ESCOLA POLITÈCNICA SUPERIOR

UNIVERSITAT DE LES ILLES BALEARS

PROJECTE DE FINAL DE CARRERA

Estudi :

Enginyeria Informàtica

Títol:

Construction of sfiCAN: a star-based fault-injection infrastructure for the Controller Area Network

Alumne: Alberto Ballesteros

Directors : Julián Proenza Arenas
Manuel Alejandro Barranco González

Data: Juliol de 2012

Fem constar que el projecte de final de carrera titulat *Construction of sfiCAN: a star-based fault-injection infrastructure for the Controller Area Network* ha estat realitzat, sota la direcció de Julián Proenza Arenas i Manuel Alejandro Barranco González, per Alberto Ballesteros. Així mateix declaram que el projecte està finalitzat i preparat per la seva presentació pública.

Palma, Juliol de 2012

Signat: Alberto Ballesteros
Projectista

Signat: Julián Proenza Arenas
Co-director del projecte final de carrera

Signat: Manuel Alejandro Barranco González
Co-director del projecte final de carrera

Contents

1. Introduction

1.1. Background and motivation

The Controller Area Network (CAN) protocol [Robert Bosch GmbH, 1991] is a field-bus designed in the 1980s, which is widely used in distributed embedded systems. CAN provides electrical robustness and good real-time performance with very low cost. Due to this, the CAN protocol is nowadays used in a wide range of applications, such as in-vehicle communication and factory automation.

In fact, the usage of CAN has steadily been increasing and, as was predicted by some authors [Navet et al., 2005], it currently coexists with newer protocols. This trend is expected to continue because CAN is still penetrating old and new markets and because of potentially upcoming new high-volume applications [Zeltwanger, 2011]. Particularly in the automotive industry this trend is expected to continue, especially under the current economic situation, since there is a reluctance to invest in newer—but more expensive—technologies. Moreover, regulations in the United States and the European Union ensure the future use of CAN in automobiles by making it a recommended or even mandatory protocol for some of their parts [United States Environmental Protection Agency, 2005, The Commission of the European Communities, 2002].

The widespread use of CAN is currently leading to the revision of the protocol itself, as evidenced by the ISO 11898-6, and the specification of new standards for CAN-based dependable and real-time applications and protocols, like CANopen Safety EN 50325-5, ARINC 825, and the upcoming CAN with flexible data-rate (CAN-FD).

However, despite this interest on CAN, there are some open issues mainly related to its dependability and real-time features. As a consequence, there is also a significant amount of research being done to address some of these shortcomings, for instance, [Fredriksson, 2002, Rufino et al., 2006, Ferreira et al., 2006, Buja et al., 2007, Short and Pont, 2007, Hall et al., 2008, Rodríguez-Navas, 2008, Gil-Castineira et al., 2008, Zeng et al., 2009, Herpel et al., 2009, Nahas et al., 2009, Martí et al., 2010, Zeng et al., 2010, Hoppe et al., 2011, Monot et al., 2011, Barranco et al., 2009, Prodanov et al., 2009, Barranco et al., 2011, Lanigan et al., 2010, Almeida et al., 2002, Cavalieri, 2005, Nolte et al., 2005, Scharbarg et al., 2005, Obermaisser, 2006, Suwatthikul et al., 2011, Waszniowski et al., 2009, Mariño et al., 2009].

One of the open issues in CAN is the lack of an adequate fault-injection system to test the response of CAN-based applications and protocols when faults and errors do occur. As explained in Sec. 4.2.1, a *fault-injection system* [Tsai and Iyer, 1997] is composed of a set of parts which work cooperatively to carry out a so-called *fault-injection experiment*, that is, an experiment during which the behaviour of a given target system (a distributed control system in this case) is analysed when it is forced to deal with errors provoked by faults. More specifically, a fault-injection system provides three different features. First, it supplies tools for specifying the workload that the target system must execute during the experiment. Second, it allows the user to specify what faults to inject and, then, it provokes these faults in the target system. Finally, it collects data about the response of the target system to the errors provoked by the injected faults.

An adequate fault-injection system capable of testing CAN production software is important to both the industry and academia. In this sense, a *physical fault-injector*, that is, an injector that provokes faults at the physical level of the system, is the best choice for several reasons. In the context of the industry, testing of real systems or prototypes by means of a physical fault injector can provide more accurate and realistic results than other techniques, such as simulation-based ones [Tsai and Iyer, 1997]. In fact, the interest on prototype-based fault injection is specially relevant in the context of field buses, as can be seen in the significant amount of work carried out in this direction for protocols like Time-Triggered Protocol (TTP), FlexRay and Ethernet, for instance, [Ademaj et al., 2003, Armengaud et al., 2008, Fugger et al., 2009, Ferrari et al., 2008].

Moreover, in certain industry domains maintenance costs can match or even surpass development costs. In these domains, for economic reasons alone it is already necessary to comprehensively test the software of a system prior to production. For instance, in the automotive industry, where CAN is used extensively, recall costs of vehicles can be extremely high [Lanigan et al., 2010] and a significant amount of these recalls are due to software errors as can be seen in recent safety recall notices by the United States National Highway Traffic Safety Administration (NHTSA).

CAN is also used in safety-related systems (usually as the underlying technology of such safety-related protocols as SafetyBUS p, CANopen Safety, and DeviceNet Safety) and many standards for such systems highly recommend fault injection as a validation technique. Examples of such standards include the upcoming ISO 26262 [Lanigan et al., 2010] standard for safety-related systems in automobiles, and the generic standard for safety-related electronic systems IEC 61508 [International Electrotechnical Commission, 1999], which makes fault injection even mandatory in some cases.

Finally, fault injection is widely accepted as a fundamental verification technique for any system that is designed to have any specific reaction to faults. This does not only include the aforementioned safety-related systems, but also other dependable systems, for instance, fault-tolerant systems designed to achieve a high reliability and systems with automatic repair for high availability.

Regarding the academia, during the last 10 years several real-time and dependability limitations of CAN have been identified and different solutions have been proposed to overcome them [Fredriksson, 2002, Rufino et al., 2006, Ferreira et al., 2006, Buja et al., 2007, Short and Pont, 2007, Hall et al., 2008, Rodríguez-Navas, 2008, Gil-Castineira et al., 2008, Zeng et al., 2009, Herpel et al., 2009, Nahas et al., 2009, Martí et al., 2010, Zeng et al., 2010, Hoppe et al., 2011, Monot et al., 2011, Pimentel et al., 2008, Navet et al., 2005]. In this context an adequate physical fault injector is fundamental to thoroughly evaluate those solutions.

The need for an adequate fault injector has leaded to the proposal or use of different tools for testing CAN-based systems. However, all these aids present important limitations, as described in Sec. 4.3. The most important one is that none of them can generate local errors independently in the signals received and transmitted by each one of the nodes and, thus, they cannot force specific an complex scenarios; for example, the scenarios leading to inconsistencies between nodes [Rufino et al., 1998, Proenza and Miro-Julia, 2000].

To overcome these limitations, we have designed and implemented *sfiCAN*, a star-based physical fault-injection infrastructure for CAN. As depicted in Fig. 1.1, in sfiCAN the bus topology CAN relies on is substituted by a star topology whose central element is a hub. More specifically, sfiCAN is based on the architecture of CANcentrate, a CAN-compliant simplex star topology previously proposed in [Barranco, 2010], which, as will be explained in Sec. 5, is logically equivalent to a CAN bus and transparent from the nodes point of view. Moreover, note that each node is connected to the hub by means of a dedicated link containing an *uplink* and a *downlink*. The main advantage of this architecture is that the hub becomes a central element that has a privileged view of the communication, knowing the stream transmitted and received by each node, bit by bit. This allows sfiCAN to use the hub to inject errors independently in every single bit in the transmitted and/or received signal of each node and, thus, to force complex error scenarios. Moreover, the use of the hub as a centralized element yields other important testability advantages, which will be explained later, in Sec. 5.2.

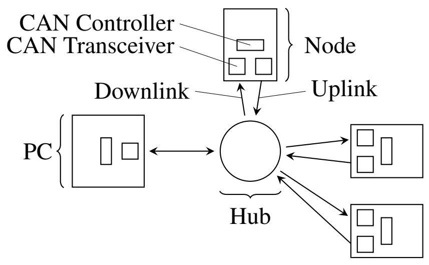


Figure 1.1.: sfiCAN architecture (Reprinted from a technical report we published Gessner, Barranco, Ballesteros, and Proenza [2011]).

The operation of sfiCAN is carried out by various logical units which we call *Network Configurable Components* (NCCs). Each one of these components is placed inside an existing physical component and carries out a specific task for a given fault-injection experiment. For this purpose the NCC extends the functionalities of the component it is placed into. There are three different types of NCCs, namely Centralized Fault Injector (CFI), Hub Logger (HL) and Node Logger (NL). There is only one Centralized Fault Injector, which is placed inside the hub and which implements the fault injector itself. There is also only one Hub Logger, which is placed in the hub, that monitors and registers information about the traffic from the hub's point of view. Finally, there is a dedicated Node Logger placed inside each node, that gathers information related to the internal activity of the nodes. Additionally, the user remotely manages all the NCCs from a PC connected to a specific port of the hub by means of the CAN network itself. For this purpose, the PC executes a tool called Fault-Injection Management Station (FIMS), which allows to configure the fault injector, collect the log report and coordinate all the NCCs. The FIMS communicate with the NCCs thanks to the *NCC protocol*, that is, a communication protocol on top of CAN specifically developed.

1.2. The goal of the project

After this overview on the background and the motivation of sfiCAN, it is easier to understand the main goal of this project.

The goal is to build a new fault-injection infrastructure capable of reproducing complex fault scenarios and, thus, to test the response of CAN-based applications and protocols when these faults do occur. For this purpose, we add different mechanisms and features to an existing prototype of a CANcentrate network. Specifically, a physical fault injector is placed centrally within the hub, whereas several log modules are distributed over the infrastructure inside the hub and the CAN nodes. Additionally, we develop a specific software, which executes on a PC connected to the hub, that allows to manage all these components remotely.

1.3. Realized tasks

This section highlights the specific work carried out for developing sfiCAN and, jointly with the next section, it delimits which tasks should be attributed to this project and which should not. In this sense, note that the implementation of sfiCAN is based on a previous CANcentrate's prototype and, thus, the nodes, the links, and the coupling and synchronization mechanisms of the hub were already available. Moreover, the major part of the design of sfiCAN was already done in the context of the research activities of the supervisors of this project and, thus, they cannot be considered as part of the realized tasks.

Next, the tasks actually carried out in this project are outlined.

Study relevant documentation

Before and during the development of sfiCAN I had to study a significant amount of literature. First, I had to familiarize with the CAN standard, particularly with the frame format and the behaviour of the nodes when errors do occur. Moreover, various advanced and non-standard concepts in CAN, such as its dependability limitations, had to be acquired. Second, I had to study both the hardware and the internals of CANcentrate and ReCANcentrate, so I could make profit of their architecture, as well as part of their internal logic. Third, in order to be able to log low-level events at the nodes, I had to inquire on the microcontroller registers and they behaviour. Fourth, as concerns the PC, I had to familiarize with its CAN interface so I could access to its operation at a low level; for this, I had to study its interface with the operating system. Finally, to assess the whole system, I had to get in touch with a CAN-compliant oscilloscope, which enforces me to inquire in its documentation.

Communicate a PC with the CANcentrate hub

In order to allow the user to manage sfiCAN remotely, it was necessary to connect a PC to the CANcentrate's hub. For this purpose, first I had to choose the PC CAN controller that is more suitable for our purposes. Specifically, we purchased a PCAN-PCI dual channel card developed by PEAK System-Technik GmbH [Peak System, 2012]. This CAN controller card is compatible with the software environment we use, that is, GNU/Linux. Moreover, it is compatible with

SocketCAN the default network stack for CAN devices, which, as explained in Sec. 18.3, allows to treat a CAN device as a regular network device. Additionally, the PCAN-PCI device provides several features that allow us to manage its operation at a low level, which makes it suitable for future projects.

Second, since the CANcentrate hub uses a non-standard CAN connection schema, I had to enable an additional hub port supporting the regular CAN connection schema for the PC controller. That is, a connection in which the contribution transmitted and received by the PC is not separated into an uplink and a downlink. On the one hand, in order to connect the PC with the hub, I constructed a custom cable. On the other hand, I had modified part of the implementation of the hub. Specifically, I had to adapt an existing port and perform specific modifications in its internal coupling mechanisms. Further information of this process is shown in Sec. 11.4 and 18.2.

Modify the bit-rate of the whole system

The original CANcentrate prototype was implemented to be compliant with a bit-rate of 921 kbps. In contrast, the CAN device used in the PC is limited to run at a discrete set of bit-rates which does not includes the 921 kbps. Thus, in order to communicate the PC with the prototype it was necessary to modify both, the hub and the nodes, to achieve a standard CAN speed. Specifically, we decided to increase it to the maximum speed set in the CAN standard, that is, 1 Mbps. This task involved the installation of new oscillators, as well as the modification of some the code related to the synchronization, in the hub and the nodes.

Design the fault-injection specification language

I defined an intuitive language, called *fault-injection specification language*, for specifying faults. This language is based on the sfiCAN fault model, that is, the set of faults sfiCAN is able to provoke, which was provided in the project proposal. Specifically, a fault is defined following a trigger schema, that is, by means of a set of conditions and the value to be injected where all these conditions are met.

Help to improve the design of the NCC protocol

We developed the *NCC protocol*, that is, the specific network protocol used to communicate the FIMS with the NCCs. Although a first sketch of this protocol was already done in the project proposal, I defined all the missing design details not present in it. Specifically, I defined the set of messages conforming the protocol, their format, and their encoding. Finally, note that the implementation of the NCC protocol was performed within the involved sfiCAN components, that is, the NCCs and the FIMS.

Remove the fault-treatment mechanisms from the hub

Since in sfiCAN the hub must be transparent from the nodes point of view, I removed all the fault-treatment mechanisms implemented in CANcentrate. However, note that some internal logic related to coupling and synchronization was reused. In this sense, after a deep study of the internals of CANcentrate, I extracted and accommodated some of the CANcentrate's modules, so the central fault injector and the central logger can make profit of them.

Design and implement the fault injector as an NCC

The development of the fault injector consisted in taking the definition of its operation, sketched in the project proposal, design its internals and implement it within the hub, by means of the VHDL language. Note that, as said previously, this is not a stand-alone module, instead, it reuses some modules from the CANcentrate's implementation to carry out its operation.

Design and implement the hub logger as an NCC

In order to obtain central information about the traffic transmitted from the point of view of the hub I developed a hub logger. First, I had to design its internals from the behaviour's description done in the project proposal. Later, I implemented this design inside the hub, by means of the VHDL language. Note that, similarly to the fault injector, the central logger needs additional information, provided by the reused modules from the CANcentrate's implementation, to carry out its operation.

Design and implement the node loggers as an NCC

In order to obtain local information, from the nodes point of view, of the events occurring at a given experiment, several node loggers were been constructed. The development of a given node logger consisted in designing its internals from the definition of its operation, provided in the project proposal. Then, this design was implemented inside the nodes as an additional module that interacts with the application.

Design and implement the FIMS

In order to manage the different components conforming sfiCAN, we developed the FIMS. Specifically, I designed and implemented the set of utilities conforming the FIMS from the initial specification performed at the project proposal. For this, I used a set of different user libraries and tools provided by the manufacturer of the PC CAN controller.

Verify the sfiCAN operation

In order to ensure the correctness of the sfiCAN operation I tested all its features at two levels. On the one hand, each of these features was tested separately by means of specific experiments. This process was possible since the construction of sfiCAN has been carried out progressively, that is, by adding the features one by one. On the other hand, I verify the global operation by carrying out various complex experiments involving various of these features, at the same time. Some examples of these experiments can be seen in Sec. 20.

1.4. Not realized tasks

In this section we list the set of tasks, related to the sfiCAN project, that have not been specifically carried out for it. Basically, it includes the physical construction of the elements of the CANcentrate's prototype used in sfiCAN, as well as the initial definitions done in the project proposal.

- Design and construct the CANcentrate hub
- Design and implementation of the CANcentrate coupler module
- Design and implementation of the CANcentrate synchronization modules
- Design and construct the CANcentrate nodes
- Design of the sfiCAN architecture
- Specify the sfiCAN fault model
- Specify the NCC concept

1.5. Document structure

The remain of the paper is divided in seven parts. Part I gives the necessary foundations to understand the bases of sfiCAN. Moreover, it also describes previous work on which this project is based. First, Chapters 2 and 3 overview the main characteristics of the CAN protocol. Second, Chapter 4 introduces the most important dependability evaluation techniques, emphasizing in the fault injection. Finally, Chapter 5 describes the main benefits of the star topologies and, in particular, the main design aspects of (Re)CANcentrate.

Part II explains the initial design sketched in the project proposal. First, Chapter 6 lists all the functional and non-functional requirements, that is, the minimum set of properties that sfiCAN must fulfil in order to reach our goals. Second, Chapter 7 describes the set of phases in which a given fault-injection experiment is divided. Finally, Chapters 8 and 9 describe the sfiCAN architecture at a two different levels. The first overviews the physical architecture, that is, the physical elements and their interconnections, whereas the second sketches the NCC architecture, that is, the specific components placed inside the physical elements that perform the sfiCAN operation and how they interact among them.

Part III describes the design of sfiCAN. First, Chapter 10 presents the fault-injection language used to specify the faults to be injected. Second, Chapters 11 and 12 overview the main hardware and software parts of the hub and the nodes, respectively. Finally, Chapter 13 describes in depth the design of the components of the NCC architecture, that is, the Fault-Injection Management Station, the Centralized Fault Injector, the Hub Logger and the Node Loggers. Moreover, it also explains the network protocol used to communicate all these components.

Part IV describes how we implemented the hardware and the software of sfiCAN. First, Chapter 14 introduces the implementation environments and platform used to define the operation of sfiCAN, focusing on VHDL, the language in which we coded the hub internals. Second, Chapters 15 and 16 describe the construction process of the hub; specifically how we adapted and extended the hardware and the internals of the ReCANcentrate hub. Third, Chapter 17 describes the implementation of the nodes, which includes both, the hardware modifications on the ReCANcentrate nodes, and the construction of the software. Finally, Chapter 18 describes the hardware and the software of the PC that executes the Fault-Injection Management Station. Moreover, we also explain the implementation of the set of tools conforming this last one.

Part V assesses, by means of a set of tests, the operation of sfiCAN. On the one hand, Chapter 19 describes the experimental platform used to carry out the tests. Moreover, it also explains the specific rules under which we execute these tests, as well as the steps followed to perform them. On the other hand, Chapter 20 describes the five different tests executed. Three of them recreate simple scenarios involving faults in a given link, whereas the other two describe complex fault scenarios involving various injections in various links.

Part VI gives the conclusions we have reached after finishing the project. First, Chapter 21 sums up the main characteristics of sfiCAN, highlighting its main contributions. Second, Chapter 22 presents my personal opinion on the development of sfiCAN. Finally, Chapter 23 proposes future extensions for the design and implementation of sfiCAN.

Part VII explains the practical applicability of sfiCAN. On the one hand, Chapter 24 specifies the publications that resulted from the work realized. On the other hand, Chapter 25 explains the interest raised in both the academia and the industry, focusing on CANbids, the project in which sfiCAN comes under.

At the end of this report, from page 185 onwards, we present a set of appendices addressing specific implementation and operation details. Here we present a quick-start guide jointly with a chapter explaining some additional operation details. Then, we describe the main steps that have to be followed in order to integrate the (Re)CANcentrate fault-treatment and fault-tolerance mechanisms in order to assess them. After that, we give a quick overview of the performance of sfiCAN, in terms of its reaction time, and the issues that this provoke. Finally, it contains the source code of the different sfiCAN components.

Finally, the bibliography referenced throughout this report can be found on the last few pages of this report.

Part I.

Foundations and previous work

2. Controller Area Network (CAN)

The Controller Area Network (CAN) protocol is a serial bus originally developed by *Robert Bosch GmbH* in the 1980s for automotive applications. Its main advantages are its low cost, simple configuration, electric robustness, prioritized medium access arbitration mechanism, as well as error-detection and containment features. Thus, although CAN was initially aimed to reduce the wiring cost in in-vehicle communications, soon after it became extremely popular in other distributed embedded control systems. Nowadays it is widely used as the communication infrastructure of a wide range of applications such as factory automation, robotics, intra-building communication, medical equipment, etc.

CAN defines only the last to layers of the ISO/OSI reference model, that is, the *Data Link Layer* and the *Physical Layer*. On the one hand, the Data Link Layer was firstly specified by Bosch in 1991 [Robert Bosch GmbH, 1991]. On the other hand, the Physical Layer was not specified until later, in 1993, when ISO standardized both in the ISO 11898 [ISO, 1993]. This standard was then updated in 2003 [ISO, 2003a,b].

This chapter summarizes the aspects of the CAN physical and data link layers that are more relevant to this document.

2.1. Physical layer

There are several specifications for the CAN physical layer. However, since the *Commercial Off-The-Self* (COTS) transceivers used in the sfiCAN prototype implement the ISO 11898-2, we only overview this standard.

In compliance with this standard, a CAN network relies on a simplex bus topology whose medium is constituted by two wires called *CAN_H* and *CAN_L*. The value on the bus is determined by the differential of voltage between these two wires. The main advantage of this approach is that it has a very good resistance to electromagnetic interferences. Additionally, in order to prevent signal reflections, the bus is terminated at both its ends with resistors of 120 Ohm and its stub lines are configured as short as possible.

One of the most important features of the medium of CAN is that it implements a *wired-AND* function of every node contribution. This is the basis of the dominant/recessive transmission property of CAN. This property guarantees that whenever one of the nodes transmits a bit with a dominant value, that is, a logical '0', this value is received by all the nodes in the network. In contrast, a bit with the recessive value, that is, a logical '1', is only received as long as every node issues a recessive value.

Moreover, CAN communication relies on a complex bit synchronization mechanism which guarantees that nodes have a quasi-simultaneous view of every single bit on the channel, that is, the so-called in-bit response. This mechanism uses the recessive to dominant transitions of the signal on the channel in order to keep the nodes of the network synchronized with respect to the node that is transmitting, the so-called leading transmitter.

Notice that the in-bit response property implies that the transmission of a bit traverses all the network and electrically stabilizes and only then the next bit can be transmitted. Thus, the bit synchronization of CAN forces an inverse relationship between the bit-rate and the achievable bus length. For example, if a CAN network operates at its higher bit-rate, that is, at 1 Mbps, the maximum achievable bus length is around 40 m [CiA]. In contrast, a CAN network operating at 125 Kbps or at 10 Kbps can respectively achieve a bus length of 500 m and 5 Km [CiA]. Although the bit synchronization of CAN limits the maximum bus length and/or the bit-rate of the network, at the same time it allows defining a number of additional mechanisms we will describe later on, for instance bit-wise arbitration, which yield important benefits in terms of dependability and real-time.

2.2. Data link layer

The data link layer of CAN provides a set of mechanisms that allow nodes to correctly exchange data even in the presence of errors and/or some permanent faults. Next, we present a brief overview on these mechanisms.

2.2.1. Frame format

To accomplish the transfer of messages, CAN uses four different types of frames called *data*, *remote*, *error* and *overload*. The format and the use of the first two is described next, whereas the *error* and *overload frames*, due to their use, are described later, in sections 2.2.5 and 2.2.7 respectively.

On the one hand, data frames are used to transport data from the transmitting node to the receiving ones. On the other hand, remote frames allow to make data requests. The format of the frame, in both cases, is similar, the only difference is that data frames can carry a data field with up to 8 bytes, as is shown in Fig. 2.1. Next, these types of frames are described in detail.

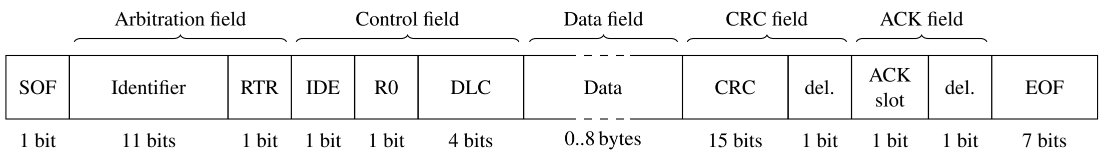


Figure 2.1.: CAN standard data frame format. (Reprinted from a technical report by Gessner [2010] with his permission).

Each data and remote frame is initiated with a dominant bit value, named *Start Of Frame* (SOF). This value contrasts with the recessive value found when the bus is idle. Moreover, note that, as explained in the previous section, the descending edge caused by the SOF forces receiving nodes to synchronize with the transmitting node. However, if more than one node transmits a SOF in the same bit time, they should compete for the right to transmit by means of the execution of the *arbitration algorithm*. This algorithm uses the arbitration field, explained next, to ensure that just one node, the one with most priority, is able to transmit.

The arbitration field is composed by a set of 11 bits named *identifier* and a bit named *Remote Transmission Request* (RTR). On the one hand, in contrast to most communication protocols, the *identifier* does not represent the destination of the message but the its content. On the other hand, the RTR bit identifies the type of the frame, that is, a data frame, when it contains a dominant value, and a remote frame, when it contains a recessive value. Note that it is possible to use a schema in which the identifier is composed by a set of 29 bits. In this case the data is said to have a *extended frame format*. More information about extended frames can be consulted in [Robert Bosch GmbH, 1991].

Next to the arbitration field is the *control* field, which is divided in three parts. First, the *IDentifier Extension* (IDE) bit is used to distinguish between extended frames, it contains a recessive value, and basic frames, it contains a dominant value. Next, the R0 bit is reserved for future protocol extensions, hence it always contains a dominant value. Finally, the *Data Length Code* (DLC) nibble has two different meanings depending on the type of the frame. In a data frame, it indicates the number of bytes of the data field, whereas in a remote frame it determines the number of bytes requested.

In data frames the control field is followed by the data field, which is just a succession of 0 to 8 bytes where the raw data is located.

Next to the data field, or the control field, depending on the type of frame, is the *CRC*. This field is used to carry a *Cyclic Redundancy Code* (CRC), which ensures the integrity of the frame in the receivers. Specifically, this field is composed by the CRC and a delimiter. The former has a 15-bit size and contains the code itself, whereas the second is a recessive bit which plays the role of a separator with the *ACK* field.

The ACK field, just as the CRC field, is divided in two parts, an ACK bit and a delimiter bit. The former is used by the receivers as a transmission window, for the purpose of informing the transmitter that a given frame has been received correctly, that is, the frame has passed the CRC test. The second, together with the CRC field delimiter, surrounds the ACK bit with recessive values. The acknowledgement algorithm is described next. During the ACK bit the transmitter sends a recessive value, while the receivers send a dominant value, whenever the frame has been correctly received. In the case the transmitter observes a dominant value in the bus, during the ACK slot, at least one receiver has accepted the frame. In contrast, if the transmitter observes a recessive value it is treated as an error, just as will be explained in Sec. 2.2.4

Finally, a succession of 7 recessive bits, called *End Of Frame* (EOF), marks the end of the frame.

Data frames, as well as remote frames, are separated from other frames by means of a succession with a non-predefined size of recessive values called *Interframe Frame Space* (IFS). This separation begins with a block of 3 recessive bits called *intermission*. Immediately the bus changes its state to idle, so it contains recessive values until any node starts the transmission of another frame. The definitive size of the IFS is conditioned by the idle time of the bus.

2.2.2. Arbitration mechanism

Any node can start the transmission of a frame whenever the bus is idle. In the case it is done by more than one node, at the same time, it is necessary to enable any mechanism that ensures that just one could send it. This mechanism is called arbitration mechanism and, as explained in the previous point, uses the content of the arbitration field to provoke the retreat of the nodes with lower priority messages.

This algorithm specifies that, the lower the value of the arbitration field, the more priority the frame has, and vice versa. During the arbitration all transmitting nodes observe bit by bit the bus, in the case of transmitting a recessive value and observing a dominant value, the node determines that his message has less priority, and thus, that it has lost the arbitration and must change its role to receiver.

Considering that the RTR bit belongs to the arbitration field, in general data frames have more priority than remote frames.

2.2.3. Frame encoding

CAN uses the *Non Return to Zero* (NRZ) encoding. This technique determines the value of the bit from the voltage in the bus, which keeps stable all the bit time. However, receiving nodes use the descending edges to synchronize with the transmitter, therefore, the transmission of various consecutive bits of the same value results in the desynchronization of them.

To avoid this behaviour CAN provides the *stuff* bit rule. Each time the transmitter sends 5 consecutive bits of the same value, adds an additional bit of the complementary value. Receiving nodes skip the processing of this bit, however, are obligated to resynchronize. This rule has some exceptions, specifically, it is not applied in the next bits: CRC delimiter, ACK delimiter, ACK bit, EOF nor IFS. Furthermore, it is not applied in error frames nor overload frames.

2.2.4. Error detection

CAN allows the detection of five different types of errors, according to [Robert Bosch GmbH, 1991]. Each type is described next.

- *Bit error*: This type of error occurs when the value in the bus does not correspond to the value sent by the transmitter. However, there are two cases, where this scenario is not detected as an error. The detection of a dominant value while the transmitter is sending a recessive value is considered, during the arbitration, as the loss of the right to transmit, whereas, during the acknowledgement, is considered as a correct transmission.

- *Stuff error*: This type of error is detected by the receiving nodes when observing a *stuff* bit violation. That is, when observing 6 consecutive bits of the same value. However, as explained in Sec. 2.2.3, this scenario is not considered an error if it occurs in certain field of the frame.

- *CRC error*: This type of error is detected by the receiving nodes when the CRC value transmitted does not correspond with the CRC value calculated. In other words, exists a divergence between the transmitted frame and the received frame.

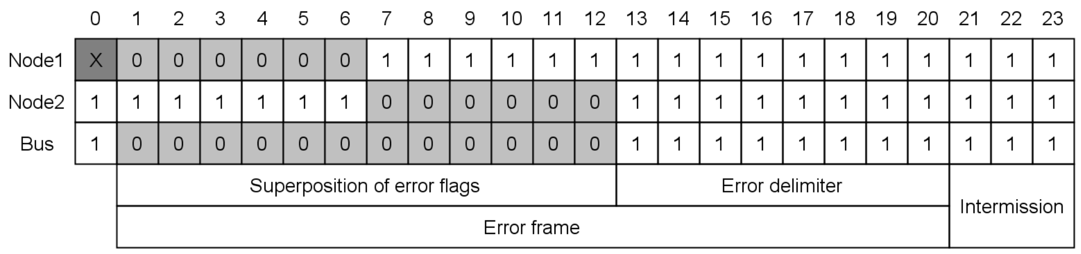
- *Form error*: This type of error is detected by the receiving nodes when detecting a frame format violation. This violation occurs when observing a non-valid value for a predefined value bit, that is, the recessive value in CRC and ACK delimiters, the dominant value of the R0 bit or any of the 7 recessive bits of the EOF. In form errors there is just one exception, a dominant value in the last bit of the EOF is not considered an error.

- *ACK error*: This type of error is detected by the transmitter when the value in the ACK slot is not dominant. That is, none receiving node has correctly receive the frame

2.2.5. Error signaling

Each time a node detects any of the errors described in the previous section, it rejects the current frame. In order to keep the consistency among nodes, local errors must be globalised, so the rest of the nodes reject the current frame too. The globalization is done by means of the transmission of a specific frame called *error frame*. This is possible since this frame violates the stuff rule and, thus, it provokes a global stuff error. The specific format of the *error frame* and the error signaling algorithm is presented next.

A regular error frame transmitted by a CAN node has two parts, a 6-bit sequence of dominant values called *error flag* and a 8-bit sequence of recessive values called *error delimiter*. However, the superposition of various error frames can generate a bigger error frame. How this frame is used to globalise an error is explained by means of the example shown in Fig. 2.2. Specifically, Node1 detects an error in the bit labelled ”0” which, as can be seen in the figure, is not detected by Node2. After that, Node1 starts the transmission of an error flag in the bit labelled ”1”. Since the size of an error flag is 6 bits, this dominant value lasts until bit labelled ”6”. At the same time, the stuff rule violation of this error flag causes a global error, which is detected by Node2. Thus, it starts the transmission of its own error flag, which mixes with the previous one provoking a superposition of error flags. Simultaneously, Node 1 indicates the end of its error flag sending recessive values. Note that, due to the superposition of errors flags this recessive values are not considered of part of the error delimiter. Node1 waits the end of the transmission of Node2’s error flag, that is, it waits until a recessive bit is seen in the channel. When so, both nodes start the transmission of the error delimiter, which consists in a set of 8 recessive bits. This ends the error frame and makes way to the intermission, after which another frame can be transmitted.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Node1	X	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Node2	1	1	1	1	1	1	1	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
Bus	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
		Superposition of error flags												Error delimiter								Intermission		
		Error frame																						

Figure 2.2.: Error signaling scenario. (The figure is based on a figure by Gessner [2010]).

2.2.6. Error containment

According to the error detection and error signaling mechanisms presented previously, a faulty node detecting errors continuously could block the communication in the channel, due to the repeated transmission of error flags. In order to avoid this behaviour, CAN implements an error-containment mechanism, referred as *Fault confinement* in [Robert Bosch GmbH, 1991]. This mechanism determines that the operation of a CAN node is driven by its internal state, which can be of three types, called *error active*, *error passive* and *bus off*. In turn, the state of a CAN controller is driven by two error counters, called *Transmission Error Counter* (TEC) and *Reception Error Counter* (REC). How the state changes, due to the value of these error counters, is shown in the automaton of Fig. 2.3 and discussed next.

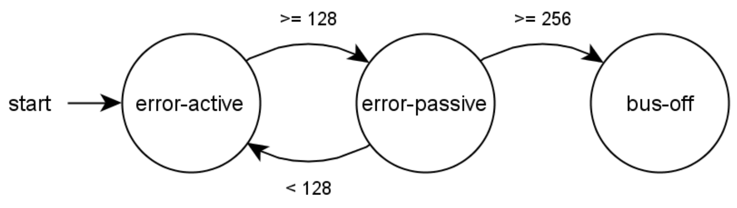


Figure 2.3.: Error containment automaton.

The default state when starting a CAN controller is error active, whereas the initial value of both error counters is 0. In error active state, the CAN controller behaves normally, that is, it can take part in the communication and, when detecting an error, it sends an active error flag, just as described in the previous section. If the value of any of the error counters equals or exceeds the threshold of 128, the CAN controller progresses to the error-passive state. Similarly to the error-active state, a CAN controller in error-passive state can take part in the communication. However, when detecting an error, instead of an active error flag, it sends a passive error flag which, in contrast to a regular active error flag, is composed of six recessive bits. Thus, the error frames transmitted by a node in error passive state cannot globalise the errors and, thus, block the channel. In case of reaching the threshold of 256, a transition to the bus-off state is performed. In this state the CAN controller cannot participate in the communication, that is, it cannot send error frames nor any other kind of CAN frames. Note, from the figure, that is possible to recover from error-passive state when the value of both counters is less than 128. However, this is not possible when reaching the bus-off state, in which the CAN controller stands until the user requests the recovery. Next, we discuss the behaviour of the error counters in front of errors.

The specific set of rules which describe how the error counters increment and decrement their values is listed in [Robert Bosch GmbH, 1991]. In broad terms, a CAN controller detecting and error when receiving a frame increases its REC by 1. On the other side, each time a transmitting CAN controller detects an error increments its TEC by 8. The types of errors been able to be detected by each of these CAN controllers have been described in Sec. 2.2.4. For instance, a transmitting node can only detect a bit error, that is, it receives a different value that the one transmitted, or an ACK error, that is, none node drives the value of the ACK slot to a dominant value. Additionally, a receiving or transmitting node increases the value of its REC or TEC, respectively, if a bit error is detected during the transmission of an active error flag. As concerns the events decreasing the value of the error counters, each time a frame is successfully transmitted the value of TEC is reduced by 1, unless the value is already 0, case in which the value remains unchanged, Moreover, each time a frame is received correctly, the value of REC is reduced by 1, unless the value is already 0 or between 128 and 255, case in which it is set to a value between 119 and 127.

Additionally to these rules CAN defines a specific mechanism called *Primary Error* (PE) that applies an extra penalization to the nodes provoking errors. This extra penalization forces any node suffering from local faults, that is, faults occurring in a subset of the nodes, to quickly switch from the error-active to the error-passive state. As explained, in the error-passive state the capacity of a node for globalizing errors is reduced and, thus, its negative impact on the communication is also minimised. As explained, in Sec. 2.2.4, when a node detects an error, it signals and globalises it by sending an error frame. A given error is normally detected by all nodes, so that all them send the error flag and the error delimiter at the same time. In this case the rules applied are the ones previously explained. However, when an error is locally detected only by a subset of the nodes, the error flags of the rest of the nodes are delayed with respect to their own. Note that, a given node can detect a local error after receiving a dominant bit at the first bit of its error delimiter. In such a case, it is said that the node detects a Primary Error. In this situation, the node CAN penalizes the specific nodes detecting a PE by further increasing their TEC or REC, depending on the role of the node, by 8.

The rest of the error-containment rules describe particular situations or highlights specific known scenarios in which they are not applied. Note that most of them have already been discussed in Sec. 2.2.4.

2.2.7. Overload signaling

CAN provides a specific signaling mechanism to inform that a given node needs an extra delay to process the last transmitted frame. Specifically, when a given node wants to delay the transmission of a data or a remote frame, it is enough to transmit the so-called *overload frame*, beginning at the first bit of the Intermission Frame Space (IFS). This frame has the same format that an error frame, described in Sec. 2.2.5. That is, it is composed by six consecutive dominant bits, called *overload flag*, followed by eight consecutive recessive bits, called *overload delimiter*. Moreover, the response of the nodes when receiving it is also the same. That is, when a given node receives the first dominant bit of an overload flag, it begins the transmission of their own overload frame. The only difference with the error frame is that it is not considered as an error signaling and, thus, no modification in the error counters is performed.

3. CAN Fault Model

Next, the different types of faults that can occur in a CAN-based system are explained. For that, we differentiate between those faults that manifest at the level of the communication subsystem, that is, faults that generate errors at the signal transmitted/received by the node, and those faults that do manifest at the application level.

As concerns faults at the level of the communication subsystem, note that the hardware components of a CAN network may suffer from different faults: damaged transceivers, bad connectors, long cables, electromagnetic interferences, etc. Since these faults manifest by generating errors that corrupt the logical values of the bits observed in the bus, we refer them to as *syntactic faults*. As it will be explained in the following sections, these faults can be classified as *stuck-at* and *bit-flipping* faults, depending on the logical value of the errors they generate.

On the other hand, we refer faults that manifest at the application level to as *semantic faults*. These can exhibit a broad range of different failure modes, depending on the application itself. For example, in a real-time application, a fault may lead to the transmission of timely-incorrect frames. In order to exhaustively categorize semantic faults, we consider as a reference the classification established in [Avizienis et al., 2001], that is, byzantine or arbitrary, authentication detectable byzantine, incorrect computation, performance, omission, crash and stopping (fail-stop) failures.

The CAN error-detection and error-containment mechanisms can deal with syntactic faults. However, as it will be explained in Sec. 5, its bus topology imposes strong limitations to the effectiveness of theses mechanisms. On the other hand, from the application point of view, CAN is supposed to provide a property known as *data consistency*, which is very interesting for fault-tolerant and real-time distributed systems. A protocol is said to provide data consistency if it guarantees that every frame is simultaneously accepted by all nodes or by none of them [Proenza and Miro-Julia, 2000]. Nevertheless, despite the CAN standard is supposed to ensure this property, some authors have detected some scenarios, that is, *inconsistency scenarios*, in which this property does not hold

In principle, since sfiCAN is primarily intended to be a physical fault injector independent from the application, its mechanisms are designed to inject syntactic faults. However, although the definition of semantic faults is normally application-dependant, the sfiCAN mechanisms also allow to inject them to some extent, for example, inconsistency scenarios. Some examples that show how sfiCAN can be used to inject inconsistency scenarios are provided in Sec. 20.

Next, the different types of syntactic faults are described, and some inconsistency scenarios are pointed out.

3.1. Stuck-at faults

These types of faults generate a constant bit value over a given period of time, or even indefinitely. This scenario may be caused, for example, by short-circuits to ground or battery, or malfunctioning or isolated controllers. There are two types of stuck-at faults, namely *stuck-at-dominant* and *stuck-at-recessive*, depending on whether the stuck-at bit consists in a dominant or a recessive value respectively. Because of the wired-AND property of CAN, a stuck-at-recessive fault can prevent all nodes from communicating only if it happens in the bus line itself.

3.2. Bit-flipping faults

These types of faults generate a random and erroneous sequence of bits, that is, a bit-flipping stream, over a finite or infinite period of time. A bit-flipping stream prevents any attempt of transmission since their dominant bits overwrites the recessive bits of the original frame. A bit-flipping may be caused, for example, by a damaged node or a bad welding on a connector.

3.3. Inconsistency scenarios

An inconsistency scenario consists in a combination of erroneous bits locally affecting the signal transmitted and/or received by different nodes, that lead a frame to be accepted by a subset of nodes only. There are mainly two types of inconsistencies: *omissions* and *duplicates*. A message omission inconsistency occurs when a set of nodes accepts a frame, whereas another set rejects the same frame. On the other hand, a duplicated message inconsistency occurs when one or more nodes accept a frame more than one time.

There are various scenarios in which these types of inconsistencies can occur. A well known set of scenarios that provoke an omission inconsistency happen when the CAN controller of a node reaches the error-passive state. As explained in Sec. 2.2.6, a CAN controller changes its behaviour depending on the amount of errors it has detected in the channel. Specifically, when it encounters a given number of errors, it switches from the error-active to the error-passive state. While being in this state, it signals errors by transmitting an error-passive flag, which cannot always force an error to be globalised. Thus, an omission inconsistency occurs every time an error-passive node cannot force the other ones to reject a frame in which it has detected a local error.

But inconsistencies may also occur even when all nodes are error-active. The specific scenarios leading to these inconsistencies are thoroughly discussed in [Rufino et al., 1998] and [Proenza and Miro-Julia, 2000]. Next, as an example, we describe one of these scenarios, which is called the *last bit inconsistence scenario* and which was first described by Proenza in [Proenza and Miro-Julia, 2000]. This scenario distinguishes three groups of nodes: the transmitter and two sets of receivers, called X and Y. The set X contains all the receiving nodes that cannot accept a frame due to the occurrence of a local error. Conversely, Y contains all the receiving nodes that accept the frame.

Fig. 3.1 shows the details of this scenario. Note that all the values depicted in the figure correspond to the signals the nodes receive. First, a bit-flip affects the last-but-one bit of the

EOF being observed by *X*, that is, they monitor a dominant value in the 6th bit of the EOF. As a consequence, the nodes of *X* reject the frame and simultaneously start transmitting an error flag in the last bit of the EOF. At this point notice that although the nodes of *Y* receive the dominant bit corresponding to the first bit of the flag during the EOF, they do not consider that bit as an error because it happens at the last bit of that field, see Sec. 2.2.4. Thus, the nodes of *Y* accept the frame. As concerns the transmitter, the scenario assumes that an extra bit-flip affects the last bit of the EOF it observes, so that the transmitted does not detect the first dominant bit of the error flag and, consequently, accepts the frame and does not retransmit it. After that, the rest of the error flag is considered by both the transmitter and *Y* as an overload frame, that is, a sequence of dominant values after the EOF, see Sec. 2.2.7.

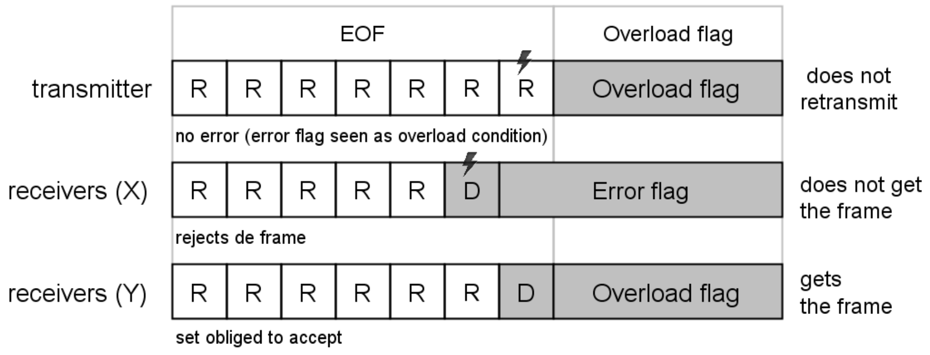


Figure 3.1.: Proenza's IMO scenario. (The figure is based on a figure by Proenza and Miro-Julia [2000]).

4. Introduction to dependability evaluation techniques in CAN

The development of complex fault-tolerant systems requires the use of a systematic and practical strategy. For this purpose, it is usual to follow the paradigm proposed in [Avizienis, 1995], which divides its construction into three activities: *specification*, *design* and *evaluation*. The specification consists in determining the requirements of the system, covering both the functional and the dependability ones. The design involves the definition and specification of the system architecture. This does not only includes the definition of each subsystem, but also the characterization of the interactions between their functionalities and fault-tolerance mechanisms. Finally, the evaluation is performed within the design process for two purposes: guide this process and ensure that the functional and dependability requirements are fulfilled. In addition, the system is also evaluated once it has been implemented, in order to check that it fulfills all the requirements. Moreover, a system can be also evaluated during its operation for maintenance purposes, for instance, to retrieve statistics about its performance or to detect anomalies.

The work presented here is related to the last kind of activities, that is, to those that are included in the evaluation phase. Specifically, it is related to the evaluation of the degree of dependability achieved by a system. In this sense, note that there are several techniques available for this purpose. In particular, this work is devoted to providing a tool, sfiCAN, for evaluating the fault-tolerance capabilities of CAN-based systems by means of an evaluation technique called *fault-injection*.

Next we outline the characteristics of the main dependability evaluation techniques in general, focusing on the different fault-injection tools that have been proposed for CAN.

4.1. Dependability evaluation techniques

Dependability evaluation techniques are classified into *qualitative* and *quantitative* evaluation ones. The former are intended to verify that the system is able to deal with all the faults included in its fault model, whereas the second ones aim at numerically corroborating that the system fulfills the required degree of dependability.

One of the most used qualitative evaluation techniques is known as *Model checking*. It allows the formal verification of system properties. To apply it, it is necessary to build a model of the system, usually in terms of interconnected automatons. Then, the user must define the properties that must be satisfied by means of queries. Finally, the compliance of these properties is checked by analyzing all the possible states of the model, using a software tool called *model checker*.

Regarding quantitative dependability evaluation techniques, they normally rely on the construction of a model of the system. Different formalisms can be used for this purpose, see [Muppala et al., 2000]. Some of the most used formalisms are *Markov Chains* and *Petri Nets*, which allow modelling the system dependability-related features and behavior in terms of stochastic processes.

The representation of a Markov chain is performed using a directed graph whose nodes are states and the transitions among states are given by a predefined probability. A given Markov model can be solved using mathematical tools, which finally provide numerical measures that characterize specific system properties. Petri Nets are a generalization of the automaton theory, focused on describing concurrent and interdependent events. A Petri Net model can be also solved numerically. In fact, in many cases, a Petri Net is transformed into an equivalent set of Markov Chains, which are then solved to calculate a given measure of dependability.

As concerns, *fault injection*, it consists in creating artificial faults directly in the system. The main advantage of this technique is the possibility of recreating specific fault scenarios that, otherwise, would need a significant amount of time to occur. Thus, fault injection is well suited for analyzing the fault-handling capabilities of the system with respect to a particular fault model. In this sense, fault injection can be used to evaluate a system either qualitatively or quantitatively. In particular, since fault injection can be used to execute a huge amount of tests, it allows to estimate the values that characterize specific fault-tolerance features of the system. This is an interesting advantage, as these estimations can be used to refine a given stochastic model of the system, by feeding the value of the parameters that characterize those features.

4.2. Fault injection

As introduced above, fault injection is well suited to quantitatively evaluate the fault-tolerance capacities of a system, for instance, its fault detection, fault isolation, system reconfiguration and recovery mechanisms. In fact, this technique is gaining in importance nowadays, as it is clearly reflected in the upcoming ISO 26262 standard for functional safety in automotive electronics, which highly recommends the inclusion of fault-injection techniques as a part of the dependability analysis of critical systems.

Next, the general architecture of a fault-injector environment is briefly sketched [Tsai and Iyer, 1997]. Then, we describe the main types of injectors, focusing on those that have been proposed for CAN so far.

4.2.1. Fault-injection environment general architecture

The general architecture of a fault-injection environment is depicted in Fig. 4.1. It is divided into two parts, the *fault-injection system* and the *target system*.

As can be seen in the figure, the fault-injection system is divided into several modules, namely the *fault injector*, the *workload generator*, the *monitor*, the *data collector*, the *data analyzer* and the *controller*. All these modules work concurrently and cooperatively not only to inject faults, but also to collect data concerning the response of the system to these faults. First, the workload generator feeds the target system with a set of commands that compose the workload itself. For this purpose, the workload generator uses the information provided by the *workload library*, which characterizes the workload to be executed during the test, for instance, a real application, benchmarks or synthetic workloads. Second, the *monitor* tracks the execution of the commands

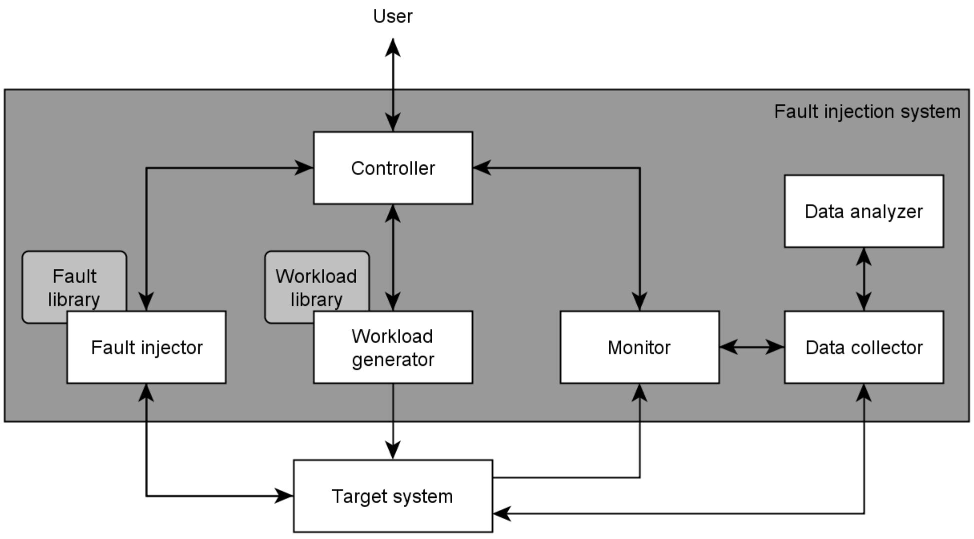


Figure 4.1.: Fault injection architecture. (The figure is based on a figure by Tsai and Iyer [1997]).

and instructs the data collector to gather the appropriate data. Note that the data collector collects data online, whereas the data analyzer processes and analyzes these data off-line. Third, the fault injector is responsible for provoking the errors in the target system. Finally, the controller is in charge of coordinating the different modules during the experiment. Specifically, it decides which workload from the workload library has to be executed, which fault from the *fault library* has to be injected, and which information must be monitored. In turn, the controller is managed by the end user.

4.2.2. Types of fault injection

Fault-injection techniques can be classified attending to both the maturity of the implementation of the system being tested, and the place where faults are injected into. As concerns the first classification criterion, we can distinguish between the two following types of fault-injection techniques:

- Prototype-based techniques. This type of fault injection operates over an existing prototype of the system. The main benefit of using a prototype-based fault injector is the realistic degree of the data obtained from the experiments. However, the cost of obtaining these data is normally high if the prototype is implemented in hardware.
- Simulation-based techniques. In this type of technique faults are injected into an abstraction of the real system, that is, into a model of the system. The main advantage of this approach is that, since the model and the injector are implemented in software, the data of an experiment can be easily collected directly from the values conforming the state of the model. However, the quality of the data obtained when executing a fault-injection test

depends on the quality of the model, which in turn depends on its operational parameters. This is an important disadvantage, as the values of these parameters are typically difficult to estimate.

Regarding the second classification aspect, we can find the next two kinds of techniques:

- Hardware fault-injection techniques. This type of fault injection uses additional hardware to introduce faults into the hardware of the target system or prototype. Hardware fault injection is well suited when the injection must be done at a low level and/or when there is a need of high time resolution. For instance, when it is necessary to inject errors in specific locations of a transmitted bit stream. However, the construction of an injector of this type is usually expensive and difficult.

- Software fault-injection techniques. This type of fault injection uses a piece of software to force high-level faults into de the system. In contrast to hardware fault injection, faults are injected at the application or at the operating system itself. Examples of software faults are data corruption or bugs in the software code. Software fault injection is very flexible and less expensive when compared with the hardware-based approach. However, the instrumentation software is an additional code that has to be executed inside the system and, thus, it can cause perturbations in the real workload.

4.3. Existing fault injectors for CAN

In this section we describe the most significant fault injectors proposed for CAN, following the classification previously presented. In this sense, note that, as far as we know, they only have been proposed for CAN prototype-based hardware and simulation-based software fault injectors. Thus, here we only address these kind of injectors.

However, it is important to note that it would be possible to inject faults in a CAN-based system by means of injectors not specifically designed for CAN. Specifically, generic prototype-based software fault injectors could be used in CAN, since they inject in a high level in which the low level details of CAN do not matter.

4.3.1. Prototype-based hardware fault injectors

Some prototype-based hardware fault injectors have been proposed for CAN. As it can be inferred from the above discussion, this kind of injectors are designed to provoke low-level faults at the hardware of the system. Two representative examples of them are the *IFI-based CANfidant-managed fault injector* and the *CANstress*.

The first injector [Rodríguez-Navas et al., 2003] was developed at the University of the Balearic Islands, and it is intended to inject faults with high spatial and time resolution in a low-invasive way. For this, this fault injector uses various Individual Fault Injectors (IFIs), each of which is located inside a specific node, just as can be seen in Fig. 4.2. A given IFI acts as a multiplexer between the CAN transceiver and the CAN controller, in order to inject local errors at specific bits. Additionally, each IFI monitors and collects the traffic received by its node locally, in order to analyze data concerning the traffic at the end of the experiment.

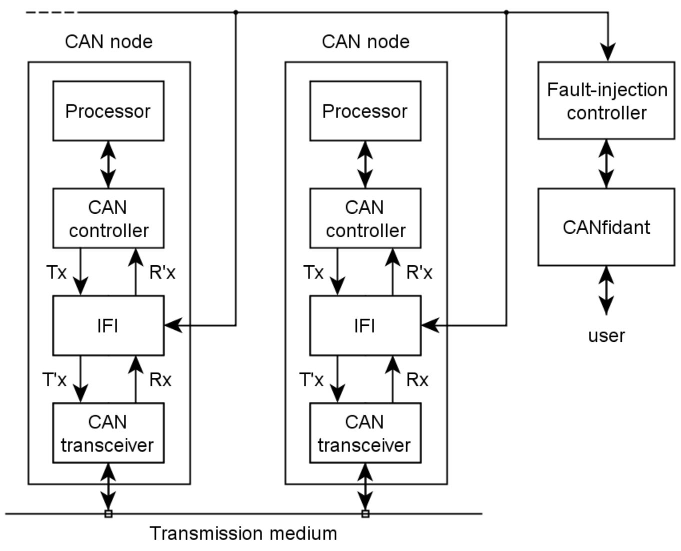


Figure 4.2.: Architecture of the IFI-based CANfidant-managed fault injector. (The figure is based on a figure by Rodríguez-Navas et al. [2003]).

In the IFI-based CANfidant-managed fault injector the fault scenario is set up externally at a PC running a specific software tool named *CAN Fault Injection Design Assistant* (CANfidant). CANfidant is a multipurpose tool developed at the University of the Balearic Islands that provides the following services: assistance in the design of fault scenarios, pre-processing of fault scenarios and analysis of the fault injection results. The assistance in the design of fault scenarios is provided by a CANfidant's CAN simulator. In a first step, the simulator is used to configure the set of messages that are going to be exchanged and, then, it calculates the resulting bit stream transmitted and received locally by each node. Then, the simulator allows the user to specify errors at specific bits of the signal transmitted and received by any node. Each time an erroneous bit is specified to be injected, the simulator automatically calculates its effects and updates the node's bit streams. The second service provided by CANfidant is intended to codify and transmit the fault scenario to the IFIs, in order to recreate it in the real system. Finally, once the physical fault injection finishes, CANfidant allows analyzing the data of the test offline. Specifically, CANfidant collects the data logged by the IFIs, which then can be compared with the results obtained previously through the simulation

Nonetheless, this fault injector presents some limitations that should be took into account. On the one hand, it does require a priori knowledge of the sequence of frames to be transmitted, as well as, what bits will be erroneous. That is, the resulting traffic should be pre-calculated with CANfidant when designing the experiment. On the other hand, no hardware prototype of the IFIs exists.

The second fault injector pointed out above, that is, CANstress [Vector, 2012b], was developed by a consolidated company called Vector, which develops hardware and software solutions and engineering services for the networking of electronic systems. CANstress is composed of a hardware module and the CANstress software. The hardware module is responsible for forcing dominant or recessive logical bit values affecting the whole CAN bus. In turn, the operation of the hardware module is set up by the user by means of the CANstress software. CANstress implements very sophisticate triggers, based on the traffic of the bus, which allow injecting stuck-at faults in specific locations within the frames.

Certainly, this fault injector allows defining complex conditions to trigger the injection of faults with a high time resolution. However, it presents a limited spatial resolution, as it can only inject *global faults*, that is, faults that affect the bit-stream broadcast though the bus, but it cannot inject *local faults*, which affect specific nodes locally.

4.3.2. Simulation-based software fault injectors

There are some simulation-based software fault injectors for CAN. One of the most recent ones is the *CANoe-based fault injection framework for AUTOSAR* [Lanigan et al., 2010]. This fault injector allows to provoke faults in well-defined AUTOSAR-based systems being simulated using CANoe. AUTOSAR [Autosar] is an open and standardized automotive software architecture jointly developed by automobile manufacturers, suppliers and tool developers. It supports the most-used network protocols in automotive, including CAN. On the other side, CANoe [Vector, 2012a] is a well-known tool from Vector that allows simulating the behaviour of Electronic Control Units (ECUs) and networks of ECUs interconnected by means of CAN or other protocols. Thus, CANoe is extensively used for the development, analysis, simulation, testing, diagnostics and start-up of automotive applications.

In order to inject errors within the AUTOSAR architecture, this fault injector proposes a *hook-based* paradigm. In this paradigm the user places and executes pieces of code, that is, the hooks, within the codebase of an AUTOSAR-based application. This fault injector defines two types of hooks. On one hand, *suppression hooks* cause errors directly by forcing the AUTOSAR Application Programming Interface (API) calls to return error codes. On the other hand, *manipulation hooks* cause errors indirectly by corrupting specific data structures allocated into the memory. The user can configure, activate and visualize the faults during the execution by using the control panel provided by CANoe.

Nonetheless, this fault injector presents some limitations that should be took into account. First, due to the limited access of the hooks, it is not possible to inject low-level faults. Second, The software fault-injection module that holds all the hooks cannot differentiate between calls made from multiple nodes. Therefore, the injector can only inject faults at one node. This fact makes it impossible to recreate scenarios involving cascading and correlated faults among multiple ECUs. Finally, this fault injector suffers from probe effects, that is, when injecting certain types of faults the entire simulation can be affected negatively. For instance, inserting artificial delays to cause timing violations can potentially block the entire simulation.

4.4. Conclusions

In this chapter we have introduced dependability-evaluation techniques in general and, then, we have briefly reported the fault-injection tools that have been provided for CAN so far.

We have shown that prototype-based hardware fault injectors for CAN make it possible to inject faults in a real system, but that most of them bring low spatial resolution, as they can only inject errors affecting the whole bus signal. The only prototype-based fault injector capable of injecting faults at the signal transmitted and received by each node locally is the IFI-based CANfidant-managed one. Nevertheless, this injector requires implementing a distributed architecture in which the fault-injection logic is placed inside each and every node.

On the other hand, simulation-based software fault injectors allow the injection of faults at any level, with high spatial and time resolution. This is possible since the fault injector interacts with a model of the system. However, the behaviour of the system depends on a set of estimated parameters, which are difficult to obtain. Thus, the results provided by this type of injectors are usually less realistic.

As it will be explained in the rest of this document, we have designed sfiCAN to inject faults with high spatial and time resolution, while overcoming the limitations of previous prototype-based hardware fault injectors. In this sense, note that sfiCAN is a physical fault injector; however it is noteworthy that, at the same time, the faults it injects at low level also allow to indirectly induce faults at the level of the software to some extent.

5. Star-based architecture for sfiCAN

In order to overcome the limitations of the fault injectors that are already available for CAN, we have designed sfiCAN as an injector based on a star topology. As will be explained in this section, some characteristics of this topology can be used to provide interesting fault-injection features, such as the possibility of injecting faults centrally with high spatial resolution.

More specifically, sfiCAN relies on the architecture proposed in the context of two CAN-compliant star topologies developed by the supervisors of this project [Barranco, 2010]. These are a simplex star topology called CANcentrate and a replicated one called ReCANcentrate, which are jointly referred to as (Re)CANcentrate. These topologies aim to improve the error-containment and the fault-tolerance limitations that CAN presents due to its bus topology

Moreover, these stars are transparent from the nodes point of view, so that *Commercial Off-The-Self* (COTS) components can be used for their construction, while supporting existing CAN applications and protocols. This feature is also important in the context of this project because, as it will be explained later, it makes possible to implement sfiCAN using COTS components.

Next, the basics of (Re)CANcentrate are explained. Later, we discuss how some of their features can be used in sfiCAN to provide advanced fault-injection capabilities.

5.1. (Re)CANcentrate basics

As said before, the aim of (Re)CANcentrate is to improve the error-containment and the fault-tolerance limitations that CAN presents due to its bus topology.

On the one hand, as concerns error containment, note that a bus has several components with direct electrical connections to each other. Thus, despite the error-containment mechanisms of CAN, see Sec.2.2.6, a single fault may generate errors that propagate all along the network preventing several, or even all, nodes from communicating. For instance, a node's transceiver output that becomes bit-flipping will corrupt any data conveyed by the bus, even though the CAN controller corresponding to that transceiver diagnoses such a fault. Moreover, bus topologies suffer from *spatial-proximity* and *common-mode* failures. The former occurs when, due to physical proximity, a fault affects different components. Similarly, a common-mode failure occurs when different components fail in the same way, either due to physical proximity or because they share a faulty resource.

In order to improve error-containment, CANcentrate relies on an active hub with improved error-detection and fault-treatment capabilities. The general architecture of CANcentrate is sketched in Fig. 5.1. As can be seen there, each node is connected to the hub by means of a dedicated link containing an uplink and a downlink.

The main advantage of such a hub is that it is a central element that can have a privileged view of the communication, so that it knows the contribution transmitted by each node, bit by

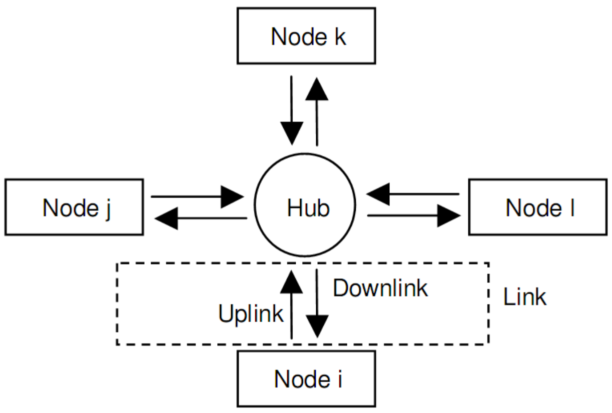


Figure 5.1.: CANcentrate's architecture. (Reprinted from a technical report by Barranco [2010] with his permission).

bit. On the one hand, this allows the hub to detect errors with higher precision than typical CAN controllers [Barranco, 2010]. On the other hand, the hub can easily isolate a faulty node just by disconnecting the corresponding hub port. Moreover, the problem of spatial-proximity and common-mode failures becomes less relevant in a star, since the hub and the nodes are separated from each other and they do not share resources.

On the other hand, CAN has no mechanisms for tolerating faults, and even replicated CAN buses still suffer from *spatial-proximity* and *common-mode* failures [Barranco, 2010]. This limitation can be overcome by using ReCANcentrate, which basically can be considered as a duplicated CANcentrate network, see Fig. 5.2, that provides fault tolerance in addition to error containment. For this purpose, ReCANcentrate includes two hubs, very similar to the one of CANcentrate, that exchange their traffics and that couple with each other to create a single broadcast communication domain. The only difference between a ReCANcentrate's and a CANcentrate's hub is that the former is able to isolate the contribution it receives from the other hub. Thanks to the hub coupling, both hubs behave like one broadcasting the same traffic, bit by bit, through their downlinks. This allows the nodes to easily manage the replicated media while tolerating faults, see [Gessner, 2010]. ReCANcentrate basically provides tolerance to faults affecting one of the hubs, no matter which one of them, and one of the connections of each node to any of the hubs.

As happens with sfiCAN, (Re)CANcentrate were developed to be independent from the application relying on them. Thus, the hubs of both CANcentrate and ReCANcentrate are designed to deal with syntactic faults. Additionally, ReCANcentrate is also able to tolerate fault-inconsistency scenarios to some extent, see Sec. 3.3. Anyway, the hubs can be extended to address semantic faults [Barranco, 2010].

The rest of this section is devoted to explaining further details about the architecture of (Re)CANcentrate. However, since ReCANcentrate is basically a duplicated CANcentrate network, we will focus on CANcentrate only. More details about the particularities of the architecture of ReCANcentrate can be found at [Barranco, 2010].

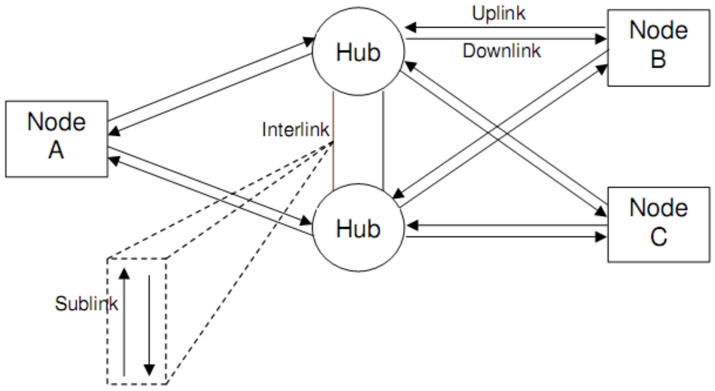


Figure 5.2.: ReCANcentrate's architecture (Reprinted from a technical report by Barranco [2010] with his permission).

The physical setup of a node is shown in Fig. 5.3. The node's microcontroller exchanges data directly with a CAN controller. In order to transmit/receive data to/from the CAN network, the CAN controller is connected to two CAN transceivers, labelled *Uplink TxRx* and *Downlink TxRx* respectively. Although these are COTS transceivers, the connection is not the usual one. The CAN controller's transmission pin, *Tx*, is connected to the transmission data input pin, *TxD*, of the uplink's transceiver. In turn, this transceiver is connected directly to the uplink. Note that since the node does not use the uplink for receiving, the reception pin of the uplink's transceiver, *RxD*, is left unconnected. Analogously, the CAN controller's reception pin, *Rx*, is connected to the reception data output pin, *RxD*, of the downlink's transceiver. The controller does not use the downlink's transceiver for transmitting, thus, a permanent logical '1', that is, a recessive value, is driven into the transmission pin of this transceiver.

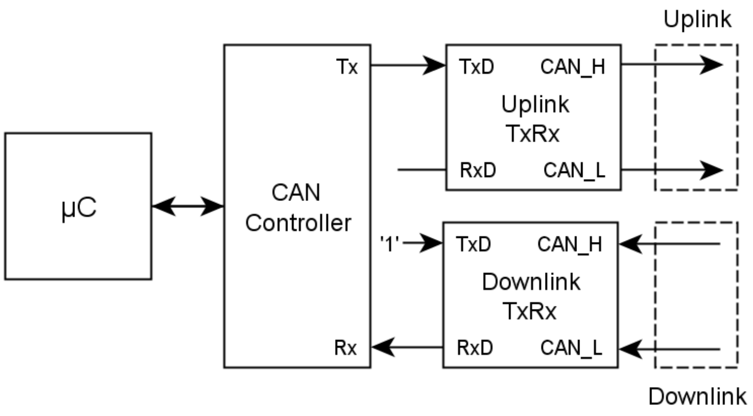


Figure 5.3.: CANcentrate node architecture. (The figure is based on a figure by Gessner [2010]).

As concerns the links themselves, note that each uplink and downlink within a given link is comprised of a pair of CAN_H and CAN_L wires, which is terminated at both its ends by appropriate resistors. This means that the structure of each uplink and downlink is equal to the one of a standard CAN bus, see Sec. 2.1.

CANcentrate's hub basic architecture is depicted in Fig. 5.4. Note that it is composed of three different parts called *Input/Output Module*, *Coupler Module* and *Fault-Treatment Module*. The former basically two sets of COTS CAN transceivers. One of the sets is used to translate the physical incoming signals from the nodes' uplinks into logic values, whereas the other one transform the logic value B_0 into a physical form that then is broadcast through each node's downlink.

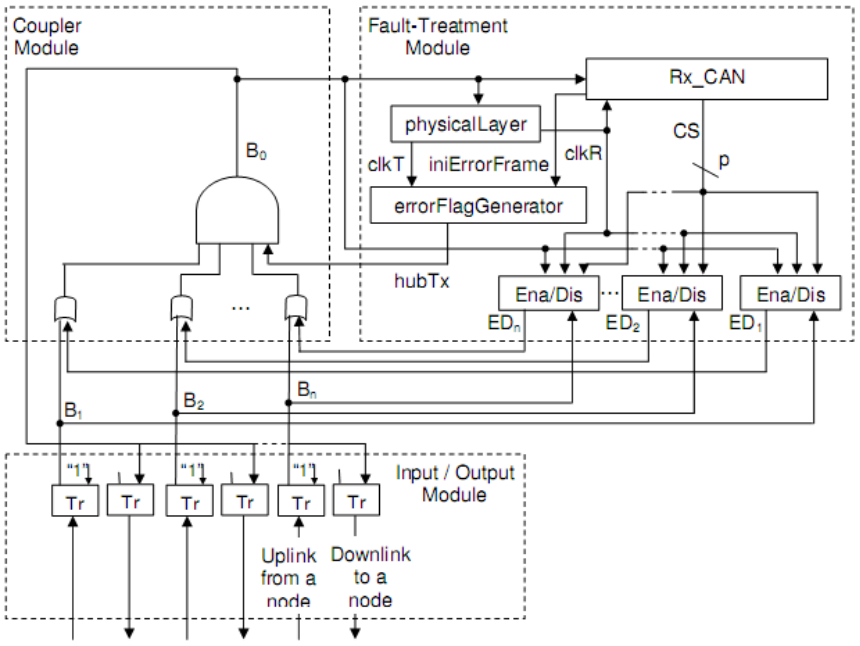


Figure 5.4.: Internal structure of a CANcentrate hub. (Reprinted from a technical report by Barranco [2010] with his permission).

The Coupler Module, couples the contributions of the nodes, $B_1, B_2, \ldots, B_n$, by means of an AND gate, thereby obtaining what we call the *resultant coupled signal*, B_0. This signal is then transmitted back to the nodes using the Input/Output Module as just explained. Note that the Coupler Module's AND gate implements the wired-AND functionality of the CAN bus, thereby making the hub coupling transparent for the nodes. In this sense note that the frame that results from the coupling is the same as the one that would be observed at a CAN bus. We call this frame the *resultant frame*. It is also noteworthy that each node's contribution can be masked by means of a dedicated OR gate, located just at its corresponding input to the AND gate. Specifically, a given OR gate is used to disable the contribution of the corresponding node when that node is diagnosed as faulty.

The logical value used to enable and disable each node's contribution is generated by the

Fault-Treatment Module. This module monitors the coupled signal B_0 and each node contribution in order to detect faulty nodes. When it diagnoses a node as being faulty, it disables its contribution by driving a logical '1' into the OR gate that corresponds to that node. This module is composed on various submodules, namely: *Physical Layer*, *Rx_CAN*, *Error Flag Generator* and a set of *Enabling/Disabling Units*.

The *Physical Layer* module monitors the coupled signal in order to synchronize at a bit level. As a result of this synchronization, it generates two signals called *clkR* and *clkT*. On the one hand, *clkR* indicates when the bit observed at the coupled signal can be correctly sampled by the different hub's submodules. On the other side, *clkT* activates at the beginning of each bit time in order to indicate the correct instant in which any module of the hub can drive its own bit contribution into the Coupler Module's AND gate. As it will be explained later, there are some modules that need to do so, for instance, a module placed within the hub that wants to send a message to a node connected to the hub.

Rx_CAN monitors the coupled signal bit by bit in order to interpret it and generate a set of signals, labelled as CS, which describes the current state of the resultant frame, that is, Rx_CAN synchronizes the hub with the nodes at the frame level. On the other hand, Rx_CAN implements the same error-detection mechanisms as a typical CAN node, so that it can detect when any of the errors described in Sec. 2.2.4 affects the resultant frame. When so, it globalises the error by compelling the *Error Flag Generator* submodule to transmit an error frame. This leads the Rx_CAN submodule and the nodes of the system to re-synchronize among them at a bit and at frame level.

Finally, there is an *Enabling/Disabling Units* dedicated to each hub port. A given Enabling/Disabling Unit monitors the contribution of a specific port, the coupled signal and the current state of the resultant frame in order to detect when the port contribution is faulty. When an Enabling/Disabling Unit diagnoses its port as being faulty, it disables it by driving a logical '1' into the corresponding OR gate of the Coupled Module.

5.2. Testability features achievable by sfiCAN

A star topology like (Re)CANcentrate can be used not only for improving error containment and fault tolerance, but also for overcoming the limitations of previous fault injectors proposed for CAN, see Sec.4.3.

The main advantage of a star, in terms of fault injection, is that the privileged location of the hub(s) within the network can be used to inject faults with a high spatial resolution. Specifically, the separation between the uplink and the downlink in (Re)CANcentrate allows injecting faults independently in the signal transmitted and received by each node, that is, faults can be injected locally at the signal transmitted/received by each node.

Moreover, the hub(s) of (Re)CANcentrate is synchronized at bit and at frame level with the resultant frame, which is obtained when coupling every node contribution, as explained before. This means the hub identifies and interprets the meaning of each one of the bits that compose the resultant frame. Thus, the hub cannot only inject with high spatial, but also with high time resolution, as it can modify the value of every single bit it receives/transmits from/to each node.

This ability of precisely control when and where faults are injected is referred in the literature as a *testability* [Armengaud et al., 2008] benefit called *controllability*. In fact, the hub enables the injection of complex scenarios such as those leading to data inconsistencies [Rufino et al., 1998] or integrity errors, that is, errors that lead some nodes to accept spurious frames. Note that a traditional bus-based fault injector, that is, an injector which is connected as a node to a CAN bus, cannot perform these types of injection, since in a bus all contributions are mixed and, then, faults cannot be injected locally at the signal transmitted/received by each node.

Besides improving the resolution and controllability of previous CAN fault injectors, the use of a star topology like the one of (Re)CANcentrate yields other important testability benefits.

First, sfiCAN provides high *observability* [Armengaud et al., 2008] by means of a distributed log system. One the one hand, sfiCAN has a central logger placed in the hub, which monitors the response of each node separately, when faults are injected. On the other hand, sfiCAN includes one node logger per node, that is, a software logger, embedded in each node application, which gathers additional information concerning the node behaviour and its state during the test. All this information can then be used to interpret the behaviour of the nodes during the injected fault scenario for an ulterior analysis.

Second, sfiCAN yields *remote testing*, since a personal computer (PC)-based management station connected to the hub can use the CAN protocol to configure the fault injector and to retrieve the data acquired by the log system.

Third, sfiCAN can be used to transparently carry out an *online testing* [Armengaud et al., 2008] of a star-based CAN system that is already delivering its service. To offer this capacity as an addition to *offline testing* is interesting since, like other technologies, for instance, [FlexRay Consortium, 2005, Kopetz and Bauer, 2003, Pedreiras et al., 2005], CAN can adopt a star topology as a means to be fit for real-time highly reliable applications [Hall et al., 2008, Barranco et al., 2009, 2011].

Finally, apart from providing high spatial resolution, the centralization of the fault injector reduces the complexity and the cost of the whole fault-injection infrastructure, since there is no need to distribute different fault injectors locally at each node.

Part II.

Specification

6. List of requirements

The current project consists in designing and building up a fault-injection infrastructure, called sfiCAN, to test the behaviour of CAN-based system when global and local faults do occur in the channel. The user is the one in charge of defining where, when and what to inject, as well as of analyzing the results in order to verify the system's correct behaviour.

Next, the minimum set of properties sfiCAN must fulfil, are presented into two different groups, functional and non-functional requirements.

6.1. Functional requirements

The functional requirements are those that are related to the functionalities of sfiCAN. Specifically, they specify how the user must interact with sfiCAN and what is the fault model the fault injector must provide, that is, what types of faults it must be able to inject. Next follows the list of these requirements.

- The user must be capable of using a program executing on a PC to specify the fault scenarios to be injected. For this purpose, the user must be provided with a simple an intuitive fault-injection specification language.
- SfiCAN must provide enough spatial resolution and granularity to independently affect the signal each node transmits and/or receives.
- SfiCAN must provide enough time resolution and granularity to independently modify the value of every single bit of a given transmitted and/or received signal.
- SfiCAN must be able to force any single bit to be dominant or recessive. This includes the feature of inverting the logical value of any bit transmitted and/or received by a node
- SfiCAN must be able to inject simple erroneous bit-patterns, that is, stuck-at and bit-flipping streams.
- SfiCAN must be able to reproduce scenarios involving several simultaneous erroneous bit-patterns.
- SfiCAN must be able to inject cascading erroneous bit-patterns, that is, to inject erroneous bits in the traffic that results when injecting previous errors.
- SfiCAN must be able to inject permanent and temporary faults, including transient and intermittent ones.

- SfiCAN must be able to inject faults without a previous knowledge of the traffic. For that purpose, sfiCAN must be able to start and stop an injection upon a specified bit pattern observed at the signal transmitted or received by a any given node.
- SfiCAN must collect enough information during the test to allow the user to check the behaviour of the system.
- The user must be capable of using a PC for retrieving the data collected during the test.

6.2. Non-functional requirements

The non-functional requirements are those that are related to how sfiCAN must be built up from both a physical and a logical point of view.

- SfiCAN must be transparent from the CAN nodes point of view.
- SfiCAN must be scalable, that is, it must be easy to enlarge sfiCAN in order to provide it with new features.
- SfiCAN must be implemented using COTS components.
- SfiCAN must be implemented by means of independent logical modules, which must be able to communicate with a PC through the CAN network itself.
- SfiCAN must be able to execute on a CAN network working at the maximum bit-rate, that is, 1 Mbps.

7. Fault-injection experiment phases

In order to use fault injection as a quantitative evaluation technique, it is necessary to carry out several fault-injection experiments to generate a statistically significant amount of data. A set of experiments is known as a *fault-injection campaign*. In turn, an experiment is carried out by injecting different faults that lead to a so-called *fault scenario*.

In sfiCAN, a given experiment is orderly performed in four different phases called *specification*, *configuration*, *execution* and *report*. Next, these phases are outlined.

- Specification phase: In this phase the user designs and specifies the fault scenario. The design implies the study of where, when and what to inject, whereas the specification involves the use of the fault-injection specification language to describe those injections.
- Configuration phase: In this phase sfiCAN self-configures its constituents components according to the fault-injection specification written in the previous phase.
- Execution phase: At this phase, the user triggers the execution of the experiment. Nodes start executing its software and generate traffic, then, sfiCAN injects faults in accordance to the conditions of the fault injection specification, and collects data regarding the experiment. Finally, the user stops the execution of the fault injection.
- Report phase: In this phase, the user retrieves the data that sfiCAN collected during the fault injection and, then, uses these data to check the system's behaviour.

8. SfiCAN architecture

The architecture of sfiCAN is composed of a set of CAN nodes, a CAN hub and a PC. Despite supporting the CAN protocol, the underlying network topology of sfiCAN is not a bus, but a star whose central element is the hub. All the nodes are attached to it following the CANcentrate's connection schema explained in Sec. 5.1. Additionally, a PC is also connected to the hub by means of and standard CAN connection. Fig. 8.1 shows all the components involved and their interconnections. Next, each one of these components is introduced.

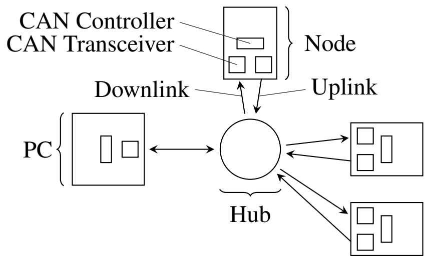


Figure 8.1.: sfiCAN architecture. (Reprinted from a technical report we published Gessner, Barranco, Ballesteros, and Proenza [2011]).

The main purpose of the hub is to couple the contributions of all the nodes and the PC, and then broadcast the resultant coupled signal. However, the hub includes some additional hardware to both inject faults and collect experimental data.

As concerns the nodes, their architecture is the one explained in Sec. 5.1. The most important aspect to take into account is that each node is connected to the hub by means of a dedicated link comprised of an *uplink* and a *downlink*, so that the injection can be performed unequivocally in any transmission direction. Similarly to the case of the hub, each node is provided with an additional software that includes mechanisms to collect experimental data.

Finally, the PC aims to interfacing the user with sfiCAN, so it can be remotely managed. In contrast to the nodes, the PC is connected to the hub through a CAN connection that is not divided into and uplink and a downlink, since no injection is performed in this connection.

Note that the fault injector is placed inside the hub in order to take advantage of the hub's privileged position. However, the experimental data collector is distributed. In order to provide not only global information about that traffic, but also local information about the state of the nodes, part of the experimental data collector is located inside the hub and part inside the nodes.

Finally, sfiCAN is scalable and modular. In fact, all the functionalities introduced above are implemented by means of distributed network-communicated independent logical modules called Network Configurable Components (NCCs). Next, this paradigm is discussed in-depth.

9. SfiCAN NCC architecture

As introduced previously, the main functionalities of sfiCAN, that is, fault injection and experimental data collection, are performed by distributed logical modules called Network Configurable Components (NCCs). Fig. 9.1 shows the general NCC-based architecture of sfiCAN.

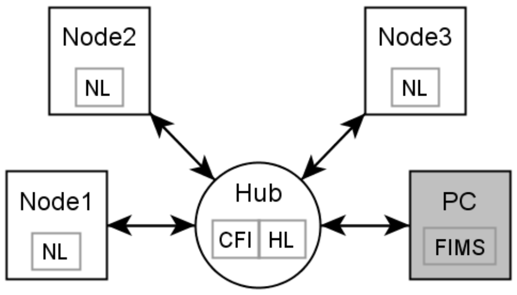


Figure 9.1.: SfiCAN NCC architecture.

An NCC is a logical unit placed inside an existing physical component and which carries out a specific task of a fault-injection experiment. For this purpose, the NCC extends the functionalities of the component it is placed into.

There are three different types of NCCs, namely *Centralized Fault Injector* (CFI), *Hub Logger* (HL) and *Node Logger* (NL). There is only one Centralized Fault Injector, which is placed inside the hub and which implements the fault injector itself. That is, it injects errors in the uplink and/or downlink of each node, on the basis of the fault-injection specification provided by the user.

There is also only one Hub Logger, which is placed in the hub. On the one hand, it monitors and registers information about the traffic from the hub's point of view during the execution of the fault-injection experiment. On the other hand, once the experiment finishes, it transfers the log information to the user, as indicated below.

Finally, there is a dedicated Node Logger placed inside each node. Analogously to the Hub Logger, it collects experimental data from the node's point of view. For this purpose, it monitors and registers specific events related to both, the application being executed, and the CAN controller. Once the fault-injection experiment finishes, the Node Logger transfers the logged information to the user.

The user remotely manages all the NCCs from the PC, which runs a specific software called *Fault-Injection Management Station* (FIMS). The FIMS works as an interface between the user and sfiCAN, and manages and coordinates the operation of the NCCs to carry out a given fault-injection experiment. Basically, it configures the Centralized Fault Injector in accordance to the fault-injection specification provided by the user and, then, compels all NCCs to start the experiment. Once the fault-injection experiment finishes, the Fault Injection Management Station retrieves the data collected by the Hub Logger and the Node Loggers during the experiment.

The NCCs behave in accordance to different *NCC operating modes*, in order to support the three last phases that constitute a fault-injection experiment. On the other hand, the communication between the FIMS and the NCCs is carried out thanks to a specific network protocol we designed on top of CAN, called *NCC protocol*. This protocol is based in the exchange of so-called NCC messages, and supports both, unicast and broadcast addressing, that is, an NCC message can be sent to one or all NCCs.

Next, the operating modes of the NCCs as well as the NCC protocol are described in more detail.

9.1. NCC operating modes

As explained before, an NCC behaves in accordance to different operating modes, in order to support the experiment phases presented in Sec. 7. Specifically, there are four different operating modes, called *configuration mode*, *idle mode*, *wait-for-whistle mode* and *execution mode*. The automaton presented in Fig. 9.2 shows the possible transitions among these operating modes. These transitions are triggered by a set of NCC messages sent by the FIMS, using the above-mentioned NCC protocol. Note that, since there is no participation of the NCCs in the specification phase, none of these modes is related to that phase.

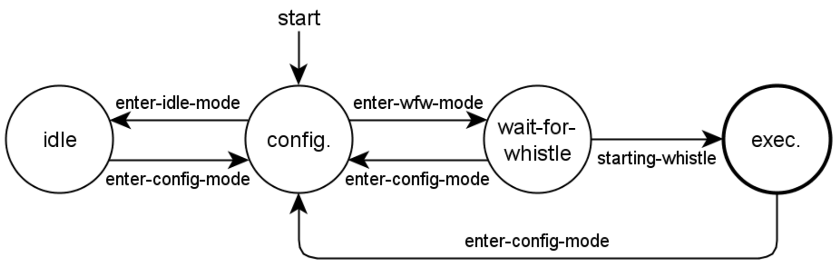


Figure 9.2.: Operating mode automaton. States involved in the execution of the experiment are bold-bordered.

When sfiCAN is initialized all the NCCs are in the configuration mode. In this mode the FIMS communicates with the NCCs to set up what is the behavior of each one of them during the experiment. After configuring a given NCC, the FIMS can send either an `enter-wfw-mode message` or an `enter-idle-mode message`. The first drives the NCC to the wait-for-whistle mode, in which it waits until the rest of the NCCs are ready to start the experiment. The second message drives the NCC to idle mode, which forces the NCC to remain disabled during the experiment.

Once all NCCs involved in the experiment are in the wait-for-whistle mode, the FIMS can start the execution of the experiment by sending a broadcast `starting-whistle message`. This compels all these NCCs to enter in the execution mode, in which they play their corresponding role during the fault-injection experiment. At the same time, nodes' applications carry out the specific task they were programmed for.

Finally, in order to finish the execution phase and then enter into the report phase, the FIMS sends a broadcast `enter-config-mode` message. This forces all NCCs to enter in configuration mode again. Next, the FIMS retrieves the log information collected by the logging NCC loggers, that is, the Node Loggers and the Hub Logger.

Note that, as can be extracted from the previous paragraphs, the operation of a given physical component can be disabled by the NCCs standing inside it. For instance, a given Node Loggers can restrict the execution of the node program, depending on its operating mode. In contrast, since the FIMS should be able to communicate with the NCCs at any time, the hubs operation is always enabled, that is, it always broadcasts the CAN messages received.

9.2. NCC protocol

As said before, the communication between the FIMS and the NCCs is performed using a so called NCC protocol on top of CAN, which is based on the exchange of a set of so-called NCC messages. Basically, each NCC message can be of three different types called *configuration*, *logging* and *mode change*, depending on its purpose.

The operation of an NCC during the execution of an experiment can be set up thanks to the configuration messages, as already explained. Once an experiment finishes, logging messages are used to retrieve log information from the Hub Logger and the Node Loggers. Finally, as seen in Sec. 9.1, mode change messages allow forcing an NCC to switch among different operating modes.

As concerns the addressing schema of the protocol, each message can be transmitted to either one or all NCCs. On the one hand, each NCC is unequivocally identified by means of an *NCC identifier* (NCC ID). On the other hand, there is one NCC ID reserved for broadcast addressing.

Part III.

Design

10. Fault-injection specification

10.1. Fault-injection specification basics

We have developed a *fault-injection specification language* in order to provide the user with a tool for specifying the faults to be injected by the sfiCAN's fault injector. Note that the types of faults this fault injector can provoke, that is, the sfiCAN's *fault-injection model*, were already explained in Sec. 6.1. At the end of this section, List 10.7 specifies the Backus-Naur Form (BNF) syntax of this language in the ISO/IEC 14977 standard [ISO and IEC, 1996]. This listing shows all the possible syntax combinations of this language, although not all of them are semantically correct. Most of the these restrictions can be inferred from how the CAN protocol works; for instance, it is not possible to inject in the 12th bit of the identifier field which, as explained in Sec. 2.2.1, has a size of 11 bits long. Later on, in this chapter, we explicitly indicate the most important restrictions.

By means of this language the user can describe a given experiment as a set of labelled *fault-injection configurations*, each of which contains a set of key-value pairs called *fault-injection parameters*. Specifically, the set of fault-injection parameters describe what, where and when to inject.

What to inject is described by the parameter `value_type`, which can be one of the values listed next: `dominant`, `recessive`, `inverse` and `pattern`. The value injected in the two first cases is obvious. To generate a bit-flip of the coupled signal, option `inverse` must be selected. Finally, the option `pattern` allows the injection of a predefined bit pattern.

Where to inject is described by the parameter `target_link`, which can select any of the links of any of the ports, except the PC port. For instance, `target_link = port1dw` selects the downlink of port 1.

The specification of when to inject is more complex since sfiCAN must be able to generate permanent and temporary faults with bit resolution. Moreover, traffic patterns must be used to provoke this injection alternations. To fulfill all these requirements the injector has been designed to behave in accordance with a deterministic automaton.

10.2. Fault-injection modes

How the injection is performed is described by the automaton shown in figure 10.1. States can be divided in two groups, called final and non-final. The difference between them is that in final states, bold-bordered circumferences, the injection is active, while in non-final states, non-bold-bordered circumferences, the injection is inactive. On the other side, transitions are shown as arrows carrying the set of conditions that must be fulfilled in order to progress to the next state. Note that, there are three conditions which divide the automaton, depending on a *fault-injection* *mode*. The rest of conditions are specified in terms of the traffic, and are explained later.

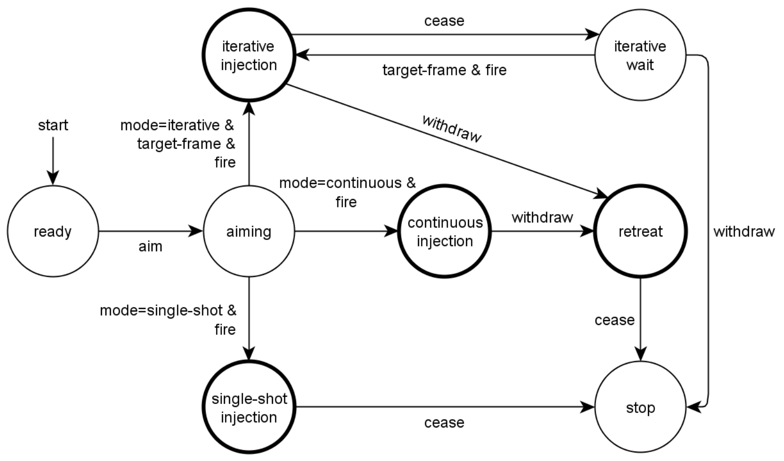


Figure 10.1.: Automaton injector state machine.

Fault-injection mode drives the execution of the automaton in order to perform transient, periodical or permanent faults. Specifically, three different modes have been defined: *single-shot*, *continuous* and *iterative*. The first mode is used when injecting limited duration transient faults. The second allows the generation of long transient, as well as permanent faults. Finally, to provoke periodical intermittent faults, the iterative mode is used. Although various of these modes can be used to specify a given fault, the final description can be more complex depending on the one chosen. The user is responsible for knowing how the injection is done, when using every mode, and choosing the one that best suits in each case.

On the other side, traffic-based conditions are used to find specific time points in which the injection starts/ends. These conditions can be divided in two groups: *frame-based conditions* and *bit-based conditions*. Next, they are introduced, however, later each traffic-based condition is described in-depth.

- Frame-based conditions: This type of conditions allows to specify an event involving the content of a frame and its number of transmissions. The condition can apply a bit filter in a frame field in order to identify them. Moreover, the source link of the frame and the role being played by the node being injected can also being taken into account. Additionally, as said, it is also possible to specify the number of times that the given frame must be seen in order to fulfill the condition. When this last aspect is dismissed the condition act as a frame selector. For instance, `count=3, filter=xxxxxxx1011, field=id, link=port2up` activates after *node2* transmits three times a frame whose identifier ends with a '1011', note that the 'x' corresponds to a don't-care value. There are three frame-based: *aim*, *withdraw* and *target-frame*.

- Bit-based conditions: This type of conditions allows to specify an event involving the current bit being transmitted. The condition can identify a specific bit within a frame field, or a bit being transmitted later from a known event, which depends on the condition. For instance, condition `bit=0, field=crc` activates when the current bit being transmitted is the first of the CRC field, note that bits are numbered from 0 to the length of the field minus 1. On the other side, condition `offset=3` activates when the current bit being transmitted is the third after the accomplishment of a previous condition. Bit-based conditions being defined as a bit-field pair don't take into account stuff bits, whereas when defining a later bit stuff bits are included. There are two bit-based conditions: *fire* and *cease*.

Next, in order to describe how each mode works, the automaton is divided and covered, taking into account the fault-injection mode.

10.2.1. Single-shot mode

This mode allows to inject limited duration transient faults, that is, a single non-permanent fault that last a predefined amount of bits. The beginning of the injection is defined with a frame-based condition plus a bit-based condition, whereas the end is defined with just a bit-based condition. Fig. 10.2 presents the branch containing single-shot mode, which manifests this fact. Due to how the end is defined, it is useful when injecting single bits or when the end of the injection is a few frames far. Next, this automaton is covered and explained in-depth.

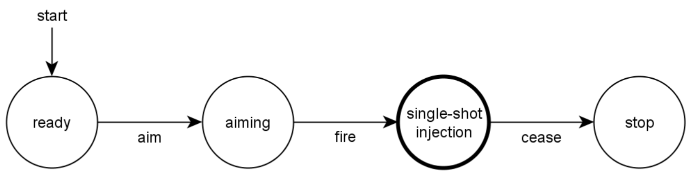


Figure 10.2.: Single-shot automaton.

The initial state of sfiCAN, when activated, is *ready*, to progress to *aiming* it is necessary to fulfill the frame-based condition called *aim*. This condition allows to start the injection after a specific event involving the occurrence of a specific frame a given number of times. Specifically, this frame is characterized by the content of the one of its field. This is specified with `aim_filter` and `aim_field`. The first allows to define a filter that is applied to the field defined in the second. Moreover, it is possible to filter frames depending on their source. Parameter `aim_link` allows to monitor only values transmitted in a particular link: downlink, uplink either the coupled signal. Additionally, it is possible to, by means of the `aim_role` parameter, append a condition involving the role being played by the target node, which can be: receiver, transmitter or don't care. Note that the specification of this parameter can be avoided, in this case `dont_care` value is assumed. Finally, the number of times that this frame must be seen, is specified with the `aim_count` parameter.

An example of an *aim* condition is shown in listing 10.1 which activates when detecting the third occurrence of a frame containing an even value in the identifier, and the target node is not the transmitter. Note that the 'x' in the `aim_filter` act as a don't care value.

```
aim_count   = 3
aim_filter  = xxxxxxxxxx0
aim_field   = identifier
aim_link    = coupled
aim_role    = rx
```

Listing 10.1: Aim condition example.

When the injector is in *aiming* state, a fulfilled *fire* condition is needed to progress to *single-shot* final state. This condition enables the possibility to start the injection in a specific bit, defined by `fire_bit`, of a specific field, defined by `fire_field`. Moreover, a `fire_offset` parameter allows to start the injection a given number of bits after the previous condition. This is useful when generating cascading faults, that is, when injecting after a specific scenario provoked by the injector itself. Additionally, the `fire_offset` parameter can be used to inject single stuff bits.

An example of a fire condition is shown in listing 10.2. In this case it is activated 2 bits after detecting the ack bit. This specification is useful, for example, when it is known that a receiving node is going to reject a frame and an injection must be performed inside the error flag.

```
fire_bit    = 0
fire_field  = ack
fire_offset = 2
```

Listing 10.2: Fire condition example.

In single-shot mode an injection ends, that is, progresses to the *stop* state, after a *cease* condition. This condition allows to be defined in terms of a bit count or a bit-field pair. The first enables the condition after a given number of bits, whereas the second when seeing a specific bit of an specific field. However, it is not possible to specify both conditions simultaneously. Note that, when defining this condition as a bit-field pair, the bit specified is not injected.

An example of a *cease* condition, using the bit-field pair, is shown in listing 10.3. The injection is active until the first bit of the CRC delimiter. Thus, the last bit of the CRC field is injected, in contrast to the bit of the CRC delimiter.

```
cease_bit   = crcdelim
cease_field = 0
```

Listing 10.3: Cease bit-field condition example.

An example of a *cease* condition, using a bit count, is shown in listing 10.4. In this case injection stops after 2 bits, the bit fulfilling the *fire* condition and the next one.

Once in *stop* state, there is no option to progress, thus, the injection is disabled until the experiment is reloaded.

```
cease_bc = 2
```

Listing 10.4: Cease count condition example.

10.2.2. Continuous mode

This mode allows to inject multi-frame transient and permanent faults. The conditions involved in continuous mode are the same as in single-shot mode, except for a new frame-based one at the end of the injection. Fig. 10.3 shows the branch corresponding to this mode. Next, this automaton is covered and explained in-depth.

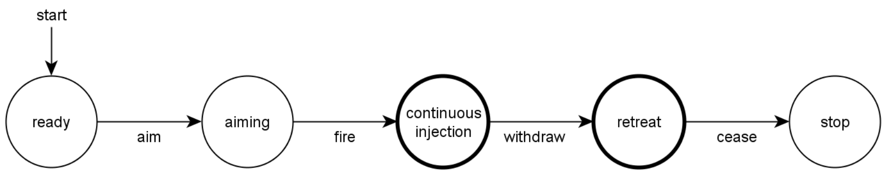


Figure 10.3.: Continuous automaton.

As in single-shot mode, to progress from *ready* to *aiming*, a fulfilling *aim* condition is needed, which allows to start the injection after a specific event involving the occurrence of a specific frame a given number of times. Next, the accomplishment of a *fire* condition allows to progress to a *continuous injection* state, in which the injection starts. In Contrast to the single-shot mode there is not a direct transition to the *stop* state. Instead, there is a new final state called *retreat*, which first forces to fulfill a *withdraw* condition. The *withdraw* condition is identical to the *aim* condition, that is, it allows to wait until a specific frame is seen a given number of times.

The specification shown in Listing 10.5 defines a condition activating after seeing node0 no transmitting 10 times the ACK, thus, either node0 is down or exists a previous injection which masks its uplink.

```
withdraw_count  = 10
withdraw_filter = 1
withdraw_field  = ack
withdraw_link   = port0up
withdraw_role   = dont_care
```

Listing 10.5: Withdraw condition example.

Once in *retreat*, a fulfilled *cease* condition is needed to progress to *stop*, where the injection ends, just like in single-shot mode. Note that, since the end of the injection is performed in a different frame from the starting one, due to the *withdraw* condition, the possibility of using a bit count as a *cease* condition is not considered. Thus, in continuous mode the injection always ends in a specific bit of a specific field after the accomplishment of the *withdraw* condition.

All this fault-injection specification, involving the *withdraw* and *cease* conditions is used when injecting a multi-frame transient faults. To inject a permanent fault it is enough to avoid the specification of the *withdraw* and *cease* conditions. When so, the state cannot progress from the *continuous injection* and, thus, the injection never ends.

10.2.3. Iterative mode

This mode allows to inject periodical intermittent faults. The idea used is similar to the one used in continuous mode, the injection is performed within a set of frames delimited by an *aim* and *withdraw* conditions. However, in this case, *fire* and *cease* conditions are used to define the range, within each frame, in which bits are injected, in contrast to continuous mode, where the whole range of frames is injected. Thus, only a set of bits are injected in each frame of the set of frames involved. Moreover, it is possible to apply a filter, by means of a *target-frame* condition, in order to restrict the injection to some of the frames of the set. Fig. 10.4 shows the branch corresponding to this mode. Next, this automaton is covered and explained in-depth.

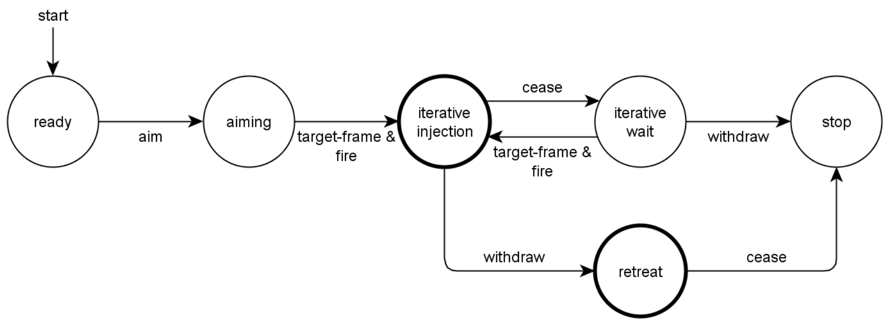


Figure 10.4.: Iterative automaton.

As in both cases described above, to progress from *ready* to *aiming*, a fulfilling *aim* condition is needed, which allows to start the injection after a specific event involving the occurrence of an specific frame a given number of times. However, to progress to the *iterative injection* final state it is necessary, not only to fulfill a *fire*, but also a *target-frame* condition. The definition of the *target-frame* condition, whose specification is described later, allows to select a specific type of frames to be injected, that is, allows a selective periodical intermittent injection. However, if its definition is avoided no selective injection is performed and, thus, the *target-frame* condition is always accomplished.

Once in *iterative injection* the injection starts. At this point, the alternation between the *iterative injection* and the *iterative wait* performs the intermittent injection. Specifically, the *cease* condition brings the injector to the *iterative wait*, where the injection stops. Then, when another frame fulfilling the *target-frame* condition appears, and the bit specified by the *fire* condition is found, a transition to the *iterative injection* is performed and, thus, the injection starts again.

If a withdraw condition is activated in any of these states, *iterative injection* or *iterative wait* the flow changes to the *retreat* or the *stop* state, depending on the source state. Since the *iterative wait* implies that the *cease* condition has been accomplished, that is, the injection is stopped, the automaton can advance safely to the *stop* state. On the other side, when being in the *iterative* *injection* state, and, after fulfilling the *withdraw* condition, the automaton forces to advance to the *retreat* state where the injection continues until a *cease* condition is activated. This ensures that the end the injection occurs in an well-known point of the frame.

The definition of the *target-frame* condition is similar to the one in *aim* or *withdraw* conditions, but without specifying an event count, thus, it activates when seeing, in a specific link (`target_frame_link`), a frame containing a specific value (`target_frame_filter`) in a specific field (`target_frame_field`) when the target node acts in a specific role (`target_frame_role`).

An example of a *target_frame* is shown in listing 10.6. This condition activates when seeing in the coupled signal a recessive value in the RTR field, that is, a remote frame.

```
target_frame_filter = 1
target_frame_field  = rtr
target_frame_link   = coupled
target_frame_role   = dont_care
```

Listing 10.6: Target-frame condition example.

```
FI_SPEC    = { '[', STRING, ']', FI_CONFIG }
FI_CONFIG  = VALUE,
             LINK,
             MODE,
             AIM, FIRE,
            [WITHDRAW], [CEASE],
            [TARGET_FRAME] ;

VALUE = 'value_type',     '=', VALUE_TYPE_VALUE,
       ['value_pattern', '=', ('0'|'1'), {'0'|'1'} ] ;

LINK = 'target_link', '=', LINK_VALUE - 'coupled' ;

MODE = 'mode' '=' 'single-shot' | 'continuous' | 'iterative' ;

AIM = 'aim_count',  '=', NATURAL,
      'aim_filter', '=', FILTER_VALUE,
      'aim_field',  '=', FIELD_VALUE,
      'aim_link',   '=', LINK_VALUE,
     ['aim_role',   '=', ROLE_VALUE ] ;

FIRE = 'fire_bit',    '=', NATURAL ,
       'fire_field',  '=', FIELD_VALUE,
       'fire_offset', '=', NATURAL ;

WITHDRAW = 'withdraw_count',  '=', NATURAL,
           'withdraw_filter', '=', FILTER_VALUE,
           'withdraw_field',  '=', FIELD_VALUE,
           'withdraw_link',   '=', LINK_VALUE,
          ['withdraw_role',   '=', ROLE_VALUE ] ;

CEASE = 'cease_bc',    '=', NATURAL
      | 'cease_bit',   '=', NATURAL,
        'cease_field', '=', FIELD_VALUE ;

TARGET_FRAME = 'target_frame_filter', '=', FILTER_VALUE,
               'target_frame_field',  '=', FIELD_VALUE,
               'target_frame_link',   '=', LINK_VALUE,
              ['target_frame_role',   '=', ROLE_VALUE ] ;

VALUE_TYPE_VALUE = 'dominant' | 'recessive' | 'inverse' | 'pattern' ;

FILTER_VALUE = ('0'|'1'|'x'), {'0'|'1'|'x'} ;

LINK_VALUE = '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' | 'tx' | 'rx' ;
```

Listing 10.7: Fault-injection specification BNF.

11. Hub design

11.1. General architecture

The central element of the sfiCAN architecture is a CANcentrate-based hub, see Sec. 5.1, which is responsible for intercommunicating the nodes, injecting faults and collecting experiment data. It is divided in two parts: the *input/output module* and the *hub core*. The former translates the physical signals received from the nodes and the PC into a logical form and vice versa. The second is the part of the hub that implements its coupling, fault-injection and logging mechanisms. Fig. 11.1 shows this division and the main components of each one.

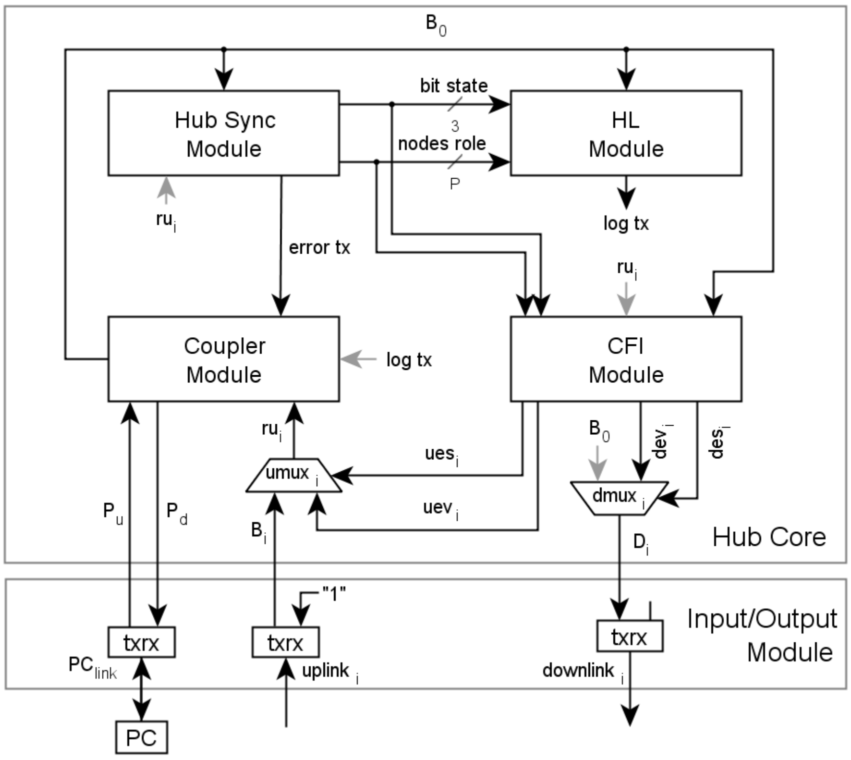


Figure 11.1.: Hub diagram

Note that there is a dedicated port for the PC connection. This connection, as explained in Sec. 8, allows the communication between the PC and the set of NCCs. However, since it should handle a standard CAN device, and no errors are injected in the PC link, its design differs from a regular node port. The issue of combining these two connections is discussed in Sec. 11.4.

11.2. Input/Output module

The input/output module transforms the CAN_H and CAN_L signals from each node to logical signals, by means of COTS transceivers. This is done for each node and transmission direction, just as CANcentrate does, see Sec. 5.1.

However, sfiCAN has an important addition, a dedicated PC port. Since a PC uses a standard CAN interface and the star uses dedicated links for each transmission direction, it is necessary to adapt this connection. This modification is described deeply in Sec. 11.4.

11.3. Hub core

The functionality of the hub core is performed by four different interconnected modules called *Coupler module*, *Hub Sync module*, *Central Fault Injector module* (CFI module) and *Hub Logger module* (HL module), just as can be seen in Fig. 11.1.

First, after the processing done in the input/output module, the contributions of the nodes, B_i, and the PC, P_u, are driven to the coupler module. Note that the contribution of a given node i can be injected using the multiplexer labelled $umux_i$, which generates the *resultant uplink* signal labelled ru_i. All those signals involved in the multiplexer are generated by the CFI module, which is explained later.

After that, the Coupler Module couples all these contributions by means of an AND gate. This module is similar to the coupler module in CANcentrate's design, just minor modifications have been done to adapt it to this new environment. Specifically, all the signals related to the fault-treatment mechanisms were removed, whereas new contributions, explained later, were added. This module generates two different signals, called B_0 and P_d. On the one hand, the first contains the signal that results from coupling all the contributions. This signal is the one used by the rest of the modules as the coupled signals. Moreover, it is also driven to the downlinks of the nodes. On the other side, P_d is specifically generated for the PC and its construction is explained later, in Sec. 11.4.

Later, the Hub Sync Module monitors the nodes' contributions and the B_0 coupled signal in order to synchronize the other modules with what is being transmitted. For this purpose, it provides them with three different signals. First, a signal called *frame state* describes the current state of the frame, that is, the meaning of the bit being broadcast. For example, the frame field it belongs to or whether or not it is a stuff bit. Second, the *nodes role* signal informs of the role being played by each node. In order to generate this signal the Hub Sync Module traces the values transmitted during the arbitration. Finally, in order to synchronize the hub at a frame level, the *error tx* signal contains all those error frames generated when an error is detected. Note that all the logic needed to generate these signals was already implemented in the internals of the CANcentrate hub, which is explained in Sec. 5.1.

The Central Fault Injector module stores the fault-injection specifications generated by the user and, then, when corresponding, injects the faults defined. A given fault consists in a single bit whose value deviates from what it is expected, according to the current state of the frame. The uplink and downlink signals of each node can be modified centrally and independently by means of dedicated *uplink multiplexers* (*umux*) and *downlink multiplexers* (*dmux*). These multiplexers are managed by a set of signals generated by this module. Specifically, for each multiplexer two signals are provided, the one that is used to enable/disable the injection and the one containing the value to be injected. In the uplink of $node_i$ a $umux_i$ is used to force a specific *uplink error value* (uev_i) when the corresponding *uplink error selection* signal (ues_i) is active. Analogously, *downlink error selection* (des_i) signal and *downlink error value* (dev_i) signal are used in $dmux_i$ to inject errors in the *resultant downlink* signal, D_i, of $node_i$.

During an experiment, the Hub Logger module logs information about the traffic. Specifically, for each received frame, it stores the source, the content and, when an error is detected, the bit provoking the error. Note that, besides using *frame state* and *nodes role* signals for correctly interpreting B_0, the Hub Logger module generates its own contribution called *log tx*, which is driven to the Coupler Module. This signal is used to send the logger information to the Fault Injection Management Station, when a fault-injection experiment finishes.

11.4. Connecting a standard CAN device to the sfiCAN hub

As introduced previously, the hub provides a dedicated port for the PC which differs from a regular node port. The main difference lies in the fact that it should handle a standard CAN device, a PC CAN controller, which uses just one link to perform a double-direction communication. Note that, a standard CAN port in the hub allows, not only to connect a standard CAN device, but a standard CAN bus. In turn, this fact allows the implementation of an hybrid topology between a bus and a star, like the one in Fig. 11.2. The solution adopted implies modifications in two different levels, at a physical level, in the input/output module, and at a logical level, in the coupler module. Next, this solution is presented and discussed in detail.

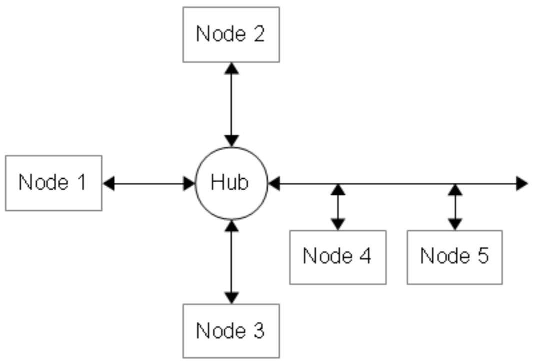
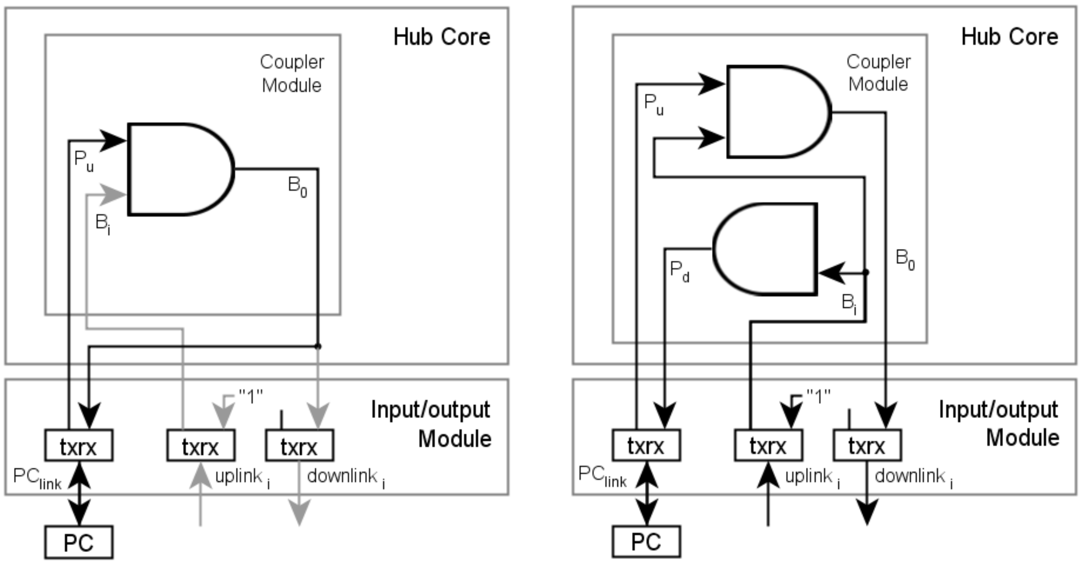


Figure 11.2.: Hybrid topology star-bus schema.

First, the CAN device must connect with the hub at a physical level. That is, set up the input/output module to translate the signal carried by the CAN_H CAN_L pair, provided by the device, into a logical form. This is done simply by using just one transceiver in the port.

Once the hub is able to translate the PC signal is necessary to couple it just as the nodes signals. This is done by means of the coupler module, which, as said in Sec. 11.3, performs the logical AND of all the contributions. However, when connecting a CAN device using a single transceiver for both transmission direction, a dominant feedback appears. The path causing the feedback can be seen in Fig. 11.3a. A dominant feedback causes a general failure since it floods all the downlinks with a dominant value.



(a) Feedback scenario in a simplified diagram of the hub. **(b)** PC dedicated AND solution.

Figure 11.3.: Problem and solution of the standard CAN device connection.

This failure occurs, in the current architecture, when any contribution provides a dominant value. First, the dominant value arrives to the coupler module where, no matter the rest of values, it sets the output as dominant. Next, this output is driven to the input/output module, where, by means of the transceivers, this signal is delivered to the nodes and the PC. In contrast to what happens in regular node links, when a value is set in the PC link it returns to its source since this link is a CAN bus. Thus, the dominant value is driven to the coupler module again, where the previous behaviour perpetuates indefinitely. Moreover, although the source of the dominant value ceases its transmission, the transceiver regenerates the signal.

The solution adopted consists in avoiding the transmission of the contribution of the PC to itself. This can be achieved by modifying the coupler module design, in order to add a new AND gate which does not includes the PC contribution. The output of this new AND gate is driven to the PC transceiver, which translates and sends the signal through the PC link. A simplified version of the final design of the internal of the coupler module, as well as its connection with the input/output module can be seen in Fig. 11.3b. Note that, although this new signal does not includes the contribution of the PC, the PC can still receive the coupled signal of all the contributions. This is possible since the PC link itself couples the PC contribution with the new generated signal, which carries the coupled value of the rest of the contributions.

The dedicated-AND solution has one important limitation, only one standard CAN device can be connected in the domain. When using the previous architecture to add one or more CAN standard devices other non-fixable dominant feedbacks appear inside the hub, preventing the correct channel operation.

12. Node design

The logic architecture of the sfiCAN nodes is designed using the three-layered schema presented in Fig. 12.1. The top is divided in two parts. On the one hand, there is an application which performs the regular operation of the node, that is, it generates the node's workload. On the other hand, we add the Node Logger, which monitors both the application and the driver to register specific events. Moreover, in order to receive instructions from the Fault Injection Management Station and send the log report, the Node Logger can interact with the driver. The medium is the driver, which contains a regular CAN driver which allows to communicate the top level with the CAN device. Finally, the CAN device conforms the node's communication hardware, that is, the CAN controller.

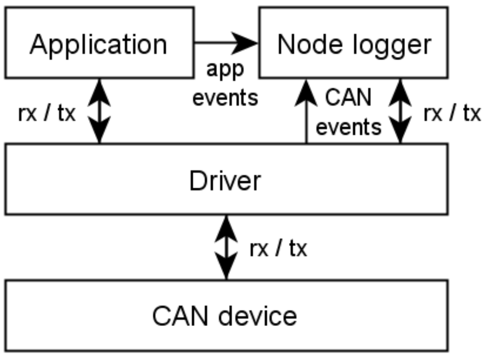


Figure 12.1.: Node diagram.

13. SfiCAN NCC architecture design

As explained in Sec. 9 the operation of sfiCAN is performed thanks to a set of various Network Configurable Components (NCCs). These logical units are placed inside existing physical components and carry out specific tasks, such as fault-injection and data collection. Moreover, their behaviour is driven by an operating mode, which aims to synchronize them in order to follow the phases that constitute an experiment, see Sec. 7. At the same time, the Fault Injection Management Station (FIMS), which runs in the PC, uses a specific network protocol on top of CAN, called NCC protocol, to communicate the user with the NCCs.

The next sections describe the design of the different elements conforming this architecture, that is, the FIMS, the set of NCCs and the NCC protocol.

13.1. Fault Injection Management Station (FIMS)

The operation of the FIMS is divided in three parts, one for each task it carries out, called *Mode Changer*, *Fault-Injection Configurator* and *Fault-Injection Log retriever*. Next, each part is described more in-depth.

13.1.1. Mode Changer

The Mode Changer enables the possibility to change the operation mode of an NCC, in order to keep consistence with the experiment phases introduced in Sec. 7. The set of messages and possible transitions have already been defined in Sec. 9.1.

The design of this part is quite simple, since the only need to change the operation mode of an NCC is an NCC message which, in turn, is mapped in a data CAN frame. In fact, no software has to be developed to achieve this feature, instead, it is possible to use the set of tools provided by the PC CAN controller manufacturer and the Linux community. Specifically, as explained later, in Sec. 18.3.1, a command provided by SocketCAN aims to send data CAN frames. Thus, by constructing a simple bash script it is possible to force an NCC to transit among operating modes.

13.1.2. Fault-Injection Configurator

The Fault-Injection Configurator allows to transfer the fault-injection specification in the CFI. The operation of the fault-injection configurator is divided in two parts. The first validates, by means of a syntactical and semantical analysis, the fault-injection specification. The second transmits the specification to the CFI, by means of the NCC protocol. Additionally, it is also possible to execute just the first part of this software, that is, only check the correctness of the fault-injection specification. In order to perform these actions, it is necessary to provide both, a fault-injection specification file, that is, a file containing a fault-injection specification compliant with the definition carried out in Sec. 10.1, and the execution mode of the program, that is, whether the program must validate and transmit the fault-injection specification, or only validate it. Next the pseudocode of this program, presented in List. 13.1, is discussed.

First, the default values of the constants representing the fault-injection specification file, `fi_spec_file`, and the program mode, `program_mode`, are generated from the arguments passed through the shell. The fault-injection specification file variable contains the fault-injection specification file itself, whereas the program mode has two possible values, `MODE_EXEC`, when requiring a validation and a transference of the fault-injection specification, and `MODE_CHECK`, when only a validation has to be performed. The initialization ends after setting to 0 the default value of variable `errors`, which is used to count the number of errors found during the validation, in order to later decide whether the transfer of the fault-injection specification can be performed.

Then, the content of the fault-injection specification file is validated. First, the program reads the file and extracts the set of fault-injection configurations into the variable `fi_configs`. Then, it processes each configuration (variable `fi_config`) included in this set. Specifically, it validates each one of the fault-injection parameters that constitute the `fi_config`, see Sec. 10.1. If a given fault-injection parameter is not valid, either because its name is syntactically incorrect, or because it has an erroneous value, the program prints a text informing of the error and increases the value of the so-called variable `errors` by one.

Once the validation has been carried out, the program transmits the fault-injection specification to the CFI, unless it has found an error or if it was invoked to simply check the fault-injection specification.

In order to transmit the specification, the program accesses to the fault-injection parameters as explained for the validation. For each one of them, it builds up an NCC message with the information needed to configure the CFI accordingly and, then, invokes a transmission routine that sends it by means of one or more CAN frames. Later, in Sec. 13.6 we explain how this transmission is carried out.

```
[program_mode, fi_spec_file] = process_shell_args();
errors = 0;

//validate fault-injection specification
fi_configs = read(fi_spec_file);
foreach fi_config in fi_configs loop
  foreach fi_param in fi_config loop
    if not is_valid(fi_param) then
      print_error(fi_param);
      errors++;
    end if;
  end foreach;
end foreach;

if errors > 0 then
  exit(EXIT_FAILURE);
end if;

if program_mode = MODE_CHECK then
  exit(EXIT_SUCCESS);
end if;

//transmit fault-injection specification
```

```
fi_configs = read(fi_spec_file);
foreach fi_config in fi_configs loop
  foreach fi_param in fi_config loop
    config_msg_content = generate_ncc_config_msg_content(fi_param);
    send_ncc_msg(CFI_ID, config_msg_content);
  end foreach;
end foreach;
```

Listing 13.1: Fault-injection configurator pseudocode.

13.1.3. Fault-Injection Log retriever

The Fault-Injection Log retriever is responsible for fetching a showing the log data stored in the log system. It has three functionalities. First, when required, generates a log request, which forces NCC loggers to transmit all the collected log information. Second, collects and decodes the responses from the NCC loggers. Finally, shows the report to the user, in order to perform the log data analysis.

As concerns the log data provided by each type of NCC logger, the specific content is described next. Since the hub has a bit-by-bit view of each node's traffic the Hub Logger is able to collect, not only the set of frames transmitted by each node, but, in case of error, the specific bit causing it. In contrast, a given Node Logger can collect the set of frames transmitted and received by that node, as well as, the value of the error counters. The general operation of the Fault-Injection Log retriever, shown in List. 13.2 as a pseudocode, is described next.

First, note that it is composed of a set of parametrized functions calls, one for the Hub Logger and three for the Node Loggers, that is, there is a specific function for each type of NCC logger. As concerns the Hub Logger function, it constructs and sends a log request, that is, a command forcing the Hub Logger to transmit all its stored log. Then, it enters in a loop, where it receives and treats all the NCC messages send by the Hub Logger. Note that the type of message can be extracted from its content, thus, a *case* structure is used to distinguish and treat the different NCC messages. Specifically, a Hub Logger can transmit two type of messages called, `hub-stored-frame message` and *end-of-log message*. On the one hand, a given `hub-stored-frame message` contains a single frame seen by the hub, thus, its possible to reconstruct all the traffic by receiving several of these messages. On the other hand, an *end-of-log message* marks the end of the log transmission, which forces to exit the loop. The behaviour of the Node Logger function is similar to the Hub Logger, the only difference is the set of NCC messages it can process. Specifically, the Node Logger function can deal with a set of messages called `node-error-counters message`, `node-stored-frame message` and *end-of-log message*. The former contains the value of both error counters, TEC and REC, see 2.2.6. The other two, analogously with the Hub Logger, contain one frame and an end of log mark, respectively. However, the information contained in the `node-stored-frame message` is less specific when compared with the `hub-stored-frame message`. Finally, the traffic and internal state of the node is reconstructed by receiving several `node-error-counters messages` and `node-stored-frame messages`.

```
print_hub_report(HUB_LOG_ID);

print_node_report(NODE0_LOG_ID);
print_node_report(NODE1_LOG_ID);
print_node_report(NODE2_LOG_ID);

function print_hub_report(hub_log_id) is
  log_request_msg_content = generate_ncc_log_request_msg_content();
  send_ncc_msg(hub_log_id, log_request_msg_content);

  log_active = true;
  while log_active loop
    received_msg = receive_ncc_msg();
    case received_msg.content.type is
      when log_hub_stored_frame_msg =>
        print_hub_stored_frame(received_msg.content);
      when log_end_of_log_msg =>
        log_active = false;
    end case;
  end loop;
end print_hub_report;

function print_node_report(node_log_id) is
  log_request_msg_content = generate_ncc_log_request_msg_content();
  send_ncc_msg(node_log_id, log_request_msg_content);

  log_active = true;
  while log_active loop
    received_msg = receive_ncc_msg();
    case received_msg.content.type is
      when log_node_error_counters_msg =>
        print_node_error_counters(received_msg.content);
      when log_node_stored_frame_msg =>
        print_node_stored_frame(received_msg.content);
      when log_end_of_log_msg =>
        log_active = false;
    end if;
  end loop;
end print_node_report;
```

Listing 13.2: Fault-injection log retriever pseudocode.

13.2. NCC design basics

Each NCC is designed dividing its operation in three different parts, called *Interface*, *Manager* and *Executer*. The former allows the communication of the other parts with the FIMS, through the CAN network, that is, receives/transmits NCC messages from/to the FIMS. The second, controls the behaviour of the NCC by monitoring and processing the NCC messages, provided by the Interface receiver. Finally, the executer performs the specific operation of the NCC, for instance, the fault injector generates the errors.

Although this is the main layout followed to design the NCCs, is possible to provide additional modules, out of these parts. For instance, inside the fault injector exists one specific module responsible for store the set of fault-injection configurations.

13.3. Centralized Fault Injector (CFI) design

The Centralized Fault Injector (CFI) is responsible for injecting errors from data provided by both, the fault-injection specification and the traffic. As explained in Sec. 11.3, this module outputs a set of signals, labelled ues_i, uev_i, des_i and dev_i, which feed a set a multiplexers, in order to force a specific vale in the uplinks and downlinks. The internal architecture of the CFI, shown in Fig. 13.1, is divided in various parts called *Interface*, *Manager*, *Configurations storage* and *Executer*. Next, each of these parts is described in-depth.

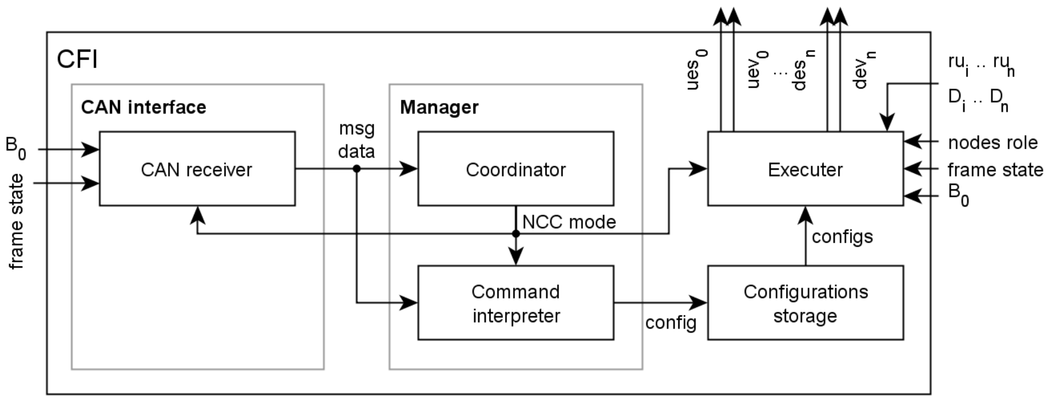


Figure 13.1.: CFI diagram.

13.3.1. Interface

The *Interface* implements the communication between the FIMS and the CFI. Since the CFI only needs to receive data, it includes just one module, called *CAN receiver*. In order to obtain each message sent from the FIMS, this module uses the coupled signal B_0 and the frame state signal `frame state`, both already defined in Sec. 11.3. Then, if appropriate, it passes the content of the message to the other modules through the signal `msg data`. In principle, the CAN receiver accepts as valid both, broadcast messages and those that are specifically addressed to the CFI. However, during the execution phase, the CAN receiver only accepts broadcast messages. This is because unicast addressing is not allowed during this phase, see Sec. 9.1. For this purpose, the current mode of the CFI, contained in signal `NCC mode` and next explained, must be provided.

13.3.2. Manager

The *Manager* controls the behavior of the CFI as a whole. It includes two different modules, the *Coordinator* and the *Command interpreter*. The former keeps the operating mode of the CFI and makes it available to the other modules through the `NCC mode` signal. Note that the Coordinator monitors the `msg data` signal in order to detect and then execute mode change commands. The second module builds up, one by one, the fault-injection configurations that constitute a given fault-injection experiment. For that purpose it interprets the configuration messages provided by the CAN receiver through the `msg data` signal. The module outputs each fault-injection configuration it builds up, by means of the `config` signal.

13.3.3. Configurations storage

The *Configurations storage* module is responsible for keeping the set of fault-injection configurations and making them available to the other modules, by means of the `configs` signal. The *Command interpreter*, through the `config` signal, provides one by one this set.

13.3.4. Executer

The *Executer* module is responsible for checking the fulfillment of the conditions involved in the fault-injection configurations and, if so, set up the values of the CFI outputs, in order to force the specific injected values. To do so, it is necessary to provide some information, that is, the set of fault-injection configurations, traffic data and the operating mode of the CFI. The former can be obtained from the Configurations storage module, by means of the `configs` signal. As concerns the traffic data, it can be obtained from the Hub core itself. Specifically, this data includes the injected incoming and outgoing signals of the nodes, $\mathtt{ru}_i..\mathtt{ru}_n$ and $\mathtt{D}_i..\mathtt{D}_n$, the role of the nodes, `nodes role`, the current bit being transmitted, `frame state`, and the coupled signal, $\mathtt{B}_0$. Finally, the current mode of the CFI, provided by the Coordinator and carried by the `NCC mode` signal, is used to enable and disable the Executer operation, that is, it remains inactive when the mode of the CFI is different from the execution mode. The internal of the executer is described next in detail.

As can be seen in the simplified Fig. 13.2 the operation of the Executer is performed sequentially by three different parts called *Config executer array*, *Value selector array* and *Value generator array*. The former returns the values that have to be injected in the links, only if the conditions specified in the fault-injection configurations are met. The second selects the value of the highest priority when more than one injection has to be performed in a link. Finally, the third generates the signals that enable the injection, as well as the value injected. Each of this parts and their interconnections are next described in detail.

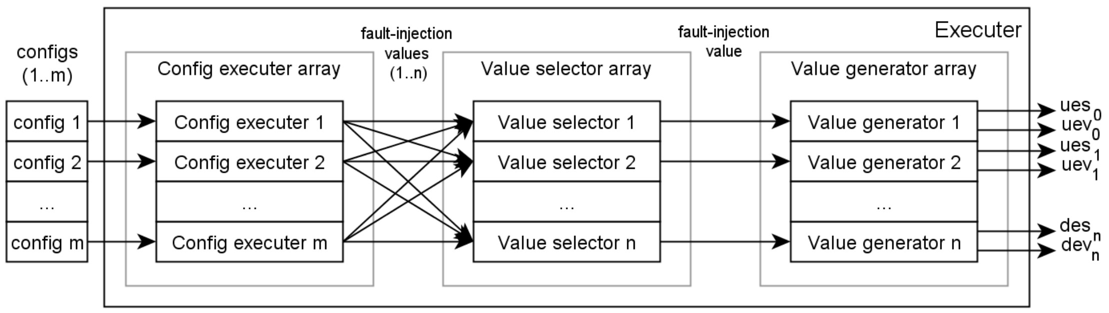


Figure 13.2.: Executer diagram.

Config executer array

The Config executer array decides, from the set of fault-injection configurations, which value has to be injected in the links, bit by bit. To do so, it contains a set of *Config executers*, each of which is connected to a single stored fault-injection configuration. The purpose of a Config executer is to check the fulfillment of the conditions defined in the fault-injection configuration it is connected and, if so, inform, by means of the set of the `fault-injection values` signals, which value has to be injected in the uplinks and downlinks. Note that, since each Config executer outputs a value for each of these link, the Config executer array generates several values for a single link. The internal architecture of a Config executer, shown in Fig. 13.3, is next described.

A given Config executer, as introduced above, is attached to a single fault-injection configuration, from which it extracts the information to both, validate the conditions involved and return the values that have to be injected. Each fault-injection configuration is divided in several fields, which correspond to the set of fault-injection parameters described in Sec. 10. The operation of a given Config executer can be performed thanks to a set of submodules called *Aim counter*, *Withdraw counter*, *Target detector*, *Checker*, *Offset delayer*, *End bit counter* and *Manager*. Note, from the figure, that some of the input signals introduced previously have been omitted to focus on the specific operation of each submodule. First, the Aim counter is used to track the number of times an aim frame has been seen. Signal `aim count` carries the specific internal value of this counter. Similarly to the Aim counter, the Withdraw counter tracks the withdraw frames, providing its count through the `withdraw count` signal. The Target-frame detector tracks target frames, but, in contrast to both modules seen previously, it does not act as a counter, instead it informs, by means of the `target-frame detected` signal, when a target frame has been seen. The Checker module is the core of the Config executer since it checks the fulfillment of the aim, withdraw, target-frame, fire and cease conditions. When so, it activates the output signal `inject` which enables the Offset delayer. This module enables the fire offset feature, that is, it acts as a bit counter which delays the `inject` signal. As concerns the End bit counter, it helps to perform the bit count cease condition, by counting the number of bits being injected. The value of the counter is carried by the `injected count signal`. Finally, the Manager generates the set of signals which contains the values that have to be injected according to the current fault-injection configuration. Note that, exists one output per link and, when no error has to be injected a null value is output. To perform its operation, the Manager needs the value that has to be injected, the target link and a injection enabler. The first two values are obtained from the *value* and *target* fields of the fault-injection configuration, respectively. The injection enabler is provided by the Offset delayer through the `delayed injection` signal.

Value selector array

The Value selector array selects a single value for each link. For this purpose, it contains a set of *Value selectors*, each of which corresponds to an specific link. A given Value selector receives a single value from all the Config executers and bypasses the one of the highest priority. The priority criteria used is based in the order of the definition of the fault-injection configurations, that is, as if they were rules to be applied, those fault-injection configurations defined later takes

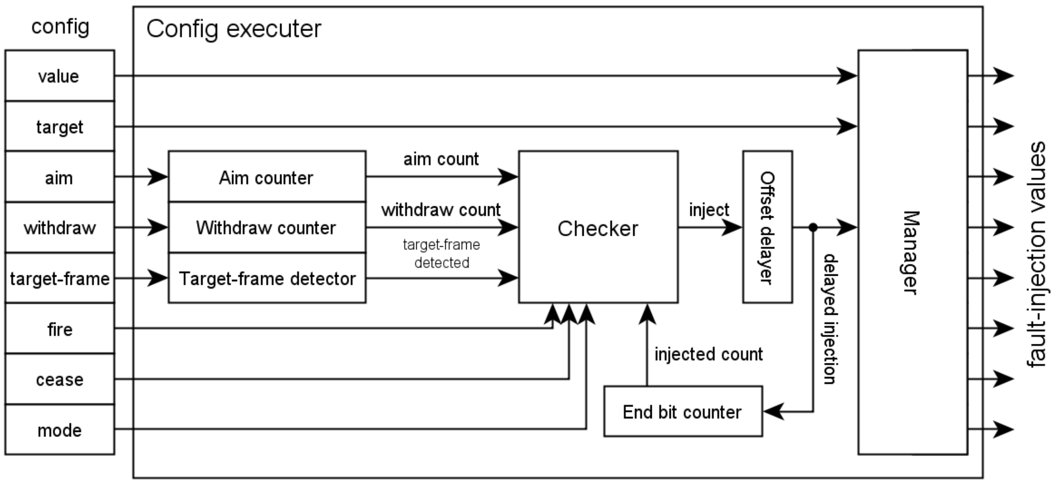


Figure 13.3.: Config executer diagram.

precedence. For instance, if fault-injection configurations N and N-1 decide, at the same time, that a value has to be injected in link M, Value selector M would bypass the value of Config executer N.

Value generator array

The Value generator array sets up the value of the outputs of the Executer, which in turn sets up the values of the CFI outputs, that is, the signals that carry the value to be injected and the ones used to force to inject these values. Specifically, similarly to the Value selector array, it contains a set of submodules called *Value generators*, each of which is dedicated to one link. A given Value generator is connected to its analogous Value selector and transforms the `fault-injection value` it provides to a set of two signals. The first contains the value that has to be injected itself, that is a recessive or a dominant, whereas the second contains an injection enabler, that is, it activates when the value that has to be injected is not null. Note that this module is useful when generating bit stream injection patterns, since it encapsulates the stream management.

13.4. Hub Logger (HL) design

The *Hub Logger* allows to monitor and collect relevant data, during an experiment, to be later transmitted to the FIMS. The organization and main modules are shown in figure 13.4. It is divided in four parts called *Interface*, *Manager*, *Gatherer* and *Reporter*. The first performs the communication with the FIMS. The second interprets and executes the FIMS messages. The third monitors and stores the data. Finally, the fourth translates the collected data into a report, which is transmitted to the FIMS.

Next, the behaviour of each part of the logger is described.

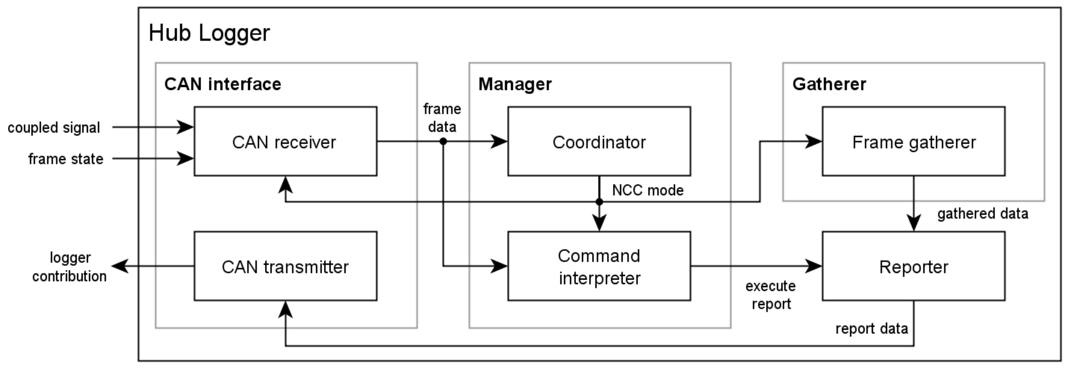


Figure 13.4.: Hub Logger diagram.

13.4.1. Interface

This group of modules abstracts the rest of the modules of the logger of accessing directly to the hub's communication subsystem, in order to receive/transmit NCC messages from/to the Fault Injection Management Station. Specifically, it includes two modules, called *CAN receiver* and *CAN transmitter*. The former is responsible for interpreting the coupled signal, filter the NCC messages destined to the logger and make them available. Note that unicast NCC messages cannot be transmitted during the execution of the test and, thus, this module must take into account the mode in which the logger is operating, in order to filter those messages. The second is responsible for transmitting the logged data, when necessary.

13.4.2. Manager

This group of modules interprets and executes commands from the *CAN receiver* module, by means of two different modules. The first module, called *Coordinator*, interprets mode change commands and keeps the current mode of the logger, in order to make it available for other logger modules. The second module, called *Command interpreter*, interprets and executes report requests, during the configuration mode.

13.4.3. Gatherer

This group of modules are responsible for monitoring and storing relevant data, during the execution of a test, and make it available in order to construct a report. Since the default goal of sfiCAN is testing regular CAN networks, the only existing module contained is the *Frame gatherer*. However, if sfiCAN is used to test new advanced fault-tolerant CAN mechanisms, other modules can be appended, in order to extend the collecting capabilities of the logger.

Frame gatherer module stores the traffic seen in the coupled signal. Moreover, due to the privileged position of the hub, it is possible to known the source of each frame and, if an error occurs, in which point of the frame it took place.

13.4.4. Reporter

This module is responsible for generating a *report* from the data stored in the *Gatherer* module. Specifically, the report is divided into frames, which are sent to the Fault Injection Management Station by means of the *CAN transmitter* module. This action is performed during the configuration mode of sfiCAN, when the user makes a report request by means of the *Command interpreter*.

13.5. Node Logger (NL) design

The Node Logger is a software running within the nodes, that allows to monitor, register and report specific events. Specifically, we distinguish two types of events: *application events* and *driver events*. On the one hand, the first type involves those events related to the operation of the application being executed by the node. For instance, the modification of the value of a particular variable. On the other hand, driver events involves all those events related to the operation of the driver. In this sense we monitor when a message is transmitted/received and any change in the value of the errors counters. The pseudocode of the internal design of this logger is shown in List. 13.3.

Note that there are two main variables called `op_mode` and `events`. On the one hand, the first contains the operation mode of the Node Logger. Thus, this software behaves differently depending on its value. On the other hand, variable `events` keeps the set of monitored events. The code is divided in four parts, one for each operation mode. However, note that most of the code is devoted to deal with the mode change messages, that is, implement the idea presented in Fig. 9.2. For instance, when the Node Logger is in wait-for-whistle mode, it waits for a NCC message to be received. Note that the `receive_ncc_msg()` function not only captures NCC messages, but discards those not addressed to the Node Logger. When so, its type is evaluated in order to only process enter-config and starting-whistle NCC messages. If the NCC message received belongs to any of these types, variable `op_mode` is updated accordingly to the possible transitions specified in Fig. 9.2.

As concerns the code executed when the Node Logger enters in execution mode, it implements a loop which activates each time a new event is detected. Note that the `new_event()` updates automatically the value of temporal variable `event`. Once a new event is captured the software determines its source, that is, whether is an application or a driver event. On the one hand, when an application event is detected, it is just stored in the `events` variable. On the other hand, when a driver event is detected, it is evaluated to check if it is a NCC messages destined to the Node Logger. If not, the events is just stored inside the `events` list. In contrast, when a NCC message destined to the Node Logger is detected, it is processed in order to execute enter-config-mode NCC messages. Note that, the `break` statement ensures that the thread of execution exits from this part of the code, and forces a new iteration of the loop.

Finally, when the Node Logger is in configuration mode, it can handle log requests. When so, it uses the set of events stored in variable `events` to generate and send the set of NCC messages conforming the log report.

```
op_mode = configuration;
events = null;

case op_mode is
  when configuration =>
    ncc_msg = receive_ncc_msg();
    case ncc_msg.type is
      when enter-idle-mode => events = null; op_mode = idle;
      when enter-wfw-mode  => events = null; op_mode = wait-for-whistle;
      when log-request     => send_report(events);
      when others          => null;
    end if;

  when idle =>
    ncc_msg = receive_ncc_msg();
    case ncc_msg.type is
      when enter-config-mode => op_mode = configuration;
      when others            => null;
    end if;

  when wait-for-whistle =>
    ncc_msg = receive_ncc_msg();
    case ncc_msg.type is
      when enter-config-mode => op_mode = configuration;
      when starting-whistle  => op_mode = execution;
      when others            => null;
    end if;

  when execution =>
    while not new_event() loop
      case event.from is
        when application_event =>
          add_event(events, event);

        when driver_event =>
          if event.type = rx_frame and is_ncc_msg(event.frame) then
            ncc_msg = extract_ncc_msg(event.frame);
            if ncc_msg.type = enter-config-mode then
              op_mode = configuration;
              break;
            end if;

          else
            add_event(events, event);
          end if;
      end case;
    end loop;
end case;
```

Listing 13.3: Node Logger pseudocode.

13.6. NCC protocol design

As introduced in Sec. 9.2 the communication between the FIMS and the NCCs is carried out using a network protocol on top of CAN, called NCC protocol, which is based in the exchange of NCC messages. Each NCC message is divided in two parts. On the one hand, the first part contains the so-called NCC ID, which is used to identify the specific NCC that must accept the message. On the other hand, the second part carried the content of the message itself. Note that a given NCC message is encapsulated within a CAN frame. That is, the destination is carried in the identifier field, whereas the content is carried in the data field. Next, these two parts are described more in-depth.

13.6.1. NCC message addressing strategy

The addressing strategy used in the NCC protocol is based in the use of NCC IDs. A given NCC ID is used to identify unequivocally a specific NCC. In fact, in order to keep the consistency of the protocol, the FIMS has its own NCC ID, so the loggers can send the NCC messages conforming the report. The representation of a given NCC ID is the same used in a regular CAN identifier, that is, it is defined as a 11-bit string. Additionally, we have enabled the NCC ID of the highest priority to be used as a broadcast address, so a given NCC message can be delivered to all the NCCs at the same time. The set of existing NCC IDs is shown in Table 13.1.

NCC	**NCC ID**
Broadcast	000
CFI	001
HL	002
NL0	003
NL1	004
NL2	005
FIMS	010

Table 13.1.: List of NCC IDs.

Finally, note that, in order to minimize the impact of the NCC protocol during the execution phase, unicast addressing is not permitted during this phase. Thus, the enter-config-mode message, used to transit from execution mode to configuration mode, has to be transmitted globally, that is, using the broadcast address. This decision allows nodes to use all the range of CAN identifiers, except the broadcast NCC ID, during the execution of an experiment. Moreover, it ensures that all the nodes end their operation quasi simultaneously.

13.6.2. NCC message content

The content of the message carries a specific command that has to be executed. In this sense, as explained in Sec. 9.2, there are three types of messages, depending on the operation to be executed, configuration, mode change and logging messages. As concerns its size, similarly to the NCC ID, it is limited by the size of the data field, that is, 8 bytes. As can be seen in Fig. 13.5, this space is divided in two parts called *message code* and *message parameter*. On the one hand, the message code identifies the specific message. It takes 1 byte, whose 3 first bits identify the type of the message. On the other hand, the message parameter is used to carry a specific value related with the message. In this case, it can take from 0 to 7 bytes, depending on the message. For instance, message *configure aim count* must be delivered with a value indicating the number of times the aim condition has to be fulfilled. Table 13.2 contain all the messages conforming the NCC protocol, the codification of the message code and the size of the message parameter. The possible values of the message parameters, as well as their codification, are not described here, instead, they can be consulted in the code of the Fault Injection Management Station, see App. F. Next, we overview how these messages are used.

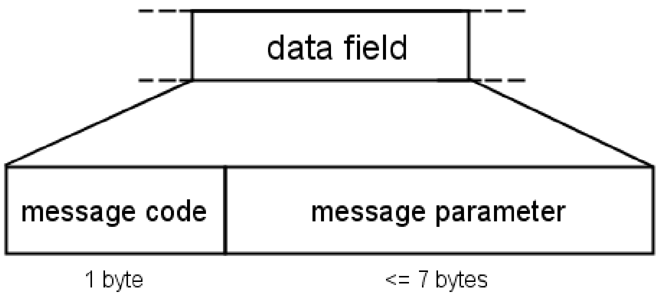


Figure 13.5.: NCC message construction.

NCC configuration messages design

This type of messages is used by the Fault-Injection Configurator (FIC) to configure the CFI. As explained in Sec. 13.1.2, the FIC encodes the fault-injection parameters conforming a fault-injection configuration into various of those messages. In fact, there is a specific configuration message for each fault-injection parameter, except for the aim, withdraw and target-frame filters, which needs more than one. Specifically, this parameter needs three, one to carry the value of the filter, one to carry the mask to be applied and the size of the filter. In this sense, note that this decision allows to use filters with don't care values of a non-specific size. Finally, an specific NCC message called *end of configuration* can be used to instruct the CFI that all the parameters of the current fault-injection configuration are set and, thus, the subsequent configuration messages belong to a new fault-injection configuration.

NCC mode change messages design

This type of messages is used by the Mode Changer (MC) to force the modification of the current operation mode of a given NCC. Note that, since no information has to be transmitted to the NCCs, these messages do not include any value in the mesage parameter space. In contrast, exist four different messages, one for each possible operation mode.

NCC logging messages design

This type of messages are used by the Node Loggers (NLs) and Hub Logger (HL) to transmit the log report to the Fault-Injection Log retriever (FIL). It exists an specific logging message for each information gathered. On the one hand, in the case of the hub, each *report hub stored frame* message carries a specific collected frame. On the other hand, in the case of the nodes, each *report node error counters* and *report node stored frame* messages carry an specific error counter modification and collected frame, respectively. Note that these messages are chronologically transmitted, so the FIC can show this data as ordered events. Moreover, the *end of report* message is used to inform the FIL that the current logger has no more events to transmit and, thus, the current report is over. Finally, note that in messages carrying log frames the data has a complex structure. This structure can be consulted from the code of the FIL, see App. F.3.

Message description	**Message code**	**Message parameter**
configure value type	`000.00000 (00)`	value type: 2 bits
configure value pattern	`000.00001 (01)`	pattern: 7 bytes
configure target link	`000.00010 (02)`	link: 4 bits
configure mode	`000.00011 (03)`	mode: 2 bits
configure aim filter	`000.00100 (04)`	filter value: 56 bit string
configure aim mask	`000.00101 (05)`	filter mask: 56 bit string
configure aim mask long	`000.00110 (06)`	filter long: 6 bits uint
configure aim field	`000.00111 (07)`	field: 4 bits
configure aim link	`000.01000 (08)`	link: 4 bits
configure aim role	`000.01001 (09)`	role: 2 bits
configure aim count	`000.01010 (0A)`	count: 2 byte uint
configure withdraw filter	`000.01011 (0B)`	filter value: 56 bit string
configure withdraw mask	`000.01100 (0C)`	filter mask: 56 bit string
configure withdraw mask long	`000.01101 (0D)`	filter long: 6 bits uint
configure withdraw field	`000.01110 (0E)`	field: 4 bits
configure withdraw link	`000.01111 (0F)`	link: 4 bits
configure withdraw role	`000.10000 (10)`	role: 2 bits
configure withdraw count	`000.10001 (11)`	count: 2 byte uint
configure target frame filter	`000.10010 (12)`	filter value: 56 bit string
configure target frame mask	`000.10011 (13)`	filter mask: 56 bit string
configure target frame mask long	`000.10100 (14)`	filter long: 6 bits uint
configure target frame field	`000.10101 (15)`	field: 4 bits
configure target frame link	`000.10110 (16)`	link: 4 bits
configure target frame role	`000.10111 (17)`	role: 2 bits
configure fire field	`000.11000 (18)`	field: 4 bits
configure fire bit	`000.11001 (19)`	bit position: 6 bits
configure fire offset	`000.11010 (1A)`	offset: 2 byte uint
configure cease field	`000.11011 (1B)`	field: 4 bits
configure cease bit	`000.11100 (1C)`	bit position: 6 bits uint
configure cease bit count	`000.11101 (1D)`	bit count: 2 byte uint
end of configuration	`000.11111 (1F)`	—
enter config mode	`001.00000 (20)`	—
enter idle mode	`001.00000 (21)`	—
enter wfw mode	`001.00000 (22)`	—
enter exec mode	`001.00000 (23)`	—
report hub stored frame	`010.00001 (41)`	hub stored frame: 5 bytes
report node error counters	`010.00000 (40)`	TEC/REC values: 2 bytes
report node stored frame	`010.00001 (41)`	node stored frame: 3 bytes
end of report	`010.11111 (5F)`	—

Table 13.2.: List of messages of the NCC protocol. For each message it is specified its description, its codification and the data type of its parameter.

Part IV.

Implementation

14. Implementation environment/platform

The implementation of sfiCAN has been a multidisciplinary work and, thus, it required the use of a set of different types of tools, both software and hardware. Next we describe the specific development environments and/or platforms used in this project.

First, we specified the internal logic of the hub by using a hardware-description language called *Very-High-Speed-Integrated-Circuit Hardware-Description Language* (VHDL), a widely-used high-level language used to describe complex digital systems. Note that next section (Sec. 14.1) provides more details about this language. Then, we synthesized the this logic within a *Field-Programmable-Gate-Array* (FPGA). That is [Wikipedia, 2012a], an integrated circuit designed to be configured by the user to behave in a specific way. A given FPGA is composed of various *logic blocks*, each of which can be configured to perform complex combinational or sequential functions including, for instance, logic gates and flip-flops. The specific hardware used to hold the hardware description is the one used in the ReCANcentrate prototype [Barranco, 2010], that is, a Xilinx XSA-3S1000 FPGA board [XESS Corporation, 2005] which contains a Xilinx Spartan-3 XC3S1000 FPGA chip. This board has various benefits that makes it suitable for the project, for instance, it has good price-quality ratio, it provides 65 general-purpose I/O pins and it can be programmed easily by means of a parallel port. Since we used a FPGA from Xilinx we built up the hub's internal logic by means of the Xilinx ISE (Integrated Software Environment) Design Suite [Xilinx, 2012]. This software tool allows the user to specify, simulate, synthesize and install hardware descriptions.

Second, the development of the sfiCAN's nodes was carried out from the nodes of the ReCANcentrate prototype Gessner [2010]. Specifically, for a given node we use a dsPICDEM 80-pin Starter Development Board [Microchip Technology, 2006], whose central component is a dsPIC30F6014A microcontroller [Microchip Technology, 2011]. Moreover, as explained we implement the nodes' software, that is, the applications and Node Loggers, which is done using the C programming language. In this sense, since this software executes in built-in nodes from Microchip, we used a specific open source Linux-based development environment for Microchip microcontrollers. More specifically, we used the Piklab integrated development environment (IDE) [Piklab, 2012] together with the MPLAB C30 cross compiler [Microchip Technology]. This last tool is a full-featured ANSI compliant C compiler for Microchip 16-bit devices. Note that we did some debugging on the software developed. However, Piklab does not support this feature. Thus, we had to use the default Microchip's IDE, that is, MPLAB [Microchip Technology, 2009], which runs under Windows. This IDE contains various features that Piklab does not provide, however, it is more complex and does not works under Linux environments.

Finally, the FIMS was constructed as a software executed in a Linux-based PC. As explained in Sec. 18, the FIMS is composed of a set of utilities, each of which is implemented separately. On the one hand, the Fault-Injection Configurator (FIC) and the Fault-Injection Log retriever (FIL) were implemented as a C++ programs. For this, we used a regular text editor jointly with the GNU Compiler Collection (GCC) [GNU]. On the other hand, the Mode Changer (MC) was implemented by means of shell scripts using a simple text editor.

14.1. Very-High-Speed Integrated Circuits Hardware Description Language (VHDL)

The implementation of the logic of the hub, that is, the Centralized Fault Injector and the Hub Logger, has been carried out using a standardized hardware description language (HDL) called Very-High-Speed Integrated Circuits Hardware Description Language (VHDL). This is a widely-used high-level language used to describe complex digital systems. Next, we describe the main characteristics of this language, as well as its use.

14.1.1. History and development

The first digital circuits design methods were based in the construction of prototypes. These were made from easier modules with well tested gates. In the mid 70s a great revolution happened in the fabrication processes of integrated circuits thanks to the creation of bipolar technologies, mainly NMOS. At the end of 70s became evident the huge disparity between the technology and the design methods.

During the first half of the 80s the design was carried on manually and at a low level. Afterwards, simulation tools, like PSPICE, were used to verify the electrical schemes from the predefined models for each technology. Progressively, the technological processes became more complex and difficult to integrate, debug and maintain. The use of low-level design methods made impossible for companies to assume the risks and costs.

Due to the complexity of the methods used thus far, various investigation groups start to develop the *hardware description languages*. IDL from IBM, TI-HDL from Texas Instruments and ZEUS from General Electric were pioneers in this sector. However, these languages did not get the expected success nor consolidation, and were not standardized.

The development of VHDL began early in the 80s by the Department of Defense (DoD) of the USA within the VHSIC (Very High Speed Integrated Circuit) project. This research had two main goals. On one side, the development of a high-level language able to solve the problems caused by the available tools, which forced to specify the circuits in a low-level manner. On the other side, the creation of a standard, which could used by the companies contracted by the DoD, with the purpose of constructing their designs.

The initial development (1983) was carried out by IBM, Texas Instruments and Intermetrics. The first version of the language was revised in public in the middle of 1985. After some reviews, performed with the collaboration of the industry and universities, in December of 1987, IEEE made public the first version of the standard VHDL, the IEEE 1076-1987 (VHDL'87). Afterwards, in 1993, an improved version was presented, the IEEE 1076-1993. However, due to some portability difficulties between design tools, related with the multi-evaluated logic, the main companies moved away from IEEE 1076 and developed their own solutions. For that reason, in order to prevent the fragmentation of the language, the standard library IEEE 1164 (standard logic) was developed.

Nowadays, VHDL is the most used hardware description language thanks to some of its characteristics: independence from the design methodology and the fabrication process, descriptive capacity in multiple domains and abstraction levels, versatility in the fabrication of complex systems, reutilization, and, to sum up, its standardization.

14.1.2. Purposes

VHDL has been introduced as an effective and clear standardized language used to specify complex hardware designs. However, thanks to the its high adoption, VHDL is not limited to the hardware description, and can be used for two purposes:

- Modeling and simulation: VHDL can be used to make a behaviour hardware description in a very abstract level, close to general-purpose languages like ADA, PASCAL or C. Moreover, it is possible to perform simulations from a working model of a circuit, which allow to verify both the behaviour and performance of the system.
- Hardware device synthesis: VHDL is also used to perform a physical description of a circuit, using VHDL hardware synthesis tools. We understand *synthesis* as the set of design compiling processes that allow the creation of the system's physical description. This translation may result in different types of hardware descriptions, that is, transistor level, logic gates or configuration files to program a PLD device. However, the synthesis cannot make profit from the whole VHDL descriptive level. It is a major challenge, or either impossible, the translation of some data types and statements into a physical description. These restrictions depend on the compiler and/or the synthesis tool used.

14.1.3. Structure of a hardware module in VHDL

In VHDL the description of a given hardware module is divided in two parts called *entity* and *architecture*. List. 14.1 shows the syntax used to define a module. On the one hand, the entity defines the interface, that is, the set of signals conforming the inputs and outputs. For each signal it is necessary to specify the name, whether it is an input or an output and its type. On the other hand, the architecture defines how the functionality is implemented. A given architecture is divided in two parts, the *declarative part*, which contains the entities, signals, constants, etc. used, and the *statement part*, which contains the set of statements generating the outputs of the module. Specifically, all these statements are executed concurrently, just like parts of a real circuit.

Note that the division between the entity and the architecture allows to see the hardware module as a black box, in which the entity provides the interface while the architecture hides the specific implementation details.

```
entity component_name is
port(
  <input/output declarations>
);
end component_name;

architecture architecture_name of component_name is
  <declarative part>
begin
  <statement part>
end architecture_name;
```

Listing 14.1: VHDL module example.

14.1.4. Design

VHDL offers more than one design paradigm in order to help the designer to construct the description of a given hardware module. Specifically, it allows to specify a behaviour of the circuit in three different ways. First, in terms of combinational and/or sequential logic, just like that a real circuit. Second, in terms of sequential high-level instructions, just like a high-level language. Finally, in terms of a set of components working concurrently and cooperatively. These three ways of describing a hardware behaviour can be used separately or together, the designer must evaluate the design and choose the best in each case. Next, these paradigms are deeply discussed.

Data flow design

In this paradigm circuits are described by defining the outputs in terms of the inputs. The behaviour is described by means of a set of signal assignments that are executed concurrently. These assignments can be conditional or not, that is, the value of a signal can depend on the evaluation of a condition. Moreover, it is allowed to use built-in primitives in the assignments, for instance, logic gates or shift registering. Finally, it is also possible to hold back the assignment of a signal by specifying a particular delay.

This paradigm is possible thanks to the assignment statement. In VHDL the value of a given signal can be modified by means of the <= operator. Specifically, as can be seen in the first example of List. 14.2, the destination signal is placed at the left of this operator, whereas the value to be used is placed at its right. Note that, as introduced in the previous paragraph, this value can be generated using built-in primitives. The specific set of primitives that can be used depend on the type of the values involved.

The value to be used in an assignment statement can also be selected depending on the fulfilment of a given condition. Specifically, as can be seen in the second example of List. 14.2, it can specified by means of a code structure similar to the if-elsif-else code structure used in imperative programming languages. Moreover, as can be seen in the third example, this structure can be specified in terms of the value of a given signal, just like a case statement in traditional imperative programming languages.

```
signal <= value;

signal <= value1 when cond1 else
          value2 when cond2 else
          ...
          valueN when condN else
          valueM;

with signal1 select
  signal2 <= value1 when signal1_value1
             value2 when signal1_value2
             ...
             valueN when signal1_valueN
             valueM when others;
```

Listing 14.2: Assignment syntax.

Procedural design

This paradigm allows to specify the behaviour of a hardware module just like a regular imperative programming languages, that is, by means of a list of high-level statements executed sequentially that change the internal state of the module. This is possible thanks to the *process* statement. This statement is executed concurrently jointly with other declared statements, just like signal assignments in data flow designs. However, the statements contained in the process are executed sequentially, like a regular high-level programming language. Sequential statements are often more powerful and intuitive than concurrent ones, but sometimes there is not a direct correspondence to a hardware implementation.

A process statement is similar to a procedure in a high-level programming language, as can be seen in List. 14.3. It is divided in two parts, the *declarative part* and the *statement part*. On the one hand, the declarative part contains the set of data objects and data types required to perform the process operation. On the other hand, the statement part holds the set of statements to be executed. Note that the execution of a process statement is conditioned to the occurrence of a given event involving the *sensitivity list*. Specifically, if any the value of any of the signals contained in the sensitivity list changes, the process is re-evaluated. That is, the content of the statement part is executed, which forces the updating of the outputs of the module.

```
process [(<sensitivity list>)] [is]
  <declarative part>
begin
  <statement part>
end process;
```

Listing 14.3: Process statement syntax.

The statement part can contain various types of sequential statements. First, it is possible to use signal assignments containing built-in primitives. This primitives depend on the values used in the assignment, they can be logical, arithmetic, shift or relational. Second, it is also possible to use conditional and iterative control statements, that is, if-elsif-else, case, while and for statements. Finally, specific logic can be encapsulated in procedures and the used inside a process. These procedures are identical to the ones used in high-level imperative programming languages.

Structural design

This paradigm uses the top-down design concept to improve the construction process of the design. The behaviour of the system is described by means of the collaboration of a set of components. Each component describes a smaller system, in fact, it is analogous to an off-the-shelf part. Thus, a hierarchical structure is generated, where components are connected together, by means of logical wires, to form the whole design. This type of design has the same advantages that top-down design, in regular software development, except for the fact that in VHDL all the components are inter-connected and their execution is concurrent. In this context, maintainability and readability are improved, thanks to encapsulation. Moreover, the possibility of reusing modules, among different projects, prevents the re-encoding of common behaviours.

List. 14.4. shows how to instantiate a component to be used in a given architecture. On the one hand, it is necessary to specify the interface of the component, that is, its inputs and outputs. This is done in the declarative part of the architecture. On the other hand, it is necessary to make an instance of the component. For this purpose, a port map must be provided. This port map contains a set of signal assignments that associate local architecture signals with the inputs and outputs of the component. Note that, multiple instances of a given component can be performed in a specific architecture.

```
architecture architecture_name of component_name is
  component subcomponent_name is
  port(
    <input/output declarations>
  );
  end component;
begin
subcomponent_label :
  subcomponent_name
    port map(
      <input/output assignments>
    );
end architecture_name;
```

Listing 14.4: Component instantiation statement syntax.

Additionally, it is possible to instantiate various components at the same time thanks to the *generation* statement. As can be seen in List. 14.5, this statement allows to use an iterative logic over an instantiation statement. Consequently, a component array can be constructed easily. Note that, although in this project the generation statement is only used to instantiate several components at the same time, it is possible to be used with any concurrent statement. Moreover, instead an iterative logic, a conditional logic can be used.

```
multiple_subcomponent_label :
  for identifier in discrete_range generate
    <subcomponent instatiation>
  end generate;
```

Listing 14.5: Multiple component instantiation statement syntax.

14.1.5. Data objects

VHDL provides different elements to keep the values conforming the state of the entities. Specifically, they are called *constant*, *variable* and *signal*. Next, these three elements are described.

Constant

This type of data object stores a non-modifiable value of a given type. Note that, since its value cannot be modified it must be set up when declaring the object.

```
constant DOMINANT : std_logic := '0';
```

Variable

This type of data object holds a value of a given type but, in contrast with constants, this value can be updated using a assignment statement. A variable can only be declared inside a process statement, that is, a behavioural description of a sequential statements. Moreover, they behave like regular variables in high-level languages, that is, the update of its value occurs just after the assignment statement.

```
variable count : integer;
```

Signal

This type of data object, just as variables, holds a modifiable value of a given type. However, since signals behave like real circuit wires, they have some important differences with variables. On one hand, its declaration is not limited to process statements, signals can be declared in any declarative statement. On the other hand, the value assignment is done concurrently. Moreover, within sequential statements signals' value is updated at the end of it.

```
signal reset : std_logic;
```

14.1.6. Data types

Each data object has a data type associated. The data type defines the set of values that a data object can keep and the set of operations that can be executed on it. Next, the most used types in VHDL are described.

Bit

The *bit* type represents the value of a regular bit. Specifically, it is limited to two logical values: '0' and '1'. The operations associated with it are the regular bitwise operations.

Bit vector

The *bit vector* type represents a sequence of bit elements, in which the size of the sequence must be defined in the declaration. It is possible to access to one or more consecutive bits of the sequence by indicating their index. As concerns the operations associated, they are the same that in the bit type, and additionally the bit shift.

Standard logic

The *standard logic* type was defined in the IEEE 1164 as a replace of the bit type, in order to increase its operational capacity, so it is compatible with multi-evaluated logic. In contrast to the bit type, the standard logic has a set of 5 possible values: '0', '1', 'Z' (high impedance), 'U' (unknown) and 'X' (don't care). The operations associated are the same that in the bit type.

Standard logic vector

The *standard logic vector* type represents a sequence of standard logic elements. Similarly to the bit vector type the size of the sequence must be defined in its declaration. Moreover, it is possible to access to one or more consecutive bits of the sequence by indicating their index. The operations associated are the same that in the bit vector type.

Integer

The *integer* type allows to work with positive and negative numbers. Specifically, the set of values supported goes from -2.147.483.647 to 2.147.483.647 and the operations associated are the arithmetic ones.

Boolean

The *boolean* type represents the regular boolean type. That is, it can hold two values: `true` or `false`. The operations associated are the boolean ones, that is, AND, OR and NOT.

Enumerated

An enumerated type is an user-defined type that is specified by means of the lists of possible values that it can hold. These values can be characters, literals or identifiers. This type is very useful when describing models at an abstract level. For instance, the example presented below shows how to define the `t_fiMode` type, that is, the type that represents the possible fault-injection modes of a given fault-injection configuration.

```
type t_fiMode is (singleShot, continuous, iterative);
```

Subtypes

An subtype is an user-defined type which is built from a limited range of the values of an existing type. For instance, the example presented below shows how to define a subtype of an enumerated type. Specifically, the `week_day` type specifies the days of the week, whereas the `weekend_day` restricts this set to only de days conforming the weekend.

```
type week_day is (monday, tuesday, wednesday, thursday,
  friday, saturday, sunday);
subtype weekend_day is week_day range saturday to sunday;
```

Array

An array is an user-defined type which is built from a set of values of an specific type. Each elements in this set is placed in a given position and can be accessed using a specific index. For instance, the example presented below shows two examples of how to define an array. On the one hand, `t_fiCfgs` constructs an array of `t_fiCfg` elements indexed by a natural value, which goes from `1` to `MAX_INJS`. On the other hand, `t_fiValues` constructs an array of `t_fiValues` elements indexed by an enumerated type, `t_fiLink`.

```
type t_fiCfgs is array (1 to MAX_INJS) of t_fiCfg;
type t_fiValues is array (t_fiLink) of t_fiValue;
```

Record

A *record* type is a user-defined type which is built from a set of elements of different types. Each of this elements can be accessed unequivocally since each one has an identifier associated. Note that, although a record type do not add any additional functionality, they help to organize the code, making it more maintainable. The example presented below shows how to define the `t_fiCfg` type from a set of eight different elements.

```
type t_fiCfg is record
  value : t_fiValue;
  link  : t_fiLink;
  mode  : t_fiMode;

  startTrigger : t_trigger;
  endTrigger   : t_trigger;
  selTrigger   : t_trigger;

  fiStart : t_fiStart;
  fiEnd   : t_fiEnd;
end record;
```

14.1.7. Package

The VHDL packages are containers used to group and encapsulate those declarations and definitions common to a set of modules. Specifically, they contain different conceptually interrelated objects like constants, data types, functions or procedures. As can be seen in List. 14.6, a given package is divided in two parts. On the one hand, the package header holds all the declarations. On the other hand, the package body contains the specific code describing its behaviour. Note that, most of the declarations do not need an entry in the body, only the implementation part of the functions and procedures is defined in this part of the package.

```
package package_name is
  <declarative part>
end package_name;

package body package_name is
  <body declarative part>
end package_name;
```

Listing 14.6: VHDL package syntax.

14.1.8. Library

VHDL provides a higher encapsulation level, known as library. This is, a repository in which compiled units are stored, making possible the code reutilization. A given unit can be one of theses elements: entity, architecture or package.

To use a given library it is first necessary to write the `libray` reserved word followed by the name of the library. Once the library has been included, a given package contained in it can be used by writing the `use` reserved word followed by the name of the library, the name of the package and the `all` attribute. As an example, List. 14.7 shows how to make visible all the components declared in the `std_logic_1164` and `numeric_std` packages, which are encapsulated inside the `ieee` library.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

Listing 14.7: VHDL library syntax.

15. Hub implementation

As previously explained, the sfiCAN hub was constructed from the previous ReCANcentrate hub prototype, see [Barranco, 2010]. In this sense, most of its hardware could be reused. In fact, only minor changes were done in order to adapt it to the sfiCAN infrastructure. Next, we discuss the main concepts of the sfiCAN hub, focusing on the modifications performed in the previous prototype.

The physical implementation of the sfiCAN's hub, shown in Fig. 15.1, is divided in two parts constructed separately using the wire-wrap technique: the *FPGA Board Module* and the *Input/Output Module*.

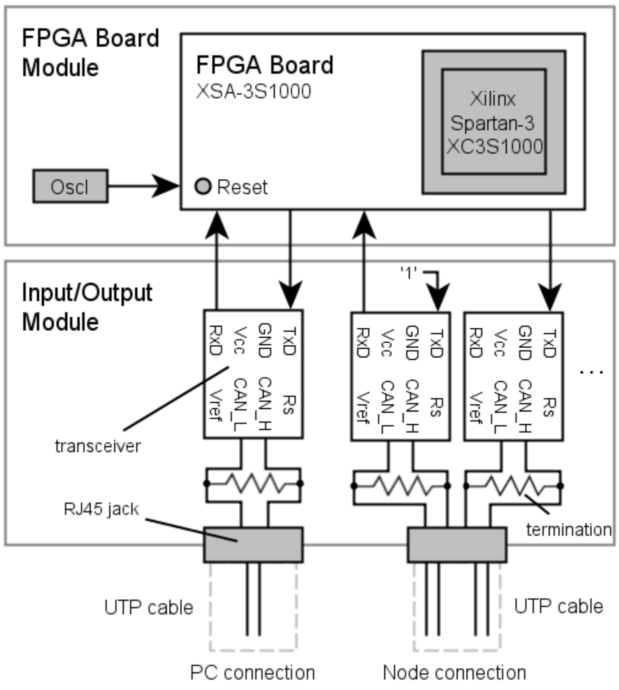


Figure 15.1.: Hub implementation diagram.

On the one hand, the FPGA Board Module implements the logic of the Hub Core. To do so, the Hub Core's description is synthesized into a FPGA chip, a Xilinx Spartan-3 XC3S1000, which stands inside a FPGA board, a Xilinx XSA-3S1000. Moreover, in order to achieve the 1 Mbps CAN bit-rate, it was necessary to attach an external 16 Mhz oscillator in this board.

On the other hand, the Input/Output Module translates the physical incoming signals from the nodes into logic values, and vice versa. Specifically, this part is constructed from a set of Philips PCA82C250 high-speed CAN transceivers [Philips, 1996] connected to a set of RJ45 jacks, following the design schema presented in Sec. 11. As can be seen in the figure, in a node, the two CAN transceivers used to translate the uplink and downlink, are attached to one RJ45 jack. Conversely in a PC the RJ45 jack is directly attached to a transceiver, since its connection only uses one. Note that this last port did not exist on the ReCANcentrate hub prototype and, thus, it was necessary to create it from a previous one.

Finally, note that, in order to communicate the hub with the nodes and the PC, the Input/Output Module uses a set of RJ45 ports. In this sense, each network component is connected to the hub using an UTP (Unshielded Twisted Pair) Category 5/5e/6 ethernet cables. In the case of a regular node port, the cable carries two pairs of CAN_H and CAN_L wires, one for the uplink and one for the downlink. In contrast, the cable used for the PC communication only carries one pair of CAN_H and CAN_L wires. The specific configuration of this last one cable can be found in Sec. 18.2.

16. Hub Core implementation

The Hub Core internal logic, depicted in Fig. 16.1, follows the design presented in Sec. 11. However, note that the figure has its own naming convention. Moreover, new signals and logic have been added to represent specific implementation issues. For instance, the system clock signal, that is `clk`, implements the time basis of the different automata or modules. All these details can be consulted in the source code of the Hub Core, which is presented as an appendix in App. D.

The Hub Core is composed of four different modules called `couplerModule`, `canModule`, `loggerModule` and `faultInjectionModule`, which correspond to the Coupler Module, Hub Sync Module, Hub Logger Module and the Centralized Fault Injector Module respectively. Next, the main operation of each one of these modules, as well as their interconnections, are introduced. Later on each module will be separately described in-depth.

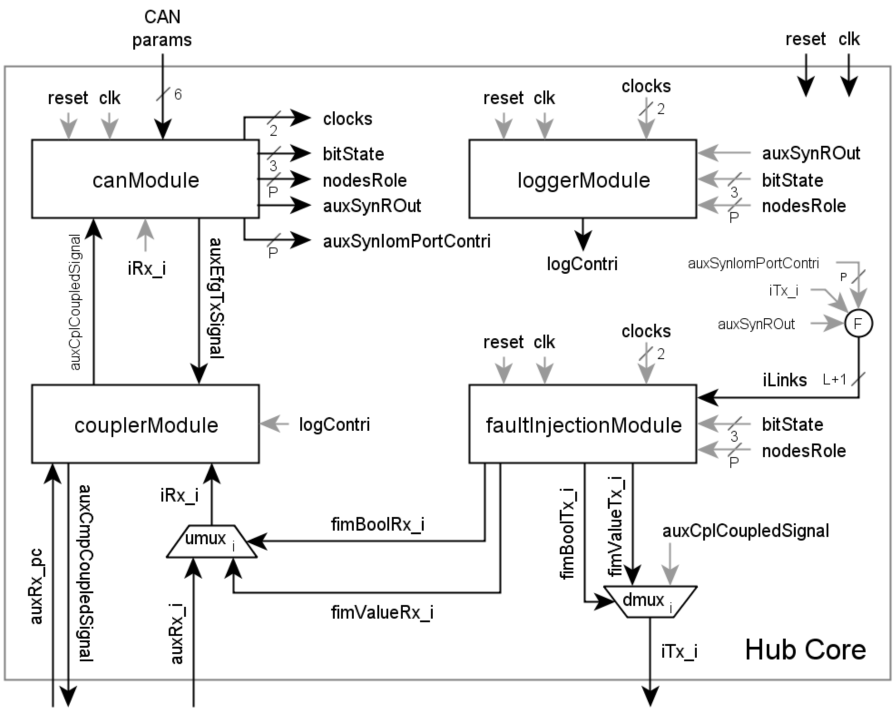


Figure 16.1.: Hub core implementation diagram.

The couplerModule is a combinational module that couples a set of contributions, in order to obtain the resultant coupled value, that is, the value that would be in a CAN bus due to its wired-and property, see Sec. 2.1. Specifically, it generates two signals called `auxCplCoupledSignal` and `auxCmpCoupledSignal`, which correspond to the B_0 and P_d signals, respectively. On the one hand, `auxCplCoupledSignal`, is the result of coupling the resultant uplink signals (`iRx_i`) and the contributions of the PC (`auxRx_pc`), the loggerModule (`logContri`) and the canModule (`auxEfgTxSignal`). This signal is used to feed the canModule module, as well as the downlinks of the nodes. On the oher hand, `auxCmpCoupledSignal`, as discussed in Sec. 11.4, does not include the PC contribution and is driven into the PC link.

The canModule is a sequential module that synchronizes the rest of the modules at bit and at frame level with the resultant frame being broadcast. In order to synchronize itself with the resultant frame, canModule uses the full coupled signal, `auxCplCoupledSignal` and a set of signals called `CAN params`, which include the set of values that conform the CAN bit timing parameters. As a result of this synchronization, it generates a set of signals that can be used by the rest of the modules to correctly interpret the resultant frame. These signals are called `auxStuBitStuffWaited`, `auxGfmGlobalFrameState` and `auxGfmGlobalBitNum`, and they are referred as `bitState` in the figure for short.

In addition, the canModule also samples `auxCplCoupledSignal` and each resultant uplink signal `iRx_i`, previously called ru_i in Sec. 11, in order to obtain a version of these signals that can then be used by other modules without taking care of the resynchronization operations needed to correctly monitor them. The sampled version of `auxCplCoupledSignal` is called `auxSynROut`, whereas the set of sampled resultant uplink signals is named `auxSynIomPortContri`. In order to notify the different modules the instant of time in which they can take as valid these sampled signals, canModule generates the *reception clock* (`clkR`). In addition, canModule also produces the *transmission clock* (`clkT`), which can be used by some modules willing to transmit, to know the correct instants of time in which they can drive their own bit contribution into the couplerModule, or into a given uplink/downlink multiplexers (umux/dmux). From now on, the `auxSynROut` signal, that is, the synchronized version of the `auxCplCoupledSignal`, will be referred as coupled signal, unless otherwise specified.

Another important information the canModule provides is the role being played by each node, that is, whether each node is acting as transmitter or receiver. This information is codified in a set of signals called `nodesRole`. Knowing each node's role is necessary, as these roles form part of the parameters of any fault-injection configuration and, on the other hand, they are essential for the loggerModule to know, and then track, which node is the transmitter of each broadcast frame.

Finally, this module generates its own contribution, carried by the `auxEfgTxSignal` signal, which is used to send an error frame when the canModule is desynchronized with the `auxCplCoupledSignal` coupled signal.

The loggerModule is a sequential module that monitors and stores specific information about the traffic during the execution of an experiment. Moreover, it is also responsible for delivering a log report at the end of an experiment. For this purpose, this module needs a set of signals provided by the canModule. Specifically, the canModule provides four different signals called, `auxSynROut`, `bitState`, `nodesRole` and `clocks`. First, the `auxSynROut` signal is used to monitor the coupled signal. Second, in order to interpret the CAN frames, the loggerModule uses the `bitState` group, which contains a set of signals that distinguishes the current bit being transmitted through the coupled signal. Third, in order to register the source port of each frame, the loggerModule uses the `nodesRole` signal, which keeps the role being played by each node and, thus, allows to identify the transmitting node. Finally, the set of clock signals grouped in `clocks` are necessary to synchronize with the CAN communication and, thus, be able to receive the coupled signal and transmit the log report. The loggerModule generates a signal called `logContri` that is used in the report phase to transmit the log report.

Note, at this point, that the `iLinks` signal, which is driven to faultInjectionModule, is an array which assembles the incoming synchronized injected signals, `auxSynIomPortContri`, the outgoing injected signals, `iTx_i`, and the synchronized coupled signal, `auxSynROut`, so they can be delivered easily.

Finally, the sequential faultInjectionModule module uses the clocks and current bit state signals provided by the canModule, as well as the iLinks signal, to receive, store, check and, when necessary, execute a set of fault-injection configurations. In order to force specific values in the signals received and transmitted from and to the nodes, this module generates a set of signals that feeds and manages a set of multiplexers. This multiplexers are located after and before the incoming and outgoing nodes signals, respectively, of the Input/Output Module. As can be seen in the figure, in the uplink, an $\texttt{umux}_i$ multiplexer is used to mask the `auxRx_i` signal coming from node$_i$. While signal `fimValueRx_i` contains the value to be injected, signal `fimBoolRx_i` contains whether the injection must be done. The resultant value is carried by the `iRx_i` signal. Similarly, in the downlink, multiplexer $\texttt{dmux}_i$ is used to mask the coupled signal addressed to node$_i$. In this case, `fimValueTx_i` contains the value to be injected and `fimBoolTx_i` contains whether the injection must be done. In turn, signal `iTx_i` contains the resultant value.

Next sections are explain in-depth the construction of the hub core. First, we explain the VHDL packages used. Then, the internals of each of the components introduced above are deeply described. Note that the source code of the hub core can be found at the and of this document, in App. D.

16.1. Packages

Next, we describe the set of VHDL packages used by the modules in order to perform their operation. Specifically, there are six packages called `defGeneral`, `defStates`, `defNCC`, `defInjection`, `defLogger` and `defInteger`. Note that the `defStates` and `defInteger` packages were already implemented in ReCANcentrate, see [Barranco, 2010]. However, some additions were made on they to enlarge their usability. As concerns the other modules, they were developed from scratch.

16.1.1. defGeneral

The `defGeneral` package defines a set of general data types and constants related with the hardware and the CAN protocol, in order to make easier the implementation process. On the one hand, it defines the `t_port` and the `t_link` enumerated types, which specify the existing ports and links, respectively. These definitions allow, for instance, to parametrize the generation of module instances, so hardware modifications have a minor impact in the implementation. On the other hand, it defines a set of data types to help to manage the data CAN frames. Some of them are the byte type, `t_byte`, the CAN identifier type, `t_id`, the CAN data type, `t_data`, and the CAN crc type, `t_crc`. Additionally, some constants are specified in order to set up default values, perform data casting and encode values in bit vectors.

16.1.2. defStates

The `defStates` package makes some data types and constants definitions related with the CAN fields. Specifically, it defines the `estadoGlobalFrame` enumerated type, which lists all the CAN fields. This type is used by the rxCAN module, later described in Sec. 16.4, to generate the `auxGfmGlobalFrameState` signal. As explained above, the `auxGfmGlobalFrameState` signal is contained inside the `bitState` group and carries the current frame field being transmitted.

Two addons have been performed on this package. On the one hand, the `fieldLong` array constant specifies the size in bits of the different frame fields. On the other hand, the `frameStateSLV` array constant specifies a bit encoding for the frame fields.

16.1.3. defNCC

The `defNCC` package is used to hold the main NCC constants and data types so they can be provided to the modules implementing the NCCs behaviour. It keeps two type of information. On the one hand, this package defines the set of NCC IDs devoted to be used by the hub, that is, the NCC ID of the Centralized Fault Injector, the NCC ID of the Hub Logger and the broadcast address. Moreover, since the Fault Injection Management Station has its own NCC ID, this package also defines it. On the other hand, this package defines a set of constants and types to manage the NCC messages, and specially the mode change operation. That is, it defines the encoding of the NCC messages and the mode change messages, as well as an enumerated type which lists the operation modes.

16.1.4. defInjection

The `defInjection` extends the `defNCC` package in order to deal with the specific issues of the fault injection. First, the `MAX_INJS` natural constant specifies the maximum number of fault-injection configurations that the CFI can hold. In this sense, note that, as will be explained in Sec. 16.5.6, the value contained in `MAX_INJS` is not only a threshold for the fault-injection configurations vector, but a value used to define the number of modules executing the fault-injection configurations. Consequently, this parameter has a great impact in the Hub Core occupation. Second, this package defines a set of constants specifying the encoding used in the fault-injection configuration NCC messages. Third, the `defInjection` package defines a set of data types, constants and functions to extract and represent the fault-injection configurations. Specifically, it defines the `t_fiCfg` record type, which keeps the information of a given fault-injection configuration. As concerns the functions, they allow to extract and decode the parameters contained in the NCC fault-injection configuration messages. Finally, this package defines a set of array and matrix types which help to transfer the injection data through the internal modules of the faultInjectionModule. The data structures using this type will be discussed in Sec 16.5.6.

16.1.5. defLogger

Similarly to the `defInjection`, the `defLogger` package extends the `defNCC` package in order to help perform the Hub Logger operation. First, a set of constant definitions specify the encoding of the NCC logger messages. Second, a set of constants and data types are defined so the loggerModule can construct the array containing the stored frames. On the one hand, constant `MAX_LOG_FRAMES` keeps the maximum number of frames that can be stored. On the other hand, the `t_logStoredFrames` array type uses this constant to define a static array whose components belong to the `t_logStoredFrame` record type. Note that this type represents a stored frame. Finally, a function called `crcCalc` defines the operation of the CRC calculation, so the loggerModule is able to transmit CAN frames.

16.1.6. defInteger

The `defInteger` package contains a set data type definitions, so the modules can deal with restricted numeric types. Specifically, this package defines various integer subtypes from a range of integer values. For instance, type `ENTER16` defines a subtype in which the value can be any integer value from 0 to 15.

16.2. Module implementation basics

The modules presented in the next sections have been implemented using different types of approaches. On the one hand, there are combinational modules, that is, modules in which the outputs do not depend on the previous state of its inputs or outputs. This type of modules are very easy to define since they can be constructed from a set of concurrent assignment statements. These statements can include different types of operators, like logical, arithmetic or concatenation, depending on the data types involved. Moreover, it is also possible to use `when-else` and `with-select-when` assignment statements, which allow to add complex conditions in the assignment process. However, note that, combinational modules are lacking of memory. On the other hand, there are sequential modules, that is, modules in which the outputs can depend on the previous state of the inputs and/or outputs. In this project the sequential modules are synchronous, that is, they depend on a specific clock signal.

As concerns the sequential modules implementing an asynchronous behaviour, we have constructed them using a specific pattern. This pattern allows to implement a module in which memory is need, but whose operation is performed after being notified by other module through a sample signal. The idea consists in using a process statement, whose sensitive list contains the system reset and the system clock signals, to periodically seek rising edges in the sample signal. When so, it executes the specific operation for which the module has been created. This implementation pattern can be seen in List. 16.1. Note that signal `lastSample` is used to store the previous value of the sample signal, so the rising edge can be detected.

```
process (reset, clk) is
  <var decl>
begin
  if reset = '1' then
    <var init>
  elsif clk'event and clk = '1' then
    if lastSample = '0' and sample = '1' then
      ...
    end if;
    lastSampleFrame := sampleFrame;
  end if;
end process;
```

Listing 16.1: Asynchronous implementation pattern.

The key of this pattern is that the system clock has a higher frequency, when compared with the reception and transmission clock signals, and, thus, it can be used to force a module to react between the activations of these clocks. In this sense, specific actions can be performed after the activation of the reception and transmission clocks.

Another VHDL pattern used frequently in the implementation of the Hub Core is the generate statement. This statement allows to instance specific components using conditional and iterative statements. However, in this project we only use the generate iterative statement. This construction allows to make various instances of the same component using a discrete range, for example, an enumerated type or an integer range. This implementation pattern provides high flexibility in designs containing a variable number of replicated modules. For instance, in order to check the whole fault-injection specification various `fimCfgExecuter` modules are generated at the same time to work concurrently and independently. Note that, as explained in Sec. 16.5.6, this is responsible for checking the conditions of a given fault-injection configuration. The main issue of this approach is that inputs and/or outputs have to be specially treated. In this sense, we usually use arrays or matrix to drive those signals to the next module(s). List. 16.2 shows the generation statement template used.

```
<label>:
for <var> in <enumerated type> generate
  <module name> : <component name>
    port map(
      <signal assignment>
    );
end generate;
```

Listing 16.2: Generation statement template.

Finally, note that in VHDL modules the signal names of the inputs depend on the definition of the interface. Thus, when discussing the internals of the modules some signals may change its name. For instance, signals conforming the bit state, that is, `auxStuBitStuffWaited`, `auxGfmGlobalFrameState` and `auxGfmGlobalBitNum`, are named `isStuffBit`, `frameState` and `frameBitNum` respectively in the faultInjectionModule. Unless it is explicitly specified, the name transformation can be resolved easily.

16.3. couplerModule

The couplerModule generates two coupled signals, called `cplCoupledSignal` and `cmpCoupledSignal`, from the contributions of the nodes, the contribution of the PC, the contribution of the loggerModule, which carries the log report, and the contribution of the canModule, which carries the resynchronization error frame. On the one hand, the `cplCoupledSignal` signal is the result of coupling all these contributions, and is the one that is provided to those modules needing the coupled signal. Moreover, it also the one driven to the nodes through the downlinks. On the other hand, the `cmpCoupledSignal` signal is specifically generated to be driven to the PC, so it cannot receive its own contribution, as explained in Sec. 11.4.

The interface of this module, that is, its inputs and outputs, is shown in Table 16.1 and explained next.

Name	I/O	Type	Description
iomPortContri_0	in	std_logic	Contribution of Node 0
iomPortContri_1	in	std_logic	Contribution of Node 1
iomPortContri_2	in	std_logic	Contribution of Node 2
iomPcContri	in	std_logic	Contribution of PC
logContri	in	std_logic	Contribution of loggerModule
efgTxSignal	in	std_logic	Contribution of canModule
cplCoupledSignal	out	std_logic	Full coupling
cmpCoupledSignal	out	std_logic	PC coupling

Table 16.1.: couplerModule interface.

As explained in the design section, this module was already implemented in ReCANcentrate and, thus, its code was available. However, we had to carry out some modifications to both append additional contributions and enable the second AND gate. Specifically, as introduced previously in Sec. 11.4, the operation that generates both coupling signals is performed thanks to two AND gates, each of which collects a specific set of contributions. Just in the previous implementation of the couplerModule, this hardware description was carried out thanks to the data flow design paradigm of VHDL, which, as explained in Sec. 14.1.4, allows to describe a circuit by defining the relation of the outputs in terms of the inputs. The statements implementing this module are executed concurrently and combinationally, following the construction pattern shown in List. 16.3.

```
output <=
  input0 and
  input1 and
  ...
  inputN;
```

Listing 16.3: couplerModule implementation pattern.

16.4. canModule

The canModule is responsible for providing useful information about the traffic to the other modules. Specifically, it is able to synchronize physically with what is being transmitted through the coupled signal, that is, it distinguishes the set of bits conforming the traffic. Table 16.2 shows the specific interface of this module. Next, this set of signals are described.

Name	**I/O**	**Type**	**Description**
reset	in	std_logic	System reset
clk	in	std_logic	System clock oscillator
cplCoupledSignal	in	std_logic	Not synchronized coupled signal
iomPortContri_0	in	std_logic	Contribution of Node 0
iomPortContri_1	in	std_logic	Contribution of Node 1
iomPortContri_2	in	std_logic	Contribution of Node 2
brp	in	std_logic_vector(5 downto 0)	Baud rate prescaler
tsegment1	in	std_logic_vector(5 downto 0)	Number of TQ for segment 1
tsegment1	in	std_logic_vector(2 downto 0)	Number of TQ for segment 2
sjw	in	std_logic_vector(1 downto 0)	Number of TQ for synch
sync	in	std_logic	Type of resynchronization
sam	in	std_logic	Number of sample times
synClkR	out	std_logic	Reception clock
synClkT	out	std_logic	Transmission clock
efgTxSignal	out	std_logic	Error frame generation
stuBitStuffWaited	out	std_logic	Current bit is stuff
gfmGlobalFrameState	out	std_logic	Current frame field
gfmGlobalBitNum	out	std_logic	Current bit number
synROut	out	std_logic	Synchronized coupled signal
synIomPortContri	out	t_portSignalArray	Synchronized contribution of the Nodes
nodesRole	out	t_portTypeArray	Nodes' role

Table 16.2.: canModule interface.

First, the canModule generates a set of clocks synchronized at the bit level. On the one hand, the `synClkT` clock activates when the value of a bit begins to be transmitted, that is, at the bit's start. On the other hand, the `synClkR` clock activates when the value of a the bit is stable and, thus, it can be read, that is, at the bit sample point. To generate these two clocks the canModule needs not only the value of the coupled signal but also the CAN bit-rate parameters. This parameters, called `brp`, `tsegment1`, `tsegment2`, `sjw`, `sync` and `sam`, are configuration constants that specify the CAN operation. Second, it generates a synchronized version of the the set of signals carrying the nodes contribution, that is, `iomPortContri_i`). These new signals, called `synIomPortContri_i`, can be accessed without taking care of the resynchronization operations needed to correctly monitor them. Third, the canModule outputs the specific bit being transmitted. To do so it uses three different signals called `stuBitStuffWaited`, `gfmGlobalFrameState` and `gfmGlobalBitNum`. On the one hand, the first signal distinguishes whether it is a stuff bit or not. On the other hand, the second and the third identifies the current field and bit currently being transmitted. Fourth, this module generates an error frame through `auxEfgTxSignal` to force a resynchronization, in case of loosing it. Finally, the canModule monitors the arbitration mechanism to provide, through the `nodesRole` signal, the specific role being played by the nodes.

The operation of the canModule is carried out by four different submodules called `physicalLayer`, `rxCAN`, `errorFrameGenerator` and `nodeRoleModule`. The former generates the clocks signals as well as the synchronized version of the nodes contribution. The second monitors the coupled signal and, thanks to the reception clock, distinguishes the specific bit and field being transmitted. When the `rxCAN` module detects a lack of synchronization, it notifies to the `errorFrameGenerator` that must send an error frame. Finally, a set of `nodeRoleModule` modules, one for each node, determines the role of the nodes, that is, whether they are transmitters or receivers. To do so, a given `nodeRoleModule` module keeps track of a nodes behaviour during the arbitration process. For this purpose it needs the nodes contribution, the coupled signal and the set of bit state signals provided by the rxCAN.

The implementation of these modules was partially performed in ReCANcentrate, see [Barranco, 2010]. In the case of the `physicalLayer`, `errorFrameGenerator` modules no modifications were carried out. As concerns the `rxCAN` module, it has been revised and modified to add a complementary output containing the position inside the field of the current bit being transmitted, that is, the `gfmGlobalBitNum` signal. Finally, although the `nodeRoleModule` module has been coded from scratch, it is deeply inspired in an existing module contained in the module implementing the fault-treatment mechanisms.

16.5. faultInjectionModule (FIM)

The faultInjectionModule (FIM) [1] implements the operation of the Centralized Fault Injector NCC. To do so it generates a set of signals which feed the set of multiplexers placed after and before the incoming and outgoing nodes signals, respectively. The values set in these signals depend on the fulfilment of the conditions specified in the fault-injection configurations, which are evaluated by the fautInjectionModule itself.

Table 16.3 shows the interface of the faultInjectionModule. First, the system clock, named `clk`, is used by some sequential submodules to implement the asynchronous pattern previously described in Sec. 16.2. Second, the reception clock (`clkR`) and the transmission clock (`clkT`) are used to determine the specific instant in which a given bit can be readed and injected, respectively. Third, the set of signals distinguishing the bit state, that is, `frameState`, `frameBitNum` and `isStuffBit`, as well as the signals contained in `iLinks`, are used to extract specific information about the traffic. On the one hand, they allow to identify the NCC messages broadcast by the FIMS. On the other hand, they are necessary to check the traffic conditions specified in a given fault-injection configuration. Fourth, the `nodesRole` signal allows to check the role conditions involved in a given fault-injection configurations. Finally, the set of outputs are used to force specific values in the uplinks and downlinks. Specifically, for a given link, exists a pair of signals named `fimBool` and `fimValue` that contain whether an injection must be done and the value that must be injected, respectively.

[1]Do not confuse FIM (fault-injection module) with FIMS (fault-injection management station)

Name	I/O	Type	Description
clk	in	std_logic	System clock oscillator
clkR	in	std_logic	Reception clock
clkT	in	std_logic	Transmission clock
iLinks	in	t_linkSignalArray	Array containing all the links values
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
nodesRole	in	t_portSignalArray	Nodes' role
fimBoolTx_0	out	boolean	Downlink of Node 0 must be injected
fimValueTx_0	out	std_logic	Downlink of Node 0 injection value
fimBoolTx_1	out	boolean	Downlink of Node 1 must be injected
fimValueTx_1	out	std_logic	Downlink of Node 1 injection value
fimBoolTx_2	out	boolean	Downlink of Node 2 must be injected
fimValueTx_2	out	std_logic	Downlink of Node 2 injection value
fimBoolRx_0	out	boolean	Uplink of Node 0 must be injected
fimValueRx_0	out	std_logic	Uplink of Node 0 injection value
fimBoolRx_1	out	boolean	Uplink of Node 1 must be injected
fimValueRx_1	out	std_logic	Uplink of Node 1 injection value
fimBoolRx_2	out	boolean	Uplink of Node 2 must be injected
fimValueRx_2	out	std_logic	Uplink of Node 2 injection value

Table 16.3.: faultInjectionModule interface.

The faultInjectionModule is divided in six interconnected modules, as can be seen in Fig. 16.2. Note that some modules follow the naming convention `component_label::component_name`. This indicates that a given component `component_name` has been instantiated from, or belongs to, a given component `component_label`. Modules instantiated from the same component share the same implementation, the only difference is the inputs and/or outputs used. In this project this allows us to reuse the implementation of the `fimFrameReader`, `fimFrameFilter` and `fimCoordinator`, to construct other NCCs. For instance, as will be explained later, the loggerModule contains a set of modules called `logFrameReader`, `logFrameFilter` and `logCoordinator`, which are instantiated from the same components that their faultInjectionModule counterparts. Next, the modules conforming the faultInjectionModule are introduced.

The first module, called `fimFrameReader` and instantiated from the `nccFrameReader` component, collects all the frames transmitted through the coupled signal, in order to process those frames carrying a NCC message. After detecting a new frame, this module extracts the values contained in the identifier and the data fields, that is, the field identifying the destination address and the content of the NCC message; this two values are output through the `auxId` and `auxData` signals, respectively. Then this module instructs the `fimFrameFilter` module, by means of the `ffrSampleFrame` signal, to perform the filter process. As concerns the `fimFrameFilter` module, it is instantiated from the `nccFrameFilter` component and it purpose is to force to accept only those NCC messages destined to the current NCC, in this case, the Centralized Fault Injector. To

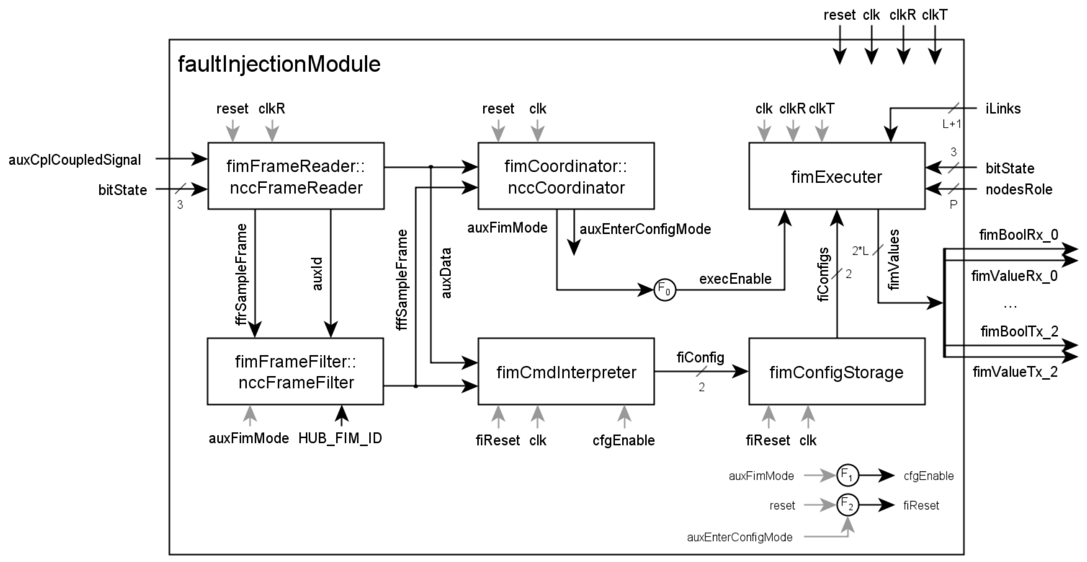


Figure 16.2.: faultInjectionModule implementation diagram.

perform this operation, this module needs, not only the destination of the message, but the identifier of the NCC and its current operation mode. As can be seen in the figure, the identifier is a constant, called `HUB_FIM_ID` provided directly to the module, whereas the current operation mode can be obtained from signal `auxFimMode`, which is provided by the `fimCoordinator`. Once a new NCC message destined to the Centralized Fault Injector is available, the `fimFrameFilter` instructs the `fimCoordinator` and the `fimCmdInterpreter`, by means of the `fffSampleFrame` signal, to accept the content of the message, that is, the data contained in `auxData` and provided by the `fimFrameReader`. As concerns the `fimCoordinator` and the `fimCmdInterpreter` modules, they deal with the content of the messages to carry out specific actions. On the one hand, the `fimCoordinator`, which is instantiated from the `nccCoordinator` component, executes the NCC mode change commands and keeps the current value of the NCC operation mode. It generates two signals named `auxFimMode` which, as discussed above, contains the current operation mode of the NCC, and `auxEnterConfigMode`, which activates when the NCC enters in configuration mode. On the other hand, the `fimCmdInterpreter` processes the fault-injection configuration messages transmitted, each of which contains the value of a given fault-injection parameter, in order to construct the whole fault-injection configuration. Then, when a given fault-injection configuration is fully constructed, a specific NCC command forces this module to send it to the `fimConfigsStorage`. To do so it uses the `fiConfig` signal which groups two real implemented signals, named `auxFiCfg` and `fciSampleFiCfg`. The former contains the fault-injection configuration itself, whereas the second is a sample signal which notifies that a new fault-injection configuration is ready to be read. In turn, the `fimConfigsStorage` collects and stores the set of fault-injection configurations sent by the `fimCmdInterpreter`. This set can be accessed through the `fiConfigs` signal, which groups two real implemented signals named `auxFiCfgs` and `auxFiCfgs_count`. The former contains the set of fault-injection configuration itself, in a static array, whereas the second keeps the number of fault-injection configurations stored. Finally, the `fimExecuter` checks the conditions involved in the fault-injection configurations and generate the set of signals that force the fault-injection.

Next, the interface and the implementation of these six modules is described in-depth.

16.5.1. nccFrameReader

The nccFrameReader is a sequential module that decodes the CAN data frames received through the coupled signal in order to collect the NCC messages transmitted. As shown in the interface of this module, Table 16.4, this module needs not only the coupled signal, but the `clkR` signal. This last signal allows to synchronize with what is being received, at a bit level. Moreover, the set of bit state signals, called `frameState`, `frameBitnum` and `isStuffBit`, are also necessary, in order to distinguish the specific bit being transmitted through the coupled signal. Once a new frame is received, the nccFrameReader extracts and outputs the values contained in its identifier and data fields. In this sense, the value of the identifier is output through the `id`, whereas the value of the data is output through the `data` signal. Note that, in a frame carrying a NCC message the identifier field contains the NCC destination address, whereas the data field carries the content of the message. Finally, this module notifies the nccFrameFilter, by means of the `sampleFrame` signal, that a new frame has been received. As explained later, this last module is responsible for identifying those frames containing a NCC message destined to the current NCC and, if so notify the corresponding of modules in order to process it.

Name	**I/O**	**Type**	**Description**
reset	in	std_logic	System reset
clkR	in	std_logic	Reception clock
coupledSignal	in	std_logict	Coupled signal
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
sampleFrame	out	std_logic	Notify new frame
id	out	t_id	Frame identifier
data	out	t_data	Frame data

Table 16.4.: nccFrameReader interface.

This module uses a process statement whose sensitive list is composed of the `reset` and `clkR` signals, that is, it wakes up each time the system resets and each time a bit can be read. A simplified version of the implemented logic is shown in List. 16.4. On the one hand, when the system is reset, the values of the internal variables, as well as the outputs, are initialized. On the other hand, when the sample point of a bit is reached and the current bit is not as stuff bit, the logic distinguishes among the possible frame fields, in order to perform specific actions. First, when the channel is idle, that is, none is transmitting, the outputs are initialized. Second, when the current field is the identifier or the data, their contents are read and stored. Third, if an error is found, that is, the error flag field is detected, the logic activates a local variable which disables the output of the stored values. Finally, when reaching the first bit of the interfield, and no error has been found, this module outputs the values contained in the identifier and the data fields, through the `id` and `data` signals. Moreover, it uses the `sampleFrame` signal to notify that a new frame has been received and that the values of the identifier and data fields have been output. Note that the `sampleFrame` signal remains active one bit, since the reception of the first bit of the interField until the reception of the second bit of the same field.

```
process (reset, clkR) is
  <var decl>
begin
  if reset = '1' then
    <var init>
  elsif clkR'event and clkR = '1' and isStuffBit = '0' then
    sampleFrame <= '0';
    case frameState is
      when idle =>
        <var init>
      when idField =>
        <capture id>
      when dataField =>
        <capture data>
      when errorFlag =>
        error := true;
      when interField =>
        if frameBitnum = 0 then
          if not error then
            <output captured values>
            sampleFrame <= '1';
          else
            error := false;
          end if;
        end if;
      when others => null;
    end case;
  end if;
end process;
```

Listing 16.4: nccFrameReader implementation.

16.5.2. nccFrameFilter

The nccFrameFilter module is a combinational module that determines whether a CAN frame contains a NCC message and, if so, whether it is destined to the current NCC. The set of signals provided and generated to and by this module are listed in Table 16.5. The operation of this modules depends directly on the `nfrSampleFrame` signal, which contains the sample frame provided by the nccFrameReader. In fact, this signal behaves in the code as an enabling signal. When it activates, this module checks whether the value of the identifier of the frame, called `id`, corresponds to the local NCC ID or it has been transmitted in broadcast. Note that the nccFrame-Filter receives the local NCC ID through the `local_id` signal. Moreover, in order to discard the frames using unicast addressing during the execution phase, as explained in Sec. 13.6, it also checks the operation mode of the NCC. This last value can be obtained from the `nccMode` signal which, in turn is generated by the nccCoordinator. Finally, if the current frame contains a NCC message destined to the current NCC, this module activates the output signal `sampleFrame` in order to notify the other modules that it must be processed.

Name	I/O	Type	Description
nfrSampleFrame	in	std_logic	Notify new frame
id	in	t_id	Frame identifier
local_id	in	t_id	Local NCC ID
nccMode	in	t_nccMode	NCC mode
sampleFrame	out	std_logic	Notify new NCC message

Table 16.5.: nccFrameFilter interface.

The logic used to implement the nccFrameFilter module is described in List. 16.5. It uses a single assignment statement which is enabled when signal `nfrSampleFrame` activates. When so, this module checks the two conditions previously explained.

```
sampleFrame <=
  '1' when nfrSampleFrame = '1' and
      ((id = BROAD_ID) or
       (id = local_id and nccMode /= execMode))
    else
  '0';
```

Listing 16.5: nccFrameFilter implementation.

16.5.3. nccCoordinator

The nccCoordinator is a sequential module that keeps the value of the NCC operating mode and executes mode change commands so it can be updates. As can be seen in the interface of this module, Table 16.6, apart from the `reset` and `clk` signals, which are used to synchronize with the nccFrameFilter, it has two inputs, called `sampleFrame` and `data`, and two outputs, called `enterConfigMode` and `nccMode`. As concerns these two inputs, the `sampleFrame` signal enables the NCC message content reading, that is, when the `sampleFrame` signals activates this module accepts the value carried by signal `data`. As explained previously, this last signal is provided by the nccFrameReader and carries the content of a NCC message whose destination is the current NCC. As concerns the outputs, the `enterConfigMode` signal activates when the NCC enters in configuration mode, whereas the `nccMode` signal outputs the current operation mode of the NCC. Note that, the default operation mode of an NCC is the configuration mode.

The nccCoordinator uses the asynchronous implementation pattern presented in Sec. 16.2, that is, it contains a process statement driven by the system clock which seeks the rising edge of a sample frame, in this case signal `sampleFrame`. This signal, as explained previously, is provided by the nccFrameFilter and activates after identifying a NCC message destined to the NCC. When this happens, the nccCoordinator accepts the content of the NCC message, carried by the `data` signal, and checks whether it is a change mode NCC message. If so, this module decodes the command and performs the operation mode modification, accordingly with the rules

Name	I/O	Type	Description
reset	in	std_logic	System reset
clk	in	std_logic	System clock
sampleFrame	in	std_logic	Notify new NCC message
data	in	t_data	Frame data
enterConfigMode	out	std_logic	Notify enter configuration mode
nccMode	out	t_nccMode	NCC operation mode

Table 16.6.: nccCoordinator interface.

specified in Sec. 9.1. A simplified version of the code implementing this modules is shown in List. 16.6.

```
nccMode <= auxNCCMode

process (reset, clk) is
  <var decl>
begin
  if reset = '1' then
    <var init>
  elsif clk'event and clk = '1' then
    enterConfigMode <= '0';
    if lastSampleFrame = '0' and sampleFrame = '1' and
      data(1 to 3)     = CMD_MCH                  then
      case data(4 to 8) is
      ...
      end case;
    end if;
    lastSampleFrame := sampleFrame;
  end if;
end process;
```

Listing 16.6: nccCoordinator implementation.

As concerns the outputs, their values are updated separately. On the one hand, the value `nccMode` is updated thanks to an assignment statement placed outside the process statement. On the other hand, the statement updating of the `enterConfigMode` value is placed inside the process statement; specifically, after the code processing an enter-config-mode NCC message. Note that the `enterConfigMode` signal remains active one system clock cycle.

16.5.4. fimCmdInterpreter

The fimCmdInterpreter is a sequential module that constructs a given fault-injection configuration from a set of NCC configuration messages. When the fault-injection configuration is fully specified the Fault Injection Management station sends a specific NCC message that forces this module to send it to the fimConfigsStorage, where it remains until the experiment ends. The fimCmdInterpreter module, similarly to the fimCoordinator, synchronizes with the fimFrameFilter. In fact, as will be explained below, the internal of both modules is very similar.

The set of signals provided and generated to and by the fimCmdInterpreter module are shown in Table 16.7. First, the `enable` signal allows to restrict its operation so it is performed only during the configuration phase. As shown in Fig. 16.2, this signal is generated outside of this module, from the current NCC operation mode. Second, similarly to the fimCoordinator, this module needs the `reset` and `clk` signals, so it can synchronize with the fimFrameFilter using the asynchronous pattern described in 16.2. However, this `reset` signal does not come from the system reset, instead, it is generated outside of this module from the system reset and the `auxEnterConfigMode` signals. This decision allows to remove a partial constructed fault-injection configuration not only when the system resets, but when the NCC enters in configuration mode, so another experiment is ready to be configured. Third, analogously to the fimCoordinator, the `sampleFrame` signal enables the NCC message content reading, whereas the `data` signal contains the content of the NCC message itself. Finally, this module outputs a sample signal, `sampleFiCfg`, and a value signal, `fiCfg`. On the one hand, the former is responsible for notifying that a new fault-injection configuration is ready to be read. On the other hand, the second contains the fault-injection configuration itself. These two signals are driven to the fimConfigsStorage, where fault-injection configurations are received and stored.

Name	**I/O**	**Type**	**Description**
enable	in	std_logic	Configuration mode enable
reset	in	std_logic	Configuration mode reset
clk	in	std_logic	System clock
sampleFrame	in	std_logic	Notify new NCC message
data	in	std_logic_vector(1 to 64)	Frame data
sampleFiCfg	out	std_logic	Notify new fault-injection configuration
fiCfg	out	t_fiCfg	Fault-injection configuration

Table 16.7.: fimCmdInterpreter interface.

The fimCmdInterpreter uses the asynchronous implementation pattern presented in Sec. 16.2, just as can be seen in the simplified version of the code implementing this modules, shown in List. 16.7. This allows to deal with the NCC messages when the fimFrameFilter launches a notification. When this happens, this module checks whether it is a fault-injection configuration message and, if so, it decodes the message and performs the operation associated. On the one hand, if it is a parameter configuration message, this module decodes and stores the parameter value. On the other hand, if it is an end-of-configuration message, this module enables the `sampleFiCfg` signal, so the fimConfigsStorage can read the fault-injection configuration through the `fiCfg` signal. Thus, a given fault-injection configuration is constructed by means a set of parameter configuration messages until an end-of-configuration message is received, when it is output. Note that the `sampleFiCfg` signal remains active one bit time, so the fimConfigsStorage is able to capture the notification using the asynchronous pattern. For this purpose, this assignment involves the `sampleFrame`.

```
process (reset, clk) is
  <var decl>
begin
  if reset = '1' then
    <var init>
  elsif clk'event and clk = '1' and
      enable = '1'           then

    if lastSampleFrame = '0' and sampleFrame = '1' and
       data(1 to 3)    = CMD_CFG                 then

      case data(4 to 8) is
      ...
      end case;
    end if;
    lastSampleFrame := sampleFrame;
  end if;
end process;

sampleFiCfg <= sampleFrame and fiCfgReady;
```

Listing 16.7: fimCmdInterpreter implementation.

16.5.5. fimConfigsStorage

The fimConfigsStorage is responsible for storing and making available all the fault-injection configurations. For this purpose, this module has to synchronize with the fimCmdInterpreter which, as explained previously, provides these faul-injection configurations one by one. As can be seen in Table 16.8, all the inputs are devoted to construct the asynchronous implementation pattern which performs this synchronization. First, the `reset` and `clk` signals feed the process statement. Note that, identically to the fimCmdInterpreter, this reset signal is generated outside of this module from the system reset and the `auxEnterConfigMode` signals. This allows to remove all the fault-injection configurations stored, when the system resets and when the NCC enters in configuration mode, so it is ready for a new fault-injection experiment. Second, the `sampleFiCfg` signal enables the fault-injection configuration reading. That is, when the `sampleFiCfg` signals activates, this module accepts the value carried by signal `fiCfg`, which contains the fault-injection configuration constructed by the fimCmdInterpreter. Finally, the fimConfigsStorage generates two different signals. On the one hand, the `fiCfgs` signal contains the set of fault-injection configurations. On the other hand, the `fiCfgs_count` signal contains the number of stored configurations standing inside the `fiCfgs` signal.

As explained above, the fimConfigsStorage uses the asynchronous implementation pattern to activate activates when the fimCmdInterpreter module sends a new fault-injection configuration. In fact, as can be seen in the simplified code of List. 16.8, the implementation is very similar to the implementations presented previously, but simpler. When a new fault-injection configuration is available, this module collects and stores it in a specific array, `fiCfgs`. Likewise, the value of the `fiCfgs_count` signal is incremented by one. Finally, these two signals are output, so they can be consulted anytime.

Name	I/O	Type	Description
reset	in	std_logic	Configuration mode reset
clk	in	std_logic	System clock
sampleFiCfg	in	std_logic	Notify new fault-injection configuration
fiCfg	in	t_fiCfg	Fault-injection configuration
fiCfgs	out	t_fiCfgs;	List of fault-injection configurations
fiCfgs_count	out	ENTER11	Number of fault-injection configurations

Table 16.8.: fimConfigsStorage interface.

```
process (reset, clk) is
  <var decl>
begin
  if reset = '1' then
    <var init>
  elsif clk'event and clk = '1' then
    if lastSampleFiCfg = '0' and sampleFiCfg = '1' and
       fiCfgs_count < MAX_INJS                          then
      fiCfgs_count++;
      fiCfgs(fiCfgs_count) <= fiCfg;
    end if;
    lastSampleFiCfg := sampleFiCfg;
  end if;
end process;
```

Listing 16.8: fimConfigsStorage implementation.

16.5.6. fimExecuter

The `fimExecuter` module performs, when corresponds, the fault injection. For this purpose, this module needs several information, listed in Table 16.9. First, similarly to the fimCmdInterpreter, this module only performs its operation during the execution phase. To do so, it is provided with an enabling signal, `enable`, which activates only when the NCC is in execution mode. Moreover, this signal is used also to reset the internal state, when the NCC is not in execution mode. Second, to both, receive and inject from and into the links, this module has to be synchronized with the coupled signal. To do so, the reception and transmission clock signals, `clkR` and `clkT`, as well as the set of signals conforming the bit state, `frameState`, `frameBitNum` and `isStuffBit`, are provided. Third, in order to carry out specific actions after receiving and injecting a bit, the system clock signal, that is, `clk`, is needed. Fourth, as concerns the list of fault-injection configurations, as explained previously, they are supplied by the fimConfigsStorage through the `fiCfgs` and `fiCfgs_count` signals. Fifth, in order to check the role conditions involved in the fault-injection configurations the role of the nodes, which is carried by the `nodesRole` signal, is provided. Finally, the fimExecuter generates `fimValues`, a link-indexed array of pairs of signals which carry the value to be injected and whether it has to be injected.

As can be seen in Fig. 16.3, the fimExecuter module is built from three different parts called `fimCfgExecuter_array`, `fimInjValueSelector_array` and `fimValueGenerator_array`. Each of these parts is a set of specific modules of the same type, just as explained in Sec.

Name	I/O	Type	Description
enable	in	std_logic	Execution mode enable
clk	in	std_logic	System clock
clkR	in	std_logic	Reception clock
clkT	in	std_logic	Transmission clock
fiCfgs	in	t_fiCfgs;	List of fault-injection configurations
fiCfgs_count	in	ENTER11	Number of fault-injection configurations
iLinks	in	t_linkSignalArray	Content of the uplinks, downlinks and coupled signal
nodesRole	in	t_portSignalArray	Nodes' role
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
fimValues	out	t_fimValues	Set of signal used to inject

Table 16.9.: fimExecuter interface.

16.2, when describing the generation statement. That is, `fimCfgExecuter_array` is a set of `fimCfgExecuter` modules, `fimInjValueSelector_array` is a set of `fimInjValueSelector` modules and `fimValueGenerator_array` is a set of `fimValueGenerator` modules. Next, each of these parts and their interconnections is introduced.

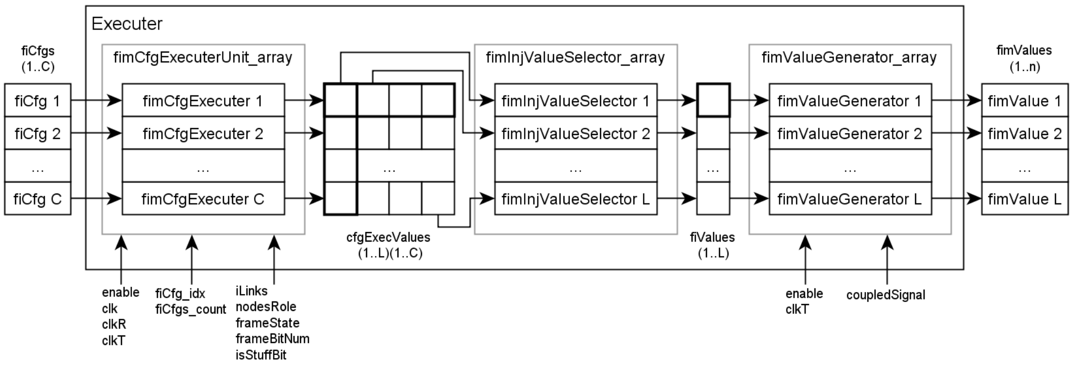


Figure 16.3.: fimExecuter implementation diagram.

First, the array of fault-injection configurations, `fiCfgs`, is driven to the fimCfgExecuter_array, where each fault-injection configuration is attached to a single fimCfgExecuter. A given fimCfgExecuter is responsible for checking the conditions involved in its fault-injection configuration and generating a set of signals informing about the value to be injected in each link, uplink and downlink, of each node. Thus, the combination of all the fimCfgExecuters generates a matrix, which is called `cfgExecValues`. Note that, since the fimCfgExecuter_array generates a set of injections for each link, one for each the fimCfgExecuter, it is necessary to apply a filter so only one injection is performed in a given link. The fimInjValueSelector_array is responsible to do so. Each of the fimInjValueSelector modules conforming the fimInjValueSelector_array is attached to the set of values of the `cfgExecValues` addressed to each of the links, just as shown in the figure. Then, each fimInjValueSelector selects only one of the injections value. The criteria used consists of selecting the last defined active injection. Then, the selected value to inject is output through the `fiValues` signal, an array, so they can be read for the fimValueGenerator_array. Finally, the fimValueGenerator_array generates the specific values to be injected from the information gathered from the `fiValues` signal. To do so it uses a set of fimValueGenerator modules, one for each link, which output their result in a array, the `fimValues` signal.

Next, the implementation of each of these parts and their internals is described in detail.

fimCfgExecuter

A given fimCfgExecuter module is responsible for instructing the fimInjValueSelector_array to inject, when the conditions of a specific fault-injection configuration are met. For this purpose, it uses the set of signals listed in Table. 16.10. First, the specific fault-injection configuration attached to the fimCfgExecuter is driven through the `fiCfg` input signal. Second, in order to activate the fimCfgExecuter two conditions have to be met. On the one hand, the faultInjectionModule has to be in execution mode. On the other hand, a specific fault-injection configuration has to be specified for the given fimCfgExecuter. The first condition can be checked thanks to the `enable` signal, which, as explained previously, activates when the NCC enters in execution mode. As concerns the second condition, since the assignment of the fault-injection configurations to each fimCfgExecuter is performed orderly, signals `fiCfgs_num` and `fiCfgs_count`, which contain the number of fault-injection configurations and the position of the current fimCfgExecuter, respectively, are used to check it. Third, in order to read the value contained in the different links it is necessary to provide the reception clock, `clkR`, the bit state, `frameState`, `frameBitNum` and `isStuffBit`, and the values of the links, `iLinks`. Fourth, in order to output the values to be injected the set of signals named `port0UpValue`..`port2DwValue` are used. Note that exists one of these signals for each port and transmission direction. For instance, signal `port0UpValue` carries the value to be injected in the uplink of port 0. Additionally, in order to output the injection value synchronously, signal `clkT`, which notifies the correct instant in which a signal can be transmitted, is provided. Finally, as explained below, the system clock signal, that is, `clk`, is used to keep the value of specific signals when they are driven between modules.

The fimCfgExecuter module is constructed from a set of interconnected modules, as can be seen in Fig. 16.4. The core of these modules are the `fimChecker` and the `fimOffsetDelayer`, which decide whether the injection must be performed. For this, they check the fulfillment of the conditions specified in the fault-injection configuration, that is, the aim, withdraw, target_frame, aim and cease conditions, which are respectively named `startTrigger`, `endTrigger`, `selTrigger`, `fiStart` and `fiEnd` in the implementation. The `fimChecker` is responsible for evaluating bit by bit all these condition, except for the offset defined in the aim condition, see Sec. 10. This last feature is carried out by the fimOffsetDelayer, which delays the injection signal provided by the previous module a given number of bits. To carry out their operation these two modules use the information provided by other modules and the fault-injection configuration itself.

Next we describe how these modules obtain this information and how the injection signals are output from the fimCfgExecuter.

Name	I/O	Type	Description
enable	in	std_logic	Execution mode enable
clk	in	std_logic	System clock
clkR	in	std_logic	Reception clock
clkT	in	std_logic	Transmission clock
fiCfg	in	t_fiCfgs	Fault-injection configuration
fiCfgs_num	in	ENTER11	Current fault-injection configuration
fiCfgs_count	in	ENTER11	Number of fault-injection configurations
iLinks	in	t_linkSignalArray	Content of the uplinks, downlinks and coupled signal
nodesRole	in	t_portSignalArray	Nodes's role
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
port0UpValue	out	t_fiValue	Injected value for the uplink of Port 0
port0DwValue	out	t_fiValue	Injected value for the downlink of Port 0
port1UpValue	out	t_fiValue	Injected value for the uplink of Port 1
port1DwValue	out	t_fiValue	Injected value for the downlink of Port 1
port2UpValue	out	t_fiValue	Injected value for the uplink of Port 2
port2DwValue	out	t_fiValue	Injected value for the downlink of Port 2

Table 16.10.: fimCfgExecuter interface.

First, the aim, withdraw and target-frame conditions can be checked thanks to the `fimStartTriggerCounter`, `fimEndTriggerCounter` and `fimSelTriggerCounter`, respectively. This three modules are implemented using the same component, the `fimTriggerCounter`, which outputs the number of times a given condition has been fulfilled. Moreover, this component also outputs a signal indicating whether the current frame fulfills the condition. Note from the figure that, the first two modules output signals `startTriggerCount` and `endTriggerCount`, which carry the number of frames fulfilling the aim and withdraw conditions, respectively, whereas the third module outputs signal `currentTriggerFrame`, which notifies whether the current frame fulfills the target-frame condition. Second, in order to check the cease bit count condition, the `fimEndBitCounter` module is used. This module is implemented from the `bitCounter` component, which, as its name suggests, implements a bit counter. As explained later, this component is also used in the implementation of the fimOffsetDelayer. Finally, additional data can be obtained from the fault-injection configuration. Specifically, this data includes the fault-injection mode, the number of times an aim condition must be seen before starting the injection or the specific bit in which the injection must end.

As concerns the `fimManager`, it generates the set of outputs of the fimCfgExecuter. For this, it collects different information from different places. On the one hand, it collects the `delayedInject` signal provided by the `fimOffserDelayer` which, as explained, determines whether the injection must be done. On the other hand, it is provided with the injection value and the target link, which can be found into the fault-injection configuration. These two values are labelled in the figure as `fiCfg.value` and `fiCfg.link`, respectively.

Finally, note that five signals are generated and used in parallel to these modules. First,

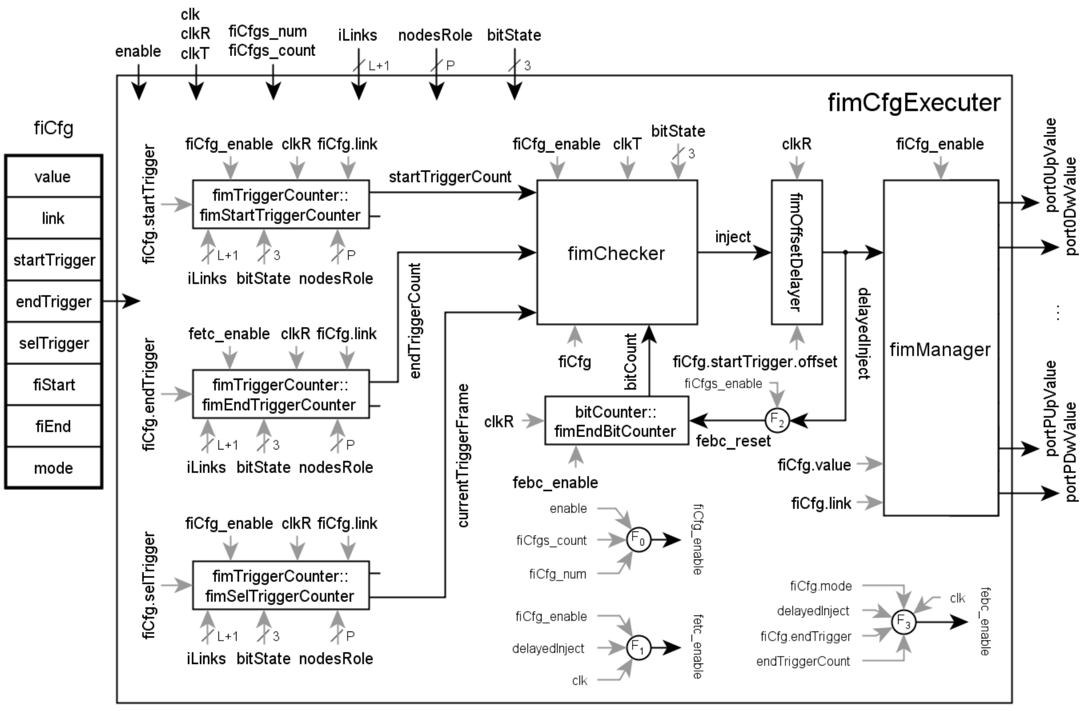


Figure 16.4.: fimCfgExecuter implementation diagram.

`fiCfg_enable` activates when the fimCfgExecuter is permitted to execute. For this purpose it checks the `enable`, `fiCfgs_num` and `fiCfgs_count`. On the one hand, the first signal notifies whether the NCC is in execution mode. On the other hand, the other two signals are used to check that exists a valid fault-injection configuration attached to the given fimCfgExecuter. For this, it is enough to compare these two values. If the position of a given fimCfgExecuter is greater than the number of specified fault-injections, there is no valid fault-injection configuration available and, thus, the fimCfgExecuter must be disabled. Second, `fetc_enable` is used as an enabler of the `fimEndTriggerCounter`. This allows to activate this module, not only when the fimCfgExecuter activates, but after an injection has started. In order to know when the injection has started signal `delayedInject` is sampled and stored thanks to a process statement fed with the `clk` signal. Finally, signals `febc_reset` and `febc_enable` are used to initialize and activate, respectively, the `fimEndBitCounter`. On the one hand, the first signal uses the `fiCfg_enable` and `delayedInject` signals to initialize the internal state of the module when the fimCfgExecuter is not enabled and when no injection is performed. On the hand, the second signal enables its operation when a cease condition is waited, which can happen in different ways, depending on the fault-injection mode. Moreover, to do so it is necessary to monitor and keep the event notifying the fulfillment of the withdraw condition, that is, check and store whether the value of the `endTriggerCount` signal reaches the specified end trigger condition count.

Next, all these modules are described in-depth. Note that, since the three trigger counters are instantiated from the same component, only one description is performed. Moreover, since the fimEndBitCounter module implements a regular bit counter its description lacks of interest, thus, it is avoided.

fimTriggerCounter

The fimTriggerCounter component is responsible for identifying those frames fulfilling a trigger condition, that is, an aim, withdraw or target_frame condition. As a result, it generates two signals. On the one hand, it outputs the number frames fulfilling the condition. On the other hand, it notifies whether the current frame fulfils the condition. To perform its operation, the fimTriggerCounter uses the set of signals listed in Table 16.11. First, the `enable` signal allows to activate this component depending on an external condition. Second, in order to check content-based conditions, that is, conditions that involve the value contained in a frame, it is necessary to provide the `iLinks`, the set signals conforming the bit state and the trigger condition itself, which is provided by the fault-injection configuration and is labelled `trigger`. The `iLinks` signal, jointly with the set signals of the bit state, are used to distinguish the specific bits being transmitted through the different links. These values are compared with the bit pattern contained within the trigger condition. Third, in order to check the role-based condition of the trigger condition, signals `fiLink` and `nodesRole` are provided. On the one hand, the first contains the link in which the injection is performed, that is, the port in which the role is applied. On the other hand, the second contains the role being played by the nodes. Fourth, the reception clock, that is, `clkR`, it is used to synchronize at a bit level with the coupled signal and, thus, check the both conditions previously explained. Finally, this module outputs both, the number of times that the trigger condition has been fulfilled, through the `triggerCount` signal, and whether the current frame fulfills the trigger condition, through the `currentTriggerFrame` signal.

Name	**I/O**	**Type**	**Description**
enable	in	std_logic	fimCfgExecuter enable
clkR	in	std_logic	Reception clock
fiLink	in	t_fiLink	Target link
trigger	in	t_trigger	Frame-based condition
iLinks	in	t_linkSignalArray	Content of the uplinks, downlinks and coupled signal
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
nodesRole	in	t_portSignalArray	Nodes' role
triggerCount	out	ENTER65536	Number of frames fulfilling the condition
currentTriggerFrame	out	boolean	Current frame fulfils the condition

Table 16.11.: fimTriggerCounter interface.

The fimTriggerCounter component perform its operation thanks to the asynchronous pattern, presented in Sec. 16.2, jointly with the `clkR` signal, so it can safely consult the bit values seen in any link. In this sense, note that the core of this module is responsible for consulting and comparing the bit values transmitted through the link specified in the trigger condition, with the with the bit pattern, also specified in the trigger condition. If both values are consistent, and the role of the target link (`fiLink`) corresponds with the role specified in the trigger condition, the value of `triggerCount` is increased by one, whereas `currentTriggerFrame` is activated. Note that the value of this last signal remains active since the condition is fulfilled until the frame is fully transmitted, that is, until the coupled signal is idle.

fimChecker

The fimChecker module is responsible for checking the conditions involved in the fault-injection configuration and notify the fimOffsetDelayer that the injection can be performed. For this purpose it uses the set of signals listed in Table 16.12. First, the `enable` signal allows to activate it only when fimCfgExecuter is enabled. Second, the `fiCfg` signal contains various constant values that are necessary to check the conditions. Specifically, this module gets from the `fiCfg` the threshold values of the start and end trigger conditions, the specific bits in which the injection must start and end and the fault-injection mode. This values are used jointly with the set of signals conforming the bit state, the signals from the trigger counters, that is, `startTriggerCount`, `endTriggerCount` and `currentTriggerFrame`, and the fault-injection bit count, that is, `endBitCounter`, to perform the checking of the conditions. As concerns the `clkT` signal, it helps to generate the output signal, `inject`, in the correct intant of time, that is, at the beginning of the bit.

Name	I/O	Type	Description
enable	in	std_logic	fimCfgExecuter enable
clkT	in	std_logic	Transmission clock
fiCfg	in	t_fiCfgs	Fault-injection configuration
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
startTriggerCount	in	ENTER65536	Number of frames fulfilling the aim condition
endTriggerCount	in	ENTER65536	Number of frames fulfilling the withdraw condition
currentTriggerFrame	in	boolean	Current frame fulfils the target-frame condition
endBitCount	in	ENTER65536	Number of injected bits
inject	out	boolean	Conditions are fulfilled

Table 16.12.: fimChecker interface.

The fimChecker module uses the asynchronous implementation pattern, jointly with the `clkT` signal to execute the Moore machine depicted in Fig. 16.5. This states machine defines three states called `waiting`, `enabled` and `disabled`. Note, from the figure, that the initial state is `waiting`, whereas the states in which the injection is performed are bold-bordered. The `waiting` state represents the situations in which no injection is performed, but it is possible to progress to `enabled`, in which so. The `enabled` state represents the situations in which the injection must be carried out. From this state it is possible to return to `waiting`, only when executing an iterative injection, or progress to `disabled`. The `disabled` state represents the situations in which, after injecting, the module remains inactive indefinitely.

Note that this states machine is a simplified version of the automaton depicted in Fig. 10.1. In fact, the conditions involved in the transitions are equivalent.

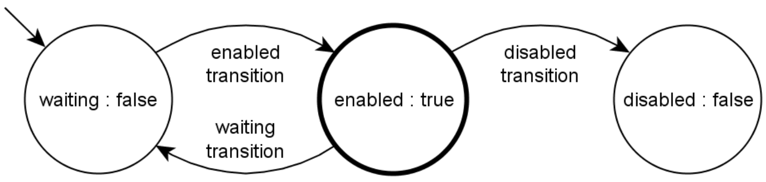


Figure 16.5.: fimChecker implementation states machine.

Finally, as a Moore machine each of these states has a value associated, which is used to feed the output `inject` signal. Specifically, `inject` is true when the current state is `enabled` and false otherwise.

fimOffsetDelayer

The fimOffsetDelayer module is responsible for delaying the fault-injection, in order to carry out the operation of the offset feature. For this purpose this module uses the set of signals listed in Table. 16.13. First, signal `inject` informs this module that the conditions involved in the fault-injection configuration are met and, thus, that the offset count can start. Second, signal `offset`, obtained from the fault-injection configuration, contains the specific value of the offset to be applied. Finally, this module generates the `delayedInject` signal, which acts like the `inject` signal, provided by the fimChecker, but with an additional delay. This delay is calculated in terms of a bit count which is determined by the `clkR` signal.

Name	**I/O**	**Type**	**Description**
clkR	in	std_logic	Reception clock
inject	in	boolean	Injection signal from fimChecker
offset	in	ENTER65536	Offset value
delayedInject	out	boolean	Delayed inject

Table 16.13.: fimOffsetDelayer interface.

The fimOffsetDelayer is constructed from the instantiation of a bit counter, which enables and resets under specific conditions. Specifically, it resets when the `inject` signal is deactivated, whereas it enables when the `inject` signal is activated and the counter has not reached the value of the offset. That is, when the counter is activated, its value cannot exceed the value of the offset. Finally, note that this bit counter is driven by the receiving clock signal. This allows to keep track the value of the counter before the fault-injection value, which is driven by the transmission clock, is generated.

fimManager

The fimManager is a combinational module that generates the outputs of the fimCfgExecuter from the data provided by the fimOffsetDelayer and the fault-injection configuration itself. The set of signals used by this module is shown in Table 16.14. First, the `enable` signal allows to activate this module only when the fimCfgExecuter is activated. Second, signal `inject` contain the value of the previously defined `delayedInject`, generated by the fimOffsetDelayer. This signal notifies the fimManager that the injection must be performed. Third, signals, `value` and `link` contain the value to be injected and the link in which the injection must be carried out, respectively. As shown in the fimCfgExecuter figure, this values are provided by the fault-injection configuration, by means of the `fiCfg` signal. Finally, signals `port0UpValue`..`port2DwValue` are used to output the values the be injected in each link. For instance, signal `port0UpValue` carry the value to be injected in the downlink of Node 0. Note that a pair of signals are output for each port, one for the uplink and one for the downlink. Moreover, these values are not bit values, but `t_fiValue` values. In this sense, the fimValueGenerator, described below, is responsible for translating this information into real bits values.

Name	I/O	Type	Description
enable	in	std_logic	fimCfgExecuter enable
inject	in	boolean	Delayed inject
value	in	t_fiValue	Value to be injected
link	in	t_fiLink	Target link
port0UpValue	out	t_fiValue	Injected value for the uplink of Port 0
port0DwValue	out	t_fiValue	Injected value for the downlink of Port 0
port1UpValue	out	t_fiValue	Injected value for the uplink of Port 1
port1DwValue	out	t_fiValue	Injected value for the downlink of Port 1
port2UpValue	out	t_fiValue	Injected value for the uplink of Port 2
port2DwValue	out	t_fiValue	Injected value for the downlink of Port 2

Table 16.14.: fimManager interface.

The fimManager is constructed from a set of assignment statements like the ones shown in List. 16.9. As can be seen in the figure, for each port two assignments are needed, one for the uplink and one for the downlink. In turn, each of these assignments contains specific logic to evaluate a condition involving the `enable`, `inject` and `link` signals. Specifically, the value to be injected is output only if the fimCfgExecuter is enabled, the fimOffsetDelayer detemines that the injection must be carried out and the link of injection corresponds to the given output. Otherwise, the value output 'value0', which means that no value has to be injected.

```
-- portN
portNUpValue <=
  value when
    enable = '1'   and
    inject = true  and
    link = portNup else
  value0;
```

```
portNDwValue <=
  value when
    enable = '1'   and
    inject = true  and
    link = portNdw else
  value0;
```

Listing 16.9: fimManager implementation pattern.

fimInjValueSelector

A given fimInjValueSelector module is responsible for selecting a single fault-injection value from the set of fault-injection values to be injected in the same link. The interface of this module can be seen in Table 16.15. Specifically, the set of fault-injection values is provided through the `fiValues` signal, whereas the selected fault-injection value is output through the `fiValue` signal.

Name	I/O	Type	Description
fiValues	in	t_cfgExecLinkValues	Set of values for a given link
fiValue	out	t_fiValue	Selected value for a given link

Table 16.15.: fimInjValueSelector interface.

To perform the selection, this module follows the criteria of selecting the fault-injection value of the last valid fault-injection configuration. That is, as if fault-injection configurations were rules to be applied, a later definition overrides a previous one. Thus, injection values of later fimCfgExecuter modules have precedence. This behaviour can be carried out by means of a conditioned assignment statement which emulates a multiplexer. That is, it can be resolved combinationally. List. 16.10 shows an example of this implementation with four fimCfgExecuter modules. Note that 'value0' is a null value, that is, no value has to be injected.

```
fiValue <=
  fiValues(4) when fiValues(4) /= value0 else
  fiValues(3) when fiValues(3) /= value0 else
  fiValues(2) when fiValues(2) /= value0 else
  fiValues(1) when fiValues(1) /= value0 else
  value0;
```

Listing 16.10: fimInjValueSelector implementation.

fimValueGenerator

A given fimValueGenerator module is responsible for generating the specific bits that are injected in the links. The set of signals used to perform this task are listed in Table 16.16. First, the module reads the `fiValue` signal, which contains the value that has to be injected. Depending on that, more information can be necessary. On the one hand, in order to perform the bit-flipping value generation, both the `enable` and the `clkT` signals are required, so, as will be explained below, they feed the sensitive list of a specific process statement. Note that this `enable` signal activates when the NCC is in execution mode. On the other hand, in order to perform the inverse value inject, it is necessary to provide the coupled signal, which can be obtained from the `iLinks` signal. Finally, the result is output through the `fimValue` signal, which belongs to the `t_fimValue` record type. This type is composed of two fields. The first contains a boolean value which indicates whether the injection must be performed, whereas the second contains an `std_logic` value which indicates the bit value to be injected.

Name	I/O	Type	Description
enable	in	std_logic	Execution mode enable
clkT	in	std_logic	Transmission clock
coupledSignal	in	std_logic	Coupled signal
fiValue	in	t_fiValue	Value for a given link
fimValue	out	t_fimValue	Bit value for a given link

Table 16.16.: fimValueGenerator interface.

The fimValueGenerator implementation is divided in two parts, as can be seen in List. 16.11. On the one hand, the first contains two assignment statements which set the values of the two fields of the output. Specifically, the first assignment sets the value of the boolean field, that is, when an injection must be carried out it is set to 'true'. In contrast, the second statement sets the value of the value field, that is, '0' when dominant, '1' when recessive, a specific calculated value when bit-flipping and the inverse of the coupled signal when inverse. On the other hand, the second part of this module contains a process statement, driven by the `enable` and `clkT` signals, so it can generate the value of the bit-flipping, if applicable. This statement wake ups at the beginning of every bit to carry out some actions. On the one hand, if the bit-flipping is applicable, it consults the next value of the bit-flipping, which is contained in the `fiValue` signal. On the other hand, it updates the value of the signal containing the bit-flipping value, that is, `bfValue`, assigned in the output value field. Note that, if the number of bits injected overcomes the number of bits conforming the bit-flipping, this is repeated. For instance, if the bit pattern is specified as "01" and the six bits are injected, the injected bit stream would be "010101".

```
with fiValue.fiType select
  fimValue.bool <=
    false when none,
    true  when others;

with fiValue.fiType select
  fimValue.value <=
    '0'               when stuckAtDominant,
    '1'               when stuckAtRecessive,
    bfValue           when bitFlipping,
    not coupledSignal when inverse,
    '1'               when others;

process (enable, clkT) is
  variable bfIdx : ENTER64;
```

```
begin
  if enable = '0' then
    bfIdx := 0;
  elsif clkT'event and clkT = '1' then
    if fiValue.fiType = bitFlipping then
      bfIdx := bfIdx + 1;
      if bfIdx > (fiValue.bfLong-1) then
        bfIdx := 0;
      end if;
    else
      bfIdx := 0;
    end if;
  end if;
  bfValue <= fiValue.bfValue(bfIdx);
end process;
```

Listing 16.11: fimValueGenerator implementation.

16.6. loggerModule

The loggerModule module implements the operation of the Hub Logger. That is, it captures specific information, during the execution of an experiment, and reports the gathered data during the report phase. Specifically, this module monitors and stores the frames seen in the coupled signal. The signals needed to perform these action, as well as the output of this module, can be seen in Table 16.17. First, the `reset` signal is used to initialize the state of the submodules when the system resets. Second, in order to implement the asynchronous implementation pattern described in Sec. 16.2, signal `clk` is provided. Third, in order to monitor the coupled signal this module needs some signals. Specifically, apart from the coupled signal itself, the `coupledSignal`, it is necessary to provide the set of bit state signals, that is, `frameState`, `frameBitNum` and `isStuffBit`, and the reception clock, `clkR`. Moreover, the information contained in the `nodesRole` helps to determine the source port of the CAN frame being transmitted. Fourth, in order to transmit the report, it is necessary to supply the `coupledSignal` and the set of bit state signals, so this module is able to determine the state of the channel and have a feedback of his own contribution. Moreover, the `clkT` signal must also be provided so the transmission is synchronized bit by bit. Finally, the report is output through the `logContri` signal which, as explained previously is driven to the couplerModule.

The loggerModule module is conformed by three different modules called `logComunicator`, `logGatherer` and `logReporter`, as can be seen in Fig. 16.6. The former is responsible for monitoring the coupled signal and capturing the NCC messages addressed to the loggerModule. Moreover, it also decodes these messages and performs the required tasks. Specifically, it keeps and outputs the current operation mode of the loggerModule, through the `auxLogMode` signal, and instructs the `logReporter` when the report must be transmitted, through the `auxSampleReport` signal. The second, the `logGatherer`, groups all the modules devoted to gather log data in the execution phase. In this implementation we only implement the `logFrameGatherer` which monitors and stores information about the frames being transmitted. This information is output through the `storedFrames` group of signals, so the `logReport` can access to them. Finally, the `logReport` constructs and sends a log report from the information collected by the `logGatherer`,

Name	I/O	Type	Description
reset	in	std_logic	System reset
clk	in	std_logic	System clock
clkR	in	std_logic	Reception clock
clkT	in	std_logic	Transmission clock
coupledSignal	in	std_logic	Coupled signal
nodesRole	in	t_portSignalArray	Nodes' role
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
logContri	out	std_logic	loggerModule contribution

Table 16.17.: loggerModule interface.

when the `logComunicator` instructs to do it. The report is sent to the couplerModule through the `logContri` signal.

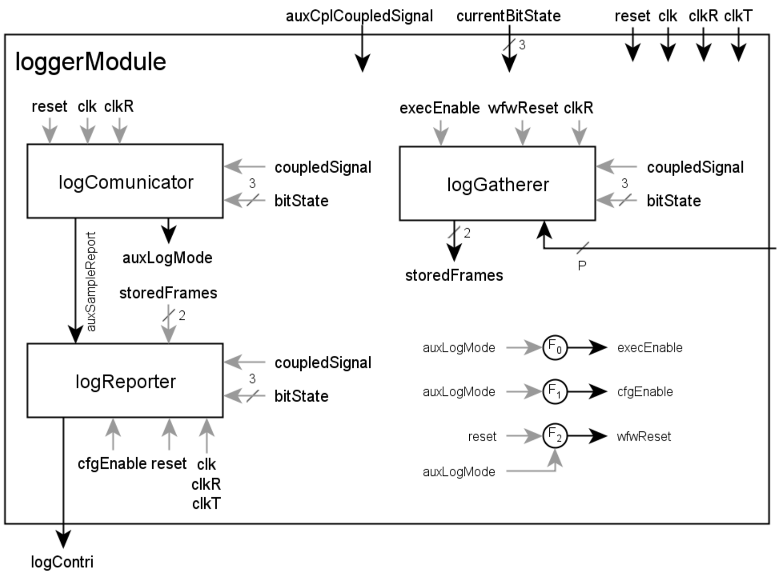


Figure 16.6.: loggerModule implementation diagram.

Note that the `execEnable`, `cfgEnable` and `wfwReset` signals are constructed to enable and reset specific modules under specific conditions. The former, activates while the loggerModule is in execution mode. The second activates while the loggerModule is in configuration mode. Finally, the `wfwReset` activates when the system resets or while the loggerModule is in wait-for-whistle mode. These signals feed the `logGatherer` and the `logReporter` as described next. On the one hand, the `execEnable` and `wfwReset` signals are used by the `logGatherer` to both, activate it and reset it, so its internal state is initialized. On the othe hand, the `cfgEnable` signal is used to active the `logReporter`.

Next, these three modules, as well as their interactions, are described in detail.

16.6.1. logComunicator

The logComunicator collects and decodes the NCC messages addressed to the loggerModule that are transmitted through the coupled signal. Moreover, when this module captures a NCC message destined to the loggerModule, it performs a specific task. On the one hand, if a mode change command is received, this module performs this operation, following the rules specified in 9.1. On the other hand, if a report retrieval command is received, this module instructs the logReporter to transmit the log report. To perform all these tasks it uses the set of signals listed in Table 16.18. First, in order to construct the asynchronous implementation pattern and, thus, synchronize the different modules, signals `reset` and `clk` are provided. Second, in order to monitor the CAN frames transmitted through the coupled signal, apart from the coupled signal itself, `coupledSignal`, this module needs the reception clock, `clkR`, and set of signals conforming the bit state, that is, `frameState`, `frameBitNum` and `isStuffBit`. Finally, it generates two signals called `logMode` and `sampleReport`. On the one hand, the former keeps the current operation mode of the loggerModule. On the other hand, the second is a sample signals that instructs the logReporter to transmit the log report.

Name	I/O	Type	Description
reset	in	std_logic	System reset
clk	in	std_logic	System clock
clkR	in	std_logic	Reception clock
coupledSignal	in	std_logic	Coupled signal
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
logMode	out	t_nccMode	loggerModule operation mode
sampleReport	out	std_logic	Send report

Table 16.18.: logComunicator interface.

The logComunicator is composed of four different modules called `logFrameReader`, `logFrameFilter`, `logCoordinator` and `logCmdInterpreter`, as can be seen in Fig. 16.7. The former captures the CAN data frames and extracts the values contained in the identifier and data fields. After that, it instructs the `logFrameFilter` module to check whether it is a NCC message addressed to the loggerModule. For this purpose, two signals are used. Specifically, signal `lfrSampleFrame` performs the notification, whereas signal `auxId` contains the identifier of the current frame. If a NCC message addressed to the loggerModule is recognized, the `logFrameFilter` module instructs both, the `logCoordinator` and `logCmdInterpreter` to accept the content of the NCC message, that is, the value of the data field of the current frame, provided by the `logFrameReader` through the `auxData` signal. On the one hand, the `logCoordinator` module identifies and executes mode change commands.

Moreover, it is responsible for providing both, the current operation mode, signal `auxLogMode`, and a enter-configuration-mode notifier, signal `auxEnterConfigMode`. On the other hand, the `logCmdInterpreter` module identifies the log report retrieval command and, if so, notifies it by means of the `sampleReport` signal.

Note that the enabling and reset signals provide to the `logCmdInterpreter` are separately generated in order to provoke a specific behaviour. On the one hand, signal `logReset` resets this module when the system resets or the loggerModule enters in configuration mode. On the other hand, signal `cfgEnable` enables this module only when the loggerModule is in configuration mode.

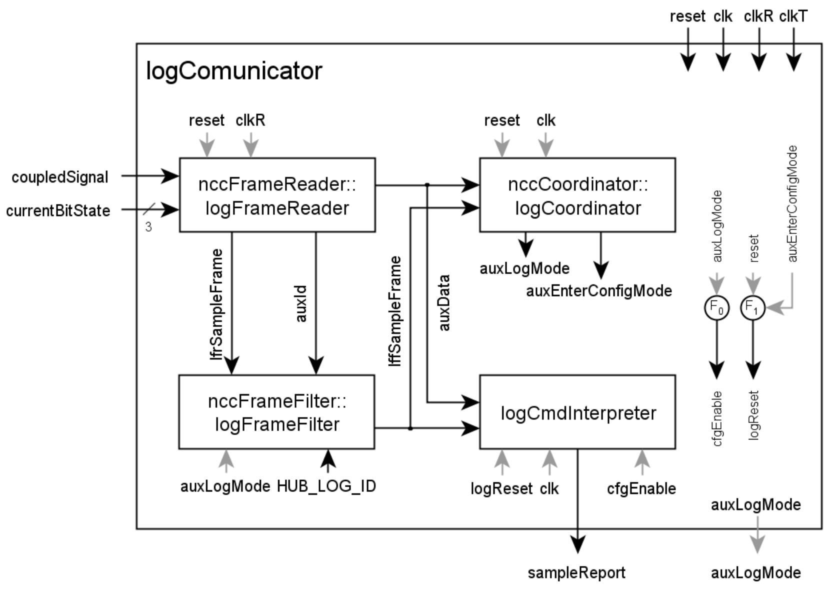


Figure 16.7.: logComunicator implementation diagram.

As concerns the implementation of these four modules, the construction of three of them, the `logFrameReader`, the `logFrameFilter` and the `logCoordinator`, has already been discussed in Sec. 16.5; specifically, they are instantiated from the nccFrameReader, nccFrameFilter and nccCoordinator components, respectively, which, were already used to instantiate some modules in the faultInjectionModule and, thus, they were already described. In contrast, the `logCmdInterpreter` module has its own implementation, which is described next.

logCmdInterpreter

The logCmdInterpreter module is responsible for detecting log report retrieval NCC messages and, when so, for sending a sample report notification to the logReporter, so it constructs and transmits the log report. For this purpose, this module needs the set of signals listed in Table 16.19. All the input signals are devoted to construct the asynchronous implementation pattern explained in Sec. 16.2, so this module is able to synchronize with the logFrameFilter. On the one hand, the `reset` and `clk` signals are placed in the sensitive list. Moreover, the `enable` signal is inserted in the process statement, so this module is only enabled when the loggerModule is in the configuration mode. On the other hand, the `sampleFrame` and `data` signals contain both, the signal notifying a new NCC message available and its content, respectively. Finally, the `sampleReport` is used to send the sample report notification.

Name	I/O	Type	Description
enable	in	std_logic	Configuration mode enable
reset	in	std_logic	System reset
clk	in	std_logic	System clock
sampleFrame	in	std_logic	Notify new NCC message
data	in	std_logic_vector(1 to 8)	Frame data
sampleReport	out	std_logic	Report request

Table 16.19.: logCmdInterpreter interface.

The logCmdInterpreter module, as explained above, is implemented using the asynchronous implementation pattern. Specifically, this module carries out a specific task when being instructed by the fimFrameFilter. A simplified version of the code used to implement this module is shown in List. 16.12. Apart from the code implementing the asynchronous pattern, an enabling condition, as well as the management of `sampleReport` output, has been included.

```
process (reset, clk) is
  <var decl>
begin
  if enable = '0' then
    <var init>
  elsif clk'event and clk = '1' and
          enable = '1'           then
    if sampleFrame = '0' then
      sampleReport <= '0';
    end if;
    if lastSampleFrame = '0' and sampleFrame = '1' and
       data(1 to 3)    = CMD_LOG                  then
      sampleReport <= '1';
    end if;
    lastSampleFrame := sampleFrame;
  end if;
end process;
```

Listing 16.12: logCmdInterpreter implementation.

16.6.2. logGatherer

The logGatherer module is responsible for monitoring and storing specific information during the execution of an experiment. Specifically, this implementation includes a module which logs information about the frames being transmitted. For this purpose, this module uses the set of signals listed in Table 16.20. First, in order to enable this module only when the loggerModule is in execution mode the `enable` signal is provided. Second, to initialize the internal state of this module when the system resets, the `reset` signal is provided. Third, in order to monitor the coupled signal, apart from the coupled signal, `coupledSignal`, the reception clock, `clkR`, and the set of signals conforming the current bit state, that is, `frameState`, `frameBitNum` and `isStuffBit`, must be supplied. Moreover, this module can determine the transmitting node by processing the `nodesRole` signal. Finally, the information about the frames is stored in an array, each of whose components contains the data of a single frame. This array, as well as the number of stored frames, is output throw the `logStoredFrames` and `logStoredFrames_cnt` signals, respectively.

Name	**I/O**	**Type**	**Description**
enable	in	std_logic	Execution mode enable
reset	in	std_logic	System reset
clkR	in	std_logic	Reception clock
nodesRole	in	t_portSignalArray	Nodes' role
coupledSignal	in	std_logic	Coupled signal
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
logStoredFrames	out	t_logStoredFrames	List of stored frames
logStoredFrames_cnt	out	t_logStoredFramesCnt	Number of stored frames

Table 16.20.: logGatherer interface.

As concerns the implementation of this module, it is divided in two parts. on the one hand, the first is an assignment statement that processes the roles of the nodes to determine the transmitting node. On the other hand, the second instantiates the logFrameGatherer module, that is, the module which gathers information about the frames being transmitted. The inputs of the logFrameGatherer module are fed with the inputs signals of the logGatherer itself, except for the `nodesRole` signal, which is replaced with the value of the transmitting node, as explained previously. Note that, this module is designed so other modules gathering different information can be allocated within it.

logFrameGatherer

The logFrameGatherer is responsible for monitoring and storing information about the frames being transmitted through the coupled signal during the execution of an experiment. Specifically, this module captures the identifier, the number of bytes conforming the data, the first byte of the data and the source port of the frame. All this data is output so the logReporter can access to it. To do so, this module needs the set of signals list in Table 16.21 and described in the previous module. First, the `enable`, `reset` and `clkR` signals to construct the process statement that monitors the coupled signal bit by bit. Second, the `coupledSignal` and the set of signals conforming the current bit state to distinguish the current bit. Third, the `txPort` which contains the source port of the frame being transmitted. Finally, signals `logStoredFrames` and `logStoredFrames_cnt` are used to output both, the array containing the information about the frames, and its occupation, respectively.

Name	I/O	Type	Description
enable	in	std_logic	Enable
reset	in	std_logic	Reset
clkR	in	std_logic	Reception clock
coupledSignal	in	std_logic	Coupled signal
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
txPort	in	t_port	Transmission port
logStoredFrames	out	t_logStoredFrames	List of stored frames
logStoredFrames_cnt	out	t_logStoredFramesCnt	Number of stored frames

Table 16.21.: logFrameGatherer interface.

The logFrameGatherer module is implemented by means of a process statement whose sensitive list contains both, the `reset` and `clkR` signals. This can be seen in List. 16.13, where we show a simplified version of the code implementing it. That is, it activates each time a bit can be read. Moreover, the condition involving the `enable` signal forces to only activate when the loggerModule is in execution mode. Each time the process statement activates a different action is performed, depending on the frame field being transmitted. During the identifier, DLC, and data fields the content of the coupled signal is collected, so this data can be stored. As concerns the source port of the transmission, note that, since it can only be known after the arbitration, it is collected during the transmission of the RTR field. Note that, the error entry in the case statement, as well as the final statement updating the `lastValidFrameState` and `lastValidFrameBitNum` ensures that, in case of error, this module can provide the bit and field where the error has been found. Finally, during the interframe field, that is, after transmitting the whole frame, the collected data is stored inside the `logStoredFrames` array. Moreover, the `logStoredFrames_cnt` is increased by one.

Note that, the last valid bit and field is stored even an error is not found. Moreover, if an error is found a new frame indicating the error frame is appended into `logStoredFrames`. This construction allows to collect as much information as possible since, even an error is found all the collected data is still available. In this sense, the Fault Injection Log retriever is responsible for showing the correct information whenever it is necessary. For instance, the last valid bit and field is only shown when an error has been detected, that is, the next frame is an error frame.

```
process (reset, clkR) is
  <var decl>
begin
  if reset = '1' then
    <var init>
  elsif clkR'event and clkR = '1' and
        enable = '1'             then
    case frameState is
      when idField =>
        auxId(frameBitNum) := coupledSignal;
      when rtrField =>
        auxP := txPort;
      when dlcField =>
        auxDlc(frameBitnum) := coupledSignal;
      when dataField =>
        auxData(frameBitnum) := coupledSignal;

      when errorFlag =>
          error := true;
      when interField and frameBitNum=0 =>
        idx := idx + 1;
        logStoredFrames(idx) <= generate_frame(
          auxP, auxId, auxDlc, auxData,
          lastValidFrameBitNum, lastValidFrameState
        );

        if error then
          idx := idx + 1;
          logStoredFrames(idx).valid <= '0';
          error := false;
        end if;
    end case;

    if frameState /= errorFlag and frameState /= errorDelimiter then
      lastValidFrameState  := frameState;
      lastValidFrameBitNum := frameBitNum;
    end if;
  end if;
end process;
```

Listing 16.13: logFrameGatherer implementation.

16.6.3. logReporter

The logReporter module is responsible for constructing and sending the log report made from the information collected by the logGatherer. For this purpose, this module uses the set of signals listed in Table 16.22. First, the `enable` signal is used to permit the operation only in the report phase, that is, when the loggerModule is in configuration mode. Second, the `clk` signal is necessary to construct the asynchronous implementation pattern. This pattern is used to both, capture the notification carried by the `sampleReport` signal and synchronize the internal submodules. Third, in order to receive from the coupled signal, signals `clkR` and `coupledSignal`, and the group of bit state signals are need. On the one hand, `clkR` and `coupledSignal` allow to synchronize and read the coupled signal. On the other hand, the set of conformed by `frameState`, `frameBitNum` and `isStuffBit` signals allow to distinguish the current bit being transmitted through the coupled signal. Fourth, signals `logStoredFrames` and `logStoredFrames_cnt` contain both the list of stored frames and the number of stored frames, respectively. Finally, `contri` signal is used to carry the log report to the couplerModule, that is, it is the loggerModule contribution.

The logReporter module contains two submodules, called `logReportConstructor` and `CANSender`, as can be seen in Fig. 16.8. On the one hand, the first monitors the `sampleReport` signal, generated by the logCmdInterpreter, in order to build the set of frames conforming the report and instruct the `CANSender` to transmit them. For this purpose it needs the log data, which is carried by the `logStoredFrames` and `logStoredFrames_cnt` signals, labelled `storedFrames` in the figure. On the other hand, the `CANSender` synchronizes with the coupled signal in order

Name	I/O	Type	Description
enable	in	std_logic	Configuration mode enable
reset	in	std_logic	System reset
clk	in	std_logic	System clock
clkR	in	std_logic	Reception clock
clkT	in	std_logic	Transmission clock
coupledSignal	in	std_logic	Coupled signal
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
sampleReport	in	std_logic	Report request
logStoredFrames	in	t_logStoredFrames	List of stored frames
logStoredFrames_cnt	in	t_logStoredFramesCnt	Number of stored frames
contri	out	std_logic	loggerModule contribution

Table 16.22.: logReporter interface.

to transmit a given CAN frame. The content of this frame is provided by means of a set of input signals which, in this case, are mostly provided by the `logReportConstructor`.

Between this two modules a two-way synchronization handshake is established. On the one hand, the `logReportConstructor` generates the `sampleData` signal, which instructs the `CANSender` to transmit a specific frame. On the other hand, the `CANSender` generates the `senderBusy`, which informs the `logReportConstructor` that a frame transmission is in process and, thus, that it cannot accept any transmission instruction.

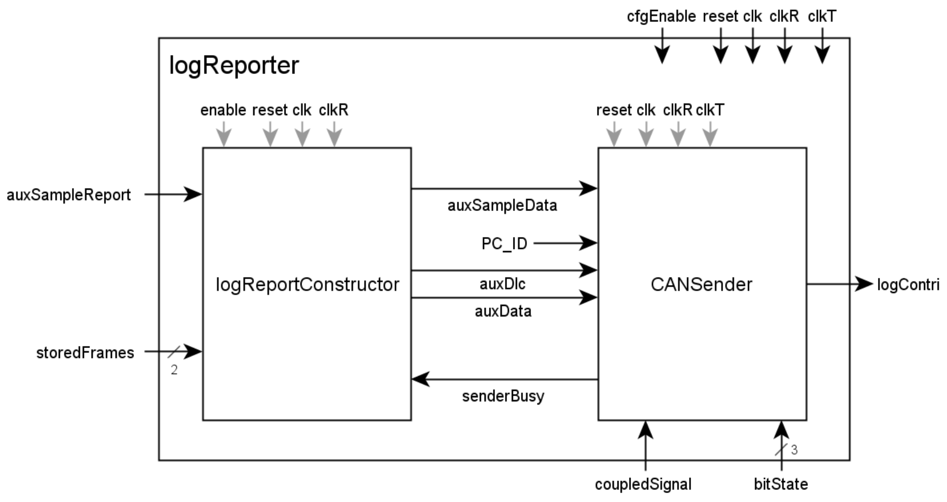


Figure 16.8.: logReporter implementation diagram.

Next this two modules are explained more in depth.

logReportConstructor

The logReportConstructor is responsible for monitoring the `sampleReport`, building up the set of frames conforming the log report and instruct the CANSender to transmit them. For this purpose, this module uses the list of signals shown in Table 16.23. First, in order to enable this module only in the report phase, signal `enable` is provided. Second, signal `reset` is used to initialize the internal state of this module when the system restarts. Third, signal `clk` is used to construct the asynchronous implementation pattern, so the `sampleReport` signal is monitored. Fourth, `logStoredFrames` and `logStoredFrames_cnt` signals provide the log data, that is, in this case, the set of frames seen in the coupled signal during the experiment. Fifth, the `senderBusy` signal notifies this module that the CANSender is transmitting a frame and, thus, that it cannot accept a frame transmission request. Finally, the `sampleData` signal is used to perform a frame transmission request, which forces the CANSender to carry out two specific tasks. On the one hand, it acquires the content of the frame, that is, the identifier, the DLC and the data. Note that the first is a constant value containing the NCC ID of the FIMS, whereas the second and the third are provided by this module through the `dlc` and `data` signals. On the other hand, the CANSender is forced to transmit the specific frame.

Name	**I/O**	**Type**	**Description**
enable	in	std_logic	Configuration mode enable
reset	in	std_logic	System reset
clk	in	std_logic	System clock
clkT	in	std_logic	Transmission clock
sampleReport	in	std_logic	Report request
logStoredFrames	in	t_logStoredFrames	List of stored frames
logStoredFrames_cnt	in	t_logStoredFramesCnt	Number of stored frames
senderBusy	in	std_logic	CANSender is busy
dlc	out	ENTER9	Frame DLC
data	out	t_data	Frame data
sampleData	out	std_logic	Notify frame available

Table 16.23.: logReportConstructor interface.

The logReportConstructor is divided in two parts, each of which is constructed from a process statement. On the one hand, the first monitors the `sampleReport` signal, provided by the logCmdInterpreter. On the other hand, the second constructs and sends the report to the CANSender, frame by frame. This two parts interact in order to perform the full operation of this module. Next, this interaction, as well as the internal construction of each part is described more in-detail.

The first part follows the asynchronous implementation pattern proposed in Sec. 16.2, in order to monitor the `sampleReport` signal. When the logCmdInterpreter instructs to transmits the log report, that is, it activates `sampleReport`, this part activates a signal called `active`. This signal is used by the second part to enable its operation. At this point, the first part is locked, waiting for the second to end. When the second part ends its operation, it activates the `reportDone` signal, which indicates to the first part that another report request can be accepted.

As concerns the second part, it is constructed from a process statement which is driven by the `clkT` signal, as seen in List. 16.14. When signal `active` is activated and the CANSender is idle, this executes a states machine to construct and send all the frames conforming the report, as well as the `end-of-log` NCC message. Since this version of sfiCAN only gathers data about the traffic, this state machine is only composed of three states called `logFrames`, `logEOL` and `none`. Note that, the state spaces can be adapted to hold additional gathered data in the report. The first state is used to construct and send the set of NCC messages conforming the report of the gathered frames. This is done by encapsulating the information of a given gathered frame in a single NCC message, which, in turn, is encapsulated in a CAN frame. When all the gathered frames are transmitted, the states machine progresses to the `logEOL` state, in which an `end-of-log` NCC message is constructed and sent. Finally, the states machine progresses to the `none` state, in which the internal variables as well the outputs are initialized so this part is prepared for another report request. Note that, this state contains the specific statement that activates the `reportDone` signal so the first part is unlocked.

```
process (reset, clkT) is
  <var decl>
begin
  if reset = '1' or active = '0' then
    <var init>
  elsif clkT'event and clkT = '1' then
    if senderBusy = '1' then
      <var init>
    else
      sampleData <= '1';
      case lrcState is
        when logFrames =>
          if logStoredFrames_cnt > 0                   and
             logStoredFramesIdx <= logStoredFrames_cnt then
            <construct log frame message>
            logStoredFramesIdx := logStoredFramesIdx + 1;
          else
            lrcState := logEOL;
          end if;
        when logEOL =>
          <construct en of log message>
          lrcState := none;
        when none =>
          <var init>
      end case;
    end if;
  end if;
end process;
```

Listing 16.14: logReportConstructor implementation.

CANSender

The CANSender module is responsible for transmitting the frames conforming the log report, provided by the logReportconstructor. For this purpose, this module uses the set of signals listed in Table 16.24. First, the `reset` signal is used to initialize the internal state of this module when the system resets. Second, clock signals `clk`, `clkR` and `clkT` are used to construct the set of process statements that, as explained below, performs the full operation of this module. Third, in order to monitor the bits transmitted it is necessary to provide the coupled signal and the bit state group, that is, `coupledSignal`, `frameState`, `frameBitnum` and `isStuffBit` signals. Fourth, the specific content to be transmitted, that is, the identifier, the number of bytes and the data, is carried by the `id`, `dlc` and `data` signals, respectively. Fifth, in order to notify that a new frame has to be transmitted, signal `sampleData` is used. Sixth, signal `busy` is activated by this module when a frame is being transmitted and, thus, it cannot accept any transmission request. Finally, signal `contri` carries the frames generated, that is, the loggerModule contribution.

Name	I/O	Type	Description
reset	in	std_logic	System reset
clk	in	std_logic	System clock
clkR	in	std_logic	Reception clock
clkT	in	std_logic	Transmission clock
coupledSignal	in	std_logic	Coupled signal
frameState	in	estadoGlobalFrame	Current frame field
frameBitNum	in	ENTER64	Current bit number
isStuffBit	in	std_logic	Current bit is stuff
id	in	t_id	Frame identifier
dlc	in	ENTER9	Frame DLC
data	in	t_data	Frame data
sampleData	in	std_logic	Notify frame available
busy	out	std_logic	CANSender is busy
contri	out	std_logic	loggerModule contribution

Table 16.24.: CANSender interface.

The CANSender module is divided in three interconnected parts, each of which is constructed from a single process statement. The first monitors the `sampleData` signal, generated by the logReportConstructor, in order to determine when a new frame has to be transmitted. The second performs the bit by bit transmission. Finally, the third monitors the coupled signal to both, determine whether the CANSender has win the arbitration and calculate the CRC. Next, each of these parts is deeply described.

The first part is constructed from a process statement driven by the `clk` signal. This part monitors the `sampleData` and, when it is activated, performs two different tasks. On the one hand, it captures the content of the frame, provided by signals `id`, `dlc` and `data`. On the other hand, it instructs the second part to transmit the frame. To do so, it notifies, by means of the `auxBusy` signal, that the content of a new frame is ready to be processed. At this point, this part stands locked until the second part notifies that the frame has been successfully transmitted.

The second part is constructed from a process statement driven by the `clkT` signal. This allows to generate the bit values synchronously with the coupled signal. As explained previously, this part is activated when the first part enables the `auxBusy` signal. Moreover, in case of losing the arbitration this part waits until the bus is idle. As concerns the bit transmission, it is output through the `contri` signal. Note that the specific value of most of the bits is already known, since they are static or can be easily determined. For instance, the value of the RTR bit is always

’0’. However, the identifier, the dlc, the data, and the CRC have to be specifically specified. While the first three are provided externally, the CRC can be obtained thanks to the third part of this module which, as explained below, calculates this value in parallel. Additionally, this part generates the values of the stuff bits thanks to the `isStuffBit` signal. Finally, when a given frame is successfully transmitted, this part activates the `active` signal, which unlocks the first part, so another frame transmission request can be served.

The third part is constructed from a process statement driven by the `clkR` signals. This part performs two different actions each time a bit is received. On the one hand, it determines whether the this module has win the arbitration and, thus, if it can transmit the current frame. For this, it monitors the values transmitted by the second part and the values carried by the coupled signal, during the arbitration. In case of winning the arbitration this part notifies the second part, by means of the `isTx` signal, that it can continue transmitting. On the other hand, this part captures the values seen in the coupled signal in order to calculate the CRC of the current frame being transmitted. The CRC value is then used by the second part to be transmitted as part of the frame. Note that the value of the CRC is provided through the `auxCrc` signal.

17. Nodes implementation

The CAN nodes used in the sfiCAN infrastructure have been inherit from the previous ReCANcentrate prototype, see [Gessner, 2010]. In this sense, we have modified three of these nodes to be compliant with CANcentrate infrastructure, that is, to use just one communication port. Then, specific logging mechanisms, that is, the Node Loggers, has been added. Next, the hardware and the basic internal architecture of these nodes is described.

17.1. Nodes hardware platform

As can be seen in Fig. 17.1, each node is constituted by two different parts, called *Node Core* and *I/O module*, that are attached to each other by means of a flat IDE cable.

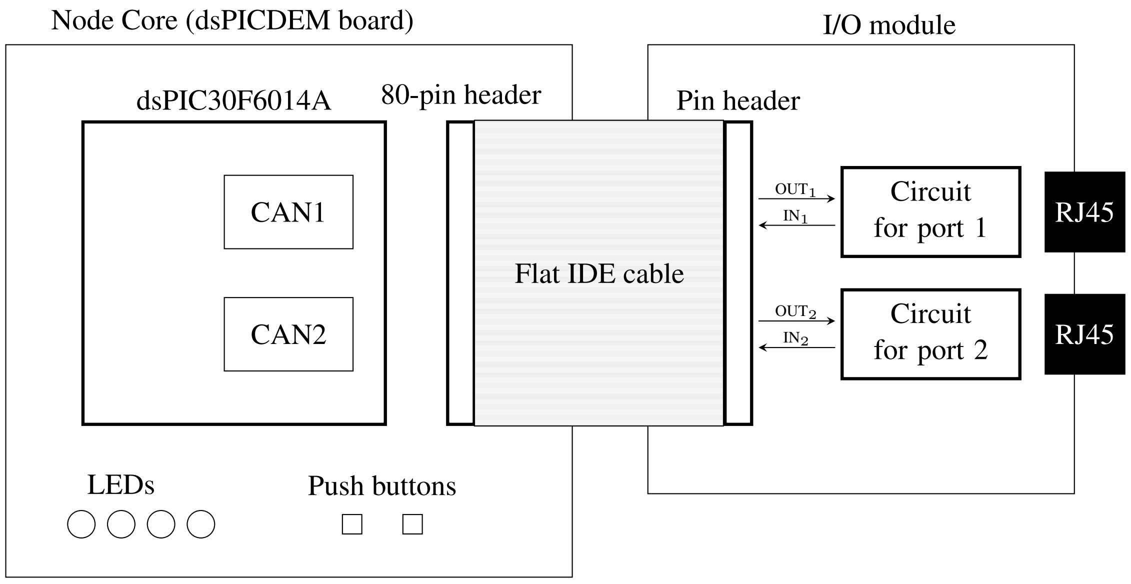


Figure 17.1.: Node implementation diagram. (The figure is based on a figure by Gessner [2010]).

On the one hand, the Node Core is implemented using a dsPICDEM 80-pin Starter Development Board [Microchip Technology, 2006], whose central component is a dsPIC30F6014A microcontroller [Microchip Technology, 2011]. Note that, although this microcontroller was chosen for its two embedded CAN controllers, in sfiCAN only one of them is used. Additionally, the dsPICDEM board provides four LEDs, two push buttons and an 80-pin header that gives easy access to the pins of the microcontroller, as well as the pins of the embedded CAN controllers.

On the other hand, the I/O module is implemented on a wire-wrap board from COTS components. Note that, since this module is intended to connect the node to two hubs, it provides a replicated circuit port. Again, in sfiCAN only one of these circuits is used. A given circuit is constructed using two CAN Philips PCA82C251 [Philips, 1996] transceivers, following the connection schema explained in Sec. 5.1. As concerns the `IN` and `OUT` wires, they are respectively connected to the reception and transmission pins of a CAN controllers, though the flat IDE cable. Finally, each circuit port is connected to a RJ45 port used to communicate with the hub by means of a RJ45 cable.

Note that the increasing of the bit-rate to 1 Mbps involved some modifications in both the nodes' hardware and software. On the one hand, we installed an external 16 Mhz oscillator on each node. On the other hand, we had to edit the driver to reconfigured the values of some of the device parameters. As concerns the bit-rate parameters, they were adapted in the `can_controller.c` file, whereas the system clock parameters were modified in the `device_config.h` file.

17.2. Nodes internal logic

As concerns the internal logic of the nodes, we follow the design presented in Sec. 12. First, a set of two interconnected pieces of software implement the workload of the node and the Node Logger. Second, the CAN driver allows this pieces of software to interact with the hardware easily. The source code of these parts can be found as an appendix in App. E. Finally, the hardware includes the CAN device, that is, the CAN controller of the microcontroller. However, note from Fig. 17.2 that the final implementation has some differences.

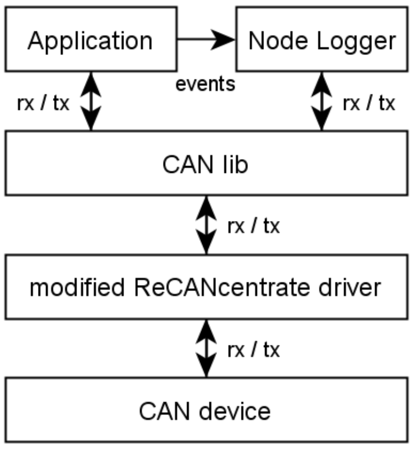


Figure 17.2.: Basic node's internal architecture.

First, note that the interactions between the Node Logger and the rest of parts is different from the ones seen in Sec. 12. After studying the way of implementing the Node Logger, we realize that the independence of this piece of software cannot be accomplish easily. Thus, we have implemented a proof of concept Node Logger, which simplifies various aspects of the design. On the one hand, during the execution phase the application is responsible for monitoring and, when a new event is detected, informing the Node Logger. In turn, the Node Logger evaluates the event and, if corresponds, it is stored. This evaluation is necessary since a receiving frame event can contain an NCC message addressed to the Node Logger. Note that, this implementation of the Node Logger depends directly on the application to perform its operation during the execution phase. On the other hand, when sfiCAN is not in the execution phase, the Node Logger holds the thread of execution and avoids the execution of the application. Moreover, it is responsible for monitoring the traffic and processing the NCC messages addressed to it.

Second, we appended an additional layer at the user level, the *CAN lib*, which allows to receive and transmit frames without taking into account the specific details involved in the operation of the driver. Moreover, it also allows to implement high-level functions in order to perform specific driver interaction. For instance, it is possible to implement the single-shot transmission feature, by means of a non-blocking transmitting function. The source code implementing this layer can be found as an appendix in App. E.

Finally, instead of using a regular CAN driver, we use a modified version of the ReCANcentrate driver. Specifically, as introduced previously, we disabled the driver's media management for replicated stars. Moreover, we avoid the use of multiple CAN devices. This fact implied performing minor modifications on the code of the driver.

18. Fault Injector Management Station (FIMS) implementation

The Fault Injector Management Station is a set of utilities that allows the user to interact with the different NCCs. Specifically, it provides with mechanisms to prepare, execute and report an experiment. As concerns the PC in which the FIMS is executed, it is a regular computer using Linux in which we installed a PCI CAN controller.

18.1. FIMS architecture

The FIMS architecture in based on the one proposed by SocketCAN, explained deeply below. Specifically, it is divided in three parts, as shown in Fig. 18.1. The first is the hardware, which involves the CAN device used in the PC. The second is the Linux kernel space, which is divided in three parts, the CAN device driver, the CAN stack and the socket layer. Finally, the third part is the user space, which includes the set of FIMS utilities, a CAN lib developed specifically for the these utilities and a set of tools described later, in the Sec. 18.3.1.

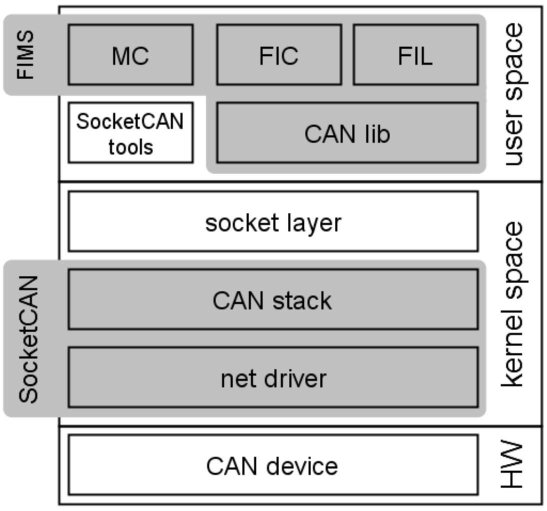


Figure 18.1.: Fault Injector Management Station CAN implementation.

18.2. FIMS hardware platform

As explained previously, the PC has a CAN device installed, so the FIMS can communicate with the sfiCAN prototype through the CAN network. Specifically, this device is a PCAN-PCI dual channel card developed by PEAK System-Technik GmbH. This card includes two CAN channels compliant with the CAN specifications 2.0A and 2.0B, each of which interfaces with a 9-pin D-Sub. Internally, each channel uses a Philips SJA1000 CAN controller [Philips, 2000] running at 16MHz. As concerns the transceivers, it uses the Philips PCA82C251 [Philips, 1996] which are compatible with the Philips PCA82C250 transceivers used in the rest of the prototype. The only difference between these two types of transceivers is that the PCA82C251 is focused on industrial applications, due to it electrical characteristics.

The manufacturer of the PCAN-PCI device provides drivers and tools for both operating systems, Linux and Windows, so it can be installed in most of the PCs. Due to our needs we use the Linux driver which, in fact, it is an open source driver. This feature allow us to set up some specific low-level parameters if it is needed, in the future. The Linux driver provides a high-level C library called `libpcan` which allows to initialize the device, as well as transmit and receive CAN frames. Moreover, this driver is compatible with SocketCAN which, as will be explained in the next section, allows to handle a given CAN device as network device.

As concerns the connection of the PC with the hub, we use the configuration shown in Fig. 18.2. As can be seen in this figure, the connection is divided in three parts. First, a DB9 to RJ45 adapter is plugged to one of the PCAN board channels. We set up this adapter so the CAN_H and CAN_L signals can be used. For this, we connected the DB9 pins labelled ”2” and ”7” to the RJ45 pins labelled ”2” and ”4” respectively. Moreover, we installed a 120 Ohm resistor between the RJ45 pins labelled ”2” and ”4” in order to terminate the CAN connection. Second we use a regular UTP (Unshielded Twisted Pair) Category 5/5e/6 ethernet cable to connect the DB9 to RJ45 adapter to the hub PC port. Note that this cable is identical the ones used to connect the nodes to the hub. Finally, in the hub, the CAN_H and CAN_L signals are driven from the RJ45 port to the CAN transceiver, just as explained in Sec. 11.4. Note that the hub already provides a 120 Ohm resistor in order to terminate the CAN connection.

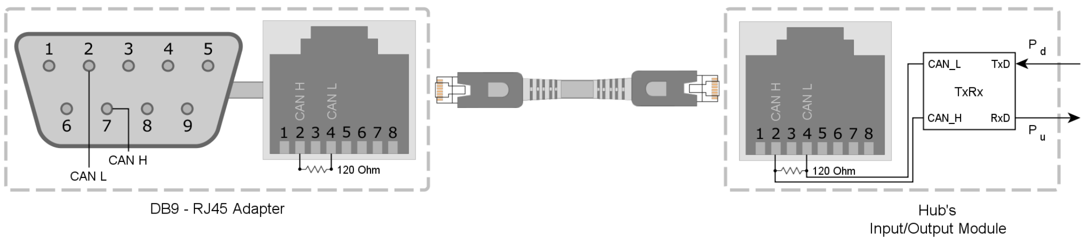


Figure 18.2.: PC to Hub connection setup.

18.3. SocketCAN

SocketCAN is a set of open source CAN drivers and a CAN networking stack for Linux. The idea of Socket is to treat a CAN device as a regular network device. Traditional CAN drivers for Linux use the character-based model, which only allows one process to access to the CAN controller at a time. In contrast, the schema followed by SocketCAN overcomes this limitation and provides new features. Fig. 18.3 shows the two different schemas used to access to a CAN controller.

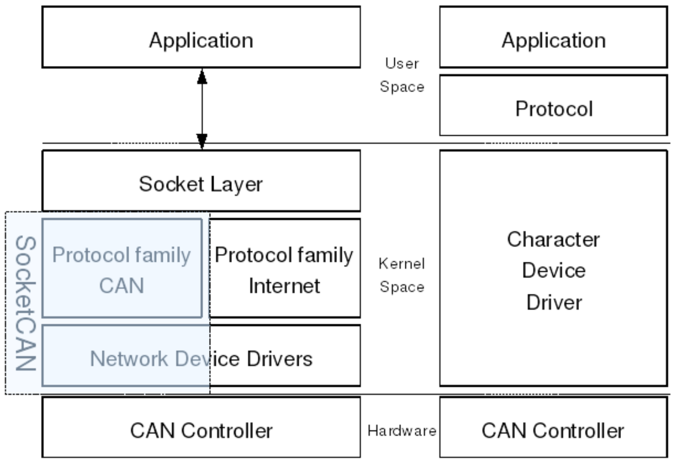


Figure 18.3.: SocketCAN approach versus traditional approach. (Reprinted from the Wikipedia Socket-CAN article).

On the one hand, the left of the figure shows the SocketCAN approach. As explained previously, this approach divides the architecture in three parts called hardware, kernel space and user space. First, the hardware involves the CAN device. Second, the kernel space includes the network driver and the CAN stack, that is, SocketCAN, and a socket layer. This last layer is used when working with network devices in order to manage different virtual connections [Wikipedia, 2012b]. Finally, The user space is conformed by the set of applications that use the CAN device. As concerns the interaction between layers, the CAN device interacts directly with its network driver, which, in turn, interacts with the CAN network stack. Analogously to a regular network stack, the CAN stack has to be accessed through sockets. Finally, the applications are responsible for creating, using and removing these sockets. Note that, although the CAN driver is a specific network driver developed by the manufacturer of the hardware, the CAN network stack is developed by the Linux community, in fact, it is part of the Linux kernel. Note also, from the figure, that this approach is analogous to the one used by the TCP/IP protocol.

On the other hand, the character-based approach proposes to develop a CAN driver in the kernel that interacts with a specific protocol in order to communicate with the application. In this case, the protocol layer allows to access to the driver using a specific interface that, normally, differs when using other character-based drivers. This fact can provoke portability issues.

18.3.1. SocketCAN tools

There are various ways of accessing to the sockets to transmit and receive CAN frames through a CAN controller. The easiest way consists in using the set of commands that SocketCAN provides (these commands are also referred to as *tools* in the SocketCAN documentation). The most used commands are `cansend`, to send a specific CAN frame, `cangen`, to send various CAN frames following a specific schema, and `candump`, to capture and show all the received CAN frames. Although complete information about these tools can be found in the man pages when installing socketCAN, we next briefly explain their syntax and their main parameters.

First, as shown in List. 18.1, to invoke the `cansend` command the CAN interface, as well as the CAN frame, has to be specified. On the one hand, the name of the interface depends on the driver and the alias used. On the other hand, as concerns the frame, its identifier and its data have to be separated by the '#' sign. Moreover, the identifier must be constructed from 3 hexadecimal values, whereas the data must be constructed from a set of pairs of hexadecimal value, being 8 the maximum number of pairs.

```
$ cansend <CAN interface> <identifier>#<data>
```

Listing 18.1: cansend syntax.

Second, as shown in List. 18.2, to invoke the `cangen` command it is only necessary to specify the CAN interface, which can be done identically as in the `cansend` command. However, additional parameters can be set up in order to configure the content and the transmission of the frames. There are various of these parameters but here we only describe the most important from our point of view. First, parameter `-g` allows to configure the delay between frames, that is, the specific time that the command will wait before sending the next frame. Note that this value must be specified in milliseconds. Second, paramaters `-I`, `-L` and `-D` allow to specify the identifier, the number of the data bytes and the data contained in the frames, that is, their content. On the one hand, as concerns the identifier and the data parameters, they can be specified as static values, random values and increasing values. When any of these values are specified as a static values, they must be constructed from hexadecimal values, just like in the `cansend` command. Additionally, note that, when any of them is not specified it is taken as a random value. On the other hand, the number of data bytes can be used when specifying a random or an increasing data value to set its size; otherwise, when the data is specified as a static value, it is useless. Third, parameter `-n` can be used to limit the number of frames to be transmitted. Finally, parameter `-i` allows to transmit frames even when the buffer is in an error state. This is very useful when it is known that errors occur in the channel, for instance, because of a fault injection, but the device must not block.

```
$ cangen <CAN interface>
         [-g <milis> -I <identifier> -L <natural> -D <data> -n <natural> -i]
```

Listing 18.2: cangen syntax example.

Finally, the `candump` command allows to monitor a set of CAN interfaces. As can be seen in List. 18.3 it is only necessary to specify the set of interfaces the user wants to monitor. However, note that this command allows to specify the value of additional parameters in order to customize its output.

```
$ candump <CAN interfaces>
```

Listing 18.3: candump syntax example.

18.4. FIMS utilities

As can be seen in Fig. 18.1, the FIMS is composed by three different utilities called Mode Changer, Fault-Injection Configurator and Fault-Injection Log retriever. The source code of all these utilities, as well as the specifically developed packages used are presented as an appendix in App. F. Next, we describe the purpose and the construction of each one of them.

18.4.1. Mode Changer

The objective of the Mode Changer is to provide the user with mechanisms to change the operation mode of the different NCCs. In principle, the user can perform this mode changes manually by sending the adequate NCC messages using SocketCAN commands. However, in order to make it easy the set up of a fault-injection experiment, specific and frequently used sequence of commands have been collected within a set of different scripts we call Mode Changer.

There are two different mode change scripts called `ecm` and `eem`. On the one hand, the first is used to force all NCCs to enter in configuration mode. This script can be used to initialize the mode of the NCCs or end the execution of an experiment. On the other hand, the second forces all the NCCs to enter first in wait-for-whistle mode and then in execution mode. This script can be used to start the execution of an experiment, after configuring the Central Fault Injector.

Specific mode change commands can be constructed by means of the NCC message encoding presented in Sec. 13.6.2. Once constructed, the frame containing the NCC message can be transmitted using the `cansend` command, whose syntax was previously described, in Sec. 18.3.1. List. 18.4 shows an example of a custom mode change command. Specifically, this command instructs the logger of the Node 3 to enter in idle mode.

```
$ cansend can0 003#21.
```

Listing 18.4: Mode change command example.

18.4.2. Fault-Injection Configurator

The Fault-Injection Configurator is a piece of software that allows to check, encode and transmit a fault-injection specification from the PC to the Central Fault Injector located in the hub. Given the complexity of this software, it has been implemented as a C++ application instead of as various scripts.

In order to perform its operation this application relies on a set of different libraries called GLib, CAN lib, Bit lib, Constants lib, File spec lib and Argument lib. The first, the GLib

(Gnome library) provides a wide set of useful functions to develop C applications and libraries. Specifically, we use its key-value file parser which helps to read and write text files containing groups of key-value pairs. Second, in order to deal with the CAN interfaces we have developed a library called CAN lib which provides CAN sockets that rely on SocketCAN, see Fig. 18.1. Third, the Bit lib provides specific bit operations so the content of the NCC messages can be set of extract easily. Fourth, the Constants lib contains the set of values used in the NCC protocol, that is, the encoding of the identifiers and the messages. Fifth, the File spec library contains the set of strings that can be found in a fault-injection specification file. Finally, the Argument lib includes a set of functions that help to read the arguments passed through the shell.

The Fault-Injection Configurator accepts a set of different shell arguments, in order to set up specific parameters or modify its behaviour. Specifically, it allows to specify the source file containing the fault-injection specification, `-f=<file>`, as well as the CAN interface used to transmit it, `-i=<iface>`. Additionally it is possible to just perform a fault-injection specification file checking by setting the `-c` flag. List. 18.5 shows how to process the `IMO.cfg` file to check and transmit the fault-injection specification through the `can0` interface.

```
$ ./fic -f=IMO.cfg -i=can0
processing file ...
   3 group(s) found!

initializing 'can0' socket ...

checking file ...
   0 error(s) found!

processing file ...
Ending...
Done!
```

Listing 18.5: Example of Fault-Injection Coordinator's execution.

18.4.3. Fault-Injection Log retriever

The Fault-Injection Log retriever is a piece of software that allows to retrieve the log report from the set of loggers. Analogously to the Fault-Injection Configurator, it has been implemented as a C++ application. However, it does not use the same set of libraries. Specifically, it relies on the set of libraries called CAN lib, Bit lib and Constants lib. These libraries have already been introduced in the previous section.

As concerns the syntax use to call the Fault-Injection Log retriever, in contrast with the Fault-Injection Configurator it does not have any shell argument. In turn, it uses a predefined CAN interface. Moreover, it supposes that all the loggers are active. Thus, if a specific logger does not respond it waits indefinitely. A simplified example of the execution of this application can be seen in List. 18.6.

```
$ ./fil
initializing 'can0' socket ...
retrieving log data...
---------
-- HUB --
---------
Ok 030 [1] 00 port0
Er 030 [1] 01 port0 (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 18.6: Example of Fault-Injection Log retriever's execution.

The information provided depends on the type of NCC logger. On the one hand, the Hub Logger collects the set of frames received. For each frame it stores:

- Whether it is a valid frame, `Ok` when so and `Er` when not.
- Three hex characters containing the value of identifier field.
- Between brackets, the value of the DLC field, that is, the number of data bytes carried into the frame.
- Two hex characters containing the value of the first byte of the data field.
- The source port of the frame.
- If it is not a valid frame, the field and the bit where the hub has detected the error.

On the other hand, Node Loggers collect the set of frames received and transmit, as well as the value of the error counter when their value changes. Specifically, for each frame it stores:

- Two decimal values containing the number of the frame.
- Whether the frames has been transmitted, `tx`, or received, `rx`.
- Three hex characters containing the value of identifier field.
- Between brackets, the value of the DLC field, that is, the number of data bytes carried into the frame.
- Two hex characters containing the value of the first byte of the data field.

Part V.

Functional verification

19. Testbed setup

As explained, the prototype of sfiCAN is composed by a CAN-compliant hub in which we connect three CAN nodes and a PC. The hub is responsible not only for coupling all the contributions, but for injecting errors in the links of the nodes and logging specific data from a central point of view of the system. As concerns the nodes, they implement a regular node software in order to behave like an ordinary CAN node. Moreover, they also include additional logic to log specific data from a distributed point of view. Finally, the PC provides the user with tools to control the progress of the experiments, configure the faults to be injected by the hub and to retrieve and show the log reports from the hub and the nodes. Note that, in order to corroborate that the injector is fast-enough to inject in every single bit, the bit-rate of the system was set up at the maximum CAN speed, that is, 1 Mbps. Regarding the cables, we used UTP (Unshielded Twisted Pair) Category 5/5e/6 ethernet cables cables of about 1 meter-long in our tests. Note that the bit-rate has a direct impact on the maximum cables length. More information about this aspect can be found in [Barranco, 2010].

This system has been extensively tested to check its correct operation. For this purpose we have built an experimental platform which, as can be seen in Fig. 19.1, is composed of the sfiCAN infrastructure and an oscilloscope. Next we describe the specific details involved in the assessment.

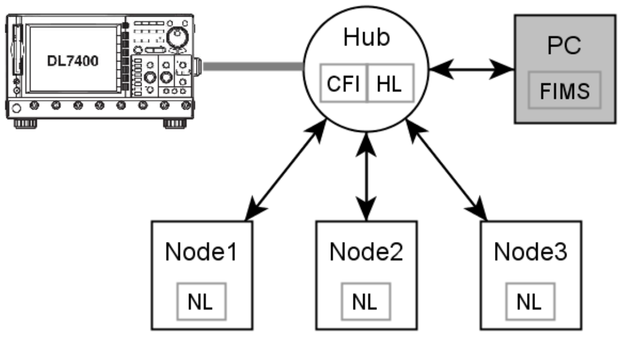


Figure 19.1.: Experimental platform used to assess sfiCAN.

First, in order to recreate the behaviour of a regular CAN node, we developed two node programs, a CAN transmitter and a CAN receiver. On the one hand, the CAN transmitter instructs the node to transmit a set of frames with a predefined identifier and data values. Specifically, the value of the identifier is static, whereas the value of the data field increases each time a frame is transmitted. Note that, when needed, it is possible to make the value of the data field static. On the other hand, the CAN receiver program collects the CAN frames and acknowledges them, only if they are valid. Additionally, in both cases the data value transmitted/received is output by means of the 4 leds installed in the node.

Second, a Yokogawa DL7440 digital oscilloscope is used to both check the values of the bits conforming the CAN frames and monitor debugging signals. Note that both values can be obtained from the hub, specifically, from its input and output pins. This oscilloscope has CAN detection mechanisms, that is, it is able to determine and identify the specific bits conforming a CAN frame. Finally, apart from executing the FIMS, the PC has several tools to send and capture CAN frames, just as seen in Sec. 18.3.1. In this sense, it has been used to both, generate a specific workload in the channel, and monitor the set of frames transmitted and, thus, help in the debugging process. Note that the CAN interface the PC provides allows to be set up in *listen-only* mode. This operation mode avoids the transmission of acknowledgements and error frames, so the PC is able to monitor the traffic without interfering in the communication.

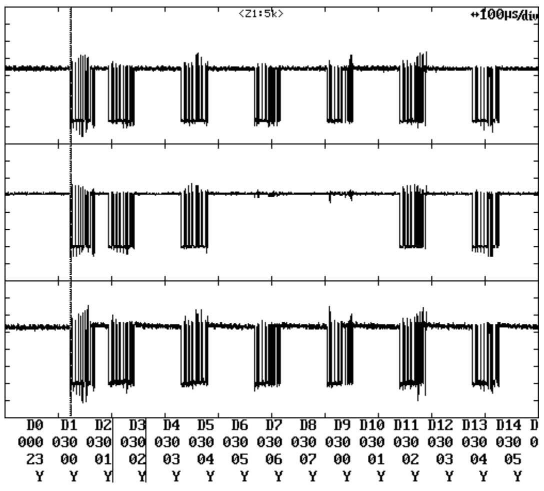
The steps followed to carry out the tests are the ones presented in Sec. 7. First, we design and specify the fault scenario. A given fault scenario is defined by means of the traffic transmitted and the set of faults. Thus, it is necessary not only to define the involved faults but to define the behaviour of the nodes. Second, we configure the sfiCAN components according to fault-injection specification defined in the previous phase. This implies the set up of the Central Fault Injector, the selection of the node program and the initialization of the logger components, that is the Node Loggers, the Hub Logger, the oscilloscope and the PC. Note that this last task is done automatically by the Node Loggers and the Hub Logger, whereas the oscilloscope and the PC must be initialized manually. Third, we start the execution of the test by means of the so-called starting-whistle NCC message. During the test nodes execute its software and generate traffic, while sfiCAN injects faults in accordance to the conditions of the fault injection specification. Moreover, jointly with the oscilloscope and the PC, it also collects data regarding the experiment. Finally, we stop the execution of the test and we retrieve the data that sfiCAN collected during the execution of the fault scenario. Then, we use these data to check the system's behaviour.

Finally, note that, each node program of these tests forces a node to keep the same role during the whole execution, that is, they are always transmitters or receivers. Thus, the role conditions in the aim, withdraw and target-frame conditions do not makes sense in this scope. Consequently, unless it is explicitly specified, these fault-injection parameters are omitted in the tests.

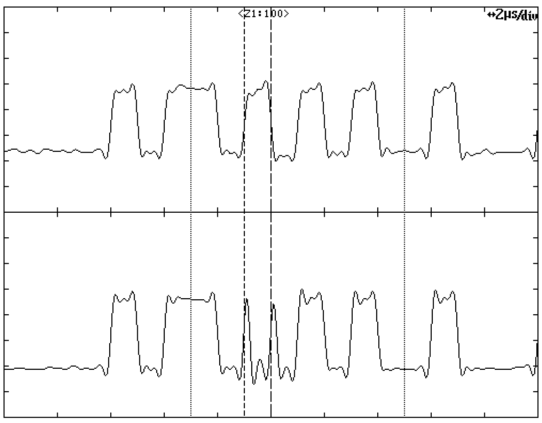
20. Tests

This section describes a set of five tests that verify different features of sfiCAN. On the one hand, each of the first three tests use one of the given fault-injection modes, that is, single-shot, continuous and iterative. On the other hand, the other two show how to recreate complex scenarios in which various bits are injected in different links.

For each of these tests an oscilloscope screenshot is provided, so the injection can be verified at a bit level. As explained in the previous section, the hardware used to do so is able to determine and identify the specific bits conforming a CAN frame. The two screenshots shown in Fig. 20.1 highlight some of the information provided by this oscilloscope.



(a) Multiple-frame oscilloscope screenshot example.



(b) Single-frame oscilloscope screenshot example.

Figure 20.1.: Oscilloscope screenshots examples.

The first figure shows a multiple-frame view of the values carried by three different links. Specifically, the first signal represents the coupled signal, whereas the second and the third represent the downlink of two different nodes. The set of frames can be identified as those perturbations of when channel is idle, that is, when it contains a recessive value. As concerns the dashed vertical line, it marks the start of frame of the first frame. As will be explained later, this line, jointly with others, are used to distinguish different fields and bits. Additionally, the part standing below contains specific information about the frames carried in the first signal, in this case the coupled signal. Specifically, it shows the position of the frame, the value of its identifier and data fields and whether the it has been acknowledged. For instance, the first frame, the one marked with the dashed vertical line, contains ’000’ and ’23’ in the identifier and data fields respectively. Moreover, since it has a ’Y’, at least one node has acknowledged the frame. Specifically, this is a starting-whistle NCC message sent by the FIMS with the purpose of starting the test, see Sec. 13.6.

The second figure shows the value of specific bits within one frame. For this purpose, it uses a higher zoom level. Moreover, since only part of one frame has to be shown, the information about the frames has been omitted. Note that, it only contains two signals. The first carries the values of the coupled signal, whereas the second carries the values of a specific downlink to a node. As concerns the vertical lines, the dotted ones determine the bits conforming an specific field, whereas the dashed lines distinguish a specific set of bits. In this case, the dotted lines determine the data field, whereas the dashed ones indicate a specific injected bit.

Note that both figures contain a number in the top right corner. This value indicates the duration of the divisions of the X axes. Thus, it should not be considered as the bit or frame length.

Additionally, when necessary, we present log data to corroborate that the error scenarios occur as expected. For this, we simplify the data gathered by the log system, that is, the Hub Logger and the Node Loggers. For this we organize the log data in a table, where each column represents one of the network components and the rows represent the different frames. Table 20.1 shows an example of a typical log after it is simplified. The first column numerates the frames so they can be chronologically ordered. The second column contains all the information collected by the hub. Specifically, for each frame, it informs whether the hub has detected any error during its transmission (`Ok` when so and `Er` when not) and the content of the frame. Specifically, the frame is printed following the `iii#dd` format, which represents the frame's identifier (`iii`) and the value it carries in its data field (`dd`). Additionally, if the frame is not successfully transmitted, the field can contain the specific bit and the frame field where the error was encountered. For instance, `eof(0)` refers to the first bit of the End Of Frame field. In this sense, note that bits are numbered from starting from 0. The third, fourth and fifth columns contain the local information collected by the nodes. Specifically, for each frame it indicates whether the frame has been sucessfully/unsuccessfully (`Suc`/`Uns`) transmitted/received (`Tx`/`Rx`). Moreover, it also contain information about the content of the frame related to the specific event. Identically to the hub log data, the field contains a specific string with the `iii#dd` format which represents the identifier and the data carried by the frame.

As concerns the information shown in Table 20.1, note that we captured the first five frames, which were transmitted by Node0 and they were successfully delivered to Node2. However, as can be extracted from the value contained in the data field of the frames, Node1 does not received the third nor the fourth frame. Additionally, the information extracted from the Hub Logger indicates that no error was found and, thus, that we are dealing with an inconsistent message omission scenario.

Finally, note that, as explained, all the log information shown here has been simplified and, thus, some details were omitted. The real log data that can be obtained from the loggers is discussed in Sec. 18.4.3.

	Hub	Node0	Node1	Node2
1	Ok 123#00	Tx Suc 123#00	Rx Suc 123#00	Rx Suc 123#00
2	Ok 123#01	Tx Suc 123#01	Rx Suc 123#01	Rx Suc 123#01
3	Ok 123#02	Tx Suc 123#02	—	Rx Suc 123#02
4	Ok 123#03	Tx Suc 123#03	—	Rx Suc 123#03
5	Ok 123#04	Tx Suc 123#04	Rx Suc 123#04	Rx Suc 123#04

Table 20.1.: Log example.

20.1. Bit-flipping

This test recreates a basic scenario in which the value of one bit is altered in the downlink of a receiving node. As a result, the receiving node should detect the error and globalise it by means of an error frame. For this purpose we use only two nodes, Node 0 acts as the transmitter, whereas Node 1 acts as the receiver. Regarding the frames transmitted, each of them contains a natural value which increases by one each time a frame is successfully transmitted. This allows us to identify easily the corrupted frame and its retransmission.

The injection is performed in the third bit of the data field of the second frame. Note that this field is the most probable for a bit-flip due to its size. To carry out this injection only one fault-injection configuration is needed, the one shown in List. 20.1. As can be seen in this listing, we inverse (line 2) the value seen by Node 1 (line 3). Since only one bit is injected we select the single-shot fault-injection mode (line 4), in which the aim and fire conditions fulfilment decide the start of the injection. On the one hand, the aim condition fulfils after the beginning of the second frame. That is, after seeing a dominant value (line 6) when the channel is idle (line 7) in the coupled signal (line 8) two times (line 9). On the other hand, the fire condition fulfils when the current bit is the third bit (line 12, note that bits are numbered starting from 0) of the data field (line 11). Finally, the injection remains active 1 bit (line 14).

```
 1  [fault injection 1]
 2  value_type  = inverse
 3  target_link = port1dw
 4  mode        = single-shot
 5
 6  aim_filter = 0
 7  aim_field  = idle
 8  aim_link   = coupled
 9  aim_count  = 2
10
11  fire_field  = data
12  fire_bit    = 2
13
14  cease_bc = 1
```

Listing 20.1: Bit-flipping fault-injection specification.

The oscilloscope's screenshot shown in Fig. 20.2 exposes the result of the injection. On the one hand, the part above shows the value of the coupled signal that is, the signal that each node should receive through its downlink. On the other hand, the part below shows the value of the downlink of Node 1. As concerns the vertical lines, the dotted ones determine the bits conforming the data field, whereas the dashed ones determine the injected bit. Note that the recessive value of the coupled signal contrasts with the dominant value of the injected signal.

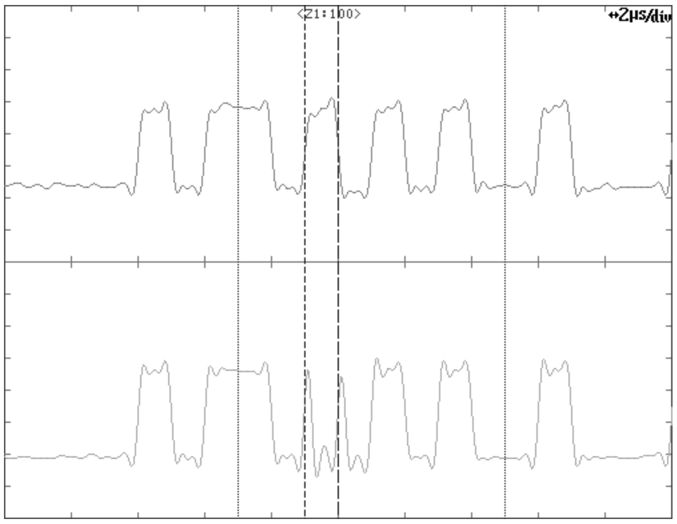


Figure 20.2.: Oscilloscope screenshot of the bit-flipping test.

Finally, the information collected from the loggers corroborates the scenario. On the one hand, the Hub Logger detects an error in the ack delimiter, that is, where a node detecting a CRC error starts, when transmitting the second frame. On the other hand, the Node Loggers detect an increasing of the error counters when transmitting/receiving the second frame. However, due to the retransmission the frame can be delivered correctly.

20.2. Recessive Downlink Message Omission (RDMO)

This experiment shows how to use the continuous fault-injection mode to inject a stuck-at-recessive fault at the reception signal of Node 1, so that it omits receiving some consecutive frames broadcast by Node 0. Additionally, Node 2 is not injected, so it can acknowledge all the frames. Specifically, the injected frames are the third and the fourth. Finally, as concerns the transmitting frames, they contain a bit count so the message omission can be verified easily.

To carry out this injection it is necessary to specify the fault-injection configuration shown in List. 20.2. That is, a recessive value (line 2) is forced in the downlink of port 1 (line 3). Since the injection lasts various frames the continuous fault-injection mode is selected (line 4). In this mode it is necessary to define both the start and the end by means of a frame plus a bit condition. On the one hand, the injection starts after the beginning of the second frame. Specifically, the aim condition activates after detecting two Start Of Frames, that is, two (line 9) dominant values (line 6) when the coupled signal (line 8) is idle (line 7). As concerns the fire condition, it activates at the first bit (line 17) of the interfield (line 16). On the other hand, the injection ends after seeing a frame containing '3', that is, after the fourth frame is transmitted. Specifically, the withdraw condition activates after seeing one time (line 14) the value '3' (line 11) in the data field (line 12) at the coupled signal (line 13). As concerns the cease condition, it disables the injection in the same point that the fire condition, that is, in the first bit (line 20) of the interfield (line 19).

```
 1  [fault injection 1]
 2  value_type      = recessive
 3  target_link     = port1dw
 4  mode            = continuous
 5
 6  aim_filter      = 0
 7  aim_field       = idle
 8  aim_link        = coupled
 9  aim_count       = 2
10
11  withdraw_filter = 0000.0011
12  withdraw_field  = data
13  withdraw_link   = coupled
14  withdraw_count  = 1
15
16  fire_field      = interfield
17  fire_bit        = 0
18
19  cease_field     = interfield
20  cease_bit       = 0
```

Listing 20.2: Recessive downlink message omission fault-injection specification

The result of this injection is shown in Fig. 20.3. It shows the bit stream the oscilloscope has captured at the receiving signal of each node. Note that the screenshot has been edited to make easier the understanding of the scenario. On the one hand, at the left we annotated the specific node which receives the signal. On the other hand, we identify each frame by surrounding it with a rectangle and by aligning the frame data below.

The events at the experiment occur as follows. First, in order to start the experiment, a starting-whistle frame is sent by the FIMS. This frame, is labelled D0 and does not take part in the experiment itself. Second, the transmitter starts to send numbered frames. Third, after the second frame, labelled D2, Node 1 receives nothing but a recessive value. Finally, after detecting a frame containing '3' in the data field, frame labelled D4, the injection stops and Node 1 can receive the rest of the frames. Since the frames are masked completely Node 1 does not detect nor globalises any error and, thus, no retransmission is performed.

As can be seen in Table 20.2 the information gathered from the loggers corroborates the scenario. First, Node 0 and the hub do not detect any error in the transmission. Second, Node 1 does not receive the third nor the fourth frame. Finally, Node 2 receive all the frames.

20.3. Iterative Integrity Error (IIE)

The Iterative Integrity Error test shows, by means of the fault-injection iterative mode, the sf-iCAN's capability of injecting faults that may provoke integrity errors. As explained in Sec. 5.2, integrity errors are those that lead some nodes to accept spurious frames. The events at the experiment occur as follows. First, Node 0 transmits a predefined frame many times. Second,

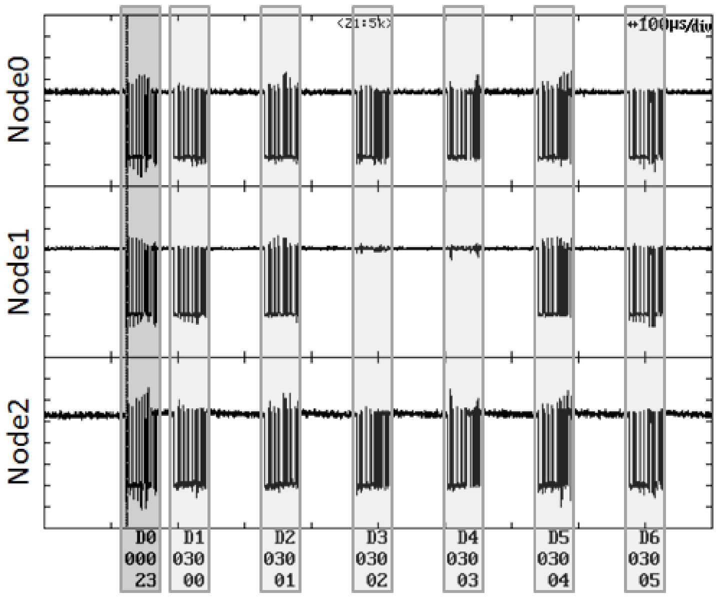


Figure 20.3.: Oscilloscope screenshot of the RDMO test.

	Hub	Node0	Node1	Node2
1	Ok 123#00	Tx Suc 123#00	Rx Suc 123#00	Rx Suc 123#00
2	Ok 123#01	Tx Suc 123#01	Rx Suc 123#01	Rx Suc 123#01
3	Ok 123#02	Tx Suc 123#02	—	Rx Suc 123#02
4	Ok 123#03	Tx Suc 123#03	—	Rx Suc 123#03
5	Ok 123#04	Tx Suc 123#04	Rx Suc 123#04	Rx Suc 123#04

Table 20.2.: Log of the RDMO test.

Node 1 is forced to receive an altered version of this frame the second, third and fourth time it is transmitted. Finally, Node 2 receives all frames correctly. Note that, in order to provoke the integrity error in Node 1, we change the value of the data and the CRC fields on the frame transmitted through the downlink to Node 1, in such a way that the CRC matches the altered content of the data field.

It is noteworthy that, in order to fit the altered values into the original values, this test assumes a previous knowledge of the content of the data and CRC fields of the frame transmitted by Node 0. Specifically, this frame contains the hexadecimal value `010` and `AA` in the identifier and data fields, respectively. Thus, taking into account the CRC calculation rules, the CRC contains the hexadecimal value `54EA`. As concerns the injected values, the data field is corrupted with the hexadecimal `5A` value, that is, all the bits of the first nibble are inversed, whereas the CRC value is corrupted with the hexadecimal `7385` value. Additionally, the iterative mode is set up, so the injection is performed after the beginning of the second frame until three altered frames have

have been seen in the downlink to Node 1.

As can be seen in List. 20.3 and 20.3, this scenario is generated thanks to two fault-injection configurations, one for each error that must be injected. Both are very similar, since they inject a bit pattern (line 2) in the downlink to Node 1 (line 4) using the iterative mode (line 5). On the one hand, the first fault-injection configuration performs the data field injection, that is, it overwrites the value of the first 4 bits of the data field (lines 17 to 21) with the `0101` binary value (line 3), yielding a data value of 5A. As concerns the specific frames involved in the injection, the aim condition fulfils after the beginning of the second frame, that is, after seeing a dominant value (line 7) when the coupled signal (line 9) is idle (line 8) two times (line 10), whereas the withdraw condition fulfils after seeing three times an injected frame, that is, after seeing the `0101` bit pattern (line 12) in the data field (line 13) being transmitted exclusively to Node 1 (line 14) three times (line 15). Note that the way this withdraw condition is defined demonstrates how the sfiCAN's fault injector can be set up to generate a frame-based condition in terms of the injected values. On the other hand, the second fault-injection overwrites the value of the CRC (lines 17 to 21) with a precalculated value that corresponds to the modified data previously injected, that is, the hexadecimal value `7385` (line 3). The conditions determining the set of frames to be injected, are similar to the ones specified for the previous fault-injection. The only difference is the withdraw condition, which, although is defined in terms of the CRC injection fulfils in the same frame. Specifically, it activates after seeing the hexadecimal value `7385` (line 12) in the CRC field (line 13) being transmitted to Node 1 (line 14) three time (line 15).

```
1  [fault injection 1]
2  value_type    = pattern
3  value_pattern = 0101
4  target_link = port1dw
5  mode        = iterative
6
7  aim_filter = 0
8  aim_field  = idle
9  aim_link   = coupled
10 aim_count  = 2
11
12 withdraw_filter = 0101
13 withdraw_field  = data
14 withdraw_link   = port1dw
15 withdraw_count  = 3
16
17 fire_field  = data
18 fire_bit    = 0
19 fire_offset = 0
20
21 cease_bc    = 4
```

Listing 20.3: First IIE fault-injection configuration.

```
1  [fault injection 2]
2  value_type    = pattern
3  value_pattern = 111.0011.1000.0101
4  target_link = port1dw
5  mode        = iterative
6
7  aim_filter = 0
8  aim_field  = idle
9  aim_link   = coupled
10 aim_count  = 2
11
12 withdraw_filter = 111.0011.1000.0101
13 withdraw_field  = crc
14 withdraw_link   = port1dw
15 withdraw_count  = 3
16
17 fire_field  = crc
18 fire_bit    = 0
19 fire_offset = 0
20
21 cease_bc = 15
```

Listing 20.4: Second IIE fault-injection configuration.

Fig. 20.4 shows a screen capture of the second frame transmitted by Node 0 ans received by nodes 1 and 2. Note that, it has been altered to both identify the bit streams and delimit the specific fields of the frame. In this sense, the first row corresponds to the uplink of Node 0, the second row to the downlink of Node 1, and the third row to the downlink of Node 2. As concerns the values received by nodes 1 and 2, note that, although they differ in the data and CRC field, that is, the frame received by Node 1 is different from the one transmitted by Node 0 and received by Node 2, none of them signals an error. This fact is corroborated by the data received from the Node Loggers, which is shown in Table 20.3. Each row corresponds to a frame. In turn, each entry indicates whether a given node transmitted or received that frame, whether it did so successfully or not, and what the identifier and data content transmitted/received was. As the table shows, Node 1 received and accepted data that was never transmitted by a node.

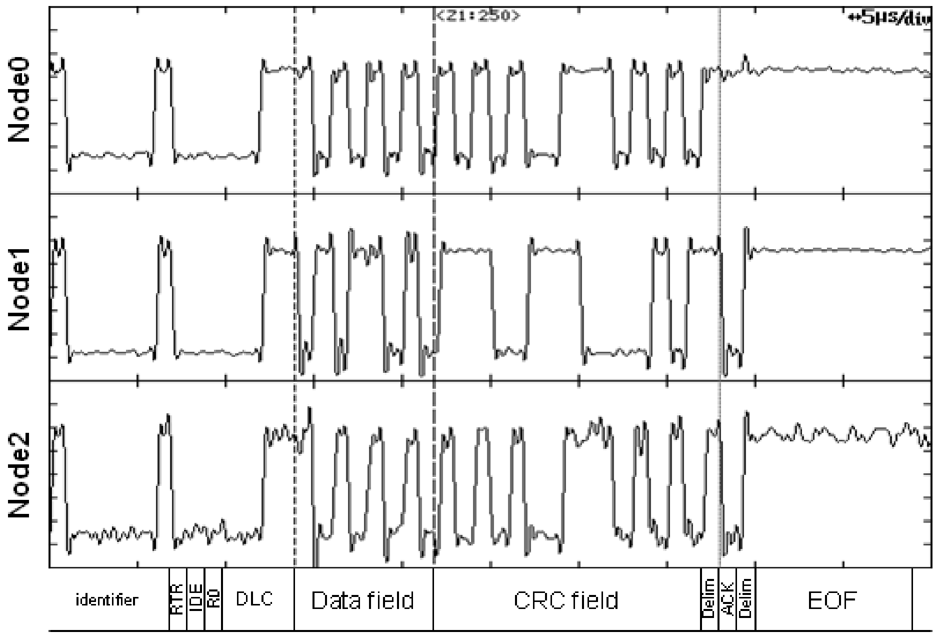


Figure 20.4.: Oscilloscope screenshot of the IIE test.

	Hub	Node0	Node1	Node2
1	Ok 010#AA	Tx Suc 010#AA	Rx Suc 010#AA	Rx Suc 010#AA
2	Ok 010#AA	Tx Suc 010#AA	Rx Suc 010#5A	Rx Suc 010#AA
3	Ok 010#AA	Tx Suc 010#AA	Rx Suc 010#5A	Rx Suc 010#AA
4	Ok 010#AA	Tx Suc 010#AA	Rx Suc 010#5A	Rx Suc 010#AA
5	Ok 010#AA	Tx Suc 010#AA	Rx Suc 010#AA	Rx Suc 010#AA

Table 20.3.: Log of the IIE test.

20.4. Inconsistent Message Omission (IMO)

In this section we recreate the inconsistence scenario presented in 3.3. That is, an inconsistency scenario in which, due to a set of errors, a node discards a specific frame. In this test Node 0 acts as the transmitter, whereas Node 1 and Node 2 perform the operation of receivers X and Y, respectively. The specific fault scenario is depicted in Fig. 20.5. Specifically, two errors are provoked. On the one hand, a dominant bit is injected in the last-by-one bit of End Of Frame field received by node labelled X. On the other hand, in order to mask the error flag generated by the previous injection, a recessive bit is forced in the next bit received by the transmitter.

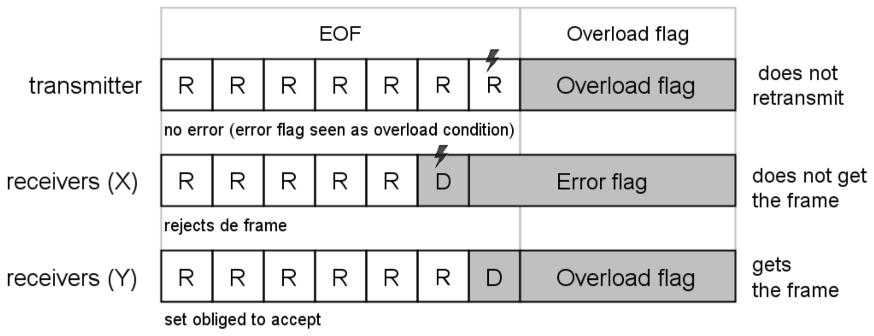


Figure 20.5.: IMO scenario.

The fault-injection specification designed to force this scenario is shown in List. 20.5 and 20.6. It is composed of two fault-injection configurations, one for each injected bit.

```
1  [fault injection 1]
2  value_type          = dominant
3  target_link         = port1dw
4  mode                = single-shot
5
6  aim_count           = 2
7  aim_filter          = 001.0010.0011
8  aim_field           = id
9  aim_link            = coupled
10
11 fire_bit            = 5
12 fire_field          = eof
13
14 cease_bc            = 1
```

Listing 20.5: First IMO fault-injection configuration.

```
1  [fault injection 2]
2  value_type          = recessive
3  target_link         = port0dw
4  mode                = single-shot
5
6  aim_count           = 2
7  aim_filter          = 001.0010.0011
8  aim_field           = id
9  aim_link            = coupled
10
11 fire_bit            = 6
12 fire_field          = eof
13
14 cease_bc            = 1
```

Listing 20.6: Second IMO fault-injection configuration.

The first injection is specified in the fault-injection configuration labelled “fault injection 1”. Specifically, the configuration indicates that the error to be injected is a dominant (line 2) and the link where to inject is the downlink of port 1 (line 3). Additionally, since only one bit has to be injected, the single-shot fault-injection mode is set (line 4). The start of the injection is specified by means of a frame and bit condition, that is, the aim and fire conditions. On the one hand, the aim fulfils after the identifier (line 8) matches a specific pattern (line 7) seen on the coupled signal (line 9) two times (line 6). As explained in previously, this allow us to inject in the second frame. On the other hand, the fault is injected in the sixth bit (line 11) of the end of frame field (line 12). Finally, the injection ends after a bit count of one (line 14).

The second injection is specified in the fault-injection configuration labelled "fault injection 2". It is very similar to the one previously presented, with the only difference of the value, the place and the specific bit to be injected. Specifically, a recessive bit (line 2) is injected in the downlink of port 0 (line 3) in the seventh bit of the end of frame field (lines 11 to 14). Since the aim condition is identical to the one used in the first fault-injection configuration the specific frame that is injected is also the same, the second one that is transmitted by Node 0.

The result of the injection can be seen in Fig. 20.6, which shows the last part of the frame in an oscilloscope screenshot. We annotated at its left with the role played by the node corresponding to each of the shown signals. The first signal from the top, shows the signal on the downlink to the Y receiving node. The centre shows the signal on the downlink to the transmitting node. Finally, the bottom shows the signal on the downlink to the X receiving node. Moreover, at the top we annotated the bit fields of the transmitted CAN frame as seen by the node labelled Y. Note that the two vertical dashed lines at the centre of the image indicate the last bit of the end of frame field.

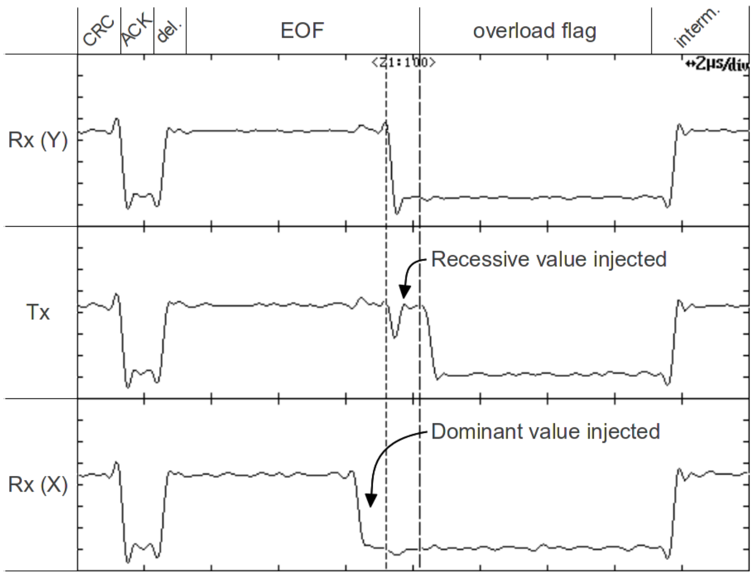


Figure 20.6.: Oscilloscope screenshot of the IMO test.

Finally, as can be seen in Table 20.4, the Node Loggers corroborate the scenario. First, Node 0, does not detect any error when transmitting. Second, Node 1, that is, X, detects an error in the second frame, thus, it is discarded. Finally, Node 2, that is, Y, as well as the hub, detect an error in the seventh bit of the End Of Frame field. However, due to the CAN rules Node 2 is forced to accept the frame.

	hub	Node0	Node1	Node2
1	Ok 123#00	Tx Suc 123#00	Rx Suc 123#00	Rx Suc 123#00
2	Er eof(6)	Tx Suc 123#01	—	Rx Suc 123#01
3	Ok 123#02	Tx Suc 123#02	Rx Suc 123#02	Rx Suc 123#02
4	Ok 123#03	Tx Suc 123#03	Rx Suc 123#03	Rx Suc 123#03
5	Ok 123#04	Tx Suc 123#04	Rx Suc 123#04	Rx Suc 123#04

Table 20.4.: Log of the IMO test.

20.5. Unfair Primary Error (UPE)

This experiment demonstrates the potential of the iterative fault-injection mode to create a complex scenario involving faults that go beyond the capacity of diagnosis of the Primary Error (PE) CAN mechanism (see Sec. 2.2.6. In this experiment Node 0 transmits a set of numbered frames, while some errors are injected in the downlink signal of Node 1 and 2, during certain frames. These errors are specifically designed to reduce the effectiveness of the PE. The complete scenario is depicted in Fig. 20.7. First, an error is injected, locally, in the downlink of Node 1 during the reception of the CRC. As a result, this node starts signalling the error at the first bit of the End Of Frame field, causing the other nodes to detect a format error and to start signalling it at the subsequent bit. After sending its error flag, Node 1 should have to monitor a dominant bit belonging to the delayed error flags sent by nodes 0 and 2, and thus detect the PE. However, a recessive bit is injected in its downlink for this bit, so that it does not detect the PE and increases its REC only by 1. In contrast, although Node 2 should not have to detect a PE, it is forced to do so by injecting a dominant bit just after it sends its own error flag. Thus, in addition to increasing its REC by 1, it further increases it by 8.

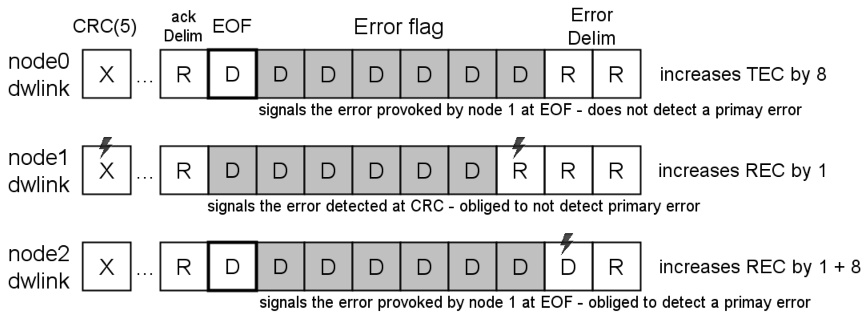

CRC(5)
ack Delim
EOF
Error flag
Error Delim
node0 dwlink
X ... R D D D D D D D R R
increases TEC by 8
signals the error provoked by node 1 at EOF - does not detect a primay error
node1 dwlink
X ... R D D D D D D R R R
increases REC by 1
signals the error detected at CRC - obliged to not detect primary error
node2 dwlink
X ... R D D D D D D D D R
increases REC by 1 + 8
signals the error provoked by node 1 at EOF - obliged to detect a primay error

Figure 20.7.: UPE scenario.

The behaviour described above is reproduced repeatedly using the iterative mode to force Node 2, exclusively, to reach the error-passive state. Note that to force this behaviour is unfair, since Node 1 is affected by two local errors, whereas Node 2 is only affected by one. Also note that in order to not monopolize the bus with errors, these are injected in alternate frames. For this purpose, the transmitter does not retransmit any frame encountering errors, instead, it tries to transmit a frame with the next natural value. This operation mode is called *single-shot transmission* and further information can be found about it in App. B.7. The fault-injection specification forcing this scenario is shown in List. 20.7, 20.8 and 20.9. First, the fault-injection configuration labelled “fault injection 1” provokes the CRC bit-flip in the signal seen by Node 1. For this purpose, it inverses (line 2) the bit value transmitted through the downlink of port 1 (line 3). Moreover, as explained it uses the iterative mode (line 4), whose start and end are defined by means of two frame and bit conditions. Specifically, the injection is performed in the fifth bit (lines 16 and 18) of the CRC field (line 15) after the second frame (lines 6 to 9) when the frame transmitted through the coupled signal (line 13) contains a dominant value in the last bit’s byte (line 11) of the data field (line 12). Second, the fault-injection configuration labelled “fault injection 2” forces a recessive value (line 2) in the downlink of Node 1 (line 3) 7 bits after the ACK delimiter (lines 15 to 19). As concerns the frames involved in the injection, they are defined identically as done in the previous fault-injection configuration. Finally, the fault-injection configuration labelled “fault injection 3” forces a dominant value (line 2) in the downlink of Node 2 (line 3) eight bits after the ACK delimiter (lines 15 to 19). Again, the frames involved in the injection are the same ones previously defined.

Note that thanks to the target frame parameter it is possible to inject only in those frames containing an odd value in their data field. Moreover, since no withdraw condition is set in any fault-injection configuration, the injection is carried out indefinitely.

```
1   [fault injection 1]
2   value_type          = inverse
3   target_link         = port1dw
4   mode                = iterative
5
6   aim_filter          = 0
7   aim_field           = idle
8   aim_link            = coupled
9   aim_count           = 2
10
11  target_frame_filter = xxxx.xxx1
12  target_frame_field  = data
13  target_frame_link   = coupled
14
15  fire_field          = crc
16  fire_bit            = 4
17
18  cease_bc            = 1
```

Listing 20.7: First UPE fault-injection configuration.

```
[fault injection 2]
value_type          = recessive
target_link         = port1dw
mode                = iterative

aim_filter           = 0
aim_field            = idle
aim_link             = coupled
aim_count            = 2

target_frame_filter = xxxx.xxx1
target_frame_field   = data
target_frame_link    = coupled

fire_field          = ackDelim
fire_bit            = 0
fire_offset         = 7

cease_bc            = 1
```

Listing 20.8: Second UPE fault-injection configuration.

```
[fault injection 3]
value_type          = dominant
target_link         = port2dw
mode                = iterative

aim_filter           = 0
aim_field            = idle
aim_link             = coupled
aim_count            = 2

target_frame_filter = xxxx.xxx1
target_frame_field   = data
target_frame_link    = coupled

fire_field          = ackDelim
fire_bit            = 0
fire_offset         = 8

cease_bc            = 1
```

Listing 20.9: Third UPE fault-injection configuration.

The result of the analysis of the information provided by the log system is shown in Table 20.5. Specifically, it summarizes, chronologically, the sequence of events logged in the experiment. Note that, since the information needed to assess the behaviour of the nodes is more complex that the one that could be shown with a regular log, we provide a log including additional information. First, the events related to a given frame are grouped into subtables. Second, the column dedicated to the hub additionally shows when an error frame has been broadcast. Third, the column of a given node shows can contain events related to the error counters modifications. Moreover, we also determine when a given node reaches the error-active and the error-passive states due to the value of its error counters. Finally, note that, although the cause of a TEC/REC change is not logged by the node, it can be easily inferred and it is indicated as a comment, for instance, `tx/rx error`, `PE` or `tx/rx ok`.

Rows 2, 3 and 4 report what happens when the errors are injected in the first frame carrying an odd natural value, that is, '01'. They confirm that the hub detects an error in the 0th bit of the End Of Frame and that the TEC/REC are increased as expected. Similarly, rows 5 and 6 show that the nodes correctly decrease their TEC/REC by 1 when successfully transmitting/receiving a frame containing an even natural value. Note that each node firstly notifies about a change on the TEC/REC and, then, about the incorrect/correct transmission/reception of the corresponding frame. Rows 8, 9 and 10 confirm that Node 2 reaches the error-passive state when its REC exceeds the threshold established by CAN for this purpose see Sec. 2.2.6, that is, 127. Finally, rows 11, 12 and 13 show how this node returns to the error-active state when it successfully receives a frame.

	Hub	**Node0**	**Node1**	**Node2**
1	Ok 030#00	Tx Suc 030#00	Rx Suc 030#00	Rx Suc 030#00
2	Er 030#01 (eof(0))	—	—	—
3	error frame	TEC:008 ; tx error → +8	REC:001 ; rx error → +1	REC:009 ; rx error + PE → +1 +8
4	—	Tx Uns 030#01	—	—
5	—	TEC:007 ; tx ok → -1	REC:000 ; rx ok → -1	REC:008 ; rx ok → -1
6	Ok 030#02	Tx Suc 030#02	Rx Suc 030#02	Rx Suc 030#02
7	...	...	...	...
8	Er 030#05 (eof(0))	—	—	—
9	error frame	TEC:127 ; tx error → +8	REC:001 ; rx error → +1	REC:128 ; rx error + PE → +1 +8
10	—	Tx Uns 030#05	—	ERROR PASSIVE
11	—	TEC:126 ; tx ok → -1	REC:000 ; rx ok → -1	REC:127 ; rx ok → -1
12	—	—	—	ERROR ACTIVE
13	Ok 030#06	Tx Suc 030#06	Rx Suc030#04	Rx Suc 030#04

Table 20.5.: Log of the UPE test

Part VI.

Conclusions

21. Summary

This document presents the design and implementation of sfiCAN, the first fault-injection infrastructure for CAN that takes full advantage of a star topology. Specifically, sfiCAN relies on the CANcentrate's architecture [Barranco, 2010], whose central element is a CAN-compliant hub to which several nodes are connected. The most relevant characteristic of this architecture is that a given node communicates with the hub using two dedicated links, called uplink and downlink, which carry the signals transmitted and received separately. In addition, we adapted a specific port of the hub in order to connect a PC that uses a standard CAN device. Note that this connection is not divided into and uplink and a downlink. As concerns the hub's internals, we modified the coupling and synchronization mechanisms that were already implemented in the hub of CANcentrate; but we discarded all the fault-treatment mechanisms of that hub, since they are not needed in sfiCAN. Note that, it was also necessary to adapt the original bit-rate of the hub and the nodes to the one of the PC's CAN interface.

The functionality of sfiCAN was implemented in a set of various distributed logical units, called Network Configurable Components (NCCs), which are placed inside the hub and the nodes. A given NCC extends the functionalities of the physical element it is placed into, by performing a specific task. We distinguish the following NCCs: the Centralized Fault Injector (CFI), the Hub Logger (HL) and a set of Node Loggers (NLs). On the one hand, the CFI and the HL are synthesized together within the hub. In this sense, note that, thanks to the CANcentrate's synchronization mechanisms kept in the hub, both NCCs have access to every bit each node transmits and receives through its corresponding uplink and downlink. This allows them to inject erroneous bits and observe the subsequent reaction with high time and spatial resolution. On the other hand, each node's application has an embedded NL implemented in software, which retrieves information about the application's actions and the status of node's CAN controller.

Additionally, we provided the user with a software utility called Fault-Injection Management Station (FIMS), which allows to manage the operation of the NCCs, as well as exchange information with them, through the CAN network. For this, the FIMS executes on a PC that is connected to the hub port we adapted to support a standard CAN interface. In order to communicate the FIMS with the NCCs, we developed a protocol on top of CAN called NCC protocol. This protocol is based on the exchange of CAN frames, each of which encapsulates a so-called NCC message.

The set of faults that must be injected in a given experiment is defined by means of a fault-injection specification language we designed for this purpose. This language allows the user to specify each one of the faults as a specific value injected in a specific link when a set of conditions are met. This makes the specification of fault scenarios highly flexible and potent. In fact, it allows sfiCAN to test the behavior of software under faults without putting any restrictions on the tested software, such as requiring it to be modified or to generate deterministic traffic on the channel. All these capabilities are beyond those of any other injector previously proposed for CAN.

Various tests have been carried out in sfiCAN to both, ensure the correctness of its operation, and demonstrate its main characteristics. In this sense, we have demonstrated that sfiCAN is capable of forcing specific bit stream patterns permanently and temporarily (transiently or intermittently). Moreover, these erroneous bit streams could be injected independently in the signals received and transmitted from and to the nodes, that is, in the uplinks and downlinks. These features allowed to recreate simple fault scenarios, such as a single erroneous bit, and complex scenarios, such as an Inconsistent Message Omission (IMO) and an integrity error at the level of the application.

To recapitulate, first, the use of the CANcentrate's star topology and coupling schema allows sfiCAN to inject faults in the signals received and transmitted by the nodes separately. Moreover, the synchronization mechanisms inherit from CANcentrate allow to target individual bits, so that their value can be modified and monitored unequivocally. This two capabilities allow sfiCAN to overcome the limited space and time resolution of other fault injectors that can be used for CAN. In this sense, sfiCAN can provoke complex fault scenarios beyond the capabilities of any of these fault injectors, as just explained. Second, the use of the NCC architecture makes sfiCAN scalable and modular, which makes easier its construction and maintenance. Additionally, since they can be managed from a PC connected to the CAN network, it is possible to manage the experiments remotely. Finally, the fault-injection specification language allows the user to define each fault that must be injected as an error pattern and a set of conditions. This language features make the specification of fault scenarios highly flexible and potent. Moreover, the conditions make it unnecessary to know the traffic a priori and, thus, sfiCAN can test the behaviour of arbitrary production software without putting any restriction on the network load they generate.

22. Personal opinion

I really enjoyed carrying out this project for several reasons. First, the development of sfiCAN implied a multidisciplinary work. Specifically, its construction was divided in a set of heterogeneous tasks involving the study of specific theory and previous work, the handling of wiring and circuits, the description of hardware using VHDL and the construction of software using C++. To carry out all these tasks I had to exploit a wide range of the knowledge I received during the course of the degree.

Second, I had the chance to participate in the design of sfiCAN jointly with a team experienced in the development of dependable systems for CAN. This allowed me, not only to learn lots new concepts related to dependable systems, but to get involved in a real research project for the first time.

Third, as explained in Sec. 24, there are two publications based on the work carried out in this project. My participation in these publications and, particularly in the current document, helped me to develop my writing skills in technical English documents. On the one hand, I have learnt how to organize the ideas. On the other hand, I have enlarged my knowledge in English, specifically, the vocabulary and the grammar.

Finally, the most noteworthy aspect of participating in this project, is that I could get involved in the field of applied research. In this sense, note that, the development of a research project implies using the knowledge to solve problems that have never been faced. In my opinion, this fact presents an intellectual challenge, which, at the end, gives a higher personal satisfaction, when compared with a regular engineering project.

23. Proposed extensions

Although the development of sfiCAN fulfils all the requirements we think its design and implementation can be enhanced in order to extend its features. This improvements can make sfiCAN more suitable in areas out of its scope.

This section describes all those features that have not been implemented, either because they go beyond our initial design, or because their implementation implies a great effort, compared with the benefits they provide. These features show that the operation of sfiCAN can be extended easily to be able to assess CAN-based systems others than a basic CAN network.

On the one hand, we distinguish four different features not included in the design but, due to its benefits, they can be considered to be implemented in the future. First, note that the management of the different components conforming sfiCAN is carried out remotely except for the nodes, which must be programmed manually one by one. In this sense, the NCC infrastructure was initially sketched to hold an NCC called *Node Boot Loader* that would allow to install a specific node application throw the CAN network itself. For this, the FIMS would have to implement a deployment server, which could be used to specify the executable to be installed in each node. Second, the data gathered by the sfiCAN loggers allow to obtain specific information about the frames received and transmitted, from the point of view of the hub and the nodes. However, depending on the additional mechanisms implemented in sfiCAN, it could be necessary to extend their log capacity. For instance, the implementation of CANcentrate, explained in Sec. C, the Hub Logger was revised so it was capable of registering the final state of the ports. Third, as explained in [Barranco, 2010], the internal mechanisms of CANcentrate were designed to be capable of carrying out their job in the worst scenario, that is, when the time for another frame to be analyzed is the minimum. Specifically, this happens when frames are as short as they can be. Consequently, these mechanisms, which have been inherit in sfiCAN, are not prepared to deal with extended frames and, thus, they must be modified when working in environments using extended frames. Finally, as explained in Sec. 13.6, each NCC message is encapsulated within a given CAN frame. This approach allows an easy implementation of the protocol, however, it also introduces some limitations. For instance, it is not possible to configure the injector to check a bit pattern bigger than 56 bits. In this sense, when needed, the protocol could be improved to overcome this and other limitations.

On the other hand, we found three features that were not be implemented due to the amount of work that implies, when compared with its benefits. First, note that as explained in Sec. 17, the software executing in the nodes has not been implemented as described in the design section. Specifically, the application instructs the logger to store the relevant events occurring, instead of capturing them transparently. Consequently, additional code has to be added to the application, which make it dependent from the logger. We have not implemented such a logger because, at this point, it would not added any interesting characteristic to sfiCAN. Second, as explained in Sec. 11.4, the solution adapted to connect a standard CAN device to the CANcentrate hub has an important shortcoming. Specifically, only one standard connection can be used in the whole network domain. Moreover, an additional delay, due to the inherent feedback of the connection, is introduced. This solution must be revised in those areas where these limitations could present a problem. For instance, as explained in Sec. C to overcome the additional delay issue the bit-rate can be decreased. Finally, the initial sketch of the project proposal includes the multicast addressing, however, due to limited number of types of NCCs its implementation has not be considered. In case of increasing the number of types of NCCs this modification could be performed easily by adding specific filters in the NCCs and by adapting FIMS.

Part VII.

Results

24. Publications

Due to the relevance of this project, it has motivated various publications that I co-authored together with David Gessner, Manuel Barranco and Julián Proenza. These publications are the following:

- David Gessner, Manuel Barranco, Alberto Ballesteros, and Julián Proenza, Designing sfiCAN: a star-based physical fault injector for CAN. In 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2011.
- Julián Proenza, Manuel Barranco, David Gessner, and Alberto Ballesteros. sfiCAN: a Star-based Physical Fault Injector for CAN networks, 2011.

The first is a work-in-progress paper that sums up the design and part of the implementations done in sfiCAN. Specifically, the design sketches its general architecture and operation, whereas the implementation introduces some of its main aspects and describes a representative experiment carried out with the prototype. This paper has been presented at the 16th ETFA conference and published in enginy@eps, an academic journal edited in the University of the Balearic Islands itself. The second is a full research paper that extends the description done in the work-in-progress paper. In this sense, note that we discuss more in-depth the design of sfiCAN. Moreover, we present three different experiments that show the most important testability characteristics developed between the writing of these two papers.

25. Practical impact of the results

As discussed in the introduction the usage of CAN has steadily been increasing to the extent that it currently coexist with newer protocols. Moreover, this trend is expected to continue because CAN is still penetrating old and new markets and because of potentially upcoming new high-volume applications. However, due to the reliability limitations of CAN in dependable and real-time applications, it has suffered various revisions last years. Moreover, there is also a significant amount of research being done to address some of these shortcomings.

All this research claims an adequate fault-injection system to test the response of CAN-based applications and protocols when faults and errors do occur. In this sense, note that, an adequate fault-injection system capable of testing CAN production software is important to both the industry and academia. Just as explained in Sec. 4.3 the existing fault injectors for CAN do not cover the fault model needed to assess these high-dependable systems. Thus, the development of sfiCAN provides the community with a useful tool to evaluate the dependability of such systems.

On the one hand, as concerns the industry, sfiCAN has already generated interest in a particular company involved in the evaluation of critical systems. Specifically, it develops software tools for the verification, optimization and code coverage of critical real-time embedded systems.

On the other hand, regarding the academia, sfiCAN has been developed in the context of CANbids (CAN-Based Infrastructure for Dependable Systems) [University of the Balearic Islands, 2011]. CANbids is a CAN-based infrastructure for supporting the execution of highly-dependable distributed control applications being developed by the Systems, Robotics and Vision group of the University of the Balearic Islands. In this sense, sfiCAN is a fundamental piece of this system that will help to assess the fault-tolerant mechanisms implemented in this infrastructure. In fact, it has already been used to evaluate qualitatively the operation of two parts of the CANbids project.

On the one hand, as explained in App. C we have already tested the CANcentrate fault-treatment mechanisms under sfiCAN. In this sense, although we did not carried out a fault-injection campaign to evaluate quantitatively the CANcentrate fault-treatment mechanisms, we perform some tests and proved that it is feasible. Moreover, we could observe that the sfiCAN's design made easy, not only the installation of the fault-treatment mechanisms, but the improvement of the log system to collect and report the new log information. On the other hand, sfiCAN was also used to evaluate the operation of a new error-signaling mechanisms called Aggregated Error Flag, which has been designed and implemented in [Winter, 2012]. Finally, note that, although sfiCAN was designed to assess CANbids, it can also be used in any CAN-compliant application.

Part VIII.

Appendix

A. Quick start guide

Once the PC has boot the CAN interfaces should be activated. For this purpose, the user must run the `restartcan` script located in `sfiCAN/PC/scripts/`. List. A.1 shows how to run it, in order to restart both interfaces of the PCI CAN controller. Note that, this script can be used at any time to reinitialize the interfaces when they stand in an error state, for instance, bus-off.

```
$ ./restartcan
```

Listing A.1: Example of the `restartcan` script call.

The state of the CAN interfaces, as well as other network interfaces, can be consulted by executing the `ip link` command. List. A.2 shows the response of this command after enabling the CAN interfaces. Note that `can0` and `can1` are enabled since both are ”UP”. Now there is at least one CAN interface enabled and, thus, the Fault-Injection Management Station (FIMS) can communicate with the NCCs.

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    qlen 1000
    link/ether bc:ae:c5:4b:14:6c brd ff:ff:ff:ff:ff:ff
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 A.2: Example of `ip link` command dump.

As depicted previously, the configuration mode is the NCCs default mode after starting sfi-CAN, thus, it is possible to configure the NCCs without having to send any mode change message. In order to set up the operation of the Centralized Fault Injector (CFI) the user must specify the fault-injection scenario and then transmit it to the CFI. On the one hand, the construction of the fault-injection specification must be done in a text file, accordingly with the fault-injection language defined in Sec. 10. Moreover, the fault-injection specification template, which can be found in `sfiCAN/PC/FIMS/fic/config0` and at the end of this appendix (app. A.1), can be used to make easier the fault-injection specification construction by summarizing all its possible values. Specifically, this template contains all the fault-injection specification parameters accompanied by the set of possible values or its type. On the other hand, the transmission of the fault-injection specification is performed thanks to the Fault-Injector Configurator. This part of the FIMS is implemented by the `fic`, placed in `sfiCAN/PC/FIMS/fic/`. Next, a brief explanation of this tool is performed.

This program accepts a set of different shell arguments, in order to set up specific parameters or modify its behaviour. Specifically, it allows to specify the source file containing the fault-injection specification, `-f=<file>`, as well as the CAN interface used to transmit it, `i=<iface>`. Additionally it is possible to just perform a fault-injection specification file checking by setting the `-c` flag. List. A.3 shows how to process the `IMO.cfg` file to check and transmit the fault-injection specification through the `can0` interface.

```
$ ./fic -f=IMO.cfg -i=can0
processing file ...
   3 group(s) found!

initializing 'can0' socket ...

checking file ...
   0 error(s) found!

processing file ...
Ending...
Done!
```

Listing A.3: Example of Fault-Injection Coordinator's execution.

Note that, when no fault-injection specification file and/or CAN interface is explicitly defined, the program searches for file named `config` and/or transmits through the `can0` respectively.

Once the CFI is configured, the user must change the operating mode of the NCCs, in order to enable or disable their function during the experiment. Specifically, to disable a given NCC it is enough to send a `enter-idle-mode` message. Note that, it is also possible to unplug a given node to disable all the NCCs contained in it. Contrary, to enable a given NCC a `enter-wfw-mode` message must be sent. Tables A.1 and A.2 show the set of NCC IDs, as well as the mode change message codes. Finally, List. A.4 shows how to force node0 and node1 loggers to enter in idle and wait-for-whistle mode respectively. Note that, an specific SocketCAN command, called `cansend`, is used.

NCC	NCC ID
Broadcast	000
CFI	001
HL	002
NL0	003
NL1	004
NL2	005
FIMS	010

Table A.1.: List of NCC IDs. NCC IDs are represented as 11-bit hexadecimal values.

Mode change message	Code
enter-config-mode	20
enter-idle-mode	21
enter-wfw-mode	22
starting-whistle	23

Table A.2.: List of mode change message codes

```
$ cansend can0 003#21
$ cansend can0 004#22
```

Listing A.4: Example of mode change messages transmission.

At this point, to start the experiment it is necessary to force all NCCs in wait-for-whistle mode to progress to execution mode. To do so it is enough to send a broadcast `starting-whistle` message. List. A.5 shows how to perform this action.

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

Listing A.5: Example of starting-whistle message transmission.

Some useful information can be provided by the CAN interface connected to the hub, during the experiment. On the one hand, SocketCAN provides an specific command, called `candump` to capture and show the set of received CAN frames. List. A.6 shows how to execute it and the generated data. On the other hand, driver information can be gathered executing `cat /proc/pcan`, a reduced version can be seen in List. A.7. The most important information on this dump, during an experiment, can be found in column `status`, which contains the current status of an interface. Note that, when an interface is in an error state its value is different from `0x0000`, and, thus, it must be restarted using the `restart` script.

```
$ 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 A.6: Example of `candump` execution.

```
$ 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 00001b5c 000028f9 0000009e 0x000c
 1    pci can1 0x0014 00000000 00000000 00000000 00000000 0x0000
```

Listing A.7: Example of `/proc/pcan/` dump.

Once the experiment is over, NCCs must be instructed to enter in configuration mode, so they stop their operation. To do so it is enough to send a broadcast `enter-config-mode` message, as shown in List. A.8. Note that, not only NCCs in execution mode are forced to enter in configuration mode but also NCCs in idle mode.

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

Listing A.8: Example of enter-config-mode message transmission.

The FIMS' Fault-Injection Log retriever, implemented in `fil` and stored in `sfiCAN/PC/FIMS/fil/`, can be used at this point to retrieve the log information collected during the experiment. List. A.9 shows a reduced example of the execution of this tool after running an IMO scenario.

```
$ ./fil
initializing 'can0' socket ...
retrieving log data...
---------
-- HUB --
---------
Ok 030 [1] 00 port0
Er 030 [1] 01 port0 (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 A.9: Example of Fault-Injection Log retriever's execution.

The information provided depends on the type of NCC logger. On the one hand, the hub logger collects the set of frames received. For each frame it stores:

- Whether it is a valid frame, `Ok` when so and `Er` when not.
- Three hex characters containing the value of identifier field.
- Between brackets, the value of the DLC field, that is, the number of data bytes carried into the frame.
- Two hex characters containing the value of the first byte of the data field.
- The source port of the frame.
- If it is not a valid frame, the field and the bit where the hub has detected the error.

On the other hand, node loggers collect the set of frames received and transmit, as well as the value of the error counter when their value changes. Specifically, for each frame it stores:

- Two decimal values containing the number of the frame.
- Whether the frames has been transmitted, `tx`, or received, `rx`.
- Three hex characters containing the value of identifier field.
- Between brackets, the value of the DLC field, that is, the number of data bytes carried into the frame.
- Two hex characters containing the value of the first byte of the data field.

A.1. Fault-injection specification template

```
#comment

[fault injection 1]

 ###################
 # value to inject #
 ###################

value_type =
  dominant  |
  recessive |
  pattern   |
  inverse

value_pattern =
  1100 #6 byte bitstring 0-48 bits


 #####################
 # link where inject #
 #####################

target_link =
  port0up | port0dw |
  port1up | port1dw |
  port2up | port2dw |
  port3up | port3dw


 ##################
 # injection mode #
 ##################

mode =
  single-shot |
  continuous  |
  iterative


 #########
 #  aim  #
 #########

aim_filter =
  xxxxxxxxx01 #7 byte 56 bitstring
```

```
aim_field  =
  idle |
  id | rtr | res | dlc | data | crc | crcDelim | ack | ackDelim | eof |
  interfield | errFlag | errDelim

aim_link =
  port0up | port0dw |
  port1up | port1dw |
  port2up | port2dw |
  port3up | port3dw |
  coupled

aim_role =
  dont_care | tx | rx

aim_count =
  10 #2 byte unsigned integer 0-65535


 ############
 # withdraw #
 ############

withdraw_filter =
  xxxxxxxxx01 #7 byte 56 bitstring

withdraw_field  =
  idle |
  id | rtr | res | dlc | data | crc | crcDelim | ack | ackDelim | eof |
  interfield | errFlag | errDelim

withdraw_link =
  port0up | port0dw |
  port1up | port1dw |
  port2up | port2dw |
  port3up | port3dw |
  coupled

withdraw_role =
  dont_care | tx | rx

withdraw_count =
  10 #2 byte unsigned integer 0-65535


 ################
 # target_frame #
 ################

target_frame_filter =
  xxxxxxxxx01 #7 byte 56 bitstring

target_frame_field  =
  idle |
  id | rtr | res | dlc | data | crc | crcDelim | ack | ackDelim | eof |
  interfield | errFlag | errDelim

target_frame_link =
  port0up | port0dw |
  port1up | port1dw |
  port2up | port2dw |
  port3up | port3dw |
  coupled

target_frame_role =
  dont_care | tx | rx


 ########
 # fire #
 ########

fire_field  =
  idle |
  id | rtr | res | dlc | data | crc | crcDelim | ack | ackDelim | eof |
  interfield | errFlag | errDelim

fire_bit =
  4  #1 byte unsigned integer 0-63

fire_offset =
  10 #2 byte unsigned integer 0-65535


 #########
 # cease #
 #########

cease_field =
  idle |
  id | rtr | res | dlc | data | crc | crcDelim | ack | ackDelim | eof |
  interfield | errFlag | errDelim

cease_bit =
  4 #1 byte unsigned integer 0-63

cease_bc =
  50 #2 byte unsigned integer 0-65535
```

B. Additional comments

B.1. PCAN driver (re)installation

Since the PCAN driver depends on the kernel, it must be configured and installed every time a kernel upgrade is performed. To do so, download the Linux version of the driver from this link, and execute the commands shown in List. B.1. Note that the Makefile located in `peak-linux-driver-<version>/driver/` allows to configure some parameters of the compilation. In this sense, is important to know that, in order to support socketCAN, the driver must be compiled enabling the `netdev` support, which is the default option in kernels newer than 2.6.25.

Peak system usually performs fixes and upgrades of its Linux driver, for instance, version 7.3 adds support for kernel version 3.0. Thus, it is useful to check often its website. Additional driver information can be found in the PCAN driver for Linux PDF user manual, jointly distributed with the driver source code.

```
$ cd peak-linux-driver-<version>/
$ make clean
$ make
$ sudo make install
```

Listing B.1: PCAN setup commands.

B.2. PCAN driver initializations

In Ubuntu, file `/etc/modprobe.d/pcan.conf` contains specific instructions to both load the PCAN module and set up the default speed of the CAN interfaces, at the startup. List. B.2 shows the complete content of this file. Note that the first part is automatically inserted when installing the PCAN driver, while the line `options pcan bitrate=0x0014` has been appended later to achieve 1 Mbps. This option, as well as other, can be modified at any time by editing this file. The codes' list of the possible default bitrates, as well as further information, can be found in Sec. 7, ”Customization of the modprobe Configuration File”, of the PCAN driver for Linux PDF user manual.

```
# pcan - automatic made entry, begin --------
# if required add options and remove comment
# options pcan type=isa,sp
install pcan /sbin/modprobe --ignore-install pcan
# pcan - automatic made entry, end ----------

# 1Mbps
options pcan bitrate=0x0014
```

Listing B.2: pcan.conf content.

B.3. Gathering driver information

There are some ways to obtain useful data of the controller operation. First, to check whether the driver is loaded, as well as its parameters, commands shown in List. B.3 can be used. Second, command `ip link`, whose execution can be seen in List. B.4, can be called when there is a need of enabling/disabling network interfaces, as well as showing it. Additionally, this command can interact with SocketCAN to configure low-level parameters, for instance, activate listen-only mode or change the bitrate. However, an error message is returned when executing these configurations in these CAN interfaces. Thus, the only way to do so is to interact with the driver. Note that, since the controller is installed as regular network device, command `ifconfig` can also be used to enable/disable and show these network interfaces. Finally, to gather closer information about the CAN interfaces, the user must execute the `cat /proc/pcan/` command. List. B.5 shows a simplified version of the dump generated by this command. In this sense, two specific columns, called `btr` and `status`, highlight over the other. On the one hand, column `btr` contains the current bitrate. On the other hand, column `status` contains the last occurred error status of the channel. Specifically, each bit contained in its value has an specific meaning described in Sec. 18, ”FAQ”, of the PCAN driver for Linux PDF user manual.

```
$ lsmod | grep pcan
$ modinfo pcan
```

Listing B.3: pcan.conf content.

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 16436 qdisc noqueue state UNKNOWN
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    qlen 1000
    link/ether bc:ae:c5:4b:14:6c brd ff:ff:ff:ff:ff:ff
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 B.4: Example of `ip link` dump.

```
$ 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 00001b5c 000028f9 0000009e 0x000c
 1    pci can1 0x0014 00000000 00000000 00000000 00000000 0x0000
```

Listing B.5: Example of `/proc/pcan/` dump.

B.4. Unblock a given CAN interface

A given CAN interface from the PC CAN controller blocks in front on some situations, for instance, it enters in bus off or the transmit buffer is full. The current status of a CAN interface can be checked anytime by executing `cat /proc/pcan` as explained. In case of error an interface restart can be performed to return its operation to the normal behaviour. To do so execute the `restartcan` script.

B.5. Scripts

Although the most useful scripts are placed inside `sfiCAN/PC/scripts/`, some of them are also placed in `/usr/local/bin/` so they can be executed from any directory. Specifically, `restartcan` aims to restart the can interfaces, while `speedcan` helps to change the CAN interfaces' bitrate.

B.6. Listen-only mode

The PC CAN controller acts like a regular CAN controller during an experiment, for instance, it sends ACKs when receiving a frame, and error frames when detecting an error. In order to avoid this behaviour we have enabled the possibility to activate the *listen-only mode*. When the PC CAN controller works in listen-only mode it does not interact with the other nodes, that is, it can still receive frames but no data is transmitted.

To activate the listen-only mode in the PC CAN controller we decide to use the high-level C library provided by the manufacturer, that is, the pcan library. As explained in Sec. 18.2, it allows to initialize the device, as well as transmit and receive CAN frames. In this sense, note that the device initialization provides a low-level access to some specific operation parameters, such as the bit-rate, the use of extended frames or the so-called listen-only mode. Additionally, we created two C programs that use this library and enable/disable the listen-only mode, respectively. New we discuss this two tasks separately.

On the one hand, we created a new version of the libpcan library adding a new initialization function called `CAN_Init_lo`. This new function is based on the `CAN_Init` function, with the only difference that it enables a new parameter allowing to specify the value of the state of the listen-only mode. This new library is placed inside the `sfiCAN/PC/listen-only/lib/` directory, jointly with a `Makefile`, which allows to compile and install it. The compilation can be done by simple executing the `make` command, whereas the installation needs extra parameters, that is, `sudo make install`. Note that sudo privileges are also needed. Finally, note that the installation of a new version of the PCAN drivers overwrites the current libpcan library installed in the system with a default version. Thus, it is necessary to install the modified libpcan library version each time the CAN device drivers are updated.

On the other hand,we created two C programs called `enable_listen-only` and `disable_listen-only` located at `sfiCAN/PC/listen-only/`, that use the modified libpcan library in order to enable and disable the listen-only mode, respectively. Additionally, a `Makefile` is provided so they can be compiled. Finally, note that, since a device initialization is needed each time the listen-only mode is enabled/disabled, other device parameters are also reset. Specifically, each time any of these two programs is executed the bit-rate and the type of frames used by the `can0` channel is set to 1 Mbps and standard frames, respectively. These parameters can be customized by editing the source code of the programs.

Finally, note that, to apply this concept to the experiment phases previously defined, the listen-only mode must be enabled just after starting the experiment, that is, after sending the `enter-execution-mode` message. In this case, the listen-only mode must be disabled just before ending the experiment, that is, before sending the `enter-config-mode` message.

B.7. Single-shot in transmission node

In some tests could be mandatory to use the single-shot transmission, that is, a transmission mode in which no retransmission is carried out even if an error is detected. We have enabled this possibility in the nodes. For this, we had to perform some modifications in both the modified ReCANcentrate driver and the CAN lib, see Sec. 17.2.

On the one hand, we edited the `can_controller.c` file from the modified ReCANcentrate driver, that is, the body of the library that implements the low-level initialization, transmission and reception functions. Specifically, we have added a piece of code in the `request_tx` function, that is, the function allowing to transmit CAN frames. This piece of code waits for the current frame to start to be transmitted, in order to unenqueue the frame from the transmit buffer. Since the transmission has started it cannot be cancelled, however, if any error is detected, the frame it is not accessible to be retransmitted. Note that, it is enough to activate this piece of code to enable the single-shot mode.

On the other hand, we edited the `can.c` file from the CAN lib, that is, the library responsible for abstracting the initialization, the transmission and the reception tasks. Specifically, we add specific code in the `tx_frame`, that is, the function used to transmit frames. This code is devoted to replace the busy wait that blocks the function until the transmission is carried out, for a timed wait that blocks the function for a specific amount of time. Note that, when enabling the single-shot mode the user must use the timed wait; otherwise, the busy wait must be used.

C. (Re)CANcentrate assessment by means of sfiCAN

In the scope of the research in CAN done at the Universitat Illes Balears two projects have special interest, CANcentrate and ReCANcentrate. As explained in Sec. 5.1, CANcentrate increases the dependability level of CAN-based systems by means of specific fault-treatment mechanisms, whereas ReCANcentrate provides fault tolerance, in addition, by means of replication.

It is possible to assess the operation of these two developments by means of sfiCAN. In this sense, note that CANcentrate set the bases for the development of the sfiCAN's infrastructure and, thus, its special hardware needs are already met in sfiCAN. Similarly, the ReCANcentrate mechanisms can be accommodated easily in sfiCAN, after adapting it to the replicated infrastructure.

Note that, depending on the CAN-based development used together with sfiCAN it can be necessary to take into account specific details. On the one hand, the operation of the given development could affect the operation of sfiCAN. For instance, note that CANcentrate can avoid the reception of a node's contribution by disabling its port. This can cause communication problems in the report phase, specifically, when nodes transmit their report. In this case, the problem can be solved by forcing all ports to be enabled at the end after an experiment is over. On the other hand, it can be necessary to revise the loggers in order to make them capable of gathering specific information about the development's operation. For instance, as explained below, when combining sfiCAN with CANcentrate the hub logger was extended to store the final state of the ports, so the user can determine the behaviour of the Fault Treatment Module.

Next, we discuss how to combine the (Re)CANcentate fault-treatment and fault-tolerance mechanisms with sfiCAN.

C.1. CANcentrate

As explained previously, the infrastructure of sfiCAN is based in the one of CANcentrate. In fact, as can be extracted from the whole document, only minor fixes should be done to accommodate the CANcentrate fault-treatment mechanisms to the sfiCAN infrastructure.

First, it is necessary to install the CANcentrate's `faultTreatmentModule`, that is, the module responsible for implementing the fault-treatment mechanisms. Specifically, this module outputs a set of signals each of which instructs the top module to enable or disable a specific node contribution. For this, the `faultTreatmentModule` needs the signals carrying the contribution of the nodes, as well as the coupled signal. How these signals are acquired and driven is described later, after discussing the synchronization issues appearing when combining the sfiCAN and CANcentrate internals.

Second, note that the `faultTreatmentModule` already implements all the CAN synchronization logic. However, as explained in Sec. 16.4, we extracted and encapsulated all the modules implementing the CAN synchronization in just one module called `canModule`, which is placed at the top level. Moreover, we added a specific module called `nodeRoleModule` that determines the current role being played by the nodes. All these modules implemented in the `canModule` are organized in such a way that they provide a higher level of abstraction and, thus, their outputs can be accessed easily. Consequently, although it is possible to use the native modules already implemented in CANcentrate, here we take the sfiCAN's approach.

Two modifications must be performed in order to use the `canModule` as the central synchronization module, together with the `faultTreatmentModule`. On the one hand, the existing CANcentrate synchronization logic must be disabled. This can be done easily by deleting all involved modules from the `faultTreatmentModule`. Then, the unconnected signals must be re-driven to the corresponding outputs of the `canModule`. On the other hand, the `canModule` must be modified to take into account the enabling/disabling signals, that is, the `thmEnaDis` signals, generated by the `faultTreatmentModule`. The reason for this is because a node suffering from a stack-at dominant could be considered as a transmitter, even after being disabled by the `faultTreatmentModule`. To avoid this behaviour the `nodeRoleModule` has to be adapted to discard a given contribution when `faultTreatmentModule` instructs to disable it.

Third, once the `canModule` and the `faultTreatmentModule` have been adapted to be able to work together, the contributions of the nodes, as well as the coupled signal, can be driven to the this last module. The signals carrying the contribution of the nodes can be acquired from the `canModule` itself. These signals, called `auxSynIomPortContri`, are a synchronized version of the injected contribution of the nodes. Similarly, the `auxSynRout` signal, also provided by `canModule`, carries a synchronized version of the coupled signal.

Fourth, it is necessary to implement the specific logic that allows the `faultTreatmentModule` to enable and disable the contribution of the nodes. To do so it is enough to mask each of them by means of an OR gate placed into the `couplerModule`. Specifically, the masked contribution of a node is obtained from the output of an OR gate fed by the regular contribution of the node and its corresponding enabling/disabling signal.

Finally, the Node Loggers must be enabled in the nodes. To do so, it is enough to install the specific code developed for this purpose. In this sense, note that, as explained in Sec. 17.2, the software running in the node must include specific sentences to update the log data.

Note that this construction has already been tested, with the only difference that the `nodeRoleModule` was implemented separately, in order to ensure the independence of the `canModule`. Moreover, we extended the capabilities of the Hub Logger, so it was able to gather information about the final state of the ports. That is, whether the `faultInjectionModule` determined that the ports were idle, active, disabled or reintegrated, see [Barranco, 2010]. In this sense, we assessed that the current design of the hub logger and the FIMS makes easy the increasing of the log capabilities.

C.2. ReCANcentrate

As explained previously, ReCANcentrate is an extension of CANcentrate in which the hub is replicated, in order to increase its fault-tolerance capabilities. Consequently, to assess ReCANcentrate it is necessary to reproduce this infrastructure in sfiCAN. Moreover, similarly to the CANcentrate assessment, the logic of the hubs must be revised in order to provide the fault-treatment mechanisms. Next, we describe the specific tasks involved in these modifications.

In order to construct the ReCANcentrate's replicated start, it is first necessary to provide sfiCAN with an additional hub connected with the existing one. The ReCANcentrate's design specifies that this connection is constructed by means of two replicated links called *sublinks* identical to the ones used in the nodes. Thus, two additional ports must be enabled in each hub. Once both hubs are physically connected, the fault-treatment mechanisms can be enabled. This is done identically as described in the previous section, that is, by installing the `faultTreatmentModule` in such a way so it can work together with the existing `canModule`. Additionally, the `couplerModule` must be modified to add the set of OR gates that allows to enable/disable the contributions of the nodes and now, of the other hub. In this sense, note that the ReCANcentrate's `faultTreatmentModule` generates two additional enabling/disabling signals, one each sublink, so the replicated contribution of the other hub can be omitted in case of transmitting errors. Finally, note that, as described in [Barranco, 2010], the `couplerModule` must implemented and additional AND gate so the coupling is performed in two phases.

As concerns the nodes, apart from enabling the Node Loggers, it is necessary to enable the driver's media management for replicated starts, so the node can communicate with the second hub.

Finally, note that the there are some details related to the replicated infrastructure in sfiCAN that must be taken into account. On the one hand, as describe in Sec. 11.4, the main shortcoming of the solution adopted to adapt a standard CAN device into the CANcentrate coupling schema, is that only one standard CAN device can be used in the whole domain. Consequently, only one hub can hold a port for a PC. In case of enabling more than one port, dominant feedbacks appear, preventing the correct channel operation. On the other hand, we have detected important bit errors when both hubs are coupled and a port for a standard CAN device is enabled in one of them. This is caused by the additional delay necessary to stabilize the signal in the PC port. When this delay is added to the propagation time in the ReCANcentrate infrastructure, which is higher than in CANcentrate, some dominant to recessive bit sequences are seen as two dominant values. This happens since the value of the first dominant remains even when the value of the recessive must be present. One solution to overcome this limitation is to decrease the bitrate used in sfiCAN and, thus, to increase the bit time.

D. Hub Core source code

D.1. Packages

D.1.1. defGeneral.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package defGeneral is

  --
  -- Hardware definitions
  --

  type t_port is (port0, port1, port2);

  type t_link is (
    port0up, port0dw,
    port1up, port1dw,
    port2up, port2dw,

    none
  );

  type t_portSignalArray is array (t_port) of std_logic;
  type t_linkSignalArray is array (t_link) of std_logic;


  --
  -- Logic definitions
  --

  -- byte type
  subtype t_byte is std_logic_vector(1 to 8);


  -- frame identifier type
  subtype t_id is std_logic_vector(1 to 11);

  -- frame data type
  subtype t_data is std_logic_vector(1 to 64);

  constant id0   : t_id   := "00000000000";
  constant data0 : t_data := x"0000000000000000";

  -- frame crc type
  subtype t_crc is std_logic_vector(1 to 15);


  --
  -- General type parsing
  --
  type t_portSLV is array (t_port) of std_logic_vector(1 to 3);
  constant portSLV : t_portSLV := (
    port0 => "000",
    port1 => "001",
    port2 => "010"
  );

  type t_to_uplink is array (t_port) of t_link;
  constant to_uplink : t_to_uplink := (
    port0 => port0up,
    port1 => port1up,
    port2 => port2up
  );

  type t_to_port is array (t_link) of t_port;
  constant to_port : t_to_port := (
    port0up => port0,
    port0dw => port0,
    port1up => port1,
    port1dw => port1,
    port2up => port2,
    port2dw => port2,

    none    => port0
  );

end defGeneral;

package body defGeneral is
end defGeneral;
```

D.1.2. defStates.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.All;

use work.defGeneral.all;
use work.defInteger.all;

package defStates is

  type estadoGlobalFrame is (idle, idField, rtrField, resField, dlcField,
        dataField, crcField, crcDelimField,
        ackSlotField, ackDelimField, eofField,
        interField, errorFlag, errorDelimiter,
        freeState1, freeState2);

  type t_fieldLong is array (estadoGlobalFrame) of ENTER65;
  constant fieldLong : t_fieldLong := (
    -- idle
    idle           => 0, -- variable

    -- Arbitration
    idField        => 11,
    rtrField       => 1,

    -- Control
    resField       => 2,
    dlcField       => 4,

    -- Data
    dataField      => 0, -- variable

    -- CRC
    crcField       => 15,
    crcDelimField  => 1,

    --ACK
    ackSlotField   => 1,
    ackDelimField  => 1,

    eofField       => 7,

    -- Other
    interField     => 3,

    -- Error frame
    errorFlag      => 6,
    errorDelimiter => 8,

    -- Free states, non existing fields
    freeState1     => 0,
    freeState2     => 0
  );


  --
  -- Bit parsing
  --

  type t_frameStateSLV is array (estadoGlobalFrame) of std_logic_vector(1 to 4);
  constant frameStateSLV : t_frameStateSLV := (
    idle           => "0000",
    idField        => "0001",
    rtrField       => "0010",
    resField       => "0011",

    dlcField       => "0100",
    dataField      => "0101",
    crcField       => "0110",
    crcDelimField  => "0111",

    ackSlotField   => "1000",
    ackDelimField  => "1001",
    eofField       => "1010",
    interField     => "1011",

    errorFlag      => "1100",
    errorDelimiter => "1101",

    freeState1 => "1110",
    freeState2 => "1111"
  );

end defStates;


package body defStates is
end defStates;
```

D.1.3. defNCC.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

use work.defGeneral.all;

package defNCC is

  -- ================== --
  -- ==== Constants ==== --
  -- ================== --

  --
  -- General IDs
  --

  constant BROAD_ID   : t_id := "00000000000"; --000
  constant PC_ID      : t_id := "00000010000"; --010

  constant HUB_FIM_ID : t_id := "00000000001"; --001
  constant HUB_LOG_ID : t_id := "00000000010"; --002


  --
  -- Command codes
  --

  constant CMD_CFG : std_logic_vector(1 to 3) := "000"; -- configuration command
  constant CMD_MCH : std_logic_vector(1 to 3) := "001"; -- mode change command
  constant CMD_LOG : std_logic_vector(1 to 3) := "010"; -- report log comand


  --
  -- Mode change codes
  --

  constant MC_ECM : std_logic_vector(1 to 5) := "00000"; -- enter config mode
  constant MC_EIM : std_logic_vector(1 to 5) := "00001"; -- enter idle mode
  constant MC_EWM : std_logic_vector(1 to 5) := "00010"; -- enter wait-for-whistle mode
  constant MC_EEM : std_logic_vector(1 to 5) := "00011"; -- enter execution mode


  -- =============== --
  -- ==== Types ==== --
  -- =============== --

  -- ncc mode type
  type t_nccMode is (configMode, idleMode, wfwMode, execMode);


  --
  -- Debugging
  --

  type t_nccModeNAT is array (t_nccMode) of natural;
  constant nccModeNAT : t_nccModeNAT := (
    configMode => 0,
    idleMode   => 1,
    wfwMode    => 2,
    execMode   => 3
  );
```

```
end defNCC;

package body defNCC is
end defNCC;
```

D.1.4. defInjection.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;

package defInjection is

  -- ================== --
  -- ==== Constants ==== --
  -- ================== --

  --
  -- General constants
  --

  constant MAX_INJS : natural := 5;


  --
  -- Param codes
  --

  constant PARAM_VALUE_TYPE              : std_logic_vector (1 to 5) := "00000";
  constant PARAM_VALUE_BFVALUE           : std_logic_vector (1 to 5) := "00001";

  constant PARAM_LINK                    : std_logic_vector (1 to 5) := "00010";

  constant PARAM_MODE                    : std_logic_vector (1 to 5) := "00011";

  constant PARAM_START_TRIGGER_FILTER    : std_logic_vector (1 to 5) := "00100";
  constant PARAM_START_TRIGGER_MASK      : std_logic_vector (1 to 5) := "00101";
  constant PARAM_START_TRIGGER_MASK_LONG : std_logic_vector (1 to 5) := "00110";
  constant PARAM_START_TRIGGER_FIELD     : std_logic_vector (1 to 5) := "00111";
  constant PARAM_START_TRIGGER_LINK      : std_logic_vector (1 to 5) := "01000";
  constant PARAM_START_TRIGGER_ROLE      : std_logic_vector (1 to 5) := "01001";
  constant PARAM_START_TRIGGER_COUNT     : std_logic_vector (1 to 5) := "01010";

  constant PARAM_END_TRIGGER_FILTER      : std_logic_vector (1 to 5) := "01011";
  constant PARAM_END_TRIGGER_MASK        : std_logic_vector (1 to 5) := "01100";
  constant PARAM_END_TRIGGER_MASK_LONG   : std_logic_vector (1 to 5) := "01101";
  constant PARAM_END_TRIGGER_FIELD       : std_logic_vector (1 to 5) := "01110";
  constant PARAM_END_TRIGGER_LINK        : std_logic_vector (1 to 5) := "01111";
  constant PARAM_END_TRIGGER_ROLE        : std_logic_vector (1 to 5) := "10000";
  constant PARAM_END_TRIGGER_COUNT       : std_logic_vector (1 to 5) := "10001";

  constant PARAM_SEL_TRIGGER_FILTER      : std_logic_vector (1 to 5) := "10010";
  constant PARAM_SEL_TRIGGER_MASK        : std_logic_vector (1 to 5) := "10011";
  constant PARAM_SEL_TRIGGER_MASK_LONG   : std_logic_vector (1 to 5) := "10100";
  constant PARAM_SEL_TRIGGER_FIELD       : std_logic_vector (1 to 5) := "10101";
  constant PARAM_SEL_TRIGGER_LINK        : std_logic_vector (1 to 5) := "10110";
  constant PARAM_SEL_TRIGGER_ROLE        : std_logic_vector (1 to 5) := "10111";

  constant PARAM_START_FIELD             : std_logic_vector (1 to 5) := "11000";
  constant PARAM_START_BIT               : std_logic_vector (1 to 5) := "11001";
  constant PARAM_START_OFFSET            : std_logic_vector (1 to 5) := "11010";

  constant PARAM_END_FIELD               : std_logic_vector (1 to 5) := "11011";
  constant PARAM_END_BIT                 : std_logic_vector (1 to 5) := "11100";
  constant PARAM_END_BC                  : std_logic_vector (1 to 5) := "11101";

  constant PARAM_EOC                     : std_logic_vector (1 to 5) := "11111";


  -- =============== --
  -- ==== Types ==== --
  -- =============== --

  -- fault-injection type
  type t_fiType is (stuckAtDominant, stuckAtRecessive, bitFlipping, inverse, none);

  -- fault-injection link
  subtype t_fiLink is t_link range port0up to port2dw;

  -- fault-injection mode
  type t_fiMode is (singleShot, continuous, iterative);

  -- end condition
  type t_endCond is (FieldBit, bitCount);

  -- trigger role
  type t_triggerRole is (dontCare, tx, rx);

  -- fault-injection state
  type t_fiState is (waiting, enabled, disabled);


  --
  -- Fault-injection configuration type
  --

  -- Type fault-injection value
  type t_fiValue is record
    fiType  : t_fiType;
    bfValue : std_logic_vector(0 to 48-1);
    bfLong  : ENTER64; -- posible values 2 to 56
  end record;

  -- Type trigger
  constant FILTER_LONG : ENTER64 := 56;
  type t_trigger is record
    filter    : std_logic_vector(0 to FILTER_LONG-1);
    mask      : std_logic_vector(0 to FILTER_LONG-1);
    mask_long : ENTER64; -- posible values 1 to FILTER_LONG
    field     : estadoGlobalFrame;
    link      : t_link;
    role      : t_triggerRole;
    count     : ENTER65536; -- trigger count
  end record;

  -- Type fault-injection start
  type t_fiStart is record
    field  : estadoGlobalFrame;
    bitNum : ENTER64;
    offset : ENTER65536;
  end record;

  -- Type fault-injection end
  type t_fiEnd is record
    cond   : t_endCond;
    field  : estadoGlobalFrame;
    bitNum : ENTER64;

    bc     : ENTER65536; -- bit count
  end record;

  -- Type fault-injection config
  type t_fiCfg is record
    value : t_fiValue;
    link  : t_fiLink;
    mode  : t_fiMode;

    startTrigger : t_trigger;
    endTrigger   : t_trigger;
    selTrigger   : t_trigger;

    -- start - end
    fiStart : t_fiStart;
    fiEnd   : t_fiEnd;
  end record;


  --
  --  Fault-injection config null values
  --

  constant value0 : t_fivalue := (
    fiType  => none,
    bfValue => x"AAAAAAAAAAAA",
    bfLong  => 56
  );

  constant trigger0 : t_trigger := (
    filter    => x"00000000000000",
    mask      => x"FFFFFFFFFFFFFF",
    mask_long => 56,
    field     => idle,
    link      => none,
    role      => dontCare,
    count     => 1
  );

  constant selTrigger0 : t_trigger := (
    filter    => x"00000000000000",
    mask      => x"FFFFFFFFFFFFFF",
    mask_long => 1,
    field     => idle,
    link      => none,
    role      => dontCare,
    count     => 1
  );

  constant fiStart0 : t_fiStart := (
    field  => idle,
    bitNum => 0,
    offset => 0
  );

  constant fiEnd0 : t_fiEnd := (
    cond   => fieldBit,

    field  => idField,
    bitNum => 11, -- non-existing bit

    bc     => 0
  );

  constant fiCfg0 : t_fiCfg := (
    value => value0,
    link  => port0dw,
    mode  => continuous,

    startTrigger => trigger0,
    endTrigger   => trigger0,
    selTrigger   => selTrigger0,

    fiStart => fiStart0,
    fiEnd   => fiEnd0
  );


  type t_fimValue is record
    bool  : boolean;
    value : std_logic;
  end record;

  type t_fimValues is array(t_fiLink) of t_fimValue;


  --
  -- Array definitions
  --

  -- Fault injection config array
  -- the first index is 0 to be able to debug fiCfgs
  type t_fiCfgs is array (1 to MAX_INJS) of t_fiCfg;

  -- fimInjValueSelector input array
  type t_cfgExecLinkValues is array (1 to MAX_INJS) of t_fiValue;

  -- injection types array depending on the link
  type t_cfgExecValues is array (t_fiLink) of t_cfgExecLinkValues;

  --
  type t_fiValues is array (t_fiLink) of t_fiValue;




  -- ================== --
  -- ==== Functions ==== --
  -- ================== --

  -- Parse injection config values
  function parseFiType       (value : in std_logic_vector(1 to 2)) return t_fiType;

  function parseLink         (value : in std_logic_vector(1 to 4)) return t_link;

  function parseFiMode       (value : in std_logic_vector(1 to 2)) return t_fiMode;

  function parseTriggerRole  (value : in std_logic_vector(1 to 2)) return t_triggerRole;

  function parseField        (value : in std_logic_vector(1 to 4)) return estadoGlobalFrame;
  function parseBoolean     (value : in std_logic              ) return boolean;


  --
  -- Debugging
  --

  type t_fiTypeSLV is array (t_fiType) of std_logic_vector(1 to 3);
  constant fiTypeSLV : t_fiTypeSLV := (
    stuckAtDominant  => "000",
    stuckAtRecessive => "001",
    bitFlipping      => "010",
    inverse          => "011",
    none             => "100"
  );

  type t_fiLinkSLV is array (t_fiLink) of std_logic_vector(1 to 4);
  constant fiLinkSLV : t_fiLinkSLV := (
    port0up => "0000",
    port0dw => "0001",

    port1up => "0010",
    port1dw => "0011",

    port2up => "0100",
    port2dw => "0101"
  );

end defInjection;


package body defInjection is

  --
  -- Command values parsing
  --

  -- parse t_fiType
  function parseFiType (value : in std_logic_vector(1 to 2)) return t_fiType is
  begin
    case value is
      when "00" => return stuckAtDominant;
      when "01" => return stuckAtRecessive;
      when "10" => return bitFlipping;
      when "11" => return inverse;

      when others => return stuckAtDominant;
    end case;
  end parseFiType;

  -- parse t_link
  function parseLink (value : in std_logic_vector(1 to 4)) return t_link is
  begin
    case value is
      when "0000" => return port0up;
      when "0001" => return port0dw;

      when "0010" => return port1up;
      when "0011" => return port1dw;

      when "0100" => return port2up;
      when "0101" => return port2dw;
      when "1111" => return none;

      when others => return none;
    end case;
  end parseLink;

  -- parse t_fiMode
  function parseFiMode (value : in std_logic_vector(1 to 2)) return t_fiMode is
  begin
    case value is
      when "00" => return continuous;
      when "01" => return iterative;
      when "10" => return singleShot;

      when others => return continuous;
    end case;
  end parseFiMode;

  -- parse t_triggerRole
  function parseTriggerRole (value : in std_logic_vector(1 to 2)) return t_triggerRole is
  begin
    case value is
    when "00" => return dontCare;
    when "01" => return tx;
    when "10" => return rx;

    when others => return dontCare;
    end case;
  end parseTriggerRole;

  -- parse estadoGlobalFrame
  function parseField (value : in std_logic_vector(1 to 4)) return estadoGlobalFrame is
  begin
    case value is
    when "0000" => return idle;
    when "0001" => return idField;
    when "0010" => return rtrField;
    when "0011" => return resField;

    when "0100" => return dlcField;
    when "0101" => return dataField;
    when "0110" => return crcField;
    when "0111" => return crcDelimField;

    when "1000" => return ackSlotField;
    when "1001" => return ackDelimField;
    when "1010" => return eofField;
    when "1011" => return interField;

    when "1100" => return errorFlag;
    when "1101" => return errorDelimiter;
    when others => return freeState1;
    end case;
  end parseField;

  -- parse boolean
  function parseBoolean (value : in std_logic) return boolean is
  begin
    case value is
      when '0' => return false;
      when '1' => return true;
      when others => return false;
    end case;
  end parseBoolean;

end defInjection;
```

D.1.5. defLogger.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

use work.defGeneral.all;
use work.defInteger.all;
use work.defStates.all;

package defLogger is

  --
  -- Constants
  --

  constant MAX_LOG_FRAMES : natural := 80;

  --type t_logCmd is (none, logPort0, logPort1, logPort2, logPort3, logPort4, logFrames);

  type t_logValue is (none, logFrames, logEOL);

  --
  -- Log codes
  --

  constant LOG_PORTS  : std_logic_vector(1 to 5) := "00000"; -- log port 0
  constant LOG_FRAMES : std_logic_vector(1 to 5) := "00001"; -- log frames
  constant LOG_EOL    : std_logic_vector(1 to 5) := "11111"; -- end of log


  --
  -- Types
  --

  type t_logStoredFrame is record
    p : t_port;

    valid  : std_logic;

    field  : estadoGlobalFrame;
    bitNum : ENTER64;

    dlc  : ENTER9;
    id   : t_id;
    data : t_byte;
  end record;

  constant logStoredFrame0 : t_logStoredFrame := (
    port0,

    '0',

    idle, 0,

    0, id0, data0(1 to 8)
  );

  subtype t_logStoredFramesCnt is natural range 0 to MAX_LOG_FRAMES;
  type t_logStoredFrames is array (1 to MAX_LOG_FRAMES) of t_logStoredFrame;

  constant logStoredFrames0 : t_logStoredFrames := (
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0,
    logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0, logStoredFrame0
  );

  function crcCalc (crc : in t_crc; b : in std_logic) return t_crc;

end defLogger;

package body defLogger is

  function crcCalc (crc : in t_crc; b : in std_logic) return t_crc is
    variable crcNext : std_logic;
    variable auxCrc  : t_crc;
  begin
    auxCrc := crc;

    crcNext := b xor auxCrc(1);

    -- sll
    auxCrc(1 to 14) := auxCrc(2 to 15);
    auxCrc(15) := '0';

    if crcNext = '1' then
      auxCrc := auxCrc xor "100010110011001"; -- 0x4599
    end if;

    return auxCrc;
  end crcCalc;

end defLogger;
```

D.1.6. defInteger.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.All;


package defInteger is

  subtype MASCARA8 is std_logic_vector(7 downto 0);

  subtype ENTER4   is integer range 0 to 3;
  subtype ENTER5   is integer range 0 to 4;
  subtype ENTER6   is integer range 0 to 5;
  subtype ENTER7   is integer range 0 to 6;
  subtype ENTER8   is integer range 0 to 7;
  subtype ENTER9   is integer range 0 to 8;
  subtype ENTER10  is integer range 0 to 9;
  subtype ENTER11  is integer range 0 to 10;
  subtype ENTER12  is integer range 0 to 11;
  subtype ENTER15  is integer range 0 to 14;
  subtype ENTER16  is integer range 0 to 15;

  subtype ENTER32  is integer range 0 to 31;

  subtype ENTER64  is integer range 0 to 63;
  subtype ENTER65  is integer range 0 to 64;

  subtype ENTER127 is integer range 0 to 126;
  subtype ENTER128 is integer range 0 to 127;

  subtype ENTER256 is integer range 0 to 255;

  subtype ENTER512 is integer range 0 to 511;

  subtype ENTER65536 is integer range 0 to 65535;

end defInteger;


package body defInteger is
end defInteger;
```

D.2. Coupler Module

D.2.1. couplerModule.vhd

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity couplerModule is
  port
  (
    iomPortContri_0  : in std_logic; -- Node 0
    iomPortContri_1  : in std_logic; -- Node 1
    iomPortContri_2  : in std_logic; -- Node 2

    iomPcContri      : in std_logic; -- PC

    logContri        : in std_logic; -- Logger

    efgTxSignal      : in std_logic; -- Error frame generator

    cplCoupledSignal : out std_logic; -- Nodes + PC
    cmpCoupledSignal : out std_logic  -- Nodes
  );
end couplerModule;


architecture Behavioral of couplerModule is

begin

  cplCoupledSignal <=
    iomPortContri_0 and
    iomPortContri_1 and
    iomPortContri_2 and
    iomPcContri     and
    logContri       and
    efgTxSignal;

  cmpCoupledSignal <=
    iomPortContri_0 and
    iomPortContri_1 and
    iomPortContri_2 and
    logContri       and
    efgTxSignal;

end Behavioral;
```

D.3. Can Module

D.3.1. canModule.vhd

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

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;

entity canModule is
  port (
  -- ======== Depuracion ========================================

    dbg_brpClk: out std_logic;

    dbg_habCRC: out std_logic;
    --dbg_estadoSincro          : out std_logic_vector(2 downto 0);
    --dbg_petSincro             : out std_logic;
    --dbg_salidaCntTseg_1       : out std_logic_vector(3 downto 0);
    --dbg_salidaCntIncTseg_1    : out std_logic_vector(1 downto 0);
    --dbg_salidaCntStaticTseg_1 : out std_logic_vector(3 downto 0);
    --dbg_salidaCntTseg_2       : out std_logic_vector(2 downto 0);
    dbg_synROut:  out std_logic;

    dbg_stuValueBitStuffWaited : out std_logic;
    dbg_gfmIniErrorFrame       : out std_logic;
    dbg_gfmLastBitEof          : out std_logic;
    dbg_gfmGlobalFrameState    : out std_logic_vector(3 downto 0);
    dbg_gfmErrorCRC            : out std_logic;

    dbg_estatErrorFrmGen : out std_logic;

  -- ============================================================

    -- Reset, oscilator and resultant signal
    reset : in std_logic;
    clk   : in std_logic;

    cplCoupledSignal: in std_logic;

    -- Contribuciones de los nodos
    iomPortContri_0 : in std_logic;
    iomPortContri_1 : in std_logic;
    iomPortContri_2 : in std_logic;

    -- Configuracion de capa fisica
    brp       : in std_logic_vector (5 downto 0);
    tsegment1 : in std_logic_vector (5 downto 0);
    tsegment2 : in std_logic_vector (2 downto 0);
    sjw       : in std_logic_vector (1 downto 0);
    sync      : in std_logic; -- 0: resincronizacion 'r' a 'd'; 1: ambos
    sam       : in std_logic; -- 0: 1 muestra; 1: 3 muestras

    synClkR : out std_logic;
    synClkT : out std_logic;

    -- Contribucion de transmision del hub
    efgTxSignal : out std_logic;
    -- State control signals
    stuBitStuffWaited      : out std_logic;
    stuValueBitStuffWaited : out std_logic;
    gfmLastBitEof          : out std_logic;
    gfmGlobalFrameState    : out estadoGlobalFrame;
    gfmGlobalBitNum        : out ENTER64;
    gfmErrorCRC            : out std_logic;

    -- Synchronized coupled signal
    synROut : out std_logic;

    -- Contribuciones de los nodos sincronizados
    synIomPortContri : out t_portSignalArray;

    nodesRole : out t_portSignalArray
  );
end canModule;

architecture Behavioral of canModule is

  -- Physical layer
  component phyLayerUnit is
  port (
  -- ======== Depuracion ========================================

    dbg_brpClk: out std_logic;

    --dbg_estadoSincro          : out std_logic_vector(2 downto 0);
    --dbg_petSincro             : out std_logic;
    --dbg_salidaCntTseg_1       : out std_logic_vector(3 downto 0);
    --dbg_salidaCntIncTseg_1    : out std_logic_vector(1 downto 0);
    --dbg_salidaCntStaticTseg_1 : out std_logic_vector(3 downto 0);
    --dbg_salidaCntTseg_2       : out std_logic_vector(2 downto 0);

  -- ============================================================

    sysReset : in std_logic;  -- Resetea senales clkt y clkr
    clk      : in std_logic;
    rx       : in std_logic;
    rx_0     : in std_logic;
    rx_1     : in std_logic;
    rx_2     : in std_logic;
    rx_3     : in std_logic;
    rx_4     : in std_logic;
    rx_5     : in std_logic;
    busIdle  : in std_logic;
    tnr      : in std_logic;

    brp       : in std_logic_vector (5 downto 0);
    tsegment1 : in std_logic_vector (5 downto 0);
    tsegment2 : in std_logic_vector (2 downto 0);
    sjw       : in std_logic_vector (1 downto 0);
    sync      : in std_logic; -- 0: resincronizacion 'r' a 'd'; 1: ambos
    sam       : in std_logic; -- 0: 1 muestra; 1: 3 muestras

    synROut   : out std_logic;
    synROut_0 : out std_logic;
    synROut_1 : out std_logic;
    synROut_2 : out std_logic;
    synROut_3 : out std_logic;
    synROut_4 : out std_logic;
    synROut_5 : out std_logic;
    synClkR : out std_logic;
    synClkT : out std_logic
  );
  end component;


  -- rxCAN
  component rxCAN is
  port(
  -- ======== Depuracion ========================================

    dbg_habCRC              : out std_logic;
    dbg_gfmGlobalFrameState : out std_logic_vector(3 downto 0);

  -- ===========================================================

    sysReset : in std_logic;
    synClkR  : in std_logic;
    synROut  : in std_logic;

    stuBitStuffWaited      : out std_logic;
    stuValueBitStuffWaited : out std_logic;
    gfmIniErrorFrame       : out std_logic;
    gfmLastBitEof          : out std_logic;
    gfmGlobalFrameState    : out estadoGlobalFrame;
    gfmGlobalBitNum        : out ENTER64;
    gfmErrorCRC            : out std_logic
  );
  end component;


  -- Error frame generator
  component errorFrameGenerator is
  port (
  -- ======== Depuracion ========================================

    dbg_estatErrorFrmGen: out std_logic;

  -- ===========================================================

    sysReset : in std_logic;
    synClkT  : in std_logic;
    gfmIniErrorFrame : in std_logic;

    efgTxSignal : out std_logic
  );
  end component;

  component nodeRoleModule is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;

    isStuffBit : in std_logic;

    iomPortContri : in std_logic;
    coupledSignal : in std_logic;
```

```
    -- '0' transmisor, '1' receptor
    nodeRole : out std_logic
  );
  end component;


  --
  -- Signals
  --

  -- Physical layer
  signal auxSynROut: std_logic;
  signal auxSynClkR: std_logic;
  signal auxSynClkT: std_logic;

  signal auxBusIdle : std_logic;
  signal auxTnr     : std_logic;

  signal auxSynIomPortContri : t_portSignalArray;

  -- rxCAN
  signal auxGfmGlobalFrameState    : estadoGlobalFrame;
  signal auxGfmGlobalBitNum        : ENTER64;
  signal auxStuBitStuffWaited      : std_logic;
  signal auxStuValueBitStuffWaited : std_logic;

  signal auxGfmIniErrorFrame : std_logic;
  signal auxGfmLastBitEof    : std_logic;
  signal auxGfmErrorCRC      : std_logic;

begin

  --
  -- Mapeo de senales de depuracion
  --

  dbg_stuValueBitStuffWaited <= auxStuValueBitStuffWaited;
  dbg_gfmIniErrorFrame       <= auxGfmIniErrorFrame;
  dbg_gfmLastBitEof          <= auxGfmLastBitEof;
  dbg_gfmErrorCRC            <= auxGfmErrorCRC;


  --
  -- Output signals
  --

  stuBitStuffWaited      <= auxStuBitStuffWaited;
  stuValueBitStuffWaited <= auxStuValueBitStuffWaited;
  gfmLastBitEof          <= auxGfmLastBitEof;
  gfmGlobalFrameState    <= auxGfmGlobalFrameState;
  gfmErrorCRC            <= auxGfmErrorCRC;

  synClkR <= auxSynClkR;
  synClkT <= auxSynClkT;

  synROut <= auxSynROut;
  synIomPortContri <= auxSynIomPortContri;


  --
  -- Calculo de senales para unidad de capa fisica
  --
```

```
with auxGfmGlobalFrameState select
  auxBusIdle <=
    '1' when idle,
    '0' when others;

with auxGfmGlobalFrameState select
  auxTnr <=
    '1' when errorFlag | errorDelimiter,
    '0' when others;


-- Unidad de capa fisica
physicalLayerUnit:
  phyLayerUnit
    port map (
    -- ======== Depuracion ========================================

      dbg_brpClk  =>  dbg_brpClk,
      --dbg_estadoSincro          => dbg_estadoSincro,
      --dbg_petSincro             => dbg_petSincro,
      --dbg_salidaCntTseg_1       => dbg_salidaCntTseg_1,
      --dbg_salidaCntIncTseg_1    => dbg_salidaCntIncTseg_1,
      --dbg_salidaCntStaticTseg_1 => dbg_salidaCntStaticTseg_1,
      --dbg_salidaCntTseg_2       => dbg_salidaCntTseg_2,

    -- ============================================================

      sysReset => reset,
      clk      => clk,
      rx       => cplCoupledSignal,
      rx_0     => iomPortContri_0,
      rx_1     => iomPortContri_1,
      rx_2     => iomPortContri_2,
      rx_3     => '1',
      rx_4     => '1',
      rx_5     => '1',

      busIdle => auxBusIdle,
      tnr     => auxTnr,

      brp       => brp,
      tsegment1 => tsegment1,
      tsegment2 => tsegment2,
      sjw       => sjw,
      sync      => sync,
      sam       => sam,

      synROut   => auxSynROut,
      synROut_0 => auxSynIomPortContri(port0),
      synROut_1 => auxSynIomPortContri(port1),
      synROut_2 => auxSynIomPortContri(port2),

      synClkR   => auxSynClkR,
      synClkT   => auxSynClkT
    );

-- Unidad de monitorizacion de recepcion de CAN
rxCANUnit:
  rxCAN
    port map (
    -- ======== Depuracion ========================================
      dbg_habCRC              => dbg_habCRC,
      dbg_gfmGlobalFrameState => dbg_gfmGlobalFrameState,

    -- ===============================================================

      sysReset => reset,
      synClkR  => auxSynClkR,
      synROut  => auxSynROut,

      stuBitStuffWaited      => auxStuBitStuffWaited,
      stuValueBitStuffWaited => auxStuValueBitStuffWaited,

      gfmIniErrorFrame       => auxGfmIniErrorFrame,
      gfmLastBitEof          => auxGfmLastBitEof,

      gfmGlobalFrameState    => auxGfmGlobalFrameState,
      gfmGlobalBitNum        => auxGfmGlobalBitNum,

      gfmErrorCRC            => auxGfmErrorCRC
    );

gfmGlobalBitNum <= auxGfmGlobalBitNum;

-- Unidad de generacion de frames de error
errorFrameGeneratorUnit:
  errorFrameGenerator
    port map (
    -- ======== Depuracion ========================================

      dbg_estatErrorFrmGen  => dbg_estatErrorFrmGen,

    -- ===============================================================

      sysReset => reset,
      synClkT  => auxSynClkT,
      gfmIniErrorFrame => auxGfmIniErrorFrame,

      efgTxSignal => efgTxSignal
    );


-- Port type
nodesRoleModule_generation:
for p in t_port generate
  nodesRoleModule_array : nodeRoleModule
    port map(
      reset => reset,
      clkR  => auxSynClkR,

      frameState  => auxGfmGlobalFrameState,
      frameBitNum => auxGfmGlobalBitNum,

      isStuffBit => auxStuBitStuffWaited,

      coupledSignal => auxSynROut,

      iomPortContri => auxSynIomPortContri(p),

      -- '0' transmisor, '1' receptor
      nodeRole => nodesRole(p)
    );
  end generate;

end Behavioral;
```

D.3.2. phyLayerUnit.vhd

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

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity phyLayerUnit is
port (
  -- ======== Depuracion ==========
  dbg_brpClk: out std_logic;

  --dbg_estadoSincro:   out std_logic_vector(2 downto 0);
  -- dbg_PetSincro:     out std_logic;
  --dbg_salidaCntTseg_1:    out std_logic_vector(3 downto 0);
  --dbg_salidaCntIncTseg_1: out std_logic_vector(1 downto 0);
  --dbg_salidaCntStaticTseg_1: out std_logic_vector(3 downto 0);
  --dbg_salidaCntTseg_2:    out std_logic_vector(2 downto 0);

  -- ======================
  sysReset: in std_logic; -- Resetea senales clkt y clkr
  clk:    in std_logic;
  rx:   in std_logic;
  rx_0:   in std_logic;
  rx_1:   in std_logic;
  rx_2:   in std_logic;
  rx_3:   in std_logic;
  rx_4:   in std_logic;
  rx_5:   in std_logic;
  busIdle:  in std_logic;
  tnr:    in  std_logic;

  brp:        in std_logic_vector (5 downto 0);
  tsegment1:  in std_logic_vector (5 downto 0);
  tsegment2:  in std_logic_vector (2 downto 0);
  sjw:        in std_logic_vector (1 downto 0);
  sync:       in std_logic; -- 0: resincronizacion 'r' a 'd'; 1: ambos
  sam:        in std_logic; -- 0: 1 muestra; 1: 3 muestras

  synROut:    out std_logic;
  synROut_0:  out std_logic;
  synROut_1:  out std_logic;
  synROut_2:  out std_logic;
  synROut_3:  out std_logic;
  synROut_4:  out std_logic;
  synROut_5:  out std_logic;

  synClkR:  out std_logic;
  synClkT:  out std_logic
);
end phyLayerUnit;

architecture Behavioral of phyLayerUnit is

  -- Ens proporciona el Rellotge de Sistema
  -- a partir d'un oscil'lador
  component baudRatePre
  port(
      reset:  in std_logic;
      clk:  in std_logic;
      brp:  in std_logic_vector (5 downto 0);

      brpClk: out std_logic
    );
  end component;

  -- Sincronitza el Canal i proporciona Rellotges
  -- pel transmissor i pel receptor (Tbit)
  component synchronizer is
  port(
      -- ======== Depuracion ==========

      --dbg_estadoSincro: out std_logic_vector(2 downto 0);
      --dbg_PetSincro: out std_logic;
      --dbg_salidaCntTseg_1: out std_logic_vector(3 downto 0);
      --dbg_salidaCntIncTseg_1: out std_logic_vector(1 downto 0);
      --dbg_salidaCntStaticTseg_1: out std_logic_vector(3 downto 0);
      --dbg_salidaCntTseg_2: out std_logic_vector(2 downto 0);

      -- ======================
      sysReset: in std_logic; -- Resetea senales clkt y clkr
      rx:   in std_logic; -- Entrada del "bus" (1 bit)
      rx_0:   in std_logic;
      rx_1:   in std_logic;
      rx_2:   in std_logic;
      rx_3:   in std_logic;
      rx_4:   in std_logic;
      rx_5:   in std_logic;
      clk:    in std_logic; -- Rellotge del sistema
      busIdle:  in std_logic; -- "hard Syncronization" habilitacion de sincronizacion dura
      tnr:    in std_logic; -- Transmetre / no REBRE
      tsegment1:  in std_logic_vector (5 downto 0);
      tsegment2:  in std_logic_vector (2 downto 0);
      sjw:    in std_logic_vector (1 downto 0);
      sync:   in std_logic; -- 0: resincronizacion 'r' a 'd'; 1: ambos
      sam:    in std_logic; -- 0: 1 muestra; 1: 3 muestras

      synROut:  out std_logic;  -- Sortida al destuff (1 bit)
      synROut_0:  out std_logic;
      synROut_1:  out std_logic;
      synROut_2:  out std_logic;
      synROut_3:  out std_logic;
      synROut_4:  out std_logic;
      synROut_5:  out std_logic;

      synClkR:  out std_logic;  -- Clk per agafar la sortida valida
      synClkT:  out std_logic -- Clk pel transmissor
    );
  end component;

  signal auxBrpClk: std_logic;

begin

  dbg_brpClk <= auxBrpClk;

  baudRatePreUnit:
    baudRatePre
      port map (
        reset =>  sysReset,
        clk =>  clk,
        brp =>  brp,

        brpClk  =>  auxBrpClk
      );

  synchronizerUnit:
    synchronizer
      port map (
        -- ======== Depuracion ==========

        --dbg_estadoSincro      =>  dbg_estadoSincro,
        --dbg_PetSincro     =>  dbg_PetSincro,
        --dbg_salidaCntTseg_1   =>  dbg_salidaCntTseg_1,
        --dbg_salidaCntIncTseg_1    =>  dbg_salidaCntIncTseg_1,
        --dbg_salidaCntStaticTseg_1 =>  dbg_salidaCntStaticTseg_1,
        --dbg_salidaCntTseg_2   =>  dbg_salidaCntTseg_2,

        -- ======================
        sysReset  =>  sysReset,
        rx       => rx,
        rx_0    =>  rx_0,
        rx_1    =>  rx_1,
        rx_2    =>  rx_2,
        rx_3    =>  rx_3,
        rx_4    =>  rx_4,
        rx_5    =>  rx_5,
        clk   =>  auxBrpClk,
        busIdle   =>  busIdle,
        tnr   =>  tnr,
        tsegment1 =>  tsegment1,
        tsegment2 =>  tsegment2,
        sjw   =>  sjw,
        sync    =>  sync,
        sam   =>  sam,

        synROut   =>  synROut,
        synROut_0 =>  synROut_0,
        synROut_1 =>  synROut_1,
        synROut_2 =>  synROut_2,
        synROut_3 =>  synROut_3,
        synROut_4 =>  synROut_4,
        synROut_5 =>  synROut_5,

        synClkR   =>  synClkR,
        synClkT   =>  synClkT
      );
end Behavioral;
```

D.3.3. rxCAN.vhd

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

use work.defStates.all;
use work.defInteger.all;

entity rxCAN is
port(
  -- ======== Depuracion ==========
  dbg_habCRC: out std_logic;
  dbg_gfmGlobalFrameState:  out std_logic_vector(3 downto 0);
  -- ======================
  sysReset: in std_logic;
  synClkR:  in std_logic;
  synROut:  in std_logic;

  stuBitStuffWaited:  out std_logic;
  stuValueBitStuffWaited: out std_logic;
  gfmIniErrorFrame: out std_logic;
  gfmLastBitEof:    out std_logic;
  gfmGlobalFrameState : out estadoGlobalFrame;
  gfmGlobalBitNum     : out ENTER64;
  gfmErrorCRC:    out std_logic
);
end rxCAN;


architecture Behavioral of rxCAN is

  -- StuffUnit
  component stuffUnit is
  port(
    sysReset: in std_logic;
    gfmStuffEnabling: in std_logic; -- Habilitacion de stuff
    synClkR:  in std_logic; -- Reloj de recepcion
    synROut:  in std_logic; -- Valor de recepcion a muestrear

    stuBitStuffWaited:  out std_logic;  -- Indica que el proximo bit debe ser de stuff
    stuValueBitStuffWaited: out std_logic -- Indica el valor esperado del proximo bit de stuff
  );
  end component;

  -- GlobalFrameMonitor
  component globalFrameMonitor is
  port(
    dbg_gfmGlobalFrameState: out std_logic_vector(3 downto 0);

    sysReset:   in std_logic;
    stuBitStuffWaited:  in std_logic;
    stuValueBitStuffWaited: in std_logic;
    synClkR:    in std_logic;
    synROut:    in std_logic;
    cscCRCValue:    in std_logic_vector(14 downto 0);

    gfmStuffEnabling: out std_logic;
    gfmIniErrorFrame: out std_logic;
    gfmLastBitEof:    out std_logic;
    gfmGlobalFrameState : out estadoGlobalFrame;
    gfmGlobalBitNum     : out ENTER64;
    gfmErrorCRC:    out std_logic
  );
  end component;

  -- CalSeqCRC
  component calSeqCRC is
  port(
    -- ======== Depuracion ==========
    dbg_habCRC: out std_logic;
    -- =====================
    sysReset: in std_logic;
    synClkR:  in std_logic;
    synROut:  in std_logic;
    gfmGlobalFrameState:  in estadoGlobalFrame;
    stuBitStuffWaited:  in std_logic;

    cscCRCValue:  out std_logic_vector (14 downto 0)
  );
  end component;

  -- PortType
  component portTypeModule is
  port(
    sysReset : in std_logic;
    synClkR  : in std_logic;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;

    isStuffBit : in std_logic;

    iomPortContri : in std_logic;
    coupledSignal : in std_logic;

    -- '0' transmisor, '1' receptor
    portType : out std_logic
  );
  end component;


  -- Senales de enlace
  signal auxGfmGlobalFrameState : estadoGlobalFrame;
  signal auxStuStuffBitWaited: std_logic;
  signal auxStuValueBitStuffWaited: std_logic;

  signal auxGfmStuffEnabling: std_logic;
  signal auxGfmErrorCRC: std_logic;
  signal cscCRCValue: std_logic_vector(14 downto 0);

begin

  -- Mapeo de senales de salida
  gfmGlobalFrameState <= auxGfmGlobalFrameState;
  stuBitStuffWaited <= auxStuStuffBitWaited;
  stuValueBitStuffWaited <= auxStuValueBitStuffWaited;
  gfmErrorCRC <= auxGfmErrorCRC;

  -- Unidad de calculo de bit de stuff
  bitStuffUnit:
    stuffUnit
      port map (
        sysReset  =>  sysReset,
        gfmStuffEnabling  =>  auxGfmStuffEnabling,
        synClkR     =>  synClkR,
        synROut     =>  synROut,

        stuBitStuffWaited =>  auxStuStuffBitWaited,
        stuValueBitStuffWaited  =>  auxStuValueBitStuffWaited
      );


  -- Unidad de globalizacion de trama global
  globalFrameMonitorUnit:
    globalFrameMonitor
      port map (
        dbg_gfmGlobalFrameState => dbg_gfmGlobalFrameState,

        sysReset    => sysReset,
        stuBitStuffWaited => auxStuStuffBitWaited,
        stuValueBitStuffWaited  => auxStuValueBitStuffWaited,
        synClkR     => synClkR,
        synROut     => synROut,
        cscCRCValue   => cscCRCValue,

        gfmStuffEnabling  => auxGfmStuffEnabling,
        gfmIniErrorFrame  => gfmIniErrorFrame,
        gfmLastBitEof   => gfmLastBitEof,
        gfmGlobalFrameState => auxGfmGlobalFrameState,
        gfmGlobalBitNum      => gfmGlobalBitNum,
        gfmErrorCRC   => auxGfmErrorCRC
      );


  -- Unidad de calculo de secuencia para CRC
  calSeqCRCUnit:
    calSeqCRC
      port map (
        -- ======== Depuracion ==========
        dbg_habCRC  => dbg_habCRC,
        -- ======================
        sysReset  => sysReset,
        synClkR   => synClkR,
        synROut   => synROut,
        gfmGlobalFrameState => auxGfmGlobalFrameState,
        stuBitStuffWaited => auxStuStuffBitWaited,

        cscCRCValue   => cscCRCValue
      );

end Behavioral;
```

D.3.4. errorFrameGenerator.vhd

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

use work.defInteger.all;

--  Uncomment the following lines to use the declarations that are
--  provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity errorFrameGenerator is
port (
    -- Senales de depuracion --
    dbg_estatErrorFrmGen: out std_logic;

    -------------------------------
    sysReset:   in std_logic;
    synClkT:    in std_logic;
    gfmIniErrorFrame: in std_logic;

    efgTxSignal:    out std_logic
    );
end errorFrameGenerator;

architecture Behavioral of errorFrameGenerator is

  type estadoErrorFrmGen is (idle, errorFlag);

  signal estatErrorFrmGen: estadoErrorFrmGen;
  signal countD: ENTER7;

begin

  -- Asignacion a senal de depuracion de estado
  with estatErrorFrmGen select
    dbg_estatErrorFrmGen <= '0' when idle,
          '1' when errorFlag;

  -- Proceso de control de la unidad
  process (synClkT, sysReset)
  begin
    if sysReset = '1' then
      estatErrorFrmGen <= idle;
      efgTxSignal <= '1';

    elsif synClkT'event and synClkT = '1' then
      case (estatErrorFrmGen) is
        when idle =>
          if gfmIniErrorFrame = '0' then
            efgTxSignal <= '1';

          else
            estatErrorFrmGen <= errorFlag;
            efgTxSignal <= '0';
            countD <= 1;
          end if;

        when errorFlag =>
```

```
          if gfmIniErrorFrame = '1' then
            efgTxSignal <= '0';
            countD <= 1;

          elsif countD < 6 then
            countD <= countD + 1;
            efgTxSignal <= '0';

          else
            estatErrorFrmGen <= idle;
            efgTxSignal <= '1';
          end if;

        when others =>
          null;
      end case;
    end if;
  end process;

end Behavioral;
```

D.3.5. nodeRoleModule.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 03/05/2011
-- Description:
--    Reading the contribution of a device checks its role (transmitter or
--    receiver).
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;

entity nodeRoleModule is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;

    isStuffBit : in std_logic;

    iomPortContri : in std_logic;
    coupledSignal : in std_logic;

    -- '0' transmisor, '1' receptor
    nodeRole : out std_logic
  );
end nodeRoleModule;


architecture Behavioral of nodeRoleModule is

begin

  process (clkR, reset) is
  begin

    if reset = '1' then
      nodeRole  <= '1';

    elsif clkR'event and clkR = '1' and
        isStuffBit = '0'          then

      case (frameState) is
        when idle =>
          if iomPortContri = '0' then
            nodeRole <= '0';
          else
            nodeRole <= '1';
          end if;

        when idField =>
          if iomPortContri = '0' then
```

```
            nodeRole <= '0';
          elsif iomPortContri = '1' and coupledSignal = '0' then
            nodeRole <= '1';
          end if;

        when eofField =>
          if frameBitNum = fieldLong(eofField)-1 then
            nodeRole <= '1';
          end if;

        when others => null;
      end case;

    end if;
  end process;

end Behavioral;
```

D.4. Fault Injection Module

D.4.1. faultInjectionModule.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.defGeneral.all;
use work.defInteger.all;
use work.defStates.all;
use work.defInjection.all;
use work.defNCC.all;

entity faultInjectionModule is
  port(
    --
    -- Input Signals
    --
    reset : in std_logic;
    clkR  : in std_logic;
    clkT  : in std_logic;
    clk   : in std_logic;

    -- injected link signals
    iLinks : in t_linkSignalArray;

    nodesRole : in t_portSignalArray;

    -- State of globalFrameMonitorUnit (current state of the
    -- resultant frame)
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;

    -- Next bit, stuff bit
    isStuffBit: in std_logic;

    --
    -- Injection data
    --

    -- Downlink
    fimBoolTx_0     : out boolean;
    fimValueTx_0    : out std_logic;

    fimBoolTx_1     : out boolean;
    fimValueTx_1    : out std_logic;

    fimBoolTx_2     : out boolean;
    fimValueTx_2    : out std_logic;

    -- Uplink
    fimBoolRx_0     : out boolean;
    fimValueRx_0    : out std_logic;

    fimBoolRx_1     : out boolean;
    fimValueRx_1    : out std_logic;

    fimBoolRx_2     : out boolean;
    fimValueRx_2    : out std_logic;

    --
    -- 7 seg
    --
    dbg_7seg_fim    : out integer;
    dbg_7seg_dp_fim : out std_logic

  );
end faultInjectionModule;

architecture Behavioral of faultInjectionModule is

  --
  -- Components
  --

  -- fimFrameReader

  component nccFrameReader is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    sampleFrame : out std_logic;

    id   : out t_id;
    data : out std_logic_vector(1 to 64)
  );
  end component;

  -- fimFrameFilter

  component nccFrameFilter is
  port(
    nfrSampleFrame : in std_logic;

    id       : in t_id;
    local_id : in t_id;

    nccMode : in t_nccMode;

    sampleFrame : out std_logic
  );
  end component;

  -- fimCoordinator

  component nccCoordinator is
  port(
    reset : in std_logic;
    clk   : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 8);
    enterConfigMode : out std_logic;
    nccMode         : out t_nccMode
  );
  end component;

  -- fimCmdInterpreter

  component fimCmdInterpreter is
  port(
    enable : in std_logic;

    reset : in std_logic;
    clk   : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 64);

    sampleFiCfg : out std_logic;
    fiCfg       : out t_fiCfg
  );
  end component;

  -- fimConfigsStorage

  component fimConfigsStorage is
  port(
    reset : in std_logic;
    clk   : in std_logic;

    sampleFiCfg : in std_logic;
    fiCfg       : in t_fiCfg;

    fiCfgs       : out t_fiCfgs;
    fiCfgs_count : out ENTER11
  );
  end component;

  -- fimExecuter

  component fimExecuter is
  port(
    enable : in std_logic;
    clk    : in std_logic;
    clkT   : in std_logic;
    clkR   : in std_logic;

    fiCfgs       : in t_fiCfgs;
    fiCfgs_count : in ENTER11; -- cambiar tipo

    iLinks : in t_linkSignalArray;

    nodesRole : in t_portSignalArray;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    fimValues : out t_fimValues
  );
  end component;
```

```
  --
  -- Signals
  --

  signal fiReset    : std_logic;

  signal cfgEnable  : std_logic;
  signal execEnable : std_logic;

  -- Aux fimFrameReader
  signal ffrSampleFrame : std_logic;
  signal auxId   : t_id;
  signal auxData : t_data;

  -- Aux fimFrameFilter
  signal fffSampleFrame : std_logic;

  -- Aux fimCoordinator
  signal auxEnterConfigMode : std_logic;
  signal auxFimMode : t_nccMode;

  -- Aux fimCmdInterpreter
  signal fciSampleFiCfg : std_logic;
  signal auxFiCfg       : t_fiCfg;

  -- Aux fimConfigsStorage
  signal auxFiCfgs       : t_fiCfgs;
  signal auxFiCfgs_count : ENTER11;

  -- Aux fimExecuter
  signal fimValues : t_fimValues;

begin

  -- Execution reset
  fiReset <=
    '1' when reset = '1' or auxEnterConfigMode = '1' else
    '0';

  cfgEnable <=
    '1' when auxFimMode = configMode else
    '0';

  execEnable <=
    '1' when auxFimMode = execMode else
    '0';


  --
  -- Port mapping
  --

  -- fimFrameReader

  fimFrameReaderUnit:
    nccFrameReader
      port map(
        reset => reset,
        clkR  => clkR,

        coupledSignal => iLinks(none),
        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        sampleFrame => ffrSampleFrame,

        id   => auxId,
        data => auxData
      );

  -- fimFrameFilter

  fimFrameFilterUnit:
    nccFrameFilter
      port map(
        nfrSampleFrame => ffrSampleFrame,

        id       => auxId,
        local_id => HUB_FIM_ID,

        nccMode => auxFimMode,

        sampleFrame => fffSampleFrame
      );

  -- fimCoordinator

  fimCoordinatorUnit:
    nccCoordinator
      port map(
        reset => reset,
        clk   => clk,

        sampleFrame => fffSampleFrame,
        data => auxData(1 to 8),

        enterConfigMode => auxEnterConfigMode,
        nccMode => auxFimMode
      );

  -- fimCmdInterpreter

  fimCmdInterpreterUnit:
    fimCmdInterpreter
      port map(
        enable  => cfgEnable,

        reset => fiReset,
        clk   => clk,

        sampleFrame => fffSampleFrame,
        data => auxData,

        sampleFiCfg => fciSampleFiCfg,
        fiCfg       => auxFiCfg
      );

  -- fimConfigStorage

  fimConfigsStorageUnit:
    fimConfigsStorage
      port map(
        reset => fiReset,
        clk   => clk,

        sampleFiCfg => fciSampleFiCfg,
        fiCfg       => auxFiCfg,

        fiCfgs       => auxFiCfgs,
        fiCfgs_count => auxFiCfgs_count
      );

  -- fimExecuter

  fimExecuterUnit:
    fimExecuter
      port map(
        enable => execEnable,
        clk    => clk,
        clkT   => clkT,
        clkR   => clkR,

        fiCfgs       => auxFiCfgs,
        fiCfgs_count => auxFiCfgs_count,

        iLinks => iLinks,

        nodesRole => nodesRole,

        -- fimTriggerExecEnabler and fimChecker
        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        -- Injection type values
        fimValues => fimValues
      );


  --
  -- Output signals
  --

  fimBoolTx_0     <= fimValues(port0dw).bool;
  fimValueTx_0    <= fimValues(port0dw).value;

  fimBoolTx_1     <= fimValues(port1dw).bool;
  fimValueTx_1    <= fimValues(port1dw).value;

  fimBoolTx_2     <= fimValues(port2dw).bool;
  fimValueTx_2    <= fimValues(port2dw).value;

  -- Uplink
  fimBoolRx_0     <= fimValues(port0up).bool;
  fimValueRx_0    <= fimValues(port0up).value;

  fimBoolRx_1     <= fimValues(port1up).bool;
  fimValueRx_1    <= fimValues(port1up).value;

  fimBoolRx_2     <= fimValues(port2up).bool;
  fimValueRx_2    <= fimValues(port2up).value;


  --
```

```
  -- 7 seg
  --

  dbg_7seg_fim <= nccModeNAT(auxFimMode);
  dbg_7seg_dp_fim <=
    '1' when auxFiCfgs_count > 0 else
    '0';

end Behavioral;
```

D.4.2. nccFrameReader.vhd

```
--------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 17/03/2011
-- Description:
--    Captures frame bits and extracts its id and data.
--    Uses a sample signal to inform that values can be readed.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defInteger.all;
use work.defStates.all;
use work.defInjection.all;

entity nccFrameReader is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    sampleFrame : out std_logic;

    id   : out t_id;
    data : out t_data
  );
end nccFrameReader;

architecture Behavioral of nccFrameReader is
begin

  process (reset, clkR) is
    variable auxId   : std_logic_vector(0 to 10);
    variable auxData : std_logic_vector(0 to 63);

    variable newFrame : boolean;
    variable error    : boolean;

  begin
    if reset = '1' then
      auxId   := id0;
      auxData := data0; -- 64 0 bits

      sampleFrame <= '0';
      error := false;

    elsif clkR'event and clkR = '1' and isStuffBit = '0' then
      sampleFrame <= '0';

      case frameState is
        -- initialize aux values
        when idle =>
          auxId   := id0;
          auxData := data0; -- 64 0 bits

        -- capture identifier
        when idField =>
          auxId(frameBitNum) := coupledSignal;

        -- capture data
        when dataField =>
          auxData(frameBitNum) := coupledSignal;

        when errorFlag =>
          error := true;

        -- Outputs values at the end of the frame
        -- unless an error has occurred
        when interField =>
          if frameBitNum = 0 then
            if error = true then
              error := false;

            else
              id   <= auxId;
              data <= auxData;

              sampleFrame <= '1';
            end if;
          end if;

        when others => null;
      end case;
    end if;
  end process;

end Behavioral;
```

D.4.3. nccFrameFilter.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 03/05/2011
-- Description:
--    Generates a new sampleFrame which filters by id and fimMode
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defNCC.all;

entity nccFrameFilter is
  port(
    nfrSampleFrame : in std_logic;

    id       : in t_id;
    local_id : in t_id;

    nccMode : in t_nccMode;

    sampleFrame : out std_logic
  );
end nccFrameFilter;

architecture Behavioral of nccFrameFilter is

begin

  sampleFrame <=
    '1' when nfrSampleFrame = '1' and
        ((id = BROAD_ID) or
         (id = local_id and nccMode /= execMode))
      else
    '0';

end Behavioral;
```

D.4.4. nccCoordinator.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 17/03/2011
-- Description:
--    Reads "data" from nccFrameReader using a filtered sample signal from
--    nccFrameFilter. Then interprets the switch mode command within (if exists).
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defNCC.all;

entity nccCoordinator is
  port(
    reset : in std_logic;
    clk   : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 8);

    enterConfigMode : out std_logic;
    nccMode         : out t_nccMode
  );
end nccCoordinator;

architecture Behavioral of nccCoordinator is

  signal auxNCCMode : t_nccMode;

begin

  --
  -- Output Signals
  --

  nccMode <= auxNCCMode;


  process (reset, clk) is
    variable lastSampleFrame : std_logic;

  begin
    if reset = '1' then
      lastSampleFrame := '0';
      auxNCCMode <= configMode;

    elsif clk'event and clk = '1' then
      -- new frame available
      enterConfigMode <= '0';

      if lastSampleFrame = '0' and sampleFrame = '1' and
         data(1 to 3)    = CMD_MCH                 then

        case data(4 to 8) is
          when MC_ECM =>
            auxNCCMode <= configMode;
            enterConfigMode <= '1';

          when MC_EIM =>
            if auxNCCMode = configMode then
              auxNCCMode <= idleMode;
            end if;

          when MC_EWM =>
            if auxNCCMode = configMode then
              auxNCCMode <= wfwMode;
            end if;

          when MC_EEM =>
            if auxNCCMode = wfwMode then
              auxNCCMode <= execMode;
            end if;

          when others => null;
        end case;

      end if;

      lastSampleFrame := sampleFrame;
    end if;

  end process;

end Behavioral;
```

D.4.5. fimCmdInterpreter.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 17/03/2011
-- Description:
--    Reads "data" from fimFrameReader using a filtered sample signal from
--    fimFrameFilter. Then interprets the config command within (if exists).
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.defInjection.all;
use work.defNCC.all;
use work.defStates.all;
use work.defInteger.all;

entity fimCmdInterpreter is
  port(
    enable  : in std_logic;

    reset   : in std_logic;
    clk : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 64);

    sampleFiCfg : out std_logic;
    fiCfg       : out t_fiCfg
  );
end fimCmdInterpreter;


architecture Behavioral of fimCmdInterpreter is
  signal fiCfgReady : std_logic;
begin

  sampleFiCfg <= sampleFrame and fiCfgReady;

  process (reset, clk) is
    variable lastSampleFrame : std_logic;
    variable auxFaultInjCfgs_count : ENTER11;

    variable auxFiCfgReady : std_logic;

    variable active : boolean;

  begin
    if reset = '1' then
      lastSampleFrame := '0';

      auxFiCfgReady := '0';
      fiCfgReady    <= '0';

      fiCfg <= fiCfg0;

      active := false;
```

```
elsif clk'event and clk = '1' and
    enable = '1'           then

  -- initialize cmdInterpreter status
  if auxFiCfgReady = '1' and sampleFrame = '0' then
    auxFiCfgReady := '0';
    active := false;
  end if;

  -- new frame available
  if lastSampleFrame = '0' and sampleFrame = '1' and
    data(1 to 3)    = CMD_CFG                    then

    active := true;

    case data(4 to 8) is
      -----------
      -- VALUE --
      -----------

      when PARAM_VALUE_TYPE =>
        -- 2 bits
        fiCfg.value.fiType <= parseFiType(data(15 to 16));

      when PARAM_VALUE_BFVALUE =>
        -- long: 6 bits => 0 to 63
        -- data must be a natural value within the range 2 to 56
        -- semantic checking in fim coordinator is needed
        -- value: 6 bytes
        fiCfg.value.bfLong  <= to_integer(unsigned(data(11 to 16)));
        fiCfg.value.bfValue <= data(17 to 64);

      ----------
      -- LINK --
      ----------

      when PARAM_LINK =>
        -- 4 bits
        -- injLink must be a valid injectable link
        -- semantic checking in fim coordinator is needed
        fiCfg.link <= parseLink(data(13 to 16));

      ----------
      -- MODE --
      ----------

      when PARAM_MODE =>
        -- 2 bits
        fiCfg.mode <= parseFiMode(data(15 to 16));

      -------------------
      -- START TRIGGER --
      -------------------

      when PARAM_START_TRIGGER_FILTER =>
        -- 7 bytes
        fiCfg.startTrigger.filter <= data(09 to 64);

      when PARAM_START_TRIGGER_MASK =>
        -- 7 bytes
        fiCfg.startTrigger.mask <= data(09 to 64);
```

```
when PARAM_START_TRIGGER_MASK_LONG =>
  -- 6 bits => 0 to 63
  -- data must be a natural value within the range 1 to 56
  -- semantic checking in fim coordinator is needed
  fiCfg.startTrigger.mask_long <= to_integer(unsigned(data(11 to 16)));

when PARAM_START_TRIGGER_FIELD =>
  -- 4 bits
  fiCfg.startTrigger.field <= parseField(data(13 to 16));

when PARAM_START_TRIGGER_LINK =>
  -- 4 bits
  -- triggerLink must be a valid injectable link or none (coupled signal)
  -- semantic checking in fim coordinator is needed
  fiCfg.startTrigger.link <= parseLink(data(13 to 16));

when PARAM_START_TRIGGER_ROLE =>
  -- 2 bits
  fiCfg.startTrigger.role <= parseTriggerRole(data(15 to 16));

when PARAM_START_TRIGGER_COUNT =>
  -- 2 bytes => 0 to 65535
  fiCfg.startTrigger.count <= to_integer(unsigned(data(09 to 24)));

-----------------
-- END TRIGGER --
-----------------

when PARAM_END_TRIGGER_FILTER =>
  -- 7 bytes
  fiCfg.endTrigger.filter <= data(09 to 64);

when PARAM_END_TRIGGER_MASK =>
  -- 7 bytes
  fiCfg.endTrigger.mask <= data(09 to 64);

when PARAM_END_TRIGGER_MASK_LONG =>
  -- 6 bits => 0 to 63
  -- data must be a natural value within the range 1 to 56
  -- semantic checking in fim coordinator is needed
  fiCfg.endTrigger.mask_long <= to_integer(unsigned(data(11 to 16)));

when PARAM_END_TRIGGER_FIELD =>
  -- 4 bits
  fiCfg.endTrigger.field <= parseField(data(13 to 16));

when PARAM_END_TRIGGER_LINK =>
  -- 4 bits
  -- triggerLink must be a valid injectable link or none (coupled signal)
  -- semantic checking in fim coordinator is needed
  fiCfg.endTrigger.link <= parseLink(data(13 to 16));

when PARAM_END_TRIGGER_ROLE =>
  -- 2 bits
  fiCfg.endTrigger.role <= parseTriggerRole(data(15 to 16));

when PARAM_END_TRIGGER_COUNT =>
  -- 2 bytes => 0 to 65535
  fiCfg.endTrigger.count <= to_integer(unsigned(data(09 to 24)));

-----------------
```

```
-- SEL TRIGGER --
-----------------

when PARAM_SEL_TRIGGER_FILTER =>
  -- 7 bytes
  fiCfg.selTrigger.filter <= data(09 to 64);

when PARAM_SEL_TRIGGER_MASK =>
  -- 7 bytes
  fiCfg.selTrigger.mask <= data(09 to 64);

when PARAM_SEL_TRIGGER_MASK_LONG =>
  -- 6 bits => 0 to 63
  -- data must be a natural value within the range 1 to 56
  -- semantic checking in fim coordinator is needed
  fiCfg.selTrigger.mask_long <= to_integer(unsigned(data(11 to 16)));

when PARAM_SEL_TRIGGER_FIELD =>
  -- 4 bits
  fiCfg.selTrigger.field <= parseField(data(13 to 16));

when PARAM_SEL_TRIGGER_LINK =>
  -- 4 bits
  -- triggerLink must be a valid injectable link or none (coupled signal)
  -- semantic checking in fim coordinator is needed
  fiCfg.selTrigger.link <= parseLink(data(13 to 16));

when PARAM_SEL_TRIGGER_ROLE =>
  -- 2 bits
  fiCfg.selTrigger.role <= parseTriggerRole(data(15 to 16));

-----------
-- START --
-----------

when PARAM_START_FIELD =>
  -- 4 bits
  fiCfg.fiStart.field <= parseField(data(13 to 16));

when PARAM_START_BIT =>
  -- 6 bits => 0 to 63
  fiCfg.fiStart.bitNum <= to_integer(unsigned(data(11 to 16)));

when PARAM_START_OFFSET =>
  -- 2 bytes => 0 to 65535
  fiCfg.fiStart.offset <= to_integer(unsigned(data(09 to 24)));


---------
-- END --
---------

when PARAM_END_FIELD =>
  -- 4 bits
  fiCfg.fiEnd.field <= parseField(data(13 to 16));
  fiCfg.fiEnd.cond <= fieldBit;

when PARAM_END_BIT =>
  -- 6 bits => 0 to 63
  fiCfg.fiEnd.bitNum <= to_integer(unsigned(data(11 to 16)));

when PARAM_END_BC =>
```

```
          -- 2 bytes => 0 to 65535
          fiCfg.fiEnd.bc <= to_integer(unsigned(data(09 to 24)));
          fiCfg.fiEnd.cond <= bitCount;


        ------------------
        -- LAST COMMAND --
        ------------------

        when PARAM_EOC =>
          auxFiCfgReady := '1';

        when others => null;

      end case;

    -- initialize fiCfg when none fiCfg is active
    elsif active = false then
      fiCfg <= fiCfg0;

    end if;

    lastSampleFrame := sampleFrame;
    fiCfgReady <= auxFiCfgReady;
  end if;

 end process;

end Behavioral;
```

D.4.6. fimConfigsStorage.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 23/03/2011
-- Description:
--    Reads fault-injection configurations from fimCmdInterpreter and stores them
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defInteger.all;
use work.defInjection.all;

entity fimConfigsStorage is
  port(
    reset   : in std_logic;
    clk : in std_logic;

    sampleFiCfg : in std_logic;
    fiCfg       : in t_fiCfg;

    fiCfgs       : out t_fiCfgs;
    fiCfgs_count : out ENTER11
  );
end fimConfigsStorage;

architecture Behavioral of fimConfigsStorage is
begin

  process (reset, clk) is
    variable lastSampleFiCfg : std_logic;
    variable auxfiCfgs_count : ENTER11;

  begin
    if reset = '1' then
      lastSampleFiCfg := '0';
      auxfiCfgs_count := 0;

    elsif clk'event and clk = '1' then

      -- new fault-injection config available
      if lastSampleFiCfg = '0' and sampleFiCfg = '1' and
         fiCfg /= fiCfg0                              and
         auxfiCfgs_count < MAX_INJS                  then
        auxfiCfgs_count := auxfiCfgs_count + 1;
        fiCfgs(auxfiCfgs_count) <= fiCfg;
      end if;

      lastSampleFiCfg := sampleFiCfg;
    end if;

    fiCfgs_count <= auxfiCfgs_count;
  end process;

end Behavioral;
```

D.4.7. fimExecuter.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 22/03/2011
-- Description:
--    Captures frame bits and extracts its id and data.
--    Uses a sample signal to inform that values can be readed
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;
use work.defInjection.all;
use work.defNCC.all;

entity fimExecuter is
  port(
    enable : in std_logic;
    clk    : in std_logic;
    clkT   : in std_logic;
    clkR   : in std_logic;

    fiCfgs       : in t_fiCfgs;
    fiCfgs_count : in ENTER11; -- cambiar tipo

    iLinks : in t_linkSignalArray;

    nodesRole : in t_portSignalArray;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    fimValues : out t_fimValues
  );
end fimExecuter;

architecture Behavioral of fimExecuter is

  --
  -- Components
  --

  -- fimCfgExecuter

  component fimCfgExecuter is
  port(
    enable  : in std_logic;
    clk  : in std_logic;
    clkT : in std_logic;
    clkR : in std_logic;

    fiCfg : in t_fiCfg;

    -- fimCfgExecEnabler
    fiCfg_num    : in ENTER11; -- cambiar tipo
    fiCfgs_count : in ENTER11; -- cambiar tipo


    -- injected link signals
    iLinks : in t_linkSignalArray;

    -- Role de los nodos
    nodesRole : in t_portSignalArray;

    -- fimTriggerExecEnabler and fimChecker
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    -- Output signals
    port0UpValue : out t_fiValue;
    port0DwValue : out t_fiValue;

    port1UpValue : out t_fiValue;
    port1DwValue : out t_fiValue;

    port2UpValue : out t_fiValue;
    port2DwValue : out t_fiValue
  );
  end component;


  -- fimInjValueSelector

  component fimInjValueSelector is
  port(
    fiValues : in  t_cfgExecLinkValues;
    fiValue  : out t_fiValue
  );
  end component;


  -- fimValueGenerator

  component fimValueGenerator is
  port(
    -- Input Signals
    enable : in std_logic;
    clkT   : in std_logic;

    coupledSignal : in std_logic;

    fiValue : in t_fiValue;

    -- Output Signals
    fimValue : out t_fimValue
  );
  end component;


  --
  -- Signals
  --

  signal cfgExecValues : t_cfgExecValues;
  signal fiValues : t_fiValues;
```

```
begin

  --
  -- Component instanciations
  --

  -- fimCfgExecuter generation

  fimCfgExecuter_generation:
  for fiCfg_idx in 1 to MAX_INJS generate
    fimCfgExecuterUnit_array : fimCfgExecuter
      port map(
        enable => enable,
        clk  => clk,
        clkT => clkT,
        clkR => clkR,

        fiCfg => fiCfgs(fiCfg_idx),

        -- fimCfgExecEnabler
        fiCfg_num    => fiCfg_idx,
        fiCfgs_count => fiCfgs_count,

        -- injected link signals
        iLinks => iLinks,

        -- Role del puerto
        nodesRole => nodesRole,

        -- fimTriggerExecEnabler and fimChecker
        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        -- Output signals
        port0UpValue => cfgExecValues(port0up)(fiCfg_idx),
        port0DwValue => cfgExecValues(port0dw)(fiCfg_idx),

        port1UpValue => cfgExecValues(port1up)(fiCfg_idx),
        port1DwValue => cfgExecValues(port1dw)(fiCfg_idx),

        port2UpValue => cfgExecValues(port2up)(fiCfg_idx),
        port2DwValue => cfgExecValues(port2dw)(fiCfg_idx)
      );
  end generate;


  -- fimInjValueSelectorGeneration

  fimInjValueSelector_generation:
  for link in t_fiLink generate
    fimInjValueSelector_array : fimInjValueSelector
      port map(
        fiValues => cfgExecValues(link),
        fiValue  => fiValues(link)
      );
  end generate;


  -- fimValueGenerator
  fimValueGenerator_generation:
  for link in t_fiLink generate
    fimValueGenerator_array : fimValueGenerator
      port map(
        -- Input Signals
        enable => enable,
        clkT   => clkT,

        coupledSignal => iLinks(none),

        fiValue => fiValues(link),

        -- Output Signals
        fimValue => fimValues(link)
      );
  end generate;

end Behavioral;
```

D.4.8. fimCfgExecuter.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 24/03/2011
-- Description:
--    Implements a frameCounter, checker and a manager for every fiCfg in the
--    fiCfgs array. Outputs a value for each link.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;
use work.defInjection.all;
use work.defNCC.all;

entity fimCfgExecuter is
  port(
    enable : in std_logic;
    clk    : in std_logic;
    clkT   : in std_logic;
    clkR   : in std_logic;

    fiCfg : in t_fiCfg;

    -- fimCfgExecEnabler
    fiCfg_num    : in ENTER11; -- pasar a t_fiIdx
    fiCfgs_count : in ENTER11; -- pasar a t_fiIdx

    -- injected link signals
    iLinks : in t_linkSignalArray;

    -- Role del puerto
    nodesRole : in t_portSignalArray;

    -- fimTriggerExecEnabler and fimChecker
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    -- Output signals
    port0UpValue : out t_fiValue;
    port0DwValue : out t_fiValue;

    port1UpValue : out t_fiValue;
    port1DwValue : out t_fiValue;

    port2UpValue : out t_fiValue;
    port2DwValue : out t_fiValue
  );
end fimCfgExecuter;

architecture Behavioral of fimCfgExecuter is

  --
```

```
-- Components
--

-- fimTriggerCounter
component fimTriggerCounter is
port(
  enable : in std_logic;
  clkR   : in std_logic;

  fiLink  : in t_fiLink;
  trigger : in t_trigger;

  -- injected link signals
  iLinks : in t_linkSignalArray;

  -- Role del puerto
  nodesRole : in t_portSignalArray;

  frameState  : in estadoGlobalFrame;
  frameBitNum : in ENTER64;
  isStuffBit  : in std_logic;

  triggerCount : out ENTER65536;
  currentTriggerFrame : out boolean
);
end component;


-- bitCounter
component bitCounter is
port(
  reset : in std_logic;
  clkR  : in std_logic;

  enable : in boolean;

  bitCount : out ENTER65536
);
end component;


-- fimChecker
component fimChecker is
port(
  enable : in std_logic;
  clkT   : in std_logic;

  -- fiCfg data
  fiCfg : in t_fiCfg;

  -- frame data
  frameState  : in estadoGlobalFrame;
  frameBitNum : in ENTER64;
  isStuffBit  : in std_logic;
  currentTriggerFrame : in boolean;

  -- trigger data
  startTriggerCount : in ENTER65536;
  endTriggerCount   : in ENTER65536;

  endBitCount       : in ENTER65536;
  inject : out boolean
);
end component;


-- fimOffsetDelayer
component fimOffsetDelayer is
port(
  clkR   : in std_logic;

  inject : in boolean;
  offset : in ENTER65536;

  delayedInject : out boolean
);
end component;


-- fimManager
component fimManager is
port(
  enable : in std_logic;

  inject : in boolean;

  value : in t_fiValue;
  link  : in t_fiLink;

  -- Output signals
  port0UpValue : out t_fiValue;
  port0DwValue : out t_fiValue;

  port1UpValue : out t_fiValue;
  port1DwValue : out t_fiValue;

  port2UpValue : out t_fiValue;
  port2DwValue : out t_fiValue
);
end component;

signal fiCfg_enable : std_logic;

-- Aux fimTriggerCounter
signal startTriggerCount : ENTER65536;
signal endTriggerCount   : ENTER65536;
signal currentTriggerFrame : boolean;

-- Aux fimEndTriggerCounter
signal hasInjected : std_logic;
signal fetc_enable  : std_logic;

-- Aux fimBitCounter
signal febc_reset  : std_logic;
signal febc_enable : boolean;
signal endTriggerEvent : boolean;
signal bitCount : ENTER65536;

-- Aux fimChecker
signal inject : boolean;

-- Aux fimOffsetDelayer
signal delayedInject : boolean;
```

```
begin

  fiCfg_enable <=
    '1' when enable        = '1'        and
           fiCfgs_count >= fiCfg_num else
    '0';


  -- fimStartTriggerCounter
  fimStartTriggerCounterUnit :
    fimTriggerCounter
      port map(
        enable => fiCfg_enable,
        clkR   => clkR,

        fiLink  => fiCfg.link,
        trigger => fiCfg.startTrigger,

        -- injected link signals
        iLinks => iLinks,

        -- Role del puerto
        nodesRole => nodesRole,

        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        triggerCount => startTriggerCount
        --currentTriggerFrame =>
      );


  -- fimEndTriggerCounter
  fimEndTriggerCounterUnit :
    fimTriggerCounter
      port map(
        enable => fetc_enable,
        clkR   => clkR,

        fiLink  => fiCfg.link,
        trigger => fiCfg.endTrigger,

        -- injected link signals
        iLinks => iLinks,

        -- Role del puerto
        nodesRole => nodesRole,

        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        triggerCount => endTriggerCount
        --currentTriggerFrame =>
      );

  -- store delayedInject to enable endTrigger
  process(fiCfg_enable, clk) is
  begin
    if fiCfg_enable = '0' then
      hasInjected <= '0';
    elsif clk'event and clk = '1' then
      if delayedInject then
        hasInjected <= '1';
      end if;
    end if;
  end process;

  fetc_enable <=
    fiCfg_enable and hasInjected;


  -- fimSelTriggerCounter
  fimSelTriggerCounterUnit :
    fimTriggerCounter
      port map(
        enable => fiCfg_enable,
        clkR   => clkR,

        fiLink  => fiCfg.link,
        trigger => fiCfg.selTrigger,

        -- injected link signals
        iLinks => iLinks,

        -- Role del puerto
        nodesRole => nodesRole,

        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        --triggerCount =>
        currentTriggerFrame => currentTriggerFrame
      );


  fimEndBitCounterUnit :
    bitCounter
      port map(
        reset => febc_reset,
        clkR  => clkR,

        enable   => febc_enable,

        bitCount => bitCount
      );

  -- fimEndBitCounter
  febc_reset <=
    '1' when fiCfg_enable  = '0'     or
          delayedInject = false else
    '0';

  -- fimEndBitEnable
  febc_enable <=
    true when (fiCfg.mode     /= continuous  and
            delayedInject   = true     )   or
           (fiCfg.mode      = continuous  and
            endTriggerEvent = true     ) else
    false;
```

```
-- store endTriggerEvent to enable fimEndBitCounter
process(fiCfg_enable, clk) is
begin
  if fiCfg_enable = '0' then
    endTriggerEvent <= false;
  elsif clk'event and clk = '1' then
    if endTriggerCount = fiCfg.endTrigger.count then
      endTriggerEvent <= true;
    end if;
  end if;
end process;


-- fimChecker
fimCheckerUnit :
  fimChecker
    port map(
      enable => fiCfg_enable,
      clkT   => clkT,

      -- fiCfg data
      fiCfg => fiCfg,

      -- frame data
      frameState  => frameState,
      frameBitNum => frameBitNum,
      isStuffBit  => isStuffBit,

      -- trigger data
      startTriggerCount   => startTriggerCount,
      endTriggerCount     => endTriggerCount,
      currentTriggerFrame => currentTriggerFrame,

      endBitCount       => bitCount,

      inject => inject
    );

-- fimOffsetDelayer
fimOffsetDelayerUnit:
  fimOffsetDelayer
    port map(
      clkR   => clkR,

      inject => inject,
      offset => fiCfg.fiStart.offset,

      delayedInject => delayedInject
    );

-- fimManager
fimManagerUnit :
  fimManager
    port map(
      enable  => fiCfg_enable,

      inject => delayedInject,

      value => fiCfg.value,
      link  => fiCfg.link,

      -- Output signals
```

```
        port0UpValue => port0UpValue,
        port0DwValue => port0DwValue,

        port1UpValue => port1UpValue,
        port1DwValue => port1DwValue,

        port2UpValue => port2UpValue,
        port2DwValue => port2DwValue
      );

end Behavioral;
```

D.4.9. fimTriggerCounter.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 06/04/2011
-- Description:
--    Implements a trigger counter. It has one module for each trigger counter.
--    Outputs the triggerCount of the current enabled trigger.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defInteger.all;
use work.defStates.all;
use work.defInjection.all;

entity fimTriggerCounter is
  port(
    enable : in std_logic;
    clkR   : in std_logic;

    fiLink  : in t_fiLink;
    trigger : in t_trigger;

    -- injected link signals
    iLinks : in t_linkSignalArray;

    -- Role del puerto
    nodesRole : in t_portSignalArray;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    triggerCount : out ENTER65536;
    currentTriggerFrame : out boolean
  );
end fimTriggerCounter;


architecture Behavioral of fimTriggerCounter is

  signal filteredSignal : std_logic;
  signal nodeRole       : std_logic; -- 0 tx, 1 rx

begin

  -- Link filter
  filteredSignal <= iLinks(trigger.link);

  -- Port type
  nodeRole <= nodesRole(to_port(fiLink));


  process (enable, clkR) is
    variable query : boolean;
    variable auxTriggerCount : ENTER65536;
    variable auxCurrentTriggerFrame : boolean;

  begin
    if enable = '0' then
      auxTriggerCount := 0;
      query := true;
      auxCurrentTriggerFrame := false;

    elsif clkR'event and clkR = '1' and
        isStuffBit = '0'         then

      if frameState = idle then
        auxCurrentTriggerFrame := false;
      end if;

      if frameState = trigger.field and frameBitNum < FILTER_LONG then
        if frameBitNum = 0 then
          query := true;
        end if;

        if (filteredSignal and trigger.mask(frameBitNum)) /= trigger.filter(frameBitNum)
        or (nodeRole = '1' and trigger.role = tx)
        or (nodeRole = '0' and trigger.role = rx)
        then
          query := false;
        end if;

        if frameBitNum = trigger.mask_long-1 and query = true then
          auxTriggerCount := auxTriggerCount + 1;
          auxCurrentTriggerFrame := true;
        end if;
      end if;
    end if;

    triggerCount        <= auxTriggerCount;
    currentTriggerFrame <= auxCurrentTriggerFrame;
  end process;

end Behavioral;
```

D.4.10. bitCounter.vhd

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

use work.defInteger.all;

entity bitCounter is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    enable : in boolean;

    bitCount : out ENTER65536
  );
end bitCounter;

architecture Behavioral of bitCounter is

begin

  process (reset, clkR) is
    variable auxBitCount : ENTER65536;

  begin
    if reset = '1' then
      auxBitCount := 0;

    elsif clkR'event and clkR = '1' then
      if enable then
        auxBitCount := auxBitCount + 1;
      end if;
    end if;

    bitCount <= auxBitCount;
  end process;

end Behavioral;
```

D.4.11. fimChecker.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 24/03/2011
-- Description:
--    Checks the condition to enable the current injection.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defInteger.all;
use work.defStates.all;
use work.defInjection.all;

entity fimChecker is
  port(
    enable : in std_logic;
    clkT   : in std_logic;

    -- fiCfg data
    fiCfg : in t_fiCfg;

    -- frame data
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;
    currentTriggerFrame : in boolean;

    -- trigger data
    startTriggerCount : in ENTER65536;
    endTriggerCount   : in ENTER65536;

    -- bit count data
    endBitCount : in ENTER65536;

    inject : out boolean
  );
end fimChecker;

architecture Behavioral of fimChecker is

begin

  process (enable, clkT) is
    variable auxInject : boolean;
    variable fiState   : t_fiState;

  begin
    if enable = '0' then
      auxInject := false;
      fiState   := waiting;

    elsif clkT'event and clkT = '1' then

      case fiState is

        -------------------
```

```
-- state waiting --
-------------------

when waiting =>

  -- disabled transition
  if
    endTriggerCount = fiCfg.endTrigger.count
  then
    fiState   := disabled;
    auxInject := false;

  -- enabled transition
  elsif
    -- trigger count condition
    ((fiCfg.mode = continuous and startTriggerCount  = fiCfg.startTrigger.count) or
     (fiCfg.mode = singleShot and startTriggerCount  = fiCfg.startTrigger.count) or
     (fiCfg.mode = iterative  and startTriggerCount >= fiCfg.startTrigger.count))
  and
    -- selective condition
    ((fiCfg.mode /= iterative) or
     (fiCfg.mode  = iterative and currentTriggerFrame))
  and
    -- field / bit condition
    isStuffBit = '0' and
    (frameState   = fiCfg.fiStart.field and
     frameBitNum  = fiCfg.fiStart.bitNum  )
  and
    -- end bc = 0
    not (fiCfg.fiEnd.cond = bitCount and
         fiCfg.fiEnd.bc   = 0          )
  then
    fiState   := enabled;
    auxInject := true;

  -- waiting transition
  else
    fiState   := waiting;
    auxInject := false;
  end if;


-------------------
-- state enabled --
-------------------

when enabled =>

  -- disabled transition
  if
    -- trigger count condition
    ((fiCfg.mode = singleShot                                                ) or
     (fiCfg.mode = continuous and endTriggerCount >= fiCfg.endTrigger.count) or
     (fiCfg.mode = iterative  and endTriggerCount >= fiCfg.endTrigger.count))
  and
    (-- end field-bit condition
     (fiCfg.fiEnd.cond = fieldBit             and
      frameState       = fiCfg.fiEnd.field    and
      frameBitNum      = fiCfg.fiEnd.bitNum     )
     or
     -- end bit count condition
     (fiCfg.fiEnd.cond = bitCount             and
              endBitCount      = fiCfg.fiEnd.bc         )
            )
          then
            fiState   := disabled;
            auxInject := false;

          -- waiting transition
          elsif
            (fiCfg.mode      = iterative               and
             endTriggerCount < fiCfg.endTrigger.count   )
          and
            (-- end trigger count condition
             (fiCfg.fiEnd.cond = fieldBit                and
              frameState       = fiCfg.fiEnd.field       and
              frameBitNum      = fiCfg.fiEnd.bitNum        )
             or
             -- end bit count condition
             (fiCfg.fiEnd.cond = bitCount                and
              endBitCount      = fiCfg.fiEnd.bc            )
            )

          then
            fiState   := waiting;
            auxInject := false;

          -- enabled transition
          else
            fiState   := enabled;
            auxInject := true;
          end if;


        --------------------
        -- state disabled --
        --------------------

        when disabled =>
          null;

      end case;

    end if;

    inject <= auxInject;
  end process;

end Behavioral;
```

D.4.12. fimOffsetDelayer.vhd

```
--------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 31/05/2011
-- Description:
--    Implements a delayer for the fimChecker module to perform the offset's
--    feature.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defInteger.all;

entity fimOffsetDelayer is
  port(
    clkR   : in std_logic;

    inject : in boolean;
    offset : in ENTER65536;

    delayedInject : out boolean
  );
end fimOffsetDelayer;

architecture Behavioral of fimOffsetDelayer is

  -- fimBitCounter
  component bitCounter is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    enable : in boolean;

    bitCount : out ENTER65536
  );
  end component;

  signal bitCount        : ENTER65536;
  signal enableInjection : boolean;
  signal enableCounter   : boolean;
  signal resetCounter    : std_logic;

begin

  with enableInjection select
    delayedInject <=
      inject when true,
      false  when others;

  enableInjection <=
    inject = true and
    bitCount = offset;

  enableCounter <=
    inject = true and
    not enableInjection and
    offset /= 0;

  with inject select
    resetCounter <=
      '0' when true,
      '1' when false;


  -- fimOffsetitCounter
  fimOffsetBitCounterUnit :
    bitCounter
      port map(
        enable  => enableCounter,
        reset => resetCounter,

        clkR  => clkR,

        bitCount => bitCount
      );

end Behavioral;
```

D.4.13. fimManager.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 24/03/2011
-- Description:
--    Checks whether the fimChecker enables the injection and parses its value
--    to the appropriate linkValue.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defInjection.all;

entity fimManager is
  port(
    enable : in std_logic;

    inject : in boolean;

    value : in t_fiValue;
    link  : in t_fiLink;

    -- Output signals
    port0UpValue : out t_fiValue;
    port0DwValue : out t_fiValue;

    port1UpValue : out t_fiValue;
    port1DwValue : out t_fiValue;

    port2UpValue : out t_fiValue;
    port2DwValue : out t_fiValue
  );
end fimManager;

architecture Behavioral of fimManager is

begin

  -- port0
  port0UpValue <=
    value when
      enable = '1'   and
      inject = true  and
      link = port0up else
    value0;

  port0DwValue <=
    value when
      enable = '1'   and
      inject = true  and
      link = port0dw else
    value0;

  -- port1
  port1UpValue <=
    value when
```

```
      enable = '1'   and
      inject = true  and
      link = port1up else
    value0;

  port1DwValue <=
    value when
      enable = '1'   and
      inject = true  and
      link = port1dw else
    value0;

  -- port2
  port2UpValue <=
    value when
      enable = '1'   and
      inject = true  and
      link = port2up else
    value0;

  port2DwValue <=
    value when
      enable = '1'   and
      inject = true  and
      link = port2dw else
    value0;

end Behavioral;
```

D.4.14. fimInjValueSelector.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 24/03/2011
-- Description:
--    Selects the value to be injected in a especified link. The value provided
--    by a greater fimCfgExecUnit has more priority.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defInjection.all;

entity fimInjValueSelector is
  port(
    fiValues : in  t_cfgExecLinkValues;
    fiValue  : out t_fiValue
  );
end fimInjValueSelector;

architecture Behavioral of fimInjValueSelector is

begin

  fiValue <=
--    fiValues(10) when fiValues(10) /= value0 else
--    fiValues( 9) when fiValues( 9) /= value0 else
--    fiValues( 8) when fiValues( 8) /= value0 else
--    fiValues( 7) when fiValues( 7) /= value0 else
--    fiValues( 6) when fiValues( 6) /= value0 else
    fiValues( 5) when fiValues( 5) /= value0 else
    fiValues( 4) when fiValues( 4) /= value0 else
    fiValues( 3) when fiValues( 3) /= value0 else
    fiValues( 2) when fiValues( 2) /= value0 else
    fiValues( 1) when fiValues( 1) /= value0 else
    value0;

end Behavioral;
```

D.4.15. fimValueGenerator.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 22/03/2011
-- Description:
--    Captures frame bits and extracts its id and data.
--    Uses a sample signal to inform that values can be readed
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defInjection.all;
use work.defInteger.all;

entity fimValueGenerator is
  port(
    -- Input Signals
    enable  : in std_logic;
    clkT : in std_logic;

    coupledSignal : in std_logic;

    fiValue : in t_fiValue;

    -- Output Signals
    fimValue : out t_fimValue
  );
end fimValueGenerator;


architecture Behavioral of fimValueGenerator is

  signal bfValue : std_logic;

begin

  with fiValue.fiType select
    fimValue.bool <=
      false when none,
      true  when others;

  with fiValue.fiType select
    fimValue.value <=
      '0'               when stuckAtDominant,
      '1'               when stuckAtRecessive,
      bfValue           when bitFlipping,
      not coupledSignal when inverse,
      '1'               when others;

  -- bit-flipping value generator
  process (enable, clkT) is
    variable bfIdx : ENTER64;
  begin
    if enable = '0' then
      bfIdx := 0;

    elsif clkT'event and clkT = '1' then
```

```
      if fiValue.fiType = bitFlipping then
        bfIdx := bfIdx + 1;
        if bfIdx > (fiValue.bfLong-1) then
          bfIdx := 0;
        end if;

      else
        bfIdx := 0;
      end if;

    end if;

    bfValue <= fiValue.bfValue(bfIdx);
  end process;

end Behavioral;
```

D.5. Logger Module

D.5.1. loggerModule.vhd

```
--------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 15/06/2011
-- Description:
--    Stores logging data during the execution of a test. At the end of the test
--    when the PC asks for log data this module transmits it using CAN frames.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

use work.defGeneral.all;
use work.defNCC.all;
use work.defLogger.all;
use work.defStates.all;
use work.defInteger.all;

entity loggerModule is
  port(
    --
    -- Input Signals
    --
    reset : in std_logic;
    clk   : in std_logic;
    clkR : in std_logic;
    clkT : in std_logic;

    coupledSignal : in std_logic;

    nodesRole : in t_portSignalArray;

    -- State of frame
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    --
    -- Log contribution
    --
    logContri : out std_logic;

    --
    -- 7 seg
    --
    dbg_7seg_log    : out integer;
    dbg_7seg_dp_log : out std_logic
  );
end loggerModule;

architecture Behavioral of loggerModule is

  --
  -- Components
```

```
--

-- logComunicator

component logComunicator is
port(
  reset : in std_logic;
  clkR  : in std_logic;
  clk   : in std_logic;

  coupledSignal : in std_logic;

  -- State current frame
  frameState  : in estadoGlobalFrame;
  frameBitNum : in ENTER64;
  isStuffBit  : in std_logic;

  -- Output signals
  logMode      : out t_nccMode;
  sampleReport : out std_logic
);
end component;


-- logGatherer

component logGatherer is
port(
  enable : in std_logic;

  reset : in std_logic;
  clkR  : in std_logic;

  --
  -- input log signals
  --

  nodesRole : in t_portSignalArray;

  -- logFrameGatherer
  coupledSignal : in std_logic;
  frameState    : in estadoGlobalFrame;
  frameBitNum   : in ENTER64;
  isStuffBit    : in std_logic;

  --
  -- output log signals
  --

  -- logFrameGatherer
  logStoredFrames     : out t_logStoredFrames;
  logStoredFrames_cnt : out t_logStoredFramesCnt
);
end component;


-- logReporter

component logReporter is
port(
  enable : in std_logic;
```

```
    reset : in std_logic;
    clk   : in std_logic;
    clkT  : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;
    frameState    : in estadoGlobalFrame;
    frameBitNum   : in ENTER64;
    isStuffBit    : in std_logic;

    sampleReport : in std_logic;

    -- log signals
    logStoredFrames     : in t_logStoredFrames;
    logStoredFrames_cnt : in t_logStoredFramesCnt;

    contri : out std_logic
  );
  end component;


  --
  -- Signals
  --

  signal execEnable : std_logic;
  signal cfgEnable  : std_logic;
  signal wfwReset   : std_logic;

  -- logComunicator
  signal auxLogMode      : t_nccMode;
  signal auxSampleReport : std_logic;

  -- logGatherer
  signal auxLogStoredFrames      : t_logStoredFrames;
  signal auxLogStoredFrames_cnt  : t_logStoredFramesCnt;

  -- logReporter

begin

  --
  -- Inner signals
  --

  execEnable <=
    '1' when auxLogMode = execMode else
    '0';

  cfgEnable <=
    '1' when auxLogMode = configMode else
    '0';

  wfwReset <=
    '1' when reset      = '1'       or
          auxLogMode = wfwMode else
    '0';


  --
  -- Port mapping
  --
```

```
-- logComunicator

logComunicatorUnit :
  logComunicator
    port map(
      reset => reset,
      clkR  => clkR,
      clk   => clk,

      coupledSignal => coupledSignal,

      -- State current frame
      frameState  => frameState,
      frameBitNum => frameBitNum,
      isStuffBit  => isStuffBit,

      -- Output signals
      logMode      => auxLogMode,
      sampleReport => auxSampleReport
    );

-- logGatherer

logGathererUnit :
  logGatherer
    port map(
      enable => execEnable,

      reset => wfwReset,
      clkR  => clkR,

      --
      -- input log signals
      --

      nodesRole => nodesRole,

      -- logFrameGatherer
      coupledSignal => coupledSignal,
      frameState    => frameState,
      frameBitNum   => frameBitNum,
      isStuffBit    => isStuffBit,

      --
      -- output log signals
      --

      -- logFrameGatherer
      logStoredFrames     => auxLogStoredFrames,
      logStoredFrames_cnt => auxLogStoredFrames_cnt
    );


-- logReporter

logReporterUnit :
  logReporter
    port map(
      enable  => cfgEnable,

      reset => reset,
        clk   => clk,
        clkT  => clkT,
        clkR  => clkR,

        coupledSignal => coupledSignal,
        frameState    => frameState,
        frameBitNum   => frameBitNum,
        isStuffBit    => isStuffBit,

        sampleReport => auxSampleReport,

        -- log signals
        logStoredFrames      => auxLogStoredFrames,
        logStoredFrames_cnt  => auxLogStoredFrames_cnt,

        contri => logContri
      );

end Behavioral;
```

D.5.2. logComunicator.vhd

```
--------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 15/06/2011
-- Description:
--
--
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defNCC.all;
use work.defLogger.all;
use work.defStates.all;
use work.defInteger.all;

entity logComunicator is
  port(
    reset : in std_logic;
    clkR  : in std_logic;
    clk   : in std_logic;

    coupledSignal : in std_logic;

    -- State current frame
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;

    -- Next bit, stuff bit
    isStuffBit: in std_logic;

    -- Output signals
    logMode      : out t_nccMode;
    sampleReport : out std_logic
  );
end logComunicator;

architecture Behavioral of logComunicator is

  --
  -- Components
  --

  -- logFrameReader

  component nccFrameReader is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;

    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;
    sampleFrame : out std_logic;

    id   : out t_id;
    data : out t_data
  );
  end component;

  -- logFrameFilter

  component nccFrameFilter is
  port(
    nfrSampleFrame : in std_logic;

    id       : in t_id;
    local_id : in t_id;

    nccMode : in t_nccMode;

    sampleFrame : out std_logic
  );
  end component;

  -- logCoordinator

  component nccCoordinator is
  port(
    reset : in std_logic;
    clk   : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 8);

    enterConfigMode : out std_logic;
    nccMode         : out t_nccMode
  );
  end component;

  -- logCmdInterpreter

  component logCmdInterpreter is
  port(
    enable  : in std_logic;

    reset : in std_logic;
    clk   : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 8);

    sampleReport : out std_logic
  );
  end component;

  --
  -- Signals
  --

  signal logReset  : std_logic;
  signal cfgEnable : std_logic;

  -- Aux logFrameReader
  signal lfrSampleFrame : std_logic;
```

```
  signal auxId   : t_id;
  signal auxData : t_data;

  -- Aux logFrameFilter
  signal lffSampleFrame : std_logic;

  -- Aux logCoordinator
  signal auxEnterConfigMode : std_logic;
  signal auxLogMode : t_nccMode;

  -- Aux logCmdInterpreter


begin

  --
  -- Output signals
  --

  logMode <= auxLogMode;


  --
  -- Inner signals
  --

  -- Execution reset
  logReset <=
    '1' when reset              = '1'   or
         auxEnterConfigMode = '1' else
    '0';

  -- Config mode enable
  cfgEnable <=
    '1' when auxLogMode = configMode else
    '0';


  --
  -- Port mapping
  --

  -- logFrameInterpreter

  logFrameReaderUnit:
    nccFrameReader
      port map(
        reset => reset,
        clkR  => clkR,

        coupledSignal => coupledSignal,

        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        sampleFrame => lfrSampleFrame,

        id   => auxId,
        data => auxData
      );
  -- logFrameFilter

  logFrameFilterUnit:
    nccFrameFilter
      port map(
        nfrSampleFrame => lfrSampleFrame,

        id       => auxId,
        local_id => HUB_LOG_ID,

        nccMode  => auxLogMode,

        sampleFrame => lffSampleFrame
      );

  -- logCoordinator

  logCoordinatorUnit:
    nccCoordinator
      port map(
        reset => reset,
        clk   => clk,

        sampleFrame => lffSampleFrame,
        data => auxData(1 to 8),

        enterConfigMode => auxEnterConfigMode,
        nccMode => auxLogMode
      );

  -- logCmdInterpreter

  logCmdInterpreterUnit:
    logCmdInterpreter
      port map(
        enable  => cfgEnable,

        reset => logReset,
        clk   => clk,

        sampleFrame => lffSampleFrame,
        data => auxData(1 to 8),

        sampleReport => sampleReport
      );

end Behavioral;
```

D.5.3. nccFrameReader.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 17/03/2011
-- Description:
--    Captures frame bits and extracts its id and data.
--    Uses a sample signal to inform that values can be readed.
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defInteger.all;
use work.defStates.all;
use work.defInjection.all;

entity nccFrameReader is
  port(
    reset : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;

    frameState   : in estadoGlobalFrame;
    frameBitNum  : in ENTER64;
    isStuffBit   : in std_logic;

    sampleFrame : out std_logic;

    id   : out t_id;
    data : out t_data
  );
end nccFrameReader;

architecture Behavioral of nccFrameReader is
begin

  process (reset, clkR) is
    variable auxId   : std_logic_vector(0 to 10);
    variable auxData : std_logic_vector(0 to 63);

    variable newFrame : boolean;
    variable error    : boolean;

  begin
    if reset = '1' then
      auxId   := id0;
      auxData := data0; -- 64 0 bits

      sampleFrame <= '0';
      error := false;

    elsif clkR'event and clkR = '1' and isStuffBit = '0' then
      sampleFrame <= '0';

      case frameState is
        -- initialize aux values
        when idle =>
          auxId   := id0;
          auxData := data0; -- 64 0 bits

        -- capture identifier
        when idField =>
          auxId(frameBitNum) := coupledSignal;

        -- capture data
        when dataField =>
          auxData(frameBitNum) := coupledSignal;

        when errorFlag =>
          error := true;

        -- Outputs values at the end of the frame
        -- unless an error has occurred
        when interField =>
          if frameBitNum = 0 then
            if error = true then
              error := false;

            else
              id   <= auxId;
              data <= auxData;

              sampleFrame <= '1';
            end if;
          end if;

        when others => null;
      end case;
    end if;
  end process;

end Behavioral;
```

D.5.4. nccFrameFilter.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 03/05/2011
-- Description:
--    Generates a new sampleFrame which filters by id and fimMode
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defNCC.all;

entity nccFrameFilter is
  port(
    nfrSampleFrame : in std_logic;

    id       : in t_id;
    local_id : in t_id;

    nccMode : in t_nccMode;

    sampleFrame : out std_logic
  );
end nccFrameFilter;

architecture Behavioral of nccFrameFilter is

begin

  sampleFrame <=
    '1' when nfrSampleFrame = '1' and
        ((id = BROAD_ID) or
         (id = local_id and nccMode /= execMode))
      else
    '0';

end Behavioral;
```

D.5.5. nccCoordinator.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 17/03/2011
-- Description:
--    Reads "data" from nccFrameReader using a filtered sample signal from
--    nccFrameFilter. Then interprets the switch mode command within (if exists).
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defNCC.all;

entity nccCoordinator is
  port(
    reset : in std_logic;
    clk   : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 8);

    enterConfigMode : out std_logic;
    nccMode         : out t_nccMode
  );
end nccCoordinator;

architecture Behavioral of nccCoordinator is

  signal auxNCCMode : t_nccMode;

begin

  --
  -- Output Signals
  --

  nccMode <= auxNCCMode;


  process (reset, clk) is
    variable lastSampleFrame : std_logic;

  begin
    if reset = '1' then
      lastSampleFrame := '0';
      auxNCCMode <= configMode;

    elsif clk'event and clk = '1' then
      -- new frame available
      enterConfigMode <= '0';

      if lastSampleFrame = '0' and sampleFrame = '1' and
         data(1 to 3)    = CMD_MCH                 then

        case data(4 to 8) is
          when MC_ECM =>
            auxNCCMode <= configMode;
            enterConfigMode <= '1';

          when MC_EIM =>
            if auxNCCMode = configMode then
              auxNCCMode <= idleMode;
            end if;

          when MC_EWM =>
            if auxNCCMode = configMode then
              auxNCCMode <= wfwMode;
            end if;

          when MC_EEM =>
            if auxNCCMode = wfwMode then
              auxNCCMode <= execMode;
            end if;

          when others => null;
        end case;

      end if;

      lastSampleFrame := sampleFrame;
    end if;

  end process;

end Behavioral;
```

D.5.6. logCmdInterpreter.vhd

```
--------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 15/06/2011
-- Description:
--    Reads "data" from logFrameReader using a filtered sample signal from
--    logFrameFilter. Then interprets the log command within (if exists).
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.defInjection.all;
use work.defNCC.all;
use work.defLogger.all;
use work.defStates.all;
use work.defInteger.all;

entity logCmdInterpreter is
  port(
    enable  : in std_logic;

    reset : in std_logic;
    clk   : in std_logic;

    sampleFrame : in std_logic;
    data : in std_logic_vector(1 to 8);

    sampleReport : out std_logic
  );
end logCmdInterpreter;


architecture Behavioral of logCmdInterpreter is
  signal cmdReady : std_logic;
begin

  process (reset, clk) is
    variable lastSampleFrame : std_logic;

  begin
    if enable = '0' then
      lastSampleFrame := '0';
      sampleReport <= '0';

    elsif clk'event and clk = '1' and
        enable = '1'           then

      -- initialize log cmd
      if sampleFrame = '0' then
        sampleReport <= '0';
      end if;

      -- new frame available
      if lastSampleFrame = '0' and sampleFrame = '1' and
        data(1 to 3)    = CMD_LOG                  then
        sampleReport <= '1';
      end if;

      lastSampleFrame := sampleFrame;
    end if;

  end process;

end Behavioral;
```

D.5.7. logGatherer.vhd

```
--------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 15/06/2011
-- Description:
--    Stores logging data during the execution of a test. Specifically, port data
--    and frame stream data.
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;
use work.defLogger.all;

entity logGatherer is
  port(
    enable : in std_logic;

    reset : in std_logic;
    clkR  : in std_logic;

    --
    -- input log signals
    --

    nodesRole : in t_portSignalArray;

    -- logFrameGatherer
    coupledSignal : in std_logic;
    frameState    : in estadoGlobalFrame;
    frameBitNum   : in ENTER64;
    isStuffBit    : in std_logic;

    --
    -- output log signals
    --

    -- logFrameGatherer
    logStoredFrames     : out t_logStoredFrames;
    logStoredFrames_cnt : out t_logStoredFramesCnt
  );
end logGatherer;

architecture Behavioral of logGatherer is

  --
  -- Components
  --

  -- Frame gatherer

  component logFrameGatherer is
  port(
    enable  : in std_logic;
    reset : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;

    -- State of frame
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    txPort : in t_port;

    logStoredFrames     : out t_logStoredFrames;
    logStoredFrames_cnt : out t_logStoredFramesCnt
  );
  end component;


  --
  -- Signals
  --

  -- logFrameGatherer


  -- logTxPort
  signal auxTxPort : t_port;


begin

  auxTxPort <=
    port0 when nodesRole(port0) = '0' else
    port1 when nodesRole(port1) = '0' else
    port2 when nodesRole(port2) = '0' else
    port2;


  -- logFrameGatherer

  logFrameGathererUnit :
    logFrameGatherer
      port map(
        enable  => enable,

        reset => reset,
        clkR  => clkR,

        coupledSignal => coupledSignal,

        -- State of frame
        frameState  => frameState,
        frameBitNum => frameBitNum,
        isStuffBit  => isStuffBit,

        txPort => auxTxPort,

        logStoredFrames     => logStoredFrames,
        logStoredFrames_cnt => logStoredFrames_cnt
      );

end Behavioral;
```

D.5.8. logFrameGatherer.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 22/06/2011
-- Description:
--    Stores the sequence of frames in the coupled signal. Specifying the
--    transmitter node and the type frame (id, id+data, valid id+data and error)
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;
use work.defLogger.all;

entity logFrameGatherer is
  port(
    enable : in std_logic;

    reset : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;

    -- State of frame
    frameState  : in estadoGlobalFrame;
    frameBitNum : in ENTER64;
    isStuffBit  : in std_logic;

    txPort : in t_port;

    logStoredFrames     : out t_logStoredFrames;
    logStoredFrames_cnt : out t_logStoredFramesCnt
  );
end logFrameGatherer;


architecture Behavioral of logFrameGatherer is
begin

  process (reset, clkR) is
    variable auxP    : t_port;
    variable auxId   : std_logic_vector(0 to 10);
    variable auxDlc  : std_logic_vector(0 to  3);
    variable auxData : std_logic_vector(0 to 63);

    variable lastValidFrameState  : estadoGlobalFrame;
    variable lastValidFrameBitNum : ENTER64;

    variable idx : t_logStoredFramesCnt;

    variable activeFrame : boolean;
    variable error       : boolean;

  begin
    if reset = '1' then
      auxP    := port0;
      auxId   := id0;
      auxDlc  := "0000";
      auxData := data0;

      idx := 0;
      activeFrame := false;
      error       := false;

      logStoredFrames     <= logStoredFrames0;
      logStoredFrames_cnt <= 0;

      lastValidFrameState  := idle;
      lastValidFrameBitNum := 0;

    elsif clkR'event and clkR = '1' and
        isStuffBit = '0'                and
        enable     = '1'               then

      case frameState is
        when idle =>
          -- SOF
          if coupledSignal = '0' then
            activeFrame := true;
          end if;

        when idField =>
          auxId(frameBitNum) := coupledSignal;

        when rtrField =>
          -- check transmitter
          auxP := txPort;

        when dlcField =>
          auxDlc(frameBitnum) := coupledSignal;

        when dataField =>
          auxData(frameBitnum) := coupledSignal;

        when errorFlag =>
          error := true;

        when interField =>
          if frameBitNum = 0 then
            if activeFrame then
              -- update stored frame
              if idx < MAX_LOG_FRAMES then
                idx := idx + 1;

                logStoredFrames(idx).valid <= '1';

                logStoredFrames(idx).p      <= auxP;

                logStoredFrames(idx).bitNum <= lastValidFrameBitNum;
                logStoredFrames(idx).field  <= lastValidFrameState;

                logStoredFrames(idx).id     <= auxId;
                logStoredFrames(idx).dlc    <= to_integer(unsigned(auxDlc));
                logStoredFrames(idx).data   <= auxData(0 to 7);
              end if;
```

```
              activeFrame := false;
            end if;

            if error then
              -- new error frame
              if idx < MAX_LOG_FRAMES then
                idx := idx + 1;
                logStoredFrames(idx).valid <= '0';
              end if;

              error := false;
            end if;
          end if;

        when others =>
          null;

      end case;

      logStoredFrames_cnt <= idx;

      if frameState /= errorFlag and frameState /= errorDelimiter then
        lastValidFrameState  := frameState;
        lastValidFrameBitNum := frameBitNum;
      end if;

    end if;
  end process;

end Behavioral;
```

D.5.9. logReporter.vhd

```
--------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 15/06/2011
-- Description:
--
--
--
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

use work.defGeneral.all;
use work.defInteger.all;
use work.defStates.all;
use work.defNCC.all;
use work.defLogger.all;

entity logReporter is
  port(
    enable : in std_logic;

    reset : in std_logic;
    clk   : in std_logic;
    clkT  : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;
    frameState    : in estadoGlobalFrame;
    frameBitNum   : in ENTER64;
    isStuffBit    : in std_logic;

    sampleReport : in std_logic;

    -- log signals
    logStoredFrames     : in t_logStoredFrames;
    logStoredFrames_cnt : in t_logStoredFramesCnt;

    contri : out std_logic
  );
end logReporter;

architecture Behavioral of logReporter is

  --
  -- Components
  --

  -- log ReportConstructor

  component logReportConstructor is
  port(
    enable : in std_logic;

    reset : in std_logic;
    clk   : in std_logic;
    clkT  : in std_logic;
    sampleReport : in std_logic;

    -- log signals
    logStoredFrames     : in t_logStoredFrames;
    logStoredFrames_cnt : in t_logStoredFramesCnt;

    -- sender signals
    senderBusy : in std_logic;

    -- output frame
    dlc        : out ENTER9;
    data       : out t_data;
    sampleData : out std_logic
  );
  end component;

  -- CANSender

  component CANSender is
  port(
    reset : in std_logic;
    clk   : in std_logic;
    clkT  : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;
    frameState    : in estadoGlobalFrame;
    frameBitNum   : in ENTER64;
    isStuffBit    : in std_logic;

    id   : in t_id;
    dlc  : in ENTER16;
    data : in t_data;

    sampleData : in std_logic;

    busy   : out std_logic;
    contri : out std_logic
  );
  end component;


  --
  -- Signals
  --

  -- reportConstructor
  signal auxDlc        : ENTER9;
  signal auxData       : t_data;
  signal auxSampleData : std_logic;

  -- logCANSender
  signal senderBusy : std_logic;

begin

  logReportConstructorUnit :
    logReportConstructor
      port map(
        enable => enable,
        reset => reset,
        clk   => clk,
        clkT  => clkT,

        sampleReport => sampleReport,

        -- log signals
        logStoredFrames     => logStoredFrames,
        logStoredFrames_cnt => logStoredFrames_cnt,

        -- sender signals
        senderBusy => senderBusy,

        -- output frame
        dlc        => auxDlc,
        data       => auxData,
        sampleData => auxSampleData
      );

  logCANSenderUnit :
    CANSender
      port map(
        reset => reset,
        clk   => clk,
        clkT  => clkT,
        clkR  => clkR,

        coupledSignal => coupledSignal,
        frameState    => frameState,
        frameBitNum   => frameBitNum,
        isStuffBit    => isStuffBit,

        id   => PC_ID,
        dlc  => auxDlc,
        data => auxData,

        sampleData => auxSampleData,

        busy   => senderBusy,
        contri => contri
      );

end Behavioral;
```

D.5.10. logReportConstructor.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 15/06/2011
-- Description:
--
--
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

use work.defGeneral.all;
use work.defStates.all;
use work.defInteger.all;
use work.defNCC.all;
use work.defLogger.all;

entity logReportConstructor is
  port(
    enable : in std_logic;

    reset : in std_logic;
    clk   : in std_logic;
    clkT  : in std_logic;

    sampleReport : in std_logic;

    -- log signals
    logStoredFrames     : in t_logStoredFrames;
    logStoredFrames_cnt : in t_logStoredFramesCnt;

    -- sender signals
    senderBusy : in std_logic;

    -- output frame
    dlc        : out ENTER9;
    data       : out t_data;
    sampleData : out std_logic
  );
end logReportConstructor;


architecture Behavioral of logReportConstructor is

  signal active     : std_logic;
  signal reportDone : std_logic;


begin

  -- receive logCmdInterpreter command

  process (reset, clk) is
    variable lastSampleReport : std_logic;
    variable lastReportDone   : std_logic;
```

```
begin
  if reset = '1' then
    active <= '0';
    lastSampleReport := '0';

  elsif clk'event and clk = '1' and
      enable = '1'           then

    if lastSampleReport = '0' and sampleReport = '1' and
      active = '0'                                 then

      active <= '1';

    elsif lastReportDone = '0' and reportDone = '1' and
        active = '1'                             then

      active <= '0';
    end if;

    lastSampleReport := sampleReport;
    lastReportDone   := reportDone;
  end if;
end process;


-- Create and signal frame to CANSender

process (reset, clkT) is
  variable logStoredFramesIdx : t_logStoredFramesCnt;
  variable lrcState : t_logValue;

begin
  if reset = '1' or active = '0' then
    sampleData <= '0';
    data <= data0;
    dlc  <= 0;

    lrcState := logFrames;
    reportDone <= '0';
    logStoredFramesIdx := 1;

  elsif clkT'event and clkT = '1' then

    if senderBusy = '1' then
      --initialize CANSender variables
      sampleData <= '0';
      dlc  <= 0;
      data <= data0;

    else
      sampleData <= '1';

      -- States machine that sends the diferent log
      -- values to the CANSender
      case lrcState is

        when logFrames =>
          if logStoredFrames_cnt > 0                    and
            logStoredFramesIdx <= logStoredFrames_cnt then

            data(01 to 03) <= CMD_LOG;
            data(04 to 08) <= LOG_FRAMES;
            data(9)        <= logStoredFrames(logStoredFramesIdx).valid;

            data(10 to 12) <= portSLV(logStoredFrames(logStoredFramesIdx).p);

            data(13 to 16) <= frameStateSLV(logStoredFrames(logStoredFramesIdx).field);

            data(19 to 24) <= conv_std_logic_vector(logStoredFrames(logStoredFramesIdx).bitNum,6);

            data(25 to 28) <= conv_std_logic_vector(logStoredFrames(logStoredFramesIdx).dlc,4);

            data(30 to 40) <= logStoredFrames(logStoredFramesIdx).id;
            data(41 to 48) <= logStoredFrames(logStoredFramesIdx).data;

            dlc <= 6;

            logStoredFramesIdx := logStoredFramesIdx + 1;

          else
            lrcState := logEOL;
          end if;

        when logEOL =>
          data(01 to 03) <= CMD_LOG;
          data(04 to 08) <= LOG_EOL;

          dlc <= 1;

          lrcState := none;

        when none =>
          reportDone <= '1';
          sampleData <= '0';
          lrcState := logFrames;

        when others =>
          null;

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

D.5.11. CANSender.vhd

```
----------------------------------------------------------------------------------
--
-- Engineer: Alberto Ballesteros
--
-- Create Date: 23/06/2011
-- Description:
--
--
--
----------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_arith.all;

use work.defGeneral.all;
use work.defInteger.all;
use work.defStates.all;
use work.defLogger.all;

entity CANSender is
  port(
    reset : in std_logic;
    clk   : in std_logic;
    clkT  : in std_logic;
    clkR  : in std_logic;

    coupledSignal : in std_logic;
    frameState    : in estadoGlobalFrame;
    frameBitNum   : in ENTER64;
    isStuffBit    : in std_logic;

    id   : in t_id;
    dlc  : in ENTER16;
    data : in t_data;

    sampleData : in std_logic;

    busy   : out std_logic;
    contri : out std_logic
  );
end CANSender;

architecture Behavioral of CANSender is

  signal auxId   : std_logic_vector(0 to 10);
  signal auxDlc  : std_logic_vector(0 to 3);
  signal auxData : std_logic_vector(0 to 63);

  signal auxContri : std_logic;
  signal crc       : t_crc;
  signal isTx      : boolean;

  signal active  : std_logic;
  signal auxBusy : std_logic;


begin

  contri <= auxContri;
```

```
busy   <= auxBusy;


-- Activate CANSender

process (reset, clk) is
  variable lastSampleData : std_logic;
  variable lastActive     : std_logic;

begin
  if reset = '1' then
    auxBusy <= '0';

    auxId   <= id0;
    auxDlc  <= "0000";
    auxData <= data0;

    lastSampleData := '0';
    lastActive     := '0';

  elsif clk'event and clk = '1' then

    if lastSampleData = '0' and sampleData = '1' and
      auxBusy = '0'                             then

      auxBusy <= '1';

      auxId   <= id;
      auxDlc  <= conv_std_logic_vector(dlc, 4);
      auxData <= data;

    elsif lastActive = '1' and active = '0' and
        auxBusy = '1'                     then

      auxBusy <= '0';
    end if;

    lastSampleData := sampleData;
    lastActive     := active;
  end if;
end process;


-- Send frame

process (reset, clkT) is
  variable error   : boolean;
  variable sendBit : std_logic;

begin
  if reset = '1' then
    active <= '0';

    error   := false;
    sendBit := '1';

  elsif clkT'event and clkT = '1' then

    if auxBusy = '1' then
      active <= '1';

      if isStuffBit = '1' then
```

```
        sendBit := not sendBit;

      elsif not isTx then
        sendBit := '1';

      else
        case frameState is
          when idle =>
            sendBit := '0';
            error   := false;

          when idField =>
            sendBit := auxId(frameBitNum);

          when rtrField =>
            sendBit := '0';

          when resField =>
            if    frameBitNum = 0 then sendBit := '0'; -- IDE
            elsif frameBitNum = 1 then sendBit := '0'; -- R0
            end if;

          when dlcField =>
            sendBit := auxDlc(frameBitNum);

          when dataField =>
            sendBit := auxData(frameBitNum);

          when crcField =>
            sendBit := crc(frameBitnum+1);

          when crcDelimField =>
            sendBit := '1';

          when ackSlotField =>
            sendBit := '1';

          when ackDelimField =>
            sendBit := '1';

          when eofField =>
            sendBit := '1';

          when interField =>
            sendBit := '1';

            if not error then
              active <= '0';
            end if;

          when errorFlag =>
            sendBit := '1';
            error   := true;

          when errorDelimiter =>
            sendBit := '1';

          when others =>
            sendBit := '1';
        end case;
      end if;
    end if;
    end if;

    auxContri <= sendBit;
  end process;


  -- Calculate CRC
  -- Do arbitration

  process (reset, clkR) is
    variable auxCrc : std_logic_vector(1 to 15);

  begin
    if reset = '1' then
      auxCrc := "000000000000000";
      isTx <= true;

    elsif clkR'event and clkR = '1' and
        isStuffBit = '0'               then

      case frameState is

        when idle =>
          isTx <= true;

          auxCrc := "000000000000000";

          -- SOF
          if coupledSignal = '0' then
            auxCrc := crcCalc(auxCrc, '0');
          end if;

        when idField =>
          if auxContri = '1' and coupledSignal = '0' then
            isTx <= false;
          end if;

          auxCrc := crcCalc(auxCrc, coupledSignal);

        when rtrField | resField | dlcField | dataField =>
          auxCrc := crcCalc(auxCrc, coupledSignal);

        when others =>
          null;

      end case;
    end if;

    crc <= auxCrc;
  end process;

end Behavioral;
```

E. Nodes software source code

E.1. Packages

E.1.1. can.h

```
#ifndef _CAN_H_
#define _CAN_H_

#include "common.h"
#include "can_frame.h"

void init_can();

void tx_frame(struct can_frame frame);
bool tx_correct();

bool received_frame();
void rx_frame(struct can_frame* frame);

#endif /* _CAN_H_ */
```

E.1.2. can.c

```
#include "can.h"

#include "can_controller.h"

extern struct can_controller ctrl1;

void init_can(){
  init_can_controllers();
}

void tx_frame(struct can_frame frame){
  unsigned int count;

  request_tx(
    &ctrl1,
    &frame
  );

  //wait until transmission is carried out
  while(!tx_buffer0_irq_occurred(&ctrl1));

  /**********************************************************************/
  /** Code enabling the single-shot transmission mode *********************/
  /**********************************************************************/
  // It waits for the frame to be completely transmitted. The default value
  // (450) can be used for standard frames containing 1 byte in their data
  // fields. Otherwise this value must be modified. Note that, this code
  // carries out the task of the above while statement and, thus, when the
  // single-shot mode is enabled it must be removed.

  /*
  count = 0;
  while(count < 450){
    count++;
  }*/

  /**********************************************************************/
  /**********************************************************************/

  //clear transmission buffer
  clear_tx_buffer0_irq(&ctrl1);
}

bool tx_correct(){
  if((&ctrl1)->tx_buffer[0]->CONbits->TXABT == 1)
    return false;
  else
    return true;
}

bool received_frame(){
  return rx_buffer0_irq_occured(&ctrl1);
}

void rx_frame(struct can_frame* frame){
  volatile struct rx_buffer_struct* rx_buffer;

  //wait for a received frame in buffer 0
  while(!rx_buffer0_irq_occured(&ctrl1));
  read_frame(
    (&ctrl1)->rx_buffer[0],
    frame
  );

  release_rx_buffer(
    &ctrl1,
    (&ctrl1)->rx_buffer[0]
  );

  clear_rx_buffer0_irq(
    &ctrl1
  );
}
```

E.1.3. sfiCAN.h

```
/*
 * sfiCAN.h
 *
 * Written by Alberto Ballesteros <ballesteros.alberto@gmail.com>
 */

#ifndef _SFICAN_H_
#define _SFICAN_H_

//////////////
// NCC IDs  //
//////////////

#define BROAD_ID 0x000
#define PC_ID    0x010

#define HUB_FIM_ID 0x001
#define HUB_LOG_ID 0x002

//defined as a compiler option
#define NODE0_LOG_ID 0x003
#define NODE1_LOG_ID 0x004
#define NODE2_LOG_ID 0x005


/////////////////////
//  Command codes  //
/////////////////////

#define CFG_CMD_CODE 0x00
#define MCH_CMD_CODE 0x01
#define LOG_CMD_CODE 0x02


///////////////////
//  Mode change  //
///////////////////

#define MC_ECM 0x00
#define MC_EIM 0x01
#define MC_EWM 0x02
#define MC_EEM 0x03


///////////////
//  Logging  //
///////////////

#define NODE_ERROR_COUNTERS_CODE  0x00
#define NODE_STORED_FRAME_CODE    0x01

#define LOG_EOL_CODE              0x1F


// Type definitions
#define uchar  unsigned char  // 1 byte
#define ushort unsigned short // 2 byte
#define uint   unsigned int   // 4 bytes

typedef enum {mode_cfg, mode_idle, mode_wfw, mode_exec} t_mode;
```

```
// Extracts from 'byte' a bit vector starting in first and ending in
// first+size-1
uchar extractBits(uchar byte, uchar first, uchar size);

#endif /* _SFICAN_H_ */
```

E.1.4. sfiCAN.c

```
#include "sfiCAN.h"

// Extracts from 'byte' a bit vector starting in first and ending in
// first+size-1
uchar extractBits(uchar byte, uchar first, uchar size){
  uchar temp_byte;

  temp_byte = byte << (first-1);
  temp_byte = temp_byte >> (8-size);

  return temp_byte;
}
```

E.2. Application

E.2.1. sfiCAN_tx.c

```
/*
 * sfiCAN_tx.c
 *
 * Written by Alberto Ballesteros <ballesteros.alberto@gmail.com>
 *
 */

#include "device_config.h"

#include "can.h"
#include "can_frame.h"
#include "led.h"

#include "sfiCAN.h"
#include "sfiCAN_log.h"

#define MAX_COUNT 50

struct can_frame frame;

uint count;
bool enabledExec = true;
uchar lastTEC;
t_log_event event;

int main(void){

  init_leds();
  init_can();

  logger(frame, none);

  count = 0;

  //prepare frame
  frame.identifier = 0x010;
  frame.data[0]    = count;
  frame.length  = 1;

  lastTEC = C1TERRCNT;

  while(1){
    if(count < MAX_COUNT || MAX_COUNT == 0){
      tx_frame(frame);
      led_display(count % 8);

      if(C1TERRCNT != lastTEC){
        event.log_can_frame.frame.length = 0;
        event.tec = C1TERRCNT;

        storeLogEvent(event);

        lastTEC = C1TERRCNT;
      }

      logger(frame, tx);
      count++;
      frame.data[0] = count % 8;

    }else{
      logger(frame, none);
    }
  }

  return 0;
}
```

E.2.2. sfiCAN_rx.c

```
/*
 * sfiCAN_rx.c
 *
 * Written by Alberto Ballesteros <ballesteros.alberto@gmail.com>
 *
 */

#include <p30f6014A.h>
#include "device_config.h"

#include "can.h"
#include "led.h"
#include "can_frame.h"

#include "sfiCAN.h"
#include "sfiCAN_log.h"

struct can_frame frame;
unsigned int lastREC;
t_log_event event;

int main(void){

  init_leds();
  init_can();

  logger(frame, none);

  while(1){
    lastREC = C1RERRCNT;

    // wait until a frame is recived
    while(!received_frame()){
      if(C1RERRCNT != lastREC){
        event.log_can_frame.frame.length = 0;
        event.rec = C1RERRCNT;

        storeLogEvent(event);

        lastREC = C1RERRCNT;
      }
    }

    rx_frame(&frame);

    // Show value
    led_display(frame.data[0] % 8);

    logger(frame, rx);
  }

  return 0;
}
```

E.3. Node Logger

E.3.1. sfiCAN_log.h

```
#ifndef _SFICAN_LOG_H_
#define _SFICAN_LOG_H_

#include "sfiCAN.h"

#include "can_frame.h"

#define MAX_EVENTS 100

typedef enum {tx=0, rx=1, none} t_role;

typedef struct{
  t_role role;
  bool correctTx;
  struct can_frame frame;
} t_log_can_frame;

typedef struct{
  t_log_can_frame log_can_frame;
  uchar tec, rec;
} t_log_event;

void logger(struct can_frame frame, t_role role);

void storeLogEvent(t_log_event event);

#endif /* _SFICAN_LOG_H_ */
```

E.3.2. sfiCAN_log.c

```
#include "sfiCAN_log.h"

#include <p30f6014A.h>
#include "sfiCAN.h"
#include "can.h"
#include "led.h"

t_log_event events[MAX_EVENTS];
uint        events_cnt = 0;

t_mode mode = mode_cfg;

// Functions
void initLog();

void logger(struct can_frame frame, t_role role){

  uchar cmd;
  uchar param;

  t_log_event event;

  uint i;

  if(mode == mode_exec){

    if(received_frame()){
      rx_frame(&frame);
    }

    //process frame
    if(frame.identifier == BROAD_ID){
      cmd   = extractBits(frame.data[0], 1,3);
      param = extractBits(frame.data[0], 4,5);

      if(cmd == MCH_CMD_CODE){
        switch(param){
          case MC_ECM : mode = mode_cfg; break;
          case MC_EIM : break;
          case MC_EWM : break;
          case MC_EEM : break;
        }
      }

    // store frame and error counters
    }else if(role != none){
      event.log_can_frame.role  = role;
      event.log_can_frame.frame = frame;
      event.log_can_frame.correctTx = tx_correct();

      event.tec = 255;
      event.rec = 255;

      storeLogEvent(event);
    }
  }

  while(mode != mode_exec){
    led_display(8 + mode);
```

```
do{
  rx_frame(&frame);
}while(!(frame.identifier == BROAD_ID || frame.identifier == NODE_LOG_ID));

cmd   = extractBits(frame.data[0], 1,3);
param = extractBits(frame.data[0], 4,5);

switch(mode){

  // configuration mode
  case mode_cfg :
    if(cmd == MCH_CMD_CODE){
      switch(param){
        case MC_ECM : break;

        case MC_EIM :
          initLog();
          mode = mode_idle;
        break;

        case MC_EWM :
          initLog();
          mode = mode_wfw;
        break;

        case MC_EEM : break;
      }

    }else if(cmd == LOG_CMD_CODE){

      frame.identifier = PC_ID;

      for(i=0; i<events_cnt; i++){
        //
        // Error counters
        //
        frame.length = 3;
        frame.data[0] = (LOG_CMD_CODE << 5) | NODE_ERROR_COUNTERS_CODE;
        frame.data[1] = events[i].tec;
        frame.data[2] = events[i].rec;

        tx_frame(frame);

        //
        // Stored frames
        //
        frame.length  = 4;
        frame.data[0] = (LOG_CMD_CODE << 5) | NODE_STORED_FRAME_CODE;

        // (role + data) and first id nibble
        frame.data[1] =
          (events[i].log_can_frame.role         << 7) |
          (events[i].log_can_frame.frame.length << 4) |
          (events[i].log_can_frame.correctTx    << 3) |
          (extractBits(events[i].log_can_frame.frame.identifier >> 8, 6, 3));

        // second and third id nibble
        frame.data[2] =
          events[i].log_can_frame.frame.identifier;

        frame.data[3] = events[i].log_can_frame.frame.data[0];
```

```
            tx_frame(frame);
          }

          //
          // End of log
          //
          frame.data[0] = (LOG_CMD_CODE << 5) | LOG_EOL_CODE;
          tx_frame(frame);
        }
      break;


      // idle mode
      case mode_idle :
        if(cmd == MCH_CMD_CODE){
          switch(param){
            case MC_ECM : mode = mode_cfg;  break;
            case MC_EIM : break;
            case MC_EWM : break;
            case MC_EEM : break;
          }
        }
      break;


      // wait for whistle mode
      case mode_wfw :
        if(cmd == MCH_CMD_CODE){
          switch(param){
            case MC_ECM : mode = mode_cfg;  break;
            case MC_EIM : break;
            case MC_EWM : break;
            case MC_EEM : mode = mode_exec; break;
          }
        }
      break;
    }

    led_display(8 + mode);
  }
}


void storeLogEvent(t_log_event event){
  if(events_cnt < MAX_EVENTS){
    events[events_cnt] = event;
    events_cnt++;
  }
}

void initLog(){
  events_cnt = 0;
}
```

E.4. Modified ReCANcentrate driver

E.4.1. common.h

```
/*
 * common.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _COMMON_H_
#define _COMMON_H_

typedef enum bool_enum {
  false = 0,
  true
} bool;

#define NULL 0

#endif /* _COMMON_H_ */
```

E.4.2. assert.h

```
/*
 * assert.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _ASSERT_H_
#define _ASSERT_H_


#ifdef NDEBUG

#define ASSERT(expr) ((void)0)

#else

void aFailed(
  char *file_name,
  int line
);

#define ASSERT(expr) if (expr) {/*Do nothing*/} else\
  aFailed(__FILE__, __LINE__)

void aFailed2(
  char *file_name,
  int line,
  char led_value
);

#define ASSERT2(expr, led_value) if (expr) {/*Do nothing*/} else\
  aFailed2(__FILE__, __LINE__, led_value)

#endif /* NDEBUG */


#endif /* _ASSERT_H_ */
```

E.4.3. assert.c

```
/*
 * assert.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "led.h"
#include <p30f6014A.h>
//#include "interrupts.h"

#ifndef NDEBUG



#define END_OF_STRING 0
#define FILE_NAME_LENGTH_MAX 100

char asserted_file_name[FILE_NAME_LENGTH_MAX + 1] = { END_OF_STRING };
int asserted_line_number = 0;





static void copy_string(
  char *source,
  char *destination
)
{
  int i = 0;

  while (source[i] != END_OF_STRING && i < FILE_NAME_LENGTH_MAX) {
    destination[i] = source[i];
    i++;
  }
  destination[i] = END_OF_STRING;
}



static void wait(void)
{
  int i, j;

  for (i = 0; i < 10000; i++) {
    for (j = 0; j < 100; j++);
  }
}



void aFailed2(
  char *file_name,
  int line,
  char led_value
)
{
  /* Disable interrupts */
  //SET_CPU_IPL(INTERRUPT_PRIORITY_MAX);
  /* Copy the file name and line number where the assert failed to a
   * fixed memory location so that these memory locations can be looked
   * up in a debugger to determine where the assert failed. */
  copy_string(file_name, asserted_file_name);
  asserted_line_number = line;
  init_leds();
  while (1) {
    /* Flash LEDs to show that an assert failed */
    led_display(0x0);
    wait();
    led_display(led_value);
    wait();
  }
}



void aFailed(
  char *file_name,
  int line
)
{
  aFailed2(file_name, line, 0xFF);
}



#endif // NDEBUG
```

E.4.4. device_config.h

```
/*
 * device_config.h
 *
 * Written by David Gessner <davidges@gmail.com>
 *
 * Edited by Alberto Ballesteros <ballesteros.alberto@gmail.com>
 *
 */

#ifndef ___DEVICE_CONFIG_H_
#define ___DEVICE_CONFIG_H_

#include <p30f6014A.h>

/******** DEVICE CONFIGURATION ********/

/* Oscillator configuration
 *
 * Oscillators provided by the dsPICDEM board:
 *     - Y1: 7.37 MHz
 *     - Y2: 32.768 KHz
 *     - Y3: external oscillator (16MHz installed)
 * dsPIC30F6014A internal oscillators:
 *     - FRC: 7.37 MHz
 *     - LPRC: 512 KHz
 */

_FOSC(
  /* Clock switching and fail safe clock monitor off, i.e.  do not detect
   * clock failures and do not switch over to internal FRC oscillator. */
  CSW_FSCM_OFF &

  /* Use a high Speed external oscillator. In this case 16MHz.
   * But FOSC is not the frequency used for the instruction cycle
   * The instruction cycle's frequency is FCY = FOSC/4 = 4MHz. */
  ECIO_PLL4);

/* Watchdog timer configuration = watchdog timer off. */
_FWDT(WDT_OFF);

/* Reset configuration */
_FBORPOR(
  /* Enable brown out at 20 volts. */
  PBOR_ON & BORV_20 &
  /* Power up timer = 64ms, gives the oscillator time to start and
   * stabilize. */
  PWRT_64 &
  /* Master clear reset enabled, i.e. use the MCLR pin as a reset signal
   * instead of using it as an IO pin.  Pulling the MCLR pin low will
   * reset the dsPIC and start execution from 0x000. */
  MCLR_EN);

/* General Code Segment configuration = Disable Code Protection */
_FGS(CODE_PROT_OFF);

#endif /* ___DEVICE_CONFIG_H_ */
```

E.4.5. can_frame.h

```
/*
 * can_frame.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _CAN_FRAME_H_
#define _CAN_FRAME_H_



#include "common.h"



/* According to the CAN specification a CAN data frame can carry at most 8
 * bytes */
#define CAN_PAYLOAD_LEN_MAX 8






struct can_frame {
  unsigned char data[CAN_PAYLOAD_LEN_MAX];
  unsigned char length;
  unsigned int identifier;
};

bool equals_frame(
  const volatile struct can_frame *const frame1,
  const volatile struct can_frame *const frame2
);

void copy_frame(
  const volatile struct can_frame *const src,
  volatile struct can_frame *const dst
);



void copy_data(
  volatile unsigned const char *const input_buffer,
  int num_bytes_to_copy,
  volatile unsigned char *const output_buffer
);



#endif /* _CAN_FRAME_H_ */
```

E.4.6. can_frame.c

```
/*
 * can_frame.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include "can_frame.h"
#include "assert.h"


bool equals_frame(
  const volatile struct can_frame *const frame1,
  const volatile struct can_frame *const frame2
)
{
  int i;

  if (frame1->identifier != frame2->identifier) {
    return false;
  }

  if (frame1->length != frame2->length) {
    return false;
  }

  for (i = 0; i < frame1->length; i++) {
    if (frame1->data[i] != frame2->data[i]) {
      return false;
    }
  }
  return true;
}


void copy_frame(
  const volatile struct can_frame *const src,
  volatile struct can_frame *const dst
)
{
  *dst = *src;
}


void copy_data(
  volatile unsigned const char *const input_buffer,
  int num_bytes_to_copy,
  volatile unsigned char *const output_buffer
)
{
  int byte_idx = 0;

  ASSERT(0 <= num_bytes_to_copy &&
    num_bytes_to_copy <= CAN_PAYLOAD_LEN_MAX);

  for (byte_idx = 0; byte_idx < num_bytes_to_copy; byte_idx++) {
    output_buffer[byte_idx] = input_buffer[byte_idx];
  }
}
```

E.4.7. can_controller.h

```
/*
 * can_controller.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _CAN_CONTROLLER_H_
#define _CAN_CONTROLLER_H_

#include <p30f6014A.h>
#include "can_frame.h"
#include "common.h"


/* The receive buffer 0 and the receive buffer 1 of a dsPIC's CAN controller
 * have different control registers, therefore we distinguish between the two
 * types of receive buffers. */
typedef enum {
  RX_BUFFER0,
  RX_BUFFER1
} t_rx_buffer_type;


/* CAN receive buffer */
struct rx_buffer_struct {
  /* First word of the receive buffer. The second, third and fourth word
   * are in contiguous memory locations following the first word */
  volatile unsigned int *data;
  /* Pointer to receive buffer standard identifier register */
  volatile CxRXxSIDBITS *SIDbits;
  /* Pointer to receive buffer data length code register */
  volatile CxRXxDLCBITS *DLCbits;
  t_rx_buffer_type type;
  /* Pointer to receive buffer control register */
  /* ... for receive buffers of type RX_BUFFER0 */
  volatile CxRX0CONBITS *CONbitsRX0;
  /* ... for receive buffers of type RX_BUFFER1 */
  volatile CxRX1CONBITS *CONbitsRX1;
  /* Pointer to the SID acceptance filter mask register */
  volatile CxRXMxSIDBITS *MaskSIDbits;
  /* TODO: Add masks for extended ids? */
};


/* CAN transmit buffer */
struct tx_buffer_struct {
  /* First word of the transmit buffer. The second, third and fourth word
   * are in contiguous memory locations following the first word */
  volatile unsigned int *data;
  /* Pointer to CAN Standard Identifier register */
  volatile CxTXxSIDBITS *SIDbits;
  /* Pointer to transmit buffer data length code register */
  volatile CxTXxDLCBITS *DLCbits;
  /* Pointer to transmit buffer control register */
  volatile CxTXxCONBITS *CONbits;
};


/* Number of receive buffers per CAN controller.
 * The dsPIC's CAN controllers have two visible receive buffers (the third
 * buffer is the message assembly buffer (MAB) and is not directly
 * accessible). */
#define RX_BUFFER_COUNT 2
/* Number of transmit buffers per CAN controller. */
#define TX_BUFFER_COUNT 3
/* Number of acceptance filters per CAN controller */
#define ACCEPTANCE_FILTER_COUNT 6


/*
 * Keeps information about a CAN controller
 */
struct can_controller {
  volatile CxINTFBITS *INTFbits;
  /* Pointer to CAN control and status register */
  volatile CxCTRLBITS *CTRLbits;
  /* Pointer to CAN baud rate configuration register 1 */
  volatile CxCFG1BITS *CFG1bits;
  /* Pointer to CAN baud rate configuration register 2 */
  volatile CxCFG2BITS *CFG2bits;
  /* Pointer to CAN interrupt enable register */
  volatile CxINTEBITS *INTEbits;
  /* The CAN controller's receive buffers */
  struct rx_buffer_struct *rx_buffer[RX_BUFFER_COUNT];
  /* Pointers to the SID acceptance filter registers */
  volatile CxRXFxSIDBITS *FilterSIDbits[ACCEPTANCE_FILTER_COUNT];
  /* The CAN controller's transmit buffers */
  struct tx_buffer_struct *tx_buffer[TX_BUFFER_COUNT];
  /* The rx_buffer that contains the received frame (NULL if no buffer
   * has a frame) */
  volatile struct rx_buffer_struct *rx_buffer_loaded;
};


/*
 * Operations of the CAN controller ADT
 */

/* Initializes both CAN controllers */
void init_can_controllers(void);

/* Returns true if 'ctrl' is the controller that has the transmission
 * controller role assigned to; returns false otherwise */
bool is_transmission_controller(
  const volatile struct can_controller *const ctrl
);

/* Shuts 'ctrl' down, disabling all its interrupts and aborting all its
 * transmissions */
void shutdown(
  volatile struct can_controller *const ctrl
);

/* Instructs one of ctrl's free transmission buffers to transmit the frame
 * 'frame_to_tx' */
void request_tx(
  volatile struct can_controller *const ctrl,
  struct can_frame *frame_to_tx
);

/* Returns true if an error occured at 'ctrl' */
bool error_irq_occurred(
  const volatile struct can_controller *const ctrl
);

/* Assigns the transmission controller role to 'ctrl' */
void set_transmission_controller(
  volatile struct can_controller *const ctrl
);

/* Returns a pointer to the controller that is currently the transmission
 * controller */
volatile struct can_controller *get_transmission_controller(void);



/*
 * Operations of the nested reception buffer ADT
 */

/* Releases the reception buffer 'buf_to_release' of 'ctrl' so that it is free
 * to receive a new frame */
void release_rx_buffer(
  volatile struct can_controller *const ctrl,
  volatile struct rx_buffer_struct *buf_to_release
);

/* Returns the receive buffer of controller 'ctrl' where the last received
 * frame has been stored */
volatile struct rx_buffer_struct* get_rx_event_causing_rx_buffer(
  volatile struct can_controller *const ctrl
);

/* Marks the receive buffer 'buf' as being the one which received the last
 * frame */
void set_rx_event_causing_rx_buffer(
  volatile struct can_controller *const ctrl,
  volatile struct rx_buffer_struct *const buf
);

/* Reads the frame containd within 'buffer_to_read' into 'frame_read' */
void read_frame(
  volatile struct rx_buffer_struct *const buffer_to_read,
  volatile struct can_frame *const frame_read
);

/* Returns true if reception buffer 0 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool rx_buffer0_irq_occured(
  const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that reception buffer 0 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_rx_buffer0_irq(
  const volatile struct can_controller *const ctrl
);

/* Returns true if reception buffer 1 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool rx_buffer1_irq_occured(
  const volatile struct can_controller *const ctrl
);
```

```
/* Clears the flag that indicates that reception buffer 1 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_rx_buffer1_irq(
  const volatile struct can_controller *const ctrl
);



/*
 * Operations of the nested transmission buffer ADT
 */

/* Returns true if transmission buffer 0 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool tx_buffer0_irq_occurred(
  const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that transmission buffer 0 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_tx_buffer0_irq(
  const volatile struct can_controller *const ctrl
);

/* Returns true if transmission buffer 1 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool tx_buffer1_irq_occurred(
  const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that transmission buffer 1 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_tx_buffer1_irq(
  const volatile struct can_controller *const ctrl
);

/* Returns true if transmission buffer 2 of 'ctrl' generated a CAN combined
 * interrupt; returns false otherwise */
bool tx_buffer2_irq_occurred(
  const volatile struct can_controller *const ctrl
);

/* Clears the flag that indicates that transmission buffer 2 of 'ctrl' caused a
 * CAN combined interrupt */
void clear_tx_buffer2_irq(
  const volatile struct can_controller *const ctrl
);

#endif /* _CAN_CONTROLLER_H_ */
```

E.4.8. can_controller.c

```
/*
 * can_controller.c
 *
 * Written by David Gessner <davidges@gmail.com>
 *
 * Edited by Alberto Ballesteros <ballesteros.alberto@gmail.com>
 *
 */

#include <p30f6014A.h>
#include "can_controller.h"
#include "common.h"
#include "assert.h"
#include "can_frame.h"



/*
 * Controller 1 receive buffer initialization
 */
static struct rx_buffer_struct ctrl1_rx_buffer0 = {
  .data = &C1RX0B1,
  .SIDbits = &C1RX0SIDbits,
  .DLCbits = &C1RX0DLCbits,
  .type = RX_BUFFER0,
  .CONbitsRX0 = &C1RX0CONbits,
  .CONbitsRX1 = NULL,
  .MaskSIDbits = &C1RXM0SIDbits,
};

static struct rx_buffer_struct ctrl1_rx_buffer1 = {
  .data = &C1RX1B1,
  .SIDbits = &C1RX1SIDbits,
  .DLCbits = &C1RX1DLCbits,
  .type = RX_BUFFER1,
  .CONbitsRX0 = NULL,
  .CONbitsRX1 = &C1RX1CONbits,
  .MaskSIDbits = &C1RXM1SIDbits,
};



/*
 * Controller 1 transmit buffer initialization
 */
static struct tx_buffer_struct ctrl1_tx_buffer0 = {
  .data = &C1TX0B1,
  .SIDbits = &C1TX0SIDbits,
  .DLCbits = &C1TX0DLCbits,
  .CONbits = &C1TX0CONbits
};

struct tx_buffer_struct ctrl1_tx_buffer1 = {
  .data = &C1TX1B1,
  .SIDbits = &C1TX1SIDbits,
  .DLCbits = &C1TX1DLCbits,
  .CONbits = &C1TX1CONbits
};

static struct tx_buffer_struct ctrl1_tx_buffer2 = {
  .data = &C1TX2B1,
  .SIDbits = &C1TX2SIDbits,
  .DLCbits = &C1TX2DLCbits,
  .CONbits = &C1TX2CONbits
};



/*
 * Controller 1 initialization
 */
struct can_controller ctrl1 = {
  .INTFbits = &C1INTFbits,
  .CTRLbits = &C1CTRLbits,
  .CFG1bits = &C1CFG1bits,
  .CFG2bits = &C1CFG2bits,
  .INTEbits = &C1INTEbits,

  .rx_buffer[0] = &ctrl1_rx_buffer0,
  .rx_buffer[1] = &ctrl1_rx_buffer1,

  .FilterSIDbits[0] = &C1RXF0SIDbits,
  .FilterSIDbits[1] = &C1RXF1SIDbits,
  .FilterSIDbits[2] = &C1RXF2SIDbits,
  .FilterSIDbits[3] = &C1RXF3SIDbits,
  .FilterSIDbits[4] = &C1RXF4SIDbits,
  .FilterSIDbits[5] = &C1RXF5SIDbits,

  .tx_buffer[0] = &ctrl1_tx_buffer0,
  .tx_buffer[1] = &ctrl1_tx_buffer1,
  .tx_buffer[2] = &ctrl1_tx_buffer2,

  .rx_buffer_loaded = NULL,
};



/*
 * Controller 2 receive buffer initialization
 */
static struct rx_buffer_struct ctrl2_rx_buffer0 = {
  .data = &C2RX0B1,
  .SIDbits = &C2RX0SIDbits,
  .DLCbits = &C2RX0DLCbits,
  .type = RX_BUFFER0,
  .CONbitsRX0 = &C2RX0CONbits,
  .CONbitsRX1 = NULL,
  .MaskSIDbits = &C2RXM0SIDbits,
};

static struct rx_buffer_struct ctrl2_rx_buffer1 = {
  .data = &C2RX1B1,
  .SIDbits = &C2RX1SIDbits,
  .DLCbits = &C2RX1DLCbits,
  .type = RX_BUFFER1,
  .CONbitsRX0 = NULL,
  .CONbitsRX1 = &C2RX1CONbits,
  .MaskSIDbits = &C2RXM1SIDbits,
};
```

```
/*
 * Controller 2 transmit buffer initialization
 */
static struct tx_buffer_struct ctrl2_tx_buffer0 = {
  .data = &C2TX0B1,
  .SIDbits = &C2TX0SIDbits,
  .DLCbits = &C2TX0DLCbits,
  .CONbits = &C2TX0CONbits
};

static struct tx_buffer_struct ctrl2_tx_buffer1 = {
  .data = &C2TX1B1,
  .SIDbits = &C2TX1SIDbits,
  .DLCbits = &C2TX1DLCbits,
  .CONbits = &C2TX1CONbits
};

static struct tx_buffer_struct ctrl2_tx_buffer2 = {
  .data = &C2TX2B1,
  .SIDbits = &C2TX2SIDbits,
  .DLCbits = &C2TX2DLCbits,
  .CONbits = &C2TX2CONbits
};



/*
 * Controller 2 initialization
 */
struct can_controller ctrl2 = {
  .INTFbits = &C2INTFbits,
  .CTRLbits = &C2CTRLbits,
  .CFG1bits = &C2CFG1bits,
  .CFG2bits = &C2CFG2bits,
  .INTEbits = &C2INTEbits,

  .rx_buffer[0] = &ctrl2_rx_buffer0,
  .rx_buffer[1] = &ctrl2_rx_buffer1,

  .FilterSIDbits[0] = &C2RXF0SIDbits,
  .FilterSIDbits[1] = &C2RXF1SIDbits,
  .FilterSIDbits[2] = &C2RXF2SIDbits,
  .FilterSIDbits[3] = &C2RXF3SIDbits,
  .FilterSIDbits[4] = &C2RXF4SIDbits,
  .FilterSIDbits[5] = &C2RXF5SIDbits,

  .tx_buffer[0] = &ctrl2_tx_buffer0,
  .tx_buffer[1] = &ctrl2_tx_buffer1,
  .tx_buffer[2] = &ctrl2_tx_buffer2,

  .rx_buffer_loaded = NULL,
};


/* The current transmission controller */
static volatile struct can_controller *tx_controller = &ctrl1;

/* CAN Module Operation Modes, used with the REQOP field of the CAN
 * control and status register (CTRLbits) of ctrl1 and ctrl2. */
typedef enum t_can_mode_enum {
  CAN_MODE_NORMAL = 00,
  CAN_MODE_DISABLE = 01,
  CAN_MODE_LOOPBACK = 02,
  CAN_MODE_LISTEN_ONLY = 03,
  CAN_MODE_CONFIG = 04,
  /* 05 and 06 are reserved in REQOP */
  CAN_MODE_LISTEN_ALL_MSGS = 07
} t_can_mode;

typedef enum {
  CAN_TX_PRIORIY_HIGHEST = 03,
  CAN_TX_PRIORIY_HIGH_INTERMEDIATE = 02,
  CAN_TX_PRIORIY_LOW_INTERMEDIATE = 01,
  CAN_TX_PRIORIY_LOWEST = 00
} t_can_tx_priority;


/* BRP_VALUE: CAN Baud Rate Prescaler. Valid values are 0, 1, ... 63
 * CAN_PROP_TQ: Length in time quanta for the propagation segment, valid values
 * are 1, 2, ... 8
 * CAN_SEG1_TQ: Length in time quanta for the phase segment 1, valid values are
 * 1, 2, ... 8
 * CAN_SEG2_TQ: Length in time quanta for the phase segment 2, valid values are
 * 1, 2, ... 8
 *
 * The nominal bit rate, NBR, is:
 *   NBR = 1 / NBT
 * where NBT is the nominal bit time. The NBT in turn is:
 *   NBT = NOMINAL_BIT_TIME_TQ * TQ
 * where NOMINAL_BIT_TIME_TQ is the number of time quanta the nominal bit time
 * is made of (defined below) and where TQ is the length of a time quantum. TQ
 * is:
 *   TQ = 2 * (BRP_VALUE + 1) / FCAN
 * Therefore the NBR is:
 *   NBR = NOMINAL_BIT_TIME_TQ * ( 2 * (BRP_VALUE + 1) / FCAN)
 *
 */

#define BRP_VALUE 0
#define CAN_PROP_TQ 1
#define CAN_SEG1_TQ 4
#define CAN_SEG2_TQ 2

#define SYNCHRONOUS_JUMP_WIDTH_TQ 1
/* Sync segment is always one time quantum */
#define CAN_SYNC_TQ 1
/* Number of time quanta the nominal bit time is made of */
#define NOMINAL_BIT_TIME_TQ (CAN_SYNC_TQ + CAN_PROP_TQ + CAN_SEG1_TQ + \
  CAN_SEG2_TQ)

/* Converts a given number of time quanta to a corresponding value which
 * can be assigned to a CAN baud rate configuration register field. For
 * instance, to configure a SJW of 1 TQ the value 0 must be assigned to
 * the SJW field of a CiCFG1 register. */
static unsigned int TQ_to_config_value(
   unsigned int time_quanta
)
{
  return time_quanta - 1;
}



static void enable_can_interrupts(
  struct can_controller *ctrl
)
{
  /* The invalid message received interrupt and the bus wake up
   * activity interrupt are left disabled. */
  ctrl->INTEbits->IVRIE = 0;
  ctrl->INTEbits->WAKIE = 0;

  /* Enable error interrupt */
  ctrl->INTEbits->ERRIE = 1;

  /* Enable transmit buffer 2 interrupt */
  ctrl->INTEbits->TX2IE = 1;
  /* Enable transmit buffer 1 interrupt */
  ctrl->INTEbits->TX1IE = 1;
  /* Enable transmit buffer 0 interrupt */
  ctrl->INTEbits->TX0IE = 1;

  /* Enable receive buffer 1 interrupt */
  ctrl->INTEbits->RX1IE = 1;
  /* Enable receive buffer 0 interrupt */
  ctrl->INTEbits->RX0IE = 1;
}



static void disable_can_interrupts(
  volatile struct can_controller *const ctrl
)
{
  /* The invalid message received interrupt and the bus wake up
   * activity interrupt are left disabled. */
  ctrl->INTEbits->IVRIE = 0;
  ctrl->INTEbits->WAKIE = 0;

  /* Disable error interrupt */
  ctrl->INTEbits->ERRIE = 0;

  /* Disable transmit buffer 2 interrupt */
  ctrl->INTEbits->TX2IE = 0;
  /* Disable transmit buffer 1 interrupt */
  ctrl->INTEbits->TX1IE = 0;
  /* Disable transmit buffer 0 interrupt */
  ctrl->INTEbits->TX0IE = 0;

  /* Disable receive buffer 1 interrupt */
  ctrl->INTEbits->RX1IE = 0;
  /* Disable receive buffer 0 interrupt */
  ctrl->INTEbits->RX0IE = 0;
}



static void init_acceptance_filters(
  struct can_controller *ctrl
)
{
  int filter_idx;

  for (filter_idx = 0; filter_idx < ACCEPTANCE_FILTER_COUNT;
    filter_idx++) {
    /* Enable filter for standard identifier and not extended
```

```
     * identifier */
    ctrl->FilterSIDbits[filter_idx]->EXIDE = 0;

    /* SID to match doesn't matter because the masks in
     * init_acceptance_filter_masks() have been configured to
     * accept every message. We can therefore initialize the
     * SID field with an arbitrary value (we use 0). */
    ctrl->FilterSIDbits[filter_idx]->SID = 0;
  }
}



/*
 * The message acceptance filters and masks determine if a message in the
 * message assembly buffer (MAB) should be loaded into one of the receive
 * buffers. The filters and masks are applied to the message identifier.
 * The mask determines which bits of the identifier should be examined and
 * the filters contain values to which those bits are compared. The bits
 * from the identifier that are masked, i.e. the corresponding mask bit is
 * zero, will always be accepted by the filters. The bits that are not
 * masked, i.e. the corresponding mask bit is one, will be accepted if
 * there is a match with the corresponding filter bit. If all the bits are
 * accepted then the message is accepted and loaded from the MAB into one
 * of the receive buffers.
 *
 * Messages whose identifier match filters RXF0 or RXF1 are loaded into
 * receive buffer 0 (RXB0), messages whose identifier match any of the
 * filters RXF2 through RXF5 are loaded into receive buffer 1 (RXB1).  The
 * mask RXM0 is used with filters RXF0 and RXF1 and the mask RXM1 is used
 * with filters RXF2-RXF5.
 */
static void init_acceptance_filter_masks(
  struct can_controller *ctrl
)
{
  int rx_buffer_idx;

  for (rx_buffer_idx = 0; rx_buffer_idx < RX_BUFFER_COUNT;
    rx_buffer_idx++) {
    /* Configure acceptance filter mask to accept all messages,
     * i.e. all the masks' bits are set to zero therefore all
     * messages are accepted independently of the value of the
     * acceptance filters. */
    ctrl->rx_buffer[rx_buffer_idx]->MaskSIDbits->SID = 0;

    /* Match only message types (SID or EID) as determined by
     * the EXIDE bit in the corresponding filters */
    ctrl->rx_buffer[rx_buffer_idx]->MaskSIDbits->MIDE = 1;
  }
}



static void init_controller(
  struct can_controller *ctrl
)
{
  int tx_buffer_idx = 0;

  /* Stop CAN module when device enters idle mode. */
  ctrl->CTRLbits->CSIDL = 1;
```

```
/* FCAN clock is FCY (instruction cycle clock) instead of
 * FOSC = 4 x FCY */
ctrl->CTRLbits->CANCKS = 1;
/* Don't generate a capture signal whenever a valid frame has been
 * accepted */
ctrl->CTRLbits->CANCAP = 0;

/* Set configuration mode */
ctrl->CTRLbits->REQOP = CAN_MODE_CONFIG;

/* Wait until the CAN module has entered configuration mode */
while (ctrl->CTRLbits->OPMODE != CAN_MODE_CONFIG);

ctrl->CFG1bits->SJW =
  TQ_to_config_value(SYNCHRONOUS_JUMP_WIDTH_TQ);

ctrl->CFG1bits->BRP = BRP_VALUE;

/* CAN bus line filter is not used for wake-up */
ctrl->CFG2bits->WAKFIL = 0;
/* The length of Phase Segment 2 is Freely programmable */
ctrl->CFG2bits->SEG2PHTS = 1;
/* Bus line is sampled once at the sample point */
ctrl->CFG2bits->SAM = 0;

/* Set number of time quanta to use for propagation segment */
ctrl->CFG2bits->PRSEG = TQ_to_config_value(CAN_PROP_TQ);
/* Set number of time quanta to use for segment 1 */
ctrl->CFG2bits->SEG1PH = TQ_to_config_value(CAN_SEG1_TQ);
/* Set number of time quanta to use for segment 2 */
ctrl->CFG2bits->SEG2PH = TQ_to_config_value(CAN_SEG2_TQ);

/*
 * Configure transmit buffers
 */
for (tx_buffer_idx = 0; tx_buffer_idx < TX_BUFFER_COUNT;
  tx_buffer_idx++) {
  /* Clear transmit request bit */
  ctrl->tx_buffer[tx_buffer_idx]->CONbits->TXREQ = 0;
  /* TODO: Should all transmit buffers have the same
   * priority? */
  ctrl->tx_buffer[tx_buffer_idx]->CONbits->TXPRI =
    CAN_TX_PRIORIY_HIGHEST;
}

/*
 * Configure receive buffers
 */
/* Clear receive full status bit */
ctrl->rx_buffer[0]->CONbitsRX0->RXFUL = 0;
ctrl->rx_buffer[1]->CONbitsRX1->RXFUL = 0;
/* Disable double buffer, i.e. no receive buffer 0 overflow to
 * receive buffer 1 */
ctrl->rx_buffer[0]->CONbitsRX0->DBEN = 0;

/* TODO: Allow the user to override the receive and transmit
 * buffer configuration by adding corresponding functions to
 * recancentrate.h */

init_acceptance_filter_masks(ctrl);

/* TODO: Allow the user to override the mask configuration and to
```

```
   * also configure the acceptance filter. */

  init_acceptance_filters(ctrl);

  enable_can_interrupts(ctrl);

  /* Set normal mode */
  ctrl->CTRLbits->REQOP = CAN_MODE_NORMAL;

/* The simulator does not model the CAN module */
#ifndef SIMULATOR
  /* Wait until the CAN module has entered normal mode */
  while (ctrl->CTRLbits->OPMODE != CAN_MODE_NORMAL);
#endif
}



void init_can_controllers(void)
{
  /*
   * Restrictions on the CAN bit time segments (see the dsPIC30F Family
   * Reference Manual for details):
   */
  ASSERT(CAN_PROP_TQ + CAN_SEG1_TQ >= CAN_SEG2_TQ);
  ASSERT(CAN_SEG2_TQ > SYNCHRONOUS_JUMP_WIDTH_TQ);
  ASSERT(8 <= NOMINAL_BIT_TIME_TQ);
  ASSERT(NOMINAL_BIT_TIME_TQ <= 25);
  ASSERT(0 <= BRP_VALUE);
  ASSERT(BRP_VALUE <= 63);

  init_controller(&ctrl1);
  init_controller(&ctrl2);
}



bool is_transmission_controller(
  const volatile struct can_controller *const ctrl
)
{
  return (tx_controller == ctrl);
}



void shutdown(
  volatile struct can_controller *const ctrl
)
{
  disable_can_interrupts(ctrl);
  ctrl->CTRLbits->ABAT = 1;
}



static bool has_tx_pending(
  struct tx_buffer_struct *tx_buffer
)
{
  return tx_buffer->CONbits->TXREQ;
}
```

```
/* Returns a transmit buffer of ctrl which has no transmission pending.
 * If no free transmit buffer is available it returns NULL. */
static struct tx_buffer_struct* get_free_tx_buffer(
  volatile struct can_controller *const ctrl
)
{
  struct tx_buffer_struct *current_tx_buffer;
  struct tx_buffer_struct *free_tx_buffer = NULL;
  /* Transmission buffer index */
  unsigned int tx_buffer_idx;
  bool free_tx_buffer_found = false;

  /* Search for a free transmit buffer, i.e. a transmit buffer which has
   * no transmission pending */
  tx_buffer_idx = 0;
  while (!free_tx_buffer_found && tx_buffer_idx < TX_BUFFER_COUNT) {
    current_tx_buffer = ctrl->tx_buffer[tx_buffer_idx];
    if (!has_tx_pending(current_tx_buffer)) {
      free_tx_buffer = current_tx_buffer;
      free_tx_buffer_found = true;
    }
    tx_buffer_idx++;
  }

  if (free_tx_buffer_found) {
    return free_tx_buffer;
  } else {
    return NULL;
  }
}



/* Instructs one of ctrl's free transmission buffers to transmit the frame
 * frame_to_tx */
void request_tx(
  volatile struct can_controller *const ctrl,
  struct can_frame *frame_to_tx
)
{
  struct tx_buffer_struct *tx_buffer_to_use;
  unsigned int count;

  tx_buffer_to_use = get_free_tx_buffer(ctrl);

  if (tx_buffer_to_use == NULL) {
    /* TODO: What shall we do if there is no free tx buffer?
     * For the moment we ASSERT(false) */
    ASSERT(false);
  }

  tx_buffer_to_use->SIDbits->SID5_0 = frame_to_tx->identifier & 0x003F;
  tx_buffer_to_use->SIDbits->SID10_6 = frame_to_tx->identifier & 0x07C0;

  /* Copy frame_to_tx's data to the transmit buffer's data register */
  copy_data(frame_to_tx->data, frame_to_tx->length,
    (unsigned char*) tx_buffer_to_use->data);

  tx_buffer_to_use->DLCbits->DLC = frame_to_tx->length;
  /* Signal transmit buffer to enqueue the loaded frame for transmission.
   * The transmission will start when the transmit buffer detects that
   * the medium is available. */
  tx_buffer_to_use->CONbits->TXREQ = 1;


  /*********************************************************************/
  /** Code enabling the single-shot transmission mode ******************/
  /*********************************************************************/

  /*
  // wait for the frame to start to be transmitted
  count = 0;
  while(count < 40){
    count++;
  }

  // signal transmit buffer to unenqueue the last frame. That results in the
  // frame to be transmitted but if an error occurs it is not retransmitted.
  tx_buffer_to_use->CONbits->TXREQ = 0;
  */
  /*********************************************************************/
  /*********************************************************************/
}



void release_rx_buffer(
  volatile struct can_controller *const ctrl,
  volatile struct rx_buffer_struct *buf_to_release
)
{
  if (buf_to_release->type == RX_BUFFER0) {
    ASSERT(ctrl->rx_buffer[0] == buf_to_release);
    buf_to_release->CONbitsRX0->RXFUL = 0;
  } else if (buf_to_release->type == RX_BUFFER1) {
    ASSERT(ctrl->rx_buffer[1] == buf_to_release);
    buf_to_release->CONbitsRX1->RXFUL = 0;
  } else {
    /* Invalid receive buffer type */
    ASSERT(false);
  }
  set_rx_event_causing_rx_buffer(ctrl, NULL);
}



/* Returns the receive buffer of controller 'ctrl' where the last received
 * frame has been stored and which, thus, caused the last CAN combined
 * interrupt for 'ctrl' that was triggered due to a reception. */
volatile struct rx_buffer_struct* get_rx_event_causing_rx_buffer(
  volatile struct can_controller *const ctrl
)
{
  ASSERT(ctrl->rx_buffer_loaded != NULL);

  return ctrl->rx_buffer_loaded;
}
```

```
void set_rx_event_causing_rx_buffer(
  volatile struct can_controller *const ctrl,
  volatile struct rx_buffer_struct *const buf
)
{
  ctrl->rx_buffer_loaded = buf;
}



void read_frame(
  volatile struct rx_buffer_struct *const buffer_to_read,
  volatile struct can_frame *const output_frame
)
{
  int num_bytes_to_read = buffer_to_read->DLCbits->DLC;

  /* Assert the frame has a standard identifier and not an
   * extended identifier */
  ASSERT(buffer_to_read->SIDbits->RXIDE == 0);

  output_frame->identifier = buffer_to_read->SIDbits->SID;

  /* Copy contents of the receive buffer to the data field of
   * output_frame */
  copy_data((unsigned char*) buffer_to_read->data, num_bytes_to_read,
    output_frame->data);

  output_frame->length = num_bytes_to_read;
}



bool tx_buffer0_irq_occurred(
  const volatile struct can_controller *const ctrl
)
{
  return ctrl->INTFbits->TX0IF;
}



void clear_tx_buffer0_irq(
  const volatile struct can_controller *const ctrl
)
{
  ctrl->INTFbits->TX0IF = 0;
}



bool tx_buffer1_irq_occurred(
  const volatile struct can_controller *const ctrl
)
{
  return ctrl->INTFbits->TX1IF;
}



void clear_tx_buffer1_irq(
  const volatile struct can_controller *const ctrl
```

```
)
{
  ctrl->INTFbits->TX1IF = 0;
}



bool tx_buffer2_irq_occurred(
  const volatile struct can_controller *const ctrl
)
{
  return ctrl->INTFbits->TX2IF;
}



void clear_tx_buffer2_irq(
  const volatile struct can_controller *const ctrl
)
{
  ctrl->INTFbits->TX2IF = 0;
}



bool rx_buffer0_irq_occured(
  const volatile struct can_controller *const ctrl
)
{
  return ctrl->INTFbits->RX0IF;
}



void clear_rx_buffer0_irq(
  const volatile struct can_controller *const ctrl
)
{
  ctrl->INTFbits->RX0IF = 0;
}



bool rx_buffer1_irq_occured(
  const volatile struct can_controller *const ctrl
)
{
  return ctrl->INTFbits->RX1IF;
}



void clear_rx_buffer1_irq(
  const volatile struct can_controller *const ctrl
)
{
  ctrl->INTFbits->RX1IF = 0;
}



bool error_irq_occurred(
  const volatile struct can_controller *const ctrl
)
{
  return (ctrl->INTFbits->ERRIF);
}



void set_transmission_controller(
  volatile struct can_controller *const ctrl
)
{
  tx_controller = ctrl;
}



volatile struct can_controller *get_transmission_controller(void)
{
  return tx_controller;
}
```

E.4.9. led.h

```
/*
 * led.h
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#ifndef _LED_H_
#define _LED_H_



void init_leds(void);

void led_display(char b);

#endif /* _LED_H_ */
```

E.4.10. led.c

```
/*
 * led.c
 *
 * Written by David Gessner <davidges@gmail.com>
 */

#include <p30f6014A.h>


void init_leds(void)
{
  /* The LEDs are connected to pins 4-7 of port D on the dspic demo
   * board. */

  /* LEDs initially turned off */
  PORTD = 0;
  /* TRISD configures each pin of port D as either an input (1) or an
   * output (0). Set RD7 to RD4, i.e. LED4 to LED1, as outputs */
  TRISD = 0xFF0F;
}



/* Lights the 4 LEDs to display 'b' in binary */
void led_display(char b)
{
  PORTD &= 0xFF0F;
  PORTD |= ((b & 0x000F)<<4);
}
```

F. Fault Injection Management Station source code

F.1. Packages

F.1.1. constants.h

```
//////////////
// NCC IDs  //
//////////////

#define BROAD_ID 0x000
#define PC_ID    0x010

#define HUB_FIM_ID 0x001
#define HUB_LOG_ID 0x002

#define NODE0_LOG_ID 0x003
#define NODE1_LOG_ID 0x004
#define NODE2_LOG_ID 0x005


/////////////////////
//  Command codes  //
/////////////////////

#define CFG_CMD_CODE 0x00
#define MCH_CMD_CODE 0x01
#define LOG_CMD_CODE 0x02


/////////////////////
//  Configuration  //
/////////////////////

// Protocol command codes

#define VALUE_TYPE_CODE              0x00
#define VALUE_BFVALUE_CODE           0x01

#define LINK_CODE                    0x02

#define MODE_CODE                    0x03

#define START_TRIGGER_FILTER_CODE    0x04
#define START_TRIGGER_MASK_CODE      0x05
#define START_TRIGGER_MASK_LONG_CODE 0x06
#define START_TRIGGER_FIELD_CODE     0x07
#define START_TRIGGER_LINK_CODE      0x08
#define START_TRIGGER_ROLE_CODE      0x09
#define START_TRIGGER_COUNT_CODE     0x0A
```

```
#define END_TRIGGER_FILTER_CODE      0x0B
#define END_TRIGGER_MASK_CODE        0x0C
#define END_TRIGGER_MASK_LONG_CODE   0x0D
#define END_TRIGGER_FIELD_CODE       0x0E
#define END_TRIGGER_LINK_CODE        0x0F
#define END_TRIGGER_ROLE_CODE        0x10
#define END_TRIGGER_COUNT_CODE       0x11

#define SEL_TRIGGER_FILTER_CODE      0x12
#define SEL_TRIGGER_MASK_CODE        0x13
#define SEL_TRIGGER_MASK_LONG_CODE   0x14
#define SEL_TRIGGER_FIELD_CODE       0x15
#define SEL_TRIGGER_LINK_CODE        0x16
#define SEL_TRIGGER_ROLE_CODE        0x17

#define START_FIELD_CODE             0x18
#define START_BIT_CODE               0x19
#define START_OFFSET_CODE            0x1A

#define END_FIELD_CODE               0x1B
#define END_BIT_CODE                 0x1C

#define END_BC_CODE                  0x1D

#define EOC_CODE                     0x1F

// Protocol values

#define SAD_VALUE         0x00
#define SAR_VALUE         0x01
#define BF_VALUE          0x02
#define INV_VALUE         0x03

#define PORT0UP_VALUE     0x00
#define PORT0DW_VALUE     0x01
#define PORT1UP_VALUE     0x02
#define PORT1DW_VALUE     0x03
#define PORT2UP_VALUE     0x04
#define PORT2DW_VALUE     0x05
#define PORT3UP_VALUE     0x06
#define PORT3DW_VALUE     0x07
#define PORT4UP_VALUE     0x08
#define PORT4DW_VALUE     0x09
#define COUPLED_VALUE     0x0F

#define FULLRANGE_VALUE   0x00
#define ITERATIVE_VALUE   0x01
#define SINGLESHOT_VALUE  0x02

#define IDLE_VALUE        0x00
#define ID_VALUE          0x01
#define RTR_VALUE         0x02
#define RES_VALUE         0x03
#define DLC_VALUE         0x04
#define DATA_VALUE        0x05
#define CRC_VALUE         0x06
#define CRC_DELIM_VALUE   0x07
#define ACK_VALUE         0x08
#define ACK_DELIM_VALUE   0x09
#define EOF_VALUE         0x0A
#define INTERFIELD_VALUE  0x0B
```

```
#define ERR_FLAG_VALUE    0x0C
#define ERR_DELIM_VALUE   0x0D

#define DC_VALUE          0x00
#define TX_VALUE          0x01
#define RX_VALUE          0x02


///////////////////
//  Mode change  //
///////////////////

// Mode codes

#define MC_ECM 0x00
#define MC_EIM 0x01
#define MC_EWM 0x02
#define MC_EEM 0x03


///////////////
//  Logging  //
///////////////

// Protocol command codes

#define HUB_FINAL_PORT_STATE_CODE 0x00
#define HUB_STORED_FRAME_CODE     0x01

#define NODE_ERROR_COUNTERS_CODE  0x00
#define NODE_STORED_FRAME_CODE    0x01

#define EOL_CODE              0x1F

// String conversion

const char* portStateStr[] = {
  "idle",     //0x00
  "active",   //0x01
  "disabled", //0x10
  "rein␣idle" //0x11
};

const char* portStr[] = {
  "port0", //0x000
  "port1", //0x001
  "port2", //0x010
  "port3"  //0x011
};

const char* fieldStr[] = {
  "idle",       //0x00
  "id",         //0x01
  "rtr",        //0x02
  "res",        //0x03
  "dlc",        //0x04
  "data",       //0x05
  "crc",        //0x06
  "crc␣delim",  //0x07
  "ack",        //0x08
  "ack␣delim",  //0x09
  "eof",        //0x0A
  "interfield", //0x0B
  "error␣flag", //0x0C
  "error␣delim" //0x0D
};
```

F.1.2. file_spec.h

```
// File key strings

#define VALUE_TYPE_STR           "value_type"
#define VALUE_BFVALUE_STR        "value_pattern"

#define LINK_STR                 "target_link"

#define MODE_STR                 "mode"

#define START_TRIGGER_FILTER_STR "aim_filter"
#define START_TRIGGER_FIELD_STR  "aim_field"
#define START_TRIGGER_LINK_STR   "aim_link"
#define START_TRIGGER_ROLE_STR   "aim_role"
#define START_TRIGGER_COUNT_STR  "aim_count"

#define END_TRIGGER_FILTER_STR   "withdraw_filter"
#define END_TRIGGER_FIELD_STR    "withdraw_field"
#define END_TRIGGER_LINK_STR     "withdraw_link"
#define END_TRIGGER_ROLE_STR     "withdraw_role"
#define END_TRIGGER_COUNT_STR    "withdraw_count"

#define SEL_TRIGGER_FILTER_STR   "target_frame_filter"
#define SEL_TRIGGER_FIELD_STR    "target_frame_field"
#define SEL_TRIGGER_LINK_STR     "target_frame_link"
#define SEL_TRIGGER_ROLE_STR     "target_frame_role"

#define START_FIELD_STR          "fire_field"
#define START_BIT_STR            "fire_bit"
#define START_OFFSET_STR         "fire_offset"

#define END_FIELD_STR            "cease_field"
#define END_BIT_STR              "cease_bit"
#define END_BC_STR               "cease_bc"

// File value strings

#define SAD_STR        "dominant"
#define SAR_STR        "recessive"
#define BF_STR         "pattern"
#define INV_STR        "inverse"

#define PORT0UP_STR    "port0up"
#define PORT0DW_STR    "port0dw"
#define PORT1UP_STR    "port1up"
#define PORT1DW_STR    "port1dw"
#define PORT2UP_STR    "port2up"
#define PORT2DW_STR    "port2dw"
#define PORT3UP_STR    "port3up"
#define PORT3DW_STR    "port3dw"
#define PORT4UP_STR    "port4up"
#define PORT4DW_STR    "port4dw"
#define COUPLED_STR    "coupled"

#define FULLRANGE_STR  "continuous"
#define ITERATIVE_STR  "iterative"
#define SINGLESHOT_STR "single-shot"

#define IDLE_STR       "idle"
#define ID_STR         "id"
#define RTR_STR        "rtr"
```

```
#define RES_STR        "res"
#define DLC_STR        "dlc"
#define DATA_STR       "data"
#define CRC_STR        "crc"
#define CRC_DELIM_STR  "crcDelim"
#define ACK_STR        "ack"
#define ACK_DELIM_STR  "ackDelim"
#define EOF_STR        "eof"
#define INTERFIELD_STR "interfield"
#define ERR_FLAG_STR   "errFlag"
#define ERR_DELIM_STR  "errDelim"

#define DC_STR         "dont_care"
#define TX_STR         "tx"
#define RX_STR         "rx"
```

F.1.3. bit.h

```
#include <linux/types.h>

// Type definitions
#define uchar unsigned char  // 1 byte

uchar extractBits(uchar byte, uchar first, uchar size);

int   extractBytes(char * str, uchar * bytes, int maxBytes);
char* cleanString(char* str, char c);

// binary values

bool  isBinary(char * str);
uchar toBinary(char * byte);


// filter values (0, 1, X)

bool isFilter(char * str);

char * getFilterValue(char * str);
char * getFilterMask (char * str);
```

F.1.4. bit.c

```
#include "bit.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

// Extracts from 'byte' a bit vector starting in first and ending in
// first+size-1
uchar extractBits(uchar byte, uchar first, uchar size){
  uchar temp_byte;

  temp_byte = byte << (first-1);
  temp_byte = temp_byte >> (8-size);

  return temp_byte;
}

// Converts a binary string into a byte array
int extractBytes(char * str, uchar* bytes, int maxBytes){
  char substr[8];
  int  byteCount = 0;
  int  length = strlen(str);

  for(int i=0; i<length; i+=8){
    memcpy(substr, &str[i], 8);
    while(strlen(substr) < 8){
      strcat(substr,"0");
    }

    if(byteCount < maxBytes){
      bytes[byteCount] = toBinary(substr);
      byteCount++;
    }else break;
  }

  return byteCount;
}

// Deletes all the characters matching 'c' inside a string
char* cleanString(char* str, char c){
  int length = strlen(str);
  char * tmp = (char *)malloc(sizeof(char) * length);

  int j = 0;
  for(int i=0; i<length; i++){
    if(str[i] != c){
      tmp[j] = str[i];
      j++;
    }
  }

  tmp[j] = '\0';
  return tmp;
}


//
// Binary
//
```

```
bool isBinary(char * str){
  int length = strlen(str);

  for(int i=0; i<length; i++){
    if(str[i] != '0' && str[i] != '1'){
      return false;
    }
  }
  return true;
}

uchar toBinary(char * byte){
  uchar value = 0;

  int length = strlen(byte);

  for(int i=0; i<length; i++){
    if(byte[i] == '1'){
      value += pow(2,length-i-1);
    }
  }

  return value;
}


//
// Filter
//

bool isFilter(char * str){
  int length = strlen(str);

  for(int i=0; i<length; i++){
    if(str[i] != '0' && str[i] != '1' && str[i] != 'x'){
      return false;
    }
  }
  return true;
}

char * getFilterValue(char * str){
  char * value = (char *)malloc(sizeof(char) * strlen(str));

  for(int i=0; i<strlen(str); i++){
    if(str[i] == '1') value[i] = '1';
    else              value[i] = '0';
  }
  return value;
}

char * getFilterMask(char * str){
  char * value = (char *)malloc(sizeof(char) * strlen(str));

  for(int i=0; i<strlen(str); i++){
    if(str[i] == 'x') value[i] = '0';
    else              value[i] = '1';
  }
  return value;
}
```

F.1.5. Argument.hpp

```
class Argument {
public:
  Argument(char* text_arg);

  char  getFlag();
  char* getValue();

private:
  char  flag;
  char* value;
};
```

F.1.6. Argument.cpp

```
#include "Argument.hpp"

#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <fnmatch.h>


char* substr(char* str, int i1, int i2);


Argument::Argument(char* text_arg){
  if(text_arg == NULL){
    //null argument
    this->flag  = 0;
    this->value = NULL;
  }

  //match -<char>
  if(fnmatch("-?", text_arg, 0) == 0){
    this->flag  = text_arg[1];
    this->value = NULL;

  //match -<char>=<string>
  }else if(fnmatch("-?=*", text_arg, 0) == 0 ){
    this->flag  = text_arg[1];
    this->value = substr(text_arg, 3,-1);

  }else{
    //null argument
    this->flag  = 0;
    this->value = NULL;
  }
}

char Argument::getFlag(){
  return this->flag;
}

char* Argument::getValue(){
  return this->value;
}


// Additional functions

char* substr(char* str, int i1, int i2){
  if(str == NULL) return NULL;

  int len = strlen(str);
  if(len == 0) return NULL;

  if(i2 == -1) i2 = len-1;

  if(i1 < 0 || i2 > len-1 || i1 > i2) return NULL;

  int tmp_len = (i2-i1 + 1) + 1; //insert  \0 at the end

  char* tmp = (char *)malloc(tmp_len * sizeof(char));
  tmp[tmp_len-1] = '\0';

  int i = 0;
  while(i < tmp_len-1){
    tmp[i] = str[i1 + i];
    i++;
  }

  return tmp;
}
```

F.1.7. CAN.h

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>

#include <linux/can.h>
#include <linux/can/raw.h>

#include "bit.h"

struct t_CANFrame {
  ushort id;
  uchar  dlc;
  uchar  data[8];
};

class CAN{
  public:
    CAN();

    int  openSocket(char* iface);
    void closeSocket();

    bool isOpen();

    void send(ushort id, uchar dlc, ...);
    void send(t_CANFrame frame);
    t_CANFrame receive();

  private:
    int sckt; //socket
    bool scktOpen;
};

void print(t_CANFrame frame);
```

F.1.8. CAN.cpp

```
#include "CAN.hpp"

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#include <stdarg.h>

CAN::CAN(){
  scktOpen = false;
}

int CAN::openSocket(char* iface){
  struct sockaddr_can addr;
  struct ifreq ifr;

  // create socket
  if ((sckt = socket(PF_CAN, SOCK_RAW, CAN_RAW)) < 0) {
    perror("**␣ERROR␣**:␣socket\n");
    return -1;
  }

  // create socket direction
  strcpy(ifr.ifr_name, iface);
  if (ioctl(sckt, SIOCGIFINDEX, &ifr) < 0) {
    perror("**␣ERROR␣**:␣SIOCGIFINDEX\n");
    return -1;
  }
  addr.can_ifindex = ifr.ifr_ifindex;
  addr.can_family  = AF_CAN;

  // bind
  if (bind(sckt, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
    perror("**␣ERROR␣**:␣bind\n");
    return -1;
  }

  scktOpen = true;
  return 0;
}


void CAN::closeSocket(){
  close(sckt);
  scktOpen = false;
}


bool CAN::isOpen(){
  return scktOpen;
}


void CAN::send(ushort id, uchar dlc, ...){
  struct can_frame frame;
  va_list ap;

  va_start(ap, dlc);
  int i = 0;
  while(i < dlc){
    frame.data[i] = va_arg(ap, int);
    i++;
  }

  va_end(ap);

  frame.can_id  = id;
  frame.can_dlc = dlc;

  write(sckt, &frame, sizeof(frame));
}


void CAN::send(t_CANFrame frame){
  struct can_frame sframe;
  va_list ap;

  for(int i=0; i<frame.dlc; i++){
    sframe.data[i] = frame.data[i];
  }

  sframe.can_id  = frame.id;
  sframe.can_dlc = frame.dlc;

  write(sckt, &frame, sizeof(sframe));
}


t_CANFrame CAN::receive(){
  struct can_frame rframe;
  read(sckt, &rframe, sizeof(struct can_frame));

  t_CANFrame frame;
  frame.id   = rframe.can_id;
  frame.dlc  = rframe.can_dlc;
  for(int i=0; i<frame.dlc; i++){
    frame.data[i] = rframe.data[i];
  }

  return frame;
}



void print(t_CANFrame frame){
  printf("%x␣[%d]␣", frame.id, frame.dlc);
  for(int i=0; i<frame.dlc; i++){
    printf("%x␣", frame.data[i]);
  }
}
```

F.2. Fault Injection Configurator

F.2.1. fic.cpp

```
/*
 * Fault-injection Configurator
 * usage: fic [options]
 *
 * options:
 *   -c          Check config file, reads the whole file to find errors
 *   -v          Verbose
 *   -f=<file>   Select the config file
 *   -i=<iface>  Select the CAN interface
 *   -h          Show this message and exit
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <glib.h>

#include <limits.h>

#include "../constants.h"
#include "../bit.h"

#include "Argument.hpp"
#include "../CAN.hpp"

#include "file_spec.h"

// Main constants
#define DEFAULT_IFACE "can0"
#define DEFAULT_CONFIGFILE_NAME "config"

#define MODE_CHECK 0
#define MODE_EXEC  1


// Global variables
GKeyFile * configFile;
char* configFileName;

CAN can;
char* iface;

int program_mode;

bool verbose;

bool checking;

// Additional functions
void parseArguments(int argc, char **argv);
int  initConfigFile();

int sendConfigCmd(gchar * group, gchar * key, bool justCheck);

bool  getBoolean(gchar * group, gchar * key, int * errors);
```

```
int   getInteger(gchar * group, gchar * key, int * errors, int min=INT_MIN, int max=INT_MAX );
uchar getField  (gchar * group, gchar * key, int * errors);

void showHelp();
void end(int status);

/*---------------------------------------------*/
/*-------------------- Main --------------------*/
/*---------------------------------------------*/

int main(int argc, char **argv){

  printf("\n");
  printf("␣*********************************\n");
  printf("␣**␣Fault-injection␣Configurator␣**\n");
  printf("␣*********************************\n");
  printf("\n");

  //
  // General variables
  //

  gchar ** groups;
  gchar *  group;
  gchar ** keys;
  gchar *  key;

  gsize * size = new gsize;

  int err;
  GError *error = NULL;

  int i,j;


  //
  // Initializations
  //

  configFileName = (char*)DEFAULT_CONFIGFILE_NAME;
  iface = (char*)DEFAULT_IFACE;
  program_mode = MODE_EXEC;
  verbose = false;

  int errors = 0;

  // parse arguments
  parseArguments(argc, argv);

  // open file
  printf("initializing␣file␣\"%s\"...\n", configFileName);
  err = initConfigFile();
  if(err < 0) end(EXIT_FAILURE);

  // read groups
  printf("processing␣file␣...\n");

  groups = g_key_file_get_groups(configFile, size);

  if(size != NULL){
    printf("␣␣␣%lu␣group(s)␣found!\n\n", *size);
  }else{
    printf("␣␣␣%d␣groups␣found!\n\n", 0);
    end(EXIT_FAILURE);
  }


  if(program_mode == MODE_EXEC){
    printf("initializing␣'%s'␣socket␣...\n\n", iface);
    err = can.openSocket(iface);
    if(err < 0) end(EXIT_FAILURE);
  }


  //
  // Main loop
  //

  for(int pm = MODE_CHECK; pm <= program_mode; pm++){

    if      (pm == MODE_CHECK) printf("checking␣file␣...\n");
    else if (pm == MODE_EXEC ) printf("processing␣file␣...\n");

    i=0;
    group = groups[i];

    while(group != NULL){
      if(verbose) printf("[%d]␣reading␣'%s'\n", i, group);

      keys = g_key_file_get_keys(
        configFile,
        group,
        size,
        &error
      );

      if(keys == NULL){
        printf(
          "**␣ERROR␣**:␣Impossible␣to␣read␣group␣\"%s\"␣-␣%s\n",
          group,
          error->message
        );
        g_error_free(error);
        end(EXIT_FAILURE);
      }

      j=0;
      key = keys[0];
      while(key != NULL){
        if(verbose) printf("␣␣␣[%.2d]␣reading␣'%s'\n", j, key);

        if      (pm == MODE_CHECK) errors += sendConfigCmd(group, key, true );
        else if (pm == MODE_EXEC ) errors += sendConfigCmd(group, key, false);

        j++;
        key = keys[j];
      }

      if(pm == MODE_EXEC && errors == 0){
        can.send(HUB_FIM_ID, 1, EOC_CODE);
      }

      i++;
      group = groups[i];
    }

    if(pm == MODE_CHECK){
      printf("␣␣␣%d␣error(s)␣found!\n\n", errors);
      if(errors > 0) end(EXIT_FAILURE);
    }
  }

  end(EXIT_SUCCESS);
}


/* ------------------------------------------ */
/* ---------- Config file functions ---------- */
/* ------------------------------------------ */

int sendConfigCmd(gchar * group, gchar * key, bool justCheck){

  //
  // Checkings
  //

  // Check whether the groups exists
  if(g_key_file_has_group(configFile, group) == FALSE){
    printf("**␣Error␣**:␣Impossible␣to␣find␣group␣'%s'\n", group);
    end(EXIT_FAILURE);
  }

  // Check whether the key exists
  GError *error = NULL;
  if(g_key_file_has_key(configFile, group, key, &error) == FALSE){
    printf("**␣Error␣**:␣Impossible␣to␣find␣key␣'%s'\n", key);
    end(EXIT_FAILURE);
  }


  //
  // Key parsing
  //

  int errors = 0;

  /* -------------------------------- */
  /* ---------- value_type ---------- */
  /* -------------------------------- */

  if(strcmp(key, VALUE_TYPE_STR) == 0) {
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool  valueInList = true;
    if      (strcmp(value, SAD_STR) == 0){ byte = SAD_VALUE;    }
    else if (strcmp(value, SAR_STR) == 0){ byte = SAR_VALUE;    }
    else if (strcmp(value, BF_STR ) == 0){ byte = BF_VALUE;     }
    else if (strcmp(value, INV_STR) == 0){ byte = INV_VALUE;    }
    else                                 { valueInList = false; }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }
    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        VALUE_TYPE_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* --------- value_bfvalue -------- */
  /* -------------------------------- */

  }else if (strcmp(key, VALUE_BFVALUE_STR) == 0){
    gchar * tvalue = g_key_file_get_string(configFile, group, key, NULL);
    char  * value  = cleanString(tvalue, '.');

    int length = strlen(value);

    // check length
    if(length > 48){
      printf(
        "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣long␣must␣be␣between␣0␣and␣48\n",
        group, key, value
      );
      errors++;
    }

    // check whether contains a binary value
    if(!isBinary(value)){
      printf(
        "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣is␣not␣a␣binary␣value\n",
        group, key, value
      );
      errors++;
    }

    // send frame
    if(errors == 0 && !justCheck){
      uchar bytes[7];
      int  byteCount;

      //extract bytes
      byteCount = extractBytes(value, bytes, 6);

      // bfvalue
      can.send(
        HUB_FIM_ID, byteCount+2,
        VALUE_BFVALUE_CODE,
        length,
        bytes[0], bytes[1], bytes[2], bytes[3],
        bytes[4], bytes[5], bytes[6]
      );
    }

  /* -------------------------------- */
  /* ------------- link ------------- */
  /* -------------------------------- */

  }else if (strcmp(key, LINK_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool valueInList = true;

    if      (strcmp(value, PORT0UP_STR) == 0){ byte = PORT0UP_VALUE; }
    else if (strcmp(value, PORT0DW_STR) == 0){ byte = PORT0DW_VALUE; }
    else if (strcmp(value, PORT1UP_STR) == 0){ byte = PORT1UP_VALUE; }
    else if (strcmp(value, PORT1DW_STR) == 0){ byte = PORT1DW_VALUE; }
    else if (strcmp(value, PORT2UP_STR) == 0){ byte = PORT2UP_VALUE; }
    else if (strcmp(value, PORT2DW_STR) == 0){ byte = PORT2DW_VALUE; }
    else if (strcmp(value, PORT3UP_STR) == 0){ byte = PORT3UP_VALUE; }
    else if (strcmp(value, PORT3DW_STR) == 0){ byte = PORT3DW_VALUE; }
    else if (strcmp(value, PORT4UP_STR) == 0){ byte = PORT4UP_VALUE; }
    else if (strcmp(value, PORT4DW_STR) == 0){ byte = PORT4DW_VALUE; }
    else                                     { valueInList = false;  }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        LINK_CODE,
        byte
      );
    }


  /* -------------------------------- */
  /* ------------- mode ------------- */
  /* -------------------------------- */

  }else if (strcmp(key, MODE_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool valueInList = true;

    if      (strcmp(value, FULLRANGE_STR ) == 0){ byte = FULLRANGE_VALUE;  }
    else if (strcmp(value, ITERATIVE_STR ) == 0){ byte = ITERATIVE_VALUE;  }
    else if (strcmp(value, SINGLESHOT_STR) == 0){ byte = SINGLESHOT_VALUE; }
    else                                        { valueInList = false;     }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        MODE_CODE,
        byte
      );
    }



  /* -------------------------------- */
  /* ------ start_trigger_filter ---- */
  /* -------------------------------- */
```

```
}else if (strcmp(key, START_TRIGGER_FILTER_STR) == 0){
  gchar * tvalue = g_key_file_get_string(configFile, group, key, NULL);
  char  * value  = cleanString(tvalue, '.');

  int length = strlen(value);

  // check length
  if(length > 56){
    printf(
      "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣long␣must␣be␣between␣0␣and␣56\n",
      group, key, value
    );
    errors++;
  }

  // check whether contains a filter value (0, 1 or X)
  if(!isFilter(value)){
    printf(
      "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣is␣not␣a␣filter␣value\n",
      group, key, value
    );
    errors++;
  }

  // send frames
  if(errors == 0 && !justCheck){
    char * fvalue = getFilterValue(value);
    char * fmask  = getFilterMask (value);

    uchar bytes[7];
    int  byteCount = 0;

    //
    // filter value
    //

    // extract filter value bytes
    byteCount = extractBytes(fvalue, bytes, 7);

    // send filter value
    can.send(
      HUB_FIM_ID, byteCount+1 ,
      START_TRIGGER_FILTER_CODE,
      bytes[0], bytes[1], bytes[2], bytes[3],
      bytes[4], bytes[5], bytes[6]
    );

    //
    // filter mask
    //

    // extract filter mask bytes
    byteCount = extractBytes(fmask, bytes, 7);

    // send filter mask
    can.send(
      HUB_FIM_ID, byteCount+1 ,
      START_TRIGGER_MASK_CODE,
      bytes[0], bytes[1], bytes[2], bytes[3],
      bytes[4], bytes[5], bytes[6]
    );
```

```
      //
      // filter long
      //

      // search last 1 in mask
      int flong = 0;
      for(int i=length-1; i>=0; i--){
        flong = i+1;
        if(fmask[i] == '1') break;
      }

      // send filter mask
      can.send(
        HUB_FIM_ID, 2 ,
        START_TRIGGER_MASK_LONG_CODE,
        flong
      );
    }

  /* -------------------------------- */
  /* ------ start_trigger_field ----- */
  /* -------------------------------- */

  }else if (strcmp(key, START_TRIGGER_FIELD_STR) == 0){
    uchar byte = getField(group, key, &errors);

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        START_TRIGGER_FIELD_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------ start_trigger_link ------ */
  /* -------------------------------- */

  }else if (strcmp(key, START_TRIGGER_LINK_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool  valueInList = true;

    if      (strcmp(value, PORT0UP_STR) == 0){ byte = PORT0UP_VALUE; }
    else if (strcmp(value, PORT0DW_STR) == 0){ byte = PORT0DW_VALUE; }
    else if (strcmp(value, PORT1UP_STR) == 0){ byte = PORT1UP_VALUE; }
    else if (strcmp(value, PORT1DW_STR) == 0){ byte = PORT1DW_VALUE; }
    else if (strcmp(value, PORT2UP_STR) == 0){ byte = PORT2UP_VALUE; }
    else if (strcmp(value, PORT2DW_STR) == 0){ byte = PORT2DW_VALUE; }
    else if (strcmp(value, PORT3UP_STR) == 0){ byte = PORT3UP_VALUE; }
    else if (strcmp(value, PORT3DW_STR) == 0){ byte = PORT3DW_VALUE; }
    else if (strcmp(value, PORT4UP_STR) == 0){ byte = PORT4UP_VALUE; }
    else if (strcmp(value, PORT4DW_STR) == 0){ byte = PORT4DW_VALUE; }
    else if (strcmp(value, COUPLED_STR) == 0){ byte = COUPLED_VALUE; }
    else                                     { valueInList = false;  }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }
    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        START_TRIGGER_LINK_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------ start_trigger_role ------ */
  /* -------------------------------- */

  }else if (strcmp(key, START_TRIGGER_ROLE_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool valueInList = true;

    if      (strcmp(value, DC_STR  ) == 0){ byte = DC_VALUE;     }
    else if (strcmp(value, TX_STR  ) == 0){ byte = TX_VALUE;     }
    else if (strcmp(value, RX_STR  ) == 0){ byte = RX_VALUE;     }
    else                                  { valueInList = false; }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        START_TRIGGER_ROLE_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------ start_trigger_count ----- */
  /* -------------------------------- */

  }else if (strcmp(key, START_TRIGGER_COUNT_STR) == 0){
    const uint min = 0;
    const uint max = 65535;

    int value = getInteger(group, key, &errors, min, max);

    //extract less and most significant bytes
    uchar ls =  value & 0x000000FF;
    uchar ms = (value & 0x0000FF00) >> 8;

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 3,
        START_TRIGGER_COUNT_CODE,
        ms, ls
      );
    }

  /* -------------------------------- */
  /* ------ end_trigger_filter ------ */
  /* -------------------------------- */
```

```
}else if (strcmp(key, END_TRIGGER_FILTER_STR) == 0){
  gchar * tvalue = g_key_file_get_string(configFile, group, key, NULL);
  char  * value  = cleanString(tvalue, '.');

  int length = strlen(value);

  // check length
  if(length > 56){
    printf(
      "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣long␣must␣be␣between␣0␣and␣56\n",
      group, key, value
    );
    errors++;
  }

  // check whether contains a filter value (0, 1 or X)
  if(!isFilter(value)){
    printf(
      "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣is␣not␣a␣filter␣value\n",
      group, key, value
    );
    errors++;
  }

  // send frames
  if(errors == 0 && !justCheck){
    char * fvalue = getFilterValue(value);
    char * fmask  = getFilterMask (value);

    uchar bytes[7];
    int  byteCount = 0;

    //
    // filter value
    //

    // extract filter value bytes
    byteCount = extractBytes(fvalue, bytes, 7);

    // send filter value
    can.send(
      HUB_FIM_ID, byteCount+1 ,
      END_TRIGGER_FILTER_CODE,
      bytes[0], bytes[1], bytes[2], bytes[3],
      bytes[4], bytes[5], bytes[6]
    );

    //
    // filter mask
    //

    // extract filter mask bytes
    byteCount = extractBytes(fmask, bytes, 7);

    // send filter mask
    can.send(
      HUB_FIM_ID, byteCount+1 ,
      END_TRIGGER_MASK_CODE,
      bytes[0], bytes[1], bytes[2], bytes[3],
      bytes[4], bytes[5], bytes[6]
    );
```

```
      //
      // filter long
      //

      // search last 1 in mask
      int flong = 0;
      for(int i=length-1; i>=0; i--){
        flong = i+1;
        if(fmask[i] == '1') break;
      }

      // send filter mask
      can.send(
        HUB_FIM_ID, 2 ,
        END_TRIGGER_MASK_LONG_CODE,
        flong
      );
    }

  /* -------------------------------- */
  /* ------- end_trigger_field ------ */
  /* -------------------------------- */

  }else if (strcmp(key, END_TRIGGER_FIELD_STR) == 0){
    uchar byte = getField(group, key, &errors);

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        END_TRIGGER_FIELD_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------- end_trigger_link ------- */
  /* -------------------------------- */

  }else if (strcmp(key, END_TRIGGER_LINK_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool  valueInList = true;

    if      (strcmp(value, PORT0UP_STR) == 0){ byte = PORT0UP_VALUE; }
    else if (strcmp(value, PORT0DW_STR) == 0){ byte = PORT0DW_VALUE; }
    else if (strcmp(value, PORT1UP_STR) == 0){ byte = PORT1UP_VALUE; }
    else if (strcmp(value, PORT1DW_STR) == 0){ byte = PORT1DW_VALUE; }
    else if (strcmp(value, PORT2UP_STR) == 0){ byte = PORT2UP_VALUE; }
    else if (strcmp(value, PORT2DW_STR) == 0){ byte = PORT2DW_VALUE; }
    else if (strcmp(value, PORT3UP_STR) == 0){ byte = PORT3UP_VALUE; }
    else if (strcmp(value, PORT3DW_STR) == 0){ byte = PORT3DW_VALUE; }
    else if (strcmp(value, PORT4UP_STR) == 0){ byte = PORT4UP_VALUE; }
    else if (strcmp(value, PORT4DW_STR) == 0){ byte = PORT4DW_VALUE; }
    else if (strcmp(value, COUPLED_STR) == 0){ byte = COUPLED_VALUE; }
    else                                     { valueInList = false;  }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }
    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        END_TRIGGER_LINK_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------- end_trigger_role ------- */
  /* -------------------------------- */

  }else if (strcmp(key, END_TRIGGER_ROLE_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool valueInList = true;

    if      (strcmp(value, DC_STR  ) == 0){ byte = DC_VALUE;     }
    else if (strcmp(value, TX_STR  ) == 0){ byte = TX_VALUE;     }
    else if (strcmp(value, RX_STR  ) == 0){ byte = RX_VALUE;     }
    else                                  { valueInList = false; }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        END_TRIGGER_ROLE_CODE,
        byte
      );
    }


  /* -------------------------------- */
  /* ------- end_trigger_count ------ */
  /* -------------------------------- */

  }else if (strcmp(key, END_TRIGGER_COUNT_STR) == 0){
    const uint min = 1;
    const uint max = 65535;

    int value = getInteger(group, key, &errors, min, max);

    if(errors == 0 && !justCheck){
      //extract less and most significant bytes
      uchar ls =  value & 0x000000FF;
      uchar ms = (value & 0x0000FF00) >> 8;

      can.send(
        HUB_FIM_ID, 3,
        END_TRIGGER_COUNT_CODE,
        ms, ls
      );
    }


  /* -------------------------------- */
  /* ------ sel_trigger_filter ------ */
  /* ------------------------------- */

  }else if (strcmp(key, SEL_TRIGGER_FILTER_STR) == 0){
    gchar * tvalue = g_key_file_get_string(configFile, group, key, NULL);
    char  * value  = cleanString(tvalue, '.');

    int length = strlen(value);

    // check length
    if(length > 56){
      printf(
        "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣long␣must␣be␣between␣0␣and␣56\n",
        group, key, value
      );
      errors++;
    }

    // check whether contains a filter value (0, 1 or X)
    if(!isFilter(value)){
      printf(
        "**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣is␣not␣a␣filter␣value\n",
        group, key, value
      );
      errors++;
    }

    // send frames
    if(errors == 0 && !justCheck){
      char * fvalue = getFilterValue(value);
      char * fmask  = getFilterMask (value);

      uchar bytes[7];
      int  byteCount = 0;

      //
      // filter value
      //

      // extract filter value bytes
      byteCount = extractBytes(fvalue, bytes, 7);

      // send filter value
      can.send(
        HUB_FIM_ID, byteCount+1 ,
        SEL_TRIGGER_FILTER_CODE,
        bytes[0], bytes[1], bytes[2], bytes[3],
        bytes[4], bytes[5], bytes[6]
      );

      //
      // filter mask
      //

      // extract filter mask bytes
      byteCount = extractBytes(fmask, bytes, 7);

      // send filter mask
      can.send(
        HUB_FIM_ID, byteCount+1 ,
        SEL_TRIGGER_MASK_CODE,
        bytes[0], bytes[1], bytes[2], bytes[3],
        bytes[4], bytes[5], bytes[6]
      );

      //
      // filter long
      //

      // search last 1 in mask
      int flong = 0;
      for(int i=length-1; i>=0; i--){
        flong = i+1;
        if(fmask[i] == '1') break;
      }

      // send filter mask
      can.send(
        HUB_FIM_ID, 2 ,
        SEL_TRIGGER_MASK_LONG_CODE,
        flong
      );
    }

  /* -------------------------------- */
  /* ------- sel_trigger_field ------ */
  /* -------------------------------- */

  }else if (strcmp(key, SEL_TRIGGER_FIELD_STR) == 0){
    uchar byte = getField(group, key, &errors);

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        SEL_TRIGGER_FIELD_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------- sel_trigger_link ------- */
  /* -------------------------------- */

  }else if (strcmp(key, SEL_TRIGGER_LINK_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool  valueInList = true;

    if      (strcmp(value, PORT0UP_STR) == 0){ byte = PORT0UP_VALUE; }
    else if (strcmp(value, PORT0DW_STR) == 0){ byte = PORT0DW_VALUE; }
    else if (strcmp(value, PORT1UP_STR) == 0){ byte = PORT1UP_VALUE; }
    else if (strcmp(value, PORT1DW_STR) == 0){ byte = PORT1DW_VALUE; }
    else if (strcmp(value, PORT2UP_STR) == 0){ byte = PORT2UP_VALUE; }
    else if (strcmp(value, PORT2DW_STR) == 0){ byte = PORT2DW_VALUE; }
    else if (strcmp(value, PORT3UP_STR) == 0){ byte = PORT3UP_VALUE; }
    else if (strcmp(value, PORT3DW_STR) == 0){ byte = PORT3DW_VALUE; }
    else if (strcmp(value, PORT4UP_STR) == 0){ byte = PORT4UP_VALUE; }
    else if (strcmp(value, PORT4DW_STR) == 0){ byte = PORT4DW_VALUE; }
    else if (strcmp(value, COUPLED_STR) == 0){ byte = COUPLED_VALUE; }
    else                                      { valueInList = false;  }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        SEL_TRIGGER_LINK_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------- sel_trigger_role ------- */
  /* -------------------------------- */

  }else if (strcmp(key, SEL_TRIGGER_ROLE_STR) == 0){
    gchar * value = g_key_file_get_string(configFile, group, key, NULL);

    uchar byte;
    bool valueInList = true;

    if      (strcmp(value, DC_STR  ) == 0){ byte = DC_VALUE;     }
    else if (strcmp(value, TX_STR  ) == 0){ byte = TX_VALUE;     }
    else if (strcmp(value, RX_STR  ) == 0){ byte = RX_VALUE;     }
    else                                  { valueInList = false; }

    if(!valueInList){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
      errors++;
    }

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        SEL_TRIGGER_ROLE_CODE,
        byte
      );
    }


  /* -------------------------------- */
  /* ---------- start_field --------- */
  /* -------------------------------- */

  }else if (strcmp(key, START_FIELD_STR) == 0){
    uchar byte = getField(group, key, &errors);

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        START_FIELD_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ----------- start_bit ---------- */
  /* -------------------------------- */

  }else if (strcmp(key, START_BIT_STR) == 0){
    const uint min = 0;
    const uint max = 63;
    int value = getInteger(group, key, &errors, min, max);

    // send frame
    if(errors == 0 && !justCheck){
      // extract byte
      uchar byte = (char)value;

      can.send(
        HUB_FIM_ID, 2,
        START_BIT_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* --------- start_offset --------- */
  /* -------------------------------- */

  }else if (strcmp(key, START_OFFSET_STR) == 0){
    const uint min = 0;
    const uint max = 65535;

    int value = getInteger(group, key, &errors, min, max);

    if(errors == 0 && !justCheck){
      //extract less and most significant bytes
      uchar ls =  value & 0x000000FF;
      uchar ms = (value & 0x0000FF00) >> 8;

      can.send(
        HUB_FIM_ID, 3,
        START_OFFSET_CODE,
        ms, ls
      );
    }


  /* -------------------------------- */
  /* ----------- end_field ---------- */
  /* -------------------------------- */

  }else if (strcmp(key, END_FIELD_STR) == 0){
    uchar byte = getField(group, key, &errors);

    if(errors == 0 && !justCheck){
      can.send(
        HUB_FIM_ID, 2 ,
        END_FIELD_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------------ end_bit ----------- */
  /* -------------------------------- */

  }else if (strcmp(key, END_BIT_STR) == 0){
    const uint min = 0;
    const uint max = 63;

    int value = getInteger(group, key, &errors, min, max);
```

```
    // send frame
    if(errors == 0 && !justCheck){
      // extract byte
      uchar byte = (char)value;

      can.send(
        HUB_FIM_ID, 2,
        END_BIT_CODE,
        byte
      );
    }

  /* -------------------------------- */
  /* ------------ end_bc ------------ */
  /* -------------------------------- */

  }else if (strcmp(key, END_BC_STR) == 0){
    const uint min = 0;
    const uint max = 65535;

    int value = getInteger(group, key, &errors, min, max);

    if(errors == 0 && !justCheck){
      //extract less and most significant bytes
      uchar ls =  value & 0x000000FF;
      uchar ms = (value & 0x0000FF00) >> 8;

      can.send(
        HUB_FIM_ID, 3,
        END_BC_CODE,
        ms, ls
      );
    }


  /* -------------------------------- */
  /* ----------- not a key ---------- */
  /* -------------------------------- */

  }else{
    printf("**␣Error␣**:␣Impossible␣to␣parse␣[%s]␣'%s'\n", group, key);
    errors++;
  }

  return errors;
}


/*---------- Extract boolean value from file ----------*/

bool getBoolean(gchar * group, gchar * key, int * errors){
  GError *gerror = NULL;
  gboolean value = g_key_file_get_boolean(configFile, group, key, &gerror);

  if(gerror != NULL){
    if(gerror->code == G_KEY_FILE_ERROR_INVALID_VALUE){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣is␣not␣a␣boolean␣value\n",
        group,
        key,
        g_key_file_get_string(configFile, group, key, NULL)
      );
    }else printf("**␣Error␣**:␣reading␣[%s]␣'%s'\n", group, key);

    (*errors)++;
  }

  return value;
}


/*---------- Extract integer value from file ----------*/

int getInteger(gchar * group, gchar * key, int * errors, int min, int max ){
  GError *gerror = NULL;
  int value = g_key_file_get_integer(configFile, group, key, &gerror);

  if(gerror != NULL){
    if(gerror->code == G_KEY_FILE_ERROR_INVALID_VALUE){
      printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣is␣not␣an␣integer␣value\n",
        group,
        key,
        g_key_file_get_string(configFile, group, key, NULL)
      );

    }else printf("**␣Error␣**:␣reading␣[%s]␣'%s'\n", group, key);

    (*errors)++;
  }

  if(*errors == 0 && (value < min || value > max)){
    printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣must␣be␣%d<x<%d\n",
      group, key,
      g_key_file_get_string(configFile, group, key, &gerror),
      min, max
    );

    (*errors)++;
  }
}


/*---------- Extract field value from file ----------*/

uchar getField(gchar * group, gchar * key, int * errors){
  gchar * value = g_key_file_get_string(configFile, group, key, NULL);

  uchar field = IDLE_VALUE;

  if      (strcmp(value, IDLE_STR       ) == 0){ field = IDLE_VALUE;       }
  else if (strcmp(value, ID_STR         ) == 0){ field = ID_VALUE;         }
  else if (strcmp(value, RTR_STR        ) == 0){ field = RTR_VALUE;        }
  else if (strcmp(value, RES_STR        ) == 0){ field = RES_VALUE;        }
  else if (strcmp(value, DLC_STR        ) == 0){ field = DLC_VALUE;        }
  else if (strcmp(value, DATA_STR       ) == 0){ field = DATA_VALUE;       }
  else if (strcmp(value, CRC_STR        ) == 0){ field = CRC_VALUE;        }
  else if (strcmp(value, CRC_DELIM_STR  ) == 0){ field = CRC_DELIM_VALUE;  }
  else if (strcmp(value, ACK_STR        ) == 0){ field = ACK_VALUE;        }
  else if (strcmp(value, ACK_DELIM_STR  ) == 0){ field = ACK_DELIM_VALUE;  }
  else if (strcmp(value, EOF_STR        ) == 0){ field = EOF_VALUE;        }
  else if (strcmp(value, INTERFIELD_STR ) == 0){ field = INTERFIELD_VALUE; }
  else if (strcmp(value, ERR_FLAG_STR   ) == 0){ field = ERR_FLAG_VALUE;   }
  else if (strcmp(value, ERR_DELIM_STR  ) == 0){ field = ERR_DELIM_VALUE;  }
  else{
    printf("**␣Error␣**:␣[%s]␣'%s'␣=␣'%s',␣invalid␣value\n", group, key, value);
    (*errors)++;
  }

  return field;
}


/*---------- Parse arguments ----------*/

void parseArguments(int argc, char **argv){
  int i = 1;
  while(i < argc){
    Argument arg(argv[i]);

    if      (arg.getFlag() == 'c'){ program_mode = MODE_CHECK; }
    else if (arg.getFlag() == 'v'){ verbose = true; }

    else if (arg.getFlag() == 'f'){ configFileName = arg.getValue(); }
    else if (arg.getFlag() == 'i'){ iface          = arg.getValue(); }

    else if (arg.getFlag() == 'h'){ showHelp(); exit(EXIT_SUCCESS); }

    else
      printf(
        "**␣WARNING␣**:␣argument␣\"%s\"␣not␣recognized␣-␣I␣ignore␣it\n",
        argv[i]
      );

    i++;
  }
}


/*---------- Initialize config-file ----------*/

int initConfigFile(){
  configFile = g_key_file_new();
  GError *error = NULL;

  if (!g_key_file_load_from_file(
      configFile,
      configFileName,
      G_KEY_FILE_KEEP_COMMENTS,
      &error)
  ){
    printf(
      "**␣ERROR␣**:␣Impossible␣to␣load␣\"%s\"␣-␣%s\n",
      configFileName, error->message
    );
    g_error_free(error);
    return -1;
  }

  return 0;
}


/*---------- Show help ----------*/

void showHelp(){
  printf("Fault-injection␣Coordinator\n"                                     );
  printf("usage:␣fic␣[options]\n"                                                   );
  printf("\n"                                                                       );
  printf("options:\n"                                                               );
  printf("␣␣-c␣␣␣␣␣␣␣␣␣␣Checks␣config␣file,␣reads␣the␣whole␣file␣to␣find␣errors\n");
  printf("␣␣-v␣␣␣␣␣␣␣␣␣␣verbose\n"                                                  );
  printf("␣␣-f=<file>␣␣␣Selects␣the␣configuration␣file\n"                           );
  printf("␣␣-i=<iface>␣␣Selects␣the␣CAN␣interface\n"                                );
  printf("␣␣-h␣␣␣␣␣␣␣␣␣␣Show␣this␣message␣and␣exits\n"                              );
}


/*---------- End ----------*/

void end(int status){
  printf("Ending...\n");

  // close socket
  if(can.isOpen()){
    can.closeSocket();
  }

  printf("Done!\n");
  exit(status);
}
```

F.3. Fault Injection Log retriever

F.3.1. fil.cpp

```
#include <stdio.h>
#include <stdlib.h>

#include "../constants.h"
#include "../bit.h"
#include "../CAN.hpp"

// Main constants
#define DEFAULT_IFACE "can0"

CAN can;
char* iface;

bool verbose;


// Functions

void hubLogPrinter (ushort hubLoggerId );
void nodeLogPrinter(ushort nodeLoggerId);

void end(int status);


int main(int argc, char **argv){

  printf("\n");
  printf("␣**********************************\n");
  printf("␣**␣Fault-Injection␣Log␣retriever␣**\n");
  printf("␣**********************************\n");
  printf("\n");

  //
  // Initializations
  //

  iface = (char*)DEFAULT_IFACE;
  verbose = false;

  printf("initializing␣'%s'␣socket␣...\n\n", iface);
  int err = can.openSocket(iface);
  if(err < 0) end(EXIT_FAILURE);

  printf("retrieving␣log␣data...\n\n");

/*
  printf("---------\n");
  printf("-- HUB --\n");
  printf("---------\n");
  hubLogPrinter (HUB_LOG_ID);

  printf("-----------\n");
  printf("-- NODE0 --\n");
  printf("-----------\n");
  nodeLogPrinter(NODE0_LOG_ID);
*/
```

```
  printf("-----------\n");
  printf("--␣NODE1␣--\n");
  printf("-----------\n");
  nodeLogPrinter(NODE1_LOG_ID);

  printf("-----------\n");
  printf("--␣NODE2␣--\n");
  printf("-----------\n");
  nodeLogPrinter(NODE2_LOG_ID);
}

void hubLogPrinter(ushort hubLoggerId){

  //
  // Poll log data
  //

  can.send(hubLoggerId, 1, (LOG_CMD_CODE<<5));

  //
  // Frame campture loop
  //

  t_CANFrame frame;

  uchar cmdCode;
  uchar paramCode;

  // stored frame data
  bool   valid;
  uchar  port;
  uchar  field;
  uchar  bitNum;
  uchar  dlc;
  ushort id;
  uchar  data;

  bool logActive = true;

  while(logActive){
    frame = can.receive();

    if(frame.id != PC_ID || frame.dlc == 0) continue;

    //extract command and param code
    cmdCode   = extractBits(frame.data[0], 1,3);
    paramCode = extractBits(frame.data[0], 4,5);

    if(cmdCode != LOG_CMD_CODE) continue;

    switch(paramCode){
      case HUB_FINAL_PORT_STATE_CODE :
        printf("port0␣:␣%s\n", portStateStr[extractBits(frame.data[1], 1,2)]);
        printf("port1␣:␣%s\n", portStateStr[extractBits(frame.data[1], 3,2)]);
        printf("port2␣:␣%s\n", portStateStr[extractBits(frame.data[1], 5,2)]);
        printf("port3␣:␣%s\n", portStateStr[extractBits(frame.data[1], 7,2)]);
        printf("\n");
      break;

      case HUB_STORED_FRAME_CODE :
        valid  = extractBits(frame.data[1], 1,1) == 1 ? true : false;
        port   = extractBits(frame.data[1], 2,3);
        field  = extractBits(frame.data[1], 5,4);
        bitNum = extractBits(frame.data[2], 1,8);
        dlc    = extractBits(frame.data[3], 1,4);

        id     = extractBits(frame.data[3], 5,4);
        id     = (id << 8) | extractBits(frame.data[4], 1,8);

        data   = extractBits(frame.data[5], 1,8);

        if(valid){
          // valid frame
          //printf("[ valid ]\n");
          //printf("   port : %s\n", portStr[port]);
          //printf("   %03X [%d] %02X\n", id, dlc, data);

          printf("Ok␣%03X␣[%d]␣%02X␣%s\n", id, dlc, data, portStr[port]);

        }else if(field != 0x00){
          // invalid frame
          //printf("[ invalid ]\n");
          //printf("   %s(%d)\n", fieldStr[field], bitNum);
          //printf("   port : %s\n", portStr[port]);
          //printf("   %03X [%d] %02X\n", id, dlc, data);

          printf(
            "Er␣%03X␣[%d]␣%02X␣%s␣(%s(%d))\n",
            id, dlc, data, portStr[port], fieldStr[field], bitNum
          );

        }else{
          // error frame
          //printf("[ error frame ]\n");
          printf("error␣frame\n");
        }

      break;

      case EOL_CODE :
        logActive = false;
        printf("\n");
      break;
    }
  }
}

void nodeLogPrinter(ushort nodeLoggerId){

  //
  // Poll log data
  //

  can.send(nodeLoggerId, 1, (LOG_CMD_CODE<<5));

  //
  // Frame campture loop
  //

  t_CANFrame frame;

  uchar cmdCode;
  uchar paramCode;
```

```
// stored frame data
typedef enum {tx=0, rx=1} t_role;

t_role role;
uchar  dlc;
ushort id;
uchar  data;
uchar  correctTx;

bool logActive = true;
unsigned int count = 1;

while(logActive){
  frame = can.receive();

  if(frame.id != PC_ID || frame.dlc == 0) continue;

  //extract command and param code
  cmdCode   = extractBits(frame.data[0], 1,3);
  paramCode = extractBits(frame.data[0], 4,5);

  if(cmdCode != LOG_CMD_CODE) continue;

  switch(paramCode){
    case NODE_ERROR_COUNTERS_CODE :
      if(!(frame.data[1] == 255 && frame.data[2] == 255)){
        printf("␣␣␣TEC:%.3d␣REC:%.3d\n", frame.data[1], frame.data[2]);
      }
    break;

    case NODE_STORED_FRAME_CODE :
      role  = extractBits(frame.data[1], 1,1) == 0 ? tx : rx;
      dlc   = extractBits(frame.data[1], 2,3);
      correctTx = extractBits(frame.data[1], 5,1);
      id    = (extractBits(frame.data[1], 6,3) << 8) |
          frame.data[2];
      data = frame.data[3];

      /*
      if(role == tx){
        printf("[ transmited ]\n");
      }else if(role == rx){
        printf("[ received ]\n");
      }*/

      if(dlc != 0){
        printf("%02d:", count);

        if      (role == tx) printf("␣tx␣");
        else if (role == rx) printf("␣rx␣");

        printf("%03X␣[%d]␣%02X", id, dlc, data);
        if(role == tx and correctTx == 0){
          printf("*");
        }
        printf("\n");
        count++;
      }
    break;

    case EOL_CODE :
      logActive = false;
```

```
        printf("\n");
      break;
    }
  }
}


/*---------- End ----------*/

void end(int status){
  printf("Ending...\n");

  // close socket
  if(can.isOpen()){
    can.closeSocket();
  }

  printf("Done!\n");
  exit(status);
}
```

F.4. Mode Changer

F.4.1. ecm.sh

```
#ECM
cansend can0 000#20
```

F.4.2. eem.sh

```
### EWM
cansend can0 000#22

### EEM
cansend can0 000#23
```

Bibliography

A Ademaj, H Sivencrona, G Bauer, and J Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proceedings. 2003 International Conference on Dependable Systems and Networks*, pages 123–132, 2003.

Luís Almeida, P. Pedreiras, and J. A. Fonseca. The FTT-CAN protocol: why and how. *IEEE Transactions on Industrial Electronics*, 49(6):1189–1201, 2002.

E. Armengaud, A. Steininger, and M. Horauer. Towards a Systematic Test for Embedded Automotive Communication Systems. *IEEE Transactions on Industrial Informatics*, 4(3):146–155, August 2008. ISSN 1551-3203. doi: 10.1109/TII.2008.2002704.

Autosar. Autosar - The worldwide automotive standard for E/E systems.

Algirdas Avizienis. Building dependable systems: How to keep up with complexity. *Special Issue of the IEEE 25th Int. Symp. Fault-Tolerant Compuitng. FTCS-25. Pasadena*, pages 4–14, 1995.

Algirdas Avizienis, J.C. Laprie, and B. Randell. Fundamental concepts of dependability. Technical Report 010028, 2001. URL `http://www.cs.ncl.ac.uk/research/pubs/trs/papers/739.pdf`.

Manuel Barranco. *Improving Error Containment and Reliability of Communication Subsystems Based on Controller Area Network (CAN) by Means of Adequate Star Topologies*. PhD thesis, University of the Balearic Islands, 2010.

Manuel Barranco, Julián Proenza, and Luís Almeida. Boosting the Robustness of Controller Area Networks: CANcentrate and ReCANcentrate. *Computer*, 42:66–73, May 2009.

Manuel Barranco, Julián Proenza, and Luís Almeida. Quantitative Comparison of the Error-Containment Capabilities of a Bus and a Star Topology in CAN Networks. *IEEE Transactions on Industrial Electronics*, 53(3):802–803, 2011. doi: 10.1109/TIE.2009.2036642.

Giuseppe Buja, Juan Pimentel, and Alberto Zuccollo. Overcoming Babbling-Idiot Failures in CAN Networks: A Simple and Effective Bus Guardian Solution for the FlexCAN Architecture. *IEEE Transactions on Industrial Informatics*, 3(3):225–233, 2007.

S Cavalieri. Meeting Real-Time Constraints in CAN. *IEEE Transactions on Industrial Informatics*, 1(2):124–135, May 2005.

CiA. CAN physical layer. Technical report, CAN in Automation (CiA). Technical report, Am Weichselgarten 26.

P Ferrari, A Flammini, D Marioli, and A Taroni. A Distributed Instrument for Performance Analysis of Real-Time Ethernet Networks. *IEEE Transactions on Industrial Informatics*, 4 (1):16–25, 2008.

Joachim Ferreira, Luís Almeida, J. A. Fonseca, P. Pedreiras, E. Martins, Guillermo Rodríguez-Navas, J. Rigo, and Julián Proenza. Combining Operational Flexibility and Dependability in FTT-CAN. *IEEE Transactions on Industrial Informatics*, 2(2):95–102, May 2006.

FlexRay Consortium. FlexRay Communications System Preliminary Central Bus Guardian Specification Version 2.0. 9, 2005. URL http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:FlexRay+Communications+System+Preliminary+Central+Bus+Guardian+Specification+Version+2.0.9#0.

L B Fredriksson. CAN for critical embedded automotive networks. *IEEE Micro, Special Issue on Critical Embedded Automotive Networks*, 22(4):28–35, 2002.

M. Fugger, E. Armengaud, and A. Steininger. Safely Stimulating the Clock Synchronization Algorithm in Time-Triggered Systems – A Combined Formal and Experimental Approach. *IEEE Transactions on Industrial Informatics*, 5(2):132–146, May 2009. ISSN 1551-3203. doi: 10.1109/TII.2009.2017526.

David Gessner. Construction of a Hardware Prototype of ReCANcentrate and Implementation of a Media Management Driver for the Nodes of the Prototype. Technical report, University of the Balearic Islands, 2010.

David Gessner, Manuel Barranco, Alberto Ballesteros, and Julián Proenza. Designing sfiCAN: a star-based physical fault injector for CAN. In *16th IEEE International Conference on Emerging Technologies and Factory Automation*, 2011.

F Gil-Castineira, F J Gonzalez-Castano, and Laurent Franck. Extending vehicular CAN fieldbuses with delay-tolerant networks. *IEEE Transactions on Industrial Electronics*, 55(9):3307–3314, 2008.

GNU. GCC, the GNU Compiler Collection. URL http://gcc.gnu.org/.

B Hall, M Paulitsch, K Driscoll, and H Sivencrona. ESCAPE CAN limitations. *SAE Trans. J. Passenger Cars - Electron. Elect. Syst*, 116:422–429, 2008.

Thomas Herpel, Kai-Steffen Hielscher, Ulrich Klehmet, and Reinhard German. Stochastic and deterministic performance evaluation of automotive CAN communication. *Computer Networks*, 53(8):1171–1185, 2009. ISSN 13891286. doi: 10.1016/j.comnet.2009.02.008.

Tobias Hoppe, Stefan Kiltz, and Jana Dittmann. Security threats to automotive CAN networks—Practical examples and selected short-term countermeasures. *Reliability Engineering & System Safety*, 96(1):11–25, 2011. ISSN 09518320. doi: 10.1016/j.ress.2010.06.026.

International Electrotechnical Commission. IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems, 1999.

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

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

ISO. ISO11898-2. Controller Area Network (CAN) - Part 2: High-speed medium access unit, 2003b.

ISO and IEC. International Standard ISO/IEC 14977 - Information technology - Syntactic metalanguage - Extended BNF, 1996.

H Kopetz and G Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1): 112–126, 2003. ISSN 0018-9219. doi: 10.1109/JPROC.2002.805821.

Patrick E. Lanigan, Priya Narasimhan, and Thomas E. Fuhrman. Experiences with a CANoe-based fault injection framework for AUTOSAR. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 569–574. IEEE, June 2010. ISBN 978-1-4244-7500-1. doi: 10.1109/DSN.2010.5544419.

Perfecto Mariño, Francisco Poza, M A Dominguez, and Santiago Otero. Electronics in automotive engineering: A top–down approach for implementing industrial fieldbus technologies in city buses and coaches. *IEEE Transactions on Industrial Electronics*, 56(2):589–600, 2009.

P Martí, Antonio Camacho, Manel Velasco, and M El Mongi Ben Gaid. Runtime Allocation of Optional Control Jobs to a Set of CAN-Based Networked Control Systems. *IEEE Transactions on Industrial Informatics*, 6(4):503–520, 2010.

Microchip Technology. MPLAB C Compiler for PIC24 MCUs and dsPIC DSCs. URL http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010065.

Microchip Technology. dsPICDEM 80-pin Starter Development Board User's Guide, 2006. URL http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en024149.

Microchip Technology. MPLAB IDE User's Guide, 2009. URL http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en019469&part=SW007002.

Microchip Technology. dsPIC30F6011A/6012A/6013A/6014A Data Sheet, 2011. URL http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en024766.

Aurélien Monot, Nicolas Navet, and B Bavoux. Impact of clock drifts on CAN frame response time distributions. *16th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 7–10, 2011.

J K Muppala, R M Fricks, and K S Trivedi. *Techniques for system dependability evaluation*, pages 445–480. Kluwer Academic Publishers, 2000.

Mouaaz Nahas, Michael J Pont, and Michael Short. Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol. *Journal of Systems Architecture*, 55(5-6):344–354, May 2009. ISSN 13837621. doi: 10.1016/j.sysarc.2009.03.004.

Nicolas Navet, Y Song, F Simonot-Lion, and C Wilwert. Trends in Automotive Communication Systems. *Proceedings of the IEEE*, 93(6), 2005.

T Nolte, M Nolin, and H.a. Hansson. Real-Time Server-Based Communication With CAN. *IEEE Transactions on Industrial Informatics*, 1(3):192–201, 2005.

Roman Obermaisser. Reuse of CAN-Based Legacy Applications in Time-Triggered Architectures. *IEEE Transactions on Industrial Informatics*, 2(4):255–268, 2006.

Peak System. PCAN-PCI CAN Interface for PCI, 2012. URL http://www.peak-system.com/Product-Details.49+M52156956e03.0.html?&L=1&tx_commerce_pi1[catUid]=6&tx_commerce_pi1[showUid]=3.

P. Pedreiras, P. Gai, Luís Almeida, and G. C. Buttazzo. FTT-Ethernet: a flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems. *IEEE Transactions on Industrial Informatics*, 1(3):162–172, 2005.

Philips. PCA82C250 / 251 CAN Transceiver, 1996.

Philips. SJA1000 Stand-alone CAN Controller, 2000. URL http://www.nxp.com/documents/data_sheet/SJA1000.pdf.

Piklab. Piklab - IDE for PIC microcontrollers, 2012. URL http://piklab.sourceforge.net/.

Juan Pimentel, Julián Proenza, Luís Almeida, Guillermo Rodríguez-Navas, Manuel Barranco, and Joachim Ferreira. Dependable Automotive CANs. In Nicolas Navet and Françoise Simonot-Lion, editors, *Automotive Embedded Systems Handbook*, chapter 6, pages 1–56. CRC Press, 2008.

W Prodanov, M Valle, and R Buzas. A Controller Area Network Bus Transceiver Behavioral Model for Network Design and Simulation. *IEEE Transactions on Industrial Electronics*, 56 (9):3762–3771, 2009. ISSN 0278-0046. doi: 10.1109/TIE.2009.2025298.

Julián Proenza and José Miro-Julia. MajorCAN: A Modification to the Controller Area Network Protocol to Achieve Atomic Broadcast. *IEEE International Workshop on Group Communication and Computations, Taipei, Taiwan*, page 8, 2000.

Robert Bosch GmbH. CAN Specification Version 2.0, 1991.

Guillermo Rodríguez-Navas. Orthogonal, Fault-Tolerant, and High-Precision Clock Synchronization for the Controller Area Network. *IEEE Transactions on Industrial Informatics*, 4(2): 92–101, 2008.

Guillermo Rodríguez-Navas, Jesús Jiménez, Julián Proenza, and J. Jimenez. An architecture for physical injection of complex fault scenarios in CAN networks. In *Proc. 9th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, volume 2, page 4. Ieee, 2003. ISBN 0-7803-7937-3. doi: 10.1109/ETFA.2003.1248681.

J Rufino, P Veríssimo, G Arroz, Carlos Almeida, L Rodrigues, and A Guillerme. Fault-Tolerant Broadcasts in CAN. In *Digest of Papers 28th Annual Int. Symposium on Fault-Tolerant Computing*, pages 150–159, Washington, DC, USA, 1998. IEEE Computer Society.

Jose Rufino, Carlos Almeida, Paulo Verissimo, and Guilherme Arroz. Enforcing dependability and timeliness in Controller Area Networks. In *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 3755–3760. IEEE, 2006. ISBN 1424401364. doi: 10.1109/IECON.2006.348102.

J-L. Scharbarg, M Boyer, J Ermont, and C Fraboul. TTCAN over mixed CAN/switched Ethernet architecture. In *10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005)*, 2005.

Michael Short and Michael J Pont. Fault-Tolerant Time-Triggered Communication Using CAN. *IEEE Transactions on Industrial Informatics*, 3(2):131–142, May 2007.

J Suwatthikul, R McMurran, and R P Jones. In-vehicle network level fault diagnostics using fuzzy inference systems. *Applied Soft Computing*, 11(4):3709–3719, 2011. ISSN 15684946. doi: 10.1016/j.asoc.2011.02.001.

The Commission of the European Communities. Commission Directive 2002/80/EC of 3 October 2002 - adapting to technical progress Council Directive 70/220/EEC relating to measures to be taken against air pollution by emissions from motor vehicles. *Official Journal of the European Communities*, 291:20–56, 2002.

T. K. Tsai and R.K. K. Iyer. Fault injection techniques and tools. 30(4):75–82, April 1997. ISSN 00189162. doi: 10.1109/2.585157.

United States Environmental Protection Agency. Control of air pollution from new motor vehicles and new motor vehicle engines; modification of federal on-board diagnostic regulations for: light-duty vehicles, light-duty trucks, medium duty passenger vehicles, complete heavy-duty vehicles and engines i. *United States Federal Register*, 70(243):75403–75411, 2005.

University of the Balearic Islands. CANbids - CAN-Based Infrastructure for Dependable Systems, 2011. URL `http://srv.uib.es/project/12`.

Vector. CANoe - ECU development and test, 2012a. URL `http://www.vector.com/vi_canoe_en.html`.

Vector. CANstressD and CANstressDR, 2012b. URL http://www.vector.com/vi_canstress_en.html.

Libor Waszniowski, Jan Krákora, and Zdeněk Hanzálek. Case study on distributed and fault tolerant system modeling based on timed automata. *Journal of Systems and Software*, 82(10): 1678–1694, 2009. ISSN 01641212. doi: 10.1016/j.jss.2009.04.042.

Wikipedia. Field-programmable gate array, 2012a. URL http://en.wikipedia.org/wiki/Field-programmable_gate_array.

Wikipedia. Network socket, 2012b. URL http://en.wikipedia.org/wiki/Network_socket.

Christian Winter. Mejora, integración y verificación experimental de los mecanismos de resolución de inconsistencias del proyecto CANbids. Technical report, University of the Balearic Islands, 2012.

XESS Corporation. XSA-3S1000 Board User Manual, 2005.

Xilinx. ISE WebPACK Design Software, 2012. URL http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm.

Holger Zeltwanger. Controller Area Network — introduced 25 years ago. *CAN Newsletter*, pages 18–20, 2011.

Haibo Zeng, Marco Di Natale, Paolo Giusto, and Alberto Sangiovanni-vincentelli. Stochastic Analysis of CAN-Based Real-Time Automotive Systems. *IEEE Transactions on Industrial Informatics*, 5(4):388–401, 2009.

Haibo Zeng, Marco Di Natale, and Paolo Giusto. Using Statistical Methods to Compute the Probability Distribution of Message Response Time in Controller Area Network. *Industrial*, 6(4):678–691, 2010.