INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS

HERRAMIENTA DE INSERCIÓN DE ERRORES EN PROTOCOLO LIN

Tesina que para obtener el grado de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presentan: López Preciado Juan Ramón

Director: Martínez Guerrero Esteban

San Pedro Tlaquepaque, Jalisco. 6 de Diciembre de 2016.

Índice

Tabla de imágenes

1 Agradecimiento

En el presente trabajo de tesis agradezco primeramente a Dios por brindarme una vida llena de aprendizajes, experiencias y felicidad.

Agradezco a mi esposa Tania Belinda por su comprensión, apoyo constante por estar a mi lado compartiendo mis alegrías, angustias y por el estimulo para que me supere día a día.

Agradezco a mis padres Juan y Rosalba por su apoyo, consejos, ánimo y bendiciones.

Agradezco al Consejo Nacional de Ciencia y Tecnología (CONACYT) y a Continental por su apoyo económico brindado para el estudio de la Especialidad en Sistemas Embebidos y en la realización de este proyecto de tesis. Gracias por la ayuda y por creer en mí.

Agradezco al ITESO por abrirme las puertas al conocimiento, darme la oportunidad de estudiar y alcanzar un peldaño más en mi vida profesional. Gracias a todos los maestros que intervinieron en mi formación por que con sus exigencias me prepararon y me dieron armas para enfrentar la competencia del mundo laboral.

2 Resumen

Una forma de probar la robustez de los *drivers* de comunicación LIN es insertando errores en la red de LIN para ver el comportamiento del *driver* bajo prueba, es decir que tenga un manejo de errores de acuerdo a alguna petición del cliente o requisito, por ejemplo, que no haya *resets* en el *software* o errores similares.

Actualmente hay algunas herramientas que introducen errores en el bus de LIN como el J8115A LIN *Tester* de Agilent (inserta errores en LIN) o el hardware de prueba VH1150 de Vector. Estas herramientas son muy completas, pero su interfaz de control es compleja, además de que son caras y necesitan licencias de software. Para la automatización de secuencias de prueba es necesario tener un profundo conocimiento de la interfaz de tal software.

Para desarrollar un sistema LIN no se requieren herramientas o equipos especializados, sólo hardware que soporte el protocolo LIN, mientras que, por otro lado para probar la implementación LIN se requiere especialización, por eso, algunas de las pruebas no pueden ser automatizadas, lo que lleva a hacer más largos los tiempos de prueba y puede provocar falsos resultados positivos o negativos debido a errores humanos.

La principal meta de este proyecto es crear un procedimiento capaz de insertar errores en el protocolo LIN a bajo costo, fácil de usar, pero precisa. Este método permitirá crear casos de prueba para robustecer drivers de LIN, ya que no todos los microcontroladores tienen LIN nativo además ayudara a encontrar deficiencias en los driver de LIN y corregirlos en etapas tempranas del proceso de desarrollo, disminuyendo los costos de corrección del error, aumentando la calidad y fiabilidad de los componentes de LIN.

3 Abstract

For testing purposes, currently, there are some tools to insert errors into the LIN bus, however, these tools are expensive and needs software license, see for instance the J8115A LIN Tester of Agilent (Error-injection in LIN Tester) or the Vector's Test Hardware VH1150. These tools are very complete, but its control interface is complex. On the other hand, for the automation of test sequences it is necessary to have a deep knowledge of the software interface.

To develop a LIN system we do not require specialized tools or equipment, just hardware supporting LIN, while, for testing the LIN implementation it also does not require specialized tools, that's why, some of the tests can't be automated. This can lead to longer testing times and false positive or negative results due to human errors.

In general, no all microcontrollers of LIN drivers dispose of native LIN silicon that could help to find errors in LIN drivers and correct them during early stages of the development process. In this work we show a simple and low cost procedure to insert errors into the LIN protocol's frames with the goal of increasing the quality and reliability of the LIN components. With this procedure, we have performed some test-cases to validate the robustness of LIN drivers.

4 Marco teórico

El bus de Red Local de Interconexión (LIN) se desarrollo para crear un protocolo de comunicación de bajo costo para comunicar diversos dispositivos inteligentes. LIN es mayormente usado en la industria automotriz.
Aunque el protocolo de comunicación CAN tiene grandes ventajas sobre LIN como un manejo de errores avanzado, la implementación de CAN es prohibitiva en módulos de importancia media como los controladores del quemacocos, asientos y vidrios eléctricos en un automóvil.
LIN es un protocolo rentable para aplicaciones donde el manejo de errores y el ancho de banda de CAN no son una prioridad.
El protocolo LIN está basado en un protocolo de comunicación serial, lo que permite usar un transmisor/ receptor estándar serial universal asíncrono (UART), que esta embebido en la mayoría de los microcontroladores modernos [1].

Las principales propiedades del bus de LIN son las siguientes [2]:

- Cuenta con un maestro y hasta 16 esclavos.
- Implementación en silicio de bajo costo basado en UART/SCI.
- Auto-sincronización sin osciladores de cristal o resonadores cerámicos para los esclavos.
- Transmisión de señales determinísticas con tiempo de propagación de la señal computable por adelantado.
- Implementación de un solo cable de bajo costo.
- Velocidad hasta 20 kbits/seg.

A continuación se resumen algunas de las implementaciones de LIN en la red automotriz [3]:

- Sensor de lluvia.
- Sensor de Luz.
- Espejos eléctricos.
- Vidrios eléctricos.
- Señales para motores pequeños.
- Control de clima.
- Radio.

El nodo maestro controla la trasmisión de datos, determina la prioridad y el orden de los mensajes, sirve como referencia con su tiempo base de reloj, el nodo maestro siempre transmite el *header* del mensaje. El maestro puede transmitir dos tipos de mensajes un *header*, que es una petición para que algún nodo en particular conteste el mensaje o si quiere transmitir un mensaje con información a la red o a algún nodo en particular enviara un *header* pero además el mismo responderá con los datos y los demás nodos esclavos escucharan el mensaje.
Un nodo esclavo solo responderá si el *Protected Identifier* (PID) del *header*, es válido para él.

La señal de LIN tiene dos niveles de voltaje [2]:

- Nivel recesivo. Este nivel es equivalente al nivel de la batería del sistema (batería del automóvil), si el nivel de voltaje es mayor al 60% del valor de la fuente se toma como

recesivo (ver Fig. 1). Cuando no se transmite información, en este nivel de voltaje se encuentra el bus.

- Nivel dominante. Este nivel es 0 Volts, si el nivel de voltaje es menor al 40% del valor de la fuente se toma como dominante. Cuando se transmite un bit de información el *TRCV* cambia el estado del bus a estado dominante.

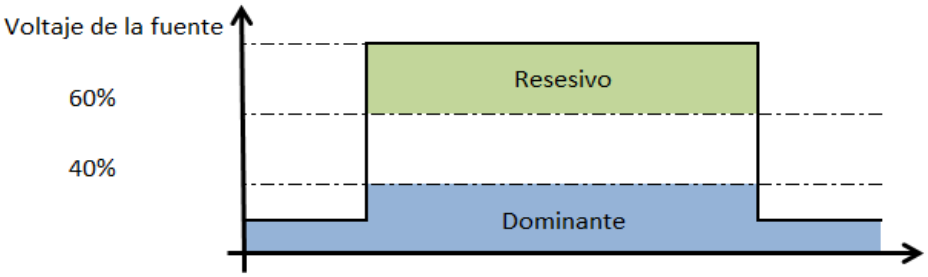

Figura 1 Niveles de voltaje

Un mensaje de LIN está estructurado de la forma que se ilustra en las Figs. 2 y 3:

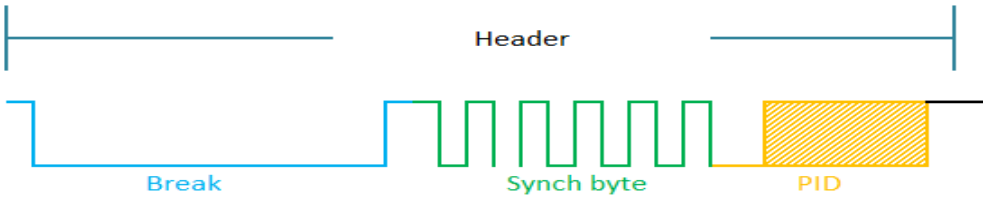

Figura 2 Header del mensaje

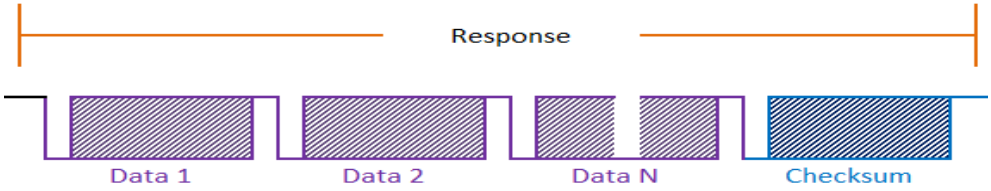

Figura 3 Respuesta del mensaje

El dato *Break* siempre es generado por el maestro y deberá ser al menos de 13 bits en estado dominante incluyendo el *start* bit.

El dato *Synch byte* contiene un valor 0x55, que especifica el patrón para determinar el tiempo base para los esclavos y siempre es generado por el maestro.

El dato PID consiste en dos sub-campos el identificador del bit del 0 al 5 y el identificador de paridad el bit 6 y 7. El ID puede usar valores del 0 al 63 y el identificador de paridad se calcula con las siguientes dos ecuaciones:

P0 = ID0 XOR ID1 XOR ID2 XORID4
P1 = -(ID1 XOR ID3 XOR ID4 XOR ID5)

Donde P0 es el bit 6 y P1 es el bit 7.

El *Data* 1, 2, …., 8, son los datos a ser transmitidos y pueden ser desde 0 hasta 8 datos, cada dato contiene 8bits.

Checksum: hay dos tipos de *checksum* el *Classic* y el *Enhance*, la especificación 1.3 de LIN soporta el *checksum Classic* y la especificación 2.0 soporta *Classic* y *Enhance* [2].

5 Desarrollo

La tarjeta seleccionada para el desarrollo del proyecto es la DEMO9S12XEP100 [4].
Esta tarjeta de desarrollo tiene un microcontrolador MC9S12XEP100, se eligió porque cuenta con el programador y el *debugger* integrado, con una licencia con versión para estudiantes. El microcontrolador cuenta con un modulo de *hardware* especializado en el protocolo CAN (MSCAN), este modulo cuenta con filtros, con un *buffer* de transmisión y otro de recepción además de un TRCV (*transceptor*) de CAN con conector, puede transmitir datos a una velocidad máxima de 1 Mbps para CAN. Los filtros permiten al microcontrolador validar los identificadores del mensaje para que este guarde solo los datos de los identificadores que se tienen configurados. Cuenta con otro modulo de *hardware* especializado en el protocolo LIN, dicho modulo tiene la capacidad de detectar un *break* de LIN, es decir tan solo configurando un registro el microcontrolador cambiara otro registro cada que el microcontrolador detecte un *break*, también cuenta con un TRCV de LIN con un canal y dos conectores de LIN conectados al mismo canal.

5.1 Requerimientos

Este proyecto tiene como alcance desarrollar un método capaz de corromper los mensajes de cualquier nodo conectado a la red LIN y a su vez comportarse como otro nodo más de la red. El objetivo de insertar errores en el protocolo LIN es para desarrollar controladores de LIN menos susceptibles a fallas, proveyendo de pruebas más exigentes y reales durante la etapa de desarrollo de los controladores, para detectar posibles fallas de diseño o funcionamiento en etapas tempranas del desarrollo. Se definieron los siguientes requisitos para el desarrollo del proyecto.

- El *driver* de LIN deberá soportar todos los *baudrate* en el rango de 1 bit/segundo hasta 20 kilo bits/segundo.
- El *driver* de LIN deberá tener una resolución de 1 bit/segundo.
- El *driver* de LIN deberá tener 2 canales, uno para leer el bus y otro para insertar errores.
- El *driver* de LIN deberá ser capaz de insertar errores en cualquier parte del mensaje siempre que el bit este en estado recesivo.
- El *driver* podrá corromper el *Header* del mensaje o la respuesta sin importar si esta última es de un maestro o de un esclavo.
- El *driver* deberá poder comportarse como esclavo.
- El canal 1 el *driver* deberá leer el bus de LIN y con este mismo deberá de transmitir datos cuando se encuentre configurado como esclavo.

- El canal 2 del *driver* solo deberá insertar errores en el bus.
- En modo esclavo el *driver* deberá transmitir desde 0 hasta 8 bytes de datos.
- En modo esclavo el *driver* podrá responder con un *checksum* de tipo *Enhance* o *Classic*.
- El *driver* de LIN contara con una estructura de configuración específica para el modo esclavo y otra para el modo de inserción de errores.

5.2 Desarrollo del driver de LIN

Se pensó en hacer un solo *driver* para el nodo esclavo y para la funcionalidad de inserción de errores en el bus de modo que se pudiera configurar en el mismo *driver* los parámetros correspondientes al nodo esclavo y a la herramienta de inserción de errores pero en estructuras diferentes.

- La estructura de configuración nodo esclavo se definió con el código que se presenta líneas abajo:

```
typedef struct {
   UINT8   u8SlaveID;                  /* ID válido */
   UINT8   u8DataBytes;                /*Cantidad de datos que se van a transmitir*/
   UINT8   u8ChecksumModel;            /*Tipo de checksum*/
   UINT8   *pu8Data;                   /*Datos*/
   UINT16 u8Checksum;                  /*Checksum*/
} tLIN_Slave_Message;
```

- La estructura de configuración para inserción de errores se definió con el código que se presenta líneas abajo:

```
typedef struct {
   UINT8   u8SlaveID;                  /* ID válido para corromper */
   UINT8   u8FallingEdgeToStart;       /*Flanco donde se inserta el error*/
   UINT32 u32ActiveTimeTicks;          /*Tiempo que está presente la falla*/
} tLIN_Tool_Error;
```

La estrategia que se siguió para hacer el *driver* fue, utilizar el canal 0 del SCI (*Serial Communication Interface*) para que sea el nodo esclavo y emplear un GPIO (*General purpose Input/Output*) para la herramienta de inserción de errores en el bus de LIN.

Para poder insertar errores en cualquier sección del mensaje pero en bytes específicos, el *driver* tiene que contar los bytes del mensaje, para esto se configuro el registro PPSP en el modo flanco de bajada, para que el *driver* cuente las interrupciones por flanco (PIFP) de un *pin* especifico del puerto P, se configuro como flanco de bajada para contar desde que se genera el *break* que sería el primer flanco de bajada en la línea del bus de LIN.

Para controlar la cantidad de datos que el *driver* tiene que corromper se utilizo un canal del TIM (*Timer Module*) del co-procesador XGATE, para contar el tiempo a partir de que el *driver*

inserta la falla, cuando el *timer* termina, inmediatamente se quita la falla en el bus. El tiempo que la falla permanece depende de dos parámetros:

- Del *baudrate* al que se encuentre la red de LIN.
- De la cantidad de datos que se quieran corromper.

El inverso del *baudrate* proporciona el tiempo de bit (Tbit), por ejemplo, si el bus de LIN se encuentra configurado a 19200 bits/seg y queremos corromper un byte, la falla tendría que estar presente en el bus por un tiempo total de 416.66 µsegundos, esto es:

$$\text{Tbit} = \frac{1}{19200} = 52.08\ \mu\ \text{seg}$$

$$\text{Tbyte} = 52.08\ \mu\ \text{seg} \times 8\ \text{bits} = 416.66\ \mu\ \text{seg}$$

Para la interrupción PIFP (Port P Interrupt Flag Register) se seleccionó el canal 2 y el *pin* PP2 al que se tiene acceso en el conector J101 a través del pin 30, dicho *pin* se debe conectar con el *pin* PS0, a dicho *pin* se tiene acceso en conector J302 *pin* 1 y a su vez conectarlo al *pin* 4 del conector J305 [4].

Como la tarjeta de desarrollo solo tiene un TRCV con un canal de LIN, es necesario utilizar otro TRCV para poder corromper el mensaje del bus de LIN. El *pin* PP3 al que se tiene acceso en el conector J101 *pin* 32 es el que se utilizo para mandar la señal de activación del TRCV 2 que va a corromper el mensaje de LIN.

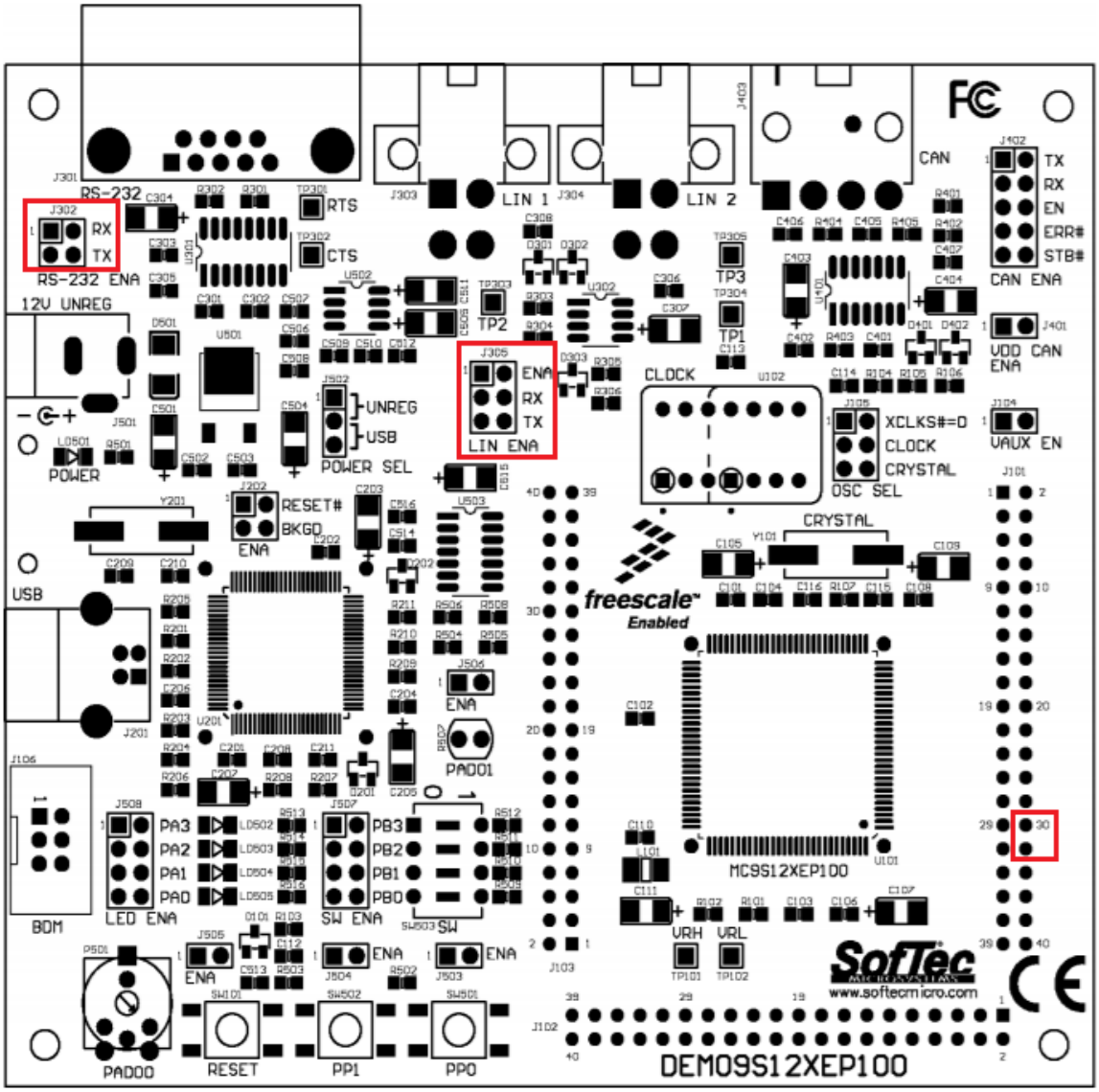

Figura 4 PCB DEMO9S12XEP100 [4]

Nota, si se desea usar el canal 1 del SCI del microcontrolador es necesario quitar los puentes entre los pines 3, 4 y 5, 6 del J305 ya que estos conectan el canal 2 del SCI con el TRCV de LIN.

Para el TRCV2 utilizamos el MC33661 de la empresa **NPX** que es el mismo que se utiliza en la tarjeta de desarrollo DEMO9S12XEP100, se utilizo la configuración del diagrama de la tarjeta de desarrollo (ver Fig. 5).

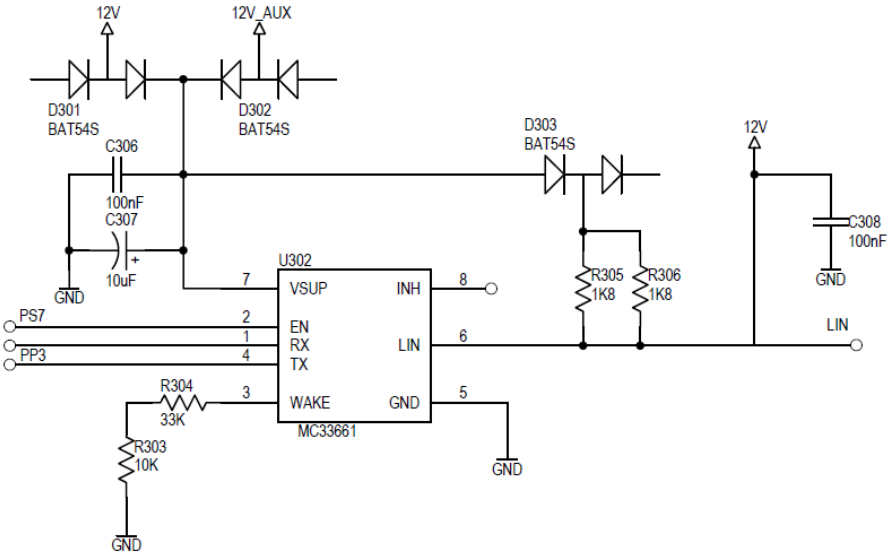

Figura 5 Diagrama esquemático TRCV2 [5]

La interrupción PIFP (*Port P Interrupt Flag Register*) la habilitamos en la inicialización del *driver*, para que siempre este contando los flancos desde el primer mensaje que salga al bus, una vez que el *driver* verifique que el flanco es el seleccionado por el usuario, el *driver* insertará el error e inmediatamente habilitara el TIM, cuando el TIM expire se genera una interrupción que remueve el error del bus.

Si es necesario introducir el error dentro del *Synch byte* (Synchronization Byte), o en el PID del mensaje, se puede hacer en el mismo código del driver. Para introducir estos tipos de error en el bus no es necesario que el *driver* conozca el PID, debido a que en ese momento el PID es desconocido, por lo tanto se introducirá el error en todos los mensajes. Por el contrario si se desea insertar un error en algún dato o en el *checksum* es posible insertarlo en un mensaje con un PID particular.

5.3 Pruebas

Para probar la funcionalidad del *driver* que se implementó se realizaron las siguientes pruebas verificando la respuesta de los nodos.

Equipo necesario:

- Una fuente de voltaje.
- Osciloscopio.
- Nodo maestro de LIN.
- Dos tarjetas de desarrollo DEMO9S12XEP100.
- Computadora.
- Cable de *debugeo* para la tarjeta DEMO9S12XEP100
- Cables para interconectar los TRCV, alimentar las tarjetas.

Entorno de prueba:
Por practicidad para las pruebas se utilizaron dos tarjetas DEMO9S12XEP100, una para programar el *driver* y usarla como nodo esclavo y la otra tarjeta se utiliza para insertar las fallas, de esta ultima tarjeta solo se utilizo el TRCV para generar las fallas en el bus.

Las tarjetas se interconectaron de la siguiente forma (ver Fig. 6):
Con un cable se conectó el *pin* 30 del conector J101 al *pin* 1 del J302 y a su vez al *pin* 4 del conector J305 de la tarjeta que hace el papel del nodo esclavo, con otro cable se conectó el *pin* 32 del conector J101 de la tarjeta que esta como nodo esclavo al *pin* 6 del J305 de la tarjeta que se va a usar para insertar errores. También se hizo la conexión del bus de LIN entre las tarjetas conectado LIN 2 (J304) del nodo esclavo con LIN 2 de la herramienta de inserción de errores y el conector LIN 1 (J303) del nodo esclavo se conectó al nodo maestro.

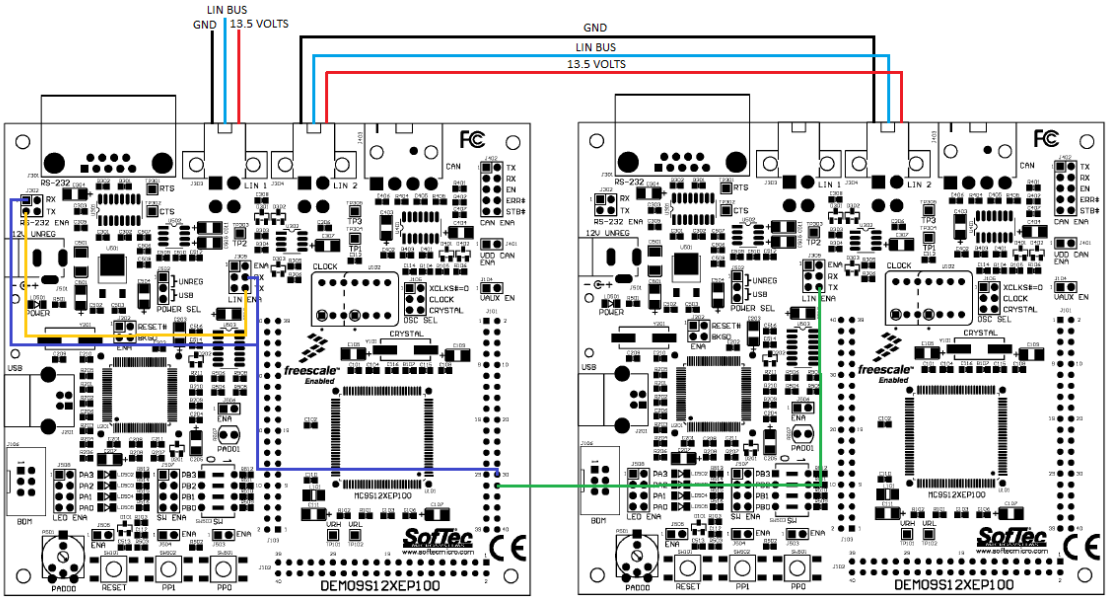

Figura 6 Conexión entre tarjetas [4]

Por medio del conector LIN 1 (J303) en el *pin* 3 se conecta a 13.5 volts de la fuente, los *pines* 1 y 2 son GND y el *pin* es el bus de LIN, estos *pines* están conectados en paralelo al conector LIN 2 (J304) por lo tanto al conectar LIN 2 del nodo esclavo se alimenta la tarjeta de la herramienta de inserción de errores.

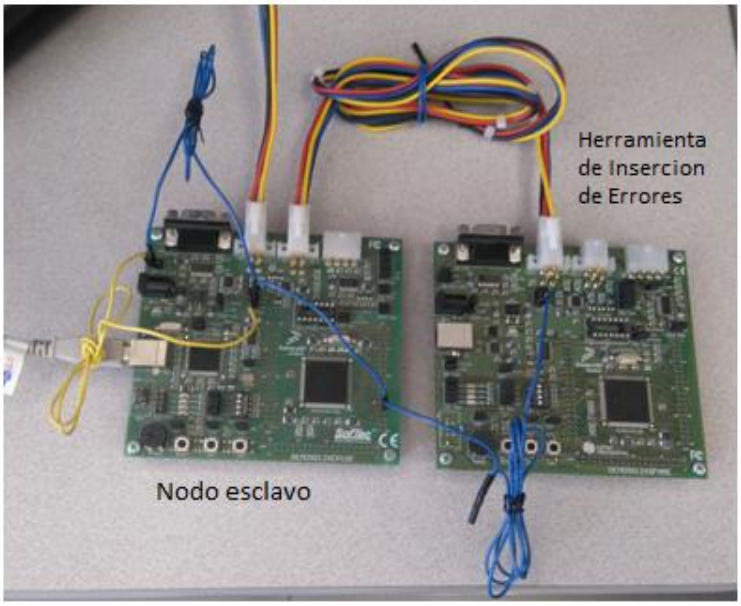

Figura 7 Conexión física de módulos para prueba del driver

Después conectamos una punta del Osciloscopio al bus de LIN y la otra punta la conectamos al *pin* 6 del J305 para monitorear la señal del error.
Una vez conectado el entorno de prueba configuramos la fuente de voltaje en 13.5 volts, cargamos el código y configuramos los parámetros para cada caso de prueba.

5.3.1 Caso de prueba 1

Descripción:
Verificar que el *driver* sea capaz de corromper el PID.

Precondiciones.
Fuente de voltaje 13.5 volts.
Baudrate = 10400 Kbits/seg.

Procedimiento:
Configuramos la variable *u8FallingEdgeToStart* = 7
Configuramos la variable *u32ActiveTimeTicks* = 5000 (480 µs) (referirse a Fig. 8)

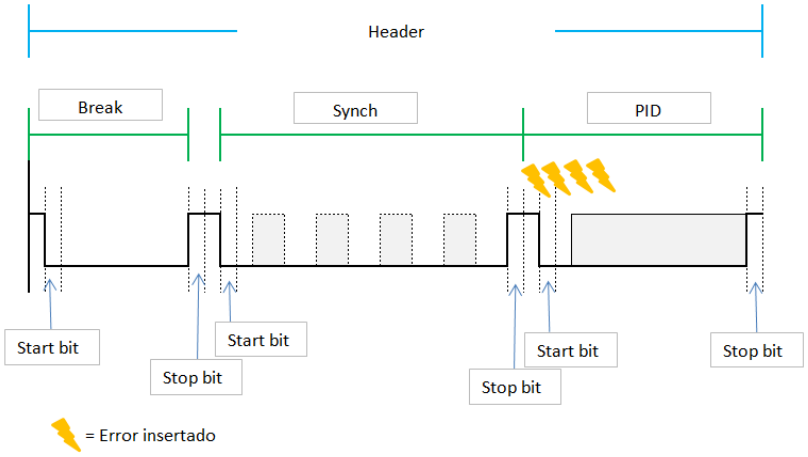

Figura 8 Caso de prueba 1

Nota:
Con los parámetros configurados como se indica en la Fig. 8, con la inserción de fallas en el *driver* se puede corromper 5 bits de información a partir del flanco 7.

Resultados:

A) En este escenario se desconectó el *pin* de inserción del bus para ver como se ve el mensaje del bus de LIN sin error (Fig. 9).

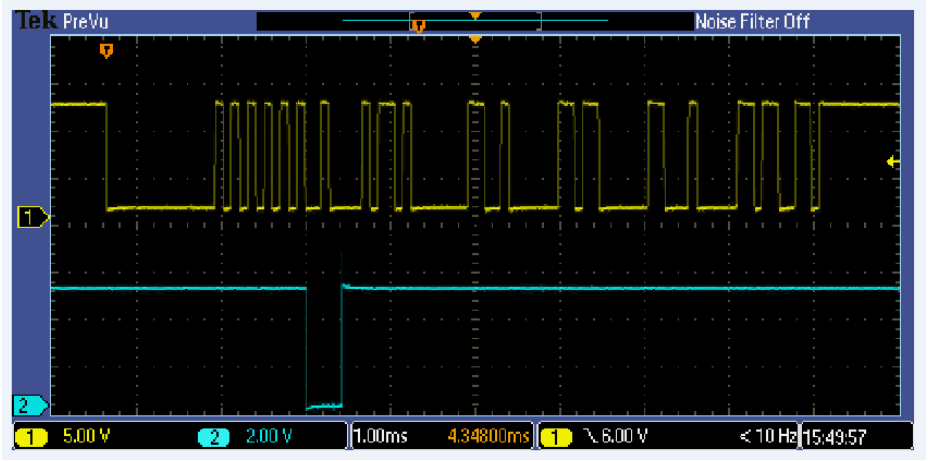

Figura 9 Resultados

B) En este escenario se muestra como corrompe el mensaje el error introducido al bus de LIN. El mensaje no sale completo al bus (ver Fig. 10), por que cuando el maestro detecta un error en la transmisión aborta el mensaje.

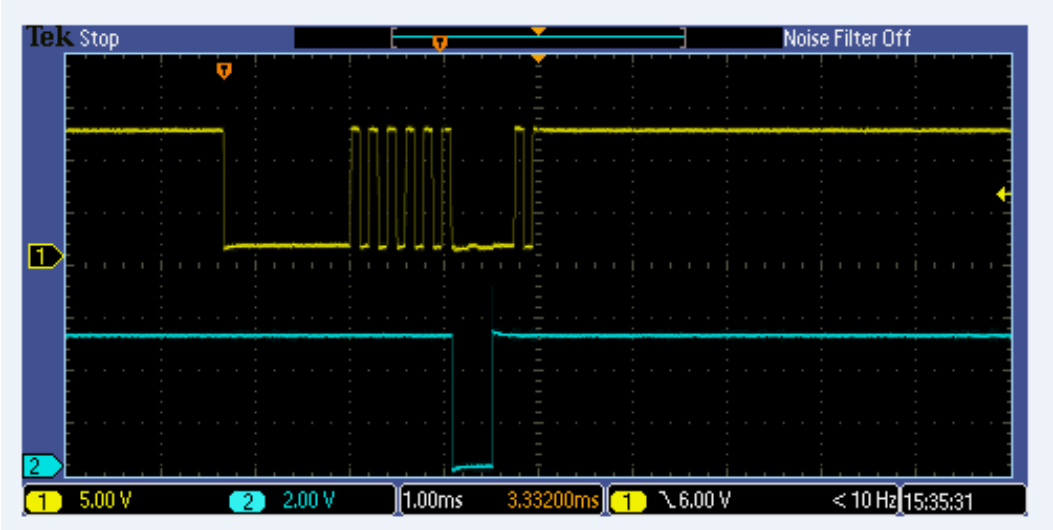

Figura 4 Resultados

5.3.2 Caso de Prueba 2

Descripción:
Verificar si el error se inserta en un bit en estado dominante el *driver* no corrompe el mensaje.

Precondiciones.
Fuente de voltaje 13.5 volts.
Baudrate = 10400 Kbits/seg.

Procedimiento:
Iteración 1. El error se inserta en el PID.

Configuramos la variable *u8FallingEdgeToStart* = 7
Configuramos la variable *u32ActiveTimeTicks* = 2000 (192 µs) (referirse a la Fig. 11)

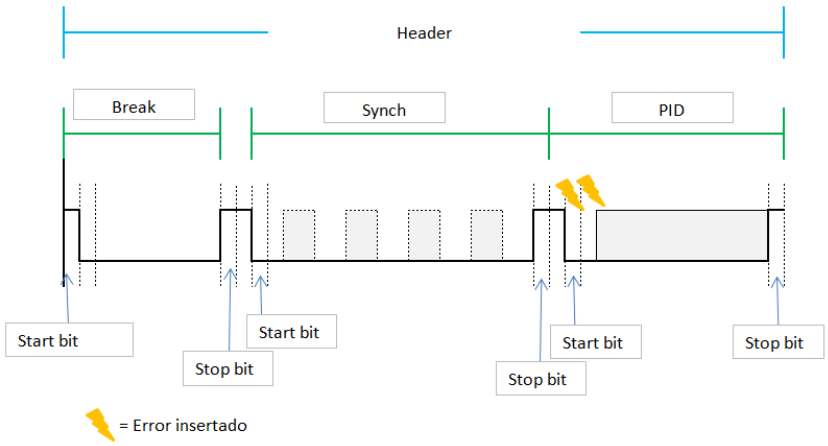

Figura 51 Caso de prueba 2 iteración 1

Procedimiento:
Iteración 2. El error se inserta en el Dato 1.

Configuramos la variable *u8FallingEdgeToStart* = 11
Configuramos la variable *u32ActiveTimeTicks* = 5000 (480 µs) (ver detalles en Fig. 12)

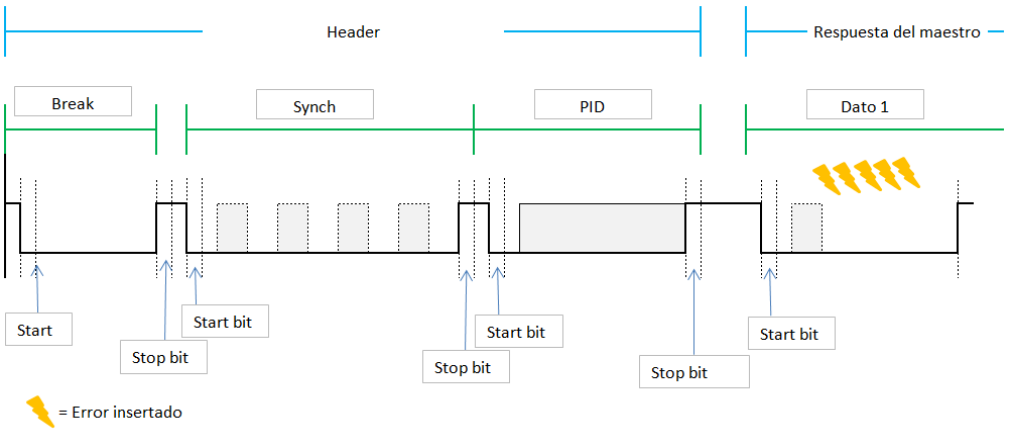

Figura 6 Caso de prueba 2 iteración 2

Procedimiento:

Iteración 3. El error se inserta en el Dato 1.

Configuramos la variable *u8FallingEdgeToStart* = 10

Configuramos la variable *u32ActiveTimeTicks* = 312 (30 µs) (referirse a Fig. 13)

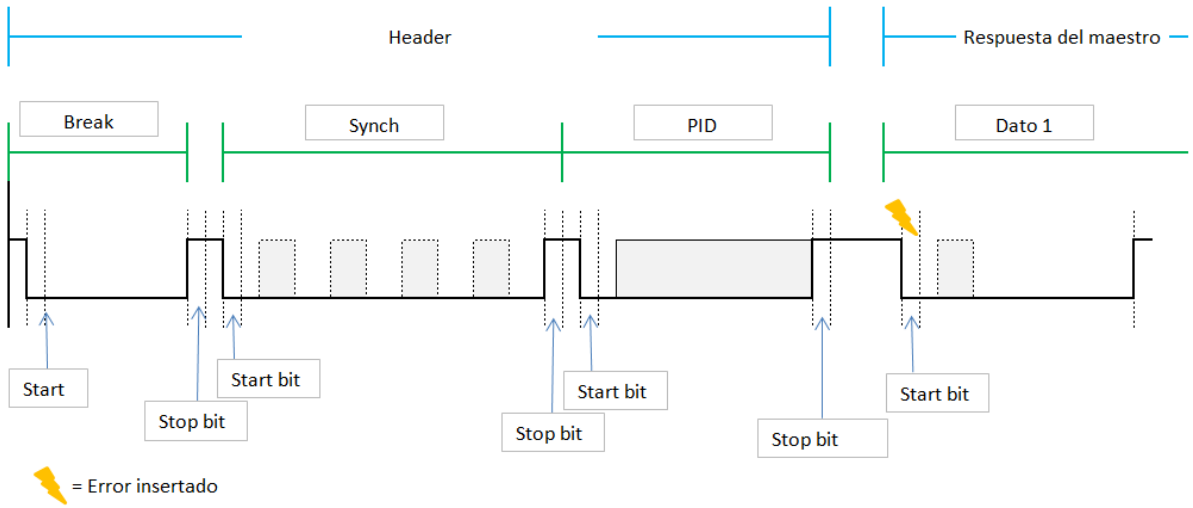

Figura 7 Caso de prueba 2 iteración 3

Resultados:

Iteración 1:

A) En este escenario el error se introdujo en los dos primeros bits del primer byte del PID, pero como podemos constatar en la Fig.14, esto no afecta el mensaje del maestro porque esos primeros bits están en estado dominante.

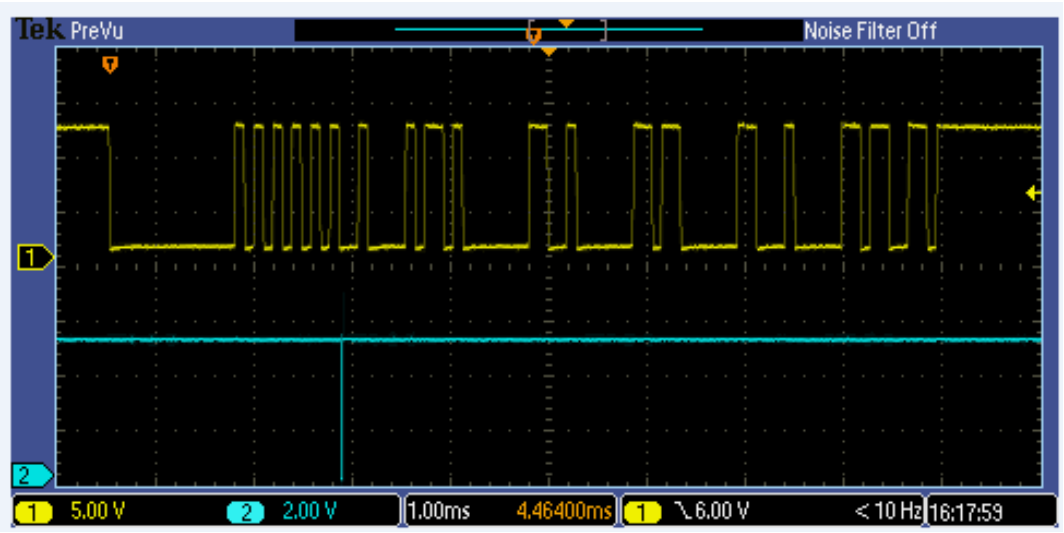

Figura 8 Resultados

Iteración 2:

A) En este escenario el error se introdujo en los cinco primeros bits del primer byte del dato 1, pero de acuerdo con la Fig. 15, esto tampoco afecta el mensaje del maestro porque esos primeros bits están en estado dominante.

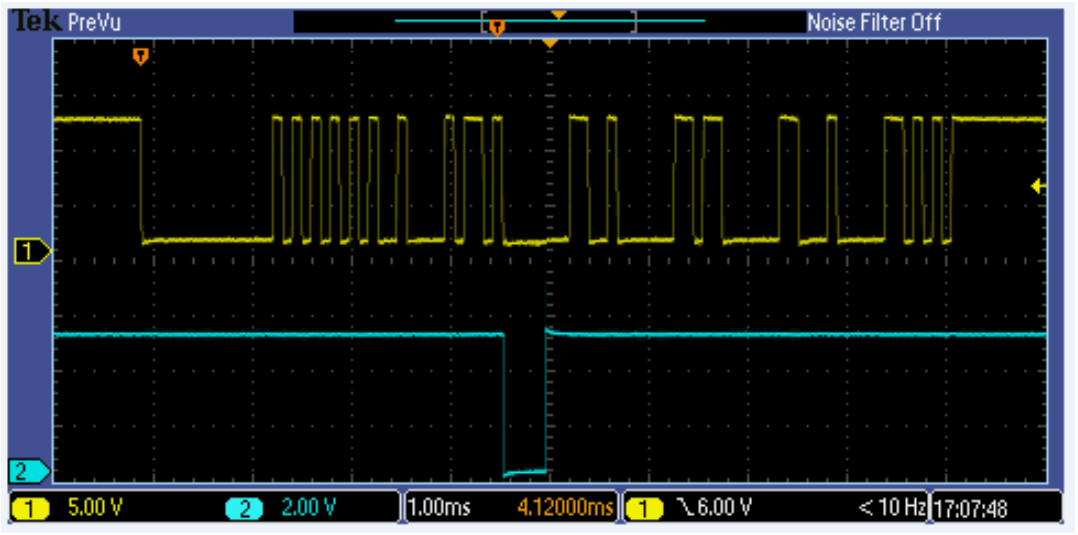

Figura 9 Resultados

Iteración 3:

A) En este escenario el error se introdujo en el primer bit del primer byte del PID, pero como observamos en la Fig. 16, tampoco afecta el mensaje del maestro porque esos primeros bits están en estado dominante y el error introducido es menor a un Tbit.

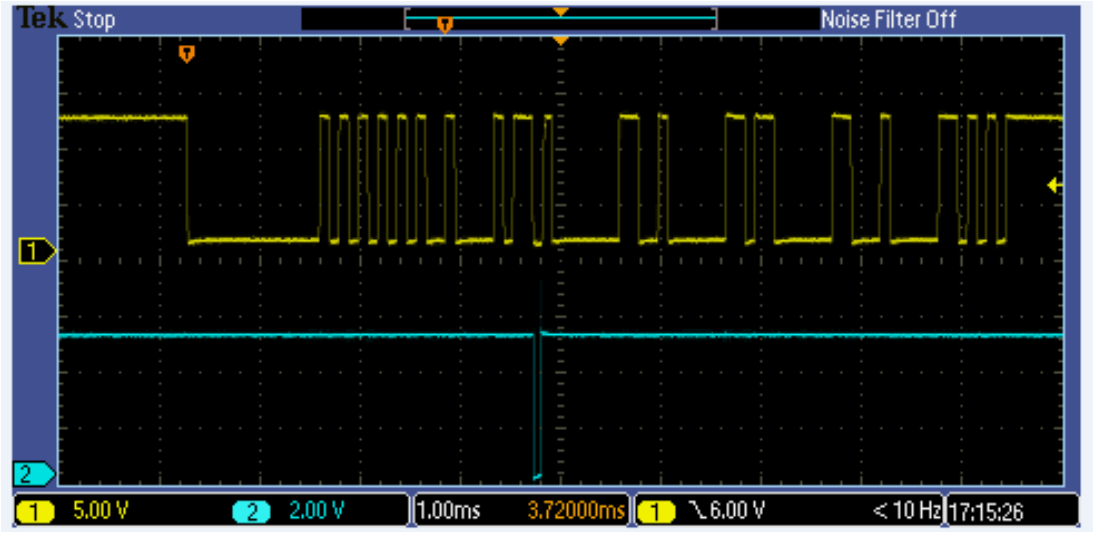

Figura 10 Resultados

5.3.3 Caso de Prueba 3

Descripción:
Verificar que el driver sea capaz de corromper un dato.

Precondiciones.
Fuente de voltaje 13.5 volts.
Baudrate = 10400 Kbits/seg.

Procedimiento:
Iteración 1. El error se inserta en Dato1.

Configuramos la variable *u8FallingEdgeToStart* = 10

Configuramos la variable *u32ActiveTimeTicks* = 5000 (480 µs) (referirse a Fig. 17)

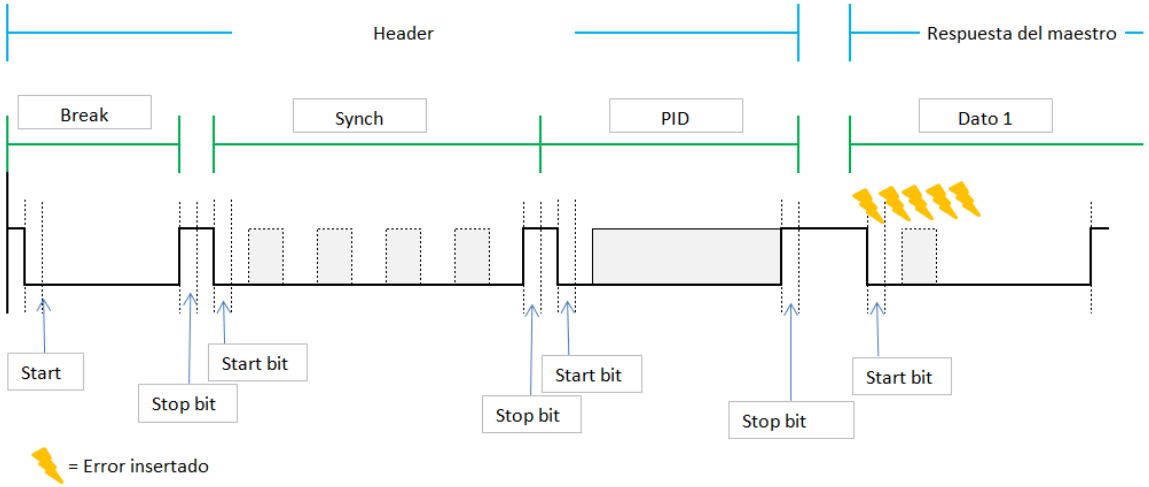

Figura 11 Casos de prueba 3 iteración 1

Resultados:

A) En este escenario se observa como era de esperarse, el mensaje si fue afectado por el error insertado en el bus (ver Fig. 18).

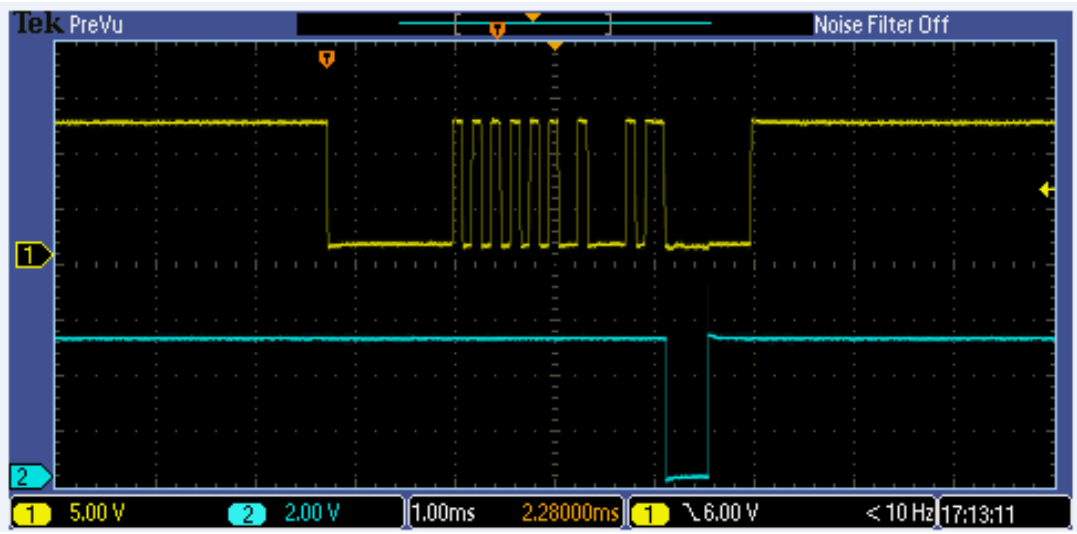

Figura 12 Resultados

Procedimiento:

Iteración 1. El error se inserta en Dato3.

Configuramos la variable *u8FallingEdgeToStart* = 14

Configuramos la variable u*32ActiveTimeTicks* = 5000 (480 µs) (ver detalles en Fig. 19).

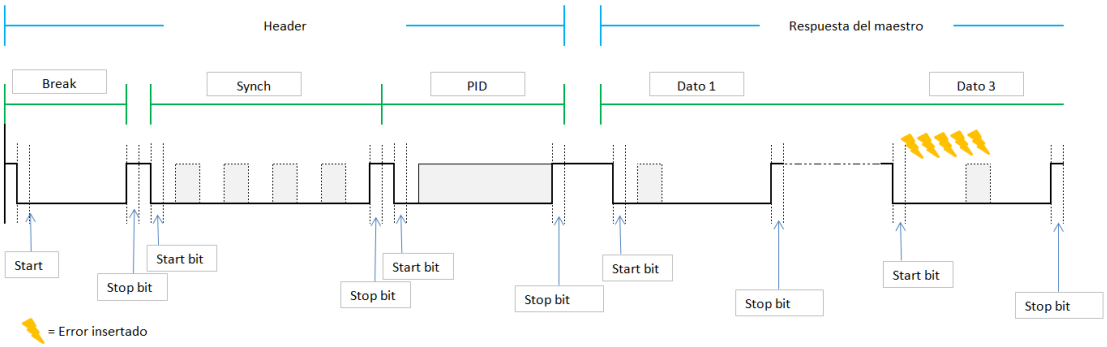

Figura 13 Caso de prueba 3 iteración 2

Resultados:

A) En este escenario se observa como el dato 3 del mensaje fue afectado por el error insertado en el bus (ver Fig. 20).

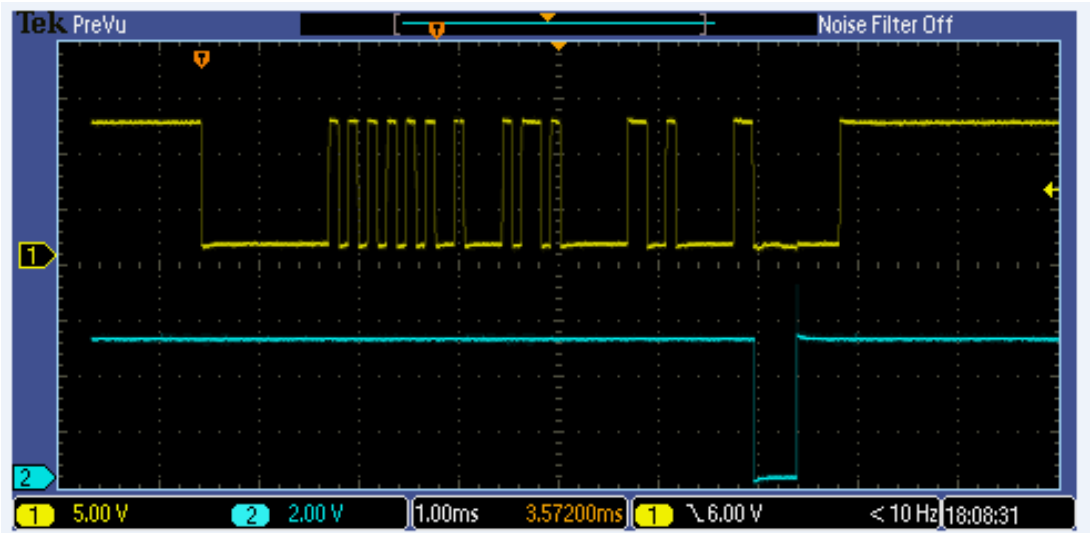

Figura 14 Resultados

5.3.4 Caso de prueba 4

Descripción:
Verificar que el driver sea capaz de corromper el stop bit del *Synch byte*.

Precondiciones.
Fuente de voltaje 13.5 volts.
Baudrate = 10400 Kbits/seg.

Procedimiento:
Configuramos la variable *u8FallingEdgeToStart* = 6
Configuramos la variable *u32ActiveTimeTicks* = 5000 (480 µs) (ver detalles de configuración en la Fig. 21).

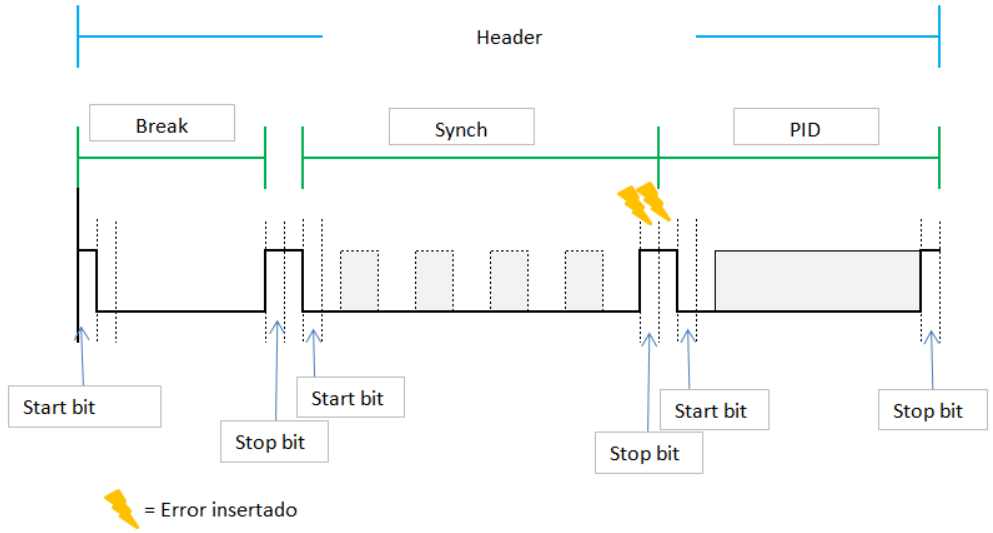

Figura 15 Caso de prueba 4

Resultados:

A) En este escenario el error está afectando stop bit de *Synch byte* (ver Fig. 22).

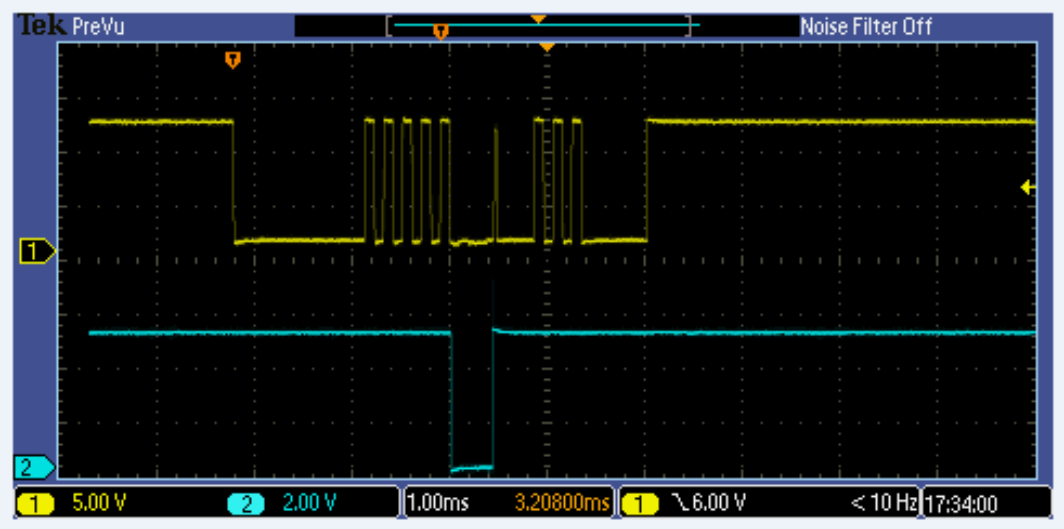

Figura 16 Resultados

5.3.5 Caso de prueba 5

Descripción:
Verificar que el driver sea capaz de corromper el Checksum.

Precondiciones.
Fuente de voltaje 13.5 volts.
Baudrate = 10400 Kbits/seg.

Procedimiento:
Configuramos la variable *u8FallingEdgeToStart* = 18
Configuramos la variable *u32ActiveTimeTicks* = 5000 (480 µs) (referirse a Fig. 23)

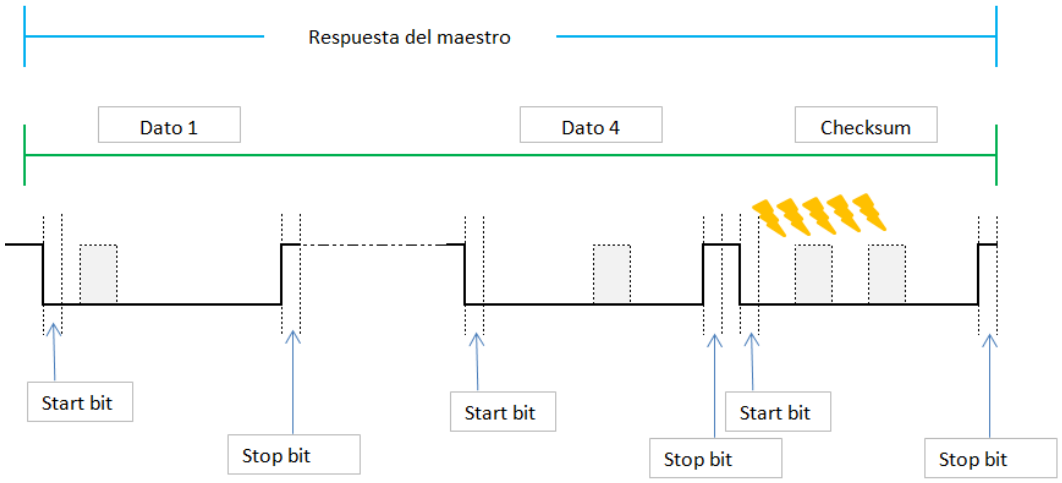

Figura 17 Caso de prueba 5

Resultados:

A) En este escenario el error está afectando el *Checksum* (ver Fig. 24).

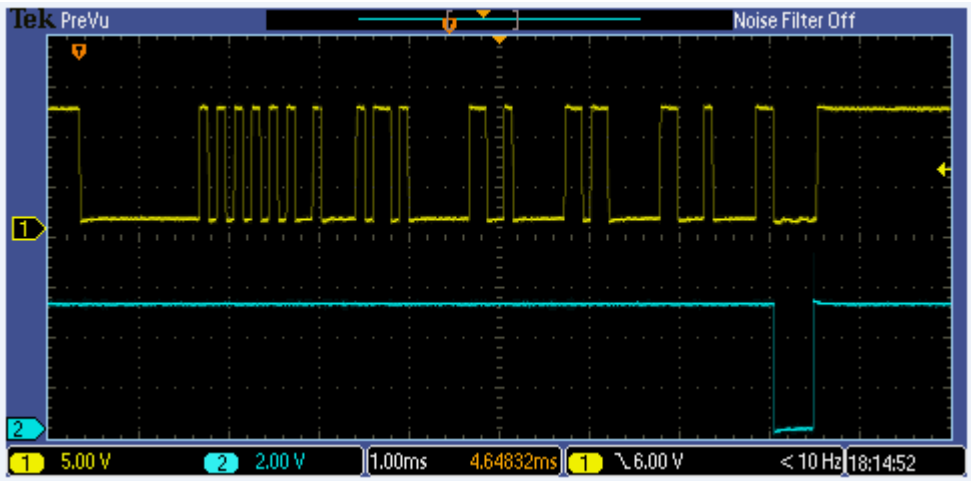

Figura 18 Resultados

6 Conclusiones

Los resultados de las pruebas hechas para verificar la funcionalidad del *driver* demuestran que la estrategia utilizada para la inserción de errores en el proyecto funcionó como se esperaba, pues permite al usuario insertar los errores precisos en cualquier parte del mensaje, con una duración correcta. La precisión en este método es muy importante porque es necesario afectar solamente a una parte específica del mensaje sin alterar otras partes, si no se cumple con esto último se pueden inducir otro tipo de errores y puede enmascarar fallas en los nodos que se encuentren bajo prueba. Otra ventaja de que tiene esta estrategia es que corrompe el mensaje de acuerdo al número de flancos, esto permite al usuario insertar el error en el bus sin importar si fue de un maestro o un esclavo.

La única desventaja que se pudo notar en el método es que si se desea insertar un error antes del PID o en el PID, el driver corromperá todos los mensajes, este comportamiento es esperado pues, la estructura del mensaje de LIN no permite identificar un mensaje antes del PID, por lo anterior si se desea insertar un error en el PID o en el *Synch byte* en una red donde hay muchos nodos conectados, se corromperán todos los mensajes que salgan al bus.

Como trabajo a futuro para este proyecto falto desarrollar la GUI y el driver de CAN. El proyecto hasta ahora es funcional, pero para poder usar la herramienta cuenta con la desventaja de tener que modificar el código, (modificar los parámetros en la estructura correspondiente) y cambiar los parámetros para poder insertar los errores o que el driver responda información específica como esclavo. La GUI es indispensable para que la herramienta sea amigable con el usuario y por supuesto el driver de CAN es necesario para cumplir este cometido.

Referencias

[1] N. Instruments. (12 May 2010). Introducción al Bus de Red Local de Interconexión (LIN). National Instruments. [En línea]. Available: http://www.ni.com/white-paper/9733/es/.

[2] Consortium LIN. (2003). LIN Specification Package 2.0.

[3] A. Corporation. (2005). AVR322: LIN v1.3 Protocol Implementation on Atmel AVR Microcontrollers. [En línea]. Available: http://www.atmel.com/images/doc7548.pdf.

[4] Semiconductors Freescale. (2006). Element14 Communty. [En línea]. Available: https://www.element14.com/community/docs/DOC-37746/l/freescale-user-manual-for-demo9s12xep100.

[5] NXP. (24 07 2006). DEMO9S12XEP100 Schematic and Bill of Material. [En línea]. Available: http://www.nxp.com/assets/downloads/data/en/schematics/DEMO9S12XEP100_SCH.pdf.