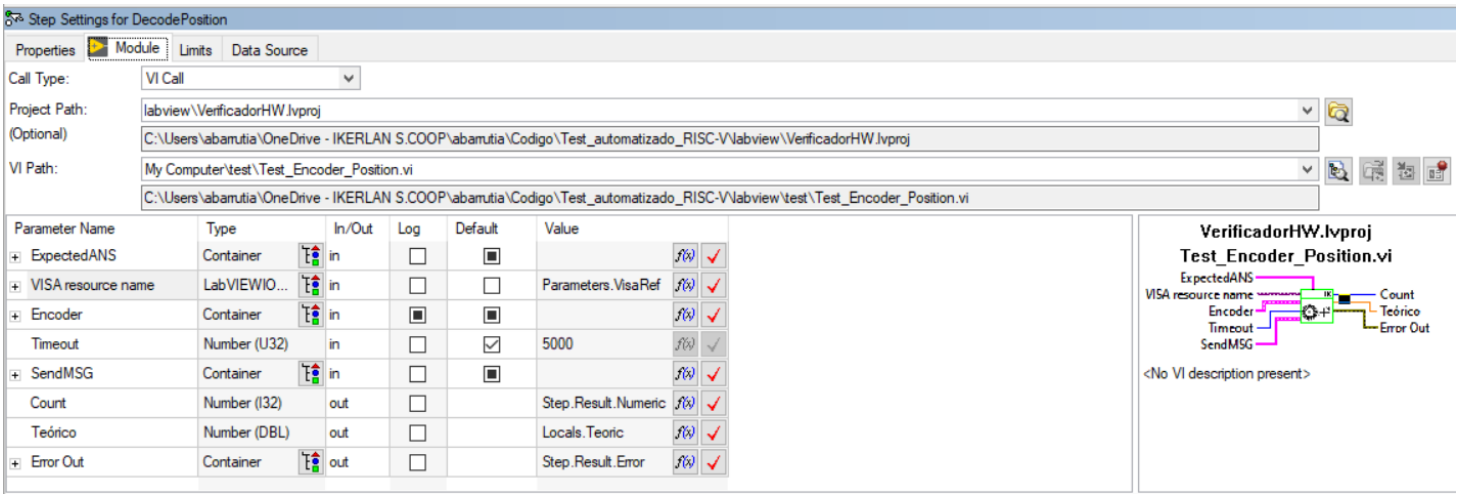

Figura 5.12: Ventana de configuración de ejecucción de TestStand

En esta ventana se pueden configurar los parámetros de entrada del VI, y guardar las salidas. En la parte derecha se puede ver el icono del VI con las señales de entrada y salida.

Por un lado, hay que configurar el mensaje UART, introduciendo el comando, el tamaño del mensaje y los datos. En la configuración de los datos se pueden introducir variables, como se puede ver en la Figura 5.13. Se puede ver como en el comando y en el primer Byte de datos se introducen las variables locales del comando y la duración de la prueba respectivamente. Además, se pueden dejar valores por defecto, como en la cabecera y el CRC, los cuales los genera el VI de envío de LabVIEW, como se veía en su implementación.

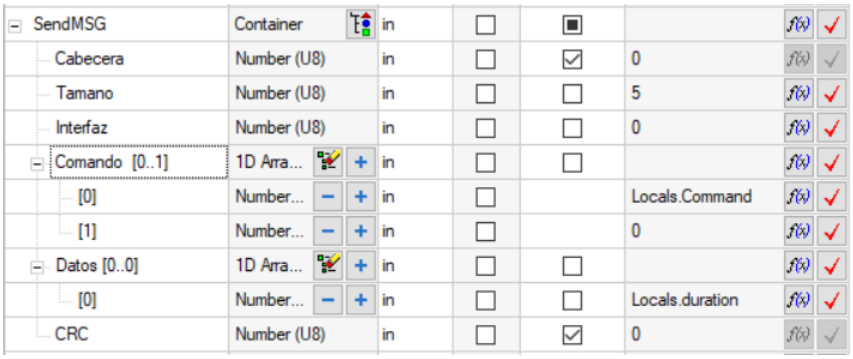

Figura 5.13: Configuración del mensaje UART

Aparte de las entradas y salidas de la prueba, también se han de configurar los límites de la prueba, es decir, los resultados que se deben cumplir para que la prueba se de por exitosa. Se pueden elegir 3 diferentes modos, el modo booleano, el modo numérico, y el modo de texto.

En el modo booleano, una salida booleana determina si el test ha sido superado o no. En el modo de texto, se pueden realizar comparaciones entre un texto objetivo y el texto recibido en

la prueba. En el modo numérico, el utilizado en esta prueba, se verifica que el valor obtenido esté dentro de unos límites permitidos, pudiendo realizar diferentes tipos de comparaciones.

Como se comentaba en el diseño de la prueba en LabVIEW, sección 2.2.1, el VI saca a TestStand el resultado de la prueba, y el valor teórico de los pulsos (los generados por la cDAQ). Además, se estableció que el error debía ser inferior al 1 %. Por tanto, realiza la configuración de la Figura 5.14. Se establece que la comparación sea de tipo >= <=. De este modo TestStand deja elegir el límite inferior y el superior.

Comparison Type	GELE (>= <=)
Low	Abs(Locals.Teoric*0.99)
High	Abs(Locals.Teoric*1.01)

Figura 5.14: Configuración del resultado de la prueba

5.3.2 | Secuencia de prueba de Decoder de velocidad

Para la secuencia de lectura de velocidad se ha seguido la misma estructura que en la secuencia de posición, la cual se puede observar en la Figura 5.15.

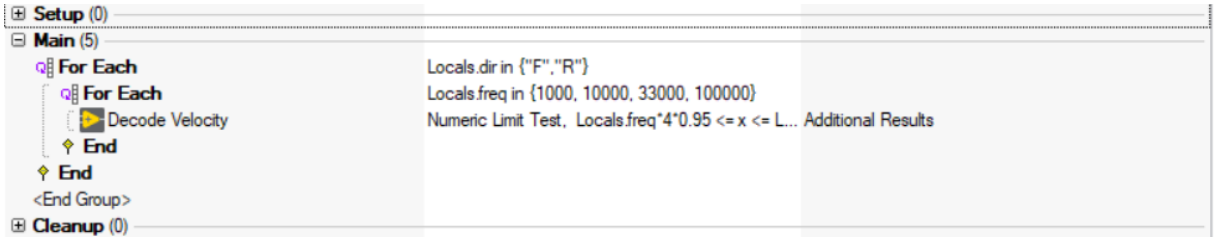

Figura 5.15: Steps de la prueba de velocidad del encoder

En este caso, las frecuencias son algo diferentes. Esto se debe a que si la velocidad del Encoder simulado es muy baja, puede darse el caso de que el Decoder no reciba ningún pulso entre dos periodos de muestreo.

En este caso concreto, el periodo de muestreo es de 1ms, por lo que el error de cuantización es de 1pulso/1ms, lo que equivale a 1000Hz. Pero dado que se utiliza cuadratura x4, esta frecuencia es 4 veces menor, 250Hz. Por tanto, si la frecuencia es menor a 250HZ, puede ocurrir que no se reciba ningún pulso entre dos periodos.

El motor tiene un encoder de 1000 pulsos/revolución, por lo que esto equivaldría a una velocidad de 0,25 revoluciones/s o lo que es igual, un error de cuantización de 15rpm. Hay que tener en cuenta que el método utilizado para el cálculo de la velocidad favorece las altas revoluciones, y que el motor puede alcanzar las 6000rpm, por lo que este valor es relativamente pequeño.

Por evitar problemas de cuantización se ha subido la frecuencia mínima a 1000Hz.

En este caso, la velocidad real de la prueba no se saca por el VI, sino que se calcula en el propio TestStand, como se puede ver en la Figura 5.16. La velocidad real es la frecuencia * 4, dado que se usa un decoder de cuadratura x4.

Comparison Type	GELE (>= <=)
Low	Locals.freq*4*0.99
High	Locals.freq*4*1.01
Units	
Numeric Format	<Default>

Figura 5.16: Verificación de los resultados de la prueba de velocidad

5.3.3 | Secuencias de lectura I2C y SPI

Dado que la estructura de ambas secuencias es igual, se van a estudiar juntas tanto la lectura I2C como la lectura SPI.

En la Figura 5.17 se puede observar la estructura de la secuencia. En este caso, también se han utilizado 2 bucles *for*. El primero de ellos sirve para repetir la prueba varias veces, se ha establecido que cada una se ejecute 6 veces. El bucle interno, se utiliza para llamar generar los datos del test aleatoriamente. Para ello se utiliza la función random de TestStand, y se rellena uno de los datos cada vez que se ejecuta el bucle.

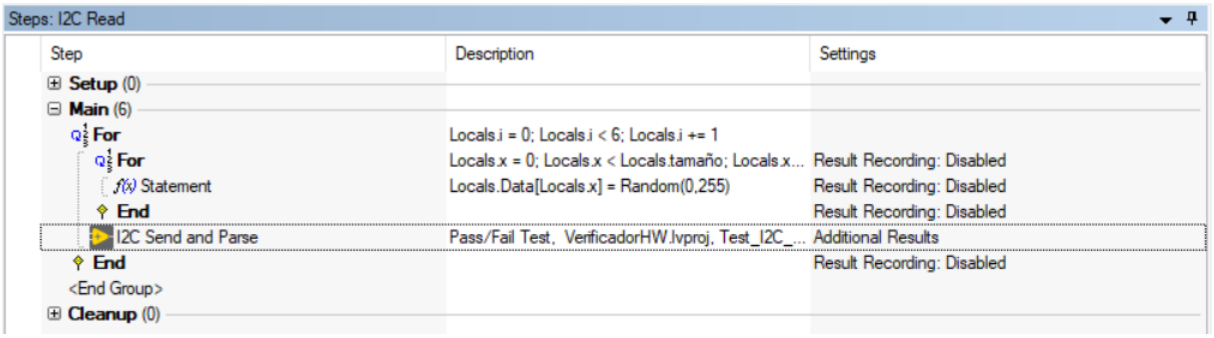

Figura 5.17: Steps de la prueba de lectura I2C

El bucle *for* va iterando sobre la variable x, la cual se utiliza de índice, hasta llegar al valor especificado en la variable *tamaño*. En la Figura 5.17 se puede ver, en el *step* llamado *statement* como se le asigna un valor generado por la función random a la variable local *Data*, en la posición x.

Se ha elegido que el tamaño sea de 50 Bytes, dado que así el driver tendrá que ejecutar varias lecturas seguidas. Normalmente, los drivers incluyen funciones para el envío y lectura de 1,

8 y 16 Bytes, y el envío de n Bytes, el cual se ejecuta utilizando las anteriores. Por tanto, es interesante transmitir más de 16 Bytes, para ver que no ocurren problemas.

Tras generar todos los valores aleatorios, se ejecuta el VI de la prueba. Este VI toma como entrada la variable de Datos y la escribe por I2C. Finalmente, se toma como resultado la salida booleana programada en el VI, como se ha visto en la sección 5.1.3.

Además, también se guardan los datos que el DUT envía a través de UART, los resultados de la lectura, para después incluirlos en el reporte.

5.3.4 | Secuencias de escritura I2C y SPI

La estructura de la secuencia de escritura es idéntica a la de lalectura. La diferencia es el tratamiento de los datos, dado que esta vez en lugar de escribirlos por las interfaces I2C/SPI, se envían por UART para que las escriba el DUT. Por tanto, se vuelven a generar datos aleatorios para la prueba, pero esta vez en lugar de mandárselos al VI por una entra, se colocan en el mensaje UART.

En la Figura 5.18 se puede observar como la variable local Data, en la cual se guardaban los datos aleatorios generados, se introduce en el campo *Datos* del mensaje UART.

Parameter Name	Type	In/Out	Log	Default	Value
SendMSG	Container	in	■	■	
Cabecera	Number (U8)	in	☐	☑	0
Tamano	Number (U8)	in	☐	☐	Locals.tamaño + 4
Interfaz	Number (U8)	in	☐	☐	2
Comando [0..1]	1D Array (U8)	in	☐	☐	Locals.Command
Datos [0..3]	1D Array (U8)	in	☑	☐	Locals.Data
CRC	Number (U8)	in	☐	☑	0

Figura 5.18: Configuración del mensaje UART de la prueba de escritura I2C

Como en el caso anterior, se evalúa la prueba dependiendo del resultado booleano que genera el VI. Además, se guardan los valores del mensaje enviado por UART, y el recibido por I2C/SPI.

En la Figura 5.18 se puede observar que se ha activado la casilla *Log* del campo de Datos.

5.3.5 | Secuencias de lectura y escritura simultánea SPI

En este caso también, la estructura seguida es igual. Se ha decidido generar una sola vez los datos aleatorios, y utilizarlos para la escritura y para la lectura, para reducir el tiempo de ejecución.

Por tanto, se genera el mensaje aleatorio como en los dos casos anteriores, y después se envía a ambas partes. Por un lado a la entrada de la escritura SPI del PC, y por el otro al mensaje UART, para la escritura del DUT.

Parameter Name	Type	In/Out	Log	Default	Value
SendMSG	Container	in	■	■	
Cabecera	Number (U8)	in	☐	☑	0
Tamano	Number (U8)	in	☐	☐	Locals.tamaño + 4
Interfaz	Number (U8)	in	☐	☐	3
Comando [0..1]	1D Array (U8)	in	☐	☐	Locals.Command
Datos [0..3]	1D Array (U8)	in	☑	☐	Locals.Data
CRC	Number (U8)	in	☐	☑	0
VISA resource name	LabVIEWIO...	in	☐	☐	Parameters.VisaRef
Timeout	Number (U32)	in	☐	☑	5000
Write [0..empty]	1D Array (U8)	in	☑	☐	Locals.Data

Figura 5.19: Configuración de los parámetros de la prueba de escritura y lectura simultánea SPI

En la Figura 5.19 se puede observar como tanto l entrada *write*, la cual escribe por SPI en el PC, como el campo de Datos del mensaje UART tienen como asignada la variable Datos, la cual guarda los datos aleatorios generados.

5.3.6 | Generación de reportes

Una de las mayores ventajas por las cuales utilizar TestStand es su generador de reportes.

Al ejecutar el entorno, el generador de reportes va automáticamente guardando los datos de las pruebas y sus resultados. Como ya se ha visto en alguna sección, en la configuración de los parámetros se puede seleccionar que parámetros se quieren incluir, con hacer un solo click en la casilla de *log*. Además de con los parámetros de los VIs, esto también se puede realizar con los demás *steps*, lo que permite realizar cosas muy interesantes.

Al iniciar TestStand, la aplicación pide al operador el nombre de usuario y la contraseña, con esto se pueden restringir algunas de las pruebas, o directamente requerir una contraseña para poder acceder al entorno de test. En este caso no se ha restringido el uso, y se a seleccionado el usuario de administrador.

Cuando se ejecuta el entorno, TestStand pide al operador introducir el número de identificación de la DUT al usuario, con la ventana emergente que se observa en la Figura 5.20.

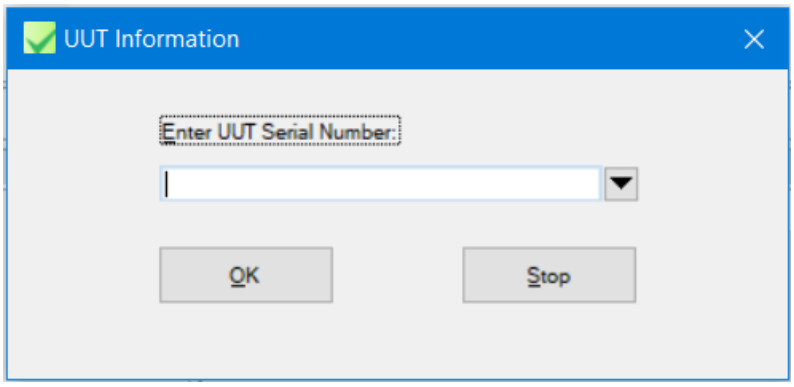

Figura 5.20: Selección del número de identificación del DUT

Como ya se ha mencionado anteriormente, TestStand permite realizar ejecuciones en serie. Para ello, esta misma ventana emergente aparece al final de cada ejecución completa del entorno. Se puede indicar un nuevo número de identificación y realizar la prueba para un nuevo dispositivo, o parar la ejecución del entorno.

Tras haber ejecutado el entorno en todos los DUT y seleccionar Stop, TestStand automáticamente generará el reporte de las pruebas. En dicho reporte se muestra al principio una cabecera informativa, con el número de identificación del DUT, si ha pasado el entorno, y el tiempo de ejecución, como se observa en la Figura 5.21

UUT Report

Station ID	21310-243
Serial Number	NONE
Date	lunes, 25 de abril de 2022
Time	12:25:45
Operator	administrator
Execution Time	31.7916 seconds
Number of Results	263
UUT Result	Passed

Figura 5.21: Cabecera del reporte de TestStand

El reporte se puede configurar de diferentes maneras. Por un lado, se puede seleccionar el formato de salida. Lo más cómodo es utilizar el formato por defecto de TestStand y utilizar el propio programa para leerlo, dado que ofrece la posibilidad de minimizar secuencias y *steps*, haciendo la lectura más sencilla. Además, se puede configurar para que el reporte se exporte automáticamente en formato PDF.

Además, se puede configurar la manera de mostrar los diferentes DUTs. La opción por defecto los muestra en serie, es decir, primero muestra los resultados del entorno entero para un DUT, después para el segundo, etc.

La otra opción es enseñarlos por secciones de ejecución. En este caso, en lugar de muestra los

resultados del entorno entero del primer DUT, se muestran los resultados de todos los DUTs para una sección, antes de pasar a la siguiente.

También se ofrece la opción de generar un reporte para cada DUT, lo cual puede ser interesante en caso de una verificación en serie con muchos DUTs y muchas pruebas, dado que sino el reporte generado sería demasiado largo para analizarlo fácilmente.

Otra opción interesante es la exclusión de resultados. Se puede elegir excluir las pruebas con resultado positivo. De esta manera se consigue que el reporte sea más corto, y los datos que se muestren sean solo de las pruebas fallidas. De este modo es más fácil centrarse en los errores ocurridos.

También se puede seleccionar el mostrar solo las pruebas superadas, y excluir los *steps* de control de flujo, como los bucles *for* utilizados en varias secciones.

5.4 | LabVIEW NXG

Después de implementar el entorno utilizando LabVIEW para la ejecución de TestStand, se ha implementado con LabVIEW NXG para comparar el funcionamiento de ambas familias del software de National Instruments.

Para ello, se ha utilizado el software de conversión de código de National Instruments, el cual permite convertir el proyecto de LabVIEW a un nuevo proyecto de LabVIEW NXG.

Esta herramienta funciona bastante bien, pero debido a que no todos los bloques antiguos pueden ser convertidos a LabVIEW NXG por falta de compatibilidad, la herramienta genera unos vacíos que se deben de completar.

En cuanto a la librería del adaptador Dln-2, no se ha conseguido importar a LabVIEW NXG, por falta de soporte para ese tipo de librerías. Por tanto, no se puede utilizar el adaptador con LabVIEW NXG. Aun así, se ha continuado con el resto de las pruebas.

Después de migrar el proyecto, se ha implementado el entorno de TestStand utilizando el nuevo proyecto de NXG, como se puede ver en la Figura 5.22

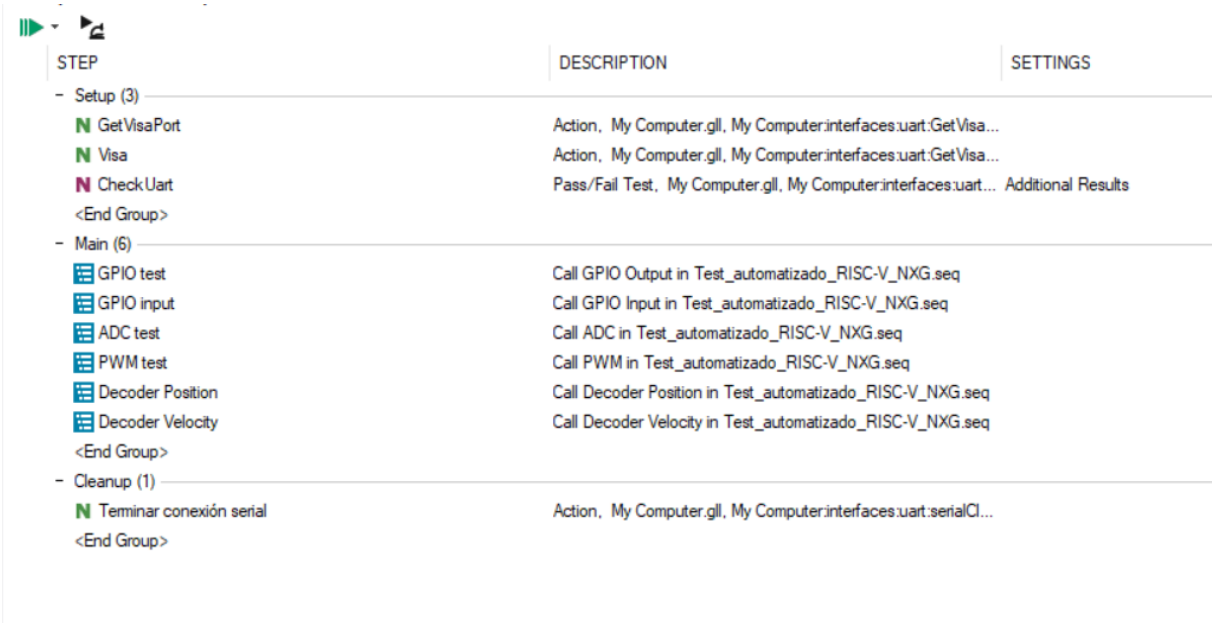

Figura 5.22: Secuencia principal de TestStand utilizando LabVIEW NXG

LabVIEW NXG tiene algunas innovaciones interesantes. Por un lado, se ha incluido la funcionalidad de NI MAX en el propio NXG. NI MAX permite al usuario gestionar los dispositivos de National Instruments conectados al ordenador, como la cDAQ, y observar los módulos disponibles, actualizar su software, generar dispositivos simulados, etc.

NXG implementa una nueva ventana de sistema, en el que se puede observar en una venta

el sistema conectado al PC con todos sus dispositivos, y en otra ventana se puede diseñar un sistema, seleccionando cualquiera de los componentes y dispositivos de NI.

Estas dos ventanas se llaman *live* y *design* respectivamente. Se puede utilizar la ventana *design* para generar un sistema simulado, o para generar el sistema en el que finalmente se va a ejecutar la aplicación.

Aunque en teoría esto implique grandes ventajas de desarrollo, lo que se observa en la realidad es que su implementación no es buena, y provoca una gran cantidad de errores. La aplicación muchas veces se confunde entre una versión del diseño o la otra, y el programa no consigue ejecutarse.

Además, debido a este error había veces en las que la cDAQ se desconectaba, y dejaba d funcionar por completo, llegando al punto de tenerla que reiniciar.

Por otro lado, la optimización de la propia aplicación es muy mala. Iniciar LabVIEW NXG podía tardar hasta 5 minutos en el portátil utilizado, el cual no tiene ningún problema para arrancar la versión normal de LabVIEW y cumple de sobra con las especificaciones requeridas para LabVIEW NXG. Además, en el proceso de arrancado de la aplicación el portátil se quedaba congelado varias veces.

Aun así, con todos estos fallos, se ha conseguido ejecutar el entorno de test, consiguiendo verificar también los micros haciendo uso de LabVIEW NXG, excepto por las interfaces I2C y SPI.

6 | Dificultades

Durante el desarrollo del proyecto han surgido varias dificultades.

Las mayores dificultades han venido por el uso de LabVIEW NXG. El Software tenía una gran cantidad de bugs, los cuales afectaban no solo al rendimiento de la aplicación, pero también al rendimiento del PC. Muchas veces el PC se quedaba colgado o daba pantallazos azules, lo cual hacía que el uso de la aplicación no fuese nada cómodo.

Con la implementación de las pruebas en los micros también hubo unos pequeños problemas, sobre todo con el ADC.

Los problemas surgieron debido a las configuraciones externas de las placas del desarrollo, y no debido al software.

Por un lado, en el trabajo que se tomo como punto de partida, se había conseguido hacer uso del ADC, pero solo hasta tensiones de 1,5V. Esto ocurrió dado que se estaba utilizando una entrada analógica la cual estaba conectada a un potenciómetro, que servía para hacer pruebas con el voltaje de referencia del ADC. Por tanto, el rango de adquisición se veía disminuido. Para solucionarlo bastó con cambiar la entrada analógica utilizada, cogiendo una que se conectase directamente al micro.

Por el otro lado, en la Delfino ocurría algo similar. Por defecto, el voltaje de referencia de los ADCs estaba a 3V en lugar de 3,3.

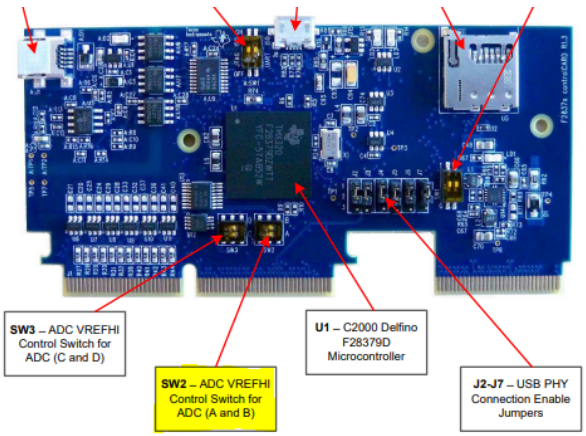

Figura 6.1: Switches de control de la Delfino

Esto ocurría porque la tarjeta de desarrollo tiene dos opciones de referencia para los ADCs, la habitual 3,3, y una referencia de 3V muy exacta que genera la tarjeta. No se mencionaba en el User Guide [15] que la referencia por defecto es la de 3V, se mencionaba en el manual de la tarjeta de desarrollo [16]. Por lo que hasta cambiar la referencia a 3,3V las medidas se computaban con un offset, por la diferencia de señal de referencia. Este cambio se puede realizar a través del switch SW2 que se observa en la Figura 6.1.

Por otra parte, los drivers del envío SPI no funcionaban correctamente. Al enviar mensajes de más de 16 Bytes, el driver enviaba los datos de 16 en 16 Bytes, y era muy común que entre dos envíos se perdiese algún Byte. Para solucionarlo se ha implementado la comunicación SPI utilizando Polling es decir, colocando los Bytes en el buffer a medida que hubiese hueco disponible.

7 | Resultados

Tras haber implementado y realizado en entorno de test en ambas tarjetas de desarrollo, los resultados obtenidos son satisfactorios.

Se ha conseguido desarrollar un entorno de test para la validación de las interfaces necesarias para el control de un motor PMSM. Además, se ha comprobado que el entorno de test es fácilmente ampliable, permitiendo así verificar en serie tiradas de fabricación.

Se han verificado las interfaces de ambas tarjetas, concluyendo que ambas son válidas para realizar el control del motor.

Tras finalizar con el entorno de test para ambas tarjetas, se ha implementado el control FOC en el motor, y se ha conseguido ajustarlo para que funcione de forma correcta.

En cuanto a los micros, durante la programación de ambos se ha observado que los drivers de la Delfino están más pulidos, y que en el caso del control del motor quizá sea mejor opción. Aun así, hay que tener en cuenta que la Delfino es una tarjeta especialmente diseñada para el control de motores, y la RISC-V una tecnología nueva y emergente, por lo que puede ser interesante continuar su crecimiento de cerca.

En cuanto a LabVIEW NXG los resultados han sido muy decepcionantes. Es cierto que se ha conseguido ejecutar el entorno de verificación, pero la gran cantidad de dificultades que han surgido hacen que no merezca la pena. Además, NI va a dejar de darle soporte.

Hay que reconocer que NXG tenía alguna idea buena, algunas de las cuales se van a mantener como productos separados, como el módulo Web NXG. Este módulo permite programar aplicaciones Web gráficas, utilizando la forma habitual de programar LabVIEW, siendo el resultado final una Web igual que el panel frontal de LabVIEW, el cual ejecuta el diagrama de bloques con lenguaje HTML. Este módulo ha pasado a llamarse *Software de Desarrollo Web G*.

Aun así, los aspectos negativos son muchos. Dejando de lado los bugs de los que se ha hablado en el diseño, los tiempos de carga son excepcionalmente lentos. Por una parte, la carga del propio programa es muy lenta y tediosa, dado que bloquea el propio ordenador. Por otra parte, la primera ejecución de un VI siempre es lenta, haciendo que la primera ejecución del entorno tarde entorno al triple que con LabVIEW. Una vez ejecutada la primera vez, se queda guardado en memoria y la ejecución coge tiempos normales, parecidos a los de LabVIEW.

Finalmente, el protocolo de comunicación de tamaño variable ha dado muy buenos resultados. Se ha mantenido el corte tamaño de mensaje para las pruebas básicas como el ADC o el Decoder, y ha permitido realizar envíos de grandes cantidades de datos para las interfaces SPI e I2C, lo que ha permitido encontrar el error en los drivers del SPI de la Delfino.

8 | Líneas futuras

Tras haber implementado el entorno de test para la verificación de las interfaces necesarias del motor, y después haber implementado el control FOC, se abren varias líneas futuras.

Por un lado, queda terminar el montaje de la maqueta *back to back* utilizando la RISC-V. Aunque se ha controlado un motor, en dicha maqueta hay que controlar dos, el que sigue la referencia de velocidad, y el que genera una carga al primero. La Delfino cuenta con dos procesadores, por lo que se utilizaba uno para cada motor. En el caso de la RISC-V habría que estudiar si es posible realizar todo con una tarjeta, o si se necesitarían dos tarjetas iguales por tiempos de ejecución. Si hiciese falta otra tarjeta, se podría utilizar el entorno para verificar su funcionalidad, y después cargar el control que se ha desarrollado para el primer motor, y modificarlo para generar una carga.

Por otro lado, queda la expansión del entorno de test. Este entorno puede ser muy interesante para verificar tiradas de fabricación de Hardware, por lo que se podría expandir con ese objetivo.

Bibliografía

[1] C.-Y. Huang and M. Lyu, "Optimal release time for software systems considering cost, testing-effort, and test efficiency," IEEE Transactions on Reliability, vol. 54, no. 4, pp. 583–591, 2005.

[2] R. Schuwer, M. van Genuchten, and L. Hatton, "On the impact of being open," IEEE Software, vol. 32, no. 5, pp. 81–83, 2015.

[3] S. Liu and S. Nakajima, "Automatic test case and test oracle generation based on functional scenarios in formal specifications for conformance testing," IEEE Transactions on Software Engineering, vol. 48, no. 2, pp. 691–712, 2022.

[4] N. Instruments, "Our commitment to labview as we expand our software portfolio." https://forums.ni.com/t5/LabVIEW/Our-Commitment-to-LabVIEW-as-we-Expand-our-Software-Portfolio/td-p/4101878, Nov. 2020.

[5] A. Amonarriz, "Diseño y desarrollo de un entorno de test automatizado para una plataforma risc-v," 2021.

[6] N. Instruments, "Ni cdaqtm-9181/9184/9188/9191 user manual." https://www.ni.com/pdf/manuals/372780k.pdf, Feb. 2017.

[7] N. Instruments, "Ni 9401 datasheet." https://www.ni.com/pdf/manuals/374068a_02.pdf, Dec. 2015.

[8] N. Instruments, "Ni 9263 datasheet." https://www.ni.com/pdf/manuals/373781b_02.pdf, Mar. 2016.

[9] Diolan, "Dln2 user manual." https://dlnware.com/user/User-Manual.

[10] GigaDevice, "Gd32vf103 risc-v 32-bit mcu datasheet." http://www.gd32mcu.com/data/documents/shujushouce/GD32VF103_Datasheet_Rev%201.1.pdf, Sept. 2019.

[11] Ikerlan, "Códigos de test del protocolo de comunicaciones pc-sut," modified 2022.

[12] Ikerlan, "Códigos de error del protocolo de comunicaciones pc-sut," modified 2022.

[13] Ikerlan, “Protocolo de comunicaciones pc-sut,” modified 2022.

[14] A. Bezbaruah, B. Pratap, and S. B. Hake, “Automation of tests and comparative analysis between manual and automated testing,” in 2020 IEEE Students Conference on Engineering Systems (SCES), pp. 1–5, 2020.

[15] T. Instruments, “Tms320f2837xd dual-core microcontrollers technical reference manual.” https://www.ti.com/lit/ug/spruhm8i/spruhm8i.pdf, Dec. 2013.

[16] T. Instruments, “Delfino™ tms320f28379d controlcard r1.3 user guide.” https://www.ti.com/lit/ug/sprui76a/sprui76a.pdf, Mar. 2017.