*Universitat de les Illes Balears*
*Máster en Ingeniería Electrónica*

TRABAJO FINAL DE MÁSTER

# Mejora, integración y verificación experimental de los mecanismos de resolución de inconsistencias del proyecto CANbids

Alumno:
Christian Peter Winter
Directores:
Julián Proenza Arenas
Guillermo Rodríguez-Navas González

30 de Abril de 2012

Se hace constar que esta memoria se ha realizado, bajo la dirección de Julián Proenza Arenas y Guillermo Rodríguez-Navas González, por Christian Peter Winter y que constituye su trabajo final de Máster.

Palma de Mallorca, Abril de 2012

Firmado: Christian Peter Winter
Estudiante de Máster Universitario

Firmado: Julián Proenza Arenas
Codirector del proyecto
Profesor Titular de la Universidad
Departamento de Ciencias Matemáticas e Informática
Universitat de les Illes Balears

Firmado: Guillermo Rodríguez-Navas González
Codirector del proyecto
Profesor Colaborador de la Universidad
Departamento de Ciencias Matemáticas e Informática
Universitat de les Illes Balears

# Resumen

El *Controller Area Network* (CAN) es un protocolo de comunicaciones que inicialmente fue diseñado para aplicaciones de comunicación internas en vehículos, pero que ha sido ampliamente adoptado en muchas otras áreas dentro del campo de los sistemas de control empotrados distribuidos. CAN es hoy en día una tecnología madura, cuyo enorme éxito se debe principalmente a su capacidad para control de errores, su baja latencia, su acceso priorizado a la red y su respuesta en tiempo real. Además, el uso tan extendido de CAN ha causado un descenso en el precio de esta tecnología hasta niveles en los cuales otras tecnologías no pueden competir. Sin embargo, para su uso en sistemas con elevada garantía de funcionamiento, CAN no resulta adecuado ya que presenta diferentes limitaciones relacionadas con la garantía de funcionamiento.

No obstante, varios investigadores, incluyendo miembros del grupo de investigación SRV del Departamento de Ciencias Matemáticas e Informática de la UIB, han propuesto mecanismos encaminados a superar las limitaciones de CAN para demostrar que sí puede ser utilizado en aplicaciones críticas de control, siempre y cuando incluya las mejoras pertinentes. Tomando toda esa investigación previa como punto de partida, el grupo SRV inició el proyecto denominado CANbids (*CAN-Based Infrastructure for Dependable Systems*), cuyo objetivo es el desarrollo de una infraestructura de comunicaciones, basada en CAN, que incluya las distintas soluciones propuestas, tanto por el propio grupo SRV como por otros grupos, como elementos constitutivos de su arquitectura.

Cuando se comenzó el presente trabajo, para la mayoría de las mejoras propuestas ya se había completado la fase de diseño. Los mecanismos dirigidos a la resolución de inconsistencias, sin embargo, aún se encontraban en una fase de inicio.

Al ser la consistencia de datos una de las propiedades esenciales de un sistema con elevada garantía de funcionamiento, se definió como objetivo principal del trabajo mejorar, integrar y verificar experimentalmente los mecanismos de resolución de inconsistencias del proyecto CANbids. Para ello, se diseñaron e implementaron dos módulos, AEFT y CANsistant. Su diseño se planteó, desde un principio, para ser totalmente compatibles con el protocolo CAN y para su uso tanto en topologías bus como estrella. La implementación de ambos mecanismos se realizó en código VHDL. Mientras CANsistant fue verificado mediante herramientas de simulación, para el AEFT se tuvo que implementar un prototipo específico basado en FPGAs. La verificación experimental del prototipo se realizó en el laboratorio, haciendo uso de un osciloscopio digital.

A modo de sumario, entre las tareas realizadas se encuentran el estudio de las características principales de los elementos constitutivos de la arquitectura, el estudio de la relevancia de los nuevos escenarios de inconsistencia identificados, la propuesta del AEFT para resolver estos nuevos escenarios, la mejora y el rediseño de CANsistant para su uso tanto en topologías bus como en topologías estrella, la propuesta de las modificaciones necesarias para poder integrar eficientemente los nuevos elementos con los ya existentes y, finalmente, la integración efectiva de los mismos.

# Índice general

# Índice de figuras

# Índice de cuadros

# Capítulo 1

# Introducción

## 1.1. Definición del problema y objetivos

El *Controller Area Network* (CAN) [ISO93] es un protocolo de comunicaciones desarrollado por *Bosch*. Las principales ventajas de este bus de campo son su bajo coste, su elevada robustez electromagnética y su buena respuesta en tiempo real. Inicialmente, CAN fue ideado para su uso en aplicaciones automotrices pero pronto su uso se extendió a muchas otras aplicaciones de control.

A pesar de estas importantes ventajas, existe la creencia de que CAN no es adecuado para aplicaciones críticas [PIME04], [KOPE99], principalmente a causa de las siguientes limitaciones relacionadas con la garantía de funcionamiento (*dependability*) [PIME08]: (1) Consistencia de datos limitada; (2) Contención de errores limitada; (3) Soporte limitado para tolerancia a fallos y (4) Ausencia de sincronización de reloj. Sin embargo, numerosos investigadores opinan que CAN sí es adecuado para soportar aplicaciones críticas de control una vez que se hayan superado dichas limitaciones mediante mejoras adecuadas. Esta posibilidad resulta especialmente interesante en muchos campos de aplicación, ya que los componentes CAN son mucho más baratos que los componentes de los competidores naturales de CAN en aplicaciones con elevada garantía de funcionamiento, como por ejemplo FlexRay o TTA. Una aplicación adecuada y específica para estos sistemas CAN mejorados serían las aplicaciones de carácter crítico propias de los sistemas *X-by-Wire* para vehículos, ya que el uso de CAN permite aprovechar la experiencia y *know-how* que los equipos

de ingeniería de los fabricantes de coches han ganado durante las dos décadas que llevan empleando y programando esta tecnología.

En este trabajo se tratará el problema relacionado con la consistencia de datos del protocolo CAN. Éste viene ligado a ciertas vulnerabilidades del mecanismo de control de errores que implementa CAN y tiene tres fuentes básicas: (1) la presencia del estado de error pasivo [RUFI98]; (2) la regla del último bit del EOF [RUFI98] [PROE00] y (3) la señalización inconsistente de errores [RODR11].

El proyecto CANbids, cuyo objetivo es diseñar, implementar y validar una infraestructura basada en CAN para soportar la ejecución de aplicaciones de control distribuidas con elevada garantía de funcionamiento, propone dos estrategias de resolución de inconsistencias: CANsistant y AEF. Estas propuestas están diseñadas para atajar los problemas de inconsistencia causados por las fuentes (2) y (3), respectivamente. Los objetivos de este trabajo son el diseño, la implementación y la verificación experimental de los mecanismos mencionados, tanto para la topología propia de CAN, el bus de campo, como para la topología en estrella de ReCANcentrate [BARR09]. Adicionalmente se realizará un estudio de relevancia de la tercera fuente de inconsistencias: la señalización inconsistente de errores.

El objetivo final es proporcionar un conjunto de mejoras, totalmente compatibles con el protocolo CAN y el resto de mecanismos del proyecto CANbids, que solucionen de manera eficiente y sin afectar el rendimiento de la red, las limitaciones referentes a la consistencia de datos de CAN. Se propondrán y realizarán, además, las modificaciones pertinentes para integrar eficazmente estos nuevos elementos con el resto de mecanismos de la arquitectura.

## 1.2. Aportaciones

### 1.2.1. Estudio de relevancia

La primera contribución importante de este trabajo es la realización de un estudio que determina la relevancia de las señalizaciones inconsistentes de errores en relación al resto de fuentes de inconsistencias del protocolo CAN.

Mientras que la relevancia del estado de error pasivo y la regla del último bit del

EOF como fuentes primarias de inconsistencias ha sido demostrada por diferentes autores (incluyendo la propuesta de soluciones), para la señalización inconsistente de errores, fuente identificada recientemente , no existe estudio alguno. Para ver qué impacto tiene esta tercera fuente sobre el correcto funcionamiento de un sistema distribuido, resulta importante determinar si la probabilidad de producirse los escenarios de inconsistencia debidos a este problema es lo suficientemente elevada como para tener que resolverlos, o si, por el contrario, se trata de escenarios "patológicos" cuya frecuencia de aparición es tan reducida que puedan ser despreciados sin perjudicar la fiabilidad del sistema.

Como bien se podrá ver en el capítulo 4, el estudio realizado demuestra que la señalización inconsistente de errores efectivamente es una causa de inconsistencias a tener en cuenta y que, por lo tanto, será necesario incluir en el sistema un mecanismo que resuelva dicho tipo de inconsistencias: el módulo AEFT.

### 1.2.2. AEFT

Al haberse demostrado que los escenarios de inconsistencia debidos a la señalización inconsistente de errores son relevantes desde el punto de vista de la garantía de funcionamiento, resulta necesario proponer una solución a dicho problema: una estrategia basada en la transmisión de los denominados AEFs (*Aggregated Error Flags*) [RODR11]. La segunda aportación del trabajo fue, consecuentemente, diseñar e implementar un mecanismo que implemente dicha estrategia: el AEFT (*Aggregated Error Flag Transmitter*).

Como se verá de forma detallada en los capítulos 3 y 5, la fuente de los nuevos escenarios de inconsistencia se encuentra en los señalizadores de error definidos por el protocolo CAN [ISO93] ya que, en ocasiones, pueden resultar demasiado cortos. La estrategia propuesta se basa en alargar los señalizadores de error para garantizar, en todo momento, la globalización de errores locales, aún en el caso de haber múltiples bits de la trama afectados por errores. Concretamente, se agregan varios *error flags* consecutivos, separados el uno del otro por bits de valor recesivo. La solución es completamente compatible con el protocolo CAN y con el resto de mecanismos de los que se compone CANbids.

El AEFT es capaz de detectar la posible ocurrencia de inconsistencias e inyectar

un AEF. En base a un mismo diseño inicial se proponen dos implementaciones: una para topologías bus y otra para topologías estrella. Ambas versiones serán implementadas físicamente mediante prototipos basados en FPGAs y verificadas experimentalmente en el laboratorio para demostrar su validez.

### 1.2.3. CANsistant

La tercera y última aportación consiste en el desarrollo de CANsistant, un mecanismo capaz de detectar inconsistencias debidas a la regla del último bit del EOF incluso en presencia de múltiples errores de canal.

Inicialmente, CANsistant fue planteado para su uso en topologías bus [PROE09]. En este trabajo, sin embargo, se planteó también la integración de CANsistant con ReCANcentrate, esto es, su uso en una topología estrella. En un bus, CANsistant es acoplado a la red como si de un nodo genérico se tratara. En una estrella, el dispositivo estará integrado en el concentrador (*hub*).

En total, en esta memoria se proponen tres versiones diferentes de CANsistant: la versión original para topologías bus, propuesta en el apartado 3.4.3, una segunda versión para topologías bus, rediseñada para reducir el número de falsas alarmas y, finalmente, una versión para topologías estrella basada en la integración de CANsistant con ReCANcentrate (capítulo 6).

## 1.3. Organización del documento

El resto del documento está organizado de la siguiente forma: en primer lugar, en el capítulo 2 se describen los conceptos básicos de la garantía de funcionamiento, poniendo énfasis en los fundamentos de la tolerancia a fallos y las estrategias de diseño de sistemas tolerantes a fallos.

El capítulo 3 presenta el proyecto CANbids. En él se exponen las propiedades del protocolo CAN, sus limitaciones y las soluciones que ofrece CANbids para superarlas. Adicionalmente se describen las tareas pendientes del proyecto, punto de partida del presente trabajo.

En los capítulos 4, 5 y 6 se describen de forma detallada las tareas desarrolladas en el trabajo. Así, en el capítulo 4 se especifica el estudio de relevancia llevado a cabo para definir la importancia de la señalización inconsistente de errores en el contexto de la consistencia de datos de CAN. En el capítulo 5 se describen el diseño, la implementación y la verificación experimental del módulo AEFT tanto para topologías bus como para topologías estrella. En el capítulo 6 se definen los pasos seguidos para la mejora, el rediseño y la implementación física de CANsistant tanto para topologías bus como topologías estrella (integrando CANsistant con la estrella ReCANcentrate).

Finalmente, en el capítulo 7 se resume el trabajo presentado en este documento, se extraen las conclusiones finales, se especifican las publicaciones que resultaron del trabajo y se proponen futuras investigaciones.

# Capítulo 2

# Sistemas distribuidos con garantía de funcionamiento

## 2.1. Introducción

Como se ha indicado en el capítulo 1, el presente trabajo tiene como objetivo garantizar la consistencia de datos en CAN y, así, aumentar la fiabilidad de esta red de comunicaciones. La fiabilidad es un atributo ligado a la denominada garantía de funcionamiento (*dependability*). La garantía de funcionamiento es una característica indispensable de los sistemas de control distribuidos. Por este motivo, en este capítulo se detallarán los conceptos relacionados con este tipo de sistemas.

La terminología adoptada en los siguientes apartados es la más ampliamente aceptada en el ámbito de la garantía de funcionamiento y se basa en la propuesta realizada por Laprie en [Lap92].

## 2.2. Conceptos básicos y terminología

En [Lap92] encontramos la siguiente definición de garantía de funcionamiento:

*"Dependability is defined as the trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers"* [Car82].

"La garantía de funcionamiento de un sistema informático es la propiedad que permite a sus usuarios depositar una confianza justificada en el servicio que les proporciona".

Los tres conceptos relacionados con la garantía de funcionamiento son: los atributos, los impedimentos y los medios. Dichos conceptos están reunidos en el denominado árbol de la garantía de funcionamiento (*dependability tree*) mostrado en la Figura 2.1.

Los **atributos** (*attributes*) son diferentes propiedades que un sistema con garantía de funcionamiento debe cumplir (dependiendo de la aplicación unos atributos serán más importantes que otros). Los principales son:

- Fiabilidad (*reliability*): Un sistema es fiable si presenta una alta probabilidad de proporcionar un servicio de manera correcta e ininterrumpida.
- Disponibilidad (*availability*): Un sistema disponible muestra una probabilidad elevada de estar listo para proporcionar servicio (incluso en el caso de haberse producido fallos).
- Seguridad (*safety*): Propiedad de un sistema que es capaz de entrar en un modo seguro en caso de producirse una avería que podría conllevar consecuencias catastróficas (muerte de personal y/o altas pérdidas económicas).
- Confidencialidad (*security*): Un sistema es confidencial si impide el acceso no autorizado.
- Adicionalmente, puede interesar que un sistema presente un nivel de rendimiento estable (*performability*), sea fácilmente reparable (*maintanability*) y su testeo sea sencillo (*testability*).

Los atributos anteriores son discutidos exhaustivamente en [Joh89] y [Sho02].

Los **impedimentos** (*impairments*) de un sistema con garantía de funcionamiento son circunstancias indeseadas que causan la pérdida de atributos o resultan de la no existencia de éstos [Lap92]. Existen tres tipos de impedimentos:

- Averías (*failures*): desviaciones del correcto funcionamiento de un sistema (el servicio proporcionado no es el deseado), se detallan en el apartado 2.2.1.

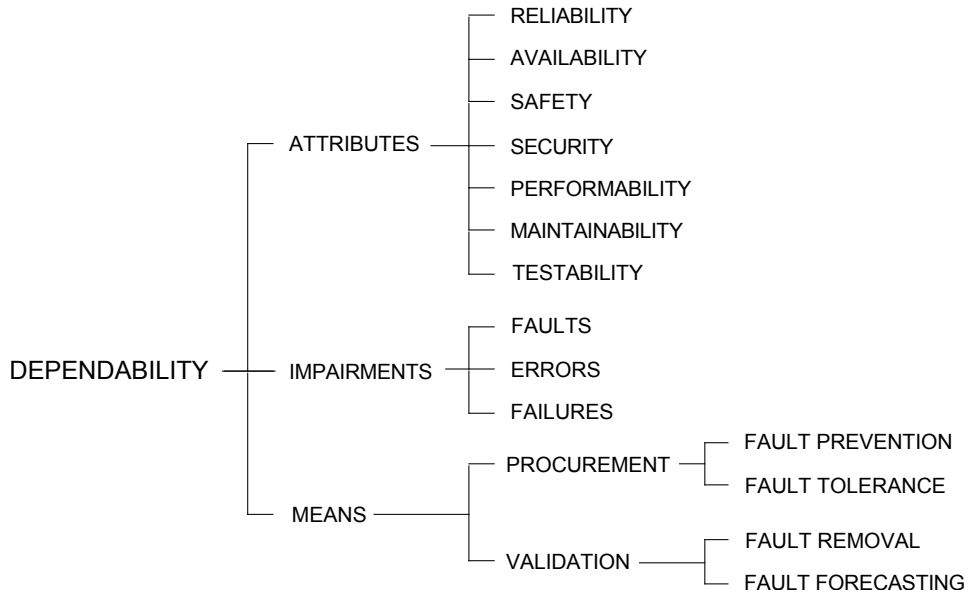



Figura 2.1: El árbol de la garantía de funcionamiento

- Errores (*errors*): valores incorrectos del estado interno del sistema que pueden llegar a provocar una avería.
- Fallos (*faults*): defectos en el diseño o durante la operación de un sistema que puede causar un error. Existen diferentes fallos según su naturaleza, origen y persistencia (en la tabla 2.2 se muestra una clasificación de fallos típicos).

Es importante denotar que existe una propagación de causas en la jerarquía, es decir, una avería de un subsistema puede causar un fallo en el sistema y así sucesivamente. En CAN, por ejemplo, un fallo intermitente en un transceiver puede generar un error en la trama transmitida y así causar una inconsistencia. Finalmente, la inconsistencia producida puede llevar a una avería del sistema completo.

Los **medios** (*means*) de un sistema con garantía de funcionamiento son métodos y técnicas usadas para asegurar que éste proporcione servicio en el que se pueda depositar confianza [Lap92]. Resulta recomendable combinar estos medios para obtener buenos resultados. Existen cuatro clases:

- Prevención de fallos (*fault prevention*): métodos destinados a prevenir la ocurrencia o introducción de fallos (se utilizan siempre, a veces con mayor intensidad).

| FALLOS TÍPICOS | NATURALEZA | | ORÍGEN | | | | | | PERSISTENCIA | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Causa fenomenológica | | Interno/Externo | | Fase | | | |
| | Fallos accidentales | Fallos intencionados | Fallos físicos | Fallos humanos | Fallos internos | Fallos externos | Fallos de diseño | Fallos operacionales | Fallos permanentes | Fallos temporales |
| Fallos físicos | X | | X | | X | | | X | X | |
| | X | | X | | | X | | X | X | |
| Fallos transitorios | X | | X | | | X | | X | | X |
| Fallos intermitentes | X | | X | | X | | | X | | X |
| | X | | | X | X | | X | | | X |
| Fallos de diseño | X | | | X | X | | X | | X | |
| Fallos por interacción | X | | | X | | X | | X | | X |
| Fallos Maliciosos | | X | | X | X | | X | | X | |
| | | X | | X | X | | X | | | X |
| Intrusiones | | X | | X | | X | | X | X | |
| | | X | | X | | X | | X | | X |

Figura 2.2: Clasificación de fallos

- Tolerancia a fallos (*fault tolerance*): técnicas para diseñar un sistema con capacidad interna para garantizar la ejecución correcta y continuada de las tareas a pesar de la ocurrencia de fallos hardware y/o software.

- Eliminación de fallos (*fault removal*): métodos usados para reducir la cantidad y la gravedad de fallos (a posteriori de haberse producido).

- Predicción de fallos (*fault forecasting*): métodos para estimar futuros incidentes y los fallos consecuentes.

### 2.2.1. Modos de avería

Como se ha mencionado en el apartado anterior, una avería es una desviación indeseada del correcto funcionamiento de un sistema. Se debe tener en cuenta que una avería puede presentarse tanto en el sistema como en cualquiera de sus componentes (subsistemas). Las distintas formas en las que un sistema genérico puede averiarse se denominan modos de avería (failure modes).

Clásicamente, los modos de avería se han caracterizado conforme tres puntos de vista [Lap92]: Según sus **consecuencias en el entorno**, es posible diferenciar dos tipos de averías: las averías benignas (*benign failures*) y las averías catastróficas (*catastrophic failures*). Las consecuencias de una avería benigna tienen un coste del mismo orden de magnitud que los beneficios que se obtendrían si el sistema funcionará correctamente. En cambio, una avería catastrófica conlleva consecuencias con un coste incomparablemente mayor a los beneficios que se generarían al funcionar correctamente el sistema. También es posible clasificar los modos de avería según el **dominio**: el comportamiento de un sistema puede ser analizado según el dominio de valores o el dominio temporal. Consecuentemente, según este criterio existen dos tipos de averías: las averías de valor (*value failures*) que se producen cuando el valor proporcionado por el sistema no cumple con las especificaciones; y las averías de sincronización (*timing failures*), producidas cuando la temporización del servicio no es la esperada (muy relevantes en sistemas de tiempo real). Finalmente, en sistemas distribuidos pueden darse dos tipos de averías según la **percepción de los usuarios**. Las averías consistentes (*consistent failures*) en las que todos los nodos tienen una visión idéntica del sistema (existe un estado consistente) y las averías inconsistentes (*inconsistent failures*) o bizantinas [LSP82] en las que diferentes nodos pueden tener diferentes visiones del estado actual del sistema.

Una agrupación más práctica de los modos de avería consiste en una clasificación jerárquica [Pol96] donde cada modo está incluido en el superior (cuanto más abajo en la jerarquía, más restringido y más benevolente será el modo de avería). La clasificación es la siguiente:

- Averías bizantinas o arbitrarias (*bizantine failures*) [LSP82]: no existe ningún tipo de restricción referente a cómo el sistema puede fallar. Incluye comportamientos del tipo *two-faced* (envío de diferentes mensajes a diferentes nodos) y falsificación de mensajes (toma de la identidad de otro nodo).
- Averías bizantinas detectables por autenticación (*authentication detectable byzantine failures*) [DS83]: el sistema presenta un comportamiento como el anterior pero no es capaz de falsificar mensajes. Esto se consigue mediante la utilización de mensajes autenticados.
- Averías por cómputo incorrecto (*incorrect computation failures*) [LMJ91]: el sistema no es capaz de proporcionar resultados correctos en el dominio de va-

lores, el dominio temporal o ambos. No presenta un comportamiento malicioso ni inconsistente.

- Averías de rendimiento (*performance failures*) [Pow92]: el sistema proporciona resultados correctos en el dominio de valores pero no en el dominio temporal. Los mensajes transmitidos pueden ser entregados antes de lo esperado o con retraso (este último caso es especialmente crítico en sistemas de tiempo real) por lo que este tipo de averías también son denominadas averías de sincronización [BMD93].

- Averías por omisión (*omission failures*) [Pow92]: el sistema proporciona resultados incorrectos en el dominio temporal con la peculiaridad de que los mensajes transmitidos presentan un retardo infinito (se pierde el mensaje). Las averías son temporales por lo que peticiones posteriores a éstas pueden ser atendidas correctamente.

- Averías tipo *crash* (*crash failures*) [Pow92]: avería permanente que resulta en la omisión infinita de mensajes.

- Averías tipo *stopping* (*stopping failures*) [Lap92]: el sistema presenta una avería tipo *crash* y proporciona un valor constante. Este valor puede ser el último valor correcto, un valor predefinido, etc. Este tipo de averías permiten saber al resto de usuarios que el sistema se ha averiado.

Como se ha mencionado, los modos de avería expuestos anteriormente constituyen una jerarquía, siendo la avería bizantina la menos restringida y benevolente; y la avería tipo *stopping* la más restringida y benevolente. La jerarquía de modos completa se muestra en la figura 2.3.

### 2.2.2. Semántica de averías

La semántica de averías (*failure semantics*) de un sistema define el modo de avería menos benevolente que éste pueden presentar. Así un sistema con semántica de averías tipo *crash* únicamente será propenso a averías tipo *crash* o *stopping* pero no a averías por omisión por ejemplo.

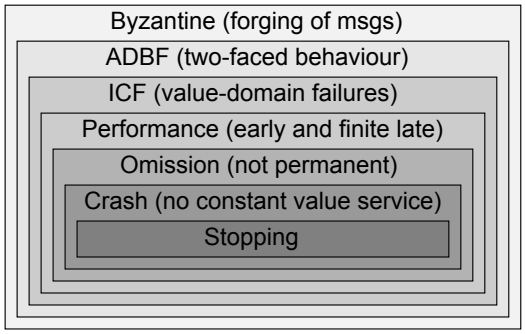



Figura 2.3: Jerarquía de modos de avería

En [Pol96] encontramos la siguiente definición:

«Un sistema genérico presenta una semántica de averías dada si la probabilidad de producirse una avería no contemplada por dicha semántica es lo suficientemente baja».

Resulta útil considerar una semántica de averías para los diferentes subsistemas que componen un sistema más complejo ya que simplifica enormemente las técnicas de tolerancia a fallos y la integración posterior de éstas.

Tomando la definición de [Pol96], existe la probabilidad, aunque muy reducida, de que se produzca una avería no cubierta por la semántica de averías. Si esto ocurre, el sistema puede llegar a fallar. Dicha circunstancia ha llevado a proponer el concepto de *Assumption Coverage* [Pow92], definido como la probabilidad de que los modos de avería descritos por la semántica de averías son correctos en condiciones reales, una vez el sistema haya fallado [Pol96]. Esta cobertura es un factor crítico en el diseño de sistemas distribuidos tolerantes a fallos y debe ser elevado. Sin embargo, siempre resulta conveniente encontrar un término medio entre la complejidad del sistema y el *assumption coverage*. Así, un sistema con una semántica de averías restringida será simple pero presentará una baja cobertura. En cambio, un sistema con una semántica de averías bizantina podrá presumir de una cobertura muy elevada (en efecto, de valor unitario) pero su complejidad será también muy elevada.

La imposición de una semántica de averías u otra es realizada mediante mecanismos específicos añadidos al sistema durante la fase de diseño (uso de mensajes autenticados para imponer una semántica tipo ADBF, desconexión del sistema al detectar errores para imponer una semántica tipo *crash*, ect.).

## 2.3. Fundamentos de la tolerancia a fallos

En este apartado se discutirá uno de los medios clave del árbol de la garantía de funcionamiento: la tolerancia a fallos.

La tolerancia a fallos engloba todos aquellos métodos usados para diseñar un sistema con capacidad interna de preservar la ejecución correcta y continuada de las tareas a pesar de la ocurrencia de fallos hardware o software.

Éstos métodos ponen especial atención en la eliminación de cualquier punto de avería único (*single point of failure*), es decir, cualquier componente del sistema que, en caso de averiarse, produzca una avería del sistema completo. Sin embargo, en casos concretos puede ser aceptable que un sistema presente un punto único de avería siempre y cuando éste tenga una probabilidad de fallo muy reducida en relación al resto de componentes del sistema.

Las técnicas de tolerancia a fallos se dividen en dos bloques bien diferenciados: el procesado de errores y el tratamiento de fallos [AL81].

El procesado de errores consiste en la eliminación de errores del estado computacional antes de que éstos causen una avería del sistema. Esto puede realizarse de dos maneras:

- Por recuperación, detectando primeramente el error y reemplazando el estado erróneo por uno libre de errores. El estado erróneo puede ser remplazado por un estado pasado, guardado anteriormente mediante un punto de recuperación o *backup* (recuperación hacia atrás) o, reemplazado por un estado nuevo, aunque asumiendo una posible pérdida de información por no ser del todo correcta la predicción (recuperación hacia adelante) [AL81].

- Por compensación, diseñando el sistema de forma redundante de manera que produzca resultados correctos hasta en el caso de haberse producido un error (no incluye detección de errores). Un buen ejemplo de compensación de errores es la técnica TMR (*Triple Modular Redundancy*) donde se utilizan un número impar de módulos idénticos y un votador (ver figura 2.4). Si uno de los módulos falla, el resultado seguirá siendo correcto (se considera que la probabilidad de que fallen dos módulos a la vez es muy baja). Sin embargo, TMR presenta

ciertos inconvenientes: el votador es un punto único de avería, eso sí, con una probabilidad de fallo mucho menor que la de los módulos (es más simple) y; al no haber detección de fallos, si un módulo se avería de forma permanente, esto no será detectado y se producirá un denominado desgaste de redundancia. Este último inconveniente puede ser solventado mediante la utilización de votadores con detección de errores (ver figura 2.5).

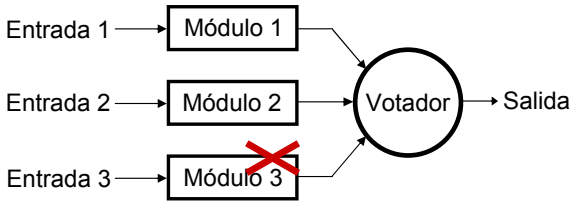



Figura 2.4: Módulo TMR

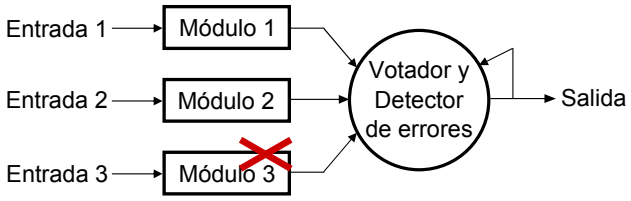



Figura 2.5: Módulo TMR con detección de errores

El tratamiento de fallos tiene como misión prevenir que un fallo cause un error por segunda vez. Dicha prevención puede realizarse de dos formas:

- Mediante el diagnóstico de fallos, utilizado para encontrar la causa del error (tanto su localización como su naturaleza).
- Mediante la pasivación de fallos, previniendo una segunda activación de un fallo conocido. Esto se suele conseguir desconectando del sistema los componentes defectuosos.

Existen dos posibles maneras de implementar las técnicas de tolerancia a fallos anteriores:

- Tolerancia a fallos específica a la aplicación: se utiliza información específica

sobre la aplicación (estimaciones del estado actual del sistema conociendo de antemano la dinámica de éste). Es una forma económica de implementación.

- Tolerancia a fallos sistemática: se basa en la replicación de componentes (redundancia) y posterior voto sobre los resultados individuales. Existen cuatro tipos de redundancia:

  - Redundancia hardware: uso de hardware adicional para detectar y/o compensar fallos. En la figura 2.6 se muestran dos ejemplos, un detector de errores basado en un comparador y un compensador TMR.
  - Redundancia software: uso de software adicional para detectar y/o tolerar, por ejemplo, fallos transitorios.
  - Redundancia en información: uso de información adicional como, por ejemplo, códigos de detección de errores.
  - Redundancia temporal: uso de tiempo adicional para detectar y, posiblemente, tolerar fallos (únicamente transitorios). La retransmisión de mensajes es un ejemplo claro de redundancia temporal.

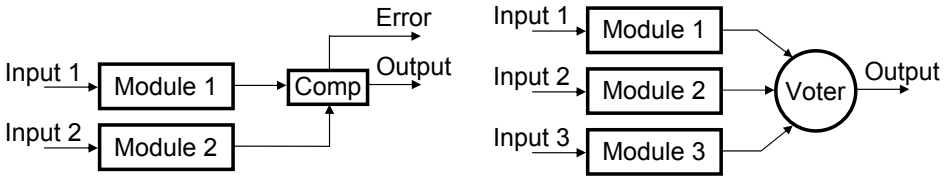



Figura 2.6: Ejemplos de redundancia hardware

Para garantizar la independencia de fallos, los componentes replicados de un sistema deben ser totalmente independientes respecto a su proceso de diseño y activación. Así por ejemplo, en un sistema diseñado para tolerar fallos físicos, los módulos pueden ser idénticos ya que se supone que componentes hardware fallan de manera independiente. En cambio, en un sistema diseñado para tolerar fallos de diseño, no es recomendable usar módulos idénticos ya que un fallo puede causar errores en todos y cada uno de ellos. En este caso la independencia de fallos únicamente se alcanzará si existe diversidad de diseños (*design diversity*).

Desgraciadamente, en sistemas distribuidos la independencia inicial de sus diferentes partes puede sufrir durante la operación. Esto ocurre si errores de elementos

defectuosos se propagan por el medio de comunicación y afectan otros elementos, inicialmente no defectuosos (figura 2.7).

Para afrontar dicha situación se suelen definir las denominadas regiones de contención de errores (*error containment regions*), incluyéndose mecanismos de protección específicos (por ejemplo un *bus guardian*, módulo agregado a cada nodo del sistema que vigila el correcto funcionamiento de éste y que, en caso de detectar un fallo, provoca un *crash*).

La efectividad final de las técnicas expuestas, por lo potentes que resulten, es limitada. Por este motivo se define un parámetro denominado cobertura (*coverage*) que expresa la mencionada efectividad. Este parámetro es estimado experimentalmente mediante la inyección de fallos al sistema. La garantía de funcionamiento final de un sistema depende sensiblemente de su valor.

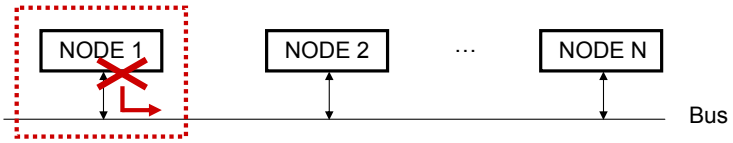



Figura 2.7: Propagación de errores

## 2.4. Estrategias para el diseño de sistemas tolerantes a fallos

Diseñar un sistema completamente tolerante a fallos es una tarea compleja. Por ello resulta interesante encontrar aproximaciones sistemáticas al problema, tales como el *Paradigma de Diseño del Prof. Avizienis* [Avi95].

Antes de describir dicho paradigma, seguidamente se exponen unas recomendaciones generales, basadas en analogías biológicas (garantía de funcionamiento en los organismos):

- Primera observación: los mecanismos de defensa inmediatos de un organismo son autónomos (basados en "hardware") y no necesitan funciones cognitivas ("software").

- Segunda observación: los mecanismos de defensa mencionados están distribuidos y se comparten por los diferentes subsistemas del organismo.
- Tercera observación: la clave es la diversidad ya que protege a las especies de la extinción.
- Cuarta observación: en las estructuras sociales es importante mantener una visión consistente de la realidad.

Regresando al *Paradigma de Diseño de Avizienis*, éste diferencia tres fases en el proceso de diseño de un sistema tolerante a fallos: las especificaciones, el diseño propiamente dicho y la posterior evaluación. En cada una de las fases anteriores existen diferentes pasos a seguir. En la figura 2.8 se exponen los pasos de mayor importancia [Avi95].

## 2.5. Conclusiones

En este apartado se han descrito los conceptos fundamentales relacionados con la garantía de funcionamiento. En posteriores capítulos a menudo se hará referencia a estos conceptos por lo que resultan esenciales para entender el trabajo expuesto en esta memoria. Primeramente se han detallado las ideas básicas relacionadas con la garantía de funcionamiento. Seguidamente se han presentado los fundamentos de la tolerancia a fallos y, finalmente, se han expuesto las estrategias básicas para el diseño de sistemas tolerantes a fallos. En esta última sección se ha descrito el Paradigma de Diseño de Avizienis, paradigma en el que se fomenta el proyecto CANbids.

Dicho proyecto, en el que se incluye este trabajo, tiene como meta integrar en el protocolo CAN las mejoras necesarias para que éste pueda ser utilizado como subsistema de comunicaciones en sistemas de control distribuidos con elevada garantía de funcionamiento y es tratado exhaustivamente en el capítulo 3.

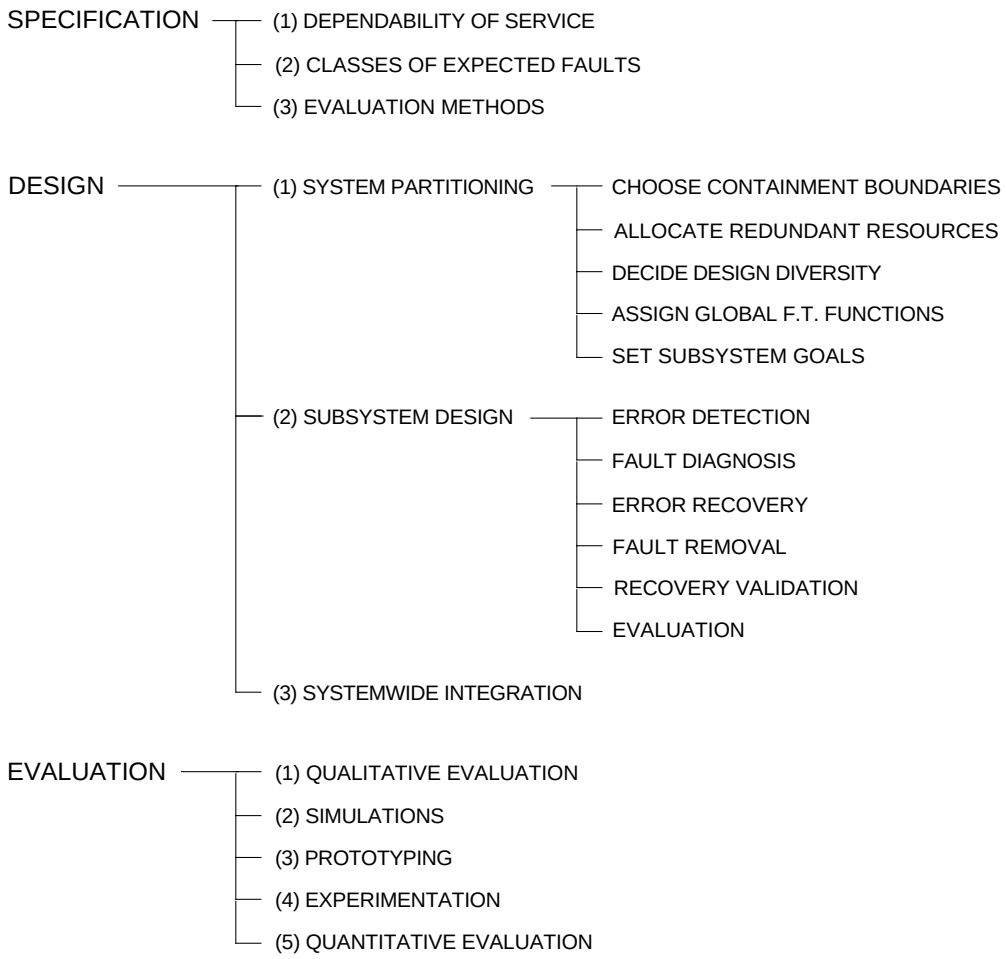



Figura 2.8: El Paradigma de Diseñp de Avizienis

# Capítulo 3

# CANbids

## 3.1. Introducción

El protocolo CAN es una tecnología madura, con excelentes prestaciones y bajo coste, pero no todo son ventajas. El protocolo presenta ciertas limitaciones por las que se le atribuye una baja garantía de funcionamiento [PIME08]. Consecuentemente, se suele considerar como poco apropiado para aplicaciones con elevada garantía de funcionamiento como por ejemplo *X-by-Wire* [PIME04] [KOPE99]. Aunque existan tecnologías competidoras con un nivel elevado de *dependability*, tales como FlexRay o TTA, éstas tienen un coste elevado. Este es el motivo por el cual diferentes grupos de investigación, incluyendo el de la Universidad de las Islas Baleares, han seguido apostando por CAN, proponiendo mecanismos y mejoras para superar sus limitaciones y aumentar su garantía de funcionamiento

En base a varios años de investigación se propuso el proyecto CANbids (*CAN-Based Infrastructure for Dependable Systems*) [SRV]. Dicho proyecto integra diferentes mecanismos propuestos por el propio grupo de investigación y por otros grupos colaboradores y tiene como objetivo final desarrollar una infraestructura basada en CAN para el soporte de sistemas de control distribuidos con elevada garantía de funcionamiento.

El capítulo se organiza de la siguiente forma: primeramente se expone las *Propiedades del protocolo CAN*, en segundo lugar se detallan las *Limitaciones de CAN*

para, seguidamente, describir las *Soluciones propuestas dentro de CANbids*. Finalmente se explican las *Tareas pendientes*.

A continuación se introducen las soluciones a las distintas limitaciones de CAN que se han ido proponiendo dentro del proyecto CANbids. Las soluciones con las que se ha tratado en este trabajo serán, además, explicadas en detalle en el apartado *Soluciones propuestas dentro de CANbids*:

- OCS-CAN (*Orthogonal Clock Subsystem for CAN*): propuesta para la sincronización de relojes sobre CAN [RODR10]. Proporciona una infraestructura para la implementación de mecanismos encaminados a solucionar (1) la falta de un servicio de sincronización de relojes y; (2) el *Jitter* elevado y variable.
- CANcentrate y ReCANcentrate: topologías estrella para CAN. Solventan la (1) limitada contención de errores y (2) el limitado soporte para tolerancia a fallos de CAN. Además, pueden mejorar el producto velocidad - distancia según la aplicación.
- sfiCAN (*Star-based physical Fault Inyector for CAN*): un inyector de fallos para la generación de escenarios de error complejos. No representa una solución en sí para alguna de las limitaciones de CAN pero es una herramienta vital para la verificación experimental del resto de mecanismos.
- CANsistant (*(CAN Assistant for Consistency)*): propuesta para la detección de escenarios de inconsistencia. Soluciona, en parte, la limitada consistencia de datos de CAN.
- Señalización de errores consistente: propuesta para solucionar la señalización inconsistente de errores. En conjunto con CANsistant, solventa la limitada consistencia de datos de CAN.

## 3.2. Propiedades del protocolo CAN

CAN (acrónimo del inglés *Controller Area Network*) es un protocolo de comunicaciones desarrollado por la empresa alemana *Robert Bosch GmbH* en los años 80, basado en una topología bus para la transmisión de mensajes en sistemas distribuidos. Presenta varias ventajas como su bajo coste, configuración simple, flexibilidad,

robustez electromagnética [FOFF04], respuesta en tiempo real (tiempos de latencia acotados), arbitraje basado en prioridades así como sus mecanismos de detección y contención de errores. En un principio, CAN fue diseñado para reducir el costo del cableado en aplicaciones automotrices pero pronto se convirtió en un protocolo muy popular, utilizado en muchas otras aplicaciones de control tales como la automatización industrial, robótica, equipos médicos, comunicación en edificios, etcétera [PIME08] [GAW09] [CAV05] [NOL05] [PIME04].

CAN implementa la capa física y la capa de enlace del modelo OSI (*Open System Interconnection Basic Reference Model*). Su capa de enlace fue especificada por *Bosh* en 1991 [Gmb91]. Posteriormente, en 1993, el protocolo fue estandarizado [ISO93] por la Organización Internacional para la Estandarización (ISO, *International Organization for Standarization*). En el 2003, el estándar fue actualizado [ISO03a]. De este modo, existen dos especificaciones de CAN normalizadas: CAN 2.0A, de baja velocidad (hasta 125 Kbps) y CAN2.0B, de alta velocidad (hasta 1 Mbps).

### 3.2.1. Capa física

CAN es un protocolo de comunicaciones serie que soporta control distribuido en tiempo real con un alto nivel de fiabilidad. Sus características vienen definidas por las dos últimas capas del modelo OSI: la capa física, que define los aspectos relacionados con las conexiones físicas de la red, tanto en lo que se refiere al medio físico como a la forma en la que se transmite la información: niveles de señal, representación, sincronización, tiempos en los que los bits se transfieren al bus, etc.; y la capa de enlace, responsable de la transferencia fiable de información a través de la red, ocupándose del direccionamiento físico, de la topología de red, del acceso al medio, de la detección y el tratamiento de errores, de la distribución ordenada de tramas y del control de flujo.

Existen varias especificaciones de la capa física de CAN. Las más importantes son: ISO 11898-2 (alta velocidad) [ISO03b], ISO 11898-3 (tolerante a fallos), SAE J2411 (cable único) e ISO 11992 (punto a punto). Para el trabajo realizado se optó por el estándar ISO 11898-2 ya que es considerado el más popular y extendido.

El subsistema de comunicaciones CAN está basado en una topología bus del tipo simplex (transmisión en un único sentido). El medio físico se constituye por

una línea de transmisión diferencial especialmente resistente a interferencias electromagnéticas, terminada por ambos lados con impedancias de 120 Ohmios para eliminar reflexiones de señal indeseadas.

Al bus pueden conectarse diferentes nodos, formados por: la aplicación propiamente dicha (basada normalmente en una CPU), un *controlador CAN* y un *Transceiver* (ver Figura 3.1). El número de nodos conectados a una red CAN está limitado, en la práctica, por el retardo de transmisión entre nodos, las cargas eléctricas sobre el bus y el tipo de *transceivers* utilizado.

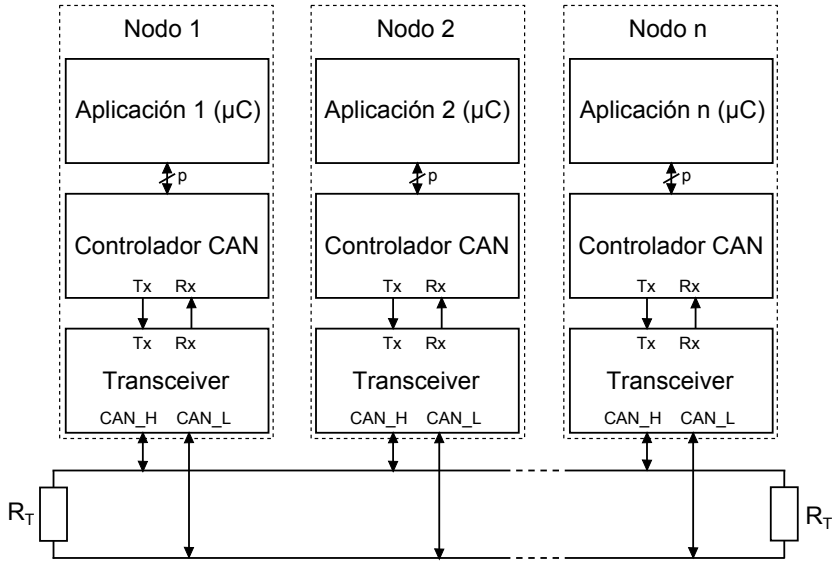



Figura 3.1: Conexionado de un bus CAN

La codificación digital utilizada por los *transceivers* para convertir las señales analógicas provenientes del medio físico a valores binarios es del tipo NRZ (del inglés *Non Return to Zero*). Dicha codificación se caracteriza por utilizar dos niveles de voltaje: un *'1'* lógico es representado por una tensión positiva y un *'0'* lógico por una tensión negativa (ver Figura 3.2).

Por definición, el bus CAN puede tomar dos valores complementarios: *dominante* (*'d'*), normalmente de valor lógico *'0'* o *recesivo* (*'r'*), de valor lógico *'1'*. Otra característica importante del bus es la implementación, cableada, de una función *AND* de todas y cada una de las contribuciones de los nodos del sistema. De este modo, siempre que un nodo transmita un valor dominante, éste será recibido por

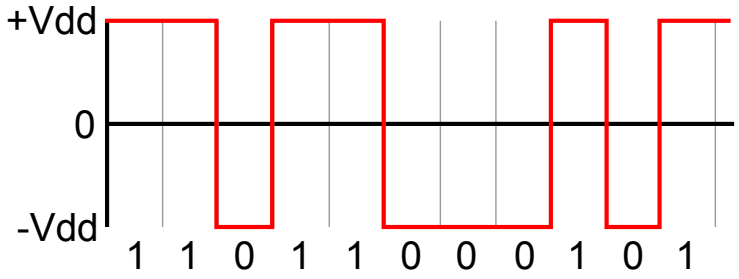



Figura 3.2: Codificación NRZ

todos los otros nodos. En cambio, si transmite un valor recesivo, éste únicamente será recibido si el resto de nodos también han enviado un recesivo.

El último aspecto definido por la capa física del protocolo CAN es el *mecanismo de sincronización de bit*. Dicho mecanismo garantiza que los nodos muestreen los bits transmitidos por el canal de manera cuasi-simultánea (propiedad denominada *in-bit response*). Se utilizan las transiciones recesivo - dominante para mantener los nodos receptores sincronizados con el nodo transmisor (*leading transmitter*). La sincronización, a nivel físico, implica que cada bit transmitido recorra el canal y se estabilice. Éste es el motivo por el cual la longitud máxima del bus es limitada y, a su vez, inversamente proporcional a la velocidad de transmisión. Así, para una velocidad de 1 Mb/s la longitud máxima del medio es de 40 m [CiAa]. En cambio, para alcanzar una longitud de 1000 m, la velocidad de transmisión deberá ser reducida a 40 Kb/s [CiAa].

### 3.2.2. Capa de enlace

La capa de enlace proporciona los mecanismos necesarios para garantizar que la información se trasmita correctamente, incluso en presencia de errores, entre los nodos conectados al bus. Se compone por dos subcapas: la subcapa *MAC* (*Medium Access Control*), que define el tipo de direccionamiento y controla el acceso al medio; y la subcapa *LLC* (*Logical Link Control*), que especifica las herramientas de control de flujo y detección/tratamiento de errores.

### Direccionamiento de la información

En CAN se utiliza direccionamiento basado en mensajes por lo que los nodos conectados al bus no disponen de una dirección propia. A cada mensaje se le asigna un identificador, independiente de su fuente y destino, que describe su prioridad y contenido. En base a esa información cada nodo decidirá si el mensaje en proceso de transmisión le es relevante o no.

La propiedad descrita hace que una red CAN sea altamente flexible y extensible ya que la agregación de nuevos nodos al sistema es inmediato (no requiere la modificación de los nodos existentes).

### Modelo de cooperación

CAN se basa en el modelo de cooperación productor/consumidor. Las transacciones (transmisión de datos) son iniciadas por los nodos que generan información (productores). Los nodos receptores (consumidores) monitorizan los datos enviados por el canal y los recuperan de la red si les son relevantes. El modelo está basado en el modo de difusión (*broadcast*). Toda la información transmitida es recibida, sin excepción, por todos los nodos del sistema.

### Formato de las tramas

La información a transmitir se divide en paquetes de datos denominados *tramas*. Como ya se ha mencionado anteriormente, existen dos especificaciones estandarizadas del protocolo: *CAN2.0A* y *CAN2.0B*. Éstas utilizan formatos de tramas ligeramente diferentes. En *CAN2.0A* se adopta el denominado formato de tramas estándar (*standard frame format*) que se caracteriza por poseer un identificador de 11 bits. El formato utilizado en *CAN2.0B* se denomina formato de tramas extendido (*extended frame format*) y presenta 29 bits de identificación del mensaje.

CAN incluye cuatro tipos de tramas diferentes: tramas de datos, tramas remotas, tramas de error y tramas de sobrecarga. En este apartado se describen los formatos de las dos primeras. Las tramas de error y las tramas de sobrecarga se explican en las secciones *Señalización de errores* y *Tramas de sobrecarga* respectivamente.

Las tramas de datos, tal como su nombre indica, sirven para transmitir datos de información. Con una trama remota, en cambio, se realiza una petición de envió de datos desde otro nodo. El formato de los dos tipos de tramas, según la especificación, se muestra en la Figura 3.3.

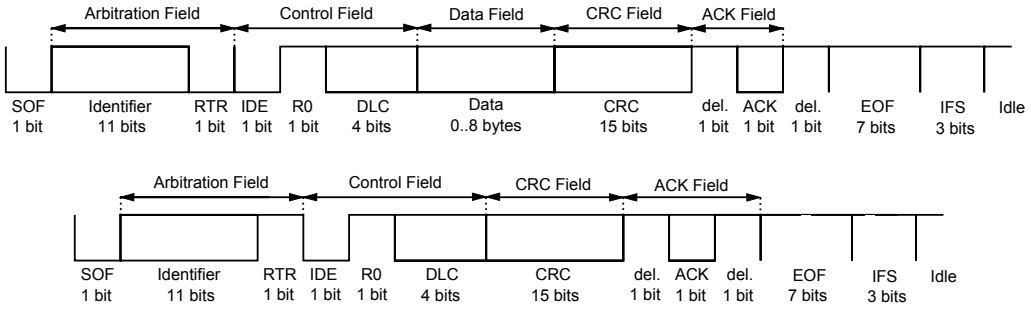



Figura 3.3: Trama de datos (arriba) y trama remota (abajo)

Como bien se puede apreciar en las figuras anteriores, las tramas se componen por los siguientes campos:

- Inicio de trama (SOF, *Start of Frame*): consta de un único bit dominante y marca el inicio de una trama de datos o una trama remota. Un nodo únicamente accederá al bus cuando éste esté libre. Así, la consecuente transición recesivo - dominante servirá para sincronizar los nodos receptores con el nodo transmisor (tal como se ha explicado en 3.2.1). En caso de que varios nodos intenten acceder cuasi-simultáneamente al medio se iniciará un proceso de arbitraje. El mecanismo de arbitraje utilizado por CAN se explica en la sección *Mecanismo de arbitraje*.
- Campo de arbitraje: utilizado para llevar a cabo el proceso de arbitraje. Consta del identificador y del bit RTR.
  - Identificador: constituido por 11 bits en formato estándar y 29 bits en formato extendido y proporcionado por la subcapa *LLC*. Como se ha explicado en *Direccionamiento de la información*, no identifica a los nodos fuente o destino sino al mensaje en sí. En las tramas remotas se facilita el identificador de la trama que se solicita.
  - Bit RTR (*Remote Transmission Request*): toma valor *'d'* si se trata de una trama de datos o *'r'* si se trata de una trama remota.

- Campo de control: consta de seis bits. Incluye el bit de extensión del identificador (*IDE*), de valor *'d'* para tramas en formato estándar y valor *'r'* para tramas en formato extendido; un bit reservado para futuras ampliaciones del protocolo denominado *R0* y cuatro bits *DLC* (*Data Length Code*) que especifican el número de bytes de datos que incluye la trama en el caso de una trama de datos o el número bytes de datos solicitados en el caso de una trama remota. En formato extendido se añaden dos bits adicionales: el *SRR* (*Substitute Remote Request*) y un segundo bit reservado *R1*.

- Campo de datos: en una trama remota (*RTR='1'*) el campo de datos es inexistente. En el caso de ser una trama de datos, su longitud viene definida por el *DLC* y contiene los datos proporcionados por la subcapa *LLC*. El número máximo de bytes incluidos en una trama es ocho.

- Campo CRC: consta de 16 bits. Contiene el CRC (Código de Redundancia Cíclica) propiamente dicho, de 15 bits, seguido por el bit delimitador de CRC (*CRC delimiter*) de valor *'r'*. Su función es la detección de posibles errores de transmisión: si la secuencia calculada por el receptor no es idéntica a la recibida la trama será rechazada. En el capítulo 4.2, página 69, se especifica su algoritmo de generación.

- Campo de confirmación: formado por dos bits, el bit *ACK* (*Acknowledge*) y el bit delimitador de *ACK* (*ACK delimiter*), siempre de valor *'r'*. Sirve para confirmar el envío correcto de una trama. El nodo transmisor envía un valor recesivo y queda a la espera de un valor dominante enviado por parte de los receptores. En caso de no recibir confirmación, el transmisor procederá a la retransmisión de la trama.

- Fin de trama (*EOF*, *End of Frame*): consta de siete bits recesivos que marcan el final de la trama

- Campo de intermisión (*IFS*, *Intermission Frame Space*): una vez enviados los siete bits del *EOF* se envían tres bits recesivos adicionales. Seguidamente el bus queda libre (estado recesivo o *Idle*) pudiéndose iniciar una nueva transmisión (ver Figura 3.4). Durante el *IFS* no está permitido iniciar una trama de datos o remota pero sí una trama de sobrecarga por lo que la detección de un valor dominante en este campo siempre será interpretado como el comienzo de una

trama de este tipo. La detección de un bit *'d'* en el campo *Idle* se interpreta como *SOF* de una nueva trama.

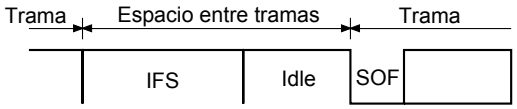



Figura 3.4: Campo de intermisión

### Mecanismo de arbitraje

Como el bus CAN es un bus multimaestro (cualquier nodo puede ser transmisor de un mensaje), es posible que varios nodos intenten iniciar una transmisión simultáneamente. Esto podría provocar colisiones indeseadas y, consecuentemente, la pérdida de datos. Por ello, siempre que se produzca un acceso múltiple al canal se pone en marcha un proceso de arbitraje, basado en la contención, para decidir qué nodo ganará el acceso al bus. Dicho proceso hace uso del denominado mecanismo de arbitraje bit a bit (*bit-wise arbitration mechanism*) de CAN.

El método de acceso al medio utilizado es el de Acceso Múltiple por Detección de Portadora, con Arbitraje por Prioridad de Mensaje (CSMA/BA, *Carrier Sense Multiple Access with Bit-wise Arbitration*). De acuerdo con este método, los nodos en la red que necesitan transmitir información deben esperar a que el bus esté libre (detección de portadora); cuando se cumple esta condición, dichos nodos transmiten un bit de inicio (acceso múltiple). Cada nodo observa el bus bit a bit mientras trasmite su campo de arbitraje (identificador más *RTR*) y compara el valor transmitido con el valor recibido. Si uno de los nodos transmisores monitoriza un valor dominante, *'0'*, habiendo transmitido un valor recesivo, *'1'*, considerará que ha perdido el arbitraje, se retirará, se convertirá en un nodo receptor e intentará transmitir de nuevo cuando el bus haya quedado libre. En la Figura 3.5 se muestra un ejemplo simple donde el *nodo A* gana el arbitraje al *nodo B* en el octavo bit del identificador.

De este modo, el nodo con el menor identificador de mensaje es el que tiene mayor prioridad y, por lo tanto, será el que gana el arbitraje. El bit *RTR* también es incluido en el arbitraje por lo que una trama de datos (*RTR='0'*) siempre tendrá mayor prioridad que una trama remota (*RTR='1'*).

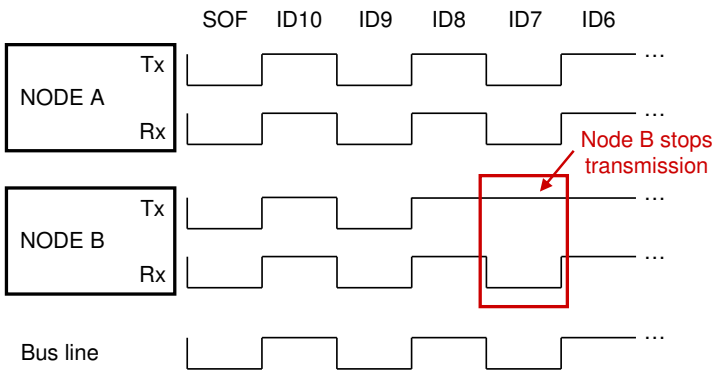



Figura 3.5: Ejemplo de arbitraje en CAN

La asignación de prioridades es estática: la asignación de identificadores a los mensajes se realiza durante el diseño del sistema y no puede modificarse una vez la red esté en funcionamiento.

### Codificación de las tramas

Como ya se ha mencionado en 3.2.1, la codificación digital utilizada en CAN es NRZ. También se ha mencionado que la sincronización entre nodos se realiza única- y exclusivamente en las transiciones recesivo - dominante del bus. Así pues, una larga cadena de bits del mismo valor podrían llegar a desincronizar los nodos.

Por el motivo anterior, los campos SOF, Identificador, RTR, control, datos y CRC son codificados mediante el método del bit complementario (*stuff rule*). Este método consiste en la inserción de bits redundantes o *stuff bits* en la secuencia transmitida con el objetivo de facilitar la sincronización de los nodos y poder detectar/señalizar posibles errores en la red.

Concretamente, siempre que el transmisor detecte cinco bits consecutivos de la misma polaridad (incluidos los mismos bits complementarios) en la trama que es transmitida, automáticamente insiere un bit de valor complementario al de la secuencia. Dicho proceso es ilustrado en la Figura 3.6.

El resto de campos de la trama (bit delimitador de CRC, campo ACK y EOF) no son codificados según la técnica de *stuffing*.

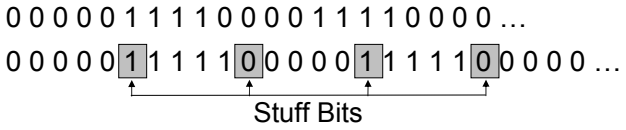



Figura 3.6: Regla del bit complementario

### Detección de errores

Una de las características más interesantes del protocolo CAN es su capacidad de gestión de errores. Cada nodo del sistema incorpora mecanismos de detección, señalización y contención de errores y es capaz de detectar cinco tipos de error diferentes [ISO93]: errores de bit complementario, errores de formato, errores de bit, errores de CRC y errores de ACK.

1. Error de bit complementario (*stuff error*): tanto los receptores como el transmisor comprueban si se respeta la regla de stuff (*stuff rule*) en los campos de la trama afectados por dicha regla. Concretamente, se detecta este tipo de error si se monitorizan seis bits consecutivos de la misma polaridad.

2. Error de formato (*format error*): los receptores y el transmisor detectan un error de formato en caso de que un campo de formato fijo contenga un valor no permitido. Los casos son los siguientes:

   - Recibir un bit *'d'* en el delimitador de CRC, bit que siempre debe tomar valor *'r'*.
   - Recibir un bit dominante en el delimitador de ACK, por definición siempre recesivo.
   - Recibir bits *'d'* en el campo EOF, campo formado únicamente por bits *'r'*. Existe una excepción: cuando los nodos receptores monitoricen un valor dominante en el último bit del EOF, éste es considerado como el inicio de una trama de sobrecarga (más información en la sección *Tramas de sobrecarga*).

3. Error de bit (*bit error*): cada nodo (sea transmisor o receptor) monitoriza el canal después de transmitir un bit. Se detecta un error de bit si se ha transmitido un valor dominante y el valor leído del bus es recesivo. Lo mismo

ocurre si se ha enviado un recesivo y se monitoriza un dominante (excepto en el campo de arbitraje y el bit ACK).

4. Error de CRC (*CRC error*): como se ha explicado anteriormente, el transmisor de una trama calcula la secuencia CRC a partir de ésta. Seguidamente transmite la trama conjuntamente con el CRC. Los receptores, al recibir la trama, vuelven a calcular el CRC, detectándose un error si la secuencia calculada no es idéntica a la recibida.

5. Error de ACK (*ACK error*): si un receptor desea aceptar una trama recibida, éste enviará un bit dominante en el campo de confirmación. Si el transmisor no monitoriza un bit de valor *'d'* en el campo ACK, supondrá que todos los receptores han rechazado la trama.

#### Señalización de errores

Con los mecanismos expuestos en el apartado anterior pueden detectarse gran cantidad de errores. Sin embargo, es conveniente identificar dos clases de errores: los errores globales, detectados por todos los nodos de la red y los errores locales, detectados solamente por uno o unos cuantos nodos. La aparición de éstos últimos viene condicionada por:

- Desplazamientos temporales de los relojes locales (*clock drift*) y los consecuentes desplazamientos de los puntos de muestreo, por lo que diferentes nodos pueden muestrear valores diferentes.

- Atenuaciones, dispersiones e interferencias sobre el medio provocando diferentes niveles de tensión en diferentes puntos del canal.

Para una garantía de funcionamiento elevada, resulta indispensable mantener una visión consistente entre todos los nodos del estado actual del sistema. Los errores locales pueden llevar a inconsistencias, provocando que unos nodos acepten una trama y otros la rechacen. Así pues, la señalización de errores tiene como objetivo principal la globalización de este tipo de errores y, de este modo, garantizar la consistencia.

La señalización se lleva a cabo mediante la transmisión de una trama de error (*error frame*). Dicha trama es enviada en el bit posterior a la detección de un error, excepto en el caso de los errores de CRC, caso en el que la trama de error es transmitida en el primer bit del EOF.

Una trama de error es formada por el denominado señalizador de error (*error flag*), seguido por el delimitador de error (*error delimiter*). El *error flag* se compone por seis bits consecutivos de la misma polaridad de manera que provocará, una vez transmitido, un error de bit complementario en los nodos no afectados por el error local. De este modo el error inicial es globalizado y la trama transmitida es rechazada de manera consistente por todos los nodos del sistema.

El formato de las tramas de error depende del estado en el que se encuentra un nodo: el estado de error activo o el estado de error pasivo (los estados de error se detallan en la sección *Contención de errores*). Un nodo en estado de error activo transmitirá un señalizador de error activo (*active error flag*), compuesto por seis bit dominantes. Si el nodo, en cambio, se encuentra en el estado de error pasivo, éste transmitirá un señalizador de error pasivo (*passive error flag*) formado por seis bits recesivos.

El envío de un señalizador de error activo siempre provocará la globalización de un error ya que los bits *'d'* sobrescriben en todo momento los bits *'r'* (ver la sección *Mecanismo de arbitraje*). Justo por este mismo motivo, un nodo en estado de error pasivo puede no ser capaz de forzar errores en el resto de nodos (los bits *'r'* no tienen influencia sobre el bus) y, por lo tanto, no se garantiza la globalización de errores.

Una vez globalizado un error, todos los nodos de la red iniciarán la transmisión de sus propios señalizadores de error (sean activos o pasivos). Esto puede dar lugar a la superposición de varios señalizadores, observándose sobre el bus una secuencia de entre seis y doce bits dominantes consecutivos. Una vez enviados los *error flags* superpuestos, los nodos envían, de manera conjunta, el delimitador de error (denominado *cooperative error delimiter*) compuesto por un mínimo de ocho bits recesivos consecutivos. Cada nodo, después de enviar su señalizador de error, trasmite bits *'r'* y monitoriza el bus hasta detectar un bit *'r'*, momento en el que se supone que todos los nodos han finalizado la transmisión de sus señalizadores de error. Entonces se transmiten, de forma cooperativa, siete bits recesivos más. Con esto se consigue que todos los nodos del sistema detecten cuasi-simultáneamente el final de la trama

de error (Figura 3.7).

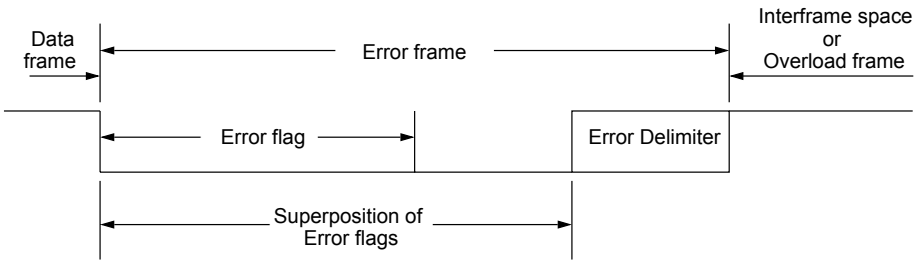



Figura 3.7: Trama de error

Finalmente, la trama de error viene seguida por el IFS y el estado *Idle* (el bus vuelve estar libre).

### Recuperación de errores

Como se ha visto en el apartado anterior, en caso de haberse detectado un error en la transmisión de una trama, se inicia un mecanismo de señalización de errores que globaliza dicho error. Una vez todos los nodos del sistema hayan enviado sus tramas de error y el bus haya quedado libre, el transmisor del mensaje erróneo se encargará de retransmitir dicho mensaje de forma automática.

### CANfidant

Los mecanismos de gestión de errores anteriores son capaces de tratar escenarios de error complejos. Para analizar y estudiar estos escenarios se propuso, dentro del proyecto CANbids, una herramienta de simulación denominada CANfidant [RODR03b].

CANfidant es un simulador diseñado por el propio grupo de investigación que permite estudiar simultáneamente el comportamiento de los diferentes controladores CAN que componen una red CAN. Permite la inyección de fallos tanto globales como locales con lo que es posible generar escenarios de error complejos. Es un instrumento muy útil y será usado a menudo para la identificación de los diferentes escenarios de inconsistencia expuestos en capítulos posteriores de esta memoria.

Incluye varios parámetros de configuración para definir entre otros qué nodo realiza el papel de transmisor, qué tipo de tramas se transmiten (de datos o remotas), cuál es el formato de las tramas (identificador, datos, etc.) y dónde inyectar los errores [RODR03b]. Una vez configurado, representa gráficamente las tramas que ha transmitido y recibido cada nodo. Esto ayuda enormemente en la identificación de escenarios de inconsistencia. En la Figura 3.8 se muestra la pantalla de inicio del programa.

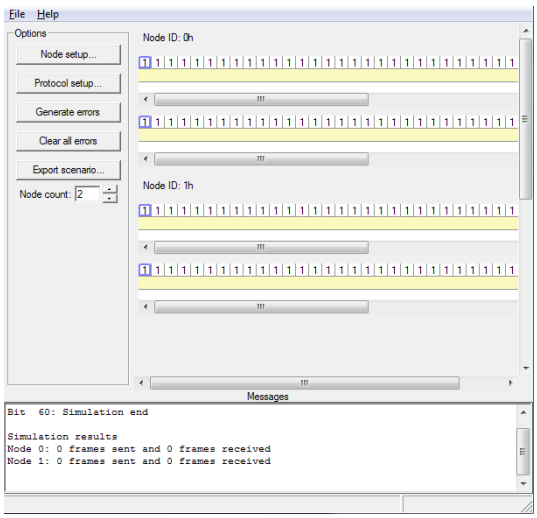



Figura 3.8: El simulador CANfidant

### Contención de errores

Un nodo CAN es capaz de distinguir entre fallos intermitentes y fallos permanentes y, consecuentemente, autodiagnosticar una posible avería.

El estándar CAN especifica diferentes mecanismos de contención de errores. Así, cada nodo CAN incluye dos contadores de errores [ISO93]: el contador de errores de transmisión (TEC, *Transmission Error Counter*) y el contador de errores de recepción (REC, *Reception Error Counter*). Dependiendo del valor de estos contadores, el nodo se encontrará en un estado u otro, minimizando en mayor o menor medida la influencia de éste sobre el canal.

Los estados del nodo son los siguientes (adicionalmente, ver la Figura 3.9):

- Estado de error activo (*error active state*)
- Estado de error pasivo (*passive error state*)
- Desconectado del bus (*bus off*)

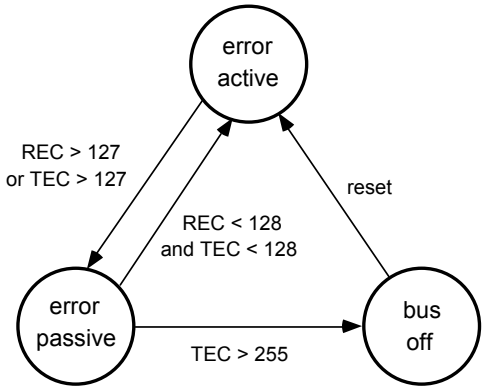



Figura 3.9: Estados de error de un controlador CAN

Un nodo, inicialmente en estado de error activo, incrementa sus contadores TEC/REC siempre que se detecte un error en el canal. Dicho incremento viene definido por unas reglas específicas [ISO93], imponiendo mayores penalizaciones a los errores producidos durante la transmisión que durante la recepción. Así, normalmente un nodo transmisor incrementa su contador TEC en ocho unidades y un nodo receptor, su contador REC en una unidad.

Un nodo incrementará sus contadores TEC o REC (dependiendo si el nodo actúa como transmisor o receptor) en ocho unidades adicionales si sospecha ser el responsable de la ocurrencia de un error. Esto ocurre cuando el nodo detecta un denominado error primario (*primary error*) [ISO93], esto es, que monitoriza un bit dominante después de enviar su *error flag*.

Aparte de las reglas para el incremento de los contadores mencionadas, también existen reglas para su decremento. De este modo, los contadores TEC/REC se decrementan en una unidad (a menos que estén a cero) si una trama ha sido transmitida/recibida satisfactoriamente. En el caso especial de tomar el contador REC un valor mayor a 127, éste será decrementado en más de una unidad (tomando normalmente un valor prefijado entre 119 y 127).

Como se ha mencionado anteriormente, un nodo inicialmente se encontrará en el estado de error activo. Sin embargo, si uno de sus contadores supera el valor de 127, el nodo entrará en el estado de error pasivo. La diferencia entre ambos estados radica en la señalización de errores. Tal como se ha explicado en la sección *Señalización de errores*, en estado de error activo el nodo enviará un señalizador de error activo (*active error flag*), en cambio, en estado de error pasivo enviará un señalizador de error pasivo (*passive error flag*). Así, un nodo en estado de error pasivo puede no ser capaz de globalizar un error local. Finalmente, un nodo se desconecta por completo del bus si el contador TEC es mayor a 255. Dicho estado, conocido como *bus-off*, lleva al nodo a autodiagnosticarse como averiado.

Bajo circunstancias determinadas, un nodo puede recuperarse de fallos intermitentes y volver al estado de error activo. Concretamente, para un nodo en estado de error pasivo, esto ocurrirá cuando sus contadores TEC y REC sean, ambos, iguales o menores a 127. Para un nodo desconectado, ocurrirá cuando éste haya monitorizado un total de 128 ocurrencias de bus libre (*CAN bus-free ocurrences*), es decir, 128 secuencias de once bits recesivos consecutivos [ISO93]. En la Figura 3.9 se muestra un diagrama de estados que esquematiza el proceso anterior.

### Tramas de sobrecarga

CAN especifica dos condiciones de sobrecarga [ISO93]. Las tramas de sobrecarga se envían, por un lado, cuando los nodos receptores necesitan unos retardos adicionales para procesar la trama recibida antes de transmitirse una nueva. Por otro lado, un nodo también inicia la transmisión de una trama de sobrecarga cuando detecta un bit *'d'* durante el campo de intermisión (*IFS*) o en el último bit del *EOF* (solo si el nodo es receptor). La segunda condición es denominada reactiva y sirve para globalizar la trama de sobrecarga disparada por la primera.

El formato de las tramas de sobrecarga es idéntico al de las tramas de error activo. Se componen por un señalizador de sobrecarga (*overload flag*) de seis bits dominantes y un delimitador de sobrecarga (*overload delimiter*) de ocho bits recesivos.

### 3.2.3. Conclusiones

El protocolo CAN es un subsistema de comunicaciones con muy buenas prestaciones para su uso en sistemas de control distribuidos. Sus principales ventajas son su configuración simple, flexibilidad, robustez electromagnética, respuesta en tiempo real, arbitraje basado en prioridades así como sus mecanismos de detección y contención de errores. Sin embargo, la propiedad más atractiva en relación a otros competidores como TTA o FlexRay es su reducido coste.

En este capítulo se han descrito las características fundamentales de CAN, poniendo especial interés en aquellas referentes a la capa física y la capa de enlace del modelo OSI. Sin embargo, el protocolo CAN presenta ciertas limitaciones no contempladas hasta el momento. Estas limitaciones, sobretodo en la contención de errores y la tolerancia a fallos, son el motivo por el cual se suele considerar que CAN no es apto para su uso en aplicaciones con elevada garantía de funcionamiento.

## 3.3. Limitaciones de CAN

Como ya mencionado en varias ocasiones, CAN presenta diferentes limitaciones. Concretamente, éstas son siete [PIME08] y se exponen a continuación:

1. ***Jitter* elevado y variable:** una propiedad importante de CAN es su mecanismo de arbitraje basado en un protocolo CSMA. Sin embargo, esta característica conlleva la aparición de un elevado *jitter* (o variación de retardo) ya que, si los instantes de transmisión no están sincronizados, un nodo que intente transmitir, siempre perderá el arbitraje contra aquel tráfico del bus que tenga mayor prioridad. El *jitter* introducido puede ser controlado mediante sincronización global y ajuste del *offset* relativo de los nodos del sistema.

2. **Falta de servicio de sincronización de relojes:** como se ha indicado anteriormente, es posible reducir el *jitter* mediante un servicio de sincronización de relojes. Desgraciadamente, el estándar CAN no incluye tal servicio. Por ello, si un sistema distribuido requiere dicha sincronización, ésta deberá ser proporcionada por la capa de aplicación. Esto se suele conseguir implementando un algoritmo de sincronización de relojes por software. También se han propuestos

soluciones hardware pero suelen ser menos comunes.

3. **Producto velocidad - distancia limitado:** para garantizar que los nodos muestreen los bits transmitidos por el bus de manera cuasi-simultánea (*in-bit response*), CAN incluye un mecanismo de sincronización de bit. Esta sincronización a nivel de bit implica, a nivel físico, que cada bit trasmitido recorra el canal completo y se estabilice. El tiempo necesario para que ello ocurra se denomina tiempo de bit. Así pues, cuanto mayor sea la longitud del canal, mayor será el tiempo de bit y, consecuentemente, menor la velocidad máxima de transmisión (ver el apartado 3.2.2 para más información). Posibles soluciones son el uso de topologías alternativas (p.e. topología estrella) o segmentación del canal (mediante el uso de *switches*).

4. **Flexibilidad limitada:** CAN es considerado un protocolo altamente flexible. No obstante, el mecanismo de arbitraje está basado en identificadores de mensaje estáticos que definen la prioridad de los diferentes mensajes. Esta asignación estática de prioridades resta flexibilidad al protocolo y dificulta, por ejemplo, la asignación de nuevos identificadores. Para solventar dicha limitación es necesario incorporar nuevos mecanismos como la actualización dinámica de identificadores o el control de transmisión a alto nivel. Para no comprometer la respuesta en tiempo real del sistema, se debe incorporar además un control de admisión que controle los cambios dinámicos realizados.

5. **Contención de errores limitada:** CAN incluye mecanismos de contención de errores. Concretamente, hace uso de dos contadores de error [ISO93]: el contador de error de transmisión TEC y el contador de error de recepción REC. Si uno de los mencionados contadores alcanza un valor predefinido, el controlador CAN entrará en estado *bus off*, es decir, se desconectará por completo del canal. El mecanismo descrito, sin embargo, tiene un tiempo de reacción relativamente elevado dependiendo del tipo de errores y su frecuencia de aparición. Otra limitación grave surge intrínsecamente de la topología bus, ya que errores que se produzcan en los *transceivers*, las líneas del bus o sus interconexiones se propagarán libremente por la red. El uso de una topología estrella en lugar de una topología bus es una posible solución. CAN tampoco incluye mecanismos de protección contra la transmisión errónea de mensajes, tanto en el dominio temporal como en el de valores. Un claro ejemplo son los fallos tipo *"babling idiot"*, es decir, cuando un nodo transmite un mensaje una y otra vez llegando

a bloquear por completo el canal. Para proteger el sistema de este tipo de fallos se requiere hardware adicional (p.e. *bus-guardians*, módulos acoplados a los nodos que controlan el correcto envío de mensajes).

6. **Soporte para tolerancia a fallos limitado:** aplicaciones con elevada garantía de funcionamiento deben ser capaces de tolerar todo tipo de fallos. Esto hace necesario el uso de técnicas de tolerancia a fallos. CAN incorpora algunas de estas técnicas (sobretodo mecanismos de detección de errores potentes) pero no alcanza el nivel de fiabilidad necesario para aplicaciones con elevada garantía de funcionamiento. Para su uso en aplicaciones de este tipo es necesario añadir mecanismos adicionales.

7. **Consistencia de datos limitada:** La consistencia de datos es uno de los factores clave de un sistema distribuido con alta garantía de funcionamiento. El protocolo CAN presenta diferentes escenarios de inconsistencia que surgen de las propiedades específicas de éste. Estos escenarios se manifiestan en forma de omisiones de mensaje inconsistentes (*inconsistent message omissions*), es decir, algunos nodos reciben el mensaje y otros no; o duplicados de mensaje inconsistentes (*inconsistent message duplicates*), es decir, algunos nodos reciben el mensaje una vez mientras que otros lo reciben dos o más veces.

Al ser la limitada consistencia de datos el objetivo principal del presente trabajo, a ésta se le dedica un apartado propio para discutir exhaustivamente los diferentes detalles y escenarios de inconsistencia.

### 3.3.1. Consistencia de datos

Según las especificaciones del protocolo CAN [ISO93], éste garantiza la consistencia de datos en presencia de errores intermitentes en el canal (en la línea o en los *transceivers*). Teóricamente, CAN exhibe el modo de difusión atómica, es decir, en todo momento está garantizado que una trama será aceptada por todos los receptores o por ninguno de ellos. El modo de difusión atómica se define por las siguientes propiedades:

- Validez: si un nodo correcto difunde un mensaje por la red, entonces este mensaje será entregado a un nodo correcto.

- Acuerdo: si un mensaje es entregado a un nodo correcto, también será entregado al resto de nodos correctos.
- Entrega única (*at-most-once delivery*): todo mensaje entregado a un nodo correcto será entregado como máximo una única vez.
- No trivialidad: todo mensaje entregado a un nodo correcto fue difundido por un nodo.
- Orden total: dos mensajes entregados a dos nodos correctos serán entregados en el mismo orden a los dos nodos.

Para cumplir con las propiedades anteriores, CAN incluye mecanismos específicos de detección y señalización de errores [ISO93]. Tal como se ha explicado en el apartado 3.2, CAN es capaz de detectar cinco tipos de errores: errores de bit complementario, errores de formato, errores de bit, errores de CRC y errores de ACK. La señalización de errores es efectuada mediante el envío de una trama de error compuesta por un señalizador de error activo, EF, (compuesto por seis bits dominantes consecutivos) y un delimitador de error (formado por ocho bits recesivos consecutivos). De esta forma el error local es globalizado y la trama es rechazada por todos los receptores y, seguidamente, reenviada por el transmisor. Así, supuestamente, se consigue mantener una visión consistente del estado global del sistema. Sin embargo, existen escenarios específicos con múltiples errores en los que el mecanismo exhibe ciertas vulnerabilidades y la consistencia de datos puede ser comprometida.

#### Vulnerabilidades del mecanismo de control de errores

Todas las vulnerabilidades del mecanismo de control de errores de CAN se basan en la siguiente observación: no todos los bits de un EF provocarán un error de canal y causarán el rechazo de la trama transmitida. Así pues, de entre los seis bits dominantes del EF, es posible distinguir entre aquellos bits que provocarán un error (EPB, *Error-Provoking Bits*) y aquellos que no lo provocarán (NPB, *Non error-Provoking Bits*). Para que un bit se pueda considerar como EPB se deben cumplir las siguientes dos condiciones:

1. Un bit de un EF únicamente podrá provocar un error si es parte de la trama. Si una trama ya ha sido aceptada por un receptor, el error ya no será capaz

de provocar un rechazo. Así, por ejemplo, si un nodo detecta un error en el último bit de la trama (es decir, en el último bit del EOF), entonces el EF subsiguiente no tendrá efecto ya que el envío de la trama ya habrá finalizado. Consecuentemente, todos los bits del EF serán NPB. En cambio, si el error es detectado en el penúltimo bit de la trama, entonces el primer bit del EF aún es parte de la trama y puede provocar un error (el primer bit será EPB y los cinco restantes NPB).

2. Un bit de un EF solo provocará un error si causa un cambio en el canal que es percibido como error. Esto significa que un EF, compuesto exclusivamente por bits dominantes, únicamente causará errores si el valor esperado en el canal es recesivo. Justo por este motivo los señalizadores de error están formados por seis bits dominantes que infringen la regla de *stuff*.

Aunque las condiciones anteriores parezcan simples, pueden llevar a escenarios complejos que deben ser analizados con precaución. Para ello se ha hecho uso del simulador CANfidant. Como se ha explicado en el apartado 3.2.2, esta herramienta permite estudiar el comportamiento de una red CAN frente a diferentes condiciones de error.

### Limitaciones conocidas

Hasta el momento se conocen dos fuentes básicas de inconsistencias en CAN:

**La existencia del estado de error pasivo**. Como se ha explicado en el apartado 3.2.2 y según el estándar CAN [ISO93], un nodo entrará en el estado de error pasivo si sus contadores de errores, TEC o REC, alcanzan un valor predeterminado. En dicho estado, el nodo usa señalizadores de error pasivos en lugar de activos (seis bits recesivos en lugar de seis bit dominantes) y, consecuentemente, puede no ser capaz de globalizar un error local. Si un nodo en estado de error pasivo es el único nodo en detectar un error y no es capaz de globalizarlo, esto provocará una inconsistencia ya que, mientras el resto de nodos aceptarán la trama, éste la rechazará. Por este motivo, no se suele permitir el estado de error pasivo en aplicaciones críticas. En el presente trabajo se considera que todos los nodos CAN se encuentran en estado de error activo.

**La regla del último bit del EOF**. En caso de ocurrir un error en el último bit del EOF de una trama, un nodo actuará de una u otra manera dependiendo de si es el transmisor de dicha trama o un receptor. Si es el transmisor de la trama, después de detectar el error enviará un señalizador de error, considerará la transmisión como errónea y procederá a la retransmisión de la trama. En cambio, si es un receptor, aceptará la trama recibida y procederá al envío de una trama de sobrecarga [RUFI98].

En escenarios concretos, identificados por Rufino *et. al*, la regla del último bit del EOF puede implicar una violación de la propiedad de consistencia de datos de CAN [RUFI98]. Considérese el caso de la Figura 3.10. Se produce un error en el penúltimo bit del EOF y éste únicamente es detectado por el subconjunto de nodos X. Los nodos pertenecientes a dicho subconjunto rechazarán la trama y enviarán un EF en el bit consecutivo, es decir, el último bit del EOF (solo el primer bit del EF será EPB, el resto serán NPB). Sin embargo, por la regla del último bit del EOF, el conjunto de receptores Y (no afectados por el error inicial) aceptarán la trama. Así, una parte de los nodos del sistema rechazarán la trama enviada y la otra parte la aceptarán, infringiéndose la segunda propiedad del modo de difusión atómica (validez) y generándose así una inconsistencia de datos [RUFI98].

El escenario descrito puede causar dos tipos de averías. Si el transmisor detecta el error y procede a la retransmisión de la trama afectada, los receptores que aceptaron la trama, la volverán a recibir por segunda vez. Esta avería se denomina duplicado de mensaje inconsistente (IMD, del inglés *Inconsistent Message Duplicate*). En cambio, si el transmisor falla y no es capaz de retransmitir la trama errónea, los nodos que rechazaron la trama jamás la recibirán. Esta avería se conoce como omisión de mensaje inconsistente (IMO, del inglés *Inconsistent Message Omission*) y es la que se produce en la Figura 3.10.

Se debe resaltar que desde el punto de vista de la garantía de funcionamiento, una omisión es mucha más crítica que un duplicado. En [RUFI98] únicamente se tuvo en cuenta una posible avería del transmisor como motivo de un IMO. Sin embargo, en [PROE00] se identificó que la presencia de múltiples errores de canal pueden causar un IMO sin que el transmisor falle. Este escenario presenta incluso una mayor probabilidad que el ilustrado en la Figura 3.10.

En conclusión, la ocurrencia de IMOs generalmente viene ligada a uno de los

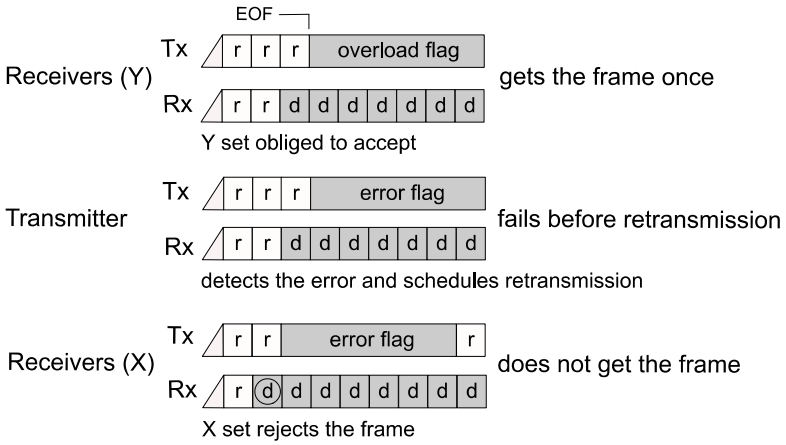



Figura 3.10: Escenario de inconsistencia

siguientes motivos [PROE00] [RODR03]:

- El transmisor se avería (crash) antes de poder retransmitir la trama corrompida.
- Un segundo error en el canal enmascara el *error flag* transmitido por el subconjunto de nodos que han detectado el error inicial por lo que el transmisor no detecta ningún error.
- El tiempo disponible para la retransmisión no es suficiente. Esta limitación es debida al uso de técnicas de contención de errores en el dominio temporal y únicamente está presente en las variantes de tiempo real de CAN como por ejemplo TTCAN, TCAN o FTT-CAN.

### Nuevos escenarios de error

Durante mucho tiempo se pensó que las inconsistencias únicamente eran causadas por errores en los últimos bits de una trama, es decir, en el campo EOF. No obstante, recientemente se identificaron escenarios en los que el razonamiento anterior no es correcto [RODR11].

Para encontrar los escenarios mencionados se hizo uso de la herramienta CANfidant. A modo de ilustración, en la Figura 3.11 se muestra uno de los escenarios de inconsistencia encontrados. El él, el nodo 0 es el transmisor ($ID = 0$ y $Data = 162Dec$)

y los nodos 1 y 2 son receptores. El nodo 1 detecta un error local en el CRC de la trama recibida (en concreto un *stuff error*) e inyecta, inmediatamente después, un *error flag*. Los nodos 0 y 2 sufren un segundo error que les impide detectar el *error flag* de manera que aceptan la trama.

Aunque el escenario mostrado sea solo uno de muchos (es imposible discutirlos todos), se pueden definir dos condiciones como causas primarias de este tipo de inconsistencias: el primer error detectado debe ser un *stuff error* o un *fomat error* y, los últimos bits del CRC deben ser varios bits dominantes consecutivos de forma que la mayoría de bits del *error flag* sean NPB. Así, un único error adicional puede afectar al único EPB del EF y enmascararlo. Este es el caso de la Figura 3.11.

Con esto se suma una tercera fuente de inconsistencias a las ya existentes: la señalización inconsistente de errores.


Figura 3.11: Escenario de inconsistencia con dos errores

### 3.3.2. Conclusiones

En este capítulo se han expuesto las limitaciones del protocolo CAN, poniendo especial interés en la consistencia de datos.

Aunque CAN, en su versión estándar, no sea apto para su uso en aplicaciones con elevada garantía de funcionamiento, mediante la inclusión de mecanismos adi-

cionales, sí puede llegar a serlo. El proyecto CANbids tiene por misión aumentar la garantía de funcionamiento de CAN y, para ello, presenta diferentes soluciones a las limitaciones expuestas.

## 3.4. Soluciones propuestas dentro de CANbids

Al inicio de este capítulo ya se enumeraron las diferentes soluciones propuestas dentro del proyecto CANbids. En este apartado se describen, en mayor o menor medida, aquellas soluciones que han tenido un papel importante en el desarrollo del presente trabajo final de Máster.

### 3.4.1. CANcentrate y ReCANcentrate

El uso de buses de campo en sistemas de control distribuidos es habitual debido a su excelente robustez electromagnética y bajo coste. Sin embargo, la topología bus presenta ciertas limitaciones en cuanto a la garantía de funcionamiento, especialmente en lo que se refiere a la contención de errores. Así, un fallo independiente en algún componente (p.e. el controlador, el *transceiver*, la conexión, el medio, etc.) del sistema puede generar errores que se propaguen por la red y resulten en una avería general. Esto es debido a que la topología bus presenta varios puntos de avería únicos (*single points of failure*) (Figura 3.12).

Usando buses replicados y/o *bus-guardians* (técnicas de tolerancia a fallos) tampoco se ve mejorada notablemente la contención de errores. El problema son las denominadas averías de modo común (*common-mode failures*), averías que afectan de forma simultánea a varios componentes del sistema. Así, por ejemplo, un nodo averiado puede transmitir información errónea a ambos buses replicados. Otra razón puede ser la proximidad física de las réplicas (una interferencia electromagnética, por ejemplo, puede afectar a ambos buses si éstos se encuentran cerca). Estas averías se denominan averías de modo común por proximidad espacial (*common-mode spatial proximity failures*). Por otro lado, los *bus-guardians* son ineficaces con respecto a fallos generados por el propio medio de transmisión. Además, resulta usual que éstos compartan componentes con los nodos tales como el oscilador o la fuente de alimentación. Consecuentemente, un fallo en estos elementos afectará tanto al nodo

como a su *bus-guardian*.

Las topologías estrella son una solución más efectiva ya que reducen el número de puntos de avería únicos. En una topología en estrella simple, cada nodo está conectado a un elemento central, el concentrador (*hub*), por una conexión independiente (*link*). De esta forma, las conexiones únicamente entran en proximidad espacial en el centro de la estrella, reduciéndose significativamente la probabilidad de las averías de modo común. La ventaja principal de esta topología es, sin embargo, la visión privilegiada que tiene el *hub* del sistema. Diseñado correctamente, éste conocerá en todo momento el estado actual de todos los nodos. Posibles inconvenientes son el aumento de cableado y la existencia de un punto de avería único, el propio *hub*. De todos modos, el coste del cableado depende de la aplicación y la probabilidad de fallo del concentrador puede ser reducido mediante técnicas de diseño especiales o, haciendo uso de una estrella replicada (*replicated star*).

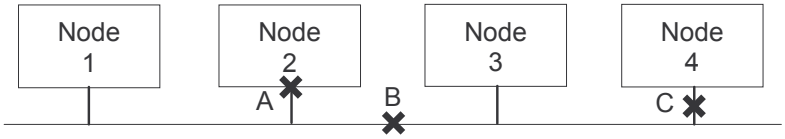



Figura 3.12: Ejemplo de errores que pueden propagarse por el bus

Muchos protocolos de comunicación (Ethernet, TTP [STO03], FlexRay [Flex05], etc.) han adoptado la topología estrella por su elevada robustez. CAN, probablemente el protocolo más ampliamente usado en sistemas distribuidos, jamás llegó a dar ese salto. Al mantener la topología bus, los mecanismos de contención de errores de CAN son poco eficientes (hasta en el caso de usar buses replicados [RUFI99] [RUSH03] y *bus-guardians* [FER05]).

En los últimos años se han ido proponiendo diferentes topologias en estrella para CAN, algunas basadas en estrellas pasivas (*passive stars*) [CiAb] y otras en estrellas activas (*active stars*) [RUC94] [IXX05] [CEN01] [SAH06]. Sin embargo, estas propuestas presentaban importantes desventajas tales como limitada contención de errores o problemas de compatibilidad. La no existencia de una propuesta realmente convincente fue la motivación para diseñar dos topologías estrella basadas en CAN: CANcentrate [BARR06a] y ReCANcentrate [BARR06b].

En CANcentrate, cada nodo está conectado al *hub* mediante una conexión que contiene un *uplink* y un *downlink* (Figura 3.13). El *hub* recibe las contribuciones de los diferentes nodos por los *uplinks*, acopla las señales y transmite el resultado, en modo difusión, por los *downlinks*.

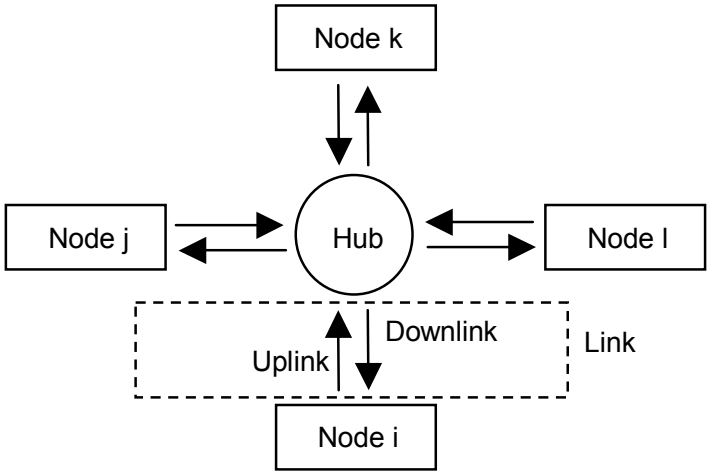



Figura 3.13: Esquema de conexiones de CANcentrate

En la figura 3.14 se muestra la arquitectura interna del *hub* [BARR06a]. Contiene tres módulos principales: el módulo de acoplamiento (*Coupler Module*), el módulo de entrada/salida (*Input/Output Module*) y el módulo de tratamiento de fallos (*Fault Treatment Module*). El módulo de acoplamiento acopla las contribuciones ($B_{1...N}$) mediante una *AND* lógica, generando la señal resultante $B_0$. Seguidamente devuelve el resultado a los nodos. La puerta *AND* realiza la función de la *AND cableada* de un bus CAN por lo que las tramas obtenidas de $B_0$ de ahora en adelante se denominarán tramas resultantes (*resultant frames*).

El módulo de entrada/salida contiene los *transceivers* que transforman las señales físicas recibidas por los *uplinks* en valores lógicos y, por otro lado, los *transceivers* que transforman la señal lógica $B_0$ en señales físicas transmitidas por los *downlinks*.

El módulo de tratamiento de fallos contiene el submódulo Rx_CAN y un conjunto de unidades de habilitación/deshabilitación (*Enabling/Disabling Units*). El módulo Rx_CAN observa la señal acoplada $B_0$ para sincronizarse con la trama resultante y generar información sobre el estado actual (*Current State*) de la trama resultante (campo de la trama, tipo de bit, etc.). Las unidades de habilitación/deshabilitación,

una por cada nodo, usan el estado actual de la trama resultante (C en la Figura 3.14) y la información sobre el puerto supervisado (si el nodo conectado es transmisor o receptor) para estimar la siguiente contribución de aquel puerto. Si la contribución estimada y la recibida discrepan, se considera una cierta condición de error y se incrementa el contador de error asociado. Los contadores serán decrementados si ha pasado un tiempo de operación predefinido libre de errores. Si un contador alcanza un umbral prefijado, el puerto correspondiente será aislado por la unidad de habilitación/deshabilitación. Para ello se conduce la señal $ED_{1...N}$, puesta a *'1'*, a la entrada de una puerta *OR* en el módulo de acoplamiento. Así, la contribución del puerto siempre valdrá *'1'* (recesivo), cosa que equivale a desconectar dicho puerto.

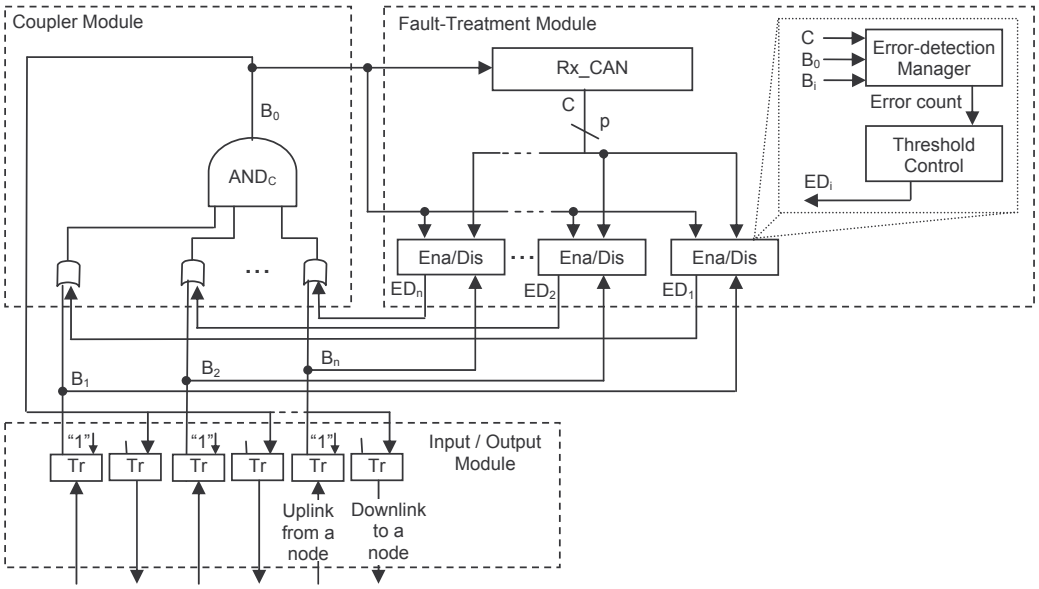



Figura 3.14: Estructura interna de CANcentrate

Al tener cada nodo un *uplink* y *downlink* independientes, el *hub* es capaz de monitorizar y detectar fallos con mayor precisión que los contadores de error de CAN [ISO93]. Además, al realizarse el acoplamiento dentro de un tiempo de bit, el *hub* es transparente para los nodos y, por lo tanto, totalmente compatible con CAN. La única pequeña modificación necesaria es el uso de dos *transceivers* por cada nodo (Figura 3.15) debido a la separación de *uplink* y *downlink* [BARR06a]. De todos modos, esto no presenta un problema grave ya que se pueden utilizar *transceivers* COTS (*Commercial Off-The-Shelf*) para su implementación.

CANcentrate introduce elevada robustez y contención de errores pero no propor-

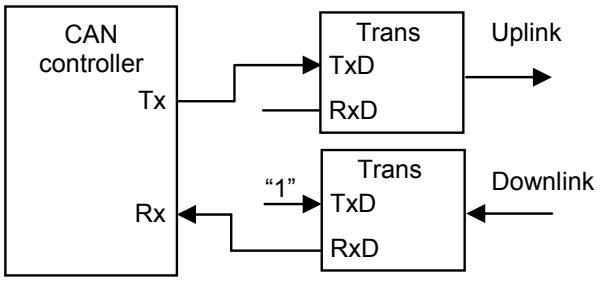



Figura 3.15: Dos *transceivers* por nodo: uno para el *uplink* y otro para el *downlink*

ciona mecanismo de tolerancia a fallos. ReCANcentrate [BARR06b], sin embargo, sí puede aumentar la fiabilidad del sistema al usar una topología en estrella replicada.

La conexión de los nodos es similar a CANcentrate: cada nodo se conecta a los *hubs* mediante un *uplink* y un *downlink* respectivamente (Figura 3.16). Los nodos reciben los mismos datos, bit por bit, y de manera paralela por ambas estrellas. Esto se garantiza, de forma totalmente transparente, acoplando los dos *hubs* mediante dos conexiones dedicadas (*interlinks*), cada una con dos conexiones (*sublinks*) (Figura 3.16). El uso de dos *interlinks* es un simple mecanismo de tolerancia a fallos. Gracias al acoplamiento entre *hubs*, el tráfico será idéntico para éstos y, por lo tanto, las tramas serán transmitidas por los nodos a un *hub* o al otro pero nunca a ambos.

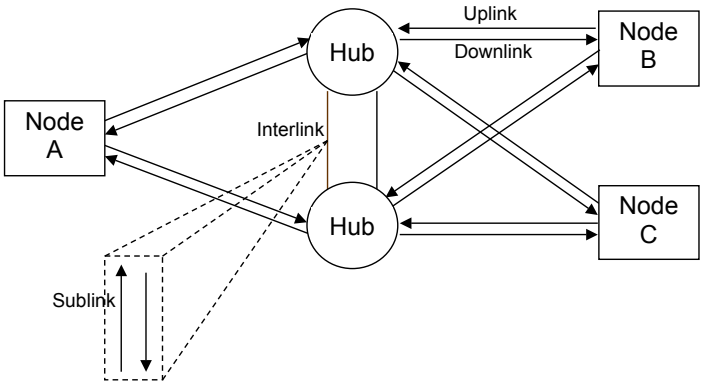



Figura 3.16: Esquema de conexión de ReCANcentrate

Internamente, un *hub* ReCANcentrate es muy similar al de CANcentrate (Figura 3.17). En el módulo de acoplamiento se añade una segunda puerta *AND*. Dicha puerta lógica acopla las contribuciones de los dos *hubs* (la propia contribución $B_0$

y las contribuciones replicadas del otro *hub* $B'_{00}$ y $B'_{01}$, recibidas por el *interlink*) y transmite el resultado a los nodos directamente conectados. A su vez se generan dos replicas de $B_0$, $B_{00}$ y $B_{01}$, que son transmitidas por el *interlink* al segundo *hub*.

Este acoplamiento permite que los *hubs* se monitoricen el uno al otro, aislándose uno de ellos en caso de detectarse errores. El monitorizado y el aislamiento se realiza en las unidades de habilitación/deshabilitación del *hub* (*Hub Enabling/Disabling Units*).

ReCANcentrate crea regiones de contención errores, incluye mecanismos de tolerancia a fallos que eliminan cualquier punto de avería único e impone una visión consistente de la red. Finalmente, el problema de la sincronización de canales replicados (*replicated channel synchronization*) es solventado gracias a la sincronización a nivel de bit de los *hubs*.

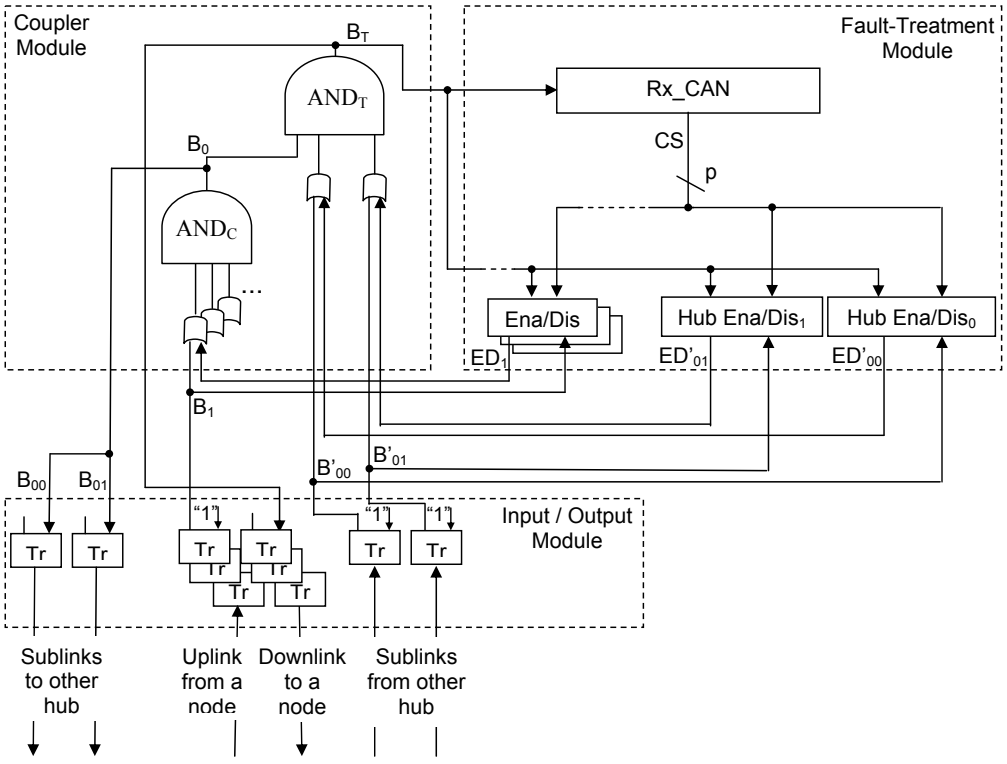



Figura 3.17: Arquitectura interna de un *hub* ReCANcentrate

A modo de conclusión, decir que el uso de topologías estrella es altamente beneficioso ya que incrementa la robustez y fiabilidad de un sistema de comunicaciones.

Las propuestas presentadas, la estrella simple CANcentrate y la estrella replicada ReCANcentrate, presentan dos niveles de garantía de funcionamiento, siendo la primera una buena opción para aplicaciones generales y la segunda para aplicaciones con elevada garantía de funcionamiento. Ambas arquitecturas son completamente compatibles con el protocolo CAN y son implementables con componentes COTS, es decir, su coste es reducido. La implementación física de CANsistant y AEFT para topologías estrella se realizó sobre ReCANcentrate por lo que será referenciado a menudo en los capítulos 5 y 6.

### 3.4.2. sfiCAN

En el capítulo *Propiedades del protocolo CAN* se ha visto que CAN incluye varios mecanismos que lo hacen robusto a todo tipo de fallos pero, aún así, éstos pueden generarse y causar consecuencias impredecibles en la red. Esto hace necesario un estudio preciso de la respuesta del sistema ante la presencia de fallos. Para ello, dentro del proyecto CANbids, se propuso el diseño de un inyector de fallos para CAN: sfiCAN.

sfiCAN (*Star-based physical Fault Inyector for CAN*) es un inyector de fallos a nivel físico para CAN. Está basado en la topología estrella de CANcentrate cosa que lo hace transparente desde el punto de vista de los nodos. En la figura 3.18 se muestra su arquitectura básica. El *hub* incluye tres módulos: el módulo de acoplamiento, el módulo CAN y el módulo de inyección de fallos. El módulo de acoplamiento implementa la función *AND* de las señales recibidas por los *uplinks* y envía el resultado por los *downlinks*. A nivel lógico se implementa un bus CAN pero con la ventaja de poder inyectar una gran variedad de escenarios de error complejos. El módulo CAN monitoriza la señal acoplada e indica en que posición de la trama se encuentra el bit que es transmitido en ese momento. Finalmente, el módulo de inyección de fallos se ocupa de inyectar los fallos propiamente dichos.

La configuración del inyector de fallos se realiza mediante un PC. Esto es posible ya que el módulo fue diseñado como un NCC (*Network Configurable Component*), un componente que, conectado a una red, recibe su configuración por esa misma red. Para su conexión con el PC, el *hub* incorpora un puerto dedicado formado por un controlador CAN simple (al no inyectarse fallos en la conexión *hub* - PC, no es necesario disponer de un *uplink* y un *downlink* separados).

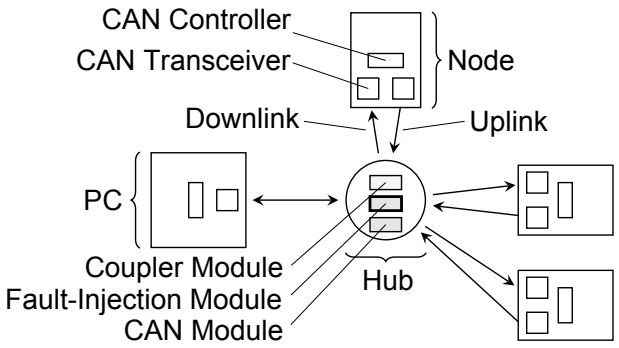



Figura 3.18: Arquitectura de sfiCAN

Como todo NCC, el módulo de inyección de fallos presenta cuatro modos de operación diferentes: el modo de configuración (*configuration mode*), el modo de espera (*idle mode*), el modo denominado *wait-for-whistle* y el modo de ejecución (*execution mode*). Si el PC transmite un mensaje de modo de configuración (*configuration-mode frame*), el NCC entra en modo de configuración. Durante la configuración, el PC transmite comandos de configuración (*configuration commands*) codificados en tramas CAN estándar. Finalizada la configuración, el PC envía un mensaje de entrada en espera (*enter-idle-mode*) o un mensaje de *wait-for-whistle*. En modo espera, el NCC únicamente acepta mensajes de modo de configuración. En modo *wait-for-whistle*, aparte de los mensajes de modo de configuración, también se aceptan mensajes del tipo *starting-whistle*, caso en el que se entrará en modo de ejecución y se empezará a inyectar fallos acorde con su configuración. El primer byte de datos de los mensajes indica el cambio de modo que se debe efectuar. En la tabla 3.1 se exponen todos los mensajes de cambio de modo y su código correspondiente.

| Mensaje de cambio de modo | Código |
|---|---|
| *enter-config-mode* | 20 |
| *enter-idle-mode* | 21 |
| *enter-wfw-mode* | 22 |
| *starting-whistle* | 23 |

Cuadro 3.1: Mensajes de cambio de modo.

La configuración del inyector de fallos viene definida en un fichero de texto denominado *fault-injection specification* (ver la figura 3.19). Esta especificación es defini-

da según la notación de Backus-Naur (BNF, del inglés *Backus-Naur Form*) usando la sintaxis normalizada en ISO/IEC 14977 [ISO96e]. Incluye diferentes parámetros de configuración que indican qué tipo de fallo se debe inyectar, dónde inyectarlo y cuándo. Para más información sobre los diferentes parámetros es recomendable consultar [BALL11] y [GESS11].

```
specification = { '[', string, ']', fi_config }
fi_config = value, target_link, mode, aim, fire,
       cease, [withdraw] ;
value = 'value_type', '=', value_type_value,
       ['value_bfvalue', '=', {'0'|'1'} ] ;
target_link = 'target_link', '=', link - 'coupled' ;
aim = 'aim_count', '=', natural,
    'aim_filter', '=', filter_value,
    'aim_field', '=', field_value,
    'aim_link', '=', link,
   ['aim_role', '=', role_value ] ;
fire = 'fire_field', '=', field_value,
       'fire_bit', '=', natural ,
       'fire_offset', '=', natural ;
cease = 'cease_bc', '=', natural
    | 'cease_field', '=', field_value,
      'cease_bit', '=', natural ;
withdraw = 'withdraw_count', '=', natural,
    'withdraw_filter', '=', filter_value,
    'withdraw_field', '=', field_value,
    'withdraw_link', '=', link,
   ['withdraw_role', '=', role_value ] ;
mode = 'continuous' | 'iterative' | 'selective' ;
value_type_value = 'stuckDominant' | 'stuckRecessive'
     | 'bitFlip' | 'inverse' ;
filter_value = ('0'|'1'|'x'), {'0'|'1'|'x'} ;
link = 'port0up' | 'port0dw' | 'port1up'
    | 'port1dw' | 'port2up' | 'port2dw'
    | 'port3up' | 'port3dw' | 'coupled' ;
field_value = 'idle' | 'id' | 'rtr' | 'res'
    | 'dlc' | 'data' | 'crc' | 'crcdelim'
    | 'ack' | 'ackdelim' | 'eof'
    | 'interfield' | 'errflag' | 'errdelim' ;
role_value = 'dont_care' | 'tr' | 're' ;
```

Figura 3.19: *Fault-injection specification*

El módulo de inyección de fallos se compone por los siguientes submódulos: (1) el extractor (*ID/data extractor*) extrae el identificador y el campo de datos de la trama en estado de transmisión. (2) El filtro (*NCC ID filter*) comprueba si el identificador de la trama se corresponde con el identificador del NCC. (3) El indicador de modo (*mode indicator*) comprueba si el campo de datos indica un cambio de modo y, en caso afirmativo, lo señaliza al resto de submódulos. (4) El intérprete de comandos (*command interpreter*) comprueba, durante el modo de configuración, si los datos recibidos incluyen parámetros de configuración en lugar de cambios de modo. A partir del conjunto de parámetros de configuración recibidos, construye una configuración de inyección de fallos. (5) El depósito de configuraciones (*configuration storage*) almacena todas las configuraciones de inyección de fallos. (6) Finalmente, el ejecutor

(*executer*) contiene varios submódulos programables (*programable configuration executers*), programados cada uno acorde con las configuraciones almacenadas. Estos módulos se encargan de inyectar los diferentes fallos al sistema durante el modo de ejecución. En la figura 3.20 se muestra la relación entre los diferentes submódulos.

La figura 3.21 muestra un ejemplo de inyección de fallos, produciéndose una omisión de mensaje inconsistente (IMO). El primer error, un bit dominante, es inyectado en el *downlink* del nodo receptor X, en el penúltimo bit del EOF. El segundo error, un bit recesivo, es inyectado en el *downlink* del nodo transmisor, concretamente en el último bit del EOF. El escenario resultante es el expuesto en el capítulo 3.3.1: el nodo X rechaza la trama, el nodo Y la acepta y el nodo transmisor no retransmite.

Adicionalmente al inyector de fallos, sfiCAN incluye un módulo de registro (*logging module*) utilizado para recopilar datos relevantes durante el modo de ejecución del inyector. Una vez finalizada la ejecución, los datos obtenidos son enviados al PC para su análisis.

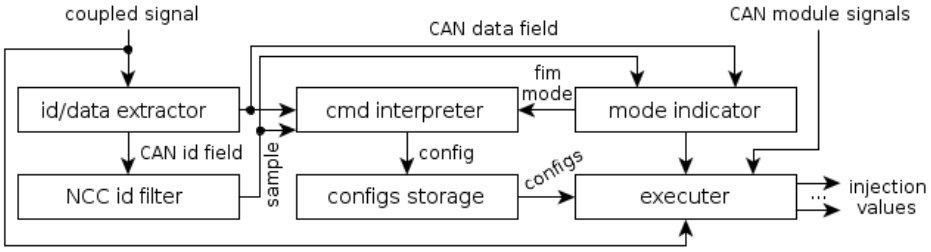



Figura 3.20: Diagrama de submódulos de sfiCAN

Para concluír este apartado hay que decir que sfiCAN es un inyector de fallos para CAN potente y versátil. En esta sección se han introducido los conceptos básicos de su funcionamiento. Para profundizar en los aspectos relacionados con su diseño, implementación y uso se recomienda consultar [BALL11] y [GESS11].

En el presente trabajo a menudo se hizo uso de sfiCAN, tanto del módulo de inyección de fallos como del módulo de registro. La creación de escenarios de error complejos fue esencial para la verificación experimental de las soluciones propuestas (ver capítulo 5.5).

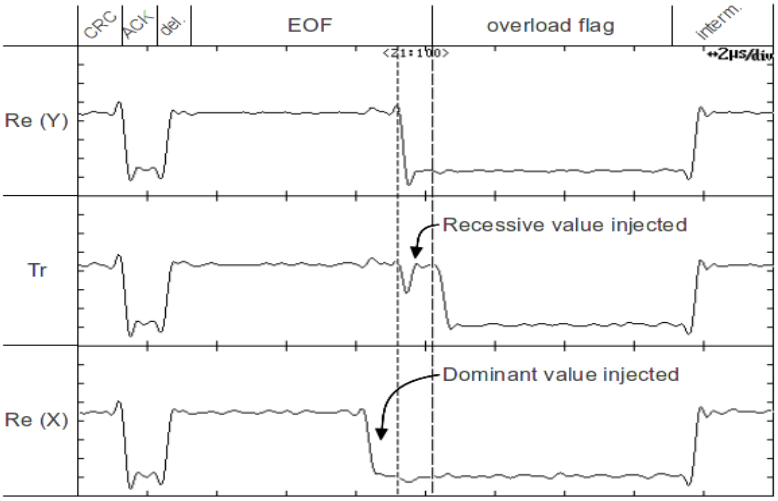



Figura 3.21: Ejemplo de inyección de fallos

### 3.4.3. CANsistant

El el capítulo 3.3.1 se explicó que la consistencia de datos de CAN es limitada. Se identificaron tres causas fundamentales: la existencia del estado de error pasivo, la regla del último bit del EOF y la señalización inconsistente de errores.

Para solventar la problemática del último bit del EOF e impedir la aparición de los escenarios de inconsistencia identificados por Rufino *et. al*, en [RUFI98] se propuso un protocolo de alto nivel denominado TOTCAN (ejecutado en la capa de aplicación del modelo OSI colapsado). Este protocolo requiere la transmisión de una trama de control por cada trama de datos.

Posteriormente, Livani [LIV99] presentó el SHAdow REtransmitter (SHARE): un mecanismo, conectado a la red como si de un nodo regular se tratara (Figura 3.22), capaz de detectar un patrón de bits específico que, según Rufino *et. al*, indica la posibilidad de una inconsistencia y, proceder a la retransmisión de la trama potencialmente inconsistente. Para ello, el nodo almacena de forma preventiva todas las tramas transmitidas por la red. Si detecta el patrón, el nodo retransmite la trama almacenada rigiéndose por el mecanismo de arbitraje de CAN. En caso de no detectar el patrón, la trama almacenada es descartada.

El patrón a detectar es una secuencia de seis bits dominantes empezando en el

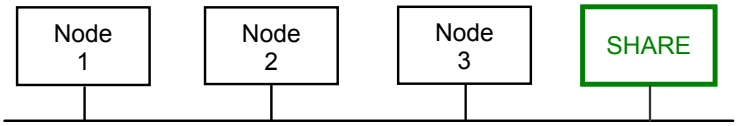



Figura 3.22: Conexión del SHARE

último bit del EOF. Este patrón de bits es característico del escenario descrito en 3.3.1 e ilustrado en la Figura 3.10, página 44.

En caso de que el transmisor original se averíe, SHARE se ocupará de la retransmisión y evitará un IMO. En cambio, si el transmisor sique funcionando, éste también retransmitirá la trama afectada cosa que puede llevar a problemas. Según [LIV99], la transmisión simultánea estará sincronizada bit a bit, evitando la producción de errores. Sin embargo, al final de la trama puede haber desviaciones del tiempo de bit que causen la detección de errores por parte de los receptores.

Adicionalmente, como se ha explicado en el capítulo 3.3.1, existen escenarios de múltiples errores en los que un segundo error puede afectar al transmisor de manera que éste no retransmita la trama corrompida. Desgraciadamente, el diseño de SHARE no tiene en cuenta dichos escenarios por lo que puede sufrir, al igual que el transmisor, de errores adicionales que le impidan detectar una omisión de mensaje inconsistente (Figura 3.23). Estas limitaciones llevaron a proponer, dentro del proyecto CANbids, un mecanismo capaz de detectar IMOs incluso en presencia de múltiples errores de canal: CANsistant (*CAN Assistant for Consistency*) [PROE09].

Aunque en un principio pensado para detectar escenarios de inconsistencia y retransmitir la trama afectada (esto es, resolver las inconsistencias), la propuesta final de CANsistant únicamente se centró en la detección dejando de lado la retransmisión. Esto es debido al creciente interés en deshabilitar la retransmisión automática de mensajes y usar el modo de transmisión única (*single-shot transmission*), modo natural en sistemas TDMA o FTDMA [ALM02]. En el contexto de CANbids, ésto puede ser útil para mejorar las prestaciones de tiempo real (eliminar el *jitter* causado por las retransmisiones). Sin embargo, la deshabilitación de las retransmisiones automáticas aumenta considerablemente la probabilidad de IMOs [RODR03].

Al estar el proyecto CANbids intencionado para su uso en aplicaciones con ele-

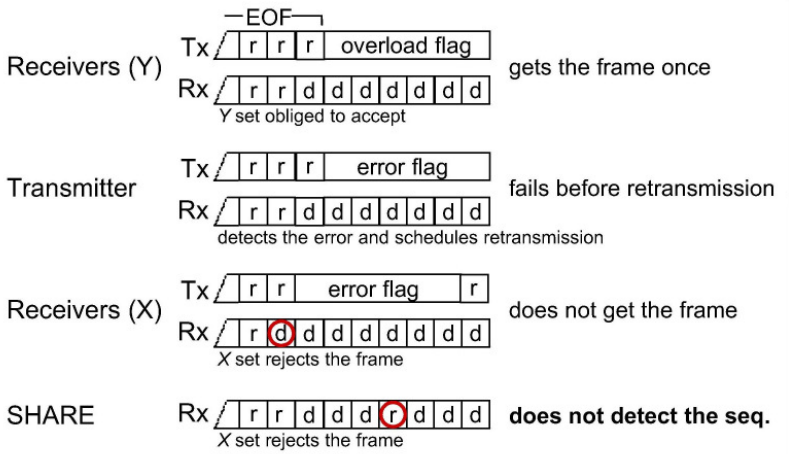



Figura 3.23: Escenario de inconsistencia no contemplado por SHARE

vada garantía de funcionamiento, la detección de escenarios de inconsistencia (sea por una avería del transmisor, errores múltiples en el canal o el modo de transmisión única), es una tarea de máxima prioridad, cosa que ponen en evidencia la necesidad de un mecanismo como CANsistant.

CANsistant (Figura 3.24) está diseñado, a diferencia de SHARE, para detectar escenarios de inconsistencia en presencia de múltiples errores de canal [PROE09]. Sin embargo, el número de errores tolerables está acotado y viene definido por el parámetro $m$. Para determinar el valor adecuado de $m$ se debe tener en cuenta que, en la ausencia de errores, después del campo EOF se transmite el *IFS* (tres bits recesivos) y, seguidamente, el bus queda en estado *Idle*. En ese momento puede comenzar la transmisión de una nueva trama, enviándose el *SOF* (un bit dominante) seguido por el identificador de la trama [ISO93]

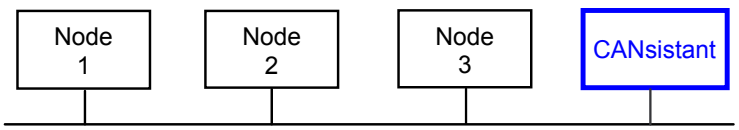



Figura 3.24: Conexión de CANsistant

Adicionalmente, supóngase el escenario ilustrado en la Figura 3.25. Los receptores del grupo X detectan un valor dominante en el penúltimo bit del EOF mientras CANsistant no lo detecta. Seguidamente, CANsistant sufre una serie de errores con-

secutivos $m-1$ que le impiden recibir los primeros $m-1$ bits dominantes del *error flag* transmitido por los nodos X. El valor de $m-1$ debe ser tal, que a partir del primer bit dominante detectado, CANsistant pueda saber si éste se correponde con un error flag o con la operación normal del protocolo en ausencia de errores. Ese valor umbral es $m-1=4$, caso en el que el primer bit dominante detectado se corresponde con el primer bit de la próxima trama, es decir, el SOF (que, como se ha indicado anteriomente, es de valor dominante). Entonces, con $m=5$, siempre que se inicie una nueva trama directamente después del IFS y esta trama presente un identificador con varios valores dominantes consecutivos en sus bits más significativos, se producirá una denominada falsa alarma, es decir, CANsistant detectará un IMO sin que se haya producido realmente. El caso anterior se produce en ausencia de errores por lo que el número de falsas alarmas puede ser elevado. Para evitar dicha situación pero garantizar la detección de todos los IMOs, se optó por el valor $m=4$.

Como se ha explicado en el párrafo anterior, CANsistant es capaz de detectar IMOs hasta en el caso de no recibir el primer bit dominante en el último bit del EOF. No obstante, después de detectarse el primer bit dominante aún pueden generarse errores adicionales, alterando la llegada de los bits dominantes consecutivos. Para solventar este problema, CANsistant incluye tres parámetros:

1. Ventana de detección del primer dominante (FDDW, del inglés *First Dominant Detection Window*): esta ventana se corresponde con un grupo de bits de la trama en el que CANsistant debe encontrar el primer bit dominante para considerar una posible situación de inconsistencia. Si no se detecta ningún dominante en el FDDW, se supone que la ocurrencia de un IMO es imposible para esa trama. A partir del escenario de la Figura 3.25 se pueden determinar el inicio y el fin de la ventana. El primer bit del FDDW es el último bit del EOF. Considerando que $m=4$, el último bit del FDDW es el último bit del IFS, bit hasta el que se puede retrasar la detección del primer dominante en caso de haberse producido $m-1=3$ errores adicionales.

2. Número de bits dominates (ND, *Number of Dominant bits*): el número de bits dominates que CANsistant debe encontrar después de detectar el primer bit dominante en el FDDW para señalizar una inconsistencia potencial. De nuevo, en base a la Figura 3.25 se puede determinar el valor de ND. En el peor

de los casos, CANsistant ha sufrido el máximo número de errores adicionales ($m - 1 = 3$) y ha detectado el primer dominante en el último bit del IFS. Al haber sufrido ya el máximo número de errores, los tres bits siguientes serán dominantes por lo que $ND = 3$. Con $ND = 4$ el IMO no se detectaría y con $ND = 2$ se generarían demasiadas falsas alarmas.

3. Ventana adicional de detección de dominantes (ADDW, *Additional Dominant Detection Window*): ventana que se corresponde con el grupo de bits que empieza después del primer bit dominante detectado en el FDDW. Si y solo si CANsistant encuentra ND bits dominantes en el ADDW, señalizará la posible inconsistencia. El inicio de la ventana es variable y depende de dónde se ha detectado el primer dominante. El último bit del ADDW, en cambio, es fijo y se corresponde con el tercer bit después del IFS (obtenido a partir del peor caso ya descrito en el punto 2). Aumentar el tamaño del ADDW únicamente causaría un mayor número de falsas alarmas.

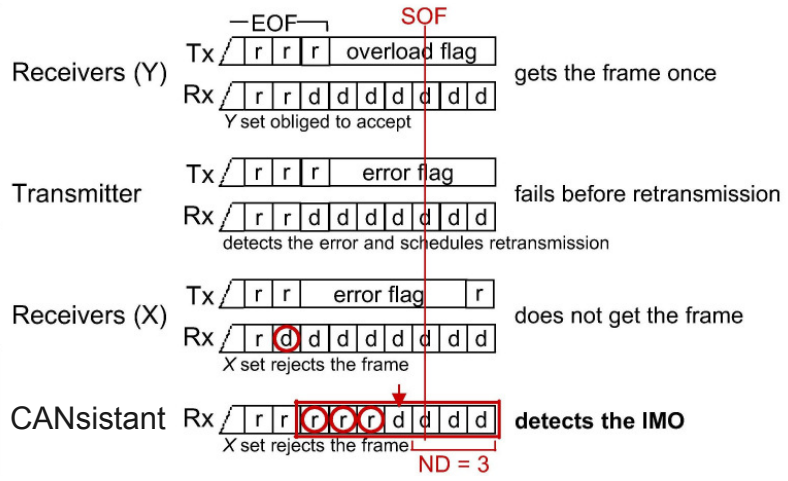



Figura 3.25: IMO detectado por CANsistant

Para terminar con este apartado se puede decir que CANsistant es un mecanismo capaz de detectar omisiones de mensaje inconsistentes incluso en presencia de hasta $m$ bits erróneos en el canal. Se ha tomado un valor de $m = 4$ de manera que las falsas alarmas se mantienen en un nivel aceptable [PROE09].

El diseño propuesto se basa en la topología bus de CAN. Sin embargo, parte del trabajo presentado en esta memoria ha sido el diseño e implementación de

CANsistant para topologías estrella (ver el capítulo 6). Para ello se integraron las funcionalidades de CANsistant en ReCANcentrate. Es especialmente importante que los *hubs* puedan detectar todos los IMOs posibles ya que en ReCANcentrate la retransmsión automática de mensajes está desactivada. Adicionalmente, CANsistant puede ser rediseñado para reducir el número de falsas alarmas .

### 3.4.4. Señalización de errores consistente

Como se ha visto en el capítulo 3.3.1, existen escenarios específicos en los que la señalización de errores es inconsistente. A modo de recordatorio, hasta hace poco se pensó que las inconsistencias únicamente eran causadas por errores en los últimos bits de una trama, es decir, en el campo EOF. No obstante, en [RODR11] se identificaron escenarios en los que el razonamiento anterior no es correcto. Por esta razón y para garantizar la total consistencia de datos en CAN, se propone una solución al problema: una estrategia basada en la transmisión de los denominados AEFs (*Aggregated Error Flags*) [RODR11].

La fuente de los nuevos escenarios de inconsistencia se encuentra en los señalizadores de error definidos por el protocolo CAN [ISO93] ya que pueden resultar demasiado cortos en presencia de múltiples errores de canal. La estrategia propuesta se basa en alargar los *error flags* y así garantizar la globalización de errores aún en el caso de haber múltiples bits afectados por errores. Sin embargo, el alargamiento conlleva un problema ya que puede incrementar de forma significativa los contadores de errores (TEC y REC) de los controladores CAN y, en el peor de los casos, llevarlos al estado *bus-off*. Para evitar esta situación se planteó, en lugar de simplemente alargar los *error flags*, agregar varios *error flags* consecutivos de formato estándar (seis bits dominantes), separados entre ellos por bits recesivos.

En la Figura 3.26 se ejemplifica el uso de AEFs para la señalización consistente de errores. Se muestran las señales $Tx$ y $Rx$ de tres controladores CAN y la contribución de los AEFs. En este caso se garantiza la consistencia a pesar de la existencia de tres errores de canal (marcados por círculos).

La estrategia sugerida es totalmente compatible con CANsistant por lo que, usando conjuntamente las dos propuesta, se consiguen eliminar dos de las causas de inconsistencias en CAN, la señalización inconsistente de errores y la regla del

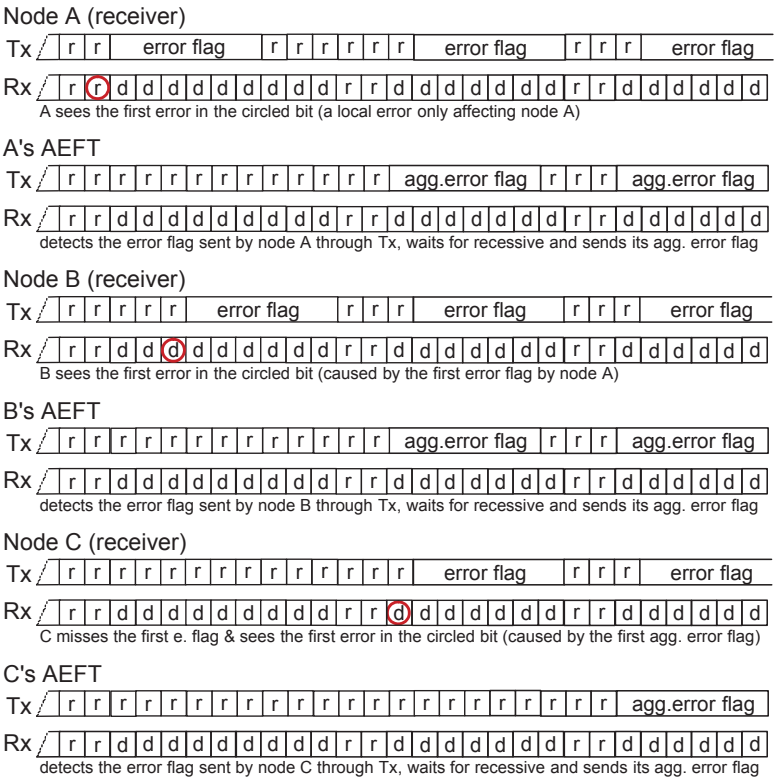



Figura 3.26: Ejemplo de señalización mediante AEFs

último bit del EOF. Prohibiendo el estado de error pasivo se elimina la tercera y última fuente de inconsistecias y se resuelve por completo el problema de la limitada consistencia de datos de CAN.

El diseño e implementación de un mecanismo que implemente la solución presentada ha sido una parte fundamental del proyecto final de Máster expuesto en estas páginas (ver el capítulo 5).

### 3.4.5. Conclusiones

Las soluciones descritas son de diferente naturaleza pero todas ellas tienen un claro objetivo: superar las limitaciones intrínsecas del protocolo CAN y abrirle el camino para su futuro uso en aplicaciones con elevada garantía de funcionamiento. Así, las propuestas CANcentrate, ReCANcentrate y sfiCAN ya se encuentran en estados avanzados de desarrollo mientras CANsistant y AEF aún están en una fase

inicial.

En el apartado siguiente, *Tareas pendientes*, se exponen las tareas realizadas durante el transcurso de este trabajo final de Máster.

## 3.5. Tareas pendientes

En apartados anteriores de este capítulo se ha hablado del protocolo CAN, de sus limitaciones y de las soluciones que se han propuesto dentro del proyecto CANbids hasta el momento. En este último apartado se expone el trabajo realizado por el alumno y la planificación de tareas.

Entre las tareas realizadas se encuentran el estudio de las características principales de los elementos constitutivos de la arquitectura CANbids (CANcentrate, ReCANcentrate, sfiCAN, etc.), el estudio de la relevancia de nuevos escenarios de inconsistencia identificados, basados en la señalización inconsistente de errores; la propuesta del AEFT para resolver estos nuevos escenarios tanto para topologías bus como para topologías estrella, la mejora y el rediseño de CANsistant para reducir el número de falsas alarmas y para su uso en topologías estrella y la propuesta de las modificaciones necesarias para poder integrar correctamente los nuevos elementos con los ya existentes.

Las tareas realizadas son, efectivamente, bastante variadas. Esto es debido a que el trabajo, aún presentando un eje temático común, CANbids, no se centra en una única solución del mencionado proyecto sino que engloba muchas de ellas.

En el siguiente listado se resumen las contribuciones de mayor importancia:

- **Estudio de relevancia:** la señalización inconsistente de errores es una causa de inconsistencias identificada recientemente y, por lo tanto, poco estudiada. Por este motivo, en el capítulo 4 se determinará si estos nuevos escenarios de inconsistencia tienen una frecuencia de aparición elevada. En caso afirmativo, éstos deberán ser resueltos para seguir garantizando la fiabilidad del sistema.
- **Diseño e implementación del AEFT:** en el capítulo 3.3 se resaltaron las limitaciones de CAN, poniendo especial atención en la limitada consistencia

de datos. En ese mismo contexto, se describió con detalle la tercera de las tres causas básicas de inconsistencias: la señalización inconsistente de errores. Para solucionar este problema, en el capítulo 3.4 se describió una estrategia basada en la agregación de señalizadores de error denominada AEF (*Aggregated Error Flag*).

En el capítulo 5 se propondrá y desarrollará el *Aggregated Error Flag Transmitter* (AEFT), dispositivo que detecta la potencial ocurrencia de inconsistencias e inyecta un AEF en dicho caso. El módulo será diseñado de forma ortogonal y totalmente compatible con el protocolo CAN. Se describirán una versión para topologías bus y otra para topologías estrella, remarcando en todo momento las ventajas e inconvenientes que presenta cada versión.

- **Diseño e implementación de CANsistant:** en el capítulo 3.4.3 se introdujo CANsistant [PROE09], un mecanismo para detección de los escenarios de inconsistencia debidos a la regla del último bit del EOF, identificados por Rufino [RUFI98] y Proenza [PROE00].

  En el capítulo 6, esta solución será tomada como punto de partida para el diseño, la implementación y la simulación de un dispositivo tanto para topologías bus como para topologías estrella. Concretamente, se propondrán 3 versiones de CANsistant: una primera para topologías bus, basada en la propuesta inicial; una segunda, también para topologías bus pero diseñada para reducir el número de falsas alarmas y; finalmente, una tercera versión para topologías estrella basada en la integración de CANsistant con ReCANcentrate.

EL objetivo final de este trabajo ha sido la integración eficiente de las diferentes soluciones del proyecto CANbids (ReCANcentrate, sfiCAN, AEFT y CANsistant).

### 3.5.1. Planificación y tareas realizadas

La planificación del trabajo de final de Máster incluyó diferentes tareas. En la tabla 3.2 aparece un listado de las tareas realizadas, su fecha de inicio y su duración. Adicionalmente, en la tabla 3.3 se muestra el diagrama de *Gannt* correspondiente a la planificación del desarrollo del proyecto. En él se muestra el tiempo de dedicación previsto para las distintas actividades a lo largo del tiempo total (aproximadamente un año).

| ID | Tarea | Fecha inicio | Duración |
|---|---|---|---|
| T1 | Preparación: lectura de diversos artículos y estudios | 04.04.11 | 4 semanas |
| T2 | Migración controlador CAN | 02.05.11 | 2 semanas |
| T3 | Diseño AEFT: Máquinas de estado | 16.05.11 | 1 semana |
| T4 | Diseño AEFT: Diagrama de bloques | 23.05.11 | 1 semana |
| T5 | Implementación AEFT para topología bus | 30.05.11 | 2 semanas |
| T6 | Integración Controlador CAN y AEFT | 13.06.11 | 2 semanas |
| T7 | Preparación artículo ETFA | 06.06.11 | 3 semanas |
| T8 | Verificación experimental Controlador CAN | 27.06.11 | 1 semana |
| T9 | Verificación experimental AEFT topología bus (conf. 1) | 04.07.11 | 2 semanas |
| T10 | Verificación experimental AEFT topología bus (conf. 2) | 18.07.11 | 2 semanas |
| T11 | Estudio de relevancia: preparación | 15.09.11 | 1 semana |
| T12 | Estudio de relevancia: CANfidant | 22.09.11 | 1 semana |
| T13 | Estudio de relevancia: Matlab | 29.09.11 | 2 semanas |
| T14 | Implementación AEFT para topologías estrella | 10.10.11 | 2 semanas |
| T15 | Integración AEFT y ReCANcentrate | 24.10.11 | 1 semana |
| T16 | Integración AEFT y sfiCAN | 31.10.11 | 1 semana |
| T17 | Verificación experimental AEFT+ReCANcentrate (conf. 1) | 07.11.11 | 1 semana |
| T18 | Verificación experimental AEFT+ReCANcentrate (conf. 2) | 14.11.11 | 2 semanas |
| T19 | CANsistant: preparación | 05.12.11 | 1 semana |
| T20 | CANsistant: diseño para topologías bus | 12.12.11 | 2 semanas |
| T21 | CANsistant: implementación para topologías bus | 02.01.12 | 2 semanas |
| T22 | CANsistant: diseño para topologías estrella | 16.01.12 | 1 semana |
| T23 | CANsistant: implementación para topologías estrella | 23.01.12 | 1 semana |
| T24 | Redacción Memoria | 12.12.11 | 12 semanas |

Cuadro 3.2: Tareas realizadas

## 3.6. Conclusiones

En este capítulo se ha presentado el proyecto CANbids. En primer lugar se han descrito las propiedades relevantes, desde el punto de vista de la garantía de funcionamiento, del protocolo CAN. Seguidamente se han identificado las limitaciones por las que CAN resulta poco recomendable para su uso en aplicaciones con elevada garantía de funcionamiento.

Definidas las limitaciones, se ha hecho hincapié en las soluciones que integra CANbids para superarlas, especialmente en aquellas que se han utilizado en el presente trabajo.

En el último apartado se han expuesto las tareas pendientes, es decir, las labores a realizar por el autor de esta memoria. En los próximos capítulos se entrará en detalle con las aportaciones de mayor importancia: en el capítulo 4 se describe el estudio

| Tareas | Abr | May | Jun | Jul | Ago | Sept | Oct | Nov | Dic | Ene | Feb |
|---|---|---|---|---|---|---|---|---|---|---|---|
| T1 | | | | | | | | | | | |
| T2 | | | | | | | | | | | |
| T3 | | | | | | | | | | | |
| T4 | | | | | | | | | | | |
| T5 | | | | | | | | | | | |
| T6 | | | | | | | | | | | |
| T7 | | | | | | | | | | | |
| T8 | | | | | | | | | | | |
| T9 | | | | | | | | | | | |
| T10 | | | | | | | | | | | |
| T11 | | | | | | | | | | | |
| T12 | | | | | | | | | | | |
| T13 | | | | | | | | | | | |
| T14 | | | | | | | | | | | |
| T15 | | | | | | | | | | | |
| T16 | | | | | | | | | | | |
| T17 | | | | | | | | | | | |
| T18 | | | | | | | | | | | |
| T19 | | | | | | | | | | | |
| T20 | | | | | | | | | | | |
| T21 | | | | | | | | | | | |
| T22 | | | | | | | | | | | |
| T23 | | | | | | | | | | | |
| T25 | | | | | | | | | | | |

Cuadro 3.3: Planificación: diagrama de Gantt

llevado a cabo sobre la relevancia de las señalizaciones inconsistentes de errores en lo que a escenarios de inconsistencia se refiere. En el capítulo 5 se especifican detalladamente los pasos seguidos en el diseño, la implementación y la verificación experimental del módulo AEFT tanto para topologías bus como topologías estrella. Finalmente, en el capítulo 6 se explican los procedimientos seguidos durante el diseño y la implementación de CANsistant.

# Capítulo 4

# Estudio de la relevancia de los nuevos escenarios de inconsistencia

## 4.1. Introducción

Como se ha visto en el capítulo 3.3.1, existen tres causas primarias de inconsistencias en CAN: el estado de error pasivo, la regla del último bit del EOF y la señalización inconsistente de errores. La relevancia de los dos primeros tipos de inconsistencia ha sido resaltada en la literatura y se han propuesto varias soluciones tales como inhabilitar el estado de error pasivo [RUFI98], modificar el protocolo CAN (MajorCAN) [PROE00] y el mecanismo CANsistant [PROE09]. Respecto a la señalización inconsistente de errores, al ser una causa de inconsistencias identificada recientemente, no existen estimaciones de probabilidad ni estudios. Por lo tanto, desde el punto de vista del diseño de sistemas con elevada garantía de funcionamiento, es importante determinar si la frecuencia de ocurrencia de estos nuevos escenarios de inconsistencia es lo suficientemente elevada como para tener que resolverlos, o si, por el contrario, se trata de escenarios "patológicos" cuya frecuencia de aparición es tan reducida que puedan ser despreciados sin perjuicio para la fiabilidad del sistema.

En el capítulo 3.3.1 ya se identificaron dos condiciones para que se produzcan

estos nuevos escenarios: que la trama transmitida presente una secuencia de CRC específica y que se produzcan dos o más errores en bits concretos de la trama que enmascaren la señalización de errores. Este estudio se centra mayoritariamente en determinar la probabilidad de aparición de las mencionadas tramas, denominadas de ahora en adelante *tramas vulnerables*.

La metodología seguida se puede dividir en tres partes:

1. Identificar las tramas vulnerables.

2. Calcular la probabilidad de las tramas vulnerables.

3. Realizar un estudio estadístico de la distribución de las tramas vulnerables.

El análisis realizado se basa en la suposición de producirse como máximo dos fallos de canal (que pueden afectar a la transmitido/recibido por uno o varios nodos). Elevando el número de errores, la cantidad de escenarios de inconsistencia aumenta. Sin embargo, estos nuevos escenarios son despreciables ya que la probabilidad de producirse tres o más errores es mucho menor en comparación con la de producirse dos. De todos modos, la solución propuesta a este tipo de inconsistencias, la transmisión de *aggregated error flags*, no solo está diseñada para mantener la consistencia de datos en presencia de dos errores sino incluso para el caso de tres o más errores tal y como se verá en el capítulo 5.

La relevancia final de los escenarios de inconsistencia debidos a la señalización inconsistente de errores no solo depende de la probabilidad de aparición de las tramas vulnerables sino también de la probabilidad de producirse dos errores en bits específicos de esta trama. Así pues, para terminar el estudio es necesario disponer de un modelo de errores de canal, el cual permita determinar la probabilidad de que existan 2 errores que puedan causar un IMO en una trama vulnerable. Sin embargo, el uso de estos modelos es muy controvertido e incluso varios autores no lo recomiendan. Por este motivo, los directores de este trabajo final de Máster decidieron no considerarlos. La pregunta decisiva de este estudio es por lo tanto: ¿Es la frecuencia de aparición de las tramas vulnerables lo suficientemente elevada como para considerar los escenarios de inconsistencia resultantes como relevantes?

## 4.2. Identificación de las tramas vulnerables

Los nuevos escenarios de inconsistencia estudiados, causados por dos fallos de canal, sólo pueden producirse como consecuencia de ciertas combinaciones de errores en los últimos bits del CRC (ver el capítulo 3.3.1). Además, estas combinaciones de errores sólo causan inconsistencias cuando el CRC termina con una determinada secuencia de *'0s'* y *'1s'*.

El primer paso del estudio de relevancia fue identificar las terminaciones de CRC que en combinación con dos fallos de canal puedan provocar una inconsistencia. A estas terminaciones se les dio el nombre de *Secuencias de CRC vulnerables*. Su búsqueda, mediante el simulador CANfidant, ha sido sistemática pero no exhaustiva por lo que puede haber otras secuencias no contempladas. Por el motivo anterior, los valores obtenidos de aquí en adelante pueden considerarse una aproximación ligeramente pesimista.

Las secuencias de CRC vulnerables (considerando, en primer lugar, los últimos 12 bits del CRC) son seis y se pueden dividir en dos grupos bien diferenciados. El primer grupo se compone por cuatro secuencias de CRC cuyo formato es peculiar: una secuencia de seis bits recesivos consecutivos con un bit de valor dominante intercalado (denominado a partir de ahora como bit discordante), seguido de cuatro bits dominantes. Teniendo en cuenta los bits de *stuff* es posible identificar un segundo grupo de secuencias de CRC (compuesto por dos secuencias adicionales). El formato de estas secuencias es ligeramente diferente al de las secuencias del primer grupo: una secuencia de cinco bits recesivos o dominantes consecutivos seguido de cuatro bits dominantes. Tras la secuencia de cinco bits del mismo valor, antes de transmitir la trama, se intercalará un *stuff bit*.

El primer grupo se compone por las siguientes secuencias de CRC vulnerables:

- 370hex (001101110000b)
- 3B0hex (001110110000b)
- 3D0hex (001111010000b)
- 2F0hex (001011110000b)

El segundo grupo está formado por dos secuencias de CRC vulnerables adicionales:

- E00hex (111000000000b)
- F00hex (111100000000b)

Como se puede observar, las secuencias de CRC vulnerables identificadas solo se componen de 12 bits. Sin embargo, también se deben tener en cuenta los tres bits más significativos del CRC ya que dependiendo de su valor se intercalará un *stuff bit* tras el quinto, sexto o séptimo bit del CRC, caso en el que la secuencia de CRC puede convertirse en no vulnerable. El conjunto total de secuencias de CRC vulnerables, incluyendo los 15 bits, son 37 y se muestran en la Tabla 4.1 (las no vulnerables aparecen tachadas).

| | | | | | |
|---|---|---|---|---|---|
| ~~0370hex~~ | ~~03B0hex~~ | ~~03D0hex~~ | ~~02F0hex~~ | ~~0F00hex~~ | 0E00hex |
| 1370hex | 13B0hex | 13D0hex | 12F0hex | 1F00hex | 1E00hex |
| 2370hex | 23B0hex | 23D0hex | 22F0hex | ~~2F00hex~~ | 2E00hex |
| 3370hex | 33B0hex | 33D0hex | 32F0hex | ~~3F00hex~~ | ~~3E00hex~~ |
| 4370hex | 43B0hex | 43D0hex | 42F0hex | ~~4F00hex~~ | 4E00hex |
| 5370hex | 53B0hex | 53D0hex | 52F0hex | 5F00hex | 5E00hex |
| 6370hex | 63B0hex | 63D0hex | 62F0hex | ~~6F00hex~~ | 6E00hex |
| 7370hex | 73B0hex | 73D0hex | 72F0hex | ~~7F00hex~~ | 7E00hex |

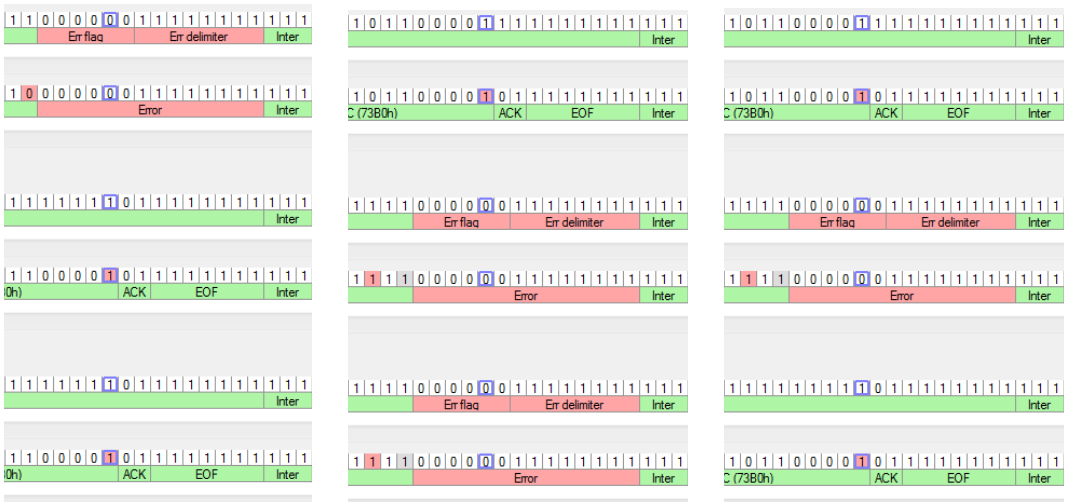
Cuadro 4.1: 37 secuencias de CRC vulnerables identificadas.

Las secuencias no vulnerables mostradas en la Tabla 4.1 y algunas terminaciones adicionales (como por ejemplo 1F0hex, D00hex y 0F0hex) no son propensas a causar inconsistencias en el caso de sufrir dos errores pero si en el caso de sufrir tres o más. Recuérdese, sin embargo, que en el análisis realizado se han descartado los escenarios con tres o más errores ya que son despreciables en relación a los escenarios con dos errores.

Los escenarios de inconsistencia analizados se generan de forma similar y se ejemplifican en la Figura 4.1 (a): si el bit discordante o el bit de *stuff* sufren un error, esto generará un *stuff error* ya que los nodos afectados verán seis bits consecutivos del mismo valor. Seguidamente, los nodos que han detectado el error intentarán señalizarlo pero, al terminar el CRC en una secuencia de cuatro bits dominantes,

los cuatro primeros bits del *error flag* serán NPB y no forzarán la globalización. El quinto bit del EF sí será EPB ya que afecta un campo de formato fijo que por definición toma valor recesivo, el delimitador de CRC. Sin embargo, si se produce un segundo error que, justamente, enmascara dicho bit, los nodos no afectados por el error inicial seguirán sin detectar errores. El sexto y último bit del EF afecta al ACK y, al ser de valor dominante, confirma de forma involuntaria la petición de confirmación por lo que será siempre NPB. De esta forma los nodos que han detectado el error inicial rechazarán la trama y el resto de nodos, sin percatarse de la presencia de un EF, la aceptarán.

La inconsistencia, sin embargo, también es posible en otros casos. En las Figuras 4.1 (b) y 4.1 (c) se muestran dos escenarios de inconsistencia adicionales generados a partir de la misma secuencia de CRC vulnerable que en el caso anterior y en presencia de dos fallos de canal.




(a) IMD (b) IMO 1 (c) IMO 2

Figura 4.1: Diferentes escenarios de inconsistencia sobre una misma secuencia de CRC vulnerable

## 4.3. Cálculo de la probabilidad de las tramas vulnerables

El segundo paso del estudio consiste en determinar la probabilidad de aparición de las tramas vulnerables. En el apartado anterior se han identificado 37 secuencias de CRC vulnerables de entre las $2^{15}$ combinaciones posibles de CRC. Teóricamente, suponiendo que los CRCs se generan siguiendo una distribución uniforme, el porcentaje de CRCs vulnerables respecto al número total de CRCs es $37/2^{15} \cdot 100 \approx 0{,}001\,\%$. Por consiguiente, se puede hipotetizar que una de cada mil tramas transmitidas será vulnerable. Esta hipótesis se puede comprobar de dos formas: algebraicamente, a partir de las propiedades matemáticas del CRC o, computacionalmente, generando todas las posibles combinaciones de *Identificador + DLC + Datos*, obteniendo sus CRCs correspondiente e identificando aquellas combinaciones que generan las secuencias de CRC vulnerables, denominadas de ahora en adelante *Combinaciones Generadoras*.

Al ser la comprobación algebraica relativamente complicada, se optó por el cálculo computacional. Esta opción además tuvo la ventaja de que permitió realizar un estudio estadístico de la distribución de las combinaciones generadoras. Para ello se implementó un programa mediante el entorno de programación *Matlab* que calcula sistemáticamente todos los CRCs e identifica aquellos que se corresponden con las secuencias de CRC vulnerables. Primero se calcularon las combinaciones para una trama sin datos, después para una trama con un byte de datos, luego dos bytes de datos, y así sucesivamente.

El algoritmo de generación de CRCs de CAN [ISO93] es una derivación del algoritmo CRC-15. Los coeficientes del polinomio a dividir vienen dados por la trama CAN (incluyendo el SOF, el campo de arbitraje, el campo de control y el campo de datos) y, para los 15 coeficientes inferiores, por 0. Este polinomio es dividido (los coeficientes son calculados mediante aritmética de Módulo 2) por el polinomio generador mostrado en la expresión 4.1. El resto de la división es la secuencia CRC transmitida por el bus.

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1 \tag{4.1}$$

Para su implementación se puede utilizar un registro de desplazamiento de 15 bits. Seguidamente se muestra un ejemplo de pseudocódigo para el cálculo de CRCs en el que `NXTBIT` denota el siguiente bit de la trama y `CRC_RG(14:0)` almacena la secuencia de CRC:

```
CRC_RG = 0 //initialize shift register
REPEAT
CRCNXT = NXTBIT EXOR CRC_RG(14)
CRC_RG(14:1) = CRC_RG(13:0) //shift left by...
CRC_RG(0) = 0 //...one position
IF CRCNXT THEN
CRC_RG(14:0) = CRC_RG(14:0) EXOR (4599 hex)
ENDIF
UNTIL (CRC SEQUENCE starts or there is an ERROR condition)
```

Matlab (*MATrix LABoratory*) es un programa de cálculo numérico orientado a matrices. Por tanto, será más eficiente si se diseñan los algoritmos en términos de operaciones vectoriales. En el caso presente se optó por una solución híbrida: para el cálculo de las combinaciones generadoras para 0 y 1 bytes de datos se utilizaron exclusivamente operaciones vectoriales. En cambio, para el caso de 2 y 3 bytes de datos se usaron operaciones vectoriales en conjunto con bucles *for*. Aunque este tipo de bucles sean extremadamente ineficientes en Matlab, no se tuvo elección ya que la cantidad de combinaciones y las correspondientes matrices eran inmensas y se generaban errores de *out of memory*, es decir, se requería más memoria de la que había disponible. La utilización de bucles *for* aumentó considerablemente los tiempos de ejecución.

El código correspondiente al cálculo de las combinaciones generadoras para dos bytes de datos se muestra a continuación. El cálculo para tres bytes de datos simplemente incluye un segundo bucle *for* superpuesto al primero y no será expuesto en esta memoria.

```
%-------------------------------------------------%
%CRC calculation and vulnerable CRC checking  %
%-------------------------------------------------%
clc;clear all;close all;
%-------------------------------------------------%
tic;          %Start execution-time counter
```

```
%-------------------------------------------------------%
RTR=0;          %RTR configuration(0/1)
DLC=2;          %DLC configuration(0-8)
%-------------------------------------------------------%
%Vulnerable CRC Sequences
crit=fliplr(de2bi(1:7,3));
vulnerable_crc1=repmat([0 0 1 1 0 1 1 1 0 0 0 0],7,1);
vulnerable_crc1=[crit vulnerable_crc1];
vulnerable_crc2=repmat([0 0 1 1 1 0 1 1 0 0 0 0],7,1);
vulnerable_crc2=[crit vulnerable_crc2];
vulnerable_crc3=repmat([0 0 1 1 1 1 0 1 0 0 0 0],7,1);
vulnerable_crc3=[crit vulnerable_crc3];
vulnerable_crc4=repmat([0 0 1 0 1 1 1 1 0 0 0 0],7,1);
vulnerable_crc4=[crit vulnerable_crc4];
crit=fliplr(de2bi([0 1 2 4 5 6 7],3));
vulnerable_crc5=repmat([1 1 1 0 0 0 0 0 0 0 0 0],7,1);
vulnerable_crc5=[crit vulnerable_crc5];
crit=fliplr(de2bi(0:6,3));
vulnerable_crc6=repmat([1 1 0 1 0 0 0 0 0 0 0 0],7,1);
vulnerable_crc6=[crit vulnerable_crc6];
crit=fliplr(de2bi([1 5],3));
vulnerable_crc7=repmat([1 1 1 1 0 0 0 0 0 0 0 0],2,1);
vulnerable_crc7=[crit vulnerable_crc7];
crit=fliplr(de2bi([2 6],3));
vulnerable_crc8=repmat([0 0 0 0 1 1 1 1 0 0 0 0],2,1);
vulnerable_crc8=[crit vulnerable_crc8];
%Generator Polynomial & CRC Generator
crc15=[1 1 0 0 0 1 0 1 1 0 0 1 1 0 0 1];
h=crc.generator(crc15);
%CAN frame
frame_bits=19+(DLC*8);          %Number of bits constituting the frame
frame=zeros(2048,frame_bits);   %Empty matrix for storing the frames
frame(:,13)=RTR;                %RTR value
num_bytes=fliplr(de2bi(DLC,4)); %Binary DLC value
num_bytes=repmat(num_bytes,2048,1);
frame(:,16:19)=num_bytes;       %Fill the matrix with the DLC values
%-------------------------------------------------------%
%CRC generation
id=fliplr(de2bi(0:2047,11));    %2048 combinations of the ID
frame(:,2:12)=id;               %Fill the matrix with the IDs
frame2=repmat(frame,256,1);
data_matrix=0:255;              %256 combinations of the Data Byte
data_matrix=repmat(data_matrix,2048,1);
data_matrix=reshape(data_matrix,(2048*256),1); %Reschape the matrix
data_matrix=fliplr(de2bi(data_matrix,8));   %Convert into binary
frame2(:,20:27)=data_matrix;     %Fill the matrix with the Data values
for n=0:255      %Loop for calculating the data value combinations
    data_matrix=ones((2048*256),1); %Vectorized calculation of the...
    data_matrix=data_matrix*n;      %...ID combinations
    frame2(:,28:35)=fliplr(de2bi(data_matrix,8));
```

```
    frame_t=frame2';
    encoded=generate(h,frame_t);      %CRC generation
    crc=(encoded(36:50,:))';
    %Vulnerable frame identification (just showing the code
    %for the first set of vulnerable CRCs)
    vulnerable1a=strmatch(vulnerable_crc1(1,:),crc);
    vulnerable1b=strmatch(vulnerable_crc1(2,:),crc);
    vulnerable1c=strmatch(vulnerable_crc1(3,:),crc);
    vulnerable1d=strmatch(vulnerable_crc1(4,:),crc);
    vulnerable1e=strmatch(vulnerable_crc1(5,:),crc);
    vulnerable1f=strmatch(vulnerable_crc1(6,:),crc);
    vulnerable1g=strmatch(vulnerable_crc1(7,:),crc);
    ...
    %Vertical concatenation of the vulnerable frames
    set=vertcat(vulnerable1a,vulnerable1b,vulnerable1c,vulnerable1d,...
    vulnerable1e,vulnerable1f,vulnerable1g,vulnerable2a,vulnerable2b,...
    vulnerable2c,vulnerable2d,vulnerable2e,vulnerable2f,vulnerable2g,...
    vulnerable3a,vulnerable3b,vulnerable3c,vulnerable3d,vulnerable3e,...
    vulnerable3f,vulnerable3g,vulnerable4a,vulnerable4b,vulnerable4c,...
    vulnerable4d,vulnerable4e,vulnerable4f,vulnerable4g,vulnerable4a,...
    vulnerable5b,vulnerable5c,vulnerable5d,vulnerable5e,vulnerable5f,...
    vulnerable5g,vulnerable4a,vulnerable6b,vulnerable6c,vulnerable6d,...
    vulnerable6e,vulnerable6f,vulnerable6g,vulnerable4a,vulnerable4b,...
    vulnerable4a,vulnerable4b);
    %Sort the set of vulnerable CRCs
    set=sort(set);
    %Identify the Generating Combinations
    vulnerable_frame((1+(448*n)):(448*(n+1)),1)=...
    bi2de(fliplr(frame2(vulnerable,2:12)));
    vulnerable_frame((1+(448*n)):(448*(n+1)),2)=...
    bi2de(fliplr(frame2(vulnerable,20:27)));
    vulnerable_frame((1+(448*n)):(448*(n+1)),3)=...
    bi2de(fliplr(frame2(vulnerable,28:35)));
end
%Stop execution-time counter
toc;
```

Las combinaciones generadoras se guardan en la matriz `critical_frame`. Otro dato importante es el tamaño de esta misma matriz ya que indica el número exacto de combinaciones generadoras encontradas según el número de bytes de datos de la trama analizada. En la Tabla 4.2 se muestran los resultados obtenidos.

Únicamente se realizó el cálculo para 0, 1, 2 y 3 bytes de datos. Esto se debe a que a partir de 4 bytes el número de combinaciones y el tiempo de ejecución se disparan (ya para 3 bytes el tiempo de ejecución ha sido de casi dos días). De todos modos, a partir de los datos recopilados ya es posible observar una tendencia clara:

| Bytes de datos | Combinaciones | Tiempo de ejecución | Nº de combinaciones generadoras |
|---|---|---|---|
| 0 | 2048 | $0,18s$ | 2 |
| 1 | 524288 | $4,5s$ | 592 |
| 2 | 134217728 | $1080s(18min)$ | 151552 |
| 3 | $3,4359 \cdot 10^{10}$ | $15660s(43h,30m)$ | 38797312 |

Cuadro 4.2: Resultados obtenidos mediante Matlab

el número de combinaciones generadoras aumenta en un factor de 256 con cada byte de datos que se añada. Esto es fácilmente demostrable a partir de los resultados obtenidos:

$$\frac{151552}{592} = \frac{38797312}{151552} = 256 \tag{4.2}$$

El factor se corresponde exactamente con el número de combinaciones que se obtienen con un byte de datos, es decir, ocho bits: $2^8 = 256$.

Para un byte de datos, calculando individualmente el número de combinaciones generadoras por cada secuencia de CRC vulnerable (no conjuntamente como en el caso anterior), se obtuvo siempre el mismo resultado, 16. Esto es debido a que, de entre las $2048 \cdot 256 = 524288$ combinaciones posibles de identificador y datos, existen 16 combinaciones generadoras que producen la misma secuencia de CRC vulnerable. Analícese esta peculiaridad: a partir de una secuencia de $n$ bits se obtienen $2^n$ combinaciones diferentes. El algoritmo de generación de CRCs de CAN se basa en un polinomio generador de grado 15, motivo por el cual, en CAN los CRCs son de 15 bits. Consecuentemente, si la secuencia de bits a partir de la cual se desea calcular el CRC es también de 15 bits, habrá un CRC distinto por cada combinación. En cambio, si es de 16 bits, habrá $2^1 = 2$ combinaciones que compartan el mismo CRC. Para el caso de 17 bits habrá $2^2 = 4$ combinaciones con CRC idéntico y así sucesivamente.

Generalizando el resultado observado, se obtuvo la Fórmula 4.3 para el cálculo del número de combinaciones generadoras según el número de bytes de datos.

$$\text{Si } d > 0 \text{ entonces } g = v \cdot 2^{id+8d-p} = 37 \cdot 2^{id+8d-15} \tag{4.3}$$

Donde $g$ es el número de combinaciones generadoras, $v$ es el número de secuencias vulnerables identificadas en el apartado anterior (37), $id$ el número de bits que componen el identificador ($11b$ en el caso de tramas estándar y $29b$ en el caso de tramas extendidas) y $d$ el número total de bytes de datos. $p$ es el grado del polinomio generador utilizado y define a partir de cuantos bits se empiezan a repetir los CRCs (en este caso 15).

La Fórmula 4.3 no solo es aplicable a las tramas CAN estándar sino también a las de formato extendido. Para demostrar la validez de la última afirmación se modificó el código Matlab original. Concretamente, se modificaron los campos de control y arbitraje y se aumentó el número de combinaciones (antes eran $2^{11}$ y ahora son $2^{29}$). Ejecutando el programa se obtuvieron los resultados esperados: $1{,}552 \cdot 10^8$ combinaciones generadoras, cifra idéntica a la calculada con la expresión 4.3. El código correspondiente, debidamente comentado, se muestra a continuación:

```
%-------------------------------------------------------------------
%CRC calculation and generating combinations identification
%for an extended CAN frame
%-------------------------------------------------------------------
SRR=1;              %Substitute Remote Request
ID_extension=1;
RTR=0;              %(0/1)
r0=0;
r1=0;
DLC=1;              %(1-8 bytes)
%-------------------------------------------------------------------
frame_bits=39+(DLC*8);              %Extended frame
frame=zeros((2^29),frame_bits);     %All combinations (2^29)
frame(:,13)=SRR;                    %Control field
frame(:,14)=ID_extension;
frame(:,33)=RTR;
frame(:,34)=r0;
frame(:,35)=r1;
num_bytes=fliplr(de2bi(DLC,4));
num_bytes=repmat(num_bytes,(2^29),1);
frame(:,36:39)=num_bytes;           %DLC field
%-------------------------------------------------------------------
%CRC generation
id=fliplr(de2bi(0:(2^29),29));      %Id combinations
frame(:,2:12)=id(:,1:11);           %Arbitration field (1)
frame(:,20:32)=id(:,12:29);         %Arbitration field (2)
frame2=repmat(frame,256,1);
data_matrix=0:255;
data_matrix=repmat(data_matrix,(2^29),1);
```

```
data_matrix=reshape(data_matrix,((2^29)*256),1);
data_matrix=fliplr(de2bi(data_matrix,8));
frame2(:,40:47)=data_matrix;      %Data field
frame_t=frame2';
encoded=generate(h,frame_t);      %CRC generation
crc=(encoded(48:62,:))';          %CRC field
%-----------------------------------------------------------------
```

A partir de la expresión 4.3 se calcularon el número de combinaciones para 4, 5, 6, 7 y 8 bytes de datos. Las combinaciones calculadas se muestran en la Tabla 4.3.

| Bytes de datos | Nº de combinaciones generadoras (aprox.) |
|---|---|
| 4 | $9,932 \cdot 10^{9}$ |
| 5 | $2,543 \cdot 10^{12}$ |
| 6 | $6,509 \cdot 10^{14}$ |
| 7 | $1,666 \cdot 10^{17}$ |
| 8 | $4,266 \cdot 10^{19}$ |

Cuadro 4.3: Cálculo de las combinaciones generadoras.

Para finalizar el estudio se calculó la probabilidad de generarse una secuencia de CRC vulnerable. Para ello se divide el número de combinaciones generadoras entre el número total de combinaciones posibles:

$$p = \frac{\text{Nº de combinaciones generadoras}}{\text{Nº de combinaciones total}} = \frac{37 \cdot 2^{id+8d-15}}{2^{id+8d}} = 37 \cdot 2^{-15} = 1,129 \cdot 10^{-3} \tag{4.4}$$

Los valores mostrados en la Tabla 4.3 y la probabilidad calculada mediante la expresión 4.4, demuestran que las tramas vulnerables aparecen con la frecuencia prevista (aproximadamente una de cada mil). Ésta es un frecuencia elevada y pone en evidencia la relevancia de las tramas vulnerables como fuente de potenciales señalizaciones inconsistentes de errores.

## 4.4. Análisis estadístico

Adicionalmente al estudio de las secuencias de CRC vulnerables se realizó un análisis estadístico de las combinaciones generadoras para estudiar la distribución

de éstas respecto a los identificadores y a los bytes de datos. Con ello, se persigue identificar posibles regiones de identificadores o datos más susceptibles a sufrir inconsistencias. Podría ser, por ejemplo, que grupos específicos de identificadores o de datos generen muchas secuencias de CRC vulnerables y otros, no generen ninguna. En ese caso hipotético se podría prohibir el uso de los identificadores mencionados en sistemas distribuidos con elevada garantía de funcionamiento, reduciéndose el riesgo de sufrir inconsistencias. Sin embargo, se puede decir de antemano que los resultados demostrarán que no existen estas agrupaciones de identificadores o datos.

En primer lugar se calcularon la media aritmética y la mediana de los dos conjuntos. Estas medidas de tendencia central pueden dar una idea sobre la distribución de los conjuntos. Si la distribución es uniforme, la media y la mediana tomarán valores cercanos a $2048/2 = 1024$ para el conjunto de identificadores y $256/2 = 128$ para el conjunto de datos. El código Matlab utilizado se muestra a continuación:

```
%-------------------------------------------------------
% Statistical Analysis
%-------------------------------------------------------
clear all;close all;clc;
load matlab2.mat;    % Load data
x=sortrows(critical_frame); % Sort the values indescending order
%-------------------------------------------------------
% Statistically relevant values
median(x(:,1))       % Median
mean(x(:,1))         % Mean
%-------------------------------------------------------
```

Los resultados obtenidos se resumen en la tabla 4.4. Se observa que los resultados se aproximan bastante a los esperados por lo que se puede suponer que la distribución de las combinaciones generadoras es uniforme.

| | Conjunto de Identificadores | Conjunto de Datos |
|---|---|---|
| Media | 1023,5 | 127,5 |
| Mediana | 1015,5 | 127,5 |

Cuadro 4.4: Media y mediana de los conjuntos de identificadores y datos (para tramas con ID de 11 bits y un byte de datos).

Para ilustrar gráficamente las distribuciones, se optó por la función *stairs* de Matlab que genera distribuciones acumulativas a partir de un conjunto de datos.

Adicionalmente se generaron histogramas a partir de esos mismos datos, agrupándolos en grupos para estudiar su frecuencia. En la Figura 4.2a y la Figura 4.2b se muestran, respectivamente, las distribuciones acumulativas de las combinaciones generadoras respecto a los identificadores y a los datos. Los histogramas con 32 grupos se muestran en la Figura 4.3a y la 4.3b.

En la figura 4.2 se puede observar que la distribución acumulativa tanto para los identificadores como para los datos viene definida por una recta, es decir, es uniforme. Se aprecian pequeños saltos pero son despreciables. En los histogramas (Figura 4.3) se observa la misma situación: los 32 grupos tienen aproximadamente el mismo número de valores por grupo (de entre 16 y 20), característica que se atribuye a una distribución uniforme. A partir de los resultados obtenidos se llega a la conclusión de que las combinaciones generadoras se distribuyen uniformemente sobre el conjunto de identificadores y el conjunto de datos. No existen, pues, identificadores o datos especialmente críticos que causen una mayor cantidad de secuencias de CRC vulnerables.

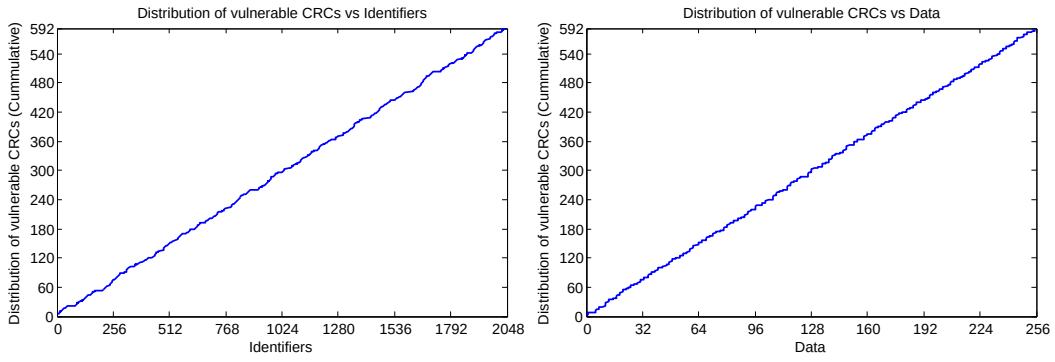



Figura 4.2: Distribución de CRCs vulnerables respecto a los identificadores (a) y los datos (b)

## 4.5. Conclusiones

En este capítulo se ha realizado un estudio de la relevancia de los escenarios de inconsistencia debidos a la señalización inconsistente de errores. El objetivo de este estudio ha sido determinar la frecuencia de aparición de las tramas vulnerables. Para ello en primer lugar se han identificado las secuencias de CRC vulnerables,

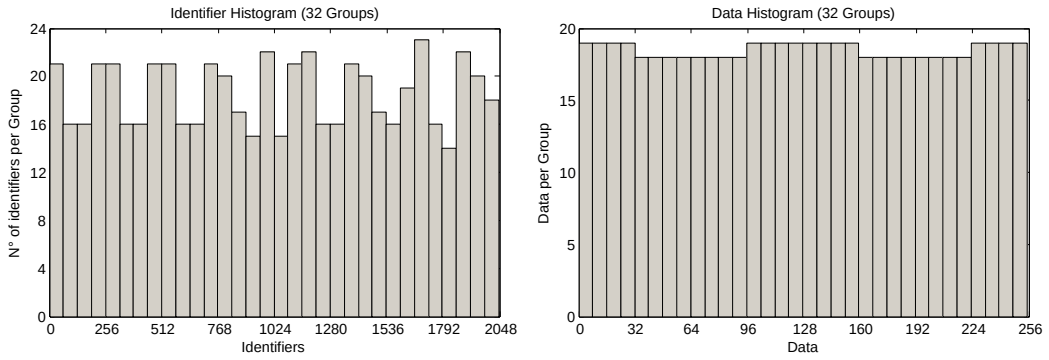



Figura 4.3: Histograma de identificadores (a) e histograma de datos (b)

secuencias que al sufrir solo dos errores ya pueden generar escenarios de inconsistencia. Seguidamente, mediante métodos computacionales (Matlab), se han hallado las combinaciones generadoras de identificador y datos que producen secuencias de CRC vulnerables. El número de estas combinaciones es elevado y representa algo más de una milésima parte del conjunto total de combinaciones. Esto demuestra que la señalización inconsistente de errores es un problema relevante, ya que puede ocurrir con una frecuencia no despreciable.

Adicionalmente, se ha realizado un análisis estadístico para determinar la distribución de las combinaciones generadoras respecto a los conjuntos de identificadores y de datos. El resultado fue una distribución uniforme tanto para un conjunto como para el otro. Con ello se pone en evidencia que no existen grupos de identificadores o datos especialmente propensos a generar secuencias de CRC vulnerables y que, por lo tanto, resulta necesario una solución que ataque la raíz del problema.

Para el caso de la arquitectura CANbids, el problema de la señalización inconsistente de errores se resolverá añadiendo el mecanismo AEFT basado en la transmisión de AEFs (tal como se ha descrito en el capítulo 3.3.1). El diseño y la implementación del AEFT ha sido uno de los objetivos centrales de este trabajo y que se describe detalladamente en el próximo capítulo 5.

# Capítulo 5

# AEFT: diseño e implementación

## 5.1. Introducción

En el capítulo 3.3 se expusieron las limitaciones del protocolo CAN, poniendo énfasis en el problema de la limitada consistencia de datos. En ese contexto, se identificaron tres causas básicas que pueden llevar a inconsistencias: la existencia del estado de error pasivo, la regla del último bit del EOF y la señalización inconsistente de errores. En 3.4 se propusieron mecanismos encaminados a resolver dichas inconsistencias. Para solventar el problema de la señalización inconsistente de errores se describió una estrategia basada en la agregación de señalizadores de error denominada AEF (*Aggregated Error Flag*).

El objetivo de este capítulo es la propuesta, el diseño y la implementación física de un dispositivo que detecte la posible ocurrencia de inconsistencias e inyecte un AEF cuando sea necesario. El dispositivo debe reunir , además, características tales como la ortogonalidad y la compatibilidad total con el protocolo CAN. A este módulo se le dio el nombre de *Aggregated Error Flag Transmitter* (AEFT).

En base a un mismo diseño, salvo ligeras modificaciones, se han realizado dos implementaciones: una para topologías bus y otra para topologías estrella. El flujo de diseño fue el siguiente:

1. Definir la tarea o tareas que debe realizar el módulo.

2. Diseñar el módulo en base a diagramas de bloques y máquinas de estado.
3. Escribir el programa usando el lenguaje VHDL.
4. Comprobar la sintaxis, compilar y simular el programa.
5. Implementar el diseño en una FPGA y construir un primer prototipo.
6. Verificar experimentalmente el correcto funcionamiento del prototipo y, así, demostrar la utilidad de la propuesta.

El capítulo se organiza de la siguiente forma: primero se describe el diseño para topologías bus, incluyendo las diferentes opciones de diseño y la arquitectura interna del AEFT. Luego se definen las modificaciones realizadas para adaptar el diseño a las topologías en estrella. Seguidamente, se explica la implementación en topologías bus (incluyendo la construcción del prototipo y su comprobación experimental)y, finalmente, se detallan los pasos seguidos para la implementación en topologías estrella.

## 5.2. Diseño: topología bus

El diseño del AEFT se realizó, en un principio, para su utilización en topologías bus y fue, posteriormente, adaptado a las topologías estrella. En primer lugar, se tuvo que elegir la localización del AEFT dentro de la red CAN y se optó por intercalarlo entre el controlador CAN y el *transceiver* de un nodo tal como se muestra en la Figura 5.1. Obsérvese que de esta forma habrá un AEFT por cada nodo (en una red CAN con $n$ nodos se deberán incluir $n$ AEFTs).

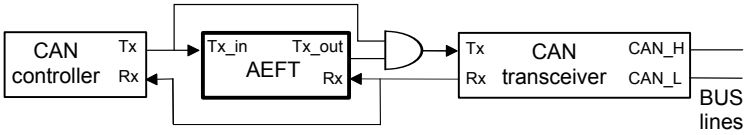



Figura 5.1: Localización del AEFT

Como se muestra en la Figura 5.1, el módulo AEFT tiene acceso a la señal $Tx$ transmitida por el controlador CAN y a la señal $Rx$ recibida por el *transceiver*. Si se detecta el envío de un *error flag* por parte del controlador, el AEFT se activa

e inyecta un AEF, superponiéndose al controlador CAN gracias a la puerta *AND*. Recuérdese que un AEF, tal como se definió en 3.4, se compone de varias secuencias de 6 bits dominantes consecutivos separadas por, almenos, un bit recesivo. Estas secuencias a partir de ahora se denominarán EFs.

### 5.2.1. Opciones de diseño

La idea original fue diseñar un dispositivo completamente independiente con únicamente dos entradas, $Tx$ y $Rx$ (tal como se ha mostrado en la Figura 5.1). Esto significa que el reloj de transmisión, $clk_T$ y el reloj de recepción, $clk_R$ se debían generar internamente. Para ello fue necesario implementar la capa física de CAN, compuesta por el BRP (*Baud-Rate Prescaler*) y el sincronizador, en el AEFT y, además, incorporar un oscilador propio para generar la señal de reloj $clk_{OSC}$.

Al estar acoplado el AEFT a un controlador CAN que ya dispone de las señales de reloj necesarias, resultó redundante generarlas internamente y, por lo tanto, se optó por eliminar la capa física del AEFT. Adicionalmente, la generación interna conllevaba problemas de sincronización ya que las frecuencias de los relojes del controlador CAN y del AEFT podían ser ligeramente diferentes produciéndose el denominado *clock drift* (desincronización de los relojes).

### 5.2.2. Arquitectura del AEFT

En la Figura 5.2 se muestra la estructura interna del AEFT. Éste está constituido por tres módulos. El módulo denominado AEFC (*Aggregated Error Flag Controller*) es la unidad de control del AEFT y decide cuándo inyectar un AEF. El EFD (*Error Flag Detector*) monitoriza la actividad del controlador CAN correspondiente e informa al AEFC cuando haya detectado la transmisión de un *error flag*. Finalmente, el EFT (*Error Flag Transmitter*), activado y controlado por el AEFC, inyecta los EFs que componen el AEF.

El AEFT presenta cuatro entradas: $Reset$, $Tx_{in}$, $Rx$, $clk_T$ y $clk_R$. La señal de reset es asíncrona y reinicializa los tres módulos. $Tx_{in}$ es la señal de transmisión proveniente del controlador CAN y $Rx$ es la señal de recepción procedente del *transceiver*. Las señales $clk_T$ y $clk_R$ se corresponden con el reloj de transmisión y el reloj

de recepción, respectivamente. La única salida del AEFT es la señal $Tx_{out}$.

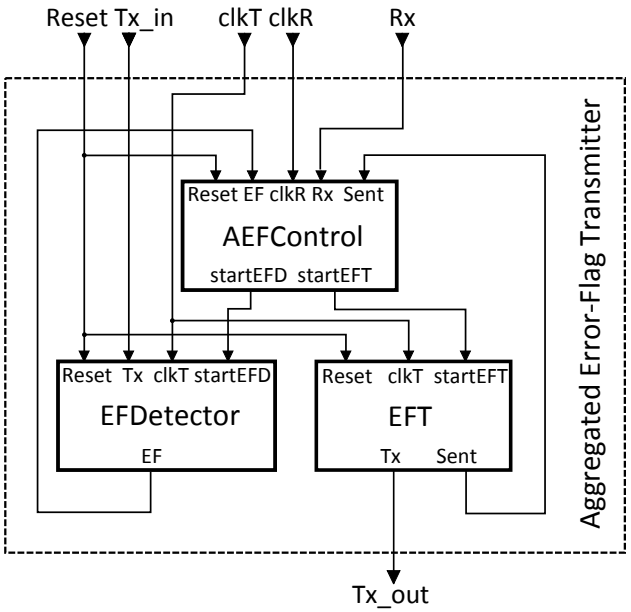



Figura 5.2: Arquitectura interna del AEFT

El módulo AEFC viene definido por la máquina de estados (FSM del inglés *Finite State Machine*) de la Figura 5.3. Dicha FSM está formada por 7 estados. Aunque no se muestre en la figura, la transición de estados viene condicionada a los flancos de subida de la señal $clk_R$, esto es, el reloj de recepción. En el estado inicial S0 se activa la señal de control $startEFD$ que pone en marcha el módulo EFD e inicia la detección de *error flags* transmitidos por el controlador CAN. En S1 se permanece hasta la activación de la señal de evento $EF$ que indica la detección de un *error flag* por parte del EFD. La transición del estado S2 al estado S3 viene condicionada a la recepción de un bit recesivo por $Rx$. En S3 se inicia la transmisión del primer EF poniendo en funcionamiento el módulo EFT mediante la señal de control $startEFT$. Nótese que un EF se compone por un bit recesivo seguido por seis bits dominantes. Adicionalmente, se incrementa el contador $transEF$ que indica el número de EFs transmitidos. La transición de S4 a S5 se realiza una vez transmitido el EF, momento en el cual el EFT activa la señal de evento $Sent$. En S5 se compara el valor del contador $transEF$ con la constante $m$. La constante $m$ es un parámetro modificable que indica el número total de EFs que componen el AEF. Si $transEF$ es menor o igual a $m$, se salta al estado S2 y se inicia el envío del siguiente EF. En cambio, si $transEF$ es mayor que $m$ se pasará al estado S6. En S6 se perdura durante 11 flancos

de subida (mediante la variable *numRec*) hasta regresar al estado S0 y reiniciar la detección de *error flags* del controlador. Este márgen de tiempo sirve para que todos los nodos terminen de transmitir sus señalizadores de error y se resincronicen. Se escogieron 11 tiempos de bit ya que esa es la duración del error delimiter (8 bits) y el IFS (3 bits).

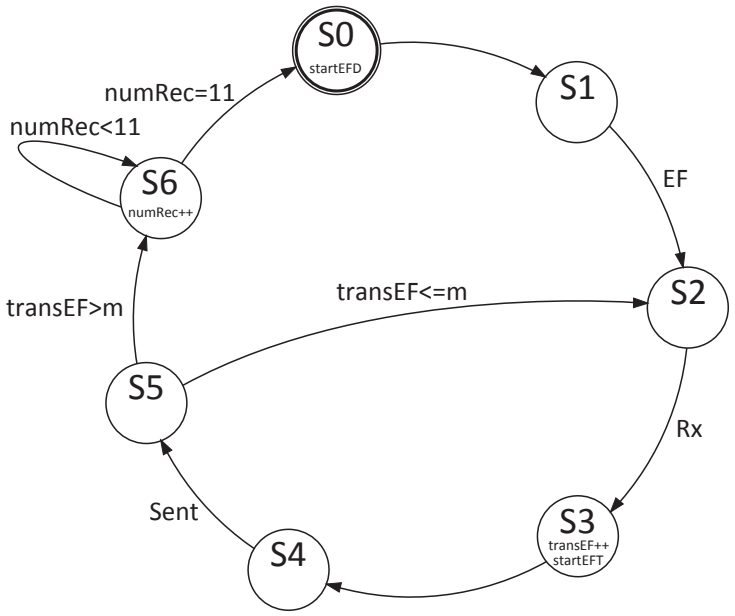



Figura 5.3: Máquina de estados del AEFC

En la Figura 5.4 se muestra la máquina de estados correspondiente al módulo EFD. Al igual que en el caso del AEFC, la transición de estados viene condicionada a los flancos de subida de una señal de reloj, en este caso del reloj de transmisión $clk_t$. La máquina de estados implementa un contador de bits que monitoriza la señal $Tx_{in}$ proveniente del controlador CAN. La transición del estado S0 al estado S1 viene condicionada por la señal de control *Start*. A partir del estado S1, se inicia un contador de seis dominantes consecutivos que avanza de estado en estado al detectar un bit dominante y regresa al estado S1 al detectar un bit recesivo. Cuando la máquina de estados alcanza el estado S7, es decir, ha contado seis bits dominantes consecutivos, se activa la señal de evento *EF* que le indica al AEFC que se ha detectado un *error flag*. Finalmente, se regresa al estado inicial S0 y se mantiene a la espera de una nueva activación por parte de la unidad de control.

El último módulo, el EFT, es un transmisor de señalizadores de error simplifica-

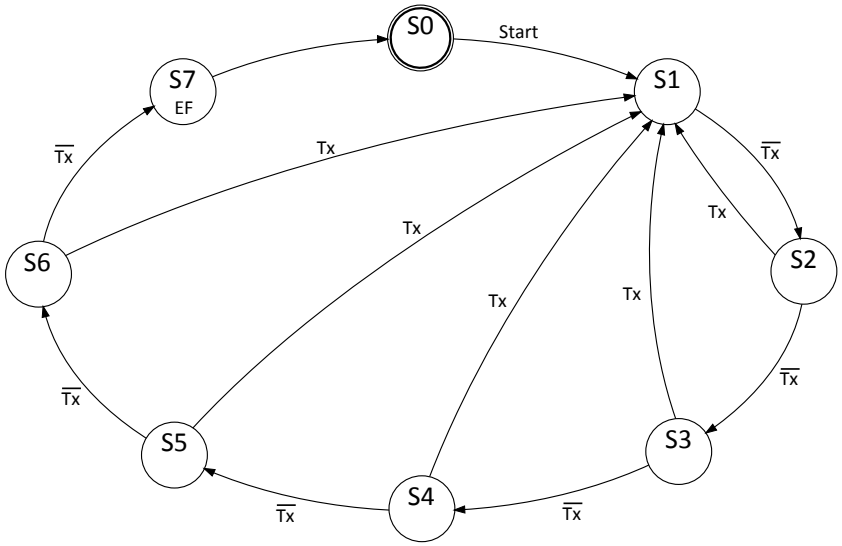



Figura 5.4: Máquina de estados del EFD

do cuya máquina de estados se ilustra en la Figura 5.5. La condición de transición es, como en el caso del EFD, un flanco de subida del reloj de transmisión $clk_T$. La máquina presenta 8 estados siendo S0 el estado inicial. Para no interferir con el funcionamiento normal del controlador CAN, en el estado S0 se envían constantemente bits de valor recesivo. La condición para la transición de S0 a S1 es la activación de la señal de control *Start* por parte del AEFC. En el estado S1 se transmite un bit recesivo por la salida $Tx_{out}$ del AEFT. Seguidamente, en los seis estados subsiguientes se transmiten los seis bits dominantes que constituyen el *error flag* propiamente dicho. Antes de volver al estado inicial S0, en S7 se activa la señal de evento *Sent*, indicando al AEFC que se ha inyectado un EF. Cabe denotar que el EFT es activado $m$ veces para la transmisión completa del AEF.

Uno de los requisitos del AEFT fue, desde un principio, la compatibilidad con cualquier controlador CAN del mercado. Por tal motivo y como se ha explicado en el párrafo anterior, el AEFT ha sido diseñado para que no interfiera con el controlador CAN en estado de funcionamiento normal. Como ya se ha mostrado en la Figura 5.1, la señal transmitida por el nodo al bus es resultado de una función *AND* de las señales $Tx$ del controlador CAN y $Tx_{out}$ del AEFT. En estado de funcionamiento normal, el AEFT únicamente envía bits de valor recesivo que no tiene influencia sobre el resultado de la función *AND* y, por consiguiente, será transparente desde el punto de vista del controlador CAN. Cuando se inyecta un AEF, en cambio, los

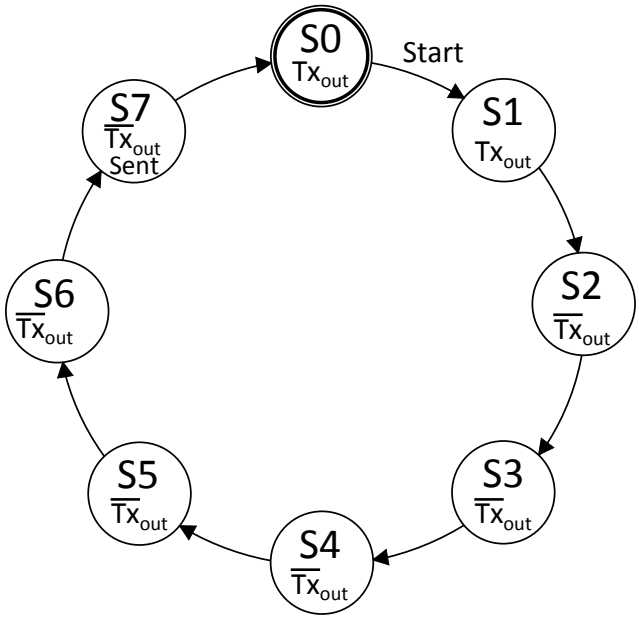



Figura 5.5: Máquina de estados del EFT

valores dominantes de los EFs se superpondrán a los valores transmitidos por el controlador.

Respecto a los relojes que controlan los diferentes módulos, decir que su elección fue intencionada. Como bien se ha podido ver, el módulo AEFC se basa en el reloj de recepción ($clk_R$) mientras que el EFD y el EFT se basan en el reloj de transmisión ($clk_T$). Por un lado esto es debido a que el AEFC monitoriza la señal $Rx$ mientras que el EFD controla la señal $Tx_{in}$ y el EFT trasnmite por la señal $Tx_{out}$. Sin embargo, el motivo principal de esta circunstancia fue otra. Usando dos relojes ligeramente desfasados, los módulos se mantienen sincronizados con el controlador CAN y el bus garantizando, a su vez, la detección de las señales de control y evento ya que los instantes de muestreo de dichas señales también estarán mínimamente desfasados.

La espera a recibir un bit recesivo y la transmisión de un segundo bit recesivo al inicio de cada EF es una medida necesaria para evitar que los contadores de errores TEC y REC del controlador CAN adyacente se incrementen de forma descontrolada. Si el AEF no incluyera estos bits recesivos entre los EFs, el controlador identificaría los bits dominantes subsiguientes como errores primarios y se penalizaría según las reglas de contención de errores del protocolo CAN [ISO93].

## 5.3. Diseño: topología estrella

### 5.3.1. Modificaciones en el diseño

Como se ha dicho en el apartado anterior, en un principio el diseño del AEFT se realizó para topologías bus. Posteriormente se realizaron las modificaciones pertinentes para su adaptación a las topologías estrella.

En primer lugar se tuvo que relocalizar el AEFT dentro de la estrella. En las Figuras 5.6 y 5.7 se muestran las dos configuraciones propuestas. En la primera configuración, el AEFT es acoplado directamente al controlador CAN siendo su localización idéntica a la de la topología bus. En la segunda configuración, el AEFT es integrado en el *hub*. Esta última configuración conlleva varias ventajas ya que únicamente se necesita un módulo AEFT independientemente del número de nodos (en la Configuración 1 se necesitan tantos AEFTs como nodos) y, además, éste tendrá una visión privilegiada del sistema y acceso a todos los recursos del *hub*.

Al ser el diseño del AEFT para la Configuración 1 idéntico al especificado en 5.2, a partir de ahora todas las modificaciones descritas únicamente harán referencia a la Configuración 2.

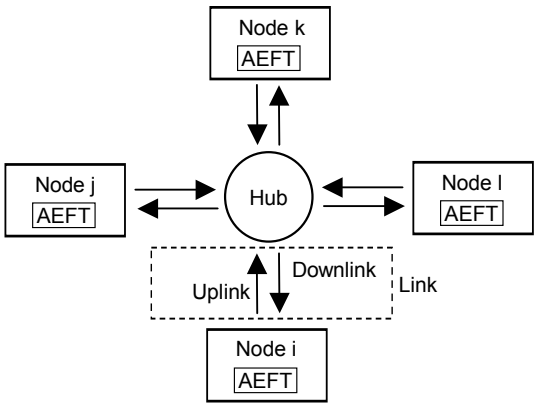



Figura 5.6: Localización del AEFT en la extrella (configuración 1)

El diseño propuesto en estas páginas se basa en la integración del AEFT con ReCANcentrate (ver 3.4.1). Por tal motivo, a la hora de realizar el diseño, en todo momento se tuvo en cuenta la arquitectura de ReCANcentrate y, sobretodo, la estructura interna del *hub*.

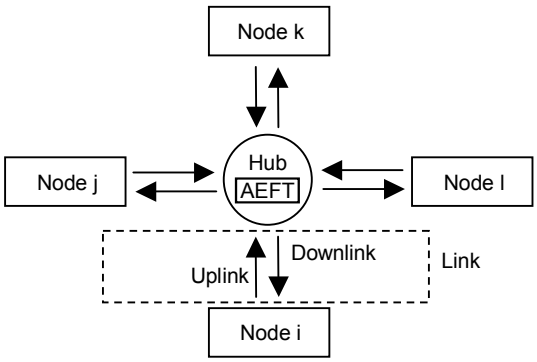



Figura 5.7: Localización del AEFT en la extrella (configuración 2)

La localización exacta del AEFT dentro del *hub* se muestra en la Figura 5.8. El AEFT se integra en el módulo de acoplamiento (*Coupler Module*), después de la primera puerta *AND* que acopla las contribuciones de los diferentes nodos. La salida de esta puerta, la señal $B_0$, es tomada por el AEFT como entrada. En este caso, pues, no se monitoriza la señal $Tx$ del controlador adyacente tal como se hacía en la topología bus, sino que se monitoriza la señal acoplada. La señal de salida del AEFT es conducida a una segunda puerta *AND* conjuntamente con la señal $B_0$.

Como se puede observar, la estructura interna del AEFT no ha tenido que ser modificada. Únicamente las señales de entrada y salida han cambiado. Esto pone en evidencia la ortogonalidad del diseño, un requisito importante desde el punto de vista de la flexibilidad.

### 5.3.2. Obsevaciones

Los escenarios de inconsistencia con dos errores de canal identificados en 3.3.1 y debidos a la señalización inconsistente de errores únicamente se estudiaron para topologías bus. Un bus está compuesto por una única línea de transmisión a la que acceden todos los nodos. Por consiguiente, un fallo físico puede afectar la línea, la conexión del nodo a la línea y el *transceiver*. En la estrella ReCANcentrate, en cambio, la conexión de los nodos al *hub* es realizada mediante dos líneas: el *uplink* (UL) y el *downlink* (DL). Así pues, para el caso de ReCANcentrate, un fallo puede afectar tanto el *uplink* como el *downlink* (incluyéndose los *transceivers* correspondientes). Esta peculiaridad hace necesario revisar los escenarios de inconsistencia identificados en 3.3.1.

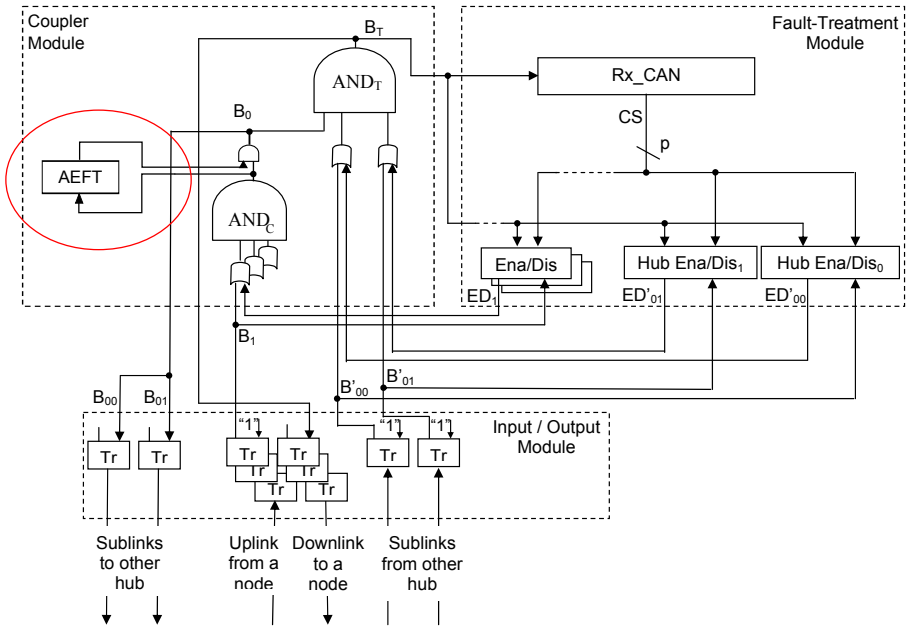



Figura 5.8: Integración del AEFT con ReCANcentrate

A continuación se detallan los diferentes escenarios de error que pueden llevar a un IMO (*Inconsistent Message Omission*):

- Escenario 1: un primer error afecta el DL del receptor X y, seguidamente, un segundo error afecta el DL del transmisor de manera que éste no detecta el *error flag* transmitido por el receptor X. Mientras que el receptor X rechazará la trama recibida, el resto de receptores la aceptarán y el transmisor no retransmitirá. Se crea un IMO que es resuelto por el AEFT ya que éste sí ha detectado el *error flag* transmitido por el receptor X.

- Escenario 2: un primer error afecta el UL del transmisor y un segundo error afecta el DL de ese mismo transmisor. Todos los receptores detectan el error y transmiten, simultáneamente, su *error flag*. Sin embargo, el transmisor no detecta el *error flag* y, consecuentemente, no retransmite la trama afectada. Se crea un IMO que es resuelto por el AEFT ya que éste sí ha detectado el *error flag* transmitido por los receptor.

- Escenario 3: un primer error afecta el DL del receptor X y un segundo error afecta el UL de ese mismo receptor X. El *error flag* transmitido es enmascarado por el segundo error y no llega a ser detectado ni por los receptores ni por el

AEFT. Se produce un IMO que el AEFT no es capaz de resolver.

- Escenario 4: un primer error afecta el UL del transmisor por lo que todos los receptores detectan un error y proceden a la transmisión de un *error flag*. Un segundo error afecta a los ULs de todos los receptores y el *error flag* queda desapercibido por el transmisor y por el AEFT. Aunque este escenario sea poco probable, se produce un IMO que no puede ser resulto por el AEFT.

Revisando los escenarios de error, se han identificado situaciones específicas en las que el AEFT, integrado en el hub, no es capaz de resolver las inconsistencias aparecidas. Aunque la inclusión del AEFT en el hub de la estrella conlleve ventajas interesantes, también implica una distanciación de la idea básica del AEFT. Se monitoriza la señal acoplada en lugar de la contribución de un controlador adyacente por lo que el manejo de errores puramente locales se ve comprometido.

Para resolver los nuevos escenarios de inconsistencia se debe recurrir a la configuración 1 (Figura 5.6) en la que se acopla un AEFT a cada nodo del sistema. Esta configuración puede resultar más costosa ya que para un sistema de $n$ nodos se necesitarán $n$ AEFTs. Sin embargo, la cobertura será máxima.

La elección de una configuración u otra dependerá de la aplicación, siendo la configuración 1 recomendable para sistemas con elevada garantía de funcionamiento y la configuración 2 para sistemas con moderada garantía de funcionamiento.

## 5.4. Implementación: topología bus

### 5.4.1. Introducción

En apartados anteriores se han descrito la motivación, las opciones de diseño y la arquitectura interna del AEFT. En este apartado se detalla la implementación del mismo para topologías bus. El módulo construido es un prototipo que incluye todos los mecanismos descritos en el apartado 5.2 y que permitirá la verificación experimental del mecanismo de resolución de inconsistencias propuesto.

Este apartado se organiza de la siguiente forma: primeramente se explican la plataforma de desarrollo y el entorno de programación utilizados para la implemen-

tación. Luego se describe el trabajo preliminar realizado, consistente en la migración, modificación y configuración de un controlador CAN en lenguaje VHDL. Dicho controlador, implementado sobre una FPGA, servirá de infraestructura para el módulo AEFT tal como se ha explicado en 5.2. Seguidamente se desarrolla la construcción del prototipo AEFT y, finalmente, se exponen los resultados obtenidos durante la verificación experimental.

### 5.4.2. Plataforma de desarrollo y entorno de programación

#### Conceptos básicos

La implementación del AEFT se realizó mediante VHDL, un lenguaje de descripción de hardware para circuitos integrados de muy alta velocidad (*VHSIC Hardware Description Language*).

VHDL es un lenguaje definido por el IEEE (*Institute of Electrical and Electronics Engineers*) [IEEE93] usado para describir circuitos digitales. Permite documentar las interconexiones y el comportamiento de un circuito electrónico, sin utilizar diagramas esquemáticos. Un rasgo importante es la independencia del hardware y la modularidad o jerarquía, es decir, una vez hecho un diseño, éste puede ser usado dentro de otro diseño más complejo y con otro dispositivo compatible.

Aunque puede ser usado de forma general para describir cualquier circuito se usa principalmente para programar dispositivos lógicos programables tales como PLDs (*Programable Logic Devices*), FPGAs (*Field Programmable Gate Arrays*) y ASICs (*Application Specific Integrated Circuits*).

En este trabajo se usaron exclusivamente FPGAs, dispositivos semiconductores que pueden reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica hasta funciones combinacionales complejas. Las FPGAs se utilizan en aplicaciones similares a los ASICs sin embargo son más lentas, tienen un mayor consumo de potencia y no pueden abarcar sistemas tan complejos como ellos. A pesar de esto, las FPGAs tienen las ventajas de ser reprogramables (lo que añade una enorme flexibilidad al flujo de diseño), sus costes de desarrollo y adquisición son mucho menores para pequeñas cantidades de dispositivos y el tiempo de desarrollo es también menor [WIS09].

La arquitectura más común de FPGAs contiene bloques de lógica (*logic blocks*), puertos de entrada/salida y canales de enrutamiento con múltiples líneas (Figura 5.9). En general, un bloque lógico está formado por varias celdas lógicas (*logic cells*). En la Figura 5.10 se muestra la arquitectura simplificada de una celda lógica, formada por una LUT (*Lookup Table*) de cuatro entradas, un sumador completo (FA, *Full Adder*) y un *flip-flop* tipo D. La LUT de cuatro entradas está formada por dos LUTs de tres entradas y un multiplexor. La celda presenta dos modos de operación: el modo normal y el modo aritmético. El modo de la celda es seleccionado mediante un segundo multiplexor. Finalmente, la salida puede ser síncrona o asíncrona dependiendo de la configuración del tercer multiplexor.

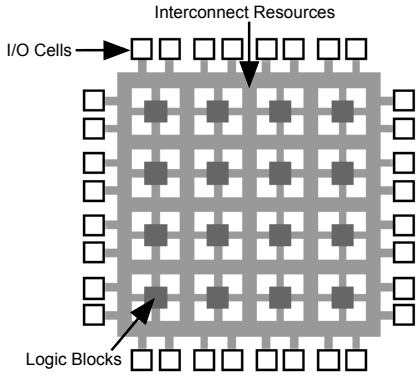



Figura 5.9: Arquitectura interna de una FPGA

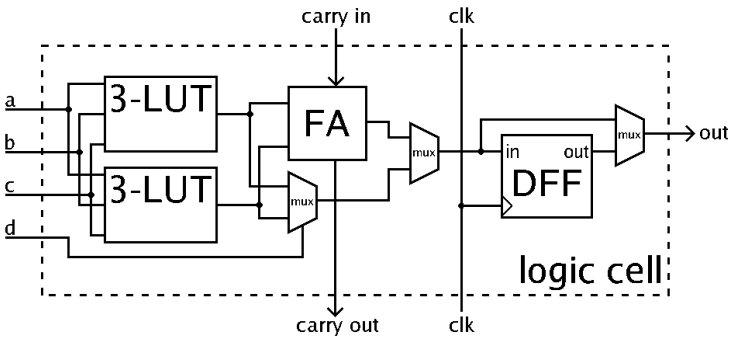



Figura 5.10: Ejemplo simplificado de una celda lógica

El enrutamiento, es decir, la interconexión de los diferentes bloques lógicos es realizado mediante interruptores programables (*programmable switches*) que pueden unir hasta cuatro líneas de forma independiente (Figura 5.11).

Los mayores fabricantes de FPGAs son las empresas *Xilinx* y *Altera* [SEE08].

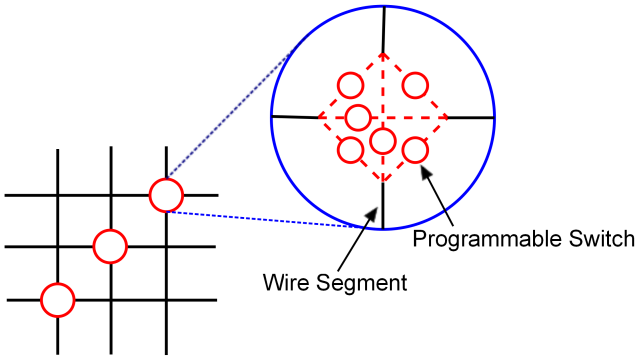



Figura 5.11: Interruptores programables

Ambos proporcionan software de diseño específico para sus productos.

### Plataforma de desarrollo

La plataforma de desarrollo utilizada para la implementación física del diseño es una placa de entrenamiento del fabricante XESS Corp. (*X Engineering Software Systems Corporation*), concretamente el modelo *XSA-3S1000* (Figura 5.12). Incorpora una FPGA *Spartan-3 XC3S1000* del fabricante líder *Xilinx* con una densidad lógica de 1 millón de puertas lógicas [XE04].

La FPGA es combinada con una memoria síncrona DRAM de 32 Mbytes y una memoria Flash de 2 Mbytes para máxima flexibilidad. Adicionalmente incluye un CPLD del tipo *XC9572XL*, un puerto paralelo para su programación, un puerto PS/2 para la conexión de un ratón o teclado, un puerto VGA, un visualizador de siete segmentos LED, 2 pulsadores, 4 interruptores DIP (*Dual In-line Package*), un interfaz de entrada/salida de 65 pines y un regulador de tensión de alimentación. El esquema de interconexionado de los diferentes módulos se muestra en la Figura 5.13.

### Entorno de programación

Para la programación de la FPGA en base a los ficheros VHDL se hizo uso del entorno de desarrollo *ISE Design Suite 13* de Xilinx. Es un entorno de programación que permite analizar y sintetizar los diseños en lenguaje VHDL. Incluye herramientas

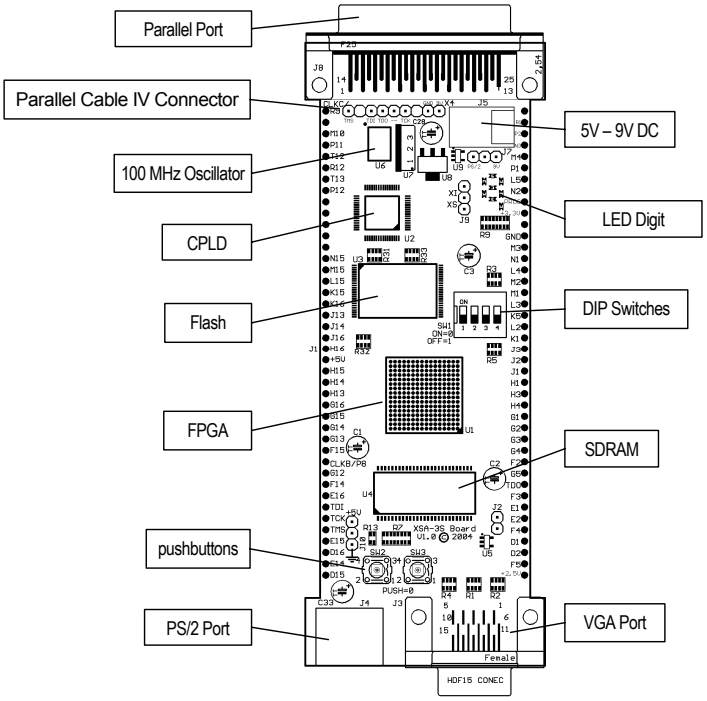



Figura 5.12: Placa de entrenamiento utilizada para la implementación

de compilación y simulación para realizar análisis temporales, examinar diagramas RTL (*Register-Transfer Level*), simular el comportamiento del diseño ante diferentes estímulos y configurar el dispositivo mediante el programador. Adicionalmente, la creación de un proyecto permite definir una jerarquía para los diferentes ficheros VHDL que componen el diseño.

La edición utilizada es la *ISE Web Edition*, una versión gratuita que únicamente proporciona soporte a un número limitado de dispositivos Xilinx. Esta versión fue totalmente suficiente ya que soporta completamente la familia de FPGAs *Spartan*.

En la Figura 5.14 se muestra la pantalla principal del entorno ISE. En la parte superior izquierda se muestran las fuentes del proyecto, es decir, los ficheros VHDL que componen el diseño. En la parte inferior izquierda se especifican los procesos en curso (procesos de síntesis, traducción, mapeado, enrutamiento, compilación, etc.). Finalmente, en la derecha se muestran los ficheros VHDL abiertos para su modificación o, como en este caso, el resumen de diseño (*design summary*) en el que se refleja, una vez compilado el diseño, información sobre el estado del proyecto y la

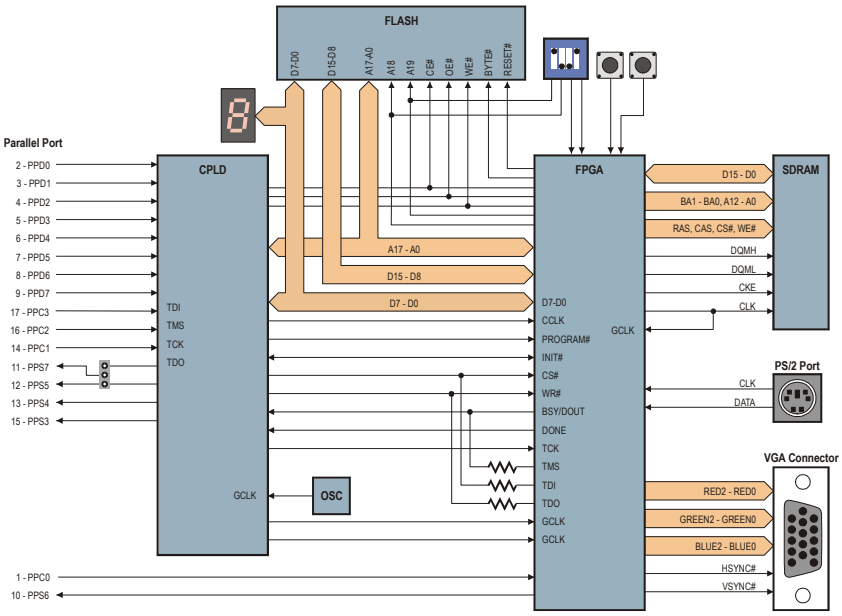



Figura 5.13: Esquema interno de la placa de entrenamiento

utilización del dispositivo.

### 5.4.3. Trabajo preliminar

#### Migración del controlador CAN

Para la implementación y posterior verificación experimental del AEFT fue necesario disponer de un controlador CAN. Para ello se recuperó un controlador CAN diseñado en lenguaje VHDL que, en su día, fue desarrollado como parte de un proyecto final de carrera. Desde entonces, su diseño fue revisado y modificado dos veces en [RODR03c] y [ROC08]. Ambas modificaciones tuvieron como objetivo el soporte de un sistema de sincronización de relojes sobre CAN.

Aunque diseñado en VHDL, el controlador CAN se implementó para dispositivos *Altera*, concretamente para la FPGA *EP1C20F324C7* de la familia *Cyclone*. Para ello se hizo uso del compilador de *Quartus II v7.0*, entorno de programación exclusivo de dicho fabricante. Al no ser compatibles los compiladores de Xilinx y Altera, el primer paso para poder usar el controlador CAN fue su migración al entorno propio de Xilinx.

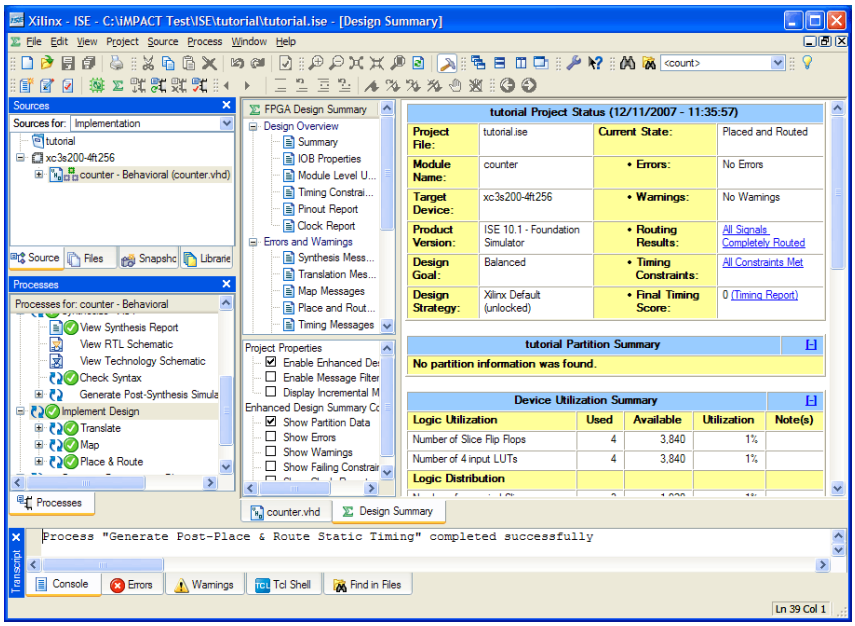



Figura 5.14: Entorno de programación ISE

La migración del controlador conllevo diferentes problemas de compilación que se tuvieron que solucionar.

En la figura 5.15 se muestra la organización jerárquica de los diferentes módulos que componen el controlador CAN. El nivel superior está compuesto por cinco módulos: *Nivel_Físico, Nivel_MAC, Nivel_LLC, Control_sistema y Supervisor_CAN*. A continuación se muestran las funciones que desempeña cada uno:

- *Nivel_Físico*: implementa la capa física de CAN. Incorpora dos submódulos, el BRP (*Baud_Rate_Prescaler*) que genera la señal de reloj del sistema a partir de la señal de reloj del oscilador y, el *Sincronizador* que implementa el mecanismo de sincronización propio de CAN: genera las señales de reloj de recepción y transmisión que indican cuándo el controlador debe muestrear el canal y cuándo transmitir, respectivamente.

- *Nivel_MAC*: implementa el control de acceso al medio. Está formado por diferentes submódulos:

  - *Receptor_CAN*: incluye la pila de recepción (*Pila_Recepción*) formada por dos memorias RAM (*RxRAM0* y *RxRAM1*) para almacenar temporal-

mente los bits recibidos, el módulo de recepción (*Receiver*) que, a partir de los bits recibidos, genera diferentes señales de control que informan al resto de módulos sobre el estado actual de la trama que es recibida y; el *acceptance filter* (*Accfilter*), un filtro que implementa el mecanismo de aceptación de mensajes de CAN.

- *Transmisor_CAN*: incluye la memoria RAM para almacenar los bits que van a ser transmitidos (*TxRAM*) y el módulo de transmisión (*TCAN*) cuya función es construir la trama a partir de los datos almacenados en la memoria y transmitirla bit a bit. *Errores_MAC*: implementa los mecanismos de control de errores de CAN y transmite un *error flag* en caso de error.
- *Comparador_CAN*: compara las señales de transmisión y recepción. Detecta errores de bit y pérdidas de arbitraje.
- *CRC_Sequence*: construye las secuencias de CRC a partir de las tramas.
- *Bit_Stuffing*: implementa la regla de *stuff*.

- *Nivel_LLC*: implementa el control de flujo de CAN. Incorpora el módulo denominado *LLC_Recovery* que despacha las peticiones de transmisión de niveles superiores y el módulo *OverloadLLC* que controla la transmisión de tramas de sobrecarga.

- *Control_Sistema*: controla todas las acciones realizadas. El módulo *Control* conoce el estado actual del sistema gracias a las señales recibidas del resto de módulos y activa las señales de control pertinentes para el correcto funcionamiento del controlador CAN. *Intermission* supervisa la transmisión del campo IFS, *Suspend_Trans* suspende transmisiones en proceso y *Control_Reset* controla el reseteo del sistema.

- *Supervisor_CAN*: formado por un único submódulo, *Error_Confi*. Implementa el mecanismo de contención de errores de CAN. Para ello incrementa/decrementa los contadores TEC y REC en caso de errores y actualiza el estado actual del controlador (recuérdese que hay tres estados: estado activo, estado pasivo o *bus-off*).

Finalmente, en *Config_CAN* se definen los diferentes parámetros de configuración del controlador.

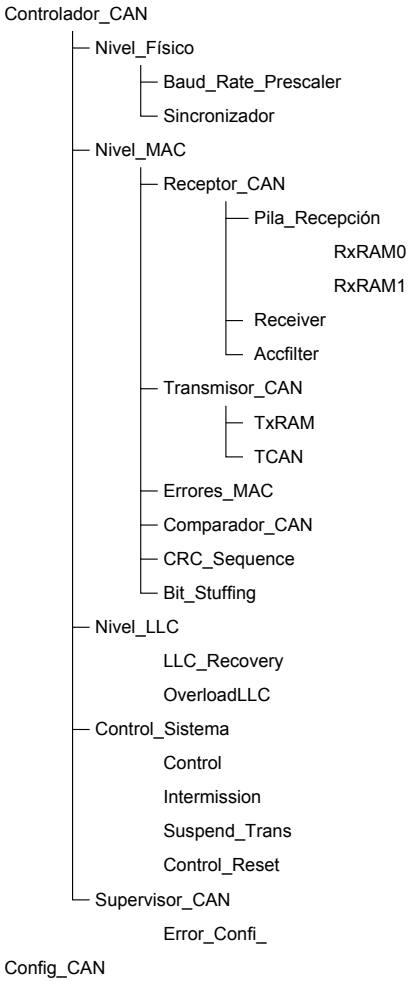



Figura 5.15: Organización jerárquica del controlador CAN

El primer problema tuvo su origen en las memorias RAM. El diseño de este tipo de memorias mediante VHDL es complejo y conlleva muchas líneas de código. Por tal motivo, los entornos de programación suelen integrar herramientas de generación avanzadas. En *Quartus II*, el entono propio de Altera, este tipo de asistentes de configuración se denominan megafunciones (*megafunctions*). El compilador automáticamente implementa la función en celdas lógicas.

Para la generación de las RAMs síncronas necesarias para los *búffers* de transmisión y recepción, se hizo uso de la megafunción *lpm_ram_dq*, una función parametrizable con puertos de entrada y salida independientes. Desgraciadamente, al migrar el diseño de Altera a Xilinx, las megafunciones deben ser reemplazadas por las he-

rramientas propias de Xilinx ya que no son compatibles. En Xilinx, la generación de memorias se basa en un constructor denominado *LogiCORE IP Block Memory Generator*, incluído en el entorno de desarrollo *ISE Design Suite*. Para generar memorias RAM idénticas a las usadas en Altera, se optó por RAMs síncronas de puerto único (*single-port RAMs*), con direccionamiento de 4 bits y datos de 1 Byte (8 bits). El código VHDL correspondiente puede ser consultado en el *Apéndice A*, página 166.

Un segundo problema resultó de las señales de reset. En el diseño inicial los resets se activaban por nivel alto, es decir, un *'1'* lógico. Sin embargo, las FPGAs utilizadas en este trabajo incluyen pulsadores por nivel bajo, esto es, la señal que proporcionan está permanentemente a '1' y se pone a '0' al pulsar. Así pues, para poder utilizar los pulsadores para resetear el sistema se tuvo que modificar el diseño. La modificación fue simple y consistió en una única línea de código:

```
xreset <= not reset;  --The reset is negated
```

Seguidamente, se tuvo que realizar la asignación de pines de entrada y salida. Para ello se generó un fichero específico encargado del enrutamiento de las señales de entrada/salida a los diferentes pines de la FPGA. Como último paso, se depuró el código para eliminar algunos errores de sintaxis.

### Configuración del controlador CAN

Una vez migrado el controlador CAN, éste tuvo que ser configurado para realizar la verificación experimental. En primer lugar se realizó la configuración de los parámetros relacionados con el tiempo de bit (*bit time*, $t_{bit}$). Adicionalmente, se diseñó una máquina de estados para configurar el controlador en modo transmisor, cargar los datos correspondientes y coordinar la transmisión de éstos.

El tiempo de bit se compone por cuatro segmentos (Figura 5.16) y se define como la suma de éstos (Ecuación 5.1)

$$t_{bit} = t_{SyncSeg} + t_{PropSeg} + t_{PS1} + t_{PS2} \tag{5.1}$$

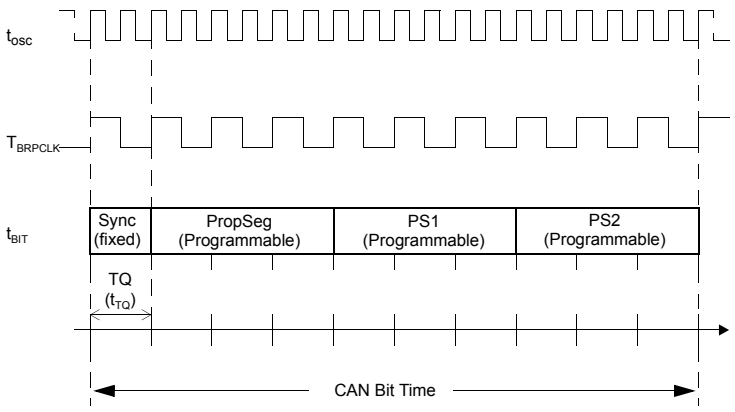



Figura 5.16: Tiempo de bit

El segmento de sincronización (*synchronization segment*) sirve para sincronizar los nodos con el bus, el segmento de propagación (*propagation segment*)compensa posibles retrasos entre nodos y los segmentos de fase 1 y 2 (*phase segments 1 and 2*) compensan errores de fase en el bus. Para ello se alarga el segmento de fase 1 o se acorta el segmento de fase 2. El tiempo de bit se define como:

Adicionalmente se define el BRP (*Baud-Rate Prescaler*), mediante el cual se escala el periodo del oscilador obteniéndose el denominado *Time Quantum* o TQ (ver Ecuación 5.2). A partir de este nuevo periodo se fija la longitud de los diferentes segmentos del tiempo de bit. Finalmente se define el SJW (*Synchronization Jump Width*), utilizado para mantener la sincronización inicial de los nodos receptores. Los parámetros descritos se configuraron para obtener una velocidad de transmisión (*bit rate*) de 250kbit/s (ver el código en el *Apéndice A*, página 166).

$$TQ = 2 \cdot BRP \cdot T_{OSC} = \frac{2 \cdot (BRP + 1)}{F_{OSC}} \tag{5.2}$$

El mecanismo para configurar el controlador en modo transmisor, cargar los datos deseados y coordinar la transmisión de éstos, se compone por dos módulos cuyas máquinas de estados se muestran en la Figura 5.17). La unidad de control (a) genera las señales de control para la unidad de carga (b). La puesta en marcha de la unidad de control viene condicionada a la señal *charge_msg* que es controlada desde el exterior mediante un pulsador. Una vez iniciada la máquina de estados, en el estado S1 se resetean las señales *byte* y *dir*. Seguidamente, en S2 se activa la

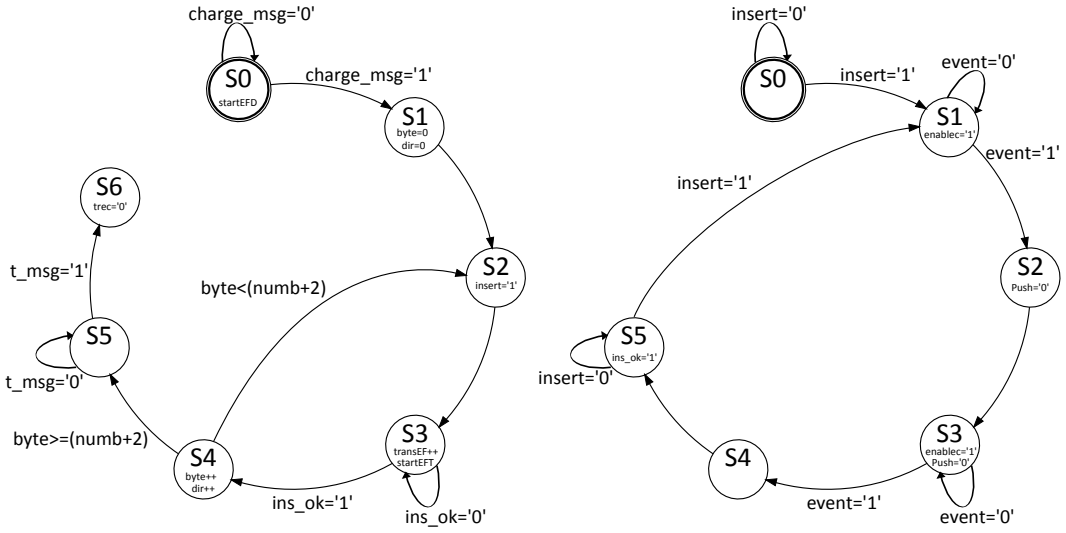
señal de control *insert* que pone en funcionamiento la unidad de carga encargada de cargar el primer byte en el *buffer* de transmisión del controlador CAN. La transición del estado S3 al estado S4 viene condicionada a la señal *ins_ok* que es activada por la unidad de carga una vez finalizada la carga del byte. A continuación, se compara el número de bytes cargados (*byte*) con el número total de bytes a cargar ($numb+2$). La constante *numb* es definida por el usuario e indica el número de bytes de datos que se desean cargar. Los 2 bytes adicionales se corresponden con el identificador (11b), el RTR (1b) y el DLC (4). Si aún faltan datos por cargar se regresará a S2, sino se pasará a S5. En S5 se permanecerá hasta recibir la petición de transmisión *t_msg*. En S6, el estado final, se activa la señal *trec* que iniciará la transmisión de los datos.

La unidad de carga se ocupa de la correcta temporización de las señales de carga y se pone en marcha cuando la unidad de control activa la señal *insert*. En el estado S1 se activa un contador de flancos de subida mediante *enablec*. Cuando el contador ha contado dos flancos, se activa la señal *event* y se pasa al estado S2. En S2 y S3 se realiza la carga del byte especificado en la dirección especificada del *buffer* de transmisión del controlador CAN. Para ello se activa la señal *Push*. Esperados otros dos flancos de subida, se salta al estado S5 en el que se notifica a la unidad de control que el byte ha sido cargado mediante la señal *ins_ok*. Cuando se active nuevamente la señal *insert*, se vuelve al estado S1 para cargar el siguiente byte.

El identificador, el RTR, el DLC y los bytes de datos de la trama que se desea transmitir se especifican como constantes modificables por el usuario. El código VHDL correspondiente se ha incluido en el *Apéndice A*, página 166.

### 5.4.4. Prototipo AEFT

En este apartado se describe la implementación física del diseño del AEFT propuesto en 5.2. La implementación se divide en dos tareas bien diferenciadas: la definición del diseño en VHDL y la construcción del prototipo.




(a) Unidad de control. (b) Unidad de carga

Figura 5.17: Máquinas de estado que controlan la configuración del controlador

### Implementación en VHDL

Como se ha visto en el apartado 5.2, la arquitectura del AEFT se compone por tres módulos: el AEFC, el EFD y el EFT. Cada uno de estos módulos viene definido por su propia máquina de estados. Consecuentemente, se generaron cuatro ficheros VHDL, tres para los módulos propiamente dichos (*AEFC.vhd*, *EFT.vhd* y *EFD.vhd*) y uno en un nivel jerárquico superior que engloba los anteriores (*TopAEFT.vhd*). Los códigos correspondientes están incluidos en el *Apéndice B*, en las páginas 167, 168, 169 y 170, respectivamente.

### Construcción del prototipo

El prototipo construido está formado por una red CAN de tres nodos. La conexión de los nodos se realizó de dos maneras: inicialmente se utilizó una *conexión lógica* basada en una puerta *AND* que desempeña la misma función que el bus físico (recuérdese que el bus CAN implementa una CAN 'cableada'). Posteriormente, para recrear las condiciones de funcionamiento específicas de un bus CAN se utilizaron *transceivers* y un cable de transmisión diferencial para la *conexión física* de los nodos.

Los nodos se componen por el controlador CAN, el AEFT y el inyector de fallos simple y están implementados en la FPGA *Xilinx Spartan-3 XC3S1000* de la placa *XESS XSA-3S1000*. La conexión lógica en forma de una puerta *AND* se implementó en una de las FPGAs de modo que no hizo falta ningún componente añadido (Figura 5.18). Para la conexión física, sin embargo, sí que se necesitó hardware adicional. En concreto, se utilizaron placas del tipo *wire-wrap* que, en su día, fueron diseñadas por el equipo investigador para su uso en ReCANcentrate. Estas placas, implementadas totalmente con componentes COTS, incorporan cuatro *transceivers* de alta velocidad del tipo *Philips PCA82C250* [PHI00]. Se utilizan dos parejas de *transceivers* ya que para conectar con los hubs se necesitan 2 *Uplinks* y 2 *Downlinks*, respectivamente (Véase el apartado 3.4.1). Adicionalmente, la placa incluye dos interfaces RJ45. El cable diferencial utilizado es del tipo UTP (*Unshielded Twisted Pair*), categoría 5/5e/6.

Para la construcción del prototipo únicamente hizo falta un *transceiver* por nodo por lo que las placas fueron reconfiguradas. En la Figura 5.19 se ilustra la configuración elegida. Solo el primer *transceiver* está conectado a la FPGA mientras los *transceivers* restantes son desconectados. La comunicación se realiza sobre la línea diferencial del *Uplink 1* como si de un bus CAN típico se tratara.

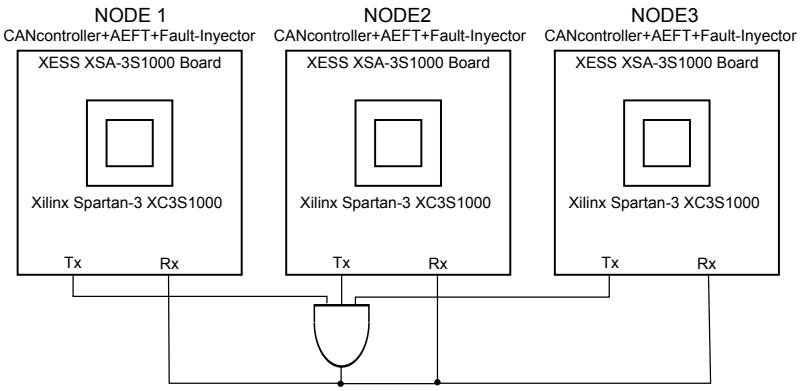



Figura 5.18: Interconexión de las FPGAs mediante una puerta *AND*

### 5.4.5. Verificación experimental

Una vez implementado el AEFT e integrado con el controlador CAN, se pasó a su verificación. Un primer test consistió en la simulación del diseño VHDL del AEFT

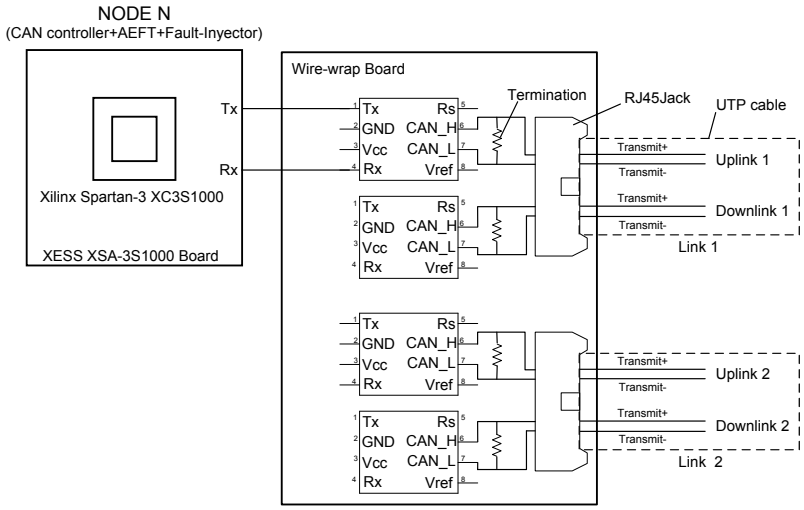



Figura 5.19: Interconexión de las FPGAs mediante transceivers

mediante la herramienta de simulación *Xilinx Isim ISE Simulator*. Se realizaron varias simulaciones en base a diferentes estímulos y, en todos los casos, la operación de las máquinas de estado que definen el AEFT así como su interacción fue correcta. Seguidamente ya se pasó a la verificación experimental de un prototipo construido especialmente para ello. En base a ese prototipo se realizaron distintas pruebas, observándose el comportamiento del AEFT ante diferentes situaciones. La prueba de mayor interés se basó en la generación de escenarios de inconsistencia para comprobar si el AEFT los resuelve y así demostrar el correcto funcionamiento del mecanismo.

En las siguientes secciones se describen las herramientas utilizadas para la verificación, las pruebas realizadas y los resultados obtenidos.

### Herramientas utilizadas

Para la verificación experimental del módulo AEFT hicieron falta diferentes herramientas. A continuación se enumeran las de mayor importancia:

- Un inyector de fallos simple para generar los escenarios de inconsistencia.
- Un osciloscopio digital para observar la respuesta de AEFT ante los escenarios de inconsistencia así como otras señales de interés.

- Una interfaz CAN para la comunicación entre los nodos y un PC. Permite el envío y la recepción de tramas por parte del PC.

Inicialmente se quiso utilizar sfiCAN para la inyección de fallos y recreación de escenarios de error. Sin embargo, sfiCAN únicamente es compatible con topologías en estrella. Por este motivo se decidió diseñar un inyector simple e incluirlo directamente en los nodos. Este inyector funciona de la siguiente forma: una vez detectado el SOF de la trama que es transmitida se pone en marcha un contador que cuenta el número de bits transmitidos. Cuando el contador alcanza un valor predeterminado, el inyector invierte el bit recibido en ese instante. De esta forma es posible generar errores locales en los diferentes nodos y crear escenarios mínimamente complejos.

En la Figura 5.20 se muestra la máquina de estados por la que viene definido el inyector. Presenta cuatro estados: *Idle* (estado inicial), *Count*, *Flip* y *Finish*. Se pasa del estado *Idle* al estado *Count* cuando se detecta un SOF, esto es, la primera transición recesivo - dominante. En *Count* se inicializa la variable *counter* y se incrementa según el reloj de transmisión ($clkT$). Alcanzado el valor prefijado por la constante *flipat*, la máquina de estados pasa al estado *Flip*. En *Flip* se activa la señal *flipbit* cuya misión es invertir el valor del bit recibido por $Rx$ en ese instante. Finalmente, la máquina de estados entra en el estado *Finish*.

El valor de la constante *flipat*, es decir, el número del bit de la trama que se desea invertir es buscado a priori para cada escenario de inconsistencia mediante el simulador CANfidant.

El osciloscopio digital utilizado es el modelo *DL7440* del fabricante *Yokogawa* [YO09]. Presenta 4 canales analógicos y 16 canales lógicos, un ancho de banda de 500Mhz y una velocidad de muestreo de 2GS/s. La característica más relevante de este instrumento es, sin embargo, su función de análisis de buses serie incluyendo CAN. Esta funcionalidad permite analizar una señal del bus e identificar automáticamente tramas y campos específicos, bits de *stuff* e incluso errores de transmisión. Es, por lo tanto, una herramienta extremadamente útil para la verificación experimental.

Adicionalmente se hizo uso de una interfaz CAN para integrar un PC en la red CAN y poder enviar tramas desde el PC a los nodos. La interfaz utilizada es una tarjeta *PCAN-PCI* de dos canales del fabricante *Peak System* [PS11]. Incorpora un

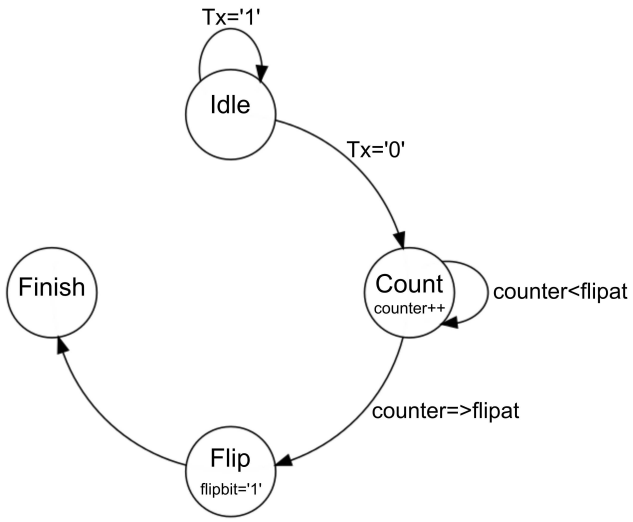



Figura 5.20: Máquina de estados del inyector de fallos simple

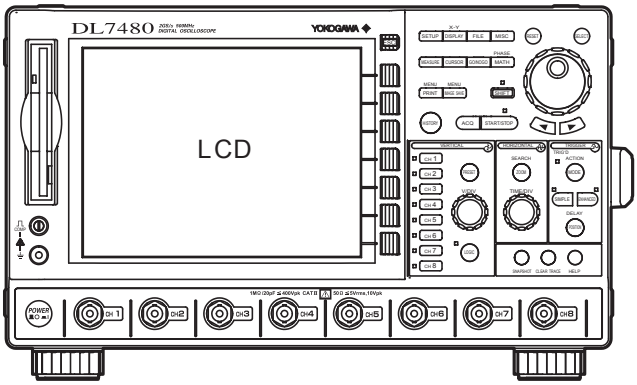



Figura 5.21: Panel frontal del osciloscopio digital

controlador CAN del tipo *SJA1000* y un transceiver *Phillips PCA82C251* [PHI00]. El control y monitorizado de la interfaz es realizado por la línea de comandos del PC.

### Pruebas realizadas y resultados obtenidos

El prototipo del AEFT ha sido extensamente testeado bajo diversas condiciones de error para verificar su correcto funcionamiento. En primer lugar se tuvo que validar la operatividad del controlador CAN ya que ésta ya no estaba garantiza tras su migración y modificación. Una vez verificado el buen funcionamiento del

controlador, se paso a la generación de los escenarios de inconsistencia con dos errores. Primeramente se analizó el comportamiento del sistema sin la presencia de los AEFTs y, seguidamente, se añadieron para ver si resolvían las inconsistencias. Se realizaron numerosas y variadas pruebas, sin embargo, en la tabla 5.1 se muestra un índice de las pruebas de mayor importancia.

| N° de prueba | Tipo de prueba | Tipo de conexión |
|---|---|---|
| 1 | Envío de ACK | Física |
| 2 | Envío trama sin datos | Lógica |
| 3 | Envío trama con datos | Lógica |
| 4 | Escenario de inconsistencia (1) | Física |
| 5 | Escenario de inconsistencia (2) | Física |
| 6 | Funcionamiento AEFT (1) | Física |
| 7 | Funcionamiento AEFT (2) | Física |
| 8 | Funcionamiento AEFT (3) | Física |
| 9 | Funcionamiento AEFT (4) | Física |

Cuadro 5.1: Pruebas realizadas.

La primera prueba consistió en observar si el controlador CAN (sin AEFT acoplado) respondía mediante un ACK a una trama genérica enviada desde la interfaz CAN del PC. Mediante el osciloscopio (configurado para un *bit rate* de 250kb/s) se monitorizaron las señales $Tx$ y $Rx$ del nodo. El resultado se muestra en la Figura 5.22 y es satisfactorio: el controlador CAN, en modo receptor, transmite bits recesivos durante la recepción de la trama y seguidamente, la acepta transmitiendo un bit de valor dominante en el campo de ACK.

Para la segunda prueba se utilizaron dos nodos (sin AEFTs), conectados entre ellos mediante una puerta *AND*. Un nodo se configuró en modo transmisor y el otro en modo receptor. El nodo transmisor se configuró según explicado en el apartado 5.4.3 para enviar una trama sin datos con identificador $007hex$. En la Figura 5.23 se ilustra el resultado. Tras apretar el pulsador correspondiente de la placa, el controlador comienza a transmitir correctamente la trama especificada.

La última prueba del controlador CAN consistió en la transmisión de un mensaje con datos. Se volvió a utilizar la configuración de los dos nodos conectados por una puerta *AND*. En este caso, ambos nodos se configuraron en modo transmisor. Uno de los nodos se configuró para enviar una trama con identificador bajo ($008hex$) y el otro para enviar una trama con identificador alto ($3FFhex$). De esta forma no

solo se pudo comprobar la correcta transmisión de datos sino también el funcionamiento del mecanismo de arbitraje. En la Figura 5.24 se muestran las señales $Tx$ (arriba) y $Rx$ (abajo) del nodo transmisor así como la señal que activa la transmisión. Adicionalmente se ilustran los detalles de la trama transmitida (ID, datos, CRC, y recepción de ACK). Como bien se puede ver, el mecanismo de arbitraje ha funcionado bien y la transmisión se ha realizado con éxito.

Con esto se concluye la validación del controlador y se pasa a la generación de los escenarios de inconsistencia.

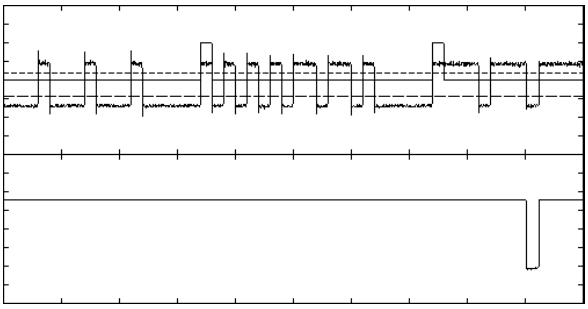


Figura 5.22: Respuesta del controlador mediante un ACK

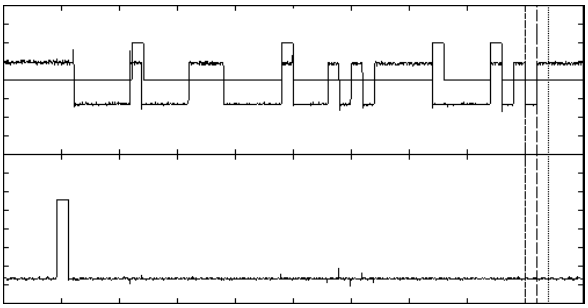


Figura 5.23: Envío de una trama sin datos

El siguiente paso de la verificación experimental fue la recreación de los escenarios de inconsistencia identificados en el capítulo 3.3.1. En base a las secuencias de CRC vulnerables y combinaciones generadoras calculadas en el estudio de relevancia del capítulo 4, se generaron diferentes escenarios con dos errores mediante el simulador CANfidant. Seguidamente se determinó la posición exacta de esos errores en la trama (incluyendo los bits de *stuff*). A partir de esa información se configuraron los inyectores de fallos de los diferentes nodos.

Para las pruebas realizadas se conectaron físicamente tres nodos con sus res-

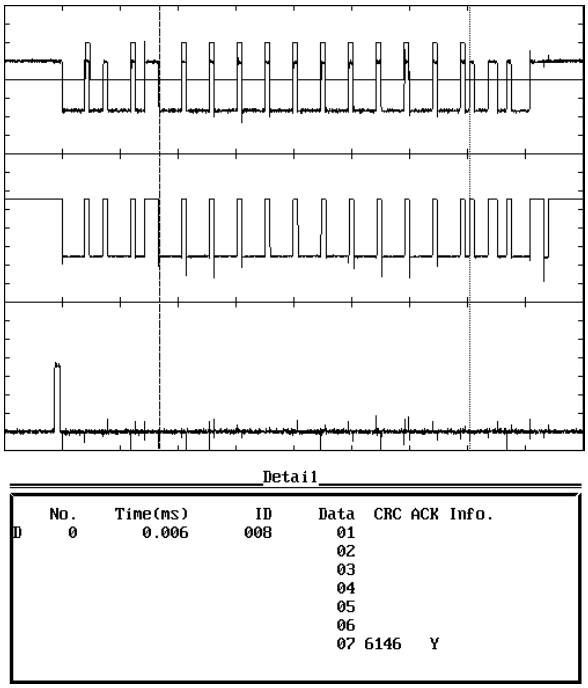



Figura 5.24: Envío de una trama con 7 bytes de datos

pectivas placas de entrada/salida. En la primera prueba se recreó un escenario de inconsistencia generado a partir de una trama con identificador $0dec$ y un byte de datos, $byte_0 = 162dec$. El primer error se genera en uno de los receptores, en la posición 39 de la trama. El segundo error se genera en el otro receptor y en el transmisor, en la posición 46 de la trama. El resultado se muestra en la Figura 5.25. La señal S1 se corresponde con la señal $Rx$ del transmisor, S2 con la señal $Rx$ del receptor X y S3 con la señal $Rx$ del receptor Y. En A se inyecta el primer error en S3 por los que el receptor Y detectará un error de bit de *stuff* en B. Seguidamente, el receptor Y transmite un *error flag* pero éste es enmascarado en C por un segundo error en el transmisor y el receptor X. Consecuentemente, el receptor Y rechazará la trama, el receptor X la aceptará y el transmisor no la retransmitirá, produciéndose la inconsistencia.

Un segundo escenario de inconsistencia se genera a partir de una trama con identificador $226dec$ y un byte de datos, $byte_0 = 0dec$. El primer error se produce en uno de los receptores, exactamente en la posición 37 de la trama. El segundo error se produce en el otro receptor y en el transmisor, en la posición 45 de la trama. Aunque la trama sea distinta y los errores se produzcan en posiciones diferentes, el

resultado (expuesto en la Figura 5.26) es idéntico al del caso anterior.

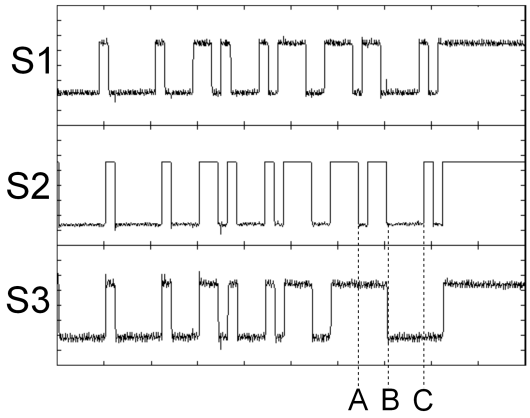



Figura 5.25: Escenario de inconsistencia con 2 errores (1)

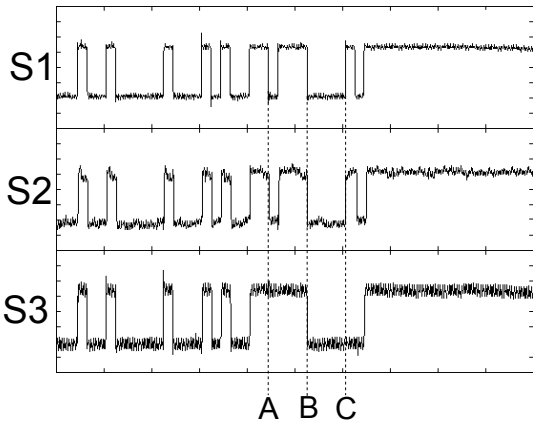



Figura 5.26: Escenario de inconsistencia con 2 errores (2)

En base a los resultados se ha demostrado que los escenarios de inconsistencia debidos a la señalización inconsistente de errores, hasta ahora solo tratados teóricamente, pueden crearse físicamente. En las siguientes pruebas se verificará si los AEFTs resuelven o no estas inconsistencias.

Para las pruebas restantes se incluyeron los AEFTs en los nodos. La configuración usada en pruebas anteriores se mantiene: tres nodos conectados físicamente mediante placas de entrada/salida. Las dos primeras pruebas se basan en los escenarios de inconsistencia descritos anteriormente. En la Figura 5.27 se muestra el resultado de la primera prueba. Las señales mostradas en las dos capturas son los mismas, siendo la primera una ampliación de la segunda. S1 se corresponde con la señal

*Rx* del transmisor, S2 con la señal *Rx* del receptor X y S3 con la señal *Rx* del receptor Y. El escenario es idéntico al de la Figura 5.26: en A se produce el primer error en S2 por los que el receptor X detectará un error de bit de *stuff* en B. Seguidamente, el receptor X transmite un *error flag* pero éste es enmascarado en C por un segundo error en el transmisor y el receptor Y. Sin embargo, el AEFT del nodo X sí ha detectado el *error flag* motivo por el cual inyecta un AEF en D. Este AEF ya sí es detectado por el transmisor y el receptor Y, haciendo que éstos también entren en estado de error y el error inicial sea globalizado. En E, los nodos finalizan la transmisión de sus AEFs, se resincronizan y la trama afectada puede ser reenviada. Con esto queda demostrado que el mecanismo basado en AEFTs ha resulto satisfactoriamente el escenario de inconsistencia.

Para el caso anterior, cada AEFT envía un AEF formado por cinco EFs (la constante $m$ se fijó a 5). Sin embargo, como se puede observar en la Figura 5.26, en total se han transmitido seis EFs. Esto es debido a que los AEFTs del transmisor y del receptor Y se han activado tras la detección del primer EF transmitido por el AEFT del receptor X, existe pues un desfase de un EF entre el nodo que ha detectado el error primario y el resto de nodos.

En la segunda prueba, aunque el escenario sea distinto, los resultados obtenidos fueron idénticos a los de la primera prueba. En la Figura 5.28 se expone este segundo escenario.

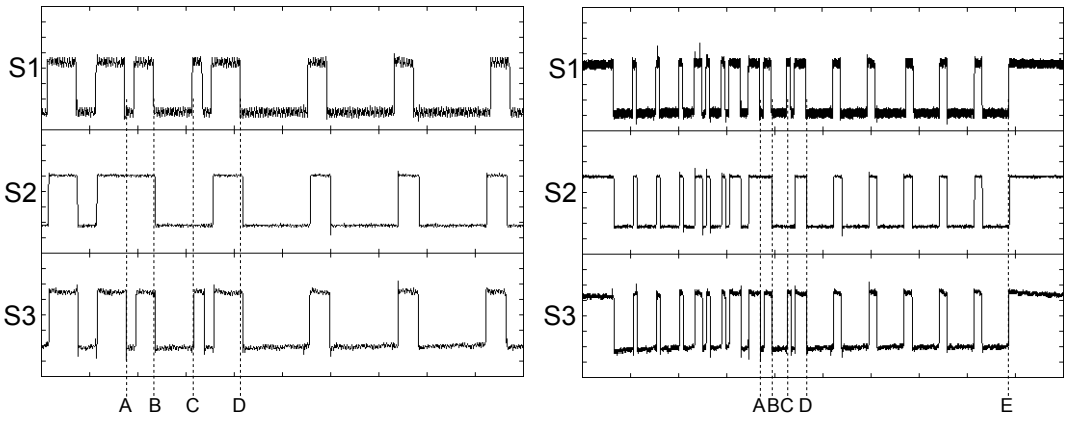



Figura 5.27: Escenario de inconsistencia (1) resuelto mediante el AEFT

Las dos tramas transmitidas en los dos escenarios anteriores presentaban las secuencias de CRC vulnerables $3B0_{hex}$ y $370_{hex}$, respectivamente. En las dos últimas

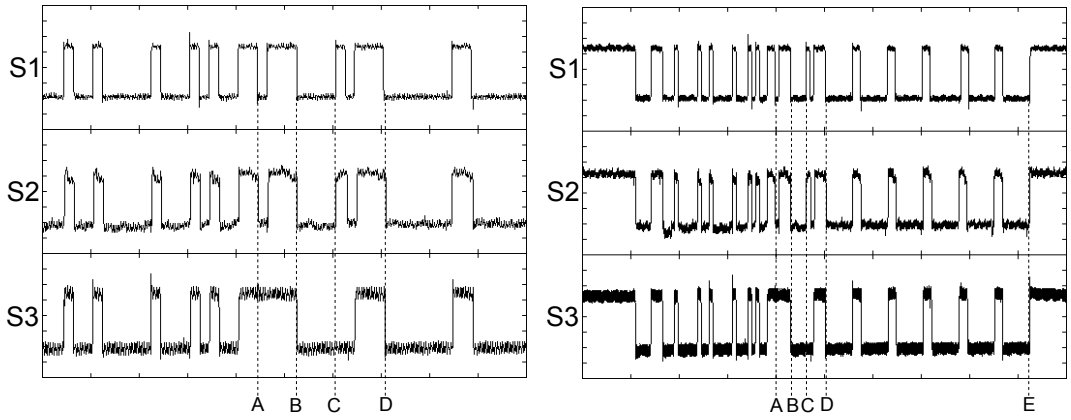



Figura 5.28: Escenario de inconsistencia (2) resuelto mediante el AEFT

pruebas se utilizaron tramas con secuencias de CRC vulnerables distintas a las anteriores para demostrar que los AEFTs son capaz de resolver cualquier inconsistencia, independientemente del formato del CRC.

En el escenario mostrado en la Figura 5.29 se transmite una trama con identificador $1333dec$ y $byte_0 = 138dec$. La secuencia de CRC vulnerable resultante es $3D0hex$. Los errores se producen en las posiciones 38 y 44 de la trama, respectivamente. Los AEFTs solucionan satisfactoriamente la inconsistencia potencial. Lo mismo ocurre en el caso de la Figura 5.30, en el que se transmite una trama con identificador $878dec$ y un byte de datos ($byte_0 = 1dec$) que generar la secuencia de CRC vulnerable $E00hex$. Los errores se producen en los bits 41 y 46, respectivamente.

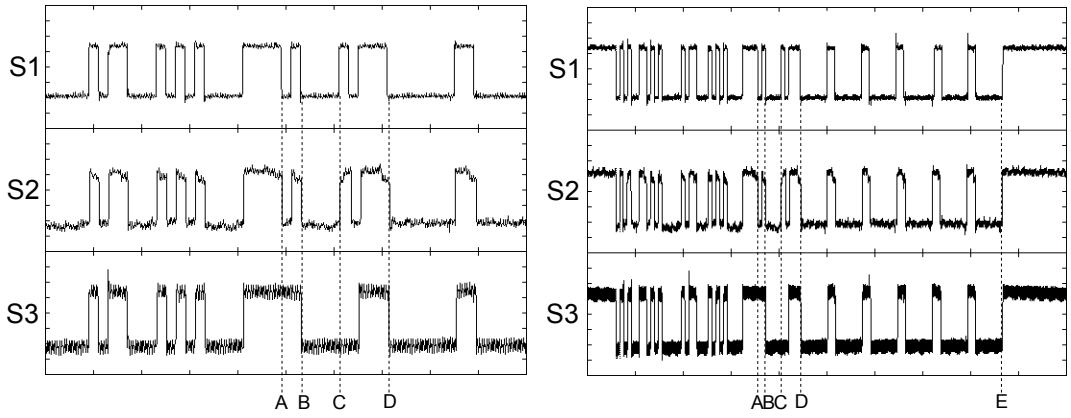



Figura 5.29: Escenario de inconsistencia (3) resuelto mediante el AEFT

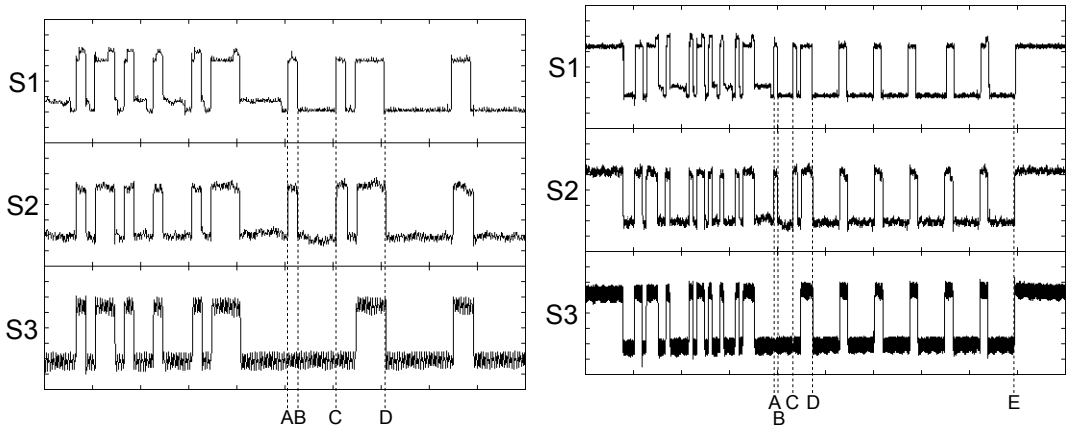



Figura 5.30: Escenario de inconsistencia (4) resuelto mediante el AEFT

Con esto se concluye la verificación experimental del AEFT para topologías bus. Se ha comprobado que el controlador CAN migrado funciona bien. Seguidamente se han reproducido varios escenarios de inconsistencia en base a las secuencias de CRC vulnerables y, considerando la producción de dos errores. Finalmente se han integrado los AEFTs en los diferentes nodos y se han repetido las pruebas. Al ser estas satisfactorias, queda demostrado que los AEFTs son un mecanismo eficiente de resolución de inconsistencias de este tipo.

## 5.5. Implementación: topología estrella

### 5.5.1. Introducción

En este apartado se detallan todos los aspectos relacionados con la implementación física del AEFT para topologías estrella. La implementación es bien diferente a la del apartado anterior y se basa, principalmente, en la integración del AEFT en ReCANcentrate.

La organización de este apartado es la siguiente: en primer lugar se describe la integración del AEFT en el hub de ReCANcentrate tal como se detalló en el apartado 5.3. En segundo lugar se explican las pruebas realizadas y los resultados obtenidos durante la verificación experimental.

### 5.5.2. Integración con ReCANcentrate

La integración del AEFT con el *hub* ReCANcentrate se realizó a nivel de código VHDL mediante el entorno de programación *ISE Design Suite* de Xilinx. Para ello, como ya se ha explicado en el apartado 5.3, se tuvieron que realizar ligeros cambios sobre los ficheros VHDL originales de ambos dispositivos. Hecho esto simplemente se incorporaron los ficheros VHDL del AEFT en la jerarquía de ficheros VHDL de ReCANcentrate y se interconectaron las señales de entrada y salida correspondientes.

El código, una vez compilado, se implementó sobre la misma FPGA que ya ha sido utilizada anteriormente: la FPGA *Xilinx Spartan-3 XC3S1000* de la placa de desarrollo *XESS XSA-3S1000*. Para la conexión del hub con los nodos se utilizó una placa dedicada (fabricada mediante la técnica de *wire-wrap*) que, en su día, fue diseñada para el primer prototipo de ReCANcentrate (Figura 5.31). Dicha placa incluye cuatro parejas de *transceivers* PCA82C250 y cuatro interfaces RJ45 (una para cada pareja de *transceivers*) para la conexión de hasta tres nodos y un interlink hacia el *hub* replicado. Los cables de transmisión son del tipo UTP, categoría 5/5e/6.

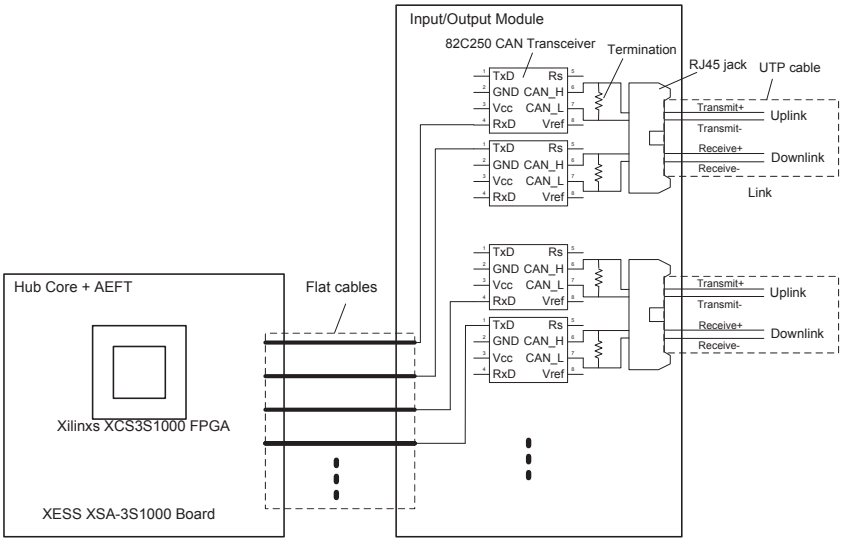



Figura 5.31: Conexionado del *hub*

Los nodos (Figura 5.32) están formados por un microcontrolador *PIC18FXX8* del fabricante *Microchip* [MIC04] que incorpora un controlador CAN y, dos pares de *transceivers* PCA82C250 (un par para cada hub, formando el *uplink* y el *downlink*).

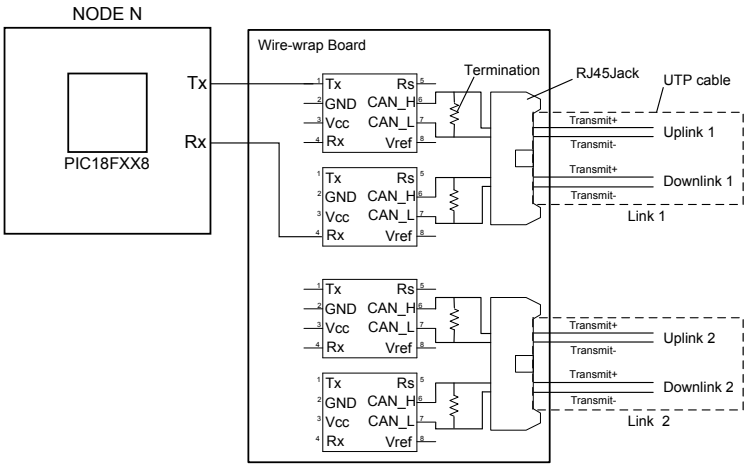



Figura 5.32: Conexionado de los nodos de la estrella

El prototipo construido se basa, fundamentalmente, en la segunda configuración propuesta durante el diseño para topologías en estrella (5.3). Recuérdese que en esta configuración, el AEFT está integrado en el *hub*. A efectos prácticos, la configuración 1 es idéntica a una topología bus. Esto es debido a que el hub de ReCANcentrate está diseñado para ser totalmente transparente desde el punto de vista de los nodos [BARR09].

### 5.5.3. Verificación experimental mediante sfiCAN

Una vez construido el prototipo se pasó a la verificación experimental de este. Para la generación de los escenarios de error se utilizó el inyector de fallos sfiCAN. Tal como se ha explicado en 3.4, sfiCAN está integrado en el *hub* de ReCANcentrate y es configurado por un PC conectado a la red mediante la interfaz PCAN.

#### Configuración de sfiCAN

SfiCAN es configurado por un PC mediante la línea de comandos. Nótese que el asistente de configuración ha sido desarrollado en base a *GNU/Linux* y, por lo tanto, la configuración de sfiCAN únicamente podrá realizarse desde un ordenador con este sistema operativo.

El primer paso para la configuración consiste en activar la interfaz PCAN. Para

ello se debe ejecutar el *script* `restartcan` localizado en `sfiCAN/PC/scripts/`. Dicho *script* puede ser ejecutado en cualquier momento para reinicializar los puertos CAN de la interfaz.

```
$ ./restartcan
```

Listing 5.1: Ejemplo de `restarcan`

Para visualizar en pantalla el estado actual de la interfaz se puede utilizar el comando `ip link`. El resultado del comando anterior se muestra en el listado 5.2. Como se puede observar, los puertos `can0` y `can1` están habilitados (`UP`).

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    qlen 1000
3: can0: <NOARP,UP,LOWER_UP> mtu 16 qdisc pfifo_fast state UNKNOWN qlen 10
    link/can
4: can1: <NOARP,UP,LOWER_UP> mtu 16 qdisc pfifo_fast state UNKNOWN qlen 10
    link/can
```

Listing 5.2: Ejemplo de `ip link`

Al iniciar sfiCAN, el modo de operación por defecto es el modo de configuración. Una vez habilitada la interfaz, se transmite el fichero de configuración (*fault-inyection specification*) al hub. Esta tarea es realizada por el *Fault-Inyector Configurator*, implementado mediante el *script* `fic`, localizado en `sfiCAN/PC/FIMS/fic/`. Presenta dos parámetros de entrada: el fichero de configuración, `-f=<file>`, y el puerto por el que se desea transmitir dicho fichero, `i=<iface>` (ver el listado 5.3).

```
$ ./fic -f=config.txt -i=can0
processing file ...
  3 group(s= found!
initializing 'can0' socket ...
checking file ...
  0 error(s) found!
processing file ...
Ending...
Done!
```

Listing 5.3: Ejemplo del *Fault-Inyector Configurator*

Una vez configurado sfiCAN, se debe cambiar el modo de operación. Como se ha explicado en el capítulo 3.4, existen cuatro modos: el modo de configuración, el modo de espera, el modo de ejecución y el modo denominado *wait-for-whistle*. Para iniciar un experimento tanto los nodos como el hub deben entrar primero en modo *wait-for-whistle* y, seguidamente, en modo ejecución. Para ello, se transmite en modo difusión un mensaje de *wait-for-whiste* seguido de un mensaje de *starting-whistle*. Para ello se utiliza el comando `cansend` (Listado 5.4).

```
$ cansend can0 000#22
$ cansend can0 000#23
```

Listing 5.4: Ejemplo de cambio de modos

Durante el experimento, mediante el comando `candump` se pueden visualizar en pantalla las tramas CAN recibidas por la interfaz. El resultado de este comando se muestra en el Listado 5.5. Adicionalmente, mediante el comando `cat /proc/pcan` es posible mostrar información sobre el *driver* de la interfaz (Listado 5.6). Especialmente el parámetro `status` es importante ya que indica si la interfaz se encuentra en estado de error (valor diferente a `0x0000`)o no (valor igual a `0x0000`). En caso de error se debe reinicializar la interfaz mediante `restartcan`.

```
$ candump can0
 can0 306 [1] CB
 can0 427 [8] 1D 87 51 73 74 BE 5D 4D
 can0 68E [4] 7E 99 F1 0C
 can0 4DC [8] C1 02 CA 37 D2 E5 21 2A
...
```

Listing 5.5: Ejemplo de `candump`

```
$ cat /proc/pcan
*------ PEAK-Systems CAN interfaces (www.peak-system.com) --------
*------------------ Release_20110912_n (7.4.0) --------------------
*n -type- ndev --btr- --read-- --write-- --irqs-- -errors- status
 0     pci can0 0x0014 00000f73 000001b5c 000028f9 0000009e 0x000c
 1     pci can1 0x0014 00000000 000000000 00000000 00000000 0x000c
```

Listing 5.6: Ejemplo de `/proc/pcan`.

Una vez finalizado el experimento, los NCCs deben volver al modo de configuración. Para ello se transmite, mediante la instrucción `cansend`, un mensaje de *enter-config-mode*, tal y como se muestra en el Listado 5.7.

```
$ cansend can0 000#20
```

Listing 5.7: Ejemplo de entrada en modo configuración.

La interfaz del PC actúa como un nodo cualquiera durante un experimento. Esto significa que transmitirá ACKs durante la recepción de tramas y *error flags* si detecta un error. Para evitar este comportamiento se debe hacer uso del modo *listen-only* del controlador CAN de la interfaz. En este modo, el PC seguirá recibiendo las tramas transmitidas por el canal pero no interactuará con el resto de nodos. Para habilitar y deshabilitar este modo de funcionamiento, el grupo de investigación desarrolló dos programas: `enable_listen_only` y `disable_listen_only`, respectivamente, situados ambos en `sfiCAN/PC/listen-only/`.

El modo *listen-only* debe ser habilitado justo despúes de iniciar el experimento, es decir, despúes de transmitir el mensaje de *starting-whistle*. La deshabilitación se debe efectuar justo antes de finalizar el experimento, es decir, antes de transmitir el mensaje de *enter-config-mode*.

Como parte de este trabajo se escribieron dos scripts: `init_exp` y `end_exp` que realizan la activación y desactivación del modo *listen-only* de forma automática.

Adicionalmente, tanto el hub como los nodos incluyen un registro (*log*) que guarda información relevante durante el experimento. Dicho registro está implementado mediante la instrucción `fil`, localizada en `sfiCAN/PC/FIMS/fil/` (Listado 5.8). La información almacenada por el *hub* es ligeramente diferente a la almacenada por los nodos. Así, el *hub* guarda las tramas recibidas por los nodos. Para cada trama se guardan:

- Si la trama se ha recibido correctamente (`Ok`) o si ha habido un error (`Er`).
- El identificador de la trama en formato hexadecimal.
- El valor del DLC (entre corchetes).
- El valor del primer byte de datos en formato hexadecimal.
- El puerto por el que se ha recibido la trama.

- En caso de error, el campo y el bit en el que el *hub* ha detectado el error.

En cambio, los nodos guardan las tramas recibidas y transmitidas así como el valor de los contadores de error en el caso de que cambien. Para cada trama se guardan:

- El número de la trama.
- Si la trama se ha recibido (rx) o transmitido (tx).
- El identificador de la trama en formato hexadecimal.
- El valor del DLC (entre corchetes).
- El valor del primer byte de datos en formato hexadecimal.

```
$ ./fil
initializing 'can0' socket ...
retrieving log data...
---------
-- HUB --
---------
Ok 030 [1] 00 port0
Er 030 [1] 01 port1 (eof(5))
error frame
Ok 030 [1] 02 port0
------------
-- NODE0 --
------------
01: tx 030 [1] 00
02: tx 030 [1] 01
03: tx 030 [1] 02
------------
-- NODE1 --
------------
01: rx 030 [1] 00
   TEC:000 REC:001
   TEC:000 REC:009
   TEC:000 REC:008
02: rx 030 [1] 02
   TEC:000 REC:007
------------
-- NODE2 --
------------
01: rx 030 [1] 00
02: rx 030 [1] 01
03: rx 030 [1] 02
```

Listing 5.8: Ejemplo del registro `fil`.

### Pruebas realizadas

En la primera prueba, se conectaron dos nodos al hub (sin el AEFT). Ambos nodos se programaron como receptores. Esta programación se realizó por USB mediante el programador *MPLAB ICD2* y el entorno *MPLAB ISE* de Microchip. Seguidamente se configuró sfiCAN para inyectar un error genérico en el *downlink* de uno de los nodos durante la transmisión del campo de datos de la segunda trama (el fichero de configuración utilizado se muestra en el Listado 5.9). Como último paso se envío por la interfaz CAN del PC un conjunto de tramas con identificador $ID = 333hex$ y un byte de datos de valor aleatorio. El resultado se muestra en la Figura 5.33. La señal S1 se corresponde con la señal $Rx$ del nodo X y las señales S2 y S3 con las señales $Tx$ y $Rx$ del nodo Y, respectivamente. En A, sfiCAN inyecta un error en el downlink del nodo Y. Este error es detectado por el nodo Y al realizar el *CRC check* y globalizado en B (el primer bit del EOF). La trama se rechaza consistentemente por ambos nodos y es retransmitida en C. Finalmente, la trama retransmitida es aceptada mediante ACKs en D. Con esto se ha verificado el correcto funcionamiento tanto de sfiCAN como del *hub*.

```
[fault injection 1]
value_type = inverse
target_link = port1dw
mode = single-shot
aim_filter = 0
aim_field = idle
aim_link = coupled
aim_count = 2
fire_field = data
fire_bit = 0
fire_offset = 0
cease_bc = 1
```

Listing 5.9: Fichero de configuración para la primera prueba.

La prueba anterior se repitió, esta vez con el AEFT incluído en el *hub*. El resultado de esta segunda prueba se expone en la Figura 5.34. En este caso, el *error flag* transmitido en B es detectado por el AEFT por lo que éste transmite su AEF en C. Se observa que el AEF se compone de cinco EFs tal como definido por el parámetro $m$. En D finaliza el envío del AEF y en E se procede a la retrasnmisión de la trama corrompida. El resultado es satisfactorio y prueba el buen funcionamiento del AEFT

integrado en el hub de la estrella ReCANcentrate. Nótese que el resultado obtenido a partir de los (*loggers*) del *hub* y de los nodos (mediante la instrucción `fil`) fueron idénticos a los obtenidos mediante el osciloscopio.

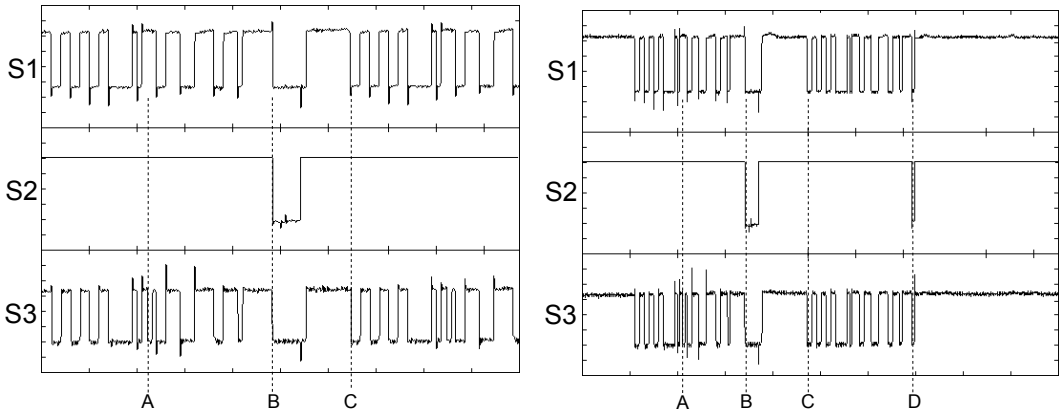



Figura 5.33: Error detectado y globalizado por los nodos

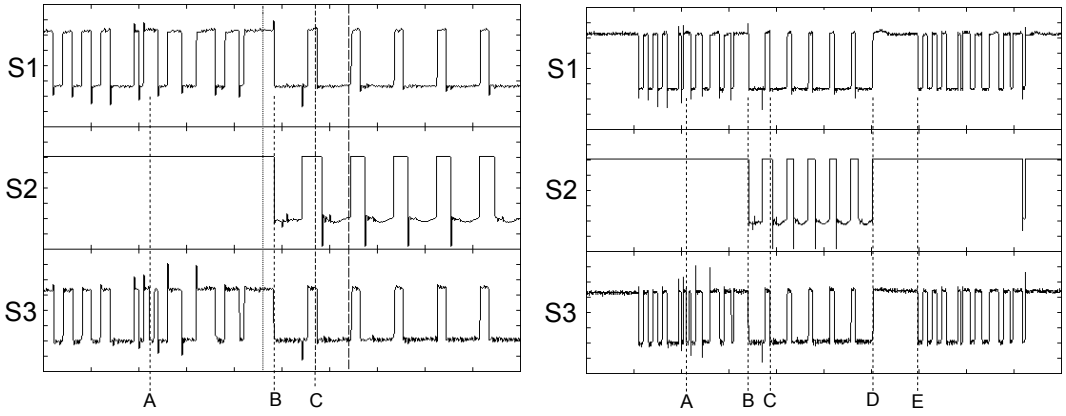



Figura 5.34: Error detectado por el AEFT integrado en el hub

Seguidamente se procedió a recrear el primer escenarios de inconsistencia. Para ello se conectaron tres nodos al hub , dos de ellos configurados como receptores (identificados como receptor X y receptor Y) y el tercero como transmisor. El transmisor se configuró (de nuevo mediante las herramientas de *Microchip*) para transmitir una trama con identificador igual a $1B9hex$ y un byte de datos igual a $A4hex$. Esta es una combinación generadora de la secuencia de CRC vulnerable $E00hex$. El inyector sfiCAN se programó para inyectar un primer error en el *downlink* del receptor Y, exactamente en el duodécimo bit del campo de CRC (un bit de *stuff*). El segundo

error se inyecta en los *downlinks* del transmisor y del receptor X, en el delimitador de CRC. El fichero de configuración se muestra en el Listado C

```
[fault injection 1]
value_type = inverse
target_link = port1dw
mode = single-shot
aim_filter = 0
aim_field = idle
aim_link = coupled
aim_count = 2
fire_field = crc
fire_bit = 10
fire_offset = 1
[fault injection 2]
value_type = inverse
target_link = port2dw
mode = single-shot
aim_filter = 0
aim_field = idle
aim_link = coupled
aim_count = 2
fire_field = crcDelim
fire_bit = 0
fire_offset = 0
[fault injection 3]
value_type = inverse
target_link = port0dw
mode = single-shot
aim_filter = 0
aim_field = idle
aim_link = coupled
aim_count = 2
fire_field = crcDelim
fire_bit = 0
fire_offset = 0
cease_bc = 1
```

Listing 5.10: Fichero de configuración para la segunda prueba.

En la Figura 5.35 se ilustra el resultado obtenido mediante el osciloscopio digital. S1, S2 y S3 son las señales de recepción ($Rx$) del transmisor, receptor X y receptor Y, respectivamente. La señal S4 muestra la señal de salida ($Tx_{out}$) del AEFT. En A se inyecta el primer error por lo que en B, el receptor Y detecta una violación de la regla del bit de *stuff* (esto es, 6 bits consecutivos del mismo valor). En el subsiguiente bit este mismo nodo procede a la transmisión de un *error flag*. Sin embargo, el error

inyectado en C evita que los otros dos nodos detecten ese *error flag*. El AEFT sí ha detectado los seis bits dominantes por lo que comenzará a enviar su AEF tras haber recibido un bit de valor recesivo. No obstante, ocurre algo inesperado: tras el primer *error flag* se transmite otro adicional que, como se puede apreciar en S4, no procede del AEFT. Se tuvo que analizar exhaustivamente la arquitectura del *hub* para identificar el orígen de este segundo señalizador de error.

El módulo Rx_CAN del *hub* ReCANcentrate (ver la Figura 3.17 de la página 51) monitoriza la señal acoplada resultante de las contribuciones de los nodos. Adicionalmente, incluye un simple mecanismo de detección y señalización de errores mediante el cual es capaz de transmitir un señalizador de error activo en caso de detectar un error. Es este mecanismo la fuente del segundo *error flag* observado. Detecta el primer *error flag* simultáneamente con el AEFT pero a diferencia de éste, no espera la recepción de un bit recesivo para iniciar la transmisión de su EF.

Volviendo a la Figura 5.35, tras el *error flag* enviado por el hub, en D, se inicia la transmisión del AEF propiamente dicho (compuesto por 5 EFs) por parte del AEFT. Como conclusión de esta prueba se puede decir que el escenario de inconsistencia se ha resulto eficientemente.

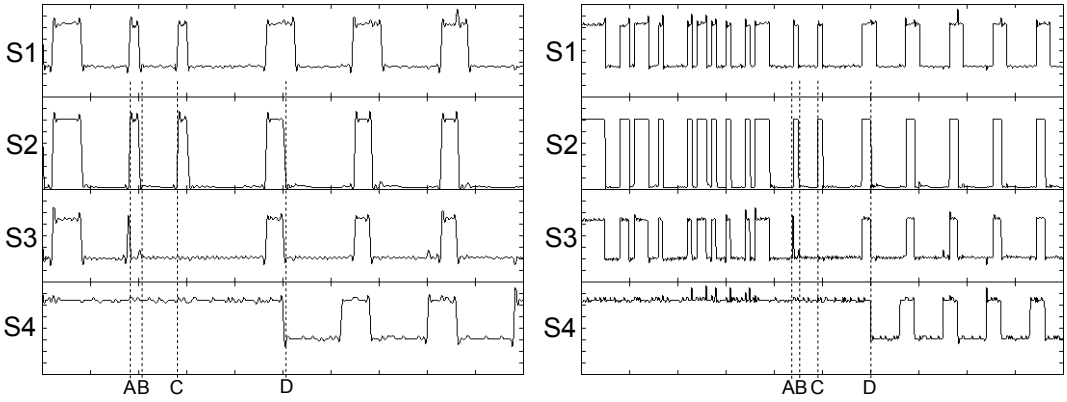



Figura 5.35: Escenario de inconsistencia resuelto mediante el AEFT integrado en el hub

Se realizaron varias pruebas adicionales con escenarios diferentes y en todos los casos los resultados (tanto del osciloscopio como de los *loggers*) fueron satisfactorios. Al ser todas estas pruebas muy similares a la anterior se ha optado por no incluirlas en esta memoria.

Una prueba que sí resulta importante se basa en el mismo escenario que el de la Figura 5.35. La diferencia es que el segundo error no se inyecta en los *downlinks* del transmisor y del receptor X sino que se inyecta en el *uplink* del receptor Y. Ese es el escenario descrito en el apartado 5.3.2, un escenario de inconsistencia que el AEFT, estando integrado en el *hub*, no es capaz de resolver. El resultado es el esperado y se muestra en la Figura 5.36. Al producirse el segundo error en el *uplink* del receptor Y, éste enmascara el *error flag* para todo el sistema, incluyendo el *hub*. Al estar el AEFT integrado en el *hub*, jamás se percatará de la transmisión del *error flag*, no enviará su AEF y no resolverá la inconsistencia.

La única solución al problema es cambiar a la segunda configuración del AEFT para topologías estrella, propuesta en el apartado 5.3.1. Esto es, retirar el AEFT del *hub* e incluír uno en cada nodo del sistema. Como ya mencionado durante el diseño, la segunda configuración es idéntica, a efectos prácticos, a un bus. Esto se debe a que el *hub* es completamente transparente desde el punto de vista de los nodos y de los AEFT (ya que los segundos están integrados en los primeros). Esta es la razón por la cual se ha optado por no realizar pruebas en base a dicha configuración.

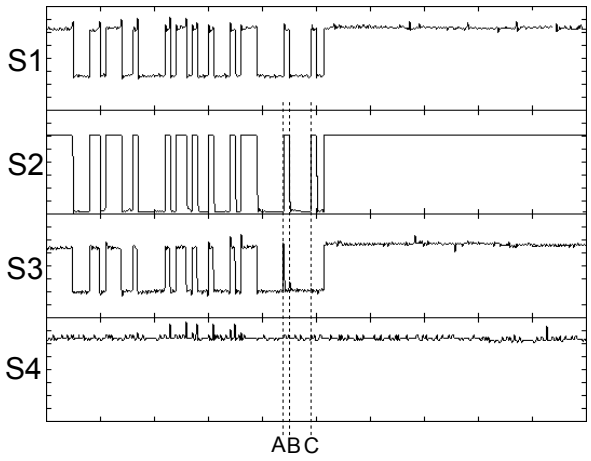



Figura 5.36: Escenario de inconsistencia no resuelto por el AEFT

**Observaciones**

Como bien se ha podido ver, ReCANcentrate incluye un mecanismo de control simple que inyecta un *error flag* si se detecta un error. Este mecanismo es capaz de resolver, por si solo, algunos de los escenarios de inconsistencia debidos a la señali-

zación inconsistente de errores. Sin embargo, este mecanismo es menos potente y menos flexible que el AEFT. Por tal motivo y al ser redundante tener dos transmisores de *error flags* en un mismo *hub*, se recomienda eliminar dicho mecanismo y reemplazarlo por el AEFT.

## 5.6. Conclusiones

El AEFT es un mecanismo propuesto para la implementación de la estrategia de resolución de inconsistencias basada en AEFs. Este mecanismo ha sido diseñado tanto para topologías bus, propias de CAN, como para topologías estrella (integrándose el diseño con el de ReCANcentrate).

El AEFT se compone por tres módulos: el AEFC, el EFD y el EFT. El AEFC es la unidad de control del dispositivo que genera las señales de control para los otros dos módulos. El EFD es un detector de *error flags* basado en un contador de bits dominantes y el EFT es un transmisor de *error flags* ligeramente modificado que, coordinado por el AEFC, transmite el AEF.

En un bus, el AEFT se localiza entre el controlador CAN y el *transceiver* de un nodo. Consecuentemente, una red con $n$ nodos deberá incluír $n$ AEFTs. Para el caso de una estrella, se han propuesto dos configuraciones diferentes: en la primera, la localización del AEFT es la misma que en un bus (en el nodo, entre el controlador CAN y el *transceiver*), necesitándose $n$ AEFTs para $n$ nodos. En la segunda configuración, el AEFT se localiza en el hub de la estrella por lo que se necesita un único AEFT independientemente del número de nodos.

Al realizarse en ReCANcentrate la conexión de los nodos con dos cables (un *uplink* y un *downlink*) en lugar de uno, fue necesario revisar los escenarios de inconsistencia resultantes de la señalización inconsistente de errores. De entre los nuevos escenarios identificados, hubo algunos que únicamente se resolvían mediante la configuración 1.

Para la implementación física del AEFT para topologías bus se ha migrado y modifico un controlador CAN en VHDL ya existente. Explicada la estructura interna de éste, se han detallado las modificaciones y la configuración realizadas para adaptarlo a las necesidades específicas del trabajo. En ese mismo contexto se han especificado

la plataforma de desarrollo y el entorno de programación utilizados. Seguidamente se ha expuesto el código VHDL escrito para implementar el AEFT. Finalmente, se han descrito el prototipo construido para realizar la verificación experimental, las pruebas realizadas y los resultados obtenidos a partir de éstas.

La implementación para topologías estrella ha tenido como primer paso la integración del AEFT en el hub de ReCANcentrate, seguida de la verificación experimental mediante el inyector de fallos sfiCAN y el osciloscopio digital. Los resultados obtenidos mediante los (*loggers*) de sfiCAN fueron, en todo momento, idénticos a los obtenidos mediante el osciloscopio. Como ya se ha identificado durante la fase de diseño, únicamente la configuración 1 resuelve todas las inconsistencias. Sin embargo, es una opción más costosa ya que se necesita un mayor número de AEFTs. Al final la elección de una configuración u otra dependerá de la aplicación y del presupuesto.

A modo de conclusión, decir que el mecanismo propuesto es simple pero efectivo. Tanto para topologías bus como estrella, los resultados de la verificación experimental han sido satisfactorios. El AEFT resuelve cualquier escenario de inconsistencia cuya fuente sea la señalización inconsistente de errores. De esta manera, una de las tres causas de inconsistencias en CAN queda eliminada. Otra de las causas, la regla del último bit del EOF, es tratada por CANsistant, un mecanismo que es descrito de forma detallada en el siguiente capítulo.

# Capítulo 6

# CANsistant: diseño e implementación

## 6.1. Introducción

Recordando lo que ya se mencionó en el capítulo 3.4.3, para resolver el problema del último bit del EOF e impedir que se produzcan los escenarios de inconsistencia identificados por Rufino *et. al* [RUFI98], este mismo autor propuso un conjunto de protocolos de alto nivel, ejecutados en la capa de aplicación. Estos protocolos requieren la transmisión de una trama de control por cada trama de datos y, por lo tanto, introducen un elevado *system overhead*.

Más tarde, Livani [LIV99] propuso SHARE (SHAdow REtransmitter): un módulo, conectado a la red como si fuera un nodo genérico, que al detectar seis bits dominantes consecutivos a partir del último bit del EOF, señaliza una posible inconsistencia y retransmite la trama potencialmente corrompida. Para ello se guardan todas las tramas transmitidas por el bus y, en caso de detectar la secuencia de bits mencionada, se retransmite la última trama almacenada. En cambio, si la secuencia no es detectada, la trama correspondiente es descartada. Este patrón de bits es característico de los escenarios de inconsistencia identificados en [RUFI98].

Sin embargo, SHARE no es capaz de resolver los escenarios de múltiples errores descritos en el capítulo 3.3.1 [PROE00]. Consecuentemente, dentro del proyecto

CANbids, se planteó un mecanismo que sí sea capaz de detectar inconsistencias incluso en presencia de múltiples errores de canal: CANsistant (*CAN Assistant for Consistency*) [PROE09] (ver el capítulo 3.4.3). Dicha solución, hasta el momento puramente teórica, es ahora tomada como punto de partida para el diseño y la implementación física de un dispositivo tanto para topologías bus como estrella.

La organización del capítulo es la siguiente: primeramente se exponen las opciones de diseño, esto es, las diferentes versiones que se han diseñado y seguidamente, se explican detalladamente los pasos seguidos para el diseño y la implementación de cada una de las versiones.

## 6.2. Opciones de diseño

Aunque la propuesta inicial de CANsistant fue para topologías bus [PROE09], en este trabajo se planteó también la integración de CANsistant con ReCANcentrate, esto es, su uso en una topología estrella. En una topología bus, CANsistant es acoplado a la red como si de un nodo genérico se tratara. En una topología estrella, en cambio, CANsistant estará integrado en el concentrador (*hub*).

En esta memoria se proponen tres versiones diferentes de CANsistant:

- Versión 1: la versión original para topologías bus, propuesta en el apartado 3.4.3.
- Versión 2: una segunda versión para topologías bus, basada en la primera pero rediseñada para reducir el número de falsas alarmas.
- Versión 3: una versión para topologías estrella basada en la integración de la segunda versión de CANsistant con ReCANcentrate.

## 6.3. Diseño e implementación: versión 1

Tomando como punto de partida la propuesta realizada en [PROE09] y expuesta en el capítulo 3.4.3, se ha diseñado un módulo que implemente las funcionalidades de éste. Dicho dispositivo se basa en la topología bus, propia del protocolo CAN.

CANsistant está diseñado para identificar escenarios de inconsistencia en presencia de múltiples errores de canal [PROE09]. A modo de recordatorio, el número de errores tolerables por el mecanismo es igual a $m = 4$. Los parámetros que definen el comportamiento de CANsistant son tres:

1. La ventana FDDW (*First Dominant Detection Window*), que se corresponde con la secuencia de bits en la que se debe encontrar el primer bit dominante.
2. El factor ND (*Number of Dominant bits*), es decir el número de bits dominates que CANsistant debe encontrar después de detectar el primer bit dominante en el FDDW.
3. La ventana ADDW (*Additional Dominant Detection Window*), que se corresponde con el grupo de bits en los que CANsistant debe encontrar ND bits dominantes adicionales.

En un principio, CANsistant fue planteado para retransmitir las tramas corrompidas tras la ocurrencia de inconsistencias. No obstante, finalmente se decidió no incluir dicha funcionalidad debido al creciente interés en usar el modo de transmisión única en los sistemas distribuidos (deshabilitando la retransmisión automática de mensajes). Esto aumenta de forma importante la probabilidad de las inconsistencias [RODR03] y pone aún más en evidencia la necesidad de CANsistant.

### 6.3.1. Arquitectura interna

CANsistant, en su versión para topologías bus, se debe comportar como un nodo adicional de la red CAN y, por lo tanto, debe incluir funcionalidades básicas de un controlador CAN genérico. Entre dichas funcionalidades se encuentran la sincronización a nivel de bit, la identificación de bits de stuff, la detección de errores y la señalización activa de errores. Para implementar estos mecanismos en CANsistant, se optó por utilizar el módulo CAN diseñado en su momento para el hub de CANcentrate/ReCANcentrate y adaptarlo debidamente. En la figura 6.1 se muestra el diagrama de bloques del nodo CANsistant, pudiéndose identificar dos bloques bien diferenciados: el módulo CANsistant propiamente dicho y, el módulo CAN que implementa la capa física del protocolo CAN así como diferentes mecanismos de la capa

de enlace. Concretamente, el módulo CAN proporciona a CANsistant las señales de reloj de recepción (Rx) y transmisión (Tx), así como una señal de activación basada en la detección del primer bit del EOF de la trama recibida actualmente.

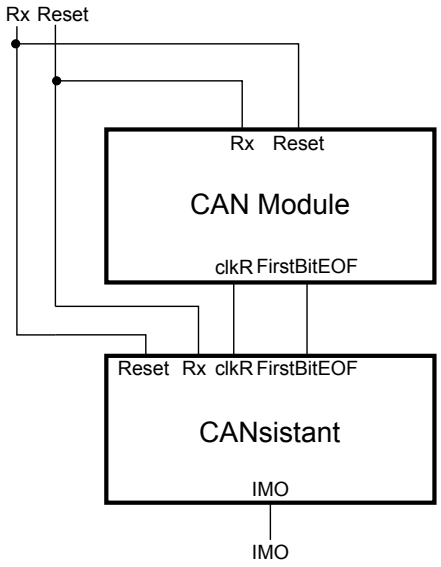



Figura 6.1: Diagrama de bloques del nodo CANsistant

Las funcionalidades de CANsistant se modelan mediante la máquina de estados finitos (*finite state machine*) de la Figura 6.2. Dicho modelo, compuesto por seis estados, define el comportamiento del sistema así como sus entradas y salidas. El estado inicial se denomina *Idle* y en él se inicializan todas las variables. Para pasar de este estado al siguiente y poner en marcha el módulo, se debe activar la señal $FirstBitEOF$. Dicha señal proviene del módulo CAN y se activa cuando el siguiente bit de la trama es el primer bit del EOF. Una vez en el estado *Waiting*, se inicializa un contador denominado *countbit*. Mediante este contador se cuentan el número de bits recibidos y se controla el comportamiento de CANsistant. Cuando $countbit = 5$, la máquina de estados pasa del estado *Waiting* al estado *FDDW*. Como ya descrito en el apartado anterior, en esta ventana se debe encontrar el primer bit dominante ($Rx =' 0'$). Si esto ocurre, la máquina de estados pasa al estado ADDW. Si dentro de la ventana FDDW no se encuentra ningún dominante ($countbit = 9$), se pasa al estado Waiting2. En ADDW se deben encontrar 3 bits dominantes adicionales ($ND = 3$) para activar la señal $IMOdetected$ y pasar al estado *Waiting2*. En caso de no encontrar 3 bits dominantes, la máquina de estados pasará al mismo estado cuando $countbit = 12$ pero sin que se active $IMOdetected$. El estado *Waiting2* simplemente

es un estado transitorio del que se pasa al estado *stateIMO* si la señal $IMOdetected$ está activa y al estado *Idle* si no está activa y $countbit = 14$. En *stateIMO* se activa la señal $IMO$ para señalizar una posible inconsistencia y, seguidamente, se vuelve al estado inicial *Idle*.

El estado *Waiting2* sirve para temporizar la activación de la señal $IMO$. Esta señal es activada siempre en el quinto bit tras el último bit del IFS. Como ya mencionado anteriormente, el diseño actual de CANsistant únicamente detecta y señaliza posibles escenarios de inconsistencia sin retransmitir la trama afectada. En el caso de que se desee habilitar la retransmisión, se deberá incluir en el diseño no solo un módulo de recepción CAN sino también un módulo de transmisión.

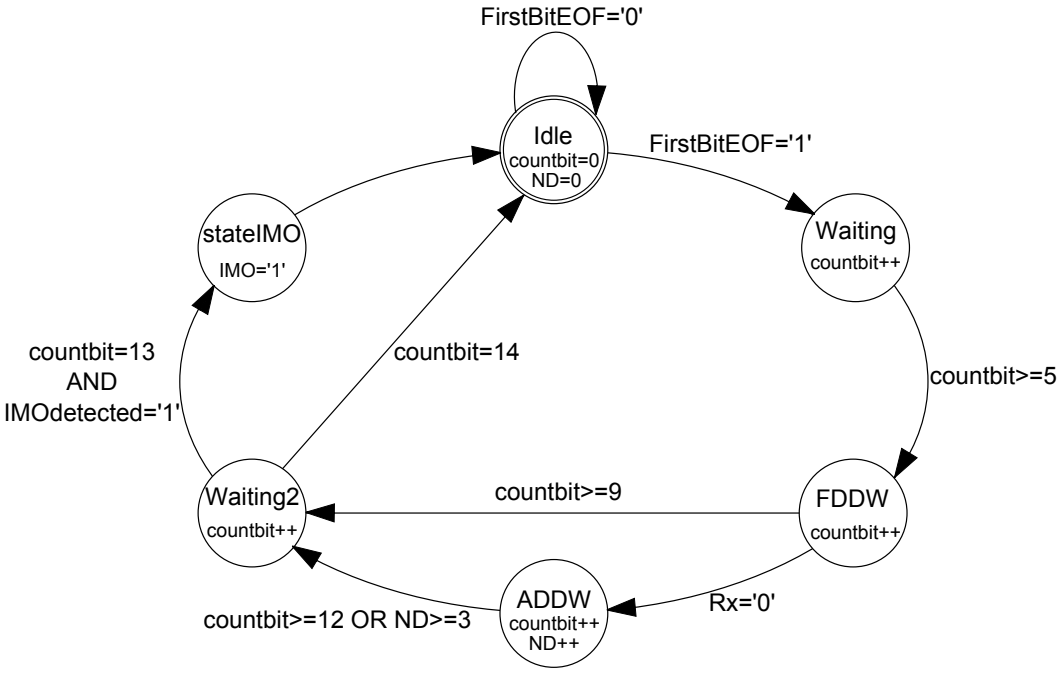



Figura 6.2: Máquina de estados de CANsistant

La arquitectura interna del módulo CAN se ilustra en la Figura 6.3. Dicho módulo está compuesto por tres bloques: la unidad denominada *Physical Layer Unit*, la unidad *Rx CAN Module* y la unidad *Error Frame Generator*. A continuación se describen sus funcionalidades:

- *Physical Layer Unit*: este módulo implementa la capa física del protocolo CAN. Está formado por el *Baud-Rate Prescaler* y el *Sincronizer*. Genera las señales

de reloj de recepción ($clk_R$) y transmisión ($clk_T$) a partir de una señal de reloj procedente de un oscilador ($clk_osc$). CANsistant únicamente necesita la señal $clk_R$ para su correcto funcionamiento. La velocidad esta configurada en 1Mb/s.

- *Rx CAN Module*: está formado por la *Stuff Unit*, encargada de identificar y extraer los bits de *stuff* de la trama en recepción; la *Frame Monitor Unit*, que indica en que posición y campo de la trama se encuentra el bit transmitido actualmente y si se ha producido o no un error y; el *CRC Calculator*, encargado de calcular el CRC de la trama recibida. Este módulo genera la señal $FirstBitEOF$ que indica si el siguiente bit de la trama es el primer bit del EOF y controla la puesta en marcha de CANsistant.

- *Error Frame Generator*: CANsistant debe ser capaz de señalizar errores de forma activa y, por lo tanto, debe incluir un módulo que realice dicha tarea. En caso de error, el *Error Frame Generator* recibe la señal *Error* del *Rx CAN Module* y transmite un *active error flag*, es decir, seis bits dominantes consecutivos.

### 6.3.2. Implementación en VHDL

Para una posterior implementación física en una FPGA, el diseño propuesto en el apartado anterior es ahora definido en código VHDL. Para ello se ha hecho uso del entorno de desarrollo *ISE Design Suite 13* de Xilinx. Este entorno permite analizar, sintetizar, compilar y simular los diseños realizados. La edición utilizada es la *ISE Web Edition*, una versión que soporta la familia completa de FPGAs *Spartan*. Los códigos correspondientes, debidamente comentados, están incluidos en el *Apéndice C*.

El código VHDL de CANsistant (ver el *Apéndice C*, página 173) presenta 4 entradas y una salida. *Reset* es una señal de reinicialización asíncrona del sistema. La señal *clkR*, proveniente de la *Physical Layer Unit*, es el reloj de recepción mediante el cual se muestrea la entrada *Rx*, es decir, el canal. La entrada $FirstBitEOF$ proviene de la *Frame Monitor Unit* e indica que el próximo bit recibido por *Rx* es el primer bit del EOF. Finalmente, la salida $IMO$ señaliza una posible inconsistencia y se activa cuando se ha detectado el patrón de bits correspondiente.

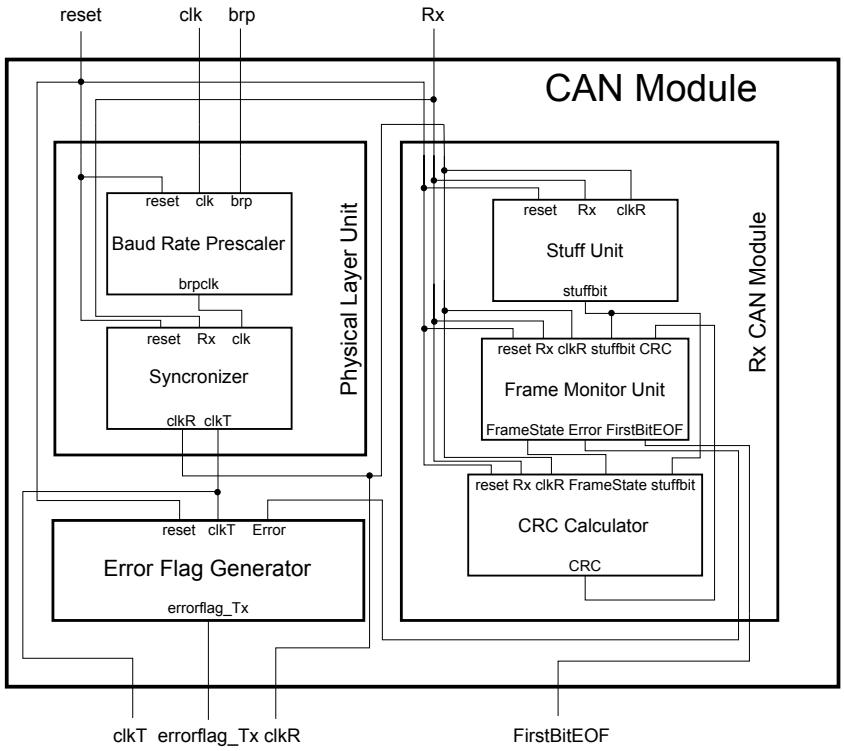



Figura 6.3: Diagrama de bloques del módulo CAN

La máquina de estados que modela el comportamiento de CANsistant está definida como un *process*. La señal de salida $IMO$ es controlada en el estado *stateIMO*, tomando valor '1' en caso de haberse detectado el patrón de bits que indica un posible escenario de inconsistencia y, '0' en el caso contrario. Así como se ha definido la activación de la señal $IMO$, ésta se resetea cuando la máquina de estados vuelve al estado *Idle*. Si se deseara mantener la señal activada una vez se haya detectado la primera inconsistencia, esto implicaría realizar cambios menores en el código actual. Nótese además, que aunque la señal de salida se denomine $IMO$, el diseño actual de CANsistant detecta tanto IMOs como IMDs.

La monitorización de las tramas recibidas es una acción indispensable para garantizar el correcto funcionamiento de CANsistant. El módulo denominado *Frame Monitor Unit* (ver el *Apéndice C*, página 175) implementa dicha funcionalidad y presenta 6 entradas y 6 salidas.

Entradas:
- *Reset*: señal de reseteo asíncrono del sistema.

- *stuffbit*: proviene de la *Stuff Unit* e indica si el siguiente bit es un bit de *stuff*.
- *stuffvalue*: al igual que la entrada anterior, proviene de la *Stuff Unit*. Indica el valor del bit de *stuff*.
- *clkR*: reloj de recepción que controla la máquina de estados (por flanco de subida).
- *Rx*: señal de recepción proveniente del bus.
- *CRCvalue*: valor del CRC calculado por el *CRC Calculator*.

Salidas:
- *StuffEnable*: señal de salida que controla la habilitación de la *Stuff Unit*.
- *error*: indica que se ha detectado un error y pone en marcha el *EF Generator*.
- *FirstBitEOF*: indica que el próximo bit es el primer bit del EOF. Es la señal de activación de CANsistant.
- *FrameState*: Indica el campo en el que se encuentra el bit recibido.
- *BitNumber*: Indica la posición en la trama del bit recibido.
- *CRCerror*: señal que se activa en caso de detectarse un error de CRC.

La monitorización es realizada mediante una máquina de estados. Cada estado se corresponde con un campo de la trama CAN (*idle, idfield, rtrfield, resfield, dlcfield, datafield, crcfield, crcDelimField, ackSlotField, ackDelimField, eofField, interfield* y *errorflag*). La transición entre estados se rige por las reglas del protocolo CAN [ISO93].

La *Stuff Unit* (ver el *Apéndice C*, página 178) se encarga de identificar los bits de *stuff* de la trama en recepción. El código VHDL correspondiente presenta 4 entradas y 2 salidas: la señal *Reset* reinicializa el sistema, *enable* es controlada por la *Frame Monitor Unit* y pone en marcha el módulo, $clkR$ es el reloj de recepción, $Rx$ es la señal de recepción, $stuffbit$ indica si el próximo bit es o no un bit de *stuff* y $stuffvalue$ indica el valor esperado del bit de *stuff*. Para identificar los *stuff bits* se hace uso de una máquina de estados que implementa un simple contador de bits recesivos/dominantes. Contados 5 bits consecutivos de un mismo valor se activa la señal de salida $stuffbit$.

El *CRC Calculator* (ver el *Apéndice C*, página 179) se encarga de calcular el CRC de la trama en recepción. Presenta 5 entradas y 1 única salida. Al igual que los módulos anteriores, presenta las entradas $reset$, $clkR$ y $Rx$. Incluye, además, la entrada $FrameState$, proveniente de la *Frame Monitor Unit* y que indica el campo

del bit recibido y, la entrada $stuffbit$ recibida desde la *Stuff Unit* y que informa sobre si el bit recibido es o no un bit de stuff. La única salida es $CRCvalue$, la secuencia de CRC calculada por el módulo.

Para el cálculo del CRC se utiliza el algoritmo de generación propio de CAN [ISO93]. Éste es una derivación del algoritmo CRC-15. Los coeficientes del polinomio a dividir vienen dados por la trama CAN (incluyendo el SOF, el campo de arbitraje, el campo de control y el campo de datos) y, para los 15 coeficientes inferiores, por 0. Este polinomio es dividido por el polinomio generador mostrado en la expresión 6.1. El resto de la división es la secuencia CRC transmitida por el bus. Para su implementación se utiliza un registro de desplazamiento de 15 bits.

$$x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1 \tag{6.1}$$

El *Error Flag Generator* (ver el *Apéndice C*, página 180) genera señalizadores de error activos. Presenta tres entradas (la señal de reinicialización $reset$, la señal de reloj de transmisión $clkT$ y la señal de habilitación $error$, activada por la *Frame Monitor Unit*) y una salida ($errorflag$). La transmisión de EFs es controlada por una máquina de estados con dos estados (*idle* y *errorFlag*). En el estado *idle*, se transmiten únicamente bits recesivos. En el estado *errorFlag* ,en cambio, se transmiten 6 bits dominantes. La transición del primer estado al segundo viene ligada a la activación de la señal de entrada $error$. Aparte de la transmisión de EF activos, CANsistant se comporta como un nodo pasivo de la red. Así pues, la señal $errorflag$ puede ser conectada directamente a la señal de transmisión ($Tx$) del nodo.

El código VHDL correspondiente a la *Physical Layer Unit* no se ha incluido en esta memoria ya que es extenso e idéntico al utilizado para el hub CANcentrate/ReCANcentrate. Simplemente decir que este bloque es el encargado de generar las señales de reloj de transmisión y recepción utilizadas por varios de los módulos expuestos anteriormente. A su vez, el código utilizado para la inteconexión de los diferentes bloques se ha omitido por su simplicidad (los módulos son definidos como componentes e interconectados mediante *port mapping*).

### 6.3.3. Simulación

Una vez implementado el diseño en VHDL, se pasó a la simulación de éste para comprobar su correcto funcionamiento. Para ello se hizo uso del *ISE Simulator* (*ISim*), simulador propio del entorno *Xilinx ISE Design Suite*. Las simulaciones se realizaron en base a diferentes *Test Benchs*, ficheros de patrones en código VHDL, definidos para generar diferentes estímulos en instantes de tiempo concretos.

Al ser el módulo CAN idéntico, excepto modificaciones menores, al módulo utilizado en el hub de CANcentrate/ReCANcentrate y el correcto funcionamiento de éste ya ha sido comprobado en varias ocasiones, en este apartado únicamente se simula y verifica el diseño de CANsistant.

En la Figura 6.4 se ilustra la primera simulación realizada. La señal de entrada $clkR$ es una señal de reloj de frecuencia 100Mhz. Dicha señal controla las transiciones de la máquina de estados y su frecuencia se eligió por defecto (la frecuencia en un sistema real, obviamente, sería mucho menor). Al inicio de la simulación, la señal *Reset* es activada durante un ciclo de reloj para garantizar que la máquina de estados regrese al estado inicial $Idle$. A los 20ns se activa la señal $FirstBitEOF$ y CANsistant salta al estado $Waiting$. 6 flancos de reloj más tarde ($countbit = 5$), la máquina pasa al estado $FDDW$ y espera la recepción del primer error, es decir, bit dominante. Sin embargo, la señal $Rx$ se mantiene a nivel '1' y, por lo tanto, cuando $countbit = 9$, la máquina de estados pasa al estado $Waiting2$. Si se ha detectado una posible inconsistencia, a los 160ns después de iniciar la simulación, CANsistant activa la señal $IMO$. En este caso, sin embargo, no se ha detectado el patrón correspondiente y, por lo tanto, la señal $IMO$ se mantiene desactivada. Finalmente, se reinicializa el sistema y se regresa al estado $Idle$.

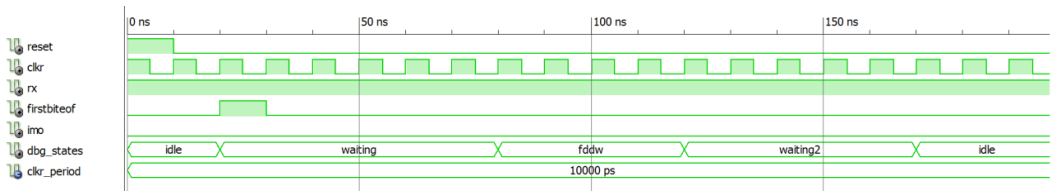



Figura 6.4: 1ª Simulación: no se detectan errores

En la simulación de la Figura 6.5, la señal $Rx$ toma valor dominante en el último bit de la ventana FDDW y, por lo tanto, la máquina de estados pasa al estado

ADDW. Sin embargo, al no detectarse bits dominantes adicionales, la máquina de estados pasa al estado $Waiting2$ y, seguidamente, al estado $Idle$, sin activar la señal $IMO$. En las Figuras 6.6 y 6.7, se detectan, respectivamente, uno y dos bits adicionales en la ventana ADDW. No obstante, al no detectarse un total de tres bits dominantes adicionales, CANsistant no identifica el escenario como potencialmente inconsistente y, por lo tanto, no activa la señal $IMO$.

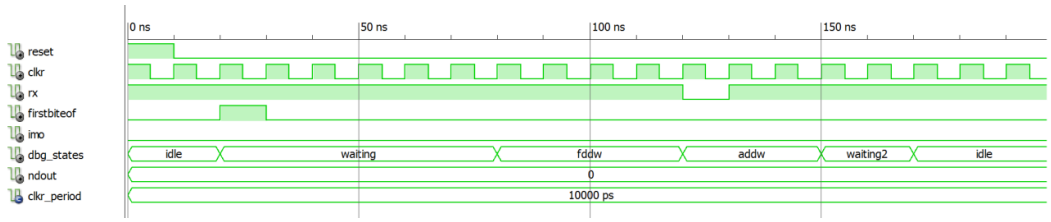



Figura 6.5: 2ª Simulación: se detecta 1 error

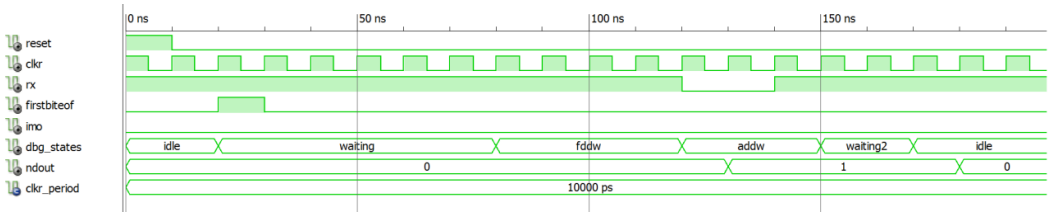



Figura 6.6: 3ª Simulación: se detectan 2 errores

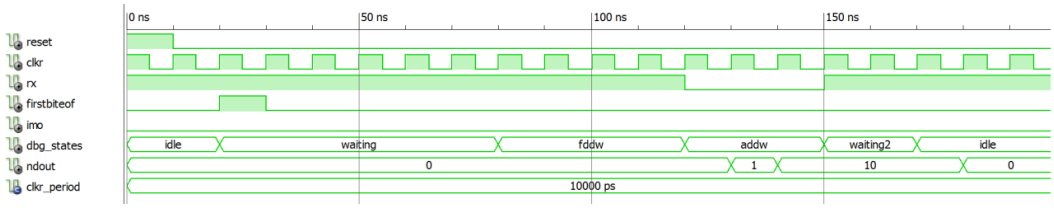



Figura 6.7: 4ª Simulación: se detectan 3 errores

El caso ilustrado en la Figura 6.8 es diferente. Se detecta un primer error en el estado FDDW y tres errores adicionales en el estado ADDW. Consiguientemente, se pasa al estado $Waiting2$ y, directamente después, al estado $stateIMO$, activándose la señal $IMO$. Recuérdese que el estado $Waiting2$, en este caso, sirve para temporizar la habilitación de la señal de salida (ésta se activará siempre a los 160ns). En el escenario de la Figura 6.9, los bits dominantes de $Rx$ no son consecutivos como en el caso anterior pero CANsistant sigue detectando el escenario como potencialmente inconsistente tal y como debería.

En base a las simulaciones anteriores queda verificado el diseño de la primera versión de CANsistant. La máquina de estados implementa de manera satisfactoria todas las funcionalidades de la propuesta original de CANsistant. Cuando no se detecta el patrón de bits correspondiente a una posible inconsistencia, el sistema permanece pasivo. Únicamente en el caso de detectar 4 bits dominantes en las ventanas FDDW y ADDW, CANsistant activará su salida para señalizar el escenario potencialmente inconsistente.

Como bien expresado anteriormente, CANsistant únicamente es capaz de detectar inconsistencias "potenciales". Así pues, una parte de los escenarios inconsistentes detectados pueden no serlo, produciéndose las denominadas falsas alarmas. En el siguiente apartado se expondrán los pasos seguidos para el diseño y la implementación en código VHDL de la segunda versión de CANsistant, mejorada para reducir el número de dichas falsas alarmas.

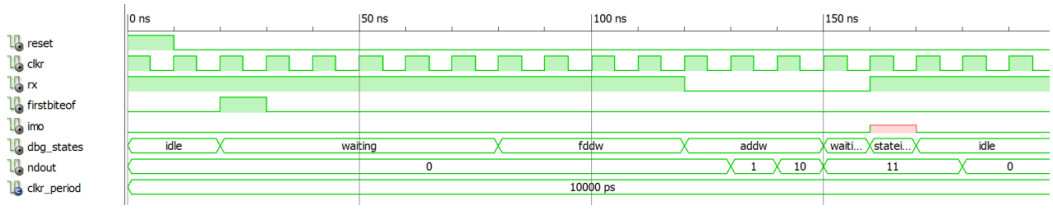



Figura 6.8: 5ª Simulación: se detectan 4 errores y se señaliza una inconsistencia

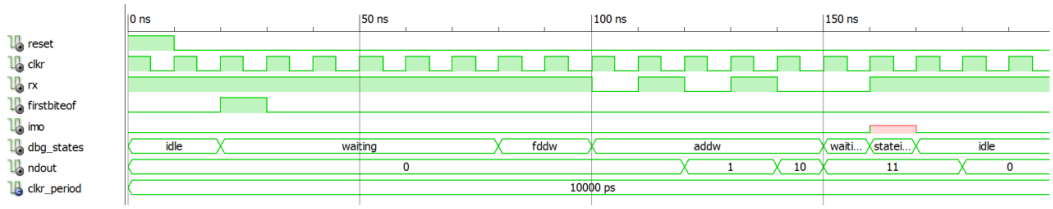



Figura 6.9: 6ª Simulación: se detectan 4 errores y se señaliza una inconsistencia

## 6.4. Diseño e implementación: versión 2

En este apartado se analizarán detenidamente las falsas alarmas y su origen para, a continuación, proponer un diseño mejorado de CANsistant, capaz de identificar con mayor precisión las inconsistencias reales y reduciendo al máximo la notificación de falsas alarmas.

### 6.4.1. Falsas alarmas

Como ya se ha mencionado en el capítulo 3.4.3, uno de los inconvenientes de CANsistant son las falsas alarmas, es decir, la detección de IMOs sin que éstos hayan ocurrido realmente. Así pues, uno de los objetivos primordiales es modificar el diseño inicial de CANsistant para reducir el número de falsas alarmas.

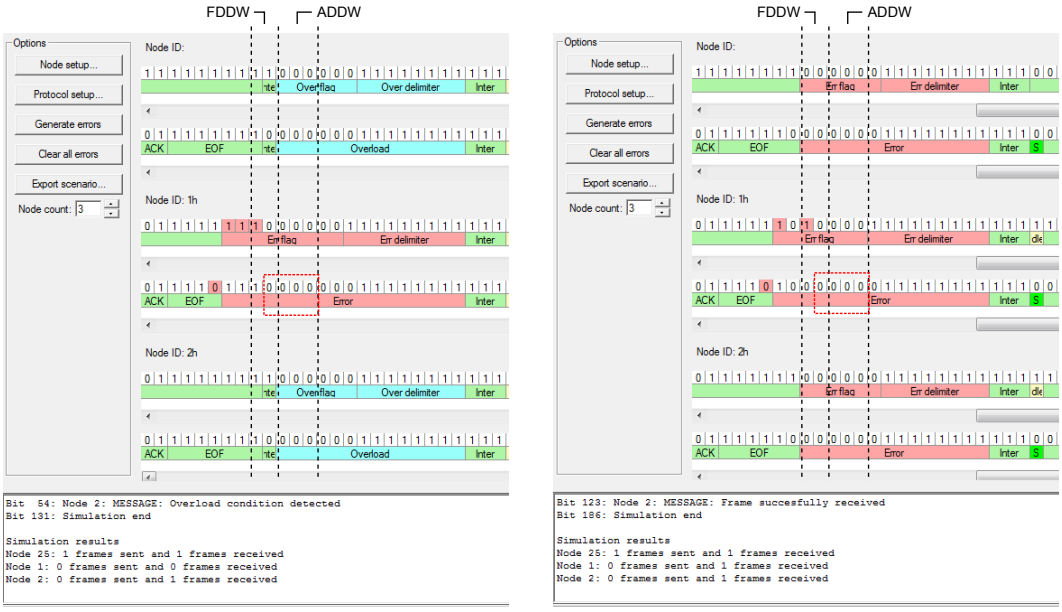
Las decisiones tomadas en este apartado se basan en un *technical report*[1] interno, no publicado que trata la ocurrencia de falsas alarmas y su procedencia. Los escenarios de inconsistencia tratados incluyen tanto IMOs como IMDs y el análisis de las falsas alarmas considera diferentes casos dependiendo de en qué bit del EOF/IFS se detecta el primer error.

En general, una falsa alarma puede tener dos causas:

- El *error flag* transmitido por uno de los nodos coincide con el EOF de la trama. El *error flag* es recibido por los nodos y la trama es rechazada de forma consistente. Sin embargo, CANsistant detecta varios bits dominantes en las ventanas FDDW y ADDW y, señaliza un IMO. En la Figura 6.10 se ejemplifica la situación anterior: en (a), el nodo 2 detecta un error en el cuarto bit del EOF. Su *error flag* se ve afectado por tres errores adicionales por lo que los otros dos nodos no detectan el error hasta el primer bit del IFS, considerándolo parte de una trama de sobrecarga. Se produce un IMO detectado por CANsistant. En (b), el escenario es idéntico al anterior salvo que no se producen tres errores adicionales sino dos. En este caso los nodos rechazan consistentemente la trama pero CANsistant seguirá detectando un IMO, en este caso, no existente.

- Una trama de sobrecarga puede ser malinterpretada por CANsistant ya que se detectan varios bits dominantes en las ventanas FDDW y ADDW. En la Figura 6.11 se ilustra esta situación: se produce un error en el segundo bit del IFS que provocará la transmisión de tramas de sobrecarga por parte de los nodos. Aunque todos los nodos hayan aceptado la trama, CANsistant detectará una secuencia de bits dominantes que le harán señalizar equivocadamente un IMO.

[1] *New inconsistency scenarios in CAN and how to treat them in networks that already use CANsistant for inconsistency detection.*




(a) (b)

Figura 6.10: IMO detectado por CANsistant (a) y falsa alarma (b)

Realícese un análisis más profundo del origen de las falsas alarmas teniendo en cuenta en qué bit del EOF CANsistant detecta el primer error y bajo la suposición de que CANsistant señaliza de forma activa la detección de errores:

1. **Primer error detectado en el primer o segundo bit del EOF:** En este caso resulta imposible que se produzca una inconsistencia ya que todos los nodos detectarán errores antes del último bit del EOF. Sin embargo, bajo la presencia de errores adicionales, CANsistant podría llegar a señalizar una inconsistencia. Para evitar dicho comportamiento, CANsistant debe ser rediseñado para no analizar la trama en caso de detectar el primer error en el primer o segundo bit del EOF.

2. **Primer error detectado en el tercer bit del EOF:** En este caso, la ocurrencia de inconsistencias es posible ya que la detección del primer bit dominante por parte de algunos nodos puede ser retrasada hasta el último bit del EOF (suponiendo que tras el primer error se producen tres fallos adicionales). CANsistant es capaz de detectar dichas inconsistencia pero también es muy propenso a notificar inconsistencias no existentes. Por ejemplo, en el caso de

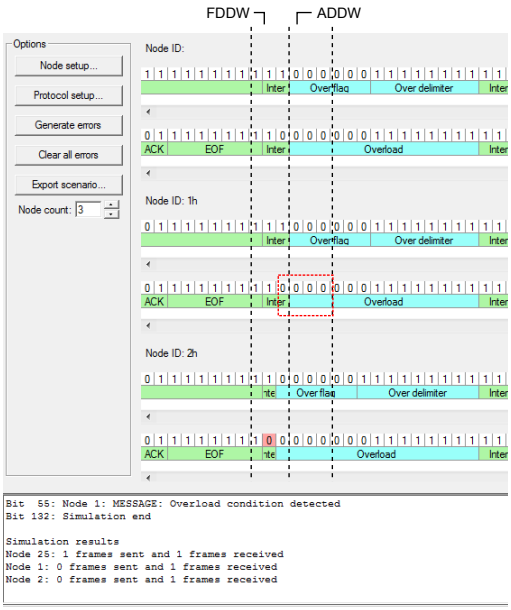



Figura 6.11: Falsa alarma de CANsistant

que CANsistant sea el primero y único en detectar un error (que es una situación relativamente probable), éste siempre generará una falsa alarma. Esto es debido a que detectará 4 bits dominantes dentro de las ventanas FDDW y ADDW. Si CANsistant no ha sido el primero y único en detectar el error, éste únicamente encontrará 3 bits dentro de los bits bajo observación y no notificará una falsa alarma.

Adicionalmente, CANsistant puede generar falsas alarmas incluso en el caso de producirse errores adicionales en los bits bajo observación. Por un lado, si uno de estos errores fuerza un valor dominante, ésto contribuirá a la notificación de una inconsistencia. Por el otro lado, si el error fuerza un valor recesivo, ésto alargará el *error flag* y también favorecerá la generación de una falsa alarma. Una excepción representa el caso en el que CANsistant ha finalizado la transmisión de su *error flag* y el bit dominante afectado pertenece a un *error flag* transmitido por algún otro nodo del sistema. En este caso concreto, se reduce el número de bits dominantes dentro de las ventanas FDDW y ADDW.

Al saber que tras la detección de un error se transmitirá un *error flag*, CANsistant puede ser rediseñado para reducir el número de falsas alarmas. Concretamente, para que el *error flag* sea enmascarado y se produzca una inconsistencia, un nodo se debe ver afectado por 3 errores adicionales y, al haberse consumido

todos los posibles errores, CANsistant deberá encontrar 7 valores dominantes en los 7 bits bajo evaluación para notificar una inconsistencia.

Esta modificación en el diseño de CANsistant no elimina por completo las falsas alarmas pero sí elimina las más probables: aquellas que se producen cuando el error es detectado por CANsistant en el tercer bit del EOF o por otros nodos en el segundo bit del EOF (esto es, forzando CANsistant a ver un error en el tercer bit del EOF).

3. **Primer error detectado en el cuarto bit del EOF:** Al igual que en el caso anterior, es posible que se produzcan inconsistencias en presencia de errores adicionales. De nuevo, CANsistant es capaz de detectar todas estas inconsistencias, con la peculiaridad de que siempre notificará una inconsistencia en caso de detectar un error en el cuarto bit del EOF. Si CANsistant es el primero y único en detectar el error, éste encontrará 5 bits dominantes en las ventanas FDDW y ADDW y, consecuentemente, notificará una inconsistencia. Si CANsistant no es el primero y único en detectar el error o el error ha sido detectado por otro nodo en el bit anterior (el tercer bit del EOF), CANsistant encontrará 4 bits dominantes en los 7 bits bajo observación y también notificará una inconsistencia. Si se producen errores en los bits bajo observación, ésto tendrá las mismas consecuencias que en el caso anterior pero jamás impedirá que CANsistant notifique una inconsistencia.

   Para eliminar las falsas alarmas más probables, de nuevo, CANsistant debe utilizar la información referente a la transmisión de *error flags*. En este caso, el *error flag* puede ser enmascarado por 2 errores adicionales y, por lo tanto, quedará un único error por consumir. Consecuentemente, para notificar una inconsistencia, CANsistant debe detectar un primer bit dominante en el último bit del EOF o en el primer bit del IFS y, seguidamente, un mínimo de 5 bits dominantes adicionales hasta el tercer bit después del IFS.

4. **Primer error detectado en el quinto bit del EOF:** Este caso es idéntico al anterior con la única diferencia de que CANsistant detecta 6 bits dominantes en el caso de ser el primero y único en detectar el error o, 5 bits dominantes en caso de no ser el primero y único en detectar el error o si el error ha sido detectado por otro nodo en el bit anterior. Incluso produciéndose errores adicionales en los bits bajo observación, CANsistant siempre detectará, como mínimo, 4 bits dominantes y notificará una inconsistencia.

La reducción de las falsas alarmas es imposible en este caso ya que CANsistant no es capaz de diferenciar entre una inconsistencia real y una falsa alarma. Así pues, si se produce una inconsistencia real, CANsistant detectará 5 bits de valor dominante en los bits bajo observación (Figura 6.12) pero, si no se produce ninguna inconsistencia real, CANsistant también detectará esos 5 bits dominantes (o incluso 6). Únicamente las falsas alarmas en las que se detectan 4 bits dominantes pueden ser eliminadas. Para ello CANsistant debe ser rediseñado para únicamente notificar inconsistencias en caso de detectar un primer bit dominante en el último bit del EOF, en el primer bit del IFS o en el segundo bit del IFS y, un mínimo de 4 dominantes adicionales hasta el tercer bit después del IFS.

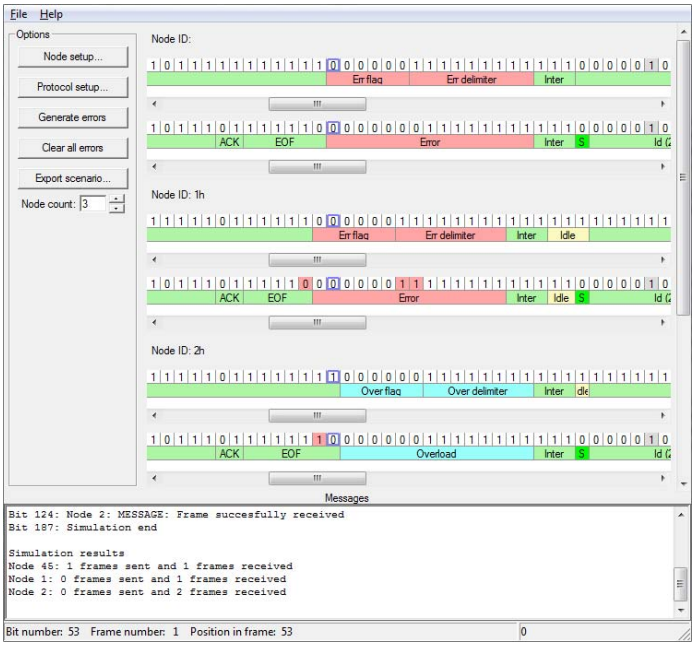



Figura 6.12: Inconsistencia con el primer error detectado en el quinto bit del EOF

5. **Primer error detectado a partir del sexto bit del EOF:** Si CANsistant detecta un error en el sexto o séptimo bit del EOF o, en uno de los 3 bits del IFS, éste siempre notificará una inconsistencia. El diseño no puede ser modificado para reducir las falsas alarmas ya que CANsistant es incapaz de discriminar entre falsas alarmas e inconsistencias reales. Por consiguiente, para estos casos específicos se mantendrá el diseño inicial de CANsistant: se debe detectar un primer dominante en el último bit del EOF, en el primer bit del

IFS, en el segundo bit del IFS o en el tercer bit del IFS y, seguidamente, 3 bits dominantes adicionales hasta el tercer bit después del IFS.

Si el primer error es detectado en el primer bit después del IFS, CANsistant no tomará medidas tal y como fue definido en su diseño inicial. Esto se debe a que resulta más probable que el bit dominante detectado sea el SOF de una nueva trama en lugar de un error.

Como bien se ha podido ver en los casos anteriores, el nuevo diseño de CANsistant deberá tomar unas medidas u otras dependiendo de en que bit del EOF se ha detectado el primer error. De esta forma se eliminan gran parte de las falsas alarmas. Sin embargo, no resulta posible eliminar ciertas falsas alamas ya que la información con la que trata CANsistant no es suficiente para discriminar las inconsistencias falsas de las inconsistencias reales. Esta limitación es inherente a la topología bus.

### 6.4.2. Rediseño de CANsistant para reducir el número de falsas alarmas

En el apartado anterior se han propuesta varias modificaciones al diseño original de CANsistant para eliminar parte de las falsas alarmas. A modo de resumen, a continuación se listan los cambios de mayor importancia:

1. CANsistant no debe señalizar inconsistencias en caso de detectar un error en el primer o segundo bit del EOF ya que, en este caso, es imposible que se produzcan inconsistencias debido a la regla del último bit del EOF.

2. Para reducir una parte importante de las falsas alarmas, los patrones de bits a detectar por CANsistant deben ser diferentes dependiendo de si el primer error se ha detectado en el tercer, cuarto, quinto o sexto bit del EOF. CANsistant debe ser capaz de autoreconfigurarse con cada nueva trama recibida.

La versión modificada de CANsistant viene modelada por la máquina de estados de la Figura 6.13. Dicha máquina de estados presenta los mismos estados que la máquina de estados del diseño original. De nuevo, el estado inicial es denominado *Idle*. La transición de *Idle* a *Waiting* viene ligada a la activación de la señal

$FirstBitEOF$ proveniente de la *Frame Monitor Unit*. Dicha señal indica que el próximo bit a recibir es el primer bit del EOF. La modificación de mayor importancia reside en el estado *Waiting*. En dicho estado se cargan diferentes valores en dos nuevas variables, denominadas $JumpBit$ y $m$, dependiendo de en que bit, a partir del tercer bit del EOF, se ha detectado el primer dominante. En el caso de detectarse el primer dominante en el primer o segundo bit del EOF, la máquina de estados salta al estado $Waiting2$. De esta forma se garantiza que CANsistant no señalice una falsa alarma ya que, como explicado anteriormente, en dicho caso es imposible que se produzcan inconsistencias reales. El resto de estados no han sufrido cambio alguno respecto a la versión original. Las transiciones sí se han visto modificadas ligeramente, viniendo ahora definidas por las nuevas variables.

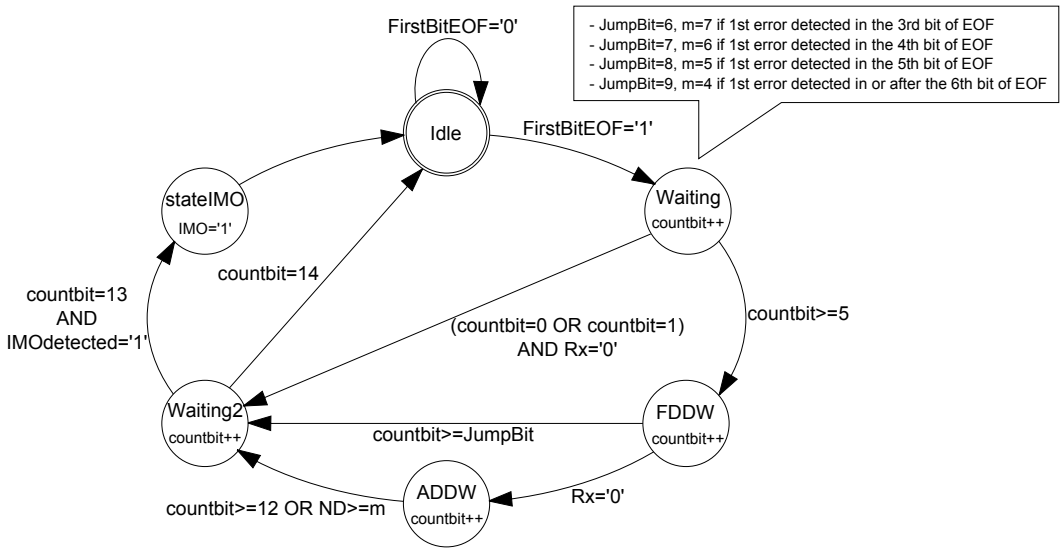



Figura 6.13: Máquina de estados de la segunda versión de CANsistant

El módulo CAN también ha sido modificado ligeramente. Estos cambios afectan la *Frame Monitor Unit*, concretamente, una de sus salidas: la señal $FirstBitEOF$. Dicha salida se ha convertido a la salida $ThirdBitEOF$, activándose ahora cuando el próximo bit es el tercer bit del EOF en lugar del primero.

### 6.4.3. Implementación en VHDL

El código VHDL del nuevo diseño de CANsistant se basa en el original, incluyendo los cambios descritos en el apartado anterior. El código, debidamente comentado,

se muestra en la página 181 del *Apéndice*.

Las entradas y salidas son idénticas a las del diseño original: la señal *Reset* reinicializa el sistema, la señal $clkR$ es el reloj de recepción, $Rx$ es la señal de recepción, la señal $FirstBitEOF$ habilita CANsistant indicando que el próximo bit recibido por $Rx$ es el primer bit del EOF y, finalmente, la salida $IMO$ señaliza una posible inconsistencia. La estructura básica del código también es parecida al del código original: la máquina de estados es implementada mediante un *process* y la función *case*. Para configurar la transición de estados, se han definido las dos variables $JumpBit$ y $m$. Dependiendo de en que bit del EOF se detecta el primer error, estas variables tomarán unos valores u otros (ver Tabla 6.1). La variable *lock* sirve para guardar la configuración actual e impedir que ésta se sobrescriba posteriormente.

| **Posición del primer error** | **JumpBit** | **m** | **Configuración** |
|---|---|---|---|
| Tercer bit del EOF | 6 | 7 | 1 |
| Cuarto bit del EOF | 7 | 6 | 2 |
| Quinto bit del EOF | 8 | 5 | 3 |
| Sexto bit del EOF o posteriores | 9 | 4 | 4 |

Cuadro 6.1: Variables de control.

### 6.4.4. Simulación

Para la primera versión de CANsistant ya se realizaron las simulaciones pertinentes para demostrar el correcto funcionamiento de la máquina de estados. Para esta segunda versión se han realizado simulaciones para verificar la configuración automática de CANsistant dependiendo de la posición en la que se ha detectado el primer error.

La primera simulación consistió en registrar si CANsistant se comporta de forma correcta si detecta un primer dominante en el primer o segundo bit del EOF. Recuérdese que en ese caso concreto, no pueden aparecer inconsistencias por lo que CANsistant no debe habilitarse. Dicha simulación se muestra en la Figura 6.14. CANsistant vigila la señal $Rx$ y detecta un '0' en el primer y segundo bit del EOF, respectivamente. La máquina de estados salta al estado $Waiting2$ y, finalmente, se

reinicializa sin activar la señal $IMO$.

En la Figura 6.15 se muestran todas las posibles configuraciones de CANsistant. En (a) se detecta un primer error en el tercer bit del EOF, en (b) se detecta en el cuarto bit del EOF, en (c) en el quinto bit del EOF y en (d) en el sexto bit del EOF. Dependiendo de la configuración, la longitud de la ventana ADDW es acortada o alargada y el parámetro $ND$ es modificado. Independientemente de la configuración y gracias al estado $Waiting2$, la señal $IMO$ es activada siempre en el mismo instante (160ns).

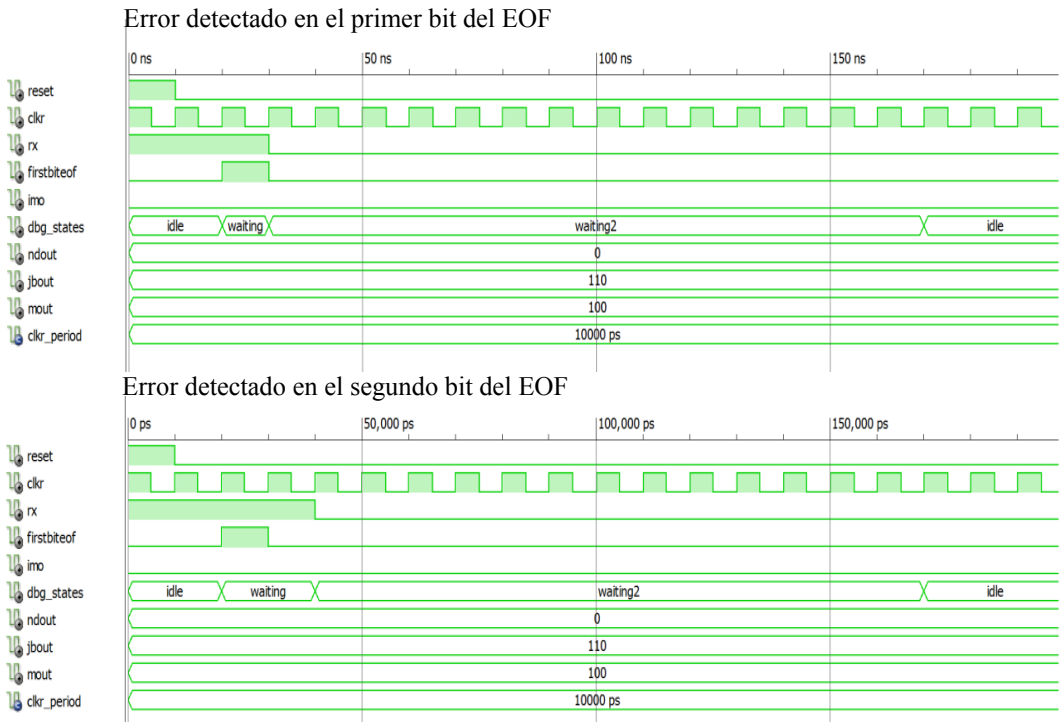



Figura 6.14: Configuración de CANsistant según la posición del primer error

En base a los resultados obtenidos se demuestra el correcto funcionamiento de esta segunda versión de CANsistant. Gracias a la configuración automática se reducen efectivamente un gran número falsas alarmas. Aunque éstas no han podido ser eliminadas por completo, sí se han podido atacar aquellas más probables. En el próximo apartado se describe una tercera versión de CANsistant, aún más potente, basada en la integración con la estrella ReCANcentrate.

## 6.5. Diseño e implementación: versión 3

En los dos apartados anteriores se han descrito dos versiones de CANsistant para topologías bus. Aunque la propuesta inicial de CANsistant fue para este tipo de topologías [PROE09], en el presente trabajo final de Máster se ha planteado también el uso de CANsistant en topologías estrella. Tomando como punto de partida la segunda versión de CANsistant, se ha rediseñado el concentrador (*hub*) de la estrella ReCANcentrate para que éste incluya las funcionalidades del mecanismo CANsistant.

La posición del mecanismo dentro del concentrador se ilustra en la Figura 6.16. Como se puede observar, la tercera versión de CANsistant no incluye el módulo CAN (que para las versiones 1 y 2, proporcionaba la señal $ThirdBitEOF$). Esto se debe a que el *hub* ya incluye funcionalidades parecidas. Concretamente, incluye el módulo denominado $Rx_CAN$, una de las salidas del cual es la señal $LastBitEOF$. Esta salida originalmente se activaba cuando el próximo bit es el último bit del EOF pero se ha convertido en la señal $ThirdBitEOF$, activándose ahora cuándo el próximo bit es el tercer bit del EOF. La entrada $Rx$ es conectada a la señal acoplada del hub $B_0$.

Durante la integración se ha dado elevada importancia a la ortogonalidad del diseño. Los cambios realizados sobre el *hub* se han mantenido a un nivel mínimo para no comprometer las funcionalidades de éste. Al monitorizar CANsistant la señal $B_0$, resultante de acoplar todas las aportaciones de los diferentes nodos, se mantienen las mismas condiciones de trabajo que en una topología bus, garantizando el correcto funcionamiento de éste.

Para la implementación en código VHDL únicamente fue necesario incluir el código de la segunda versión de CANsistant en la jerarquía de ficheros del *hub* ReCANcentrate, conectar las señales correspondientes mediante *port mapping* y convertir la señal $LastBitEOF$ en la señal $ThirdBitEOF$.

### 6.5.1. Observaciones

Si el objetivo de la integración fuera reducir aún más las falsas alarmas en lugar de garantizar la ortogonalidad, el diseño del conjunto ReCANcentrate/CANsistant

debería ser revisado y modificado completamente. Efectivamente, el *hub* tiene una visión privilegiada del sistema y dispone, por lo tanto, de mucha más información que un nodo en una topología bus. A modo de ejemplo, en lugar de monitorizar la señal acoplada $B_0$, CANsistant podría vigilar las señales de los diferentes nodos de forma independiente. Sin embargo, un bus está formado por una única línea de transmisión a la que todos los nodos tiene acceso. Así pues, un fallo físico puede afectar la línea, la conexión del nodo a la línea o el *transceiver*. En la estrella ReCANcentrate, en cambio, la conexión de los nodos al *hub* se realiza un *uplink* (UL) y un *downlink* (DL). Esto supone un problema ya que un fallo puede afectar tanto el *uplink* como el *downlink* (o los *transceivers* correspondientes) y, por lo tanto, resulta necesario revisar todos y cada uno de los escenarios de inconsistencia identificados para topologías tipo bus.

Al ya haber reducido las falsas alarmas de forma significativa mediante la segunda versión de CANsistant y, teniendo en cuenta que el objetivo final del diseño es mantener la compatibilidad total de los diferentes mecanismos con el protocolo CAN, se ha decidido mantener en diseño expuesto en el apartado anterior.

## 6.6. Conclusiones

CANsistant es un mecanismo diseñado para resolver una de las tres fuentes de inconsistencias presentes en CAN, el problema del último bit del EOF. Tomando como punto de partida la estrategia expuesta en el capítulo 3.4.3 se han propuesto tres diseños diferentes: un primer diseño para topologías bus basado en la propuesta original, un segundo diseño también para topologías bus pero mejorado para reducir el número de falsas alarmas y un último diseño, esta vez para topologías estrella. Se debe denotar que los diseños aquí presentados se han planteado, desde un principio, para detectar las inconsistencias pero no para resolverlas (por ejemplo, para resolver las inconsistencias tipo IMO, sería necesario retransmitir la trama afectada, convirtiéndose el IMO en un IMD).

Para todas las versiones se han descrito paso a paso las fases de diseño e implementación. Adicionalmente se han realizado diversas simulaciones para verificar el correcto funcionamiento de los diseños. No se ha construido un prototipo por cada una de las versiones de CANsistant propuestas por falta de tiempo. Sin embargo,

la experiencia acumulada con los anteriores prototipos realizados en este trabajo, permite confiar en que el comportamiento de CANsistant en un sistema real será idéntico al conseguido mediante simulación con las herramientas de Xilinx. En cualquier caso, la construcción de estos prototipos puede ser llevada a cabo a partir de la documentación proporcionada en esta memoria y está previsto que forme parte de algún Proyecto Final de Carrera de la EPS.

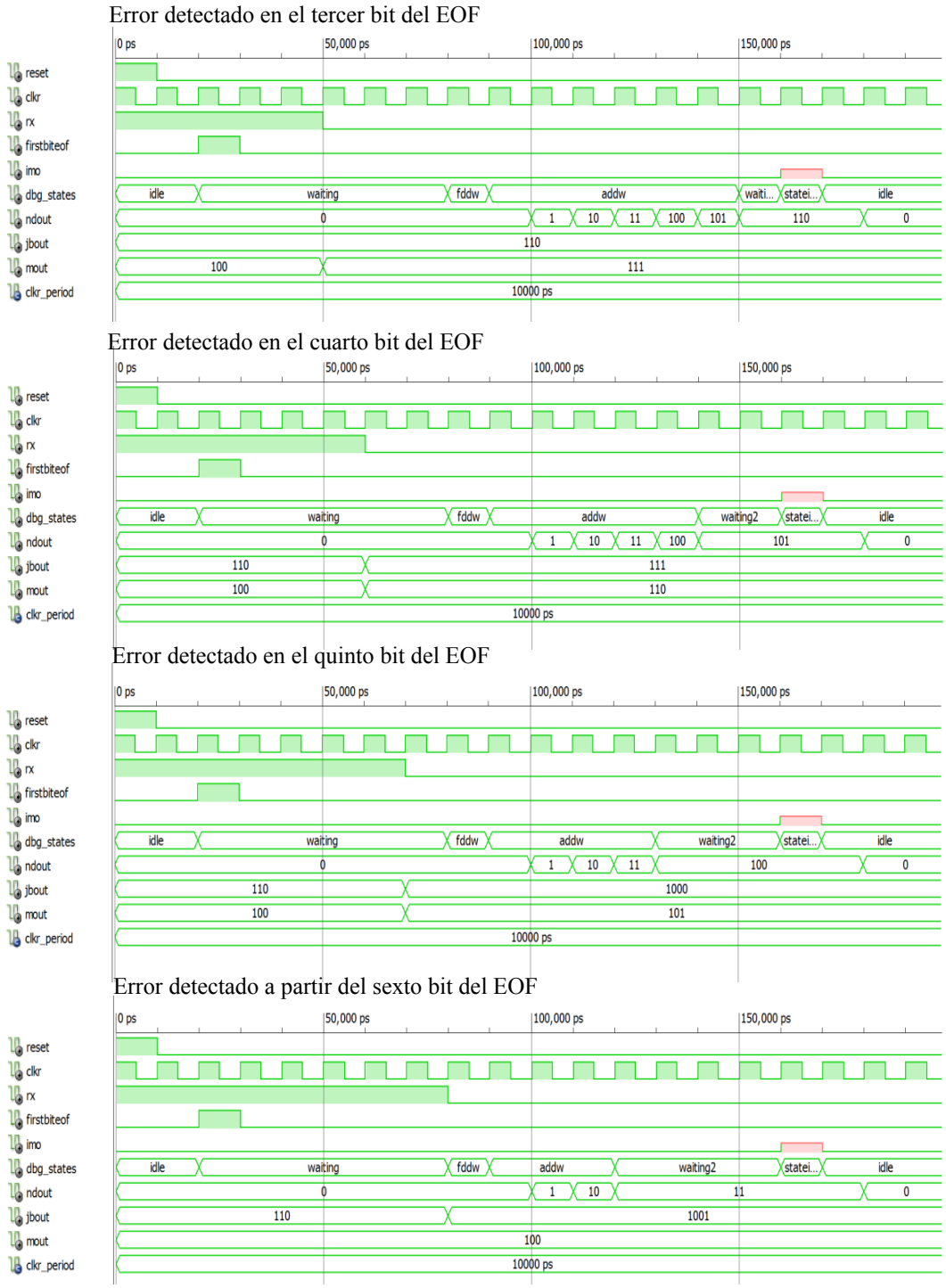



Figura 6.15: Señalización de inconsistencias según diferentes configuraciones

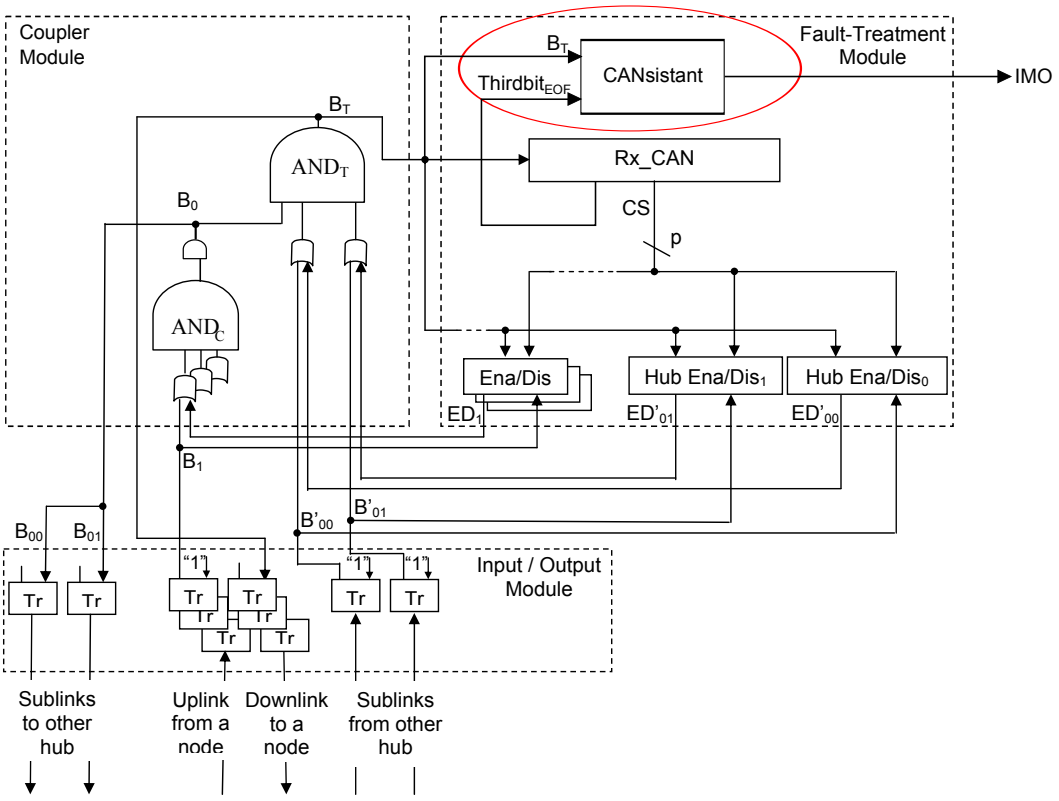



Figura 6.16: Integración del módulo CANsistant en el hub ReCANcentrate

# Capítulo 7

# Conclusiones finales

El bus CAN ha sido utilizado extensamente en muchas aplicaciones diferentes. Hay, además, un gran interés en extender su uso también en sistemas que requieren elevada garantía de funcionamiento. Sin embargo, el uso de CAN en éstas últimas es controvertido ya que, como se ha visto, muchos expertos opinan que CAN no es adecuado para aplicaciones con elevada garantía de funcionamiento, a causa de las varias limitaciones.

Aún presentando dichas limitaciones, CAN sigue acaparando un elevado interés por parte de la industria debido a su bajo coste, su robustez electromagnética, su buena respuesta en tiempo real y su amplio uso, sobretodo en las industrias de la automoción y la automatización industrial. En particular, de entre las diferentes limitaciones del protocolo CAN, el trabajo presentado en esta memoria se centra en la consistencia de datos limitada. Ésta viene ligada a ciertas vulnerabilidades del mecanismo de control de errores que implementa CAN. Como se ha mencionado en varias ocasiones, las fuentes de inconsistencias son tres: (1) la presencia del estado de error pasivo; (2) la regla del último bit del EOF y (3) la señalización inconsistente de errores. Dichas causas se han tratado exhaustivamente en esta memoria.

Este trabajo final de Máster, incluido en el proyecto CANbids, propone diferentes soluciones a las anteriores causas de inconsistencias que, usadas en su conjunto, resuelven la limitada consistencia de datos de CAN y aumentan la garantía de funcionamiento de la red. Seguidamente se exponen las conclusiones particulares de las principales aportaciones realizadas en este trabajo.

## 7.1. Contribuciones

### 7.1.1. Estudio de relevancia

Una de las aportaciones del trabajo ha sido la realización de un estudio de la relevancia de los escenarios de inconsistencia debidos a la tercera fuente de inconsistencias de CAN, la señalización inconsistente de errores. El objetivo de este estudio ha sido determinar la probabilidad de las tramas vulnerables (tramas propensas a causar inconsistencias). Para ello primeramente se han identificado las secuencias de CRC vulnerables, secuencias que al sufrir solo dos errores ya pueden generar escenarios de inconsistencia. En segundo lugar, mediante el entorno de programación *Matlab*, se han hallado las combinaciones de identificador y datos que generan secuencias de CRC vulnerables, denominadas combinaciones generadoras. El número de combinaciones representa algo más de una milésima parte del conjunto total de combinaciones por lo que es elevado. Esto pone en evidencia que la señalización inconsistente de errores es una fuente de inconsistencias relevante y, por lo tanto, no despreciable desde el punto de vista de la garantía de funcionamiento.

Adicionalmente, se ha realizado un análisis estadístico para determinar la distribución de las combinaciones generadoras respecto a los conjuntos de identificadores y de datos. El resultado fue una distribución uniforme tanto para un conjunto como para el otro. Con ello se pone en evidencia que no existen grupos de identificadores o datos especialmente propensos a generar secuencias de CRC vulnerables y que, por lo tanto, resulta necesario una solución que ataque la raíz del problema.

Para el caso del proyecto CANbids, el problema de la señalización inconsistente de errores se resuelve integrando el mecanismo AEFT, basado en la transmisión de AEFs, en la arquitectura.

### 7.1.2. AEFT

El AEFT implementa la estrategia de resolución de inconsistencias basada en AEFs y ha sido diseñado tanto para topologías bus como para topologías estrella (integrándolo con ReCANcentrate).

El dispositivo se compone por el AEFC, el EFD y el EFT. El AEFC es la unidad

de control del dispositivo encargada de generar las señales de control para los otros módulos. El EFD es un detector de señalizadores de error activos basado en un contador de bits dominantes. Finalmente, el EFT es un transmisor de *error flags* , coordinado por el AEFC, transmite el AEF.

En un bus, el AEFT se localiza entre el controlador CAN y el *transceiver* del nodo. Por este motivo, una sistema distribuido formado por $n$ nodos deberá incluir $n$ AEFTs. Para el caso de la estrella se han propuesto dos configuraciones diferentes. En la primera configuración, el AEFT, de nuevo, entre el controlador CAN y el *transceiver*), necesitándose $n$ AEFTs para $n$ nodos. En la segunda configuración, el AEFT se localiza en el hub de la estrella por lo que se necesita un único AEFT independientemente del número de nodos.

Al realizarse en ReCANcentrate la conexión de los nodos con dos cables (un *uplink* y un *downlink*) en lugar de uno, fue necesario revisar los escenarios de inconsistencia resultantes de la señalización inconsistente de errores. De entre los nuevos escenarios identificados, hubo algunos que únicamente se resolvían mediante la configuración 1.

Para la implementación física del AEFT para topologías bus se ha migrado y modifico un controlador CAN en VHDL ya existente. Explicada la estructura interna de éste, se han detallado las modificaciones y la configuración realizadas para adaptarlo a las necesidades específicas del trabajo. En ese mismo contexto se han especificado la plataforma de desarrollo y el entorno de programación utilizados. Seguidamente se ha expuesto el código VHDL escrito para implementar el AEFT. Finalmente, se han descrito el prototipo construido para realizar la verificación experimental, las pruebas realizadas y los resultados obtenidos a partir de éstas.

Para la implementación para topologías estrella se ha tenido que integrar el AEFT en el *hub* de la estrella ReCANcentrate. Seguidamente se ha montado el prototipo correspondiente y se ha realizado la verificación experimental mediante el inyector de fallos sfiCAN y el osciloscopio digital. Los resultados obtenidos mediante los (*loggers*) de sfiCAN concordaron con los obtenidos mediante el osciloscopio y demostraron la validez del diseño. Respecto a las dos configuraciones propuestas, únicamente la configuración 1 resuelve todas las inconsistencias pero su coste es mayor al de la configuración 2 ya que hacen falta un mayor número de AEFTs. La elección final de una configuración u otra depende de la aplicación.

El AEFT es un mecanismo a su vez simple como efectivo. Para ambas topologías los resultados de la verificación experimental fueron satisfactorios, demostrando que el AEFT es capaz de resolver cualquier escenario de inconsistencia debido a la señalización inconsistente de errores.

### 7.1.3. CANsistant

Se han descrito todos los pasos seguidos para el desarrollo del mecanismo CANsistant, cuya misión es detectar las inconsistecnias debidas a la segunda fuente de inconsistencias del protocolo CAN, la regla del último bit del EOF. En base a la estrategia descrita en el capítulo 3.4.3 se han propuesto tres versiones diferentes: un primer diseño para topologías bus, basado en la propuesta original; un segundo diseño también para topologías bus pero modificado para reducir el número de falsas alarmas y un último diseño, esta vez para topologías estrella. Se debe decir de nuevo que las versiones presentadas se han diseñado para detectar las inconsistencias pero no para resolverlas.

Para las tres versiones se han expuesto paso a paso las fases de diseño, implementación y verificación por simulación. Las dos primeras versiones se componen por dos bloques bien diferenciados: CANsistant propiamente dicho y el módulo CAN, cuyo propósito es proporcionar las señales necesarias para el correcto funcionamiento de CANsistant. Para la tercera versión se ha eliminado el módulo CAN, obteniéndose las señal en el interior del *hub*. La implementación en VHDL y la simulación posterior se han realizado mediante las herramientas de Xilinx. Los resultados obtenidos han sido satisfactorios y demuestran el correcto funcionamiento del mecanismo.

## 7.2. Publicaciones

El trabajo descrito en esta memoria se ha basado en diferentes publicaciones. Sobretodo, en aquellas publicadas por el propio grupo investigador. Un artículo directamente relacionado con el trabajo pero anterior a éste es el que describe el mecanismo de CANsistant:

- Julián Proenza Arenas y Ernesto Sigg. *A first design for CANsistant: A me-*

*chanism to prevent inconsistent omissions in CAN in the presence of multiple errors*. Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference pp 1-4. September 2009.

En base a los resultados obtenidos en este trabajo ha surgido la siguiente publicación:

- Guillermo Rodríguez-Navas González, Christian Peter Winter y Julián Proenza Arenas. *Injection of Aggregated Error Flags as a Means to Guarantee Consistent Error Detection in CAN*. Proceedings of the 16th International IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Toulouse, France. pp 1-4. September 2011.

Adicionalmente, se está preparando un segundo artículo de revista a partir del material de esta tesis:

- Guillermo Rodríguez-Navas González, Christian Peter Winter y Julián Proenza Arenas. *Identifying and Solving the Final Data Inconsistency Scenarios of CAN*.

## 7.3. Trabajo futuro

En base al trabajo descrito en esta memoria se pueden definir futuras tareas a realizar. Una primera tarea es la verificación experimental de CANsistant, en todas sus versiones, para demostrar que éste efectivamente detecta los escenarios de inconsistencia debidos a la regla del último bit del EOF. La segunda tarea podría consistir en la modificación de CANsistant para que éste no únicamente detecte las inconsistencias, sino que las resuelva. Esto se que podría realizar mediante un protocolo de alto nivel como por ejemplo TOTCAN [RUFI98]. Una tercera tarea podría ser la verificación formal de los módulos AEFT y CANsistant. Finalmente, otra tarea podría ser la construcción de un prototipo que implemente todas las soluciones existentes del proyecto CANbids (ReCANcentrate, sfiCAN, OCS-CAN, AEFT y CANsistant) para, posteriormente, verificarlo experimentalmente.

# Apéndices

# Apéndice A

# Códigos VHDL: Trabajo preliminar

**Unidad RAM:**

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY XilinxCoreLib;
ENTITY RAMunit IS
  PORT (clka : IN STD_LOGIC;  --System clock
        wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);  --Write enable
        addra : IN STD_LOGIC_VECTOR(3 DOWNTO 0);  --Adress
        dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0); --Data in
        douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));  --Data out
END RAMunit;

ARCHITECTURE RAMunit_a OF RAMunit IS
COMPONENT wrapped_RAMunit
  PORT (clka : IN STD_LOGIC;
        wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
        addra : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        dina : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
        douta : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END COMPONENT;
-- Configuration specification
  FOR ALL : wrapped_RAMunit USE ENTITY XilinxCoreLib.blk_mem_gen_v6_1(
        behavioral)
    GENERIC MAP (
        c_addra_width => 4,   --4 bit addressing
        c_family => "spartan3",   --Spartan 3 family
```

```
      c_read_depth_a => 16,    --16 addresses for reading
      c_read_width_a => 8,     --8 data bits
      c_rst_type => "SYNC",    --memory type=synchronous
      c_wea_width => 1,      --single write enable
      c_write_depth_a => 16,     --16 addresses for writing
      c_write_width_a => 8,    --8 data bits
      c_xdevicefamily => "spartan3"); --Spartan 3 device family
BEGIN
U0 : wrapped_RAMunit
  PORT MAP (clka => clka,
      wea => wea,
      addra => addra,
      dina => dina,
      douta => douta);
END RAMunit_a;
```

## Configuración de la velocidad de transmisión:

```
-- Bit rate configuration
-- baud rate prescaler
-- TQ = 2 * (brp + 1) / clk
brp => "000011",
tsegment1 => "000100", -- n° of TQs = tsegment1 + 1 (includes propseg)
tsegment2 => "001", -- n° of TQs = tsegment2 + 1
-- Nominal Bit Time
-- NBT = (1 + (tsegment1 + 1) + (tsegment2 + 1)) * TQ
sjw => "00", -- n° of TQs = sjw + 1
```

## Constantes modificables por el usuario:

```
--Values defined by the user
CONSTANT BYTE0:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100110"; -- Id(11..3)
CONSTANT BYTE1:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100001"; -- Id(2..0)+RTR+DLC
CONSTANT BYTE2:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100110"; -- Data0
CONSTANT BYTE3:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100001"; -- Data1
CONSTANT BYTE4:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100110"; -- Data2
CONSTANT BYTE5:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100001"; -- Data3
CONSTANT BYTE6:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100110"; -- Data4
CONSTANT BYTE7:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100001"; -- Data5
CONSTANT BYTE8:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100001"; -- Data6
CONSTANT BYTE9:STD_LOGIC_VECTOR(7 DOWNTO 0):="10100001"; -- Data7
CONSTANT NUMB:INTEGER RANGE 0 TO 7:=7; --Number of data bytes
```

# Apéndice B

# Códigos VHDL: AEFT

**Módulo superior del AEFT (Aggregated Error Flag Transmitter):**

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

entity TopAEFT is
    Port ( Tx_in:in STD_LOGIC; --Signal to the bus (from the CAN controller)
        Rx:in STD_LOGIC; --Signal from the bus
        clkR:in STD_LOGIC; --Reception clock
        clkT:in STD_LOGIC; --Transmission clock
        Reset:in STD_LOGIC; --System reset
        Tx_out:out STD_LOGIC; --Signal to the bus (transmitted by the AEFT)
        Sent1:out STD_LOGIC); --AEF succesfully sent
end TopAEFT;

architecture Behavioral of TopAEFT is
  signal EF, Sent : STD_LOGIC; --Event signals
  signal StartEFD, StartEFT : STD_LOGIC; --Control signals

  component AEFControl --Control unit of the AEFT
  port (clkR,Reset,EF,Rx,Sent: in STD_LOGIC; StartEFD, StartEFT: out STD_LOGIC
      );
  end component;
  component EFDetector --Error flag detection unit
  port (clkT,Reset,StartEFD,Tx: in STD_LOGIC; EF: out STD_LOGIC);
  end component;
  component EFTransmitter --Error Flag Transmitter
  port (clkT, Reset, StartEFT: in STD_LOGIC; Tx,Sent: out STD_LOGIC);
  end component;
```

```
begin
--Portmapping of the modules integrating the AEFT
  AEFC : AEFControl port map (clkR=>clkR, Reset=>Reset, EF=>EF, Rx=>Rx, Sent=>
      Sent, StartEFD=>StartEFD, StartEFT=>StartEFT);
  EFD : EFDetector port map (clkT=>clkT, Reset=>Reset, StartEFD=>StartEFD, Tx
      =>Tx_in, EF=>EF);
  EFT : EFTransmitter port map (clkT=>clkT, Reset=>Reset, StartEFT=>StartEFT,
      Tx=>Tx_out, Sent=>Sent);

  Sent1<=Sent; --AEF succesfully send (debugging signal)
end Behavioral;
```

## Aggregated Error Flag Controller:

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

entity AEFControl is
    Port ( Rx,clkR,Reset,EF,Sent:in STD_LOGIC; --Input signals
        StartEFD,StartEFT:out STD_LOGIC); --Control signals
end AEFControl;

architecture Behavioral of AEFControl is
  constant m : INTEGER := 5;  --Number of AEFs transmitted
  constant del : INTEGER:=11; --Frame delimiter (used for resetting the AEFT)
  type states is (s0,s1,s2,s3,s4,s5,s6,s7); --FSM with 8 states
  signal statemachine : states;
begin
  process (clkR, Reset) --Process implementing the FSM
  variable transEF : INTEGER :=0;
  variable numRec : INTEGER:=0;
  begin
    if Reset = '0' then statemachine <= s0; transEF := 0; numRec := 0;
    elsif clkR'event and clkR = '1' then
      case statemachine is
        when s0 => statemachine <= s1;
        when s1 => statemachine <= s2;
        when s2 => if EF = '1' then statemachine <= s3;
                   else statemachine <= s2;
                   end if;
        when s3 => if Rx = '1' then statemachine <= s4;
                   else statemachine <= s3;
                   end if;
        when s4 => statemachine <= s5;
        when s5 => if Sent='1' then statemachine <= s6; transEF := transEF+1;
                   else statemachine <= s5;
                   end if;
        when s6 => if (transEF<m) then statemachine <= s3;
                   else statemachine <= s7;
```

```
                end if;
        when s7 => if  (numRec<del) then
                    if Rx='1' then statemachine <= s7; numRec:=numRec+1;
                    else statemachine <= s7; numRec:=0;
                    end if;
                   else statemachine <= s1;
                    end if;
      end case;
    end if;
  end process;
  with statemachine select StartEFD <= '1' when s1, '0' when others;
  with statemachine select StartEFT <= '1' when s4, '0' when others;
end Behavioral;
```

## Error Flag Transmitter:

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

entity EFTransmitter is
    Port(StartEFT:in  STD_LOGIC; --Control signal from the AEFC
      clkT:in STD_LOGIC; --Transmission clock
      Reset:in STD_LOGIC; --System reset
         Sent:out  STD_LOGIC; --Event signal (an AEF has been successfully
             sent)
         Tx:out  STD_LOGIC); --Signal to the bus (from the AEFT)
end EFTransmitter;

architecture Behavioral of EFTransmitter is
  type states is (s0, s1, s2, s3, s4, s5, s6, s7); --FSM with 8 states
  signal statemachine : states;

begin
  process (clkT, Reset) --Process implementing the FSM
  begin
    if Reset = '0' then statemachine <= s0;
    elsif clkT'event and clkT = '1' then
      case statemachine is
        when s0 => if StartEFT = '1' then statemachine <= s1;
          else statemachine <= s0;
          end if;
        when s1 => statemachine <= s2;
        when s2 => statemachine <= s3;
        when s3 => statemachine <= s4;
        when s4 => statemachine <= s5;
        when s5 => statemachine <= s6;
        when s6 => statemachine <= s7;
        when s7 => statemachine <= s0;
```

```
      end case;
    end if;
  end process;
  --Transmision of a recessive bit and an active EF (6 consecutive dominant
      bits)
  Tx <= '1'   when statemachine=s1 or statemachine=s0 else '0' when
      statemachine=s2 or statemachine=s3
            or statemachine=s4 or statemachine=s5 or statemachine=s6
            or statemachine=s7;
  with statemachine select
    Sent <= '1' when s7, '0' when others;
end Behavioral;
```

**Error Flag Detector**:

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

entity EFDetector is
    Port ( Tx:in  STD_LOGIC; --Signal to the bus (from the CAN controller)
        clkT:in  STD_LOGIC; --Transmision clock
        StartEFD:in  STD_LOGIC; --Control signal from the AEFC
        Reset:in  STD_LOGIC; --System reset
        EF:out  STD_LOGIC); --Event signal (EF detected)
end EFDetector;

architecture Behavioral of EFDetector is
  type states is (s0,s1,s2,s3,s4,s5,s6,s7); --FSM with 8 states
  signal statemachine: states;
begin
  process (clkT, Reset) --Process implementing the FSM
  begin
    if Reset = '0' then statemachine <= s0;
    elsif clkT'event and clkT = '1' then
      case statemachine is
        when s0 => if StartEFD = '1' then statemachine <= s1;
          else statemachine <= s0;
          end if;
        when s1 => if Tx = '0' then statemachine <= s2;
          else statemachine <= s1;
          end if;
        when s2 => if Tx = '0' then statemachine <= s3;
          else statemachine <= s1;
          end if;
        when s3 => if Tx = '0' then statemachine <= s4;
          else statemachine <= s1;
          end if;
        when s4 => if Tx = '0' then statemachine <= s5;
          else statemachine <= s1;
```

```
          end if;
        when s5 => if Tx = '0' then statemachine <= s6;
          else statemachine <= s1;
          end if;
        when s6 => if Tx = '0' then statemachine <= s7;
          else statemachine <= s1;
          end if;
        when s7 => statemachine <= s0;
      end case;
    end if;
  end process;
  --Output enable
  with statemachine select EF <= '1' when s7, '0' when others;
end Behavioral;
```

# Apéndice C

# Códigos VHDL: CANsistant

**CANsistant:**

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; use work.defStates.all;

entity CANsistant1 is
port( Reset:     in std_logic; -- Reset
      clkR:      in std_logic; -- Reception clock
      Rx:      in std_logic; -- Value of received bit
      FirstbitEOF:in std_logic; -- Enable of the module
      IMO:       out std_logic; -- Output (IMO detected)
      dbg_states: out CANsistant_state;
      NDout: out integer range 1 to 3);
end CANsistant1;

architecture Behavioral of CANsistant1 is
  --type CANsistant_state is (idle, waiting, FDDW, ADDW, waiting2, stateIMO);
  signal statemachine: CANsistant_state;
  signal countbit: integer range 0 to 7;
  signal ND: integer range 0 to 3;
  signal inconsistency: std_logic;
begin
IMO <= inconsistency;
dbg_states <= statemachine;
NDout <= ND;
process (clkR, Reset)
  variable IMOdetected : std_logic;
  variable NDvar : integer range 0 to 3;
  begin
  if Reset = '1' then
```

```
    statemachine <= Idle; -- Asynchronous reset
    countbit <= 0;
    ND <= 0;
    IMO <= '0';
    IMOdetected := '0';
    NDvar := 0;
  elsif clkR'event and clkR = '1' then
    case statemachine is -- FSM
    when Idle =>
      countbit <= 0; -- Reset of all signals and variables
      ND <= 0;
      IMO <= '0';
      IMOdetected := '0';
      NDvar := 0;
      if FirstbitEOF = '1' then statemachine <= waiting; -- Enable CANsistant
      else statemachine <= Idle;
      end if;
    when waiting => -- Count bits till reaching FDDW
      if countbit >= 5 then statemachine <= FDDW;
      else  statemachine <= waiting;
      end if;
      countbit <= countbit + 1;
    when FDDW => -- First Dominant Detection Window
      if Rx = '0' then statemachine <= ADDW; -- First dominant detected
      elsif countbit >= 9 then statemachine <= waiting2;
      else statemachine <= FDDW;
      end if;
      countbit <= countbit + 1;
    when ADDW => -- Additional Dominant Detection Window
      if Rx = '0' then
        ND <= ND + 1; -- Dominants increase ND counter
        NDvar := NDvar + 1;
      end if;
      if NDvar >= 3 then
        IMOdetected := '1'; -- There is a possible IMO
        statemachine <= waiting2;
      elsif countbit >= 12 then   statemachine <= waiting2;
      else statemachine <= ADDW;
      end if;
      countbit <= countbit + 1;
    when waiting2 => -- Delaying state to control the output time
      if countbit >= 13 and IMOdetected = '1' then statemachine <= stateIMO;
      elsif countbit >= 14 then statemachine <= idle; -- No IMO
      else statemachine <= waiting2;
      end if;
      countbit <= countbit + 1;
    when stateIMO => -- Notify IMO and return to Idle
      statemachine <= idle;
    end case;
  end if;
```

```
end process;
with statemachine select IMO <= '1' when stateIMO, '0' when others;
end Behavioral;
```

**Frame Monitor:**

```
library IEEE; use work.defInteger.all; use work.defStates.all;
use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FrameMonitor is
port( reset:     in std_logic; -- Reset
      stuffbit: in std_logic; -- Next bit is a stuff bit (from the stuff unit)
      stuffvalue: in std_logic; -- Stuff bit value (from the stuff unit)
      clkR:      in std_logic; -- Reception clock
      Rx:      in std_logic; -- Value of received bit
      CRCvalue: in std_logic_vector(14 downto 0); -- CRC value
      StuffEnable:out std_logic; -- Enable stuff unit
      error:     out std_logic; -- Enable EF generator
      FirstBitEOF:out std_logic; -- Enable CANsistant
      FrameState: out GlobalFrameState; -- Frame field
      BitNumber:  out ENTER64; -- Frame bit
      CRCerror: out std_logic); -- CRC error
end FrameMonitor;

architecture Behavioral of FrameMonitor is
  signal intFrameState: GlobalFrameState; -- Frame field
  signal countBit: ENTER64; -- Bit counter
  signal intdlcValue: ENTER9; -- DLC value in decimal
  signal dlcValue: std_logic_vector (3 downto 0); -- DLC value in binary
  signal seqCRC: std_logic_vector (14 downto 0); -- Received CRC
  signal regCRCValue: std_logic_vector(14 downto 0); -- Calculated CRC
  signal FrameType, auxStuffEnabling, auxCRCerror: std_logic;
begin
  FrameState  <= intFrameState; BitNumber <= countBit;
  StuffEnable <= auxStuffEnabling; CRCerror <= auxCRCerror;
  with dlcValue select -- Binary to decimal conversion of DLC value
    intdlcValue <=  0 when "0000",1 when "0001",2 when "0010",3 when "0011",
    4 when "0100", 5 when "0101",6 when "0110",7 when "0111",8 when "1000",
    0 when others;
  process (clkR, reset)
    variable indexSeqCRC: ENTER15;
  begin
    if reset = '1' then -- System reset
      intFrameState <= idle; countBit <= 0; auxStuffEnabling <= '0';
      error <= '0'; FirstBitEOF <= '0'; auxCRCerror <= '0';
    elsif clkR'event and clkR = '1' then
      if stuffbit = '0' then -- No stuff bit
```

```
case (intFrameState) is -- Finite State Machine
  when idle =>
    if Rx = '0' then -- SOF detected
      intFrameState <= idField; countBit <= 0; -- Bit counting
      auxStuffEnabling <= '1'; -- Enable the stuff unit
    end if;
  when idField =>
    -- See if ID field is finished
    if countBit < fieldLong(idField)-1 then countBit <= countBit + 1;
    else intFrameState <= rtrField; countBit <= 0;
    end if;
  when rtrField =>  -- Determine the frame type
    intFrameState <= resField; countBit <= 0; FrameType <= Rx;
  when resField => -- Check if field is finished
    if countBit < fieldLong(resField)-1 then countBit <= countBit + 1;
    else intFrameState <= dlcField; countBit <= 0;
     end if;
  when dlcField => -- Get the DLC value
    case (countBit) is
      when 0 => dlcValue  (3) <= Rx; countBit <= countBit + 1;
      when 1 => dlcValue  (2) <= Rx; countBit <= countBit + 1;
      when 2 => dlcValue  (1) <= Rx; countBit <= countBit + 1;
      when 3 => dlcValue  (0) <= Rx;
        -- Jump to next field depending on the frame type
        if FrameType = '0' and not (dlcValue(3 downto 1) = "000"
        and Rx = '0') then intFrameState <= dataField;
        else intFrameState <= crcField;
        end if;
        countBit <= 0; indexSeqCRC := 14;
        regCRCValue (14 downto 0) <= CRCvalue(14 downto 0);
      when others => null;
    end case;
  when dataField => -- Check if data field is finished
    if countBit < (intdlcValue * 8)-1 then countBit <= countBit + 1;
    else intFrameState <= crcField; countBit <= 0; indexSeqCRC := 14;
    end if;
  when crcField => -- Get CRC
    seqCRC(indexSeqCRC) <= Rx;
    if countBit < fieldLong(crcField)-1 then
      if countBit = 0 then -- The CRC calculator deliveres the CRC
        regCRCValue (14 downto 0) <= CRCvalue(14 downto 0);
      end if;
      indexSeqCRC := indexSeqCRC - 1; countBit <= countBit + 1;
    else intFrameState <= crcDelimField;
      countBit <= 0; auxStuffEnabling <= '0';
    end if;
  when crcDelimField =>
    if seqCRC(14 downto 0) = regCRCValue(14 downto 0) then
      auxCRCerror <=  '0'; -- Received CRC=calculated CRC?
    else auxCRCerror <= '1';
```

```
          end if;
          if Rx = '1' then intFrameState <= ackSlotField; countBit <= 0;
          else intFrameState <= errorFlag;
            countBit <= 0; error <= '1'; auxStuffEnabling <= '0';
          end if;
        when ackSlotField => intFrameState <= ackDelimField; countBit <= 0;
        when ackDelimField => -- If there is no error, jump to EOF
          if Rx = '1' and auxCRCerror = '0' then intFrameState <= eofField;
            countBit <= 0; FirstBitEOF <= '1';
          else intFrameState <= errorFlag; -- In case of an error, send EF
            countBit <= 0; error <= '1'; auxStuffEnabling <= '0';
          end if;
        when eofField =>
          if Rx = '1' then -- Check if EOF is complete
            if countBit < fieldLong(eofField)-1 then
              countBit <= countBit + 1;
            else  intFrameState <= interField; countBit <= 0;
            end if;
          else intFrameState <= errorFlag;
            countBit <= 0; error <= '1'; auxStuffEnabling <= '0';
            FirstBitEOF <= '0';
          end if;
        when interField =>
          if Rx = '1' then -- Check if IFS is finished
            if countBit < fieldLong(interField)-1 then
              countBit <= countBit + 1;
            else  intFrameState <= idle; countBit <= 0;
            end if;
          else intFrameState <= errorFlag;
            countBit <= 0; error <= '1'; auxStuffEnabling <= '0';
          end if;
        when errorFlag => error <= '0';  countBit <= countBit + 1;
          if Rx = '1' then intFrameState <= errorDelimiter; countBit <= 1;
          end if;
        when errorDelimiter =>
          if Rx = '1' then -- Check if error delimiter is finished
            if countBit < fieldLong(errorDelimiter)-1 then
              countBit <= countBit + 1;
            else  intFrameState <= interField; countBit <= 0;
            end if;
          else intFrameState <= errorFlag;
            countBit <= 0; error <= '1'; auxStuffEnabling <= '0';
          end if;
      end case;
    else -- Evaluate stuff bit errors
      if stuffvalue /= Rx then intFrameState <= errorFlag; -- Send EF
        countBit <= 0; error <= '1'; auxStuffEnabling <= '0';
      end if;
    end if;
  end if;
```

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

**Stuff Unit:**

```
library IEEE; use IEEE.std_logic_1164.ALL; use IEEE.std_logic_ARITH.ALL;
use IEEE.std_logic_UNSIGNED.ALL; use work.defInteger.all;

entity stuffUnit1 is
port( reset:      in std_logic; -- Reset
      enable:     in std_logic;     -- Enable
      clkR:       in std_logic; -- Reception clock
      Rx:         in std_logic; -- Value of received bit
      stuffbit:   out std_logic;  -- Next bit is a stuff bit
      stuffvalue: out std_logic   -- Estimated value of the stuff bit
    );
end stuffUnit1;

architecture Behavioral of stuffUnit1 is
  signal counter: ENTER16;
begin
  process(clkR, reset)
  begin
    if reset = '1' then
      -- Initial state of the FSM, we asume there is a '1' on the bus
      stuffbit <= '0'; stuffvalue <= '0'; counter <= 6;
    elsif clkR'event and clkR = '1' then
      if enable = '1' then --Module is enabled
        case counter is -- '0/1' counter
          when 0 =>
            if Rx = '0' then counter <= 1; -- 1st dominant
            else counter <= 6;
            end if;
            stuffbit <= '0';
          when 1  =>
            if Rx = '0' then counter <= 2; -- 2 dominants
            else counter <= 6;
            end if;
            stuffbit <= '0';
          when 2  =>
            if Rx = '0' then counter <= 3; -- 3 dominants
            else counter <= 6;
            end if;
            stuffbit <= '0';
          when 3  =>
            if Rx = '0' then counter <= 4; -- 4 dominants
            else counter <= 6;
            end if;
```

```
            stuffbit <= '0';
          when 4  =>
            if Rx = '0' then counter <= 0; -- 5 dominants
              stuffbit <= '1';  -- Next bit is a stuff bit!
              stuffvalue <= '1'; -- The stuff bit is recessive
            else  counter <= 6;
                stuffbit <= '0';
            end if;
          when 6  =>
            if Rx = '1' then counter <= 7; -- 2 recessives
            else counter <= 1;
            end if;
            stuffbit <= '0';
          when 7  =>
            if Rx = '1' then counter <= 8; -- 3 recessives
            else counter <= 1;
            end if;
            stuffbit <= '0';
          when 8  =>
            if Rx = '1' then counter <= 9; -- 4 recessives
            else counter <= 1;
            end if;
            stuffbit <= '0';
          when 9 =>
            if Rx = '1' then counter <= 0; -- 5 recessives
              stuffbit <= '1';  -- Next bit is a stuff bit!
              stuffvalue <= '0'; -- The stuff bit is dominant
            else  counter <= 1;
                stuffbit <= '0';
            end if;
          when others => stuffbit <= '0';
                      counter <= 0;
        end case;
      else -- Module is not enabled
        stuffbit <= '0';
        stuffvalue <= '0';
        if Rx = '0' then counter <= 1; -- The received bit is dominant
        else counter <= 6; -- The received bit is recessive
        end if;
      end if; -- End enable
    end if; -- End event
  end process;
end Behavioral;
```

## CRC Calculator:

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; use work.defStates.all;
```

```
entity CRCcalculator is
port( reset:      in std_logic; -- Reset
      clkR:       in std_logic; -- Reception clock
      Rx:         in std_logic; -- Value of received bit
      FrameState: in GlobalFrameState; -- Frame field
      stuffbit:   in std_logic; -- Received bit is a stuff bit
      CRCvalue:   out std_logic_vector (14 downto 0) -- Final CRC of the frame
    );
end CRCcalculator;

architecture Behavioral of CRCcalculator is
  signal enable, auxflip: std_logic;
begin
  with FrameState select -- Enable CRC calculation with frame fields
    enable <= '1' when idField | rtrField | resField | dlcField | dataField,
    '0' when others;
  process (clkR)
    variable crcNext: std_logic;
    variable shiftReg: std_logic_vector(14 downto 0); -- CRC value
  begin
    if reset = '1' then --Reset all signals and variables
      shiftReg := "000000000000000"; crcNext := '0'; auxflip <= '0';
    elsif clkR='1' and clkR'event then
      if enable = '0' then --check if CRC calculator is enabled
        shiftReg := "000000000000000";
        crcNext := '0';
      elsif stuffbit = '0' then
        auxflip <= not auxflip; -- Calculate CRC with generator algorithm
         crcNext:= Rx xor shiftReg(14);
        shiftReg(14 downto 1) := shiftReg(13 downto 0);
         shiftReg(0):= '0';
        if crcNext = '1' then
          shiftReg(14 downto 0) := shiftReg(14 downto 0) xor "100010110011001"
        end if;
      end if;
    end if;
    CRCvalue <= shiftReg; --Output CRC value
  end process;
end Behavioral;
```

## Error Flag Generator:

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; use work.defInteger.all;

entity errorFrameGenerator1 is
port (  reset:      in std_logic; -- Reset
```

```
        clkT:       in std_logic; -- Transmission clock
        error:      in std_logic; -- Erroneous bit
        errorflag:  out std_logic); -- Value transmitted
end errorFrameGenerator1;

architecture Behavioral of errorFrameGenerator1 is
  type state is (idle, errorFlag);
  signal GenState: state;
  signal countD: ENTER7;
begin
  -- FSM controlling the generator
  process (clkT, reset)
  begin
    if reset = '1' then -- Reset system
      GenState <= idle; errorflag <= '1';
    elsif clkT'event and clkT = '1' then
      case (GenState) is
        when idle =>
          if error = '0' then
            errorflag <= '1'; -- No error so send recessive
          else
            GenState <= errorFlag;
            errorflag <= '0'; -- Error so send first dominant of EF
            countD <= 1; -- Count dominants
          end if;
        when errorFlag =>
          if error = '1' then
            errorflag <= '0'; -- Restart EF
            countD <= 1;
          elsif countD < 6 then
            countD <= countD + 1; -- Send EF (6 dominant bits)
            errorflag <= '0';
          else  GenState <= idle; -- When finished return to idle
              errorflag <= '1'; -- Send recessive
          end if;
        when others => null;
      end case;
    end if;
  end process;
end Behavioral;
```

## CANsistant (versión 2):

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL; use work.defStates.all;

entity CANsistant_V2 is
port( Reset:     in std_logic; -- Reset
```

```
      clkR:      in std_logic; -- Reception clock
      Rx:      in std_logic; -- Value of received bit
      FirstbitEOF:in std_logic; -- Enable of the module
      IMO:       out std_logic; -- Output (IMO detected)
      dbg_states: out CANsistant_state;
      NDout: out integer range 0 to 6);
end CANsistant_V2;

architecture Behavioral of CANsistant_V2 is
  signal statemachine: CANsistant_state;
  signal countbit: integer range 0 to 14;
  signal ND: integer range 0 to 6;
  signal inconsistency: std_logic;
begin
IMO <= inconsistency;
dbg_states <= statemachine;
NDout <= ND;

process (clkR, Reset)
  variable IMOdetected: STD_LOGIC;
  variable JumpBit: INTEGER RANGE 6 TO 9;
  variable m: INTEGER RANGE 4 TO 7;
  variable NDvar : integer range 0 to 6;
  variable lock : std_logic;
begin
  if Reset = '1' then
    statemachine <= Idle; -- Asynchronous reset
    countbit <= 0;
    ND <= 0;
    IMO <= '0';
    IMOdetected := '0';
    NDvar := 0;
    m := 4;
    JumpBit := 9;
    lock := '0';
  elsif clkR'event and clkR = '1' then
    case statemachine is -- FSM
    when Idle =>
      countbit <= 0; -- Reset of all signals and variables
      ND <= 0;
      IMO <= '0';
      IMOdetected := '0';
      NDvar := 0;
      m := 4; -- Initialize with values of original CANsistant design
      JumpBit := 9;
      lock := '0';
      if FirstbitEOF = '1' then statemachine <= waiting; -- Enable
      else statemachine <= Idle;
      end if;
    when waiting =>
```

```
    if lock = '0' then
      -- Change configuration of FSM depending on where the first error is
          detected
      if countbit = 2 and Rx = '0' then
        JumpBit := 6; -- 1st error in the 3rd bit of EOF
        m := 7;
        lock := '1';
      elsif countbit =3 and Rx = '0' then
        JumpBit := 7; -- 1st error in the 4th bit of EOF
        m := 6;
        lock := '1';
      elsif countbit = 4 and Rx = '0' then
        JumpBit := 8; -- 1st error in the 5th bit of EOF
        m := 5;
        lock := '1';
      elsif countbit >= 5 and Rx = '0' then
        JumpBit := 9; -- 1st error in or after the 6th bit of EOF
        m := 4;
        lock := '1';
      end if;
    end if;
    -- Count bits till reaching FDDW
    if (countbit = 0 or countbit = 1) and Rx = '0' then
      statemachine <= waiting2;
    elsif countbit >= 5 then
      statemachine <= FDDW;
    else  statemachine <= waiting;
    end if;
    countbit <= countbit + 1;
  when FDDW => -- First Dominant Detection Window
    if Rx = '0' then statemachine <= ADDW; -- First dominant detected
    elsif countbit >= JumpBit then statemachine <= waiting2;
    else statemachine <= FDDW;
    end if;
    countbit <= countbit + 1;
  when ADDW => -- Additional Dominant Detection Window
    if Rx = '0' then
      ND <= ND + 1; -- Dominants increase ND counter
      NDvar := NDvar + 1;
    end if;
    if NDvar >= (m-1) then IMOdetected := '1'; -- There is a possible IMO
      statemachine <= waiting2;
    elsif countbit >= 12 then   statemachine <= waiting2;
    else statemachine <= ADDW;
    end if;
    countbit <= countbit + 1;
  when waiting2 => -- Delaying state to control the output time
    if countbit >= 13 and IMOdetected = '1' then statemachine <= stateIMO;
    elsif countbit >= 14 then statemachine <= idle; -- No IMO
    else statemachine <= waiting2;
```

```
      end if;
      countbit <= countbit + 1;
    when stateIMO => -- Notify IMO and return to Idle
      statemachine <= idle;
    end case;
  end if;
end process;
with statemachine select inconsistency <= '1' when stateIMO, '0' when others;
end Behavioral;
```

# Bibliografía

[AL81] T. Anderson and P.A. Lee. *Fault Tolerance : Principles and Practice*. Prentice Hall, 1981.

[ALM02] L. Almeida, P. Pedreiras and J. A. Fonseca. The FTT-CAN protocol: Why and how. *IEEE Transactions on Industrial Electronics, 49(2)*, December 2002.

[Avi95] Algirdas Avizienis. Building dependable systems: How to keep up with complexity. In *Special Issue of the IEEE 25th Int. Symp. Fault-Tolerant Computing. FTCS-25. Pasadena, CA*, pages 4-14, June 27-30 1995.

[BALL11] A. Ballesteros, *Construction of sfiCAN: a star-based fault-injection infrastructure for the Controller Area Network*. Palma de Mallorca 2012.

[BARR06a] M. Barranco, J. Proenza, G. Rodríguez-Navas and L. Almeida. An active star topology for improving fault confinement in CAN networks, *IEEE Transactions on Industrial Informatics, vol. 2, num. 2, USA*, May 2006.

[BARR06b] M. Barranco, L. Almeida and J. Proenza. ReCANcentrate: A replicated star topology for CAN networks. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005), Catania, Italy*, 2005.

[BARR09] M. Barranco. Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate. *Computer, Flagship Publication of the IEEE Computer Society, Volume 42, Number 5*. May 2009.

[BMD93] Michael Barborak, Miroslaw Malek, and Anton Dahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171-220, June 1993.

[Car82] W.C. Carter. A time for reflection. In *Proceedings of the IEEE 12th Int. Symp. Fault-Tolerant Computing. FTCS-12. Santa Monica, California, USA*, June 1982.

[CAV05] S. Cavalieri. Meeting Real-Time Constrains in CAN. *IEEE Transactions on Industrial Informatics, pages 124- 135*, May 2005.

[CEN01] G. Cena, L. Durante, and A. Valenzano. A new CAN-like field network based on a star topology. *Computer Standards and Interfaces, vol. 23, issue 3*, July 2001.

[CiAa] CiA. CAN data link layer. Technical report, CAN in Automation (CiA), Am Weichselgarten 26.

[CiAb] CiA. CAN physical layer. CAN in Automation (CiA), Am Weichselgarten 26, Technical Report.

[DS83] D. Dolev and H. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656-666, 1983.

[Ets01] Konrad Etschberger. *Controller Area Network. Basics, Protocols, Chips and Applications*. Germany, 2001.

[FER05] J. Ferreira, L. Almeida and J.A. Fonseca. Bus Guardians for CAN: a Taxonomy and a Comparative Study. *Proc. of WDAS 2005, Workshop on Dependable Automation Systems, Salvador, Brazil*. October 2005.

[Flex05] FlexRay. *FlexRay Communications System Protocol Specification Version 2.1*, 2005.

[Flo03] Thomas L. Floyd. *Fundamentos de sistemas digitales*. Editorial Pearson Prentice Hall. Madrid, 2003.

[FOFF04] J. Ferreira, A. Oliveira, P. Fonseca and J. Fonseca. An Experiment to Assess Bit Error Rate in CAN. In *Proceedings of the 3rd International Workshop on Real-Time Networks, Catania, Italy*, 2004.

[GAW09] H. Guo, J.J. Ang and Y. Wu. Estracting Controller Area Network data for reliable car communications. In *IV IEEE Intelligent Vehicles Symposium, Xi'an, China*, June 2009.

[GESS11] D. Gessner, M. Barranco, A. Ballesteros and J. Proenza. Designing sfi-CAN: a star-based physical fault inyector for CAN. In *Proceedings of the 16th International IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Toulouise, France*. pp 1-4. September 2011

[Gmb91] Robert Bosch GmbH. CAN Specification, Version 2.0, 1991.

[IEEE93] ANSI/IEEE Standard 1076-1993. IEEE Standard VHDL Language Reference Manual. The Institute of Electrical and Electronics Engineers, Inc. 345 East 47th Street, New York, NY 10017-2394, USA.

[ISO93] ISO. ISO11898. Road vehicles-Interchange of digital information-Controller Area Network (CAN) for high-speed communication, 1993.

[ISO96e] International Standard ISO/IEC 14977. Information technology. Syntactic metalanguage. Extended BNF, 1996.

[ISO03a] ISO. ISO11898-1. Controller Area Network (CAN). Part 1: Data link layer and physical signalling, 2003.

[ISO03b] ISO. ISO11898-2. Controller Area Network (CAN). Part 2: Highspeed medium access unit, 2003.

[IXX05] IXXAT. Innovative products for industrial and automotive communication systems, 2005. [Online]. Available: http://www.ixxat.de/index.php.

[Joh89] Barry W. Johnson. *Design and Analysis of Fault Tolerant Digital Systems.* Addison-Wesley Publishing Company, 1989.

[KOPE99] H.Kopetz, A Comparison on CAN and TTP, HK 1998-99.

[Lap92] Jean-Claude Laprie. *Dependability: Basic Concepts and Terminology.* Springer-Verlag Wien New York, 1992.

[LIV99] M. Livani. SHARE: A transparent approach to fault-tolerant broadcast in CAN. In *Proceedings of the 6th International CAN Conference*, 1999.

[LMJ91] L. Laranjeira, M. Malek, and R. Jenevein. On tolerating faults in naturally redundant algorithms. In *Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems. Pisa, Italy*, pages 118-127, September 1991.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382-401, July 1982.

[MIC04] PIC18FXX8 Data Sheet - 28/40-Pin High Performance, Enhanced Flash Microcontrollers with CAN Module, 2004.

[NOL05] T. Nolte, M. Nolin and H.A. Hansson. Real-Time Server-Based Communications with CAN. *IEEE Transactions on Industrial Informatics, pages 192 - 201*, August 2005.

[PHI00] PHILIPS. Data sheet - PCA82C250 - CAN controller interface - Product Specification, 2000.

[PIME04] J.R. Pimentel and J.A. Fonseca. FlexCAN: A Flexible Arquitecture for highly dependable embedded applications, RTN 2004-3rd Int. *Workshop on Real-Time Networks*, held in conjunction with the *16th Euromicro Intl. Conference on Real-Time Systems, Catania, Italy*, June 2004.

[PIME08] J. Pimentel, J. Proenza, L. Almeida, G. Rodríguez-Navas, M. Barranco and J. Ferreira. Dependable automotive CAN networks. *Handbook of Automotive Embedded Systems, N. Navet and F. Simonot-Lion, Pages 6â€“(1â€“51), CRC Press, 2009.*

[Pol96] S. Poledna. System model and terminology. In Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism, Real-Time Systems, chapter 3. In *Engineering and Computer Science*, pages 21-30. Kluwer Academic Publishers, Boston, Dordrecht, London, 1996.

[Pow92] David Powell. Failure mode assumptions and assumption coverage. In *Digest of Papers of the IEEE 22th Int. Symp. Fault-Tolerant Computing FTCS-22*, pages 386-395, Boston, Massachusetts-USA, July 1992.

[PROE00] J. Proenza and J. Miro. MajorCAN. A Modification to the Controller Area Protocol to Achieve Atomic Broadcast *ICDCS Workshop on Grop Communications and Computations*, 2000.

[PROE09] J. Proenza and E. Sigg. A first design for CANsistant: A mechanism to prevent inconsistent omissions in CAN in the presence of multiple errors. In

*Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference pp 1-4. September 2009.*

[PS11] PEAK-System Technik GmbH. *PCAN PCI to CAN Interface User Manual V2.1.0*, 2011.

[ROC08] S. Roca. *Implementació i test d'un sistema de sincronització de rellotge sobre CAN*. Proyecto final de carrera. Ingenieria Tecnica Industrial, especialidad en Electronica Industrial. G. Rodriguez-Navas (Supervisor). Universitat de les Illes Balears. Palma de Mallorca. Spain.

[RODR03] G.Rodríguez-Navas and J.Proenza. Analyzing Atomic Broadcast in TT-CAN Networks. In *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications (FET 2003), Aveiro, Portugal*, pp 153-156. 2003.

[RODR03b] G. Rodríguez-Navas, J. Jimenez and J. Proenza. An architecture for physical injection of complex fault scenarios in CAN networks. In *Proceedings of the IEEE Conference in Emerging Technologies and Factory Automation, pp. 125-128 vol.2*. September 2003.

[RODR03c] G. Rodríguez-Navas, J. Bosch and J. Proenza. Hardware Design of a High-Precision and Fault-Tolerant Clock Subsystem for CAN Networks. *5th IFAC International Conference on Fieldbus Systems and their Applications (FET'03), Aveiro (Portugal)*. 2003.

[RODR10] G. Rodríguez-Navas. *Design and Formal Verification of a Fault-tolerant Clock Synchonization Subsystem for the Controller Area Network*. Ph.D. Thesis. J. Proenza (supervisor), Universitat de les Illes Balears, Palma de Mallorca, Spain, 2010.

[RODR11] G. Rodríguez-Navas, C. Winter and J. Proenza. Injection of Aggregated Error Flags as a Means to Guarantee Consistent Error Detection in CAN. In *Proceedings of the 16th International IEEE Conference on Emerging Technologies and Factory Automation (ETFA), Toulouise, France*. pp 1-4. September 2011.

[RUC94] M. Rucks. Optical layer for CAN. *1st International CAN Conference*, November 1994.

[RUFI98] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida and L. Rodrígues. Fault-tolerant broadcasts in CAN. *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing, 1998. Munich, Germany*, 1998.

[RUFI99] J. Rufino, P. Veríssimo, and G. Arroz. A Columbus' Egg Idea for CAN Media Redundancy. *FTCS-29. The 29th International Symposium on Fault-Tolerant Computing, Winconsin, USA*, June 1999.

[RUSH03] J. Rushby. A Comparison of Bus Architectures for Safety-Critical Embedded Systems, *SRI International, Menlo Park, California*. Contractor Report, 2003.

[Ruz03] Jose Jaime Ruz Ortiz. *VHDL. de la tecnología a la arquitectura de computadores*. Editorial Sintesis. Madrid, 2003.

[SAH06] H. Saha. Active High-Speed CAN HUB. *Proceedings of the 11th international CAN Conference (iCC 2006), Stockholm, Sweden*, 2006.

[SEE08] Seeking Alpha. Altera and Xilinx Report: The Battle Continues. July 17, 2008.

[Sho02] Martin L. Shooman. *Reliability of Computer Systems and Networks*. John Wiley and Sons, Inc., 605 Third Avenue, New York, USA, 2002.

[SRV] SRV, *CANbids Project Website*. (http://srv.uib.es/project/12)

[STO03] G. Stoeger, A. Mueller, S. Kindleysides and L. Gagea. Improving Availability of Time-Triggered Networks: The TTA StarCoupler, *SAE 2003 World Congress, Detroit*, USA, 2003.

[WIS09] R. Wisniewski. Synthesis of compositional microprogram control units for programmable devices. *Zielona Gora, University of Zielona Gora. pp. 153* ISBN 978-83-7481-293-1.

[XE04] X Engineering Software Systems Corporation. *XSA-3S1000 Board V1.0 User Manual*, 2004.

[YO09] Yokogawa Electric Corporation. *DL7440/DL7480 Digital Oscilloscope User's Manual 5th Edition*, June 2009.


UIB