Department of Mathematical Sciences and Informatics. UIB.
Technical report A-03-2005

Design and implementation of CANcentrate: an active star topology for improving fault confinement in CAN networks

Manuel Barranco, Julián Proenza
and Guillermo Rodríguez-Navas
Dpt. Matemàtiques i Informàtica
Universitat de les Illes Balears, Spain

Luís Almeida
DET/IEETA
Universidade de Aveiro, Portugal

October 11, 2005

Abstract

The Controller Area Network (CAN) is a field bus that is nowadays widespread in distributed embedded systems due to its electrical robustness, low price, and deterministic access delay. However, its use in safety-critical applications has been controversial due to dependability limitations, such as those arising from its bus topology. In particular, in a CAN bus there are multiple components such that if any of them is faulty, a general failure of the communication system may happen. In this document, we propose the design of a new active star topology called CANcentrate[1]. Our design solves the limitations indicated above by means of an active hub which prevents error propagation from any of its ports to the others. CANcentrate exhibits improved fault diagnosis and isolation mechanisms with respect to both all communication systems that rely on a CAN bus and all commercially available CAN communication systems based on a hub. Due to the specific characteristics of our hub, CANcentrate is fully compatible with existing CAN controllers, but requires double links. The present document is devoted to report in detail the work we have done regarding CANcentrate. First, the document compares bus and star topologies, analyzes related work and describes the CANcentrate basics, paying special attention to the mechanisms used for detecting faulty ports. Afterwards, the document explains the reintegration policy the hub performs to deal with transient faults and addresses some issues concerning the cabling length in a star topology. Finally it describes the implementation and tests of a prototype of CANcentrate.[2].

[1]The contents of this document have been the subject of a patent filing submitted on the 16th of September of 2004.

[2]This work was partially supported by the European Commission through the Network of Excellence ARTIST2 (IST-004527).

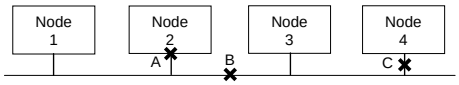

Figure 1: Examples of failures of the communication system

1 Introduction

The Controller Area Network (CAN) protocol is a fieldbus which fulfills the communication requirements of many distributed embedded systems. In particular, CAN provides high reliability 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 factory automation or in-vehicle communication.

Nevertheless, communication systems based on CAN present several specific dependability problems, some of which are caused by the bus topology of this protocol. The main drawback of any protocol using a bus topology is that the structure of the network presents multiple components, i.e. cables, connectors and circuits in nodes, which have direct electrical connections to each other without proper error containment. As a consequence, a fault in the bus interface of one node may generate errors that propagate to the remaining nodes and effectively prevent further communication to take place, leading to a global failure of the communication system. This situation is depicted in case A of Figure 1 in which a fault in the medium access circuitry of node 2, e.g. with the transmitted bits stuck at a fixed value (a dominant value in case of CAN), blocks the communication channel and none of the nodes can communicate with each other. Similar situations can happen with short circuits in the bus transmission medium or the connectors.

Moreover, a bus is shared by all communication paths between every subset of nodes. Consequently, a partition in just one point necessarily leads to a disruption of many communication paths. Even if both partitions can continue operating independently, i.e. the respective nodes can still communicate with each other, the global communication capabilities may have been substantially reduced. This is depicted in case B of Figure 1 in which a partition in the bus mid point blocks any further communication between nodes 1 and 2 with nodes 3 and 4.

Finally, case C shows the situation in which there is a partition in the local connection of node 4 with the bus that does not affect the bus integrity and which leaves the inputs of the node's reception port floating. Consequently, node 4 becomes isolated but the communication among the remaining nodes is unaffected. From the communication system point of view, this is the desired behavior when a fault occurs in one node or node bus interface, because it exhibits the least impact on the communication system itself.

The general framework within which this work has been developed addresses two main objectives. The first objective is to prevent situations in which one single fault in the communication system affects the communication capabilities of more than one node, e.g. cases A and B in Figure 1. In practice, we achieve this objective by using an appropriate topology, namely a star, whose hub enforces the necessary error contain-

ment. As it will be seen later, the hub in a star is a natural location to perform error detection and containment, by blocking error propagation from faulty nodes to the rest to the system at the respective hub port. Furthermore, the links that connect each node to the hub are dedicated and thus, faults occurring on them can be isolated together with only the respective node.

Nevertheless, the star topology still contains one single point of failure, i.e. the hub, which if faulty may lead to a global communication failure. Even so, we consider the star topology to be a good choice because it is easier to improve dependability for the unique single point of failure of the star, e.g. the hub can be placed in a more protected area within the system, than for the multiple components that may cause a communication failure in a bus topology. Moreover, replicated star topologies can be used to tolerate either hub as well as link faults. A broader analysis of the communication system dependability aspects can be found in [1] concerning a comparison between TTP/C and Flexray. The specific issue of network topology is also therein discussed, in which the benefits of a star topology over a bus are clear. Such benefits are also the reason for the shift in Ethernet networks, which occurred through the 90s, from a bus to a star topology.

The second objective of our general framework is to exploit the possibilities the star topology offers to further improve dependability of a CAN network. This dependability improvement can be achieved by taking advantage of an intelligent hub, provided with the necessary capabilities to mitigate the impact on the entire system caused by faults not included in the scope of the first objective, i.e. those which do not make it impossible for the nodes to communicate. For instance, the hub could prevent that any node impersonates another node, thus restricting the failure semantics of the nodes.

This document is devoted to the first objective, only, and it addresses the design of a simplex star topology with one active hub. We call the resultant communication infrastructure CANcentrate. Both the design of a replication scheme for CANcentrate as well as the achievement of the second objective pointed out above will be the subject of future work.

One requirement was imposed from the beginning on the CANcentrate's design: the preservation of all the characteristics of the CAN protocol that are related to dependability. Concerning this requirement, particular care was taken to maintain the frame format and all mechanisms for channel error detection and signalling exactly as they are defined in CAN. As a consequence of this compatibility with the standard CAN specification, off-the-shelf CAN controllers can be used in the nodes of CANcentrate.

In the following section we discuss the properties of existing solutions to improve the dependability properties of CAN, focusing on the advantages of a simplex star topology with respect to simplex and replicated bus topologies. Moreover, existing work on star topologies for CAN is also presented. Section 3 presents the architecture of CANcentrate. Section 4 discusses the mechanisms that the hub of CANcentrate includes in order to diagnose faulty nodes and disconnect them from the network. Special attention is paid to explain the mechanisms the hub provides for diagnosing bit-flipping faults. These mechanisms are deeply explained in Section 5. Moreover, Section 6 makes several considerations about the advantages and disadvantages of the mechanisms for diagnosing bit-flipping faults, as well as Section 7 discusses about the values of the parameters for configuring these mechanisms. Section 8 explains the pol-

icy the hub follows in order to re-enable ports that are blocked due to a previous error confinement decision. Section 9 addresses issues related to the cabling length. A prototype implementation of CANcentrate is described in Section 10. Finally, Section 11 considers future work and Section 12 concludes the document.

2 Problem statement

Despite the good dependability properties exhibited by the CAN protocol, it still presents some drawbacks. In particular, there are multiple components of the network such that a single fault in any of them may make impossible the communication of more than one node, as referred in Section 1. In spite of the existence of several techniques previously proposed in the literature to confine the effect of such faults in CAN systems (see Section 2.2), such techniques are not completely effective in the sense that they still allow single faults of different types and occurring in different physical points to make impossible the communication of more than one node, leading in some cases to a global communication failure.

For the purpose of this document, we will define *severe communication failures*, as those in which the communication capabilities of two or more nodes in the system are permanently affected. Note that severe communication failures include global communication failures, since a global communication failure can be considered as a particular case of a severe communication failure in which the communication capabilities of *all* nodes are affected. Once we have defined severe communication failure, we can rewrite the objective of this work as preventing the existence of multiple components in the network such that a single fault in any of them may cause a severe communication failure.

In order to better understand how CANcentrate achieves this objective, next we describe the fault model and fault assumptions considered in this work.

2.1 Fault model

The fault model gathers the different kinds of faults that may happen in the components of a CAN network and that may cause a severe communication failure:

- *Stuck-at node* fault. It occurs whenever a given node is damaged and issues a constant bit value. Two types of stuck-at fault exist: stuck-at-dominant and stuck-at-recessive faults, depending on whether the bit value issued by the faulty node has dominant or recessive value respectively. Since the physical layer of CAN is equivalent to a logic-AND of every node's contribution, only a stuck-at-dominant fault may cause a severe communication failure (the recessive bit is implemented as the logical '1' value).

- *Shorted medium* fault. This occurs whenever the medium is electrically connected to battery or to ground due to a short-circuit. For obvious reasons, this fault prevents any communication. When fault-tolerant cabling is used, as recommended in CAN [2], such a fault only happens when both wires are shorted to a fixed low impedance electrical source.

- *Medium partition* fault. It occurs whenever the medium is interrupted in such a way that the network is broken into several subnetworks, which are called *network partitions*. Therefore, any two nodes which are each one in a different partition can no longer communicate with each other. Moreover, signal reflections at the open extremities may cause channel errors that prevent nodes in the same partition from communicating properly [2].

- *Bit-flipping* fault. This occurs whenever a component of the network (either a node or a medium) exhibits a *fail-uncontrolled* behavior and starts sending erroneous and random bits with no restrictions in the value domain. In this case, even if a node is trying to send a correct bit stream, this is destroyed by the dominant bits of the bit-flipping stream. Some potential causes of this fault are: a damaged node that sends random bit values; a bad welding on the medium connector that generates random bit values, etc.

- *Babbling-idiot* fault. It occurs whenever a node sends messages that are erroneous in the time domain, then consuming more resources that it really needs and starving the other nodes of the appropriate resources for communicating [3]. For instance, this type of faults may happen as a consequence of a software fault in a node that results in an infinite loop that sends messages continuously. To deal with this kind of faults requires knowledge about the scheduling of the messages, which depends on the application executed at nodes. However, in the context of this present work we focus on a solution independent of the application. Therefore, although a babbling-idiot fault may provoke a severe failure of the communication system, we postpone its treatment for future work. Note that, for instance, it is suitable to include in the hub a bus-guardian, similar to the one proposed in [3], for dealing with faults in the time domain.

Beyond the types of faults considered in this work as state above, we make no assumptions on the frequency and duration of errors that may occur in any port or group of ports. The only assumption made, since hub replication is not considered, is that the hub itself will not fail.

2.2 Potential solutions

Some of the faults presented above can be confined in bus-based CAN systems, up to a certain extent, using techniques that are already known. These techniques rely on the use of replicated transmission media as well as on the use of *bus guardians*. However, due to the characteristics that are inherent to the bus topology, said techniques do not prevent the existence of multiple components such that a single fault in any of them may cause a severe failure of the communication system.

The use of replicated transmission media generally allows nodes to detect a faulty medium by comparing the values received from each of the replicas [4] [5]. In this way, nodes can disable the faulty medium so that the communication system can still provide a correct service. Nevertheless, this solution does not prevent a faulty node (e.g. a stuck-at-dominant node) from causing a failure of the whole communication system by sending erroneous information to all replicated media. Moreover, this solution has

a more subtle weakness; regardless of the routing of the replicated media, they have to come together near every node of the system. This spatial proximity is a potential cause of *common-mode* failures of the replicated media system. For instance, a smash near any node may cause a partition of all media, thus leading to a severe communication failure [6].

A different architecture, which is used in RedCAN [7], connects nodes by means of a special ring in which one of its sectors is redundant and left inactive, so that the resultant topology is a bus. The ring can be reconfigured by shutting down one or more adjacent sectors and activating the redundant one. This is carried out when a stuck-at fault of the medium occurs in one or some active and adjacent sectors or if the node or nodes that connect adjacent sectors crash. The main disadvantage of RedCAN is that it only deals with faults occurring in adjacent sectors or contiguous nodes. Moreover, this solution increases the complexity of the network nodes, thus increasing their probability of failure, and uses specific RedCAN hardware.

On the other hand, the use of bus guardians prevents the propagation of errors from any node, enforcing a fail-silent behavior [3] of the nodes. A bus guardian is a device which supervises the output of a node to its bus interface in order to detect incorrect behavior. In this way, a faulty node, such as a stuck-at-dominant node or a bit-flipping node, can be easily detected and isolated from the rest of the system. Nevertheless, the weak point of this approach is that fault independence between a node and its corresponding bus guardian is not completely ensured due to potential common-mode failures. These can be caused either by spatial proximity of a node and its bus guardian, or by sharing resources or procedures, e.g. power supply, system clock, clock synchronization algorithm. Moreover, the use of bus guardians is useless for containing error propagation from a faulty medium.

From the above discussion we can conclude that even though the use of replicated media as well as bus guardians significantly improves the dependability characteristics of CAN, these mechanisms –even if they are used together in the same system– still allow multiple components to cause severe failures of the communication system, and therefore they do not fulfill the aim of this work. Thus, alternative solutions have been researched, namely those based on a star topology.

In a star topology, each node is connected to a central element, the *hub*, by its own *link*. This provides a natural way of enforcing confinement of faulty transmission media by isolating the respective links at the respective hub ports. Furthermore, the hub has a privileged view of the system, as it simultaneously knows the contribution from every node and thus, it can play the role of bus guardian of each node. In this way, spatial proximity between a node and its corresponding bus guardian is avoided. Moreover, the links of a star topology only come into spatial proximity at the center of the star.

It is obvious that the main drawback of a star topology is that the hub represents a single point of failure. In addition, the complexity of the error-detection and fault-treatment mechanisms included in the hub implies that its probability of failure is higher than it could be for simpler components, such as a bus guardian. Nevertheless, different strategies can be adopted in order to face this problem. For instance, the hub reliability can be increased by placing it in a well-protected zone inside the physical system or by investing in its quality or even by adopting a replicated star topology.

However, as stated in Section 1, this problem is out of the scope of this document and will be addressed in the future.

2.3 Available star topologies for CAN

Even though the star topology provides a good basis to improve the dependability of the communication system, the adoption of such a topology is not enough. Additional mechanisms should be included in the hub in order to detect and isolate faulty components and achieve the behavior of what we call the *ideal star*, that is a star-based system in which the hub includes all the mechanisms which are necessary to ensure containment of all errors which may cause a severe communication failure. Some star topologies for CAN can be found in the literature [8] [9] [10] [11]. Although these solutions provide some mechanisms to deal with faulty components, any of them either do not address severe communication failures [8] [9] [11] or deal only with stuck-at-dominant faults [10]. Thus, none of them includes the mechanisms which are required to prevent all the faults described in Section 2.1 from generating severe communication failures.

In [8] [9], a passive star network topology for CAN is presented. This solution relies on the use of a central element, called the *star coupler*, which acts as a concentrator where all the incoming signals are coupled. The result of this coupling is then broadcast to the nodes. In what concerns dependability, the only advantage this solution presents when compared to a bus-based CAN system is the reduction of the spatial proximity problem between different links. This reduction is possible since the links only come into physical proximity at the center of the star. However, this kind of star couplers shows some technical drawbacks that discourage its use from the practical point of view. On the one hand, large coupling loses impose strong limitations on the star radius (5 to 10 meters) and hence force nodes to communicate at low bit rates. On the other hand, the coupling of the incoming signals causes some electrical problems, such as resonances, harmonics or disturbances, which require the use of complex hardware solutions.

Other star topologies for CAN are presented in in [9] and [10]. These topologies rely on an active star coupler, which receives the incoming signals from the nodes bit by bit, implements a logical AND function, and retransmits the result to all nodes. By means of this coupling these stars overcome the technical problems of passive star topologies.

Concerning the active star presented in [9], each node is connected to the star coupler by an independent link constituted by two optical paths. In the star coupler, there is a transceiver for each link (a so-called *node-coupler transceiver*) as well as an internal CAN bus with few centimeters of length. In a first stage, signals from each link are received by the respective node-coupler transceiver and transmitted without any processing into the internal CAN bus. In a second stage, the resultant signals from the internal CAN bus are received by each node-coupler transceiver and retransmitted towards the corresponding node. Therefore, from the dependability point-of-view it just reduces the spatial proximity problem between different links.

Regarding the star topologies available in [10], they present fault-diagnosis and fault-isolation mechanisms for detecting and isolating ports that present a permanently

stuck-at-dominant fault. Since there are not technical information available describing the mechanisms included in their star couplers, it is difficult to evaluate the performance of their error detection mechanisms. However, they only use one link to connect each node to the coupler; hence, it seems that they will present a significant amount of time for diagnosing (they have to deal with the fact that nodes' contributions are not separated in the space). Beside this possible disadvantage, it is also worth noting that they do not lead either with stuck-at-recessive faults or with bit-flipping faults.

In [11] other active star topology is suggested for CAN, namely StarCAN. The main goal of this solution does not address network dependability, but network performance. In particular, StarCAN achieves either an extension more than 10 times longer than a typical CAN network or a bit rate 10 times higher than a typical CAN network. Nevertheless, in order to fulfill this goal, StarCAN sacrifices one of the most important characteristics of CAN, the in-bit response [2]. This decision has an enormous impact on the dependability properties of the network. On the one hand, the lack of in-bit response jeopardizes the so-called *data consistency* of the CAN network, since inconsistency scenarios [12] [13] turn out to be more likely. On the other hand, despite the use of some CAN mechanisms, e.g. arbitration and error signaling, off-the-shelf CAN controllers cannot be used, raising issues about the practicality of the solution.

Therefore, none of the star topologies for CAN that have been studied fulfills our goal of preventing the existence of multiple components such that a single fault in any of them may cause a severe failure of the communication system. In fact, almost all the studied star topologies do not even address this kind of failures, behaving as a bus with enhanced resilience to spatial proximity faults.

This justifies the design of a new star topology for CAN, with special focus on achieving the goal specified above, i.e. obtaining an ideal star.

3 Design of CANcentrate

In Section 2, it has been shown that neither bus topologies nor existing star topologies do fulfill the strong dependability requirements of many safety-critical systems, since they allow a single fault in anyone of multiple network components to cause a severe failure of the communication system. Due to this, we have proposed a new star topology, called CANcentrate, which does not exhibit this drawback.

In the CANcentrate architecture, each node is connected through a dedicated link to a different port of a central hub. As explained later on in this section, a node together with its dedicated link constitute an error-containment region. From the hub perspective, a permanent fault within a given error-containment region manifests as a permanently faulty port. Therefore, the hub can prevent a severe failure by detecting and isolating any permanently faulty port.

This section is devoted to describing the CANcentrate architecture, paying special attention to the internal structure of the hub. The fault diagnosis mechanisms that the hub includes are thoroughly discussed in Sections 4 and 5.

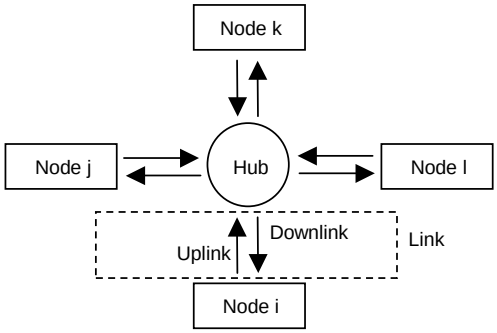

Figure 2: Architecture of CANcentrate

3.1 Design rationale

Probably the most important characteristic of CAN is the dominant/recessive transmission. This property guarantees that whenever one of the nodes transmits a bit with the dominant value, this value is received by all the nodes in the network. In contrast, a bit with the recessive value 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 (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 which is transmitting (the so-called *leading transmitter*). This bit synchronization limits the maximum bit rate of the network, but at the same time allows definition of a number of additional mechanisms (e.g. bit-wise arbitration, ACK bit, error frame), which significantly improve the dependability and real-time properties of CAN networks [2]. Due to the relevance of these mechanisms, it is very important to preserve them even if a star topology is used instead of a bus.

Assuming that the typical assignment is done, i.e. logical '1' to recessive value and logical '0' to dominant value, in order to keep the dominant/recessive transmission, the hub must implement a logical AND function of the contributions received from every node. Moreover, and in order to preserve the in-bit response, this logical AND must be performed within a fraction of one bit time, despite the extra delay which the internal circuitry of the hub may cause.

Furthermore, the hub must include some mechanisms in order to identify permanently faulty ports. These mechanisms, which are thoroughly described later on, require the hub to be able to discriminate the signal that any node transmits from the signal resulting of the logical AND that the hub broadcasts to the nodes. A simple way to separate both signals is through the use of two different cables for each link that connects each node to the hub. Figure 2 shows the corresponding architecture in which there are only point-to-point unidirectional electrical connections.

The cable that carries the signal from a node to the hub is called the *uplink*, whereas the cable that carries back the resulting signal from the hub to the node is called the

downlink. Each cable is of the same type as the twisted copper wiring used for implementing typical CAN buses, which have a good resilience against electromagnetic interferences. Moreover, each cable is terminated at both ends, the node and the hub.

Therefore, two transceivers are required at the end of each link; one for the uplink and another one for the downlink. Figure 3 illustrates how the transceivers are connected at the end of the node. Note that the *receive data output* pin (RxD) of the uplink transceiver is left open whereas the *transmit data input* pin (TxD) of the downlink transceiver is forced to have a recessive level (the logical '1' value). It is important to remark that the CANcentrate architecture can be implemented with both off-the-self CAN controllers and off-the-shelf CAN transceivers. This makes the solution practical and relatively low-cost. Nevertheless, the hub requires some specifically designed hardware, as discussed next.

3.2 Internal structure of the hub

The hub plays a crucial role in the star topology since it performs two fundamental functions. On the one hand, it implements the logical AND function which allows preservation of the dominant/recessive transmission of CAN as well as the rest of dependability mechanisms of CAN. On the other hand, it includes a number of fault-treatment mechanisms in order to identify and isolate permanently faulty ports.

The hub is divided into three modules, namely the *Input/Output Module*, the *Coupler Module*, and the *Fault-Treatment Module*. The structure and interconnections of these modules are depicted in Figure 4.

The Input/Output Module is made up of a number of transceivers; two for each link. As Figure 4 shows, one transceiver is assigned to every uplink in order to convert the physical signal received from each node into a logical value that the hub can process, $B_{1..n}$. Moreover, one transceiver is assigned to every downlink so that the logical output of the hub, the resultant coupled signal B_0, is converted into a physical signal that is broadcast to every node.

The Coupler Module is made up of an AND gate, which performs the coupling of the uplink signals, and a number of OR gates, one per link, which allow the hub to disable the contribution to the global AND from a specific uplink. In particular, the contributions that are disabled are those from ports that have been diagnosed as being permanently faulty. Since the AND gate replaces the wired-AND functionality of the CAN bus, this means that the output of the Coupler Module, B_0, would be the same of a CAN bus where there were no permanently faulty component. The frame that results from coupling the frames from the enabled ports (and, therefore, that would result in a CAN bus without faulty nodes) is called the *resultant frame* hereafter.

This configuration causes an additional delay on the signal that the nodes receive. For the bit synchronization of the nodes, this additional delay has to be taken into account as a part of the *propagation time* [2]. For all purposes it is similar to the extra delay caused by an equivalently longer cable in a bus system.

Note that the output of the AND gate is connected to each and every one of the downlink transceivers. In this way, the output of the hub (i.e. the coupled signal) does not interfere with the signals received through the uplinks, so the contribution of every

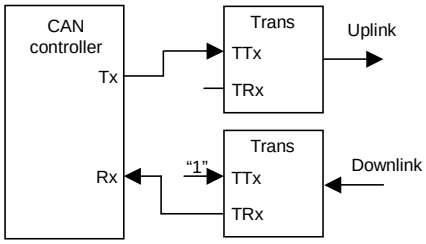

Figure 3: Configuration of the transceivers to connect a node to its link

node remains separated and further mechanisms can be applied in order to identify a permanently faulty port.

The main purpose of the Fault-Treatment Module is to detect permanently faulty ports and to isolate them from the system, so they cannot cause severe communication failures. This function is carried out by performing both *fault diagnosis*, which aims at finding out the permanently faulty port, together with *fault passivation*, which aims at isolating the permanently faulty port from the system.

The fault-diagnosis mechanisms of the Fault-Treatment Module require the identification of the contributions from every uplink as well as knowledge of the *current state* of the *resultant frame*. This current state represents what all nodes are supposed to have received from the hub until this moment, and therefore permits to forecast which should be the proper contribution of each node for the following bit. Fortunately, the use of two cables for each link keeps the contribution from each link separated, and therefore the physical location of the faults can be more easily established. However, this architecture does not allow the hub to discriminate between faults that are caused by a faulty transmission medium and faults that are caused by a faulty node. Therefore, as stated before, from the point of view of the hub, either a permanently faulty medium or a permanently faulty node are viewed as a permanently *faulty port*. The mechanisms that have been devised in order to diagnose a permanently faulty port are thoroughly described in Sections 4 and 5.

The current state of the *resultant frame* describes the meaning of the bit of the *resultant frame* that is currently being broadcast to all ports. The knowledge of such current state requires to keep the synchronization of the hub with the *resultant frame* at bit level as well as at frame level. The synchronization at bit level allows the hub to agree with all the nodes about the beginning and the end of each bit time in order to perform a correct sampling of the bit value; whereas the synchronization at frame level allows the hub to agree with all the nodes about the location of each bit inside the frame.

On the one hand, the *Physical Layer Module* uses the typical CAN synchronization mechanisms [2] for allowing the hub to synchronize with the bit stream at bit level and this generates the reception and the transmission clocks (clkR and clkT respectively in Figure 4). As in a normal CAN node, the reception clock indicates to the hub the instant of time at which the input signal from the medium (the coupled signal and each

port contribution in the case of the hub) must be sampled; whereas the transmission clock indicates the instant of time at which a transmission bit value could be issued to the medium if the hub needs to transmit a bit.

On the other hand, the *Rx_CAN Module* observes the bit stream at the coupled signal in order to achieve the synchronization at frame level. As a result of this synchronization, it generates a set of signals, C, that together with B_0 describes the current state of the *resultant frame* that will be used for fault diagnosis by the Enabling/Disabling units (Ena/Dis in Figure 4).

It is very important to note that an error detected by any CAN node (of CANcentrate or of a CAN bus) may arise due to a previous desynchronization at frame level that leads to an inconsistent vision about the current state of the *resultant frame*. Therefore, since a node cannot know the reason that provoked the error, it must assume that it has lost the synchronization at frame level. In such situations, the CAN standard specifies that any normally operating (*error-active*) node must globalize an error condition by means of an active *error flag* and must cooperatively transmit an *error delimiter* with the other nodes, in order to enforce the agreement among all the nodes about the current state of the *resultant frame* [2]. More details about the error detection and error globalization in CAN are explained in Section 3.3.

As stated above, the fault-diagnosis mechanisms of the Fault-Treatment Module require knowledge of the current state of the *resultant frame*. Thus, the hub must keep the synchronization at frame level with the other nodes in spite of the presence of errors. To achieve this, when the hub detects an error in the resultant frame, it forces a re-synchronization by means of an active error flag transmission. This is performed by using the *Error Flag Generator Module* within the Fault-Treatment Module (*errorFlagGenerator* in Figure 4). This module receives the order of globalizing error conditions detected in the *resultant frame* from the Rx_CAN Module and transmits an active error flag through a dedicated contribution, *hubTx*, driven into the global AND.

The specific error conditions that compel the hub to globalize an error condition are thoroughly discussed in Section 3.3. It is worth noting that the error globalization mechanism provided by the Error Flag Generator allows the hub to abort the transmission of a frame at any moment. Therefore, it allows inclusion of further fault-treatment mechanisms, which are not included in the CAN protocol, e.g. to abort the transmission of a forged message due to a masquerading fault. Nevertheless, these issues will be addressed in future work.

The ultimate *fault diagnosis* and *fault passivation* are carried out by the *Enabling / Disabling* units (*Ena/Dis* in Figure 4). Each one of these units uses the set of signals C, which are better described in Section 3.4, and the resultant coupled signal, B_0, to know the current state of the *resultant frame*. This information is used together with the contribution from its corresponding port (either B_1, B_2, etc.) in order to diagnose whether its port is permanently faulty or not, as described in Sections 4 and 5.

Whenever a given Enabling/Disabling Unit diagnoses its corresponding hub port as being permanently faulty, it removes the contribution of this port from the system by issuing a logical '1' to the corresponding *Enabling/Disabling* signal, $ED_{1..n}$, which is connected to the OR gate that corresponds to the faulty port (see Figure 4). This effectively removes the contribution of this port to the global AND, being equivalent to disconnecting the link, and the corresponding node, from the hub. In general, this

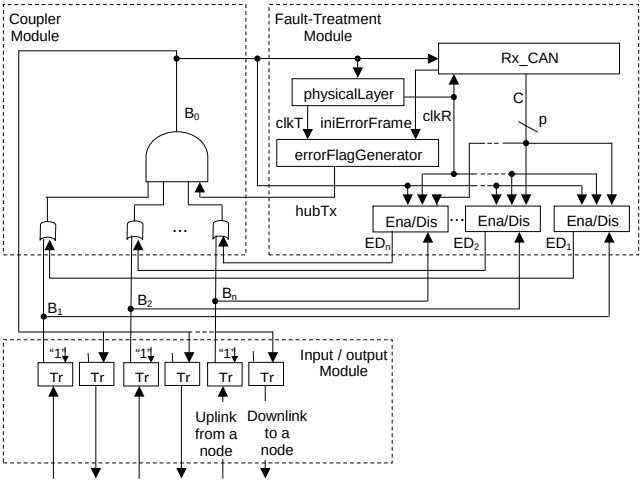

Figure 4: Internal structure of the hub

mechanism is similar to the one proposed in [4] to manage, locally in each node, the media redundancy in a replicated bus topology.

3.3 Error detection and globalization for synchronization

As described in Section 3.2, the hub detects and globalizes some errors in order to maintain the synchronization with the other nodes. In order to understand these functions it is important to remember some error management issues of the CAN protocol.

Any CAN node is able to detect five different error types [2]: *stuff error*, *format error*, *bit error*, *CRC error* and *ACK error*. Such errors are detected by means of several error detection mechanisms that check the correctness of the frame that the node transmits or receives. These mechanisms, also specified in [2], respectively are: *stuff rule check*, *frame check*, *monitoring*, *15 bit cyclic redundancy check* and *acknowledge check*. Next, a summary about how such mechanisms are used for detecting the different error types is presented.

- Stuff Error. Both, the transmitter and the receivers perform a *stuff rule check* to test if the stream sent by the transmitter fulfils the stuff rule. This rule basically specifies that the transmitter must insert a complementary bit, called *stuff bit*, whenever it has already transmitted five consecutive bits of the same polarity (including stuff bits).

- Format error. Both, the transmitter and the receivers perform a *frame check* to test if the frame obeys the format rules. These rules define the characteristics of each field of the frame: order within the frame, length and allowed bit values.

- Bit error. Both, the transmitter and the receivers [3] perform a *monitoring* of the *resultant frame* in order to check that whenever they transmit a dominant bit, the resultant bit in the channel is a dominant bit. Moreover, the transmitter also checks that whenever it transmits a recessive bit, the resultant bit is recessive except in the *arbitration field* [2] and in the ACK slot.

- CRC error. The transmitter calculates a 15 bit cyclic redundancy code (CRC) based on the bits of the frame that has already transmitted and, next, it transmits such CRC in the last but two field of the frame. The receivers also calculate the CRC and check if it matches with the CRC received (i.e. they perform a *15-bit cyclic redundancy check*).

- ACK error. The receivers send a dominant bit value at a specific field called *ACK slot* if they wish to acknowledge the correct reception of the frame. The transmitter performs an *ACK check* to test if there is a dominant bit value in the ACK slot and thus, to detect if at least one node has acknowledged the frame.

Depending on the number of the errors detected up to a given instant of time, a node is in the *error-active* state, in the *error-passive* state or in the *bus-off* state. Thus, we consider that a node is error-active, error-passive or bus-off respectively. When the number of errors an *operational node* (i.e. a node involved in bus activities) has detected is less than a specific threshold, it is *error-active*; otherwise, it is *error-passive*. A node is bus-off after an initialization operation (p.e. when the node is powered up) or after detecting a number of errors that exceeds a given threshold which indicates that the node must isolate itself from the system (i.e. a bus-off node is not involved in bus activities).

Whenever a node detects an error, it signals it by transmitting an *error flag*. In the case of an error-active node, such error flag is called *active error flag* and is constituted by 6 consecutive dominant bits, whereas the error flag of an error-passive node, called *passive error flag*, is constituted by 6 consecutive recessive bits. An active error flag always violates the stuff rule and provokes all the nodes to detect an error and to signal it too. In such a way, the error is globalized and the frame that was being transmitted is rejected by all the nodes, i.e. we say that an *error globalization* occurs. Nevertheless, a passive error flag does not always force the other nodes to detect an error [2] and thus, no globalization is ensured when an error-passive node signals an error. This is an important issue since *data consistency* (which actually means that a frame must be accepted by all nodes or by none of them [2]) can only be ensured if all nodes which detect an error are able to globalize it.

In case an *error flag* provokes a globalization, after transmitting their own error flags, all the nodes cooperatively transmit an *error delimiter* which is constituted by a minimum of 8 recessive bits. The frame constituted by all the overlapped error flags, followed by the cooperative error delimiter is called *error frame*. The error delimiter is built as follows. After transmitting their own error flags, the nodes continue transmitting recessive bits until they monitor a pattern of 8 consecutive recessive bits in the

[3] A receiver can transmit a dominant bit during the signaling of an error flag as well as at the ACK slot to acknowledge the frame (for more information about the CAN frame format, please see [2])

channel. Note that since dominant values always overwrite recessive values, all nodes will detect such bit pattern at the same time (as long as any node is transmitting its active error flag, all nodes monitor dominant values) and then, they can consider that are re-synchronized at frame level again after detecting it.

However, if a passive error flag does not provoke an error globalization, the error-passive node continues transmitting recessive bits in order to detect the pattern of 8 consecutive recessive bits within the *resultant frame*. In this case, the detection of such bit sequence not only can imply that an error delimiter ended, but that a transmitter node ended a frame. This is because the last field of a non-error frame, the *End-Of-Frame* (EOF), is constituted by 8 consecutive recessive bits. Note that the *resultant frame* reaches the *intermission field* [2] after both, the transmission of the EOF and the transmission of an error delimiter. Therefore, when the error-passive node detects the pattern of 8 consecutive recessive bits, it can assume that considers again the same state of the *resultant frame* the other nodes do, i.e. the intermission field, achieving the synchronization at frame level.

Note that, since only an active error flag ensures the globalization of any error, the hub must behave as an error-active node in what concerns error signaling. In this way, the hub enforces re-synchronization at frame level despite the occurrence of an error. However, as the hub is not the original transmitter of the message it can only detect in the *resultant frame* those errors that a receiving CAN node would detect. More specifically, the hub includes a subset of the CAN error detection mechanisms in order to detect the: stuff error, format error and CRC error. Notice that the bit error is not included in this list. The hub could also detect this kind of error by just monitoring the downlinks. However this monitoring is not required because any error in the downlink actually desynchronizes the corresponding node at frame level, which implies that, sooner or later, this node will cause a stuff error, a format error or a CRC error in the *resultant frame*.

As depicted in Figure 4, this error detection for ensuring the synchronization is performed by the Rx_CAN Module, which orders the Error Flag Generator to globalize an error by means of the *iniErrorFrame* signal.

3.4 State signals for the Enabling/Disabling units

As indicated in Section 3.2, the Rx_CAN Module, besides being responsible for ensuring the synchronization at frame level, also provides the Enabling/Disabling units with a set of signals, C, that gathers all the information that, together with B_0, is needed to know the meaning of the bit that is currently broadcast to all nodes, i.e. the current state of the *resultant frame* (see Figure 4).

Next, these signals are explained (see Figure 5). Nevertheless, the reason why these are the signals required by the Enabling/Disabling units for describing the current state of the *resultant frame* will be more easily understood later on in Sections 4 and 5.

First, the meaning of a bit takes into account its type, i.e. whether the bit is a stuff bit or not. The *bitStuffWaited* and the *valueBitStuff* signals indicate whether the bit that is currently observed is a stuff bit or not and, in case it is a stuff bit, the expected correct bit value according to the stuff rule, respectively.

Second, the type of frame and the specific field of the frame in which the bit is located also determines the meaning of the bit. CAN specifies five kinds of frames: data frames (which carry data), remote frames (which do not carry data, but that request for the transmission of an specific data frame), active error frames, passive error frames and overload frames (which are aimed at achieving an extra delay between two different data or remote frames). In addition, data frames an remote frames are always separated from the preceding frame (regardless the type of the preceding frame) by means of a sequence of recessive bits called *interframe space*. The *interframe space* is constituted by two fields: the *intermission* field and the *idle* field. The information about both the kind of frame and the specific field is codified by means of two signals: the *frameField* and the *lastBitEOF* signals. On the one hand, the *frameField* is a vector (a signal of k bits) that specifies the kind of frame the bit belongs to as well as the specific field within the frame in which the bit is located. In particular, when a *globally detectable error* (i.e. an error in the *resultant frame* which is detectable by using the error detection mechanisms of a standard CAN receiver) is detected, the hub will signal it and thus, such signal changes to indicate that an active error flag is being transmitted. On the other hand, in spite of the fact that the last frame field, called *End of frame* (EOF), is constituted by 7 consecutive recessive bits, a dominant bit detected at the last bit of the EOF does not provoke a format error, but an overload situation [2]. Therefore, the additional signal, *lastBitEOF*, is needed to indicate whether the bit currently observed is the last bit of the *End of frame* field (EOF) or not.

Finally, as will be explained later on in Sections 5.1, 5.2, 5.4 and 5.5, some error detection mechanisms included in the Enabling/Disabling units also need to know whether the frame sent through the hub has passed the CRC checking performed by the Rx_CAN Module or not. This information is provided by means of the *CRCPassed* signal.

4 Fault-diagnosis mechanisms

The main objective of the fault-treatment mechanisms the hub includes is to detect and isolate permanently faulty ports. However, two additional requirements are imposed: to reduce the probability of isolating non-faulty ports and to reduce the probability of data inconsistency.

The fault-passivation phase of fault treatment is performed by the ED signals as indicated in Section 3.2. Current section is devoted to discussing how the fault-diagnosis phase is carried out by the hub.

4.1 Fault-diagnosis rationale

The failures that are diagnosed by the Fault-Treatment Module cover all the failure model presented in Section 2.1: stuck-at-dominant, stuck-at-recessive and bit-flipping faults. Notice that a shorted medium manifests itself at the hub port as a stuck-at fault, whereas a medium partition can manifest itself as either a stuck-at fault or a bit-flipping fault. That is because a shorted medium is stuck to a constant voltage level (battery or ground), whereas a medium partition can either force the medium to be stuck at a

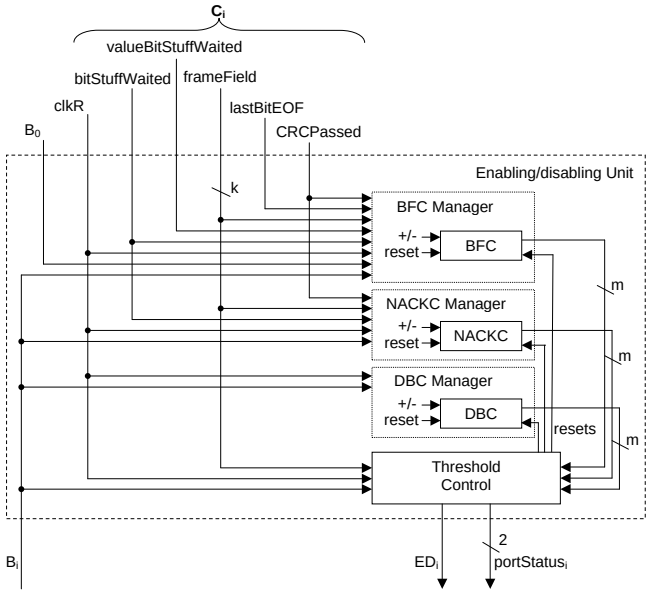

Figure 5: Internals of the Enabling/Disabling Unit

constant voltage level, or cause channel errors due to signal reflections at the open extremities of the cable.

The fault-diagnosis mechanisms are essentially included in the Enabling/Disabling units that operate separately on each port. As will be explained in Section 6.2, such separately fault-diagnosis evaluation on each port allows the hub to achieve a high accuracy when diagnosing a port as permanently faulty and, then, to reduce the probability of isolating non-faulty ports. The internals of each one of these Enabling/Disabling units are shown in the Figure 5.

On the one hand, each Enabling/Disabling Unit has a dedicated *event counter* and an associated *manager module* for each type of fault that must be detected: the *Dominant Bit Counter* (DBC) and the *DBC Manager Module* for stuck-at-dominant faults; the *Non-Acknowledge Counter* (NACKC) and the *NACKC Manager Module* for the stuck-at-recessive; and the *Bit-Flipping Counter* (BFC) and the *BFC Manager Module* for the bit-flipping faults. Each management module basically analyzes the coupled signal, B_0, the port contribution B_i and the control signals C, from Rx_CAN, in order to decide how to increase or decrease its corresponding event counter.

On the other hand, the Enabling/disabling Unit has a *Threshold Control Module* that is aimed at declaring the port as permanently faulty when it corresponds as well as to isolate its contribution. The Threshold Control Module takes into account the value registered by each event counter and is programmed with a specific threshold for each of them: the *Dominant Bit Threshold* (DBT), the *Non-Acknowledge Threshold* (NACKT) and the *Bit-Flipping Threshold* (BFT). Whenever any of the event counters

exceeds its corresponding threshold, the Threshold Control Module isolates the port contribution by setting the corresponding ED_i signal to '1' and resets all the event counters and their managers. However, in order to increase the tolerance to transient errors, the Threshold Control Module may use a specific reintegration policy to re-enable the port contribution and to allow the operation of all managers again, after a given period of *inactivity* is observed at the port. This reintegration policy is explained later on in Section 8.

Concerning the additional requirement of reducing the probability of data inconsistency, the following issues have to be taken into account. It is worth noting that since data consistency is not ensured whenever a CAN node reaches the error-passive state (an error-passive node cannot always force the globalization of an error, see Section 3.3), the hub must not accept error-passive nodes. Therefore, the fault-diagnosis mechanisms of the hub are devised in order to consider the behavior of such nodes as incorrect. This will eventually lead to the isolation of the hub's port corresponding to an error-passive node. Moreover, as will be explained later on in Section 7, the hub can be configured in order to reduce as much as possible the probability of any node achieving the error-passive state and then, to even reduce the probability of data inconsistency of standard CAN network.

4.2 Stuck-at-dominant faults

In order to detect stuck-at-dominant faults, each DBC counts the number of consecutive dominant bits that are received from its corresponding port. The DBC is increased in one unit each time its corresponding manager observes a dominant bit value on the uplink, and it is reset as soon as its manager observes a recessive value.

The DBC value is compared to the *Dominant Bit Threshold* (DBT) and, whenever a DBC exceeds the DBT, the Threshold Control Module isolates the corresponding port.

The DBT is configured in order to maximize the chances to differentiate between situations in which a stuck-at-dominant fault really exists and situations in which the channel is occupied by many consecutive dominant bits, although there is not a stuck-at-dominant fault. The maximum number of allowed consecutive dominant bits before diagnosing a stuck-at-dominant fault takes into account two different contributions:

$$\text{DBT} = (T_{\textit{stuff}} + 1) + N * T_{\textit{errorFlag}}$$

The first term, $T_{\textit{stuff}} + 1$, specifies the minimum number of consecutive dominant bits that violates the stuffing rule in a CAN network (6 bits). This term includes the maximum number of consecutive dominant bits allowed in CAN, $T_{\textit{stuff}}$, plus the additional dominant bit needed for violating the stuff rule. Whenever the stuff rule is violated, it is expected that all nodes start to send an active error flag. Nevertheless, it is possible that a node detects a second error during its own error flag and starts sending again an active error flag, thereby prolonging the sequence of consecutive dominant bits. In the worst case, a node will see this second error in the last bit of its first error flag, and will send a consecutive active error flag. The second term, $N * T_{\textit{errorFlag}}$, is intended to cover these situations. It specifies the maximum number of consecutive dominant bits that are considered as overlapped or consecutive active error flags. In

other words, it indicates the maximum amount of time a node is allowed to transmit overlapped or consecutive error flags, measured in number of bits.

Note that for $N = 2$ the threshold coincides with the one proposed in [4]. In that case, the threshold can be exceeded if two additional errors occur in the error flag that follows a violation of the stuff rule, leading to an erroneous diagnosis of a stuck-at-dominant fault. Using a higher value of N reduces the probability of performing an erroneous stuck-at-dominant diagnosis.

The value of N can be configured depending on the application. For instance, in a hazarding environment, we may consider that $N = 4$ is tolerant enough and does not imply a significant loss of reactivity in diagnosing stuck-at-dominant faults.

4.3 Stuck-at-recessive faults

Due to the AND function that the hub implements, a port suffering a stuck-at-recessive fault does not interfere the communication among the rest of the nodes in the star. Therefore, this kind of fault does not generate a severe failure of the communication system. Nevertheless, detection of this kind of faults may still be useful in order to implement additional fault-tolerance mechanisms at higher levels of the system architecture, for example to detect a crashed or absent node.

The detection of stuck-at-recessive faults poses an additional difficulty because a CAN node may be without transmitting, which actually means sending recessive values, for a long time. Therefore, it would be theoretically impossible to differentiate between a stuck-at recessive node and an operational but non-transmitting node. Nevertheless, the CAN protocol specifies that every CAN controller must transmit a dominant bit in the ACK slot of every frame that is correctly received [2]. Therefore, the absence of this bit can be used to detect stuck-at-recessive ports.

For each port, such detection is carried out by a specific *Non-Acknowledge Counter* (NACKC) and its management module, *NACKC Manager Module*. Whenever the NACKC Manager detects, thanks to the C signals, that the current state of the *resultant frame* is the ACK slot and that the frame has passed the CRC checking, it checks in B_i if the node is sending a dominant value to acknowledge the frame. If this dominant value is not sent, then the NACKC Manager increases the NACKC.

The NACKC Manager decreases the NACKC whenever a dominant bit is issued through the port. It is important to note that by decreasing the counter, instead of resetting it when detecting a dominant bit value, the hub can detect not only stuck-at-recessive failures, but also nodes that tend to be stuck-at-recessive.

When the NACKC exceeds the *Non-Acknowledge Threshold* (NACKT), the Threshold Control Module does not disable the port, but notifies the user about the inactivity of the port by means of a LED. The specific value for the NACKT can be configured depending on how strict we want to be when considering a port as being stuck-at-recessive. For instance, since an error-active node should send an active error flag after omitting an ACK bit, even a NACKT value equal to 1 can be considered if we want to be very strict when detecting a crashed or absent node.

4.4 Bit-flipping faults

As said before, a bit-flipping fault occurs whenever a component of the network sends erroneous and random bits with no restrictions in the value domain. From the hub point of view, such kind of fault manifests as any of its ports receiving too many arbitrarily erroneous sequences of bits.

The CAN standard specifies some mechanisms in each node that can be used in order to detect such kind of faults. Each CAN node includes a *Transmission Error Counter* (TEC) and a *Reception Error Counter* (REC). These counters are increased and decreased following some rules established in the CAN specification. When any of these counters exceeds a given threshold, the corresponding CAN node reduces its impact on the communication process by going in the *error passive* state. Moreover, a CAN node may disconnect itself from the network by entering into the *bus-off* state [2] if a second threshold is also exceeded. This fault-confinement mechanisms are aimed at preventing further propagation of local errors. See Section 3.3 for a further explanation about the error-passive and bus-off states.

Nevertheless, these mechanisms based on the TEC/REC included in each CAN node present some deficiencies that make little advisable for the hub to rely on these mechanisms for achieving fault confinement. First, normal CAN nodes can fail in arbitrary ways and for this reason may stop performing fault confinement. Second, if a medium is the source of bit-flipping faults affecting all nodes, it cannot be isolated by the nodes. Finally, the accuracy of the error detection strategy followed by the TEC/REC is limited by the restricted vision that the bus imposes, in which the contributions of all nodes are mixed. Thus, we decided to implement in each Enabling/Disabling Unit a dedicated *Bit-Flipping Counter* (BFC) and its associated *BFC Manager*.

Each BFC Manager is aimed at detecting errors in its port contribution. The BFC Manager increases and decreases its BFC depending on the errors it detects. Whenever the BFC exceeds a given *Bit-Flipping Threshold* (BFT), the BFC Manager diagnoses its port as being permanently faulty and isolates it by mean of the corresponding ED signal.

The details of the error detection mechanisms included by the BFC Manager are thoroughly explained in Section 5. Whereas the specific values for increasing and decreasing the BFC, as well as the value for the BFT are explained in Section 7.

5 Fault-diagnosis of bit-flipping faults

As explained in Section 4.4, the fault-diagnosis mechanisms for detecting permanently bit-flipping faults are implemented in each Enabling/Disabling Unit by a *Bit-Flipping Counter Manager* (BFC Manager) and its corresponding *Bit-Flipping Counter* (BFC).

The privileged vision of the hub about each port contribution allows each BFC Manager to evaluate the correctness of its corresponding port by checking that its contribution does not deviate from the correct behavior expected according to the current state of the *resultant frame*. The expected behavior of a given node depends also on the role currently played by this node (i.e. whether the node is a transmitter or a receiver),

on the error conditions detected on the *resultant frame* by the Rx_CAN Module and, finally, on the error conditions detected by each BFC Manager on the contribution of its corresponding node.

Although the behavior of a CAN node is quite complex in the general case, we have been able to identify four independent types of behaviors. These types of behaviors are: the behavior of a transmitter node during a data frame and a remote frame in which no error has been detected so far (see Section 3.4 for a further explanation about each type of frame); the behavior of a receiver node during a data frame, a remote frame and the interframe space[4] in which no error has been detected so far; the behavior of a node after an error condition is detected; and the behavior of a node after the detection of an overload condition.

Note that each BFC Manager performs the error detection based on the expected behavior according to the current state of the *resultant frame*. Although the BFC Manager monitors the couple signal, B_0, and then it knows which is the bit value currently being broadcasted, it does not calculate the rest of the properties that describes the current state of the resultant frame. Instead, in order to save circuitry resources, the BFC Manager uses the set of signals C provided by the Rx_CAN Module (see Section 3.4). Furthermore, each BFC Manager observes its corresponding contribution, B_i, and has the necessary knowledge about the CAN protocol to detect errors in its contribution according to the current state of the *resultant frame*.

It is also worth noting that a BFC Manager does not observe the activity of the other BFC Managers, but takes into account the current state of the *resultant frame* and the contribution of its node in order to take error detection decisions. This is because the evaluation of the correctness of a node can be based only on the node's correct behavior according to the node's knowledge about the current state of the *resultant frame* (it has no sense to demand that a CAN node detects erroneous situations beyond the error detection capabilities of the CAN protocol).

This section is devoted to explaining the different types of error detection mechanisms included in the BFC Manager.

However, before discussing all these issues, note that all the error types detected by the BFC Manager take into account the current role played by the node connected to its port (i.e. if it is a transmitter or a receiver). The role is decided during the arbitration phase, in which the conflict that arises when more than one node tries to gain access to the medium for transmitting is resolved by a contention-based arbitration using the identifiers of the nodes and the AND-logic property of the physical layer [2]. During such arbitration phase, a BFC Manager assumes that its corresponding node has lost the arbitration and becomes a receiver whenever its bit contribution is recessive and the resultant coupled signal has a dominant value. Under normal circumstances, after the arbitration phase only one node considers itself as the transmitter and only the BFC Manager corresponding to such node considers it to be the transmitter (a special type of scenarios in which this is not ensured is explained later on in Section 6.3).

[4]Notice that all the nodes are considered as receivers during the interframe space.

5.1 Error detection on the transmitter node's contribution

The first type of errors the BFC Manager can detect is based on the expected behavior of the transmitter node during a data frame or a remote frame in which no error has been detected so far. The error detection for evaluating this type of behavior is not trivial because although a transmitter node must respect the restrictions imposed by the CAN standard, it is rather free to send the bit values it wishes. Fortunately, the most complete set of error-detection mechanisms available for detecting errors in the contribution of a transmitter is already specified in the CAN protocol [2]. These mechanisms are: *stuff rule check*, *frame check*, *monitoring*, *15-bit cyclic redundancy check* and *acknowledge check*. However, notice that these error-detection mechanisms are based on the observation of the *resultant frame* in which the contribution of the transmitter is mixed with the contributions of all the other nodes. Therefore, the BFC Manager adapts all these error-detection mechanisms in order to directly detect errors in the contribution of the transmitter node.

To implement these error detection mechanisms, a typical CAN node needs to observe the bit value it wishes to send and the current value on the bus. By observing the bus, the CAN node is able to know which is the resultant bit in the bus (the bus acts as a logical AND gate), as well as to calculate, bit by bit, the current state of the *resultant frame*.

In order to adapt the error detection mechanisms of the CAN protocol, the BFC Manager observes its corresponding port's contribution, B_i, the couples signal B_0, and the set of signals C provided by the Rx_CAN Module. The B_i signal allows the BFC Manager to know which is the value of the bit sent by the node (since a node together with its corresponding link are considered as an error-containment region, the BFC Manager assumes that the bit issued through the port is the bit the node wishes to send). Additionally the BFC Manger uses the signal B_0 to know the value of the resultant bit that (as in a bus) all the nodes should see. Finally, the set of signals C complete the description of the current state of the *resultant frame*.

Also note that since after the arbitration phase only one node is considered as the transmitter (see Section 5), only one BFC Manager will consider its corresponding contribution as the transmitter node's contribution during the rest of the data or remote frame. Next, it is specifically explained how the BFC Manager corresponding to the transmitter node adapts each CAN error detection mechanism.

(1) *Stuff rule check*. The BFC Manager corresponding to the transmitter node performs a *stuff rule check* on the stream sent by the transmitter. The signal *bitStuffWaited* included in C indicates whether the current bit of the *resultant frame* is a stuff bit or not, whereas the signal *valueBitStuff* included in C indicates the correct bit value expected for fulfilling the stuff rule. Whenever the signal *bitStuffWaited* indicates that the current bit is a stuff bit, the BFC Manager corresponding to the transmitter checks if the stuff rule is fulfilled by analyzing whether the bit issued through its corresponding port, B_i, matches with the bit value indicated in the signal *valueBitStuff*.

(2) *Frame check*. The BFC Manager corresponding to the transmitter checks that the transmitter's contribution respects the frame format (during a data or a remote frame) specified in the CAN protocol, i.e. the BFC Manager performs a *frame check* on the transmitter's contribution (see Section 3.3 for a further explanation of the different

types of frames). For data and remote frames, the transmitter is allowed to send dominant bits and recessive bits depending on the current field being transmitted [2]. In what concerns the signals involved to perform such *frame check*, the vector *frameField* included in C indicates which is the kind of frame and the field within the frame the current bit belongs to. The BFC Manager uses its knowledge about the CAN protocol in order to check whether the contribution of its corresponding port, B_i, respects the frame format according to the kind of frame and the field indicated by the vector *frameField*.

(3) *Monitoring*. As explained in Section 3.3, the transmitter node in a CAN bus performs a *monitoring* of the *resultant frame* in order to check that whenever it transmits a bit, such bit value is the value seen in the *resultant frame* (except during the arbitration phase and in the ACK slot, where it is allowed that a dominant bit overwrites a recessive bit sent by the transmitter).

As said before, the BFC Manager considers its corresponding node and its link (that includes the uplink and the downlink) as a single error-containment region and, thus, it assumes that the bit observed in its corresponding port's contribution is the bit sent by the node. Nevertheless, the BFC Manager cannot be sure about which is the real bit value the node sends and receives since errors can occur in the uplink and the downlink. Therefore, the BFC Manager corresponding to the transmitter cannot know when the transmitter detects a bit error.

However, notice that if a bit error occurs, the BFC Manager corresponding to the transmitter is able to detect, sooner or later within the frame, that the transmitter had any kind of problem by means of any of the error detection mechanisms based on the CAN protocol it adapts (stuff error, format, CRC error). This is because the transmitter node should signal an error when detecting a bit error. Thus, such error signaling forces the BFC Manager corresponding to the transmitter to detect an error by means of any of the CAN error detection mechanisms it adapts. In addition, even if the transmitter does not signal the bit error (due to extra errors the transmitter may not detect that the bit value it sent changed when reaching the hub), the BFC Manager will also detect an error in the frame later. That is because the error mechanisms of the CAN protocol ensure that an error will be detected if less than 5 errors occur within the same frame [2]. For instance, a bit-error that is not signaled by the transmitter will provoke a CRC error.

(4) *15-bit cyclic redundancy check*. As said before in Section 3.3, the transmitter node calculates and sends within the frame a 15-bit cyclic redundancy code based on the bits of the frame it has already transmitted. Note that if the contribution of any port corresponding to a receiver node changes any of the bits that constitute the content of the CRC or that are included in the set of bits from which the CRC is calculated, then the transmitter node should detect such situation as a bit error and should abort the transmission of the frame by signaling an error. Therefore, it is assumed that the responsible for the correctness of the value of the CRC that can be seen in the *resultant frame* is the transmitter. Thus, only the BFC Manager corresponding to the transmitter has to check that the frame passes the *15-bit cyclic redundancy check* performed by the Rx_CAN Module over the *resultant frame* within the hub. Regarding the signals involved in the CRC error detection, the signal *CRCPassed* included in C indicates whether the frame has passed the CRC check performed by the Rx_CAN Module or

not. The vector *frameField* included in C indicates the frame and the field that is currently being transmitted and, indirectly, when the CRC field ends. The BFC Manager corresponding to the transmitter observes the vector *frameField* in order to know when the CRC field ended, and then, checks by means of the signal *CRCPassed* if the frame has passed the CRC *15-bit cyclic redundancy check*.

(5) *Acknowledge check*. In CAN, when no receiver node acknowledges the frame, the transmitter detects an ACK error by means of the *ACK check*. Notice that since in CAN the transmitter uses the *ACK check* in order to know if the frame it sent passes such CRC check, it does not perform the *15-bit cyclic redundancy check*. However, since the BFC Manager corresponding to the transmitter already performs the *15-bit cyclic redundancy check*, it does not use the knowledge in which the *ACK check* is based on to detect a CRC error in the transmitter contribution. Also note that to use the *ACK check* instead of the *15-bit cyclic redundancy check* in order to detect a CRC error in the transmitter contribution is not feasible. An *ACK error* can also occur, even if the contribution of the transmitter is correct, if the receiver nodes (or their links) have any problem and are not able to acknowledge a correct frame. Therefore, since the aim of the hub is to isolate permanently faulty ports and not to isolate correct ports, the *15-bit cyclic redundancy check* but not the *ACK check* should be performed for detecting CRC errors in the transmitter's contribution.

5.2 Error detection on the contribution of receiver nodes

As stated in Section 5, BFC Manager is able to detect a second type of errors, namely errors in the contribution of a receiver node during data frames, remote frames and interframe spaces in which no error has been detected so far.

Contrary to the case of error detection on the contribution of a transmitter node (explained above in Section 5.1), the error detection on the contribution of a receiver node is easy. This is because a receiver is only allowed to send recessive bits, except in two cases: during the ACK slot within a data and a remote frame, and during the *idle* field.

Next, it is explained how the BFC Manager corresponding to a receiver node specifically checks its node contribution during the ACK slot and the *idle* field.

In what concerns the ACK slot, it is important to note that a receiver node must only acknowledge a frame if it has not detected any error (including the CRC error). In the case a receiver node detects an error in the CRC, it does not acknowledge the frame, but signals the error in the first bit of the *End Of Frame* field (EOF) of the frame (which is located after the ACK slot).

The BFC Manager corresponding to a receiver node expects that the result of the CRC checking performed by the receiver is equal to the result of the CRC checking performed by the Rx_CAN Module. If the CRC checking performed by the receiver node does not coincide with the CRC checking performed by the Rx_CAN Module it means that an error occurred in the receiver node or in its downlink. Therefore, the BFC Manager corresponding to a receiver node will allow its corresponding node to send a dominant bit in the ACK slot only if the frame has passed the CRC check performed by the Rx_CAN Module. Otherwise, the BFC Manager will expect the receiver to send a recessive bit in the ACK slot.

With regard to the signals that are involved in such error detection, the vector *fieldFrame* included in C indicates which kind of frame and which field within the frame the current bit belongs to, whereas the signal *CRCPassed* indicates whether the frame has passed the CRC check performed by the Rx_CAN Module. When the *fieldFrame* indicates that the current bit belongs to the ACK slot, the BFC Manager corresponding to the receiver will expect its corresponding node to send a dominant bit in the ACK slot if the signal *CRCPassed* indicates that the frame has passed the CRC check; otherwise, it will expect its corresponding receiver node to send a recessive bit in the ACK slot. Nevertheless, note that since the lack of the ACK bit is used for detecting stuck-at-recessive faults, a node not sending a expected ACK slot should not be considered as a bit-flipping behavior.

Note that since each BFC Manager corresponding to a receiver node independently checks its node contribution, a receiver node omitting the acknowledge can be detected even though in the *resultant frame* the ACK slot has a dominant value. This represents an improvement when compared with CAN, where it is impossible to detect a node omitting an ACK if any other node is acknowledging the frame.

Beside the case of the ACK slot, the other exception in which the BFC Manager corresponding to a receiver allows its receiver node to send a dominant bit is during the *idle* field (when no frame is being sent). During such field, any receiver node that wishes to send a frame starts the transmission of such frame by means of a dominant bit which constitutes the so called *Start Of Frame* (SOF). When this occurs, the BFC Manager of the receiver node will not consider it as an error, but instead that the receiver node becomes a transmitter.

Hence, after monitoring a SOF through its port, the BFC Manager will apply the error detection mechanisms for the contribution of a transmitter node described above in Section 5.1. However, the BFC Manager will consider that its corresponding node becomes again a receiver if during the arbitration phase its node sends a recessive bit and at the same time a dominant bit is observed in the coupled signal. This actually means that the node loses the arbitration against any other node that sends a frame with an identifier with higher priority. Thus, if a BFC Manager detects that its corresponding node has lost the arbitration, it will check that this node acts as a receiver during the rest of the frame, thus using the error detection mechanisms just described above.

5.3 Error detection on nodes' contribution after detecting error conditions

As explained in Section 5, there are different types of errors that can be detected on each port contribution. In Sections 5.1 and 5.2 the error detection mechanisms for detecting erroneous contributions of a transmitter and of a receiver as long as no error is detected has been explained. These error detection mechanisms are independently used by each BFC Manager, which basically detects an error in its corresponding port by checking that its contribution agrees with the expected behavior according to the current state of the *resultant frame*. The third type of errors that can be detected on each port contribution takes into account the behavior of a given port after an error has been detected on this port or on any other port during data frames, remote frames or

interframe spaces in which no error has been detected so far.

Note that an error affecting a port can be not only detected by the BFC Manager corresponding to that port on its contribution, but also can provoke a *CAN detectable error* in the resultant frame. We define a *CAN detectable error* as either, an error that a CAN node is able to detect by means of any of the error-detection mechanisms specified in the CAN protocol, or an error that can be detected by the hub by adapting these error detection mechanisms. For the shake of simplicity, we will assume that when generally talking about the detection of errors in the resultant frame, we are actually referring to the detection of *CAN detectable errors*.

An error affecting a given port will be detected by its corresponding BFC Manager by means of the error detection mechanisms explained before in Sections 5.1 and 5.2. In contrast, an error affecting the *resultant frame* will be detected by the Rx_CAN module adapting the error detection mechanisms the CAN protocol already specifies as will be seen later on.

The type of contribution that a BFC Manager expects after an error is detected during data frames, remote frames or interframe spaces in which no error has been detected so far depends on where the error was detected: in the port contribution, regardless it is also detected in the *resultant frame* or not; or in the *resultant frame*, only.

In the case of a BFC Manager detecting an error in its corresponding contribution (regardless the error is also detected in the *resultant frame* or not), it will assume that the error is provoked by one of two possible reasons. On the one hand, note that a node that locally detects an error (i.e. that detects a *local error*) will signal it by means of an active error flag. This error flag will provoke an error in the port contribution. Thus, the first reason why a BFC Manager will detect an error in its contribution is its corresponding node signaling a local error. On the other hand, a bit-flipping fault in a node or in a link will generate a bit-flipping stream that will provoke errors in the port contribution. So, the second reason of a BFC Manager detecting an error in its contribution is a bit-flipping fault located in its corresponding port. The type of error detection mechanisms that the BFC Manager includes in order to check the contribution of its corresponding port after detecting an error take into account these two possible reasons for an error to appear on its port.

In contrast, the BFC Manager expects a different type of contribution if it does not detect an error in the contribution of its corresponding port, but an error on the *resultant frame*. Obviously, CAN nodes implement the error detection mechanisms of the CAN protocol for detecting errors in the *resultant frame*. Thus, when a BFC Manager detects an error in the *resultant frame* it will assume that its corresponding node has also detected it. In this situation, the BFC Manager will expect its corresponding node to signal the error by means of an active error flag.

As said above, the BFC Manager does not adapt all the error detection mechanisms of the CAN protocol for detecting errors in the *resultant frame*. Notice that as explained in Section 3.3, the Rx_CAN Module implements a subset of the error detection mechanisms of the CAN protocol in order to keep the synchronization at frame level. Specifically, the Rx_CAN Module implements the following CAN error detection mechanisms: *stuff rule check*, *frame check*, and *15 bit cyclic redundancy check*. When the Rx_CAN Module detects any of these errors in the *resultant frame*, it in-

forms the different BFC Managers about the error by indicating (by means of the set of signals C) in the next bit that the current state of the *resultant frame* takes into account the transmission of an active error flag (remember that in Section 3.3 it is explained that the Rx_CAN Module orders the Error Flag Generator to transmit an active error flag when detecting an error)). Therefore, in order to detect errors in the *resultant frame* each BFC Manager does not need to adapt the error detection mechanisms of the CAN protocol that the Rx_CAN Module already implements, but the rest of them as will be explained later in 5.5.

Deeper explanations about the error detection mechanisms the BFC Manager includes for checking the correctness of the contribution of its corresponding port in both cases after detecting an error in its corresponding contribution and after detecting an error in the *resultant frame* can be found next in Sections 5.4 and 5.5.

5.4 Error detection after an error condition on the contribution of a port is detected

In Section 5.3 it has been explained that the expected contribution of a port after an error is detected during a data frame, a remote frame or the interframe space in which no error has been detected so far depends on whether the error is detected in the contribution of the port, or the error is not detected in the contribution of the port but on the *resultant frame*. This section is devoted to explaining the error detection mechanisms that the BFC Manager uses in order to evaluate the contribution of its corresponding port after it has detected an error on it.

Each BFC Manager observes its corresponding contribution and uses the error detection mechanisms described in Sections 5.1 and 5.2 in order to detect whether each bit is erroneous according to the current state of the *resultant frame*. As said before, an error in a given port contribution that is detected by means of these error detection mechanisms can appear due to two different reasons. First, the BFC Manager detects an error if the corresponding node detected a local error and is signaling it. Second, a bit-flipping fault can also provoke an error in the port contribution. Thus, the error detection mechanisms used by the BFC Manager to check the contribution of its corresponding port after detecting an error on it take into account these two cases.

The specific contribution the BFC Manager expects in these two cases depends on multiple factors such as the value of the bit, issued through the port which causes the error, the role played by the node corresponding to the port when the error is detected, etc. Next, the different error detection mechanisms the BFC Manager uses for checking the contribution in these cases are explained.

First, lets figure the case of an error occurring during a data frame, a remote frame or a interframe space in which no error has been detected so far.

If the erroneous bit is dominant, then the corresponding BFC Manager assumes that its node has detected a local error and is signaling it. The BFC Manager will check that the node correctly signals the error by means of an active error flag followed by a cooperatively error delimiter (see Section 3.3 for a further explanation of the signaling of errors). First, the BFC Manager will check that the node sends a correct active error flag constituted by 6 consecutive dominant bits. Notice that since the BFC

Manager assumes that the node is signaling a local error, the BFC manager considers that the dominant bit that provoked the error and all the preceding consecutive dominant bits are part of the active error flag. When the active error flag is finished, the BFC Manager will check that the node sends recessive bits until all nodes have finished transmitting the cooperative error delimiter. In what concerns the signals involved in this checking, the BFC Manager observes its corresponding port contribution, B_i, in order to check that the node sends the expected dominant and recessive bits during the error signaling. Moreover, the BFC Manager will observe which is the field being transmitted by means of the vector *frameField* included in C for detecting when the cooperative error delimiter is finished (after the error delimiter, the field indicated by the current state of the *resultant frame* is *idle*).

In contrast, if the bit that provokes the error during a data frame, a remote frame or a interframe space in which no error has been detected so far is recessive, the expected contribution depends on the role played by its corresponding node just before the error is detected.

If the node is acting as a transmitter, the unique error that it can provoke when transmitting a recessive bit is a stuff error. In this case, the expected contribution will also depend on the value of the resultant bit at the coupled signal, B_0, when the transmitter sends the incorrect recessive bit (it is possible that a receiver node incorrectly sends a dominant bit which coincides with the recessive bit sent by the transmitter). If the resultant bit is recessive, the BFC Manager of the transmitter expects its corresponding node to detect the stuff error and to signal it by means of an active error flag, followed by a cooperative error delimiter. Otherwise, the BFC Manager will not expect its corresponding node to signal an error, but continuing with the current transmission. This is because the erroneous recessive bit will be considered as a bit-flipping bit which cannot be detected by any node since it is masked by a dominant bit. In the case that the BFC Manager corresponding to the transmitter assumes that its corresponding node has to detect the stuff error, the BFC Manager will check that the transmitter send an active error flag constituted by 6 consecutive dominant bits and that, afterwards, the transmitter will cooperatively transmit an error delimiter constituted by recessive bits. With regard to the signals used by the BFC Manager, beside observing the coupled signal, B_0, it observes the contribution of its corresponding node, B_i, and the vector *frameField* included in C. The BFC Manager will observe the signal B_i for taking into account which is the value of the bit issued through its port and will observe the vector *frameField* included in C to detect when the error delimiter is finished.

If the node that sends the incorrect recessive bit was acting as a receiver just before the error is detected, it only can be the case of a receiver node not sending a dominant bit in the ACK slot of a frame that has passed the CRC checking performed by the Rx_CAN Module (see Section 5.2 for further details about the error detection on the contribution of a receiver in the ACK slot). Remember that the CAN protocol specifies that a receiver node does not send a dominant bit in the ACK slot if it has detected a CRC error. In such a case, the receiver must start to signal the error in the first bit of the EOF field. Therefore, when the BFC Manager related to a receiver detects that its corresponding node incorrectly does not acknowledge a correct frame (a frame that has passed the CRC checking performed by the Rx_CAN Module), the BFC Manager will assume that the receiver node had a local error that leaded it to incorrectly perform

the CRC checking. Thus, the BFC Manager will expect the node to start sending an active error flag in the first bit of the EOF field and to cooperatively transmit an error delimiter afterwards.

Nevertheless, as said in Section 5.2, the detection of an ACK omission is not considered as a bit-flipping error, because it is used to detect stuck-at-recessive faults. Hence, if a node does not start to transmit an error flag at the first bit of the EOF filed after it has omitted the ACK, it will not be considered as a bit-flipping error. In contrast, if the node effectively starts to send the error flag, the previous ACK omission will be considered as a bit-flipping error and not as a sight of an stuck-at-recessive fault.

Regarding the signals involved in this error detection, as in the previous case, the BFC Manager observes the contribution of its port, B_i, and the vector *frameField* included in C to check that the error signaling is correct. In addition, the BFC Manager also uses the signal *CRCPassed* to know if the frame has passed the CRC checking performed by the Rx_CAN Module, as well as the vector *frameField* for detecting when the EOF field starts.

Until this point the error detection mechanisms the BFC Manager uses for checking the contribution of its related port after detecting an error in this port during data frames, remote frames or interframe spaces in which no error has been detected so far have been described. Moreover, it has been also explained that the type of expected contribution depends on the value of the bit which provoked the error, as well as on the role of the node connected to the port.

Notice that in all these cases the BFC Manager expects its port to correctly signal the particular error situation. Hence, one may wonder what are the error detection mechanisms performed by the BFC Manager for dealing with bit-flipping streams that do not match with such expected error signallings, i.e. what are the mechanisms for evaluating a corrupted error signaling.

A possible solution to deal with bit-flipping streams would be to check bit by bit during an error signaling if the contribution is correct and, in case of detecting an erroneous bit, to pretend to know which is the expected contribution after observing this erroneous bit. Unfortunately, due to the random content of a bit-flipping stream, the BFC Manager cannot be sure of the type of contribution it has to expect when detecting a bit-flipping bit. Thus, if during a bit-flipping situation, the BFC Manager pretends to know which is the expected contribution after observing each bit-flipping bit, it will unfairly detect erroneous contributions during all the bit-flipping stream. Therefore, in order to not unfairly detect these errors, the BFC Manager will not be strict when checking the correctness of an error signaling.

Specifically, the BFC Manager will check the contribution as follows. When the BFC Manager detects an erroneous bit in the contribution of its port during a data frame, a remote frame or an interframe space in which no error has been detected so far, it will expect the node to signal an error as explained before in this section. However, during the supposed error signaling, the BFC Manager may detect an incorrect bit. This incorrect bit can be either a recessive bit detected before the active error flag should be considered finished, or a dominant bit when it is supposed that the node is participating in the transmission of the error delimiter.

In the case of detecting an erroneous recessive bit during the supposed active error flag, the BFC Manager will assume that the node has started to send again correct

bits according to the current state of the *resultant frame*. If the current state of the *resultant frame* indicates that no error is being signaled (the *resultant frame* is still considering the transmission of a data frame, a remote frame or an interframe space in which no error has been detected so far), then the BFC Manager assumes that the previous dominant bits were a bit-flipping sequence of consecutive dominant bits not provoked by the node, but by its uplink. Thus, the BFC Manager will consider that its node is not signaling an error and that the following bits sent by that node will be correct according to the current state of the *resultant frame*. Otherwise, if the current state of the *resultant frame* indicates that an error signaling is currently being transmitted, the BFC Manager assumes that its corresponding node is already transmitting recessive bits for building the error delimiter and that the error flag was to short due to extra errors during its signaling.

For example, imagine a receiver node that starts to send erroneous dominant bits during the data field of a frame. Its corresponding BFC Manager will consider that it is signaling a local error. However, the BFC Manager receives a recessive bit before the supposed active error flag is finished. In such situation, if the current state of the *resultant frame* indicates that the actual field being transmitted is still the data field, the BFC Manager will assume that the receiver node is also considering that field and that it will send correct recessive bits. In contrast, if the current state of the *resultant frame* indicates that an error is being signaled, then the BFC Manager will assume that the receiver node has finished its own active error flag and that it is participating in the transmission of the error delimiter, although extra errors have lead the BFC Manager to observe a too short error flag.

In contrast, the error detection holds as follows when the BFC Manager detects an erroneous dominant bit when it supposed that its node was participating in the error delimiter. First, if the dominant bit is detected when the BFC Manager has already observed any recessive bit of the supposed error delimiter in its contribution, then it can assume two different things depending on the current state of the *resultant frame*. In the case the current state of the *resultant frame* during the previous bit indicated that and error delimiter was being transmitted, and at the current bit it indicates that an error has been detected in the *resultant frame*, then the BFC Manager assumes that the dominant bit issued by its node is correct because it is signaling a *CAN detectable error*. Otherwise, the BFC Manager considers that the node detected an extra local error during its participation in the error delimiter and that it is signaling it by means of a new active error flag. In such situation, the BFC Manager will check that the new error signaling is correct by means of the error detection mechanisms already explained in this section (it will check that the node transmits an active error flag followed by a cooperatively error delimiter).

Second, if the BFC Manager detects an erroneous dominant bit in the very first bit of the supposed error delimiter (just in the bit following the previously supposed active error flag), then the BFC Manager assumes that the node detected an extra local error during the transmission of its own active error flag, and that it is transmitting an overlapped or a consecutive error flag. In this case, as before, the BFC Manager will check the correctness of the new error signaling, but allowing an active error flag constituted by 1 up to 6 consecutive dominant bits. This is because the extra local error could happen in any of the bits that constituted the previous active error flag.

Finally, it is important to note that all the error detection mechanisms explained above are applied after detecting an erroneous bit in a port contribution, except in the contribution of a port corresponding to a transmitter node after detecting a CRC error. Remember, that the CRC sequence is calculated based on many of the bits that constitute the frame, then the contribution of the transmitter cannot be considered as incorrect in the moment in which a CRC error is detected (the error can be in any of the bits from which the CRC is calculated) and thus its BFC Manager cannot assume that the transmitter will imminently signal an error. Moreover, since a transmitter node does not performs a CRC checking, its BFC Manager will not expect an error signaling when detecting a CRC Error.

Furthermore, notice that although the hub always considers the transmitter node as the responsible of a CRC error, the CRC checking is performed by the Rx_CAN Module based on the *resultant frame* (see Section 5.1). Thus, the CRC error can be considered as an error that is detected in the *resultant frame* and, therefore, the error detection mechanisms for evaluating the expected contribution of each port after detecting it are discussed later on in Section 5.5.

5.5 Error detection after an error condition in the *resultant frame* is detected

In Section 5.4 the different types of contributions a BFC Manager expects from its corresponding port after detecting an error on it during data frames, remote frames or interframe spaces in which no error has been detected so far are explained. However as said in Section 5.3, an erroneous contribution may provoke an error (a *CAN detectable* error) in the *resultant frame*. The current section is devoted to explaining how the BFC Manager detects errors in the *resultant frame*, as well as how it checks the correct contribution of its port after such kind of errors are detected.

The BFC Manager adapts a subset of the error detection mechanisms of the CAN protocol for detecting errors in the *resultant frame*. Specifically, the error detection mechanisms that adapts are: *monitoring* and *acknowledge check*. The rest of the CAN error detection mechanisms are performed by the Rx_CAN Module, which informs each BFC Manager about the detection of an error by means of some signals included in C. There error detection mechanisms are: *stuff rule check*, *frame check* and *15 bit cyclic redundancy check*.

Next, each one of the different error detection mechanisms the BFC Manager implements for checking the contribution of its corresponding port after detecting an error in the *resultant frame*, as well as the signals involved in each case are explained. Note that the adaptation of the *monitoring* and the *acknowledge check* mechanisms carried out by the BFC Manager are also next described when explaining the *Bit error signaling check* and the *ACK error signaling check* respectively.

(1) *Stuff/format error signaling check*. The CAN protocol specifies that whenever a node detects a stuff or a format error, it must start to signal the error in the next bit. Therefore, as soon as the Rx_CAN Module detects a stuff error or a format error in the *resultant frame*, each BFC Manager that is not already checking its port's contribution after detecting an error in it, will check that its corresponding port starts signaling the

stuff error or the format error in the next bit. The signals involved in this case are used as follows. Once the Rx_CAN Module detects a stuff or a format error in the *resultant frame*, it will indicate (by means of the vector *frameField*) at the next bit that the state of the *resultant frame* is accounting the transmission of an active error flag (remember again that the Rx_CAN Module orders the Error Flag Generator to transmit an active error flag when detecting an error, see Section 3.3). Notice that if all nodes are synchronized with each other at frame level, they must detect a global error at the same time and they have to start signaling it at the same bit. Therefore, when the BFC Manager observes by means of the vector *frameField* that the transmission of an active error flag has been started in the *resultant frame*, it checks by means of B_i that its corresponding node has also started sending an active error flag.

(2) *Bit error signaling check.* As explained above in Section 3.3, a CAN node detects a bit error whenever it sends a dominant bit, but observes a recessive bit. Furthermore, a CAN transmitter node also detects an error when it transmits a recessive bit and observes a dominant bit during a data or a remote frame (except at the *ACK slot* field and during the arbitration phase, see Section 3.3). A node that detects a bit error must signal it in the bit after the error is detected.

Nevertheless, to be able to check that a node correctly signals a bit error, the hub needs to detect when this node should detect it. Note that, a given node can detect a bit error due to different situations (a electromagnetic interference that locally affects the vision the node has about the channel, a synchronization problem due to clock drift that provokes the node to perform an error when sampling, etc). However, the hub cannot detect all the situations that may provoke a bit error because it only has knowledge about the bits that receives and sends through the links.

Fortunately, since the hub independently observes the contribution of each port, it can use the following mechanism in order to detect one situation in which a node should detect a bit error. The hub will assume that a bit error should be detected by a node if the bit issued by the node is incorrectly changed by the bit issued from any other node. This is actually the only mechanism that the hub can include to adapt the *monitoring* mechanism of CAN.

However this adaptation has one more restriction: when a given node sends a dominant bit, the hub cannot detect if such node may detect a bit error. This is because a dominant bit sent by a node cannot be changed by the bit issued by any other node. Thus, the hub can only detect that a node may detect a bit error when the node sends a recessive bit.

The BFC Manager corresponding to the transmitter includes this mechanism in order to detect when its corresponding node should detect a bit error. If the BFC Manager corresponding to the transmitter detects that its node sends a recessive bit and that, at the same time, any other port sends a not allowed dominant bit, it will check that in the next bit the transmitter node starts signaling an error. Notice that, in contrast, the BFC Manager corresponding to a receiver does not performs such error detection. This is because a CAN receiver that observes a dominant bit when it has issued a recessive bit does not detect a bit error.

In what concerns the signals involved in this error detection mechanism, the BFC Manager corresponding to the transmitter observes the contribution of its port, B_i, the vector *frameField* included in C and the coupled signal, B_0. The BFC Manager cor-

responding to the transmitter will use the signal *frameField* and the CAN format rules to know when the transmitter and the receivers are allowed to send dominant bits and recessive bits. When the transmitter is allowed to send dominant and recessive bits, but the receiver is only allowed to send dominant bits, the BFC Manager corresponding to the transmitter compares the port contribution of the transmitter with the coupled signal. In such situation if a recessive bit is issued through the port of the hub corresponding to the transmitter node and the coupled signal has a dominant value, then the BFC Manager corresponding to the transmitter assumes that a receiver is sending an incorrect dominant bit which should trigger a bit error detection at the transmitter (it is enough a dominant bit issued from one port to force a dominant bit at the coupled signal).

Once the BFC Manager considers that its node should detect a bit error, it observes the contribution of its corresponding port,B_i, in order to check that the node starts to signal an active error flag in the next bit.

(3) *CRC error signaling check*. The CAN protocol specifies that whenever a receiver node detects a CRC error, it must start to signal such error in the first bit of the *End of frame* field (EOF). Therefore, when the *resultant frame* does not pass the CRC checking performed by the Rx_CAN Module, each BFC Manager corresponding to a receiver expect its node to start sending an active error flag in such bit of the EOF. For checking the CRC error signaling, each BFC Manager corresponding to a receiver observes the signal *CRCPassed* to know if the *resultant frame* has passed the CRC checking; and the vector *frameField* for knowing when the EOF begins. When the BFC Manager corresponding to a receiver detects that the frame has not passed the CRC checking, it will observe the vector *frameField* and the contribution of its port, B_i, in order to check that its corresponding node starts sending an active error flag in the first bit of the EOF field.

(4) *ACK error signaling check*. The CAN protocol specifies that whenever a transmitter detects an ACK error, it must start to signal such error in the bit following the *ACK slot*. Therefore, when no receiver sends a dominant bit at the *ACK slot*, the BFC Manager corresponding to the transmitter expects the transmitter to detect an ACK error and, therefore, to start to send an active error flag in the next bit.

In order to adapt the *acknowledge check* mechanism for detecting the ACK error, the BFC Manager corresponding to the transmitter first observes the vector *frameField* for detecting when the ACK slot is being broadcast. In addition, the BFC Manager also has to observe the coupled signal, B_0, for detecting whether the ACK slot has a dominant bit or a recessive bit (all the contributions must be recessive to force a recessive value at the coupled signal). If the BFC Manager corresponding to the transmitter detects a recessive bit in B_0 during the ACK slot, it assumes that an ACK error occurred.

When the BFC Manager corresponding to the transmitter detects an ACK error, it will observe its port contribution, B_i, to check that the transmitter starts sending an active error flag in the next bit.

All these error detection mechanism are used to check if the corresponding node starts to signal errors adequately. But, in addition, the BFC Manager checks the correctness of such error signaling. Specifically, the BFC Manager will check that the node sends an active error flag constituted by 6 consecutive dominant bits, followed by a cooperative error delimiter constituted by recessive bits.

Note that if the BFC Manager detects an erroneous bit in its contribution which violates this format of the error signaling, it will assume that its node detected an extra local error. Therefore, in this situation, the BFC Manager will check the contribution of its corresponding port by means of the error detection mechanism described before in Section 5.4 for detecting errors during a bit-flipping stream.

However, it is also possible that an error occurs in the *resultant frame* during the error signaling itself. In particular, this will occur when during the error delimiter a dominant bit is observed at the coupled signal as a consequence of an incorrect dominant bit issued by any port. In such situation, each BFC Manager, corresponding to a port not responsible of the new error situation, will expect the corresponding node to start to signal an error again (an active error flag followed by a cooperative error delimiter).

5.6 Error detection after detecting overload conditions

The fourth type of expected contribution identified in Section 5 corresponds to the behavior of a node after detecting an overload condition. Remember that there are two kinds of overload conditions [2]. On the one hand, an overload frame is sent by a receiver node that needs an extra delay before a new frame can be transmitted on the bus. Specifically, this overload frame is called *LLC-requested overload frame* and a receiver node can start sending it only in the first bit of the intermission field. On the other hand, the CAN protocol specifies a second type of overload condition. Any receiver node detecting a dominant bit either in the last bit of the EOF field or in any bit of the intermission field, as well as the transmitter node detecting a dominant bit in any bit of the intermission field react sending a so called *reactive overload frame*. Actually, *reactive overload frames* are the mechanism for globalizing an overload condition.

The format of an overload frame is the same as the format of an active error frame [2]. It is constituted by an overload flag of 6 consecutive dominant bits followed by a cooperative overload delimiter of at least 8 consecutive recessive bits.

The transmission of an overload flag provokes the globalization of the overload condition by forcing the transmission of *reactive overload frames*. Whereas the cooperative error delimiter is used for synchronizing all nodes at the end of the overload condition. Notice that, even in the case of some nodes globalizing an overload condition while other nodes are globalizing an error condition (e.g. a dominant bit at the last bit of the EOF will provoke the transmitter and the receivers to signal an error and an overload respectively), all them will be synchronized at the end of the error (and the overload) signaling. This is because an overload flag and an active error flag have the same format and for both types of frames (overload and active error frames) the nodes cooperatively transmit a delimiter.

In what concerns the behavior of the BFC Manager during overload conditions, first notice that it can detect an overload condition in two different contexts: an overload triggered by the contribution of its corresponding port; or an overload not triggered by its port, but by any other port.

On the one hand, if the BFC Manager detects that its corresponding port issues a dominant bit in the first bit of the intermission field, it will assume its node have started to signal an overload condition. For being able to detect such overload situation, a BFC

Manager observes the vector *frameField*, included in the set of signals C, in order to know when the current bit is the first bit of the intermission field. In addition, the BFC Manager observes its corresponding port contribution, B_i, to detect if the port issues a dominant bit that triggers the overload condition.

On the other hand, the BFC Manager detects an overload condition not triggered by its port contribution, but by any other port in two different situations. First, if its port corresponds to a receiver node, the BFC Manager will detect an overload condition when observing, in the *resultant frame*, a dominant bit at the last bit of the EOF field or during the intermission field. Second, if the corresponding node is acting as a transmitter, the BFC Manager will detect an overload condition if a dominant bit is sent by other port during the intermission field. In the case the BFC Manager corresponding to the transmitter detects a dominant bit sent by other port in the last bit of the EOF field, it will not expect its node to signal an overload condition in the next bit, but an error condition.

Regarding the signals involved in the detection of an overload condition in the contribution of any other port, the BFC Manager observes the vector *frameField* included in C. When the Rx_CAN Module detects a dominant bit in the *resultant frame* at the last bit of the EOF or during the intermission field, it indicates in the next bit, by means of the *frameField*, that an overload condition is being signaled.

The second issue regarding the behavior of the BFC Manager during an overload condition is the way in which it checks the contribution of its corresponding port during it. Upon the detection of an overload condition, the BFC Manager will expect that its port issues an overload frame with the correct format. For checking this, it will use the same error detection mechanisms it includes for checking its port contribution when signaling an active error frame. See Section 5.4 for a detailed explanation of these mechanisms.

6 Considerations on the mechanisms for diagnosing bit-flipping faults

The error detection mechanisms the hub includes for diagnosing bit-flipping faults were presented in Section 4.4 and were thoroughly discussed later in Section 5.

The current section is aimed at discussing some aspects related to these fault-diagnosis mechanisms. First, Section 6.1 presents some considerations about the complexity of the design of these mechanisms. Second, remember that in Section 4.4, the advantages of these mechanisms for diagnosing bit-flipping faults over the typical CAN fault-diagnosis mechanisms (based on the TEC/REC) where only briefly outlined. Thus, Section 6.2 deeply explains these advantages. Finally, Section 6.3 discusses some drawbacks of these mechanisms, as well as possible solutions to overcome them.

6.1 Evaluation of the complexity of the mechanisms for diagnosing bit-flipping faults

The complexity of the circuitry included in a device is an important aspect that has to be taken into account when devising dependable systems. Specifically, the probability of failure of a device decreases with its complexity. As said before, in CANcentrate the hub is the most critical element concerning dependability since it is the single point of failure of the communication system. Thus, the fault-diagnosis mechanisms the hub includes were devised in order to reduce as much as possible their complexity in terms of circuitry.

As described before, the mechanisms for diagnosing bit-flipping faults are mainly included in each BFC Manager, which independently operates over a given port. Each BFC Manager basically observes its port contribution, B_i, the coupled signal B_0, the set of signals C from Rx_CAN and uses its acknowledge about the CAN protocol to check if its contribution is correct according with the current state of the *resultant frame*.

The first characteristic of the fault-diagnosis mechanisms that allows reducing the circuitry complexity is the way in which the BFC Manager gets the current state of the *resultant frame*. As stated before in Section 5, the BFC Manager does not calculate this current state by observing the resultant coupled signal, B_0. Instead, the Rx_CAN Module is aimed at calculating bit by bit this current state by observing the coupled signal. Then, the Rx_CAN Module provides the different BFC Managers with a set of signals, C, that, together with B_0, describes this state. Hence, since the logic for calculating the current state of the *resultant frame* is implemented once in the hub (in the Rx_CAN Module), the cost in terms of circuitry is less than it would be if all BFC Manager had to implement it.

This way of reducing the circuitry is even more important if one takes into account that the calculation of the current state includes the detection of several types of errors in the *resultant frame*: *stuff rule check*, *frame check*, and *15 bit cyclic redundancy check*. Between them, the most important reduction is achieved in the checking of the CRC sequence, since the BFC Managers do not need to include the logic for calculating the CRC sequence, which is expensive in terms of circuitry.

The second characteristic of the fault-diagnosis mechanisms that makes possible reducing the amount of circuitry needed for implementing the fault-diagnosis mechanisms is that, as explained in Section 5, each BFC Manager does not monitor the activity of the other BFC Managers. The major benefits of this are that it actually reduces the number of needed interconnections inside the hub and the complexity of the state machines included in each BFC Manager. Moreover, this also makes the hub design more flexible and extensible for further improvements. For instance, to add new ports and their respective Enabling/Disabling units will not require to change the circuitry of the existing modules and units within the Fault-Treatment Module.

6.2 Advantages of the mechanisms for diagnosing bit-flipping faults

As explained in Section 4.4, the hub cannot rely on the fault-confinement mechanisms (based on the TEC/REC) that CAN nodes include for dealing with bit-flipping faults.

The deficiencies of the fault-confinement mechanisms based on the TEC/REC were also outlined there: first, faulty nodes may stop performing fault-confinement operations, second, a bit-flipping fault located in a medium bothers all nodes so that none of them can isolate the fault, and third, the bus imposes a mixed vision of all nodes' contributions, thus, reducing the accuracy of the fault-diagnosis mechanisms. In contrast, the hub we have devised does not present such deficiencies. These section is devoted to explaining in detail how the hub overcomes the deficiencies of the fault-confinement mechanisms based on the TEC/REC.

First, notice that the contribution from all nodes have to pass through the hub, which operates independently from them. Therefore, the fault-containment capacities of the hub do not depend on the correct fault-containment operations performed by the nodes. Hence, isolation of faulty port is guaranteed even if nodes stop performing fault containment operations.

Second, the deficiencies of the mechanisms based on the TEC/REC regarding the impossibility of dealing with faulty media is overcame in CANcentrate since each link is dedicated and, together with its corresponding node, constitutes a single fault-containment region that the hub can treat.

Third, the hub is also able to improve the deficiency of a CAN bus in what concerns the accuracy of the fault-diagnosis mechanisms implemented by typical CAN nodes. In order to understand how the hub does it, it is important to remember how a fault (a faulty node) is diagnosed in CAN [2]. Any CAN node uses the TEC and the REC to basically increase them when the node considers itself as the responsible of an error condition. As explained in Section 4.4, when either the TEC or the REC reach a given threshold, the node enters the *error passive state* which actually reduces the impact of the node on the communication [2]. A second threshold is used if the error passive state is not enough. If the node reaches this second threshold, it enters the *bus-off state* and it is not involved in bus activities. This last situation corresponds to a node diagnosing itself as being faulty.

More particularly, a CAN node decides that it is the responsible of an error condition and thus increases its TEC or its REC when it detects a *primary error* [2]. A node detects a primary error when it monitors a dominant bit after its own error flag. That is because a node that detects this dominant bit can assume that it was one of the nodes that firstly detected an error and started to signal it, thus provoking the other nodes to detect an error and to perform an error signaling too. In other words, a node detecting a primary error means that the node did not detect an error as a consequence of the error signaling from other node, but due to either a local error or an error that could affect more than one node in the network.

Notice that the detection of the primary error is the basis of the mechanisms that any CAN node uses in order to diagnose itself as being faulty. Unfortunately, the detection of a primary error is performed on the *resultant frame*. The limited vision about the contribution of each node, imposed by the coupling on the bus, forces each CAN node to make assumptions about the contributions of the other nodes. Thus, the detection of a primary error cannot be performed when the error occurs, but only some bits later. Specifically, when the node can extract enough information from the *resultant frame* in order to evaluate the behavior of the other nodes. This delay for evaluating the behavior of other nodes may lead a CAN node to incorrectly detect a primary error if extra errors

occur during the transmission of the error flag.

One possible case in which this can occur arises when a node that started to signal an error situation as a consequence of a local error detects an additional local error during its own active error flag. In this situation the node immediately starts the transmission of an additional active error flag, leading the other nodes to incorrectly detect a primary error. Other example of an incorrect detection of the primary error occurs when a node that detects a local error does not monitor a dominant bit after its own active error flag due to an extra local error. In this case the node, which actually should have been detected a dominant bit belonging to the error flags of other nodes, will not detect a primary error.

Due to the inaccurate strategy for detecting the CAN node that is responsible for an error condition, the fault-diagnosis mechanisms specified in CAN are inaccurate too. In contrast, the hub improves the detection of guilty nodes because it has an independent vision of each node contribution. On the one hand, each BFC Manager is able to detect that its port is the responsible of an error condition, without making any kind of assumption on the contribution of any other port. On the other hand, each BFC Manager detects that its port is guilty of an error condition at the instant of time it issues an incorrect contribution. Therefore, a guilty node is detected regardless of whether any other node (or itself) detects extra errors or not. See Section 5 for a thorough explanation of the error-detection mechanisms the BFC Manager implements.

Finally, it is worth noting that the hub also improves the fault-diagnosis mechanisms of CAN by means of a higher capability of error detection. In CAN the error detection is only performed on the *resultant frame*. In contrast, as said in Section 5.3, the hub can detect errors in two different levels: on the *resultant frame* as well as on the contribution of each port. This, allows the hub to improve the capabilities of error detection of CAN by means of two different ways.

First, the fact that the hub adapts some error-detection mechanisms, already specified in CAN, to detect errors at ports makes the hub able to improve the error-detection capabilities of CAN. Note that since in a CAN bus all the contributions are coupled, an incorrect bit issued by a given node may be *masked* by the bit issued by any other node in a way that the erroneous bit cannot be detected at the *resultant frame*. For instance, if a receiver node sends a not-allowed dominant bit in a specific frame field, but this bit coincides with an allowed dominant bit sent by the transmitter, then it would be impossible to detect such erroneous situation in the *resultant frame*. In contrast, the hub monitors each node contribution separately and then, an erroneous bit sent by a port cannot be *masked* by a bit issued from other port.

Other possible scenario in which an erroneous bit is *masked* in a CAN bus was described in Section 5.2. There, a receiver CAN node does not acknowledge a correct frame sending a recessive bit in the ACK slot. In this scenario the transmitter CAN node is not able to detect such ACK omission in the *resultant frame* because the recessive bit sent by the incorrect receiver node is overwritten by the dominant bits issued from other nodes. In contrast, in the same situation, the BFC Manager corresponding to the receiver node that incorrectly does not acknowledge the frame can detect the ACK omission in its corresponding port contribution.

The second way in which the hub improves the error detection capabilities of CAN consists in the use of new error-detection mechanisms that are not specified in the CAN

protocol. These new error-detection mechanisms are mainly related to the detection of errors in each port contribution after an error condition or an overload condition is detected in any port contribution or in the *resultant frame*. See Sections 5.3, 5.4, 5.5 and 5.6 for an explanation of these new error-detection mechanisms.

6.3 Drawbacks of the mechanisms for diagnosing bit-flipping faults and their solution

In Section 6.2 the advantages of the fault-diagnosis mechanisms the hub includes over the fault-diagnosis mechanisms specified in the CAN protocol for detecting bit-flipping faults have been explained. However, the fault-diagnosis mechanisms of the hub present a problem that must be faced.

As explained in Section 5, the mechanisms for detecting errors provoked by bit-flipping faults are included in the BFC Managers. Each BFC Manager basically checks that the contribution of its port is correct according with the current state of the *resultant frame*. Thus, this error-detection mechanisms are based on the assumption that the hub and the nodes are synchronized with each other at frame level, so that they have a consistent view of the current state of the *resultant frame*.

When due to an error a node loses the synchronization at frame level, the hub will detect, sooner or later, an error in the *resultant frame*. In this case, as explained in Section 3.3, the hub will globalize this error in order to force all the nodes to be re-synchronized at frame level at the end of the active error frame.

However, it is worth noting that during all the time that elapses between a node gets de-synchronized at frame level and the instant of time in which this node is re-synchronized at the end of the active error frame, the contribution of this de-synchronized node may not match with the current state of the *resultant frame*. Thus, it is possible that although only one error leads a node to lose the synchronization, the hub detects many errors in its contribution, believing that the node is bit-flipping, until this node is re-synchronized again.

This means that, in some circumstances, the hub unfairly detects errors in a node contribution, thus decreasing the accuracy of the error detection. Since the accuracy of the error detection is a key issue that influences the accuracy of the fault diagnosis, it is mandatory to reduce the possibilities of the hub performing unfair error detections. Although the number of scenarios that can lead the hub to perform unfair error detections is huge, we could be able to identify three kind of these scenarios. Next, this kind of scenarios, as well as the solutions for dealing with them are explained.

The first kind of scenarios regarding unfair error detections are those concerning situations in which the hub detects an error and globalizes it. As explained before, the re-synchronization of a previously de-synchronized node can only be ensured at the end of the active error frame. Thus, the BFC Manager of the de-synchronized node cannot forecast which will be the contribution of this node during the error signaling.

Fortunately, the mechanisms already included in each BFC Manager are devised to be less strict when checking the contribution of its node during an error signaling, than during situations in which no error has been detected so far. These mechanisms, explained in Section 5.4, take into account that a supposed active error frame issued

through a port can actually be a bit-flipping stream or a corrupted active error frame. Thus, such mechanisms are suitable for reducing the number of unfair error detections during scenarios concerning an error signaling.

The second type of scenarios that may lead the hub to perform unfair error detections is related to the error-detection mechanisms, explained in Section 5.5, for detecting a transmitter node erroneously not signaling a bit error. See Section 3.3 for a further explanation of the bit error. As described in Section 5.5, when the BFC Manager corresponding to the transmitter observes that the bit issued through its port is recessive and, at the same time, the bit at the coupled signal is dominant, it assumes that the transmitter node should detect a bit error (remember that the resultant signal the hub broadcast it the logical AND of every node contribution).

If this incorrect dominant bit, which should lead the transmitter to detect a bit error, does not provoke an error in the *resultant frame* detectable by the error-detection mechanisms of a receiver CAN node (*stuff error*, *format error*, *CRC error*) and, additionally, the transmitter does not detect the error, neither the hub nor the nodes will signal an error. Thus, the transmission of the frame will continue.

In such situation, the transmitter should get de-synchronized at frame level unless it actually wanted to send a dominant bit, but due to errors, a recessive bit reached its corresponding hub port. If the transmitter gets de-synchronized, its corresponding BFC Manager will detect later on a *stuff error*, a *format error* or a *CRC error* in its contribution. Therefore, the BFC Manager will detect two errors in its contribution. The first error is detected when the transmitter does not signal the bit error. Whereas the second error is detected when the transmitter violates the stuff rule, any format rule or the CRC sent by the transmitter is checked as incorrect.

This detection of two error is unfair since the transmitter node only behaved incorrectly when it did not detect the bit error, but not when it provoked an error later on. A possible solution for avoiding this unfairly detection would be to not detect an error in the transmitter contribution when it does not signal a bit error. This solution is feasible since, in almost all cases, the BFC Manager of a transmitter node that does not signal a bit error will detect a stuff error, a format error or a CRC error later on. Nevertheless, this solution also implies that the BFC Manager of the transmitter will not detect an error in the case, explained above, in which the transmitter does not signal the bit error, but also does not provoke any error later.

Although to detect an error in the transmitter contribution when it incorrectly does not signal a bit error will usually lead to unfairly detect an extra error in its contribution, it has been decided to not avoid it. This is because it can be considered that this kind of scenarios in which a transmitter does not detect a bit error are very unlikely. Moreover, since to diagnose a port as being permanently faulty should require to detect many errors (see Section 7 later for a discussion about this issue), the unfair but unlikely detection of one extra error should not lead to an erroneous fault diagnosis. Moreover, if we keep the error detection of a transmitter not signaling a bit error, even the exceptional scenario in which the transmitter will not provoke an error later can be detected. This actually increases the error detection capabilities of the hub.

Finally, the last kind of scenarios in which the hub can perform unfair error detections is related to the arbitration phase. During the arbitration phase, each Enabling/Disabling Unit monitors its port contribution in order to decide if its port be-

comes a receiver or stays as a transmitter. An Enabling/Disabling Unit assumes that its corresponding port becomes a receiver if it issues a recessive bit and, at the same time, other port issues a dominant bit (the Enabling/Disabling unit detects such dominant bit at the coupled signal). However, since the bits sent and received by the nodes can be corrupted, it is possible that after the arbitration phase more than one node considers that it has won the arbitration and has became the transmitter. Moreover, since each BFC Manager independently monitors its port and the couple signal, it is possible that more than one BFC Manager assumes that its corresponding node has won the arbitration (regardless its node has considered itself as becoming transmitter or not). Any situation in which either more than one node considers itself as the transmitter or more than one BFC Manager considers its node as the transmitter, will be referred as an *arbitration misunderstanding* hereafter.

It is worth noting that an *arbitration misunderstanding* can happen even if only one error occurred during the arbitration phase. For instance, it is enough that during the arbitration phase a transmitter does not monitor a dominant bit that would lead it to become a receiver and that, thereafter, its identifier coincides with the identifier of the node that wins the arbitration.

Upon an *arbitration misunderstanding*, the BFC Manager should consider the contribution of its corresponding node as being bit-flipping, if this node is assuming itself as playing a role (transmitter or receiver) different from the role the BFC Manager assumes the node is playing. This is because the BFC Manager will expect a different kind of contribution depending on whether it considers its node as a transmitter or as a receiver. Therefore, even if the *arbitration misunderstanding* was provoked by only one error during the arbitration, a BFC Manager can unfairly detect many errors in its contribution.

The number of unfair error detections after an *arbitration misunderstanding* has happened depends on the amount of time any node or the hub needs to detect an error in the *resultant frame* and starts signaling, thus forcing a re-synchronization. An error in the *resultant frame* can be detected in several ways. The hub and the receiver nodes will detect an error if the *resultant frame* violates the stuff rule, any format rule or if the CRC of the *resultant frame* is incorrect. On the other side, a transmitter will detect an error if the *resultant frame* violates the stuff rule, any format rule or, in addition, if it observes a collision (if it detects a bit error provoked by the dominant bit of any other transmitter).

In the general case it is expected that after an *arbitration misunderstanding*, any transmitter observes a collision soon. However, a transmitter can observe a collision only if it sends a recessive bit and, at the same time, any other transmitter sends a dominant bit. Thus, if the transmitters send the same data and take into account the same stuff bits, the observation of a collision will only occur after a considerable amount of time. Specifically, the collision should occur, at least, during the transmission of the CRC field. Because it is very unlikely that different frames have the same CRC.

It has been explained that one error during the arbitration phase is enough to provoke an *arbitration misunderstanding*. Thus, it can be considered that the probability of occurrence of an *arbitration misunderstanding* is not negligible. Therefore, although an *arbitration misunderstanding* rarely will lead to the detection of a high number of unfair error detections, it has been decided to include mechanisms within the hub to

deal with this problem

One possible solution for reducing the impact of the *arbitration misunderstanding* is to decrease the amount of time the hub needs to re-synchronize after a de-synchronization has occurred. Specifically, the hub can force an error globalization (by means of an active error flag) not only when the Rx_CAN Module detects an error in the *resultant frame*, but also when any BFC Manager detects an error. This would effectively abort the frame and re-synchronize the nodes sooner, stopping the unfair detection of errors provoked by the *arbitration misunderstanding*. However, this solution negatively affects the performance of the network since it will abort frames that would not be aborted in a CAN bus (since an error in a port contribution may not cause an error in the *resultant frame*).

An alternative solution has been adopted for our hub: to restrict the number of times that the BFC Manager increases the BFC during a data or a remote frame. This is done by means of a *Bit-Flipping Detection Counter* (BFDC) and a *Bit-Flipping Detection Threshold* (BFDT). See Section 4.4 for an explanation about the BFC and the other elements included in each Enabling/Disabling Unit concerning the detection of bit-flipping errors.

At the beginning of each data or remote frame, the BFC Manager resets the BFDC, but increases it each time that it increases the BFC during such frame. Whenever the value of the BFDC exceeds the BFDT, the BFC Manager orders the Error Flag Generator Module to globalize an error. This globalization will abort the frame and will lead the nodes to re-synchronize because the error flag sent by the hub will force all nodes to send their active error flags. All BFC Managers (including the BFC Manager of the port whose BFDC exceeded its BFDT) continue using its BFC without any restriction during the error signaling in order to check if the node respects the active error flag format (see Section 5.4 for an explanation of the error detection during the error signaling).

However, as in the previous solution in which the hub forces the globalization of any error detected by any BFC Manager, to use this alternative solution based on the BFDC and the BFDT will abort frames that would not be aborted in a standard CAN system. This can occur if a bit-flipping source only generates dominant bits that coincide with correct dominant bits sent by the other node during a data or a remote frame. In such situation no error will be detected in the *resultant frame*, but in one of the ports of the hub. Thus, to force an error globalization whenever the BFDT of this port is exceeded will abort frames that would not be aborted in a standard CAN system.

Fortunately, it can be considered that such situations do not imply a considerable loss of performance. First of all, the probability of a bit-flipping source not provoking an error in the *resultant frame* detectable by the hub or the nodes during more than one frame, but in contrast leading the BFDC to exceed the BFDT must be taken as negligible. This is because it can be considered that, in average, a CAN node should send the same number of dominant bits and recessive bits during a data or a remote frame. Thus, during a data or a remote frame, the probability that a dominant bit generated by a bit-flipping source coincides with a dominant bit send by the transmitter (thus not provoking an error in the *resultant frame*) is of 0.5. Based on this probability, the probability that N dominant bits generated by a bit-flipping source only coincide with dominant bits during a data or a remote frame can be calculated as 0.5^N. A bit-

flipping source should provoke several error detections inside one frame in order to lead the BFDC to exceed the BFDT. This means that, during a data or a remote frame, N should be high enough to provoke the BFDT to be exceeded. Which actually implies that the probability of these N bits not provoking an error in the *resultant frame* is very low.

Furthermore, it can be considered that the solution based on the BFDC and the BFDT may, in some cases, improve the performance. Consider the case in which, during a data or a remote frame, a source of bit-flipping bits will generate an error in the *resultant frame* sooner or later. Note that if the bit-flipping bits are masked by the contribution of the transmitter node, in the worst case, the bit-flipping bits may only provoke an error next to the end of the frame. Therefore, to use the BFDC and the BFDT to abort the frame earlier will save bandwidth and will improve the performance.

Finally, there is a third reason why the adoption of the solution based on the BFDC and the BFDT does not implies a loss of performance. Note that the hub will finally isolate any bit-flipping port, thus, a possible loss of performance provoked by the use of this solution must be taken as temporary.

7 Considerations on the configuration of the BFC Manager Module

In Section 4.4 it has been explained that for diagnosing ports as being bit-flipping faulty, the hub includes for each of them a *Bit-flipping Counter Manager Module* (BFC Manager Module). Each BFC Manager independently operates on its port and increases or decreases its corresponding *Bit-flipping Counter* (BFC) in order to take into account the correctness of the contribution of its port. The BFC Manager diagnoses its port as permanently faulty when the value of the BFC exceeds a given *Bit-flipping Threshold* (BFT).

Additionally, in Section 6.3 it has been explained that each BFC Manager restricts the number of times it increases its corresponding BFC in each data or remote frame in order to reduce the impact of unfair error detections in its port contribution. For this purposes, the BFC Manager uses a dedicated *Bit-Flipping Detection Counter* (BFDC) and a dedicated *Bit-Flipping Detection Threshold* (BFDT).

The present section is aimed at discussing what should be the specific values for increasing and decreasing the counters, as well as the specific values of configuring the thresholds involved in the diagnosing of bit-flipping faults, i.e. the *penalization policy*.

The BFT and the number of units that the BFC has to be increased or decreased can depend on how restrictive the application is. For instance, a very high dependable application may claim for specific issues that enhance the dependability of the communication system. Concretely, as explained in Section 4.1, it is important to not allow a node to be in the error-passive state in order to reduce the probability of data inconsistency. The fault-diagnosis mechanisms of the hub consider the behavior that characterizes the error-passive nodes as incorrect (i.e. passive error flags are considered as erroneous contributions) and then, such nodes are eventually isolated. However, it may be also necessary to apply a tight *penalization policy* in order to minimize as much

as possible the probability of any node entering in the error-passive state.

We consider that a first good approach is to adopt a *penalization policy* based on the standard CAN [2], but introducing slightly modifications in order to be more strict. Such policy presents the following rules.

- When the BFC Manager detects an error in its corresponding contribution, it increases its BFC in 8 units. This corresponds with the value a CAN node must increase the TEC or the REC when detecting a primary error in a CAN bus. As explained in Section 6.2, BFC Managers improve the accuracy of detecting which nodes are the responsible for an error situation and, in addition, they can detect when nodes issue incorrect bits that cannot be detected in a CAN bus. Thus, although the BFC Managers use the same values used in CAN to increase the error counters of guilty nodes, the BFC Managers are stricter.

- When the BFC Manager detects in its corresponding contribution an error related to the format of the active error frame, it will not increase its BFC in 8 units as it was just said, but in 16 units. This is because extra errors during an error signaling are a good indication of a bit-flipping behavior. See Section 5.4 for a detailed explanation about the mechanisms for detecting errors during an error signaling.

- When the state of the *resultant frame* reaches the idle field (i.e. no transmission is on the channel) after a frame transmission ends, all the BFC are decreased in 1 unit. Notice that, this asymmetric approach of increments and decrements is intended to require high reliability of the nodes and links. Moreover, note that the idle field always follows the transmission of an active error frame. Thus, since almost any error detected by a BFC Manager implies the globalization of this error later by means of an active error frame, this asymmetric approach is needed to effectively take into account the errors.

- The bit-flipping threshold can be set to 128 units. Which corresponds to the threshold specified in the CAN protocol to lead a node to enter in the *error passive* state. Note that in the case the accuracy of the error detection performed by the BFC Manager was the same as the accuracy of the error detection of a CAN node, this threshold should lead a BFC Manager to disable the contribution of its node when it enters the error passive state. However, since the error detection performed by the BFC Manager is more accurate than the one performed by a CAN node, it is even less likely that a CAN node enters in the error passive state before its corresponding BFC Manager disables its contribution.

Even when this policy is more exacting than the policy specified in CAN, it is important to note that the hub cannot ensure that a node never enters in the error-passive state by simply adopting a stricter *penalization policy*. It should be necessary to do a further analysis in order to find a proper policy that ensures the hub isolates any node before entering in the error-passive state. Fortunately, an additional solution may be used without addressing this further and complex analysis. This solution is based on the fact that it is possible to force a CAN node to enter in the error-active state after

a reset action. Thus, it is possible to avoid a node to enter into the error-passive state by building the software so that it resets a node whenever it enters into such state. Note that to restart the operation of a faulty node in this way does not negatively affect the communication. This is because the hub will eventually isolate the port corresponding to a faulty node, as well as the hub will keep isolated the port of this node as long as it generates errors.

Finally, with regard to the BFDC and the BFDT it is necessary to choose values that does not imply a loss of performance, compared with a standard CAN system, when applying the solution proposed in 6.3. This solution consists of aborting any data or remote frame when any BFDC exceeds its corresponding BFDT. Note that in that section it was also explained that it should be considered that the loss of performance when applying this solution is negligible. However, this assessment assumes that the value of the BFDT is not exceeded until several errors are detected in its corresponding port during a data or a remote frame.

Thus, we propose to increase the BFDC in 1 unit and to set the BFDT to 5 units. This choice is feasible since the probability of 5 erroneous bits not provoking the abortion of a data or a remote frame in CAN is around $0.5^5 = 0.06$. Furthermore, note that the standard CAN ensures that a data frame or a remote frame is aborted if the number of errors during the frame are less or equal to 5. Therefore, if the BFDT is set to 5 units and the BFDC in increased in 1 unit, any BFC Manager will force the abortion of a frame only when the error detection mechanisms of CAN are not able to do it. Actually, this implies that the performance will note be negatively affected.

8 Reintegration policy

In Section 4.1 it was explained that each Enabling/Disabling Unit has a Threshold Control Module that isolates its port contribution an resets all the event counters and their managers when detects that any of the event counters exceeds its specific threshold.

However, in order to increase the tolerance to transient errors, the hub implements an automatic reintegration policy of disabled ports. Basically, the reintegration mechanisms consists on re-enabling the contribution of any port and to allow the operation of all its corresponding managers again, after a given period of *inactivity* is observed at the port. The state machine that describes this reintegration policy is depicted in Figure 6.

A port is set to the *idle* state whenever the hub is initialized or a stuck-at-recessive failure is detected. In such state, the contribution of the port is enabled. Note that a port diagnosed as being stuck-at-recessive enters into the *idle* state and its contribution is not isolated. This is because it does not generate errors that propagate to other ports

As soon as the hub receives a *meaningful* contribution from a port, e.g. an ACK signaling, an error flag transmission or a dominant bit contribution during the arbitration, the corresponding Threshold Control Module sets that port to the *active* state. The single difference between the *idle* and *active* states is that the second one indicates that the node is regularly participating in the communication process.

Whenever the stuck-at-dominant or the bit-flipping thresholds are exceeded, the Threshold Control Module sets the port to the *disabled* state. This actually implies that

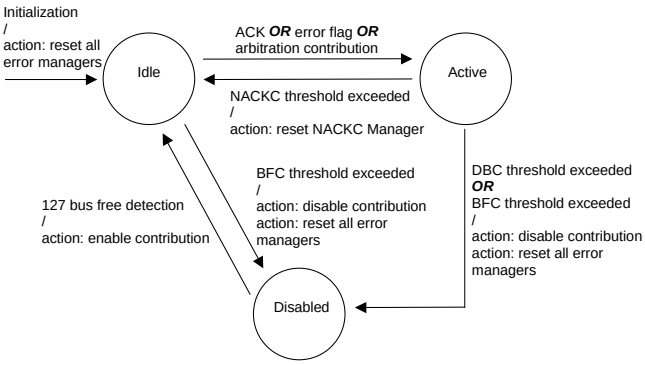

Figure 6: Reincorporation policy schema

the contribution of the port is disabled, as well as the different event counters and their respective managers are reset.

Once a port is in that state, the corresponding Threshold Control Module waits to observe a constant recessive contribution during 128 CAN bus free occurrences, i.e. 128 occurrences of 11 consecutive recessive bits [2]. This coincides with the number of consecutive recessive bits that a CAN node in the bus-off state must observe before being able to re-enter in the active error state. After detecting this period of *inactivity*, the port will be set again to the *idle* state, the contribution of the port will be re-enabled and the even counters and their manager will be operational.

The reintegration policy allows an autonomous performance of the hub because it is able to return to normal operation by itself.

9 Considerations on the cable length

The length of the cabling is an important factor in a distributed embedded system. In CAN, due to the synchronization at the bit level among all nodes, there is an inverse relationship between the bit rate and the maximum bus length. In CANcentrate, these relationship is preserved as the bit level synchronization of CAN is maintained. However, since the signals travel to the hub and then in parallel in all links back to the nodes, the maximum length applies only to every pair of links. This feature may represent a substantial increase in the capacity to interconnect nodes when compared with the bus topology. Consider a system with N nodes separated in space. The total length of the bus that interconnects such nodes is Lb (see Figure 7b). On the other hand, consider all nodes interconnected by means of a hub with link i having length L_i (see Figure 7a). Despite depending on the nodes placement, for the general case, $Lb >> L_i + L_j, \forall_{i,j}$ (see Figure 7). This is a major benefit of the star topology. On the other hand, also for the general case, $Lb < \sum_i (L_i)$ meaning that the total length of the cabling system is longer in the star topology. Nevertheless, the superior connectivity of the star may allow using higher bit rates than with a bus due to the stronger limitation on the bus

length.

In what concerns the length of each star link, the bit level synchronization imposes a limitation on the sum of the lengths of every pair, as stated above. Let this limitation be $Lmax_s$, the star diameter. In order to have the lengths of all links independent of each other, the previous constraint implies that $\forall_i L_i < Lmax_s/2$.

To derive $Lmax_s$, the maximum diameter of the star, we need to analyze the propagation of the electrical signals from end-to-end. With respect to a bus topology, the star presents an extra delay caused by the hub (additional transceivers and internal gates). This delay is dominated by the former factor since the gate delays are negligible (order of 1ns or less using modern technologies) when compared with the transceiver delay (around 150ns for fast transceivers, including bus to reception pin and transmission pin to bus [14]). For a given bit rate B, the bit time $1/B$ has now to account for both propagation effects as in a bus plus hub delay. For the former aspect, consider all the parts that contribute to establish the bit time in CAN using the normal bus topology. Let this be t_{pb} (notice that $t_{pb} = 1/B$ by definition). In a star, all these parts related to propagation effects also have to be considered, taking t_{ps}. However, the bit time now also includes the hub delay t_h, thus $t_{ps} = 1/B - t_h$. Note that since a signal must go through the hub two times (from the transmitting node to the receiving node and viceversa), t_h includes twice the time a signal is delayed when crossing the hub in one way.

Therefore, from the point of view of signal transmission, we can define a star equivalent bus, with propagation effects taking t_{ps} and operating at a bit rate B' so that

$$B' = \frac{1}{t_{ps}} = \frac{1}{1/B - t_h} = \frac{B}{1 - B * t_h} > B$$

The previous equation shows that a star is, from an electrical signal transmission point of view, equivalent to a bus operating at a higher bit rate. Moreover, the higher the bit rate, the larger the difference. Therefore, the maximum diameter of the star $Lmax_s$, operating at bit rate B, is the maximum length of standard CAN operating at bit rate B'. For example, given the 150ns figure of hub delay referred above, a star operating at $B = 1Mbit/s$ has a maximum diameter equal to the length of a bus operating at 1.18Mbit/s. On the other hand, if $B = 125Kbit/s$ then the maximum diameter of the star equals the length of a bus operating at 127.4Kbit/s which implies a negligible reduction in length. To calculate the effective bus length for these transmission rates refer to [15]

10 CANcentrate prototype implementation

This section is aimed at describing the basics of the first prototype of CANcentrate. The experimental platform that has been set up in order to test this prototype is also discussed. Finally the main results of these tests are presented.

10.1 Description of the prototype

The prototype is divided into several parts. Each of them corresponds to a given part or parts of the CANcentrate architecture. The details of such architecture can be found in Section 3. When building our prototype, we differentiated the following parts: the Coupler and the Fault-Treatment modules (referred hereafter as the *internal part of the hub*), the Input/Output Module, the links, and the CAN nodes. Next, a general description of the characteristics of each part implementation is given.

The internal part of the hub has been implemented using the VHSIC Hardware Description Language (VHDL) and the *Xilinx Virtex XCV300-PQ240* FPGA (Field Programmable Gate Array), which is placed in the Xilinx prototype board *PQ240-100 Prototype Platform* (*HW-AFX-PQ240-100* version).

A dedicated board has been used for implementing the Input/Output Module, following the wire-wrap technique. This board mainly contains four pairs of Philips PCA82C250 high-speed CAN transceivers and four RJ45 jacks (one jack for each pair of transceivers), so that up to four CAN nodes can be connected to the hub at the same time. The pin CANL (LOW level CAN voltage input/output) and the pin CANH (HIGH level CAN voltage input/output) of the transceiver are then connected to the appropriate pins of the corresponding RJ45 jack. The interconnection between the Input/Output Module and the internal part of the hub is made by means of a flat cable, which connects the specific reception and transmission pins of the CAN transceivers with the corresponding pins of the Xilinx prototype board.

One UTP (Unshielded Twisted Pair) Category 5/5e/6 ethernet cable is used for implementing each link, which is constituted by an uplink and an independent downlink (as explained in Section 3.1). Both the uplink and the downlink use two-wire differential lines. The uplink uses the Transmit pair while the downlink uses the Receive pair of the Ethernet cable. On the one hand, the CAN_H and the CAN_L wires of the uplink are implemented with the Transmit+ and the Transmit- ethernet wires respectively. On the other hand, the CAN_H and the CAN_L wires of the downlink are implemented with the Receive+ and the Receive- ethernet wires respectively.

The CAN nodes have been implemented using off-the-self components. Each node is constituted by two different boards that are attached to each other: a *CANivete* board and a *starLink* board. The CANivete board is a previous development of the Universidade de Aveiro (UA) for standard CAN applications and implements a typical CAN node. In contrast, the starLink board was specifically designed for this project. It includes all the additional components needed for modifying the CAN interface of the CANivete in order to build the schema of double transceivers needed for connecting each CAN node to the uplink and the downlink of CANcentrate (see Figure 3 and Section 3.1 for a description of such schema).

On the one hand, the CANivete is based on a printed board where the components are welded. Its main components are a Philips 82C592 micro-controller which integrates a CAN controller; several sets of input/output pins that are connected to different parts of the board for digital or analog I/O; an external EPROM memory of 64k (for storing the program); two RS-232 drivers, one connected to the internal UART of the micro-controller and another one connected to the I/O pins of the board; and a Philips PCA82C250 high-speed CAN transceiver, which is connected to the CAN controller

located within the 82C592 micro-controller.

On the other hand, the starLink is a wire-wrap board which contains a Philips PCA82C250 high-speed CAN transceiver and a RJ45 jack. The transceiver located within the CANivete is used for connecting the CAN node to the downlink, whereas the transceiver located within the starLink board is used for the uplink. The transmit data input pin of the transceiver of the CANivete (TxD pin of the PCA82C250) has been left open, and the printed track of the printed board which connected such pin with the CAN controller is now connected to the transmit data input pin of the transceiver of the starLink. The pins CANL and CANH of both transceivers are then connected to the appropriate pins of the RJ45 jack.

10.2 Experimental platform

The prototype of CANcentrate was extensively tested to check its correct operation under error-free conditions and in the presence of faults, as well as to measure its performance. To perform these tests, an experimental platform was built. Specifically, the issues that were taken into account when devising this platform are the configuration of the application that is executed at the CAN nodes, the configuration of the network and the fault-injection mechanisms. Additionally, two requirements are imposed on this experimental platform: to achieve the maximum network utilization with a given bit rate, and to force an arbitration at the beginning of the transmission of each frame.

The first issue concerning the experimental platform is the configuration of the application that the CAN nodes execute. All CAN nodes run the same application, but with different sets of CAN identifiers. Thus, it is impossible that two nodes try to send a frame with the same identifier at the same time (this is a general requirement for any CAN application).

The application the nodes execute is constantly trying to send data frames with different identifiers and different data lengths, in order to test different frames. In addition, for fulfilling the requirement of achieving the maximum network utilization with a given bit rate, the application follows two basic rules: it must trigger a new transmission whenever it successfully transmits a frame and it must restart the CAN controller whenever, due to errors, it reaches the *bus-off* state (in such state, a CAN controller is not involved in bus activities, see Section 3.3).

With regard to the second issue of the experimental platform, namely the configuration of the network, it covers several aspects that are related to the nodes, to the links and to the bit rate. Concerning the nodes, it is worth noting that at least three CANivete nodes are needed for fulfilling the requirement of forcing arbitration to take place in the transmission of each frame. This is because our CAN nodes are not able to perform a new transmission just after finishing a previous one (they have a single transmission buffer and, thus, an extra delay is needed for configuring and ordering a new transmission). Also note that, as stated before in Section 10.1, the Input/Output Module has been built to allow the connection of four CAN nodes at the same time. However, one of the ports of the hub is reserved for fault-injection purposes as will be explained later in this section. Therefore, the network is configured with three non-faulty CAN nodes plus a port for fault injection.

Regarding the other aspects covered in the network configuration, the links and the bit rate, several Ethernet cables of different lengths, as well as different bit rates have been used in order to measure the performance of the network depending on the star diameter and the bit rate. Nevertheless, due to implementation limitations on the clock oscillators of the CAN nodes, the maximum bit rate that has been used for testing the performance is 690kbit/sec.

Finally, the last issue related to the experimental platform is the set of fault-injection mechanisms that are used to validate the fault-treatment capabilities of the hub. As explained in Section 4, the hub is able to detect permanently faulty ports which present stuck-at-recessive faults, as well as to diagnose and isolate permanently faulty ports which present stuck-at-dominant or bit-flipping faults. Stuck-at-recessive faults can be easily injected by disconnecting the link of an operational CAN node from the hub. However, a more complex fault-injection mechanism is needed for stuck-at-dominant and bit-flipping faults.

For injecting both stuck-at-dominant and bit-flipping faults, a special CAN node, called *faulty node*, has been implemented. Such node is implemented with a stand-alone starLink board that is connected to a signal generator device (see Section 10.1 for a detailed explanation of such board). The *faulty node* is connected as any other node (by means of an Ethernet cable) to the port of the hub that is reserved for fault-injection purposes. Note that since the *faulty node* only has one transceiver, which is connected to the uplink within the cable, the downlink is left open at the end of the faulty node.

The transmit data input pin of the transceiver of the *faulty node* is connected to the signal generator device. In this way, different bit stream patterns, consisting of a periodic signal that alternates from the recessive to the dominant value with a given frequency, can be transmitted to the hub. How to use the *faulty node* to inject stuck-at-dominant and bit-flipping faults is explained in the next section.

10.3 Functional tests

As explained in Section 10.2, the correct operation of the prototype under error-free conditions, as well as in the presence of faults was checked by means of several functional tests. The aspects that have been tested under error-free conditions are the correctness of the:

- Operation of the different state machines that constitute the hub.
- Calculation of the *resultant frame* upon all node contributions.
- Correct synchronization at bit level and at frame level.
- Assignation of the roles of the nodes after the arbitration phase.

In contrast, the aspects that have been tested in the presence of faults are the correct of the:

- Increase and decrease of the different error counters during different fault scenarios.

- Detection of ports suffering stuck-at-recessive faults, as well as the isolation of ports suffering stuck-at-dominant or bit-flipping faults.
- Reintegration of ports following the policy explained in Section 8.

All the issues indicated above were tested at two different levels: at the level of the VHDL design of the hub and at the level of the physical network. However, the tools that have been used at each of these two levels impose different limitations. Thus, the different aspects listed above have been tested in different depths at the two levels.

The first level of testing, the functional testing of the VHDL design of the hub, has been done by means of the simulation tool *ModelSim XE II 5.7g* (provided by Mentor Graphics Corporation). Several simulations were done in order to check all the issues specified above and, in all cases, the operation of the hub was correct. Special attention has been paid to check the correct operation of the different state machines that constitute the hub, as well as their correct mutual interaction.

In what concerns the second level of testing, physical limitations in the layout of the FPGA board discourage an exhaustive testing of all the state machines that constitute the hub. In contrast, many more fault scenarios can be injected at physical level than at simulation level.

For this physical level of testing different parts of the physical network have been observed by means of a logical analyzer and a digital oscilloscope. In particular, the ports of the hub were observed in order to know which is the contribution of each node as well as the value of the coupled signal. Since the Rx_CAN Module and the Enabling/Disabling units are key modules for synchronizing the hub at bit level and at frame level, as well as for diagnosing and isolating faulty ports respectively, they have also been observed.

For physically testing the correct operation of the network under error-free conditions (like during the phase of the tests of the VHDL design), the correct calculation of the coupled signal, the correct synchronization at bit and at frame level and the correct assignment of the roles during the arbitration phase have been checked. With regard to the physical testing of the fault-treatment mechanisms the hub implements, extensive tests that include stuck-at-recessive, stuck-at-dominant, bit-flipping faults and the reintegration policy have been performed.

Specifically, in order to physically testing the actions carried out by the hub in the presence of stuck-at-recessive faults, the link of a previously operating node has been mechanically disconnected. When the link is disconnected a transient bit-flipping behavior is observed in its corresponding port. However, these erroneous bits are not enough for leading the hub to isolate the port. In contrast the contribution of the port quickly stabilizes to the recessive value and then, the hub indicates that the port is at the *idle* state. Which actually means that the port is stuck-at-recessive (see Section 8).

For physically testing the operations the hub performs in the presence of stuck-at-dominant faults, the *faulty node* described in Section 10.2 was used to transmit a periodic signal that keeps the dominant value during many frames. It was observed that the hub correctly increases the DBC and isolates the corresponding port whenever the configured Dominant Bit Threshold (DBT) is exceeded.

In what concerns the fault-diagnosis and fault-isolation operations the hub performs in the presence of bit-flipping faults, two kinds of techniques for injecting them have

been used. On the one hand, a bit stream, which has random values, was injected by means of mechanically connecting/disconnecting a given link into its plug. On the other hand, the *faulty node* was used for injecting a bit stream that changes from the recessive to the dominant value with different frequencies that do not match with the bit rate the nodes use for communicating. Notice that in both cases, the beginning of the bit-flipping injection was randomly chosen. Many tests were performed with both techniques and in all situations the results were correct.

Finally, for the physical testing of the reintegration policy, the state (*idle*, *active* or *disabled*) of a given port was observed in different situations (see Section 8 for an explanation of the different states of the ports). After the system start-up the port was in the *idle* state. When the node sent an ACK bit or when it tried to send a frame, the port state changed to the *active* state. If the node was at the *active* state and its link was disconnected from the hub, the port returned to the *idle* state. After the port was isolated due to a stuck-at-dominant or a bit-flipping fault, a recessive value was forced in this port by disconnecting its link or by compelling its node to send recessive bits. In these cases, the hub re-enabled the contribution of the port, which agrees with the expected behavior related to the reintegration policy.

10.4 Performance measurements

In what concerns the performance tests, some measurements have been made. The values of the FPGA device utilization needed for implementing the hub prototype (with 4 ports) are: 758 out of 3072 slices, 278 out of 6144 Flip-flops, 1396 out of 6144 LUTs, 91 out of 170 IOBs, 4 out of 4 GLCKs.

The extra delay introduced by the internal part is 35 ns, whereas the average value of the extra delay introduced by the entire hub is 155 ns. Notice that the value of the extra delay introduced by all the hub is of the order of 1/6 of the bit time when operating at the higher bit rate allowed in CAN [15] (1Mbit/sec).

In addition, the internal part of the hub has been also built with 16 ports. The values of the FPGA device utilization in this case are: 2534 out of 3072 slices, 869 out of 6144 Flip-flops, 4662 out of 6144 LUTs, 91 out of 170 IOBs, 4 out of 4 GLCKs. It has been observed that the extra delay introduced by the internal part of the hub does not visibly depend on the number of ports it is provided with.

Finally, several Ethernet cables of different lengths, as well as different bit rates have been used in order to measure the performance of the network depending on the star diameter and the bit rate. As said before, due to implementation limitations, the maximum bit rate that has been used is 625 kbit/sec. At this bit rate, the maximum star diameter that was used without generating errors is 70 meters which implies a negligible reduction in length when compared with a CAN bus operating at the same bit rate (maximum length of approximately 85 meters) [15]. Moreover, remember that in a star the maximum length applies only to every pair of links, thus the star increases the capacity to interconnect nodes when compared with the bus topology (see Section 9 for a further explanation of this issue).

Notice that these last performance results imply that the hub of this prototype of CANcentrate introduces a delay equivalent to have an extra cable length of 15 meters, whereas this equivalence in the active star topologies available in [10] is around 30

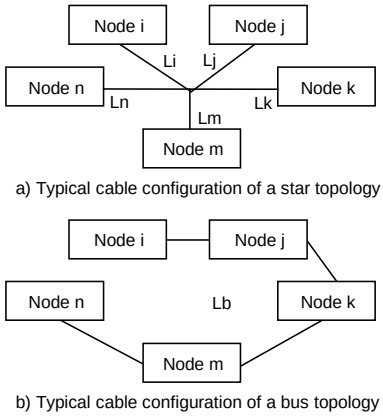

Figure 7: Comparison between the lengths of the cabling system in a star and in a bus

meters. Furthermore, the delay introduced by the hub could be reduced using not commercial transceivers in the hub. In this way the current delay of the hub, 150 ns, could be reduced until 20 or 30 ns, without losing compatibility with COTS components when building nodes and links.

11 Future work

The star topology offers many possibilities to improve dependability and real-time capabilities of a CAN network. In fact, more fault detection mechanisms can be integrated into a central privileged node, the hub, that may allow either further restricting the failure semantics of CAN-based communication systems, as well as notifying and isolating faulty regions. Moreover, the star topology in CAN imposes a cabling length constraint on pairs of nodes, only, and thus more nodes and larger areas can be served for a given transmission rate, or higher rates can be used for a given length constraint. Given that the real-time properties of the CAN medium access control are strictly maintained, the proposed star topology for CAN can thus improve the real-time performance of the network by means of a higher throughput.

In the short term we will consider some improvements over the simplex star solution namely, the use of a single cable in each link to reduce the costs of the wiring as well as the replication of the hub to eliminate the only single point of failure that remains in the system.

A particular approach that will be considered in future work is the use of switched strategies, which can be used to reduce the overhead introduced by damaged frames and error signaling, as well as to segment the network and further increase the global throughput.

12 Conclusions

Despite being widespread in distributed embedded systems, the use of CAN in safety-critical applications has been a controversial topic. This is due to a few factors such as the bus topology. In fact simplex bus topologies suffer from several impediments to enforce error containment while replicated buses may exhibit common mode and spatial proximity faults. On the other hand, star topologies may represent a positive step due to the key role that the hub can perform to diagnose and passivate faults. In fact, it allows reducing the number of components whose failure can cause a severe failure of the communication system, to a unique single point of failure, i.e. the hub.

In this document we discuss the characteristics of bus and star topologies in what concerns the ability to confine errors. We propose the design and the implementation of a new active star topology, called CANcentrate, that is compatible with off-the-shelf CAN controllers and that can be used with any CAN-based protocol (e.g. TTCAN [6], FTT-CAN [16], Timely CAN [17], MajorCAN [13], etc).

We describe the architecture of the central device of CANcentrate, a hub, which can be built using off-the-shelf FPGA technology. We discuss the fault-diagnosis and passivation mechanisms of the hub and we explain their advantages over the fault-diagnoses mechanism of CAN.

Moreover, we address the specific issue of link length, which, in CAN, is a particularly important topic due to the bit level synchronization and the resulting coupling between bit rate and link length. We have shown that for a given bit rate a star may have a diameter generally similar to the length of a bus except for higher bit rates, in which case it is slightly lower. On the other hand, the star may cover a substantially larger area than the bus or, for the same area, to use a higher bit rate.

Finally, we explain the implementation of a first prototype of CANcentrate. We checked the correctness of the fault-treatment mechanisms of the hub prototype and we evaluated the performance of CANcentrate. In particular, it has been observed that the extra delay introduced by the hub does not depend on the number of ports. Moreover, this extra delay implies a negligible reduction of length when compared with a CAN bus operating at the same bit rate.

In general, the simplex star topology proposed in this document is a further step towards improving both dependability and real-time performance of CAN networks.

References

[1] H. Kopetz, “Fault Containment and Error Detection in TTP/C and FlexRay,” Vienna University Of Technology, TU Wien, Research Report 23, August 2002.

[2] ISO, “ISO11898. Road vehicles - Interchange of digital information - Controller area network (CAN) for high-speed communication,” 1993.

[3] I. Broster and A. Burns, “An Analyzable Bus-Guardian for Event-Triggered Communication,” in *Proceedings of the 24th Real-time Systems Symposium (RTSS)*, University of York, UK. Cancun, Mexico: IEEE, Dec 2003, pp. 410–419.

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

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

[6] H. Kopetz, "Time-Triggered Protocols for Safety-Critical Applications," Presentation, Vienna University Of Technology, TU Wien, Karlsplatz 13, 1040 Vienna, Austria, March 2003.

[7] L.-B. Fredriksson, "CAN for critical embedded automotive networks," *IEEE Micro, Special Issue on Critical Embedded Automotive Networks*, vol. 22, no. 4, pp. 28–35, July-August 2002.

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

[9] CiA, "CAN physical layer," CAN in Automation (CiA), Am Weichselgarten 26, Tech. Rep. [Online]. Available: headquarters@can-cia.de

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

[11] G. Cena, L. Durante, and A. Valenzano, *A new CAN-like field network based on a star topology*, Polytechnic Institute Torino Std. 23, July 2001.

[12] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues, "Fault-tolerant broadcasts in CAN," *FTCS-28, The 28th International Symposium on Fault-Tolerant Computing, Munich, Germany*, 1998.

[13] J. Proenza and J. Miro-Julia, "MajorCAN: A modification to the Controller Area Network to achieve Atomic Broadcast," *IEEE Int. Workshop on Group Communication and Computations, Taipei, Taiwan*, 2000.

[14] Infineon(technologies), "CAN-Transceiver TLE 6250," Infineon technologies, 2002.

[15] CiA, "CAN data link layer," CAN in Automation (CiA), Am Weichselgarten 26, Tech. Rep. [Online]. Available: headquarters@can-cia.de

[16] L. Almeida, P. Pedreiras, and J. A. Fonseca, "The FTT-CAN Protocol: Why and How," in *IEEE Transactions on Industrial Electronics - special issue on Factory Communication Systems*, vol. 49, no. 6, December 2002.

[17] I. Broster and A. Burns, "Timely use of the CAN Protocol in Critical Hard Real-time Systems with Faults." in *Proceedings of the 13th Euromicro Conferencs on Real-time Systems (ECRTS)*. IEEE, 2001, pp. 95–102.